The Hough Transform

The Normal form

Series: The Hough Transform:

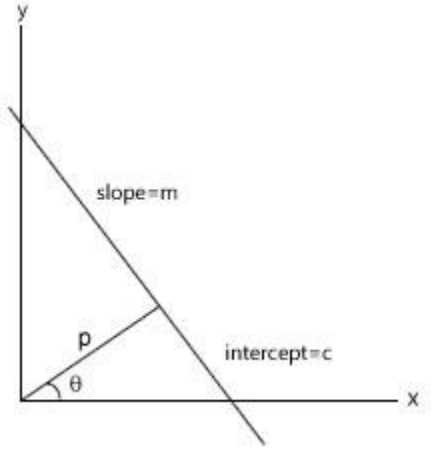
1. The Basics
2. **The Normal form**

The flaw

The Hough transform described in the previous article has an obvious flaw. The value of m (slope) tends to infinity for vertical lines. So you need infinite memory to be able to store the mc space. Not good.

The solution

The problem is resolved by using a different parametrization. instead of the slope-intercept form of lines, we use the normal form.



The normal form

In this representation, a line is formed using two parameters - an angle θ and a distance *p*. p is the length of the normal from the origin (0, 0) onto the line. and θ is the angle this normal makes with the x axis. This solves the flaw perfectly.

The angle θ can be only from -90° to +90° and the length p can range from 0 to length of diagonal of the image. These values are finite, and thus you can "store" them on a computer without much trouble.

In this representation, the equation of the line is:

$$p = x_1 \cos\theta + y_1 \sin\theta$$

where (x_1, y_1) is a point through which the line passes.

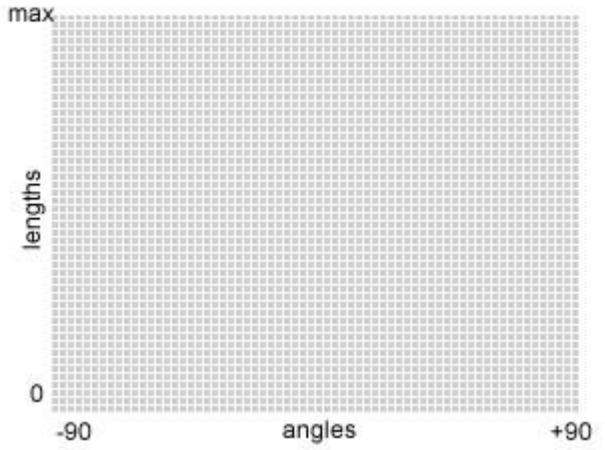
With this new equation, we have a few changes in the end-point to line "transition" from the xy space to the pθ space. A line in the xy space is still equivalent to a point in the pθ space. But a point in the xy space is now equivalent to a sinusoidal curve in the pθ space.

The **image at the top** of this article is an actual pθ space. The sinusoidal curves have been generated while trying to figure out lines an image.

Implementation

With those ideas through, we're ready to implement the Hough transform. The idea is to let each pixel "vote". So an array of accumulator cells is created.

In our case, the accumulator cells form a 2D array. The horizontal axis is for the different θ values and the vertical axis for p values.



The accumulator

Next, you loop through every pixel of the edge detected image (Hough works only on edge detected images... not on normal images).

If a pixel is zero, you ignore it. It's not an edge, so it can't be a line. So move on to the next pixel.

If a pixel is nonzero, you generate its sinusoidal curve (in the pθ space). That is, you take θ = -90 and calculate the corresponding p value. Then you "vote" in the accumulator cell (θ, p). That is, you increase the value of this cell by 1. Then you take the next θ value and calculate the next p value. And so on till θ = +90. And for every such calculation, making sure they "vote".

So for every nonzero pixel, you'll get a sinusoidal curve in the pθ space (the accumulator cells). And you'll end up with an image similar to the one at the top.

Detecting lines

The image at the top has several "bright spots". A lot of points "voted" for this spot. And these are the parameters that describe the lines in the original image. Simple as that.

Accuracy

The accuracy of the Hough transform depends on the number of accumulator cells you have. Say you have only -90^{0}, -45^{0}, 0^{0}, 45^{0} and 90^{0} as the cells for θ values. The "voting" process would be terribly inaccurate. Similarly for the p axis. The more cells you have along a particular axis, the more accurate the transform would be.

Also, the technique depends on several "votes" being cast into a small region. Only then would you get those bright spots. Otherwise, differentiating between a line and background noise is one really tough job.

Done

Now you know how the standard Hough transform works! You can try implementing it... should be simple enough.

Hough Line Transform

Goal

In this chapter,

- We will understand the concept of Hough Tranform.
- We will see how to use it detect lines in an image.
- We will see following functions: **cv2.HoughLines()**, **cv2.HoughLinesP()**

Theory

Hough Transform is a popular technique to detect any shape, if you can represent that shape in mathematical form. It can detect the shape even if it is broken or distorted a little bit. We will see how it works for a line.

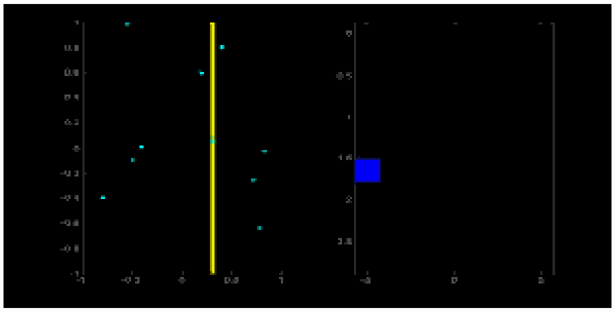
A line can be represented as $y = mx + c$ or in parametric form, as $\rho = x \cos \theta + y \sin \theta$ where $\rho$is the perpendicular distance from origin to the line, and θ is the angle formed by this perpendicular line and horizontal axis measured in counter-clockwise (That direction varies on how you represent the coordinate system. This representation is used in OpenCV). Check below image:

So if line is passing below the origin, it will have a positive rho and angle less than 180. If it is going above the origin, instead of taking angle greater than 180, angle is taken less than 180, and rho is taken negative. Any vertical line will have 0 degree and horizontal lines will have 90 degree.

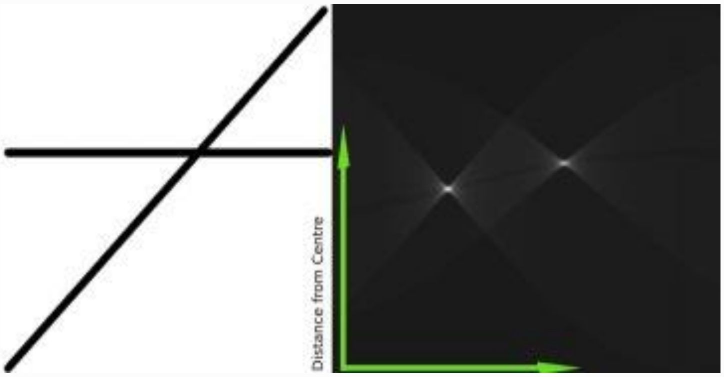
Now let's see how Hough Transform works for lines. Any line can be represented in these two terms, (ρ, θ). So first it creates a 2D array or accumulator (to hold values of two parameters) and it is set to 0 initially. Let rows denote the ρ and columns denote the θ. Size of array depends on the accuracy you need. Suppose you want the accuracy of angles to be 1 degree, you need 180 columns. For ρ, the maximum distance possible is the diagonal length of the image. So taking one pixel accuracy, number of rows can be diagonal length of the image.

Consider a 100x100 image with a horizontal line at the middle. Take the first point of the line. You know its (x,y) values. Now in the line equation, put the values $\theta = 0, 1, 2, ..., 180$ and check the ρyou get. For every (ρ, θ) pair, you increment value by one in our accumulator in its corresponding (ρ, θ) cells. So now in accumulator, the cell (50,90) = 1 along with some other cells.

Now take the second point on the line. Do the same as above. Increment the the values in the cells corresponding to (ρ, θ) you got. This time, the cell (50,90) = 2. What you actually do is voting the (ρ, θ) values. You continue this process for every point on the line. At each point, the cell (50,90) will be incremented or voted up, while other cells may or may not be voted up. This way, at the end, the cell (50,90) will have maximum votes. So if you search the accumulator for maximum votes, you get the value (50,90) which says, there is a line in this image at distance 50 from origin and at angle 90 degrees. It is well shown in below animation (Image Courtesy: Amos Storkey)



This is how hough transform for lines works. It is simple, and may be you can implement it using Numpy on your own. Below is an image which shows the accumulator. Bright spots at some locations denotes they are the parameters of possible lines in the image. (Image courtesy: Wikipedia)

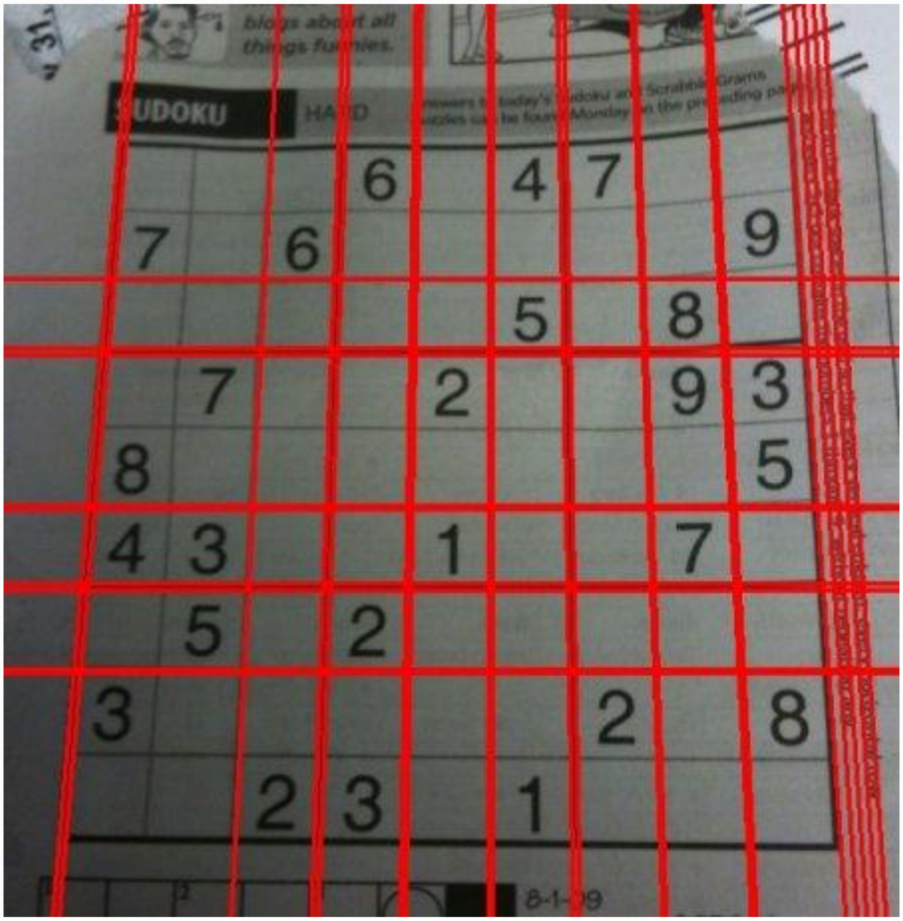


Hough Tranform in OpenCV

Everything explained above is encapsulated in the OpenCV function, **cv2.HoughLines()**. It simply returns an array of (ρ, θ) values. ρ is measured in pixels and θ is measured in radians. First parameter, Input image should be a binary image, so apply threshold or use canny edge detection before finding applying hough transform. Second and third parameters are ρ and θ accuracies respectively. Fourth argument is the *threshold*, which means minimum vote it should get for it to be considered as a line. Remember, number of votes depend upon number of points on the line. So it represents the minimum length of line that should be detected.

```
import cv2
import numpy as np

img = cv2.imread('dave.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray,50,150,apertureSize = 3)

lines = cv2.HoughLines(edges,1,np.pi/180,200)
for rho,theta in lines[0]:
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a*rho
    y0 = b*rho
    x1 = int(x0 + 1000*(-b))
    y1 = int(y0 + 1000*(a))
    x2 = int(x0 - 1000*(-b))
    y2 = int(y0 - 1000*(a))

    cv2.line(img,(x1,y1),(x2,y2),(0,0,255),2)

cv2.imwrite('houghlines3.jpg',img)
```

Check the results below:



DEEP LEARNING HAAR CASCADE EXPLAINED

Alright! This is where we start having some fun! The concept behind the Haar Cascade and how it is used in the real world is nothing short of amazing. So what is it?

HAAR CASCADE

Haar Cascade is a machine learning object detection algorithm used to identify objects in an image or video and based on the concept of features proposed by Paul Viola and Michael Jones in their paper "Rapid Object Detection using a Boosted Cascade of Simple Features" in 2001.

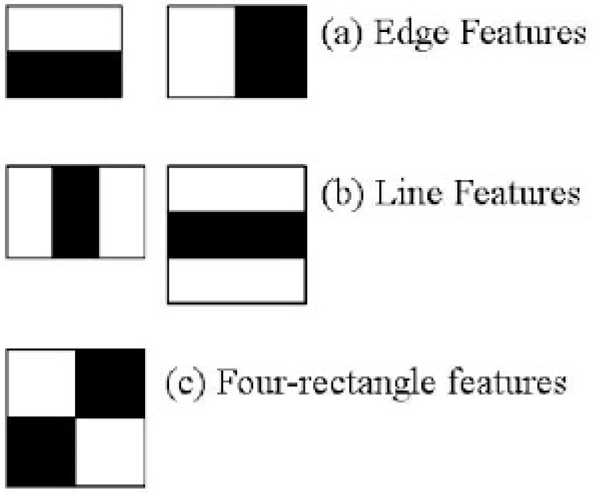
It is a machine learning based approach where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images. The algorithm has four stages:

1. Haar Feature Selection
2. Creating Integral Images
3. Adaboost Training
4. Cascading Classifiers

It is well known for being able to detect faces and body parts in an image, but can be trained to identify almost any object.

Lets take face detection as an example. Initially, the **algorithm needs a lot of positive images of faces and negative images** without faces to train the classifier.   Then we need to extract features from it.

First step is to collect the Haar Features. A Haar feature considers adjacent rectangular regions at a specific location in a detection window, sums up the pixel intensities in each region and calculates the difference between these sums.

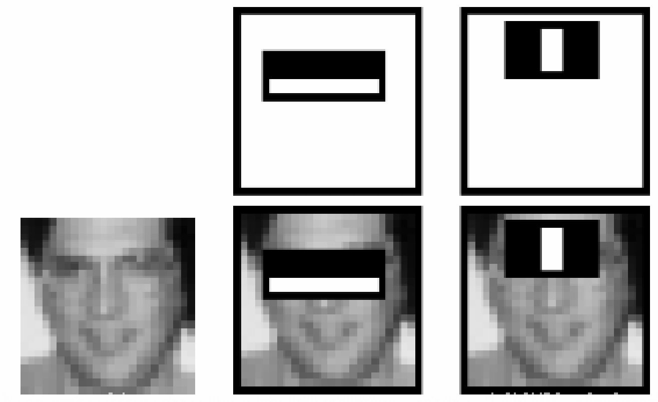

(a) Edge Features

(b) Line Features

(c) Four-rectangle features

Integral Images are used to make this super fast.

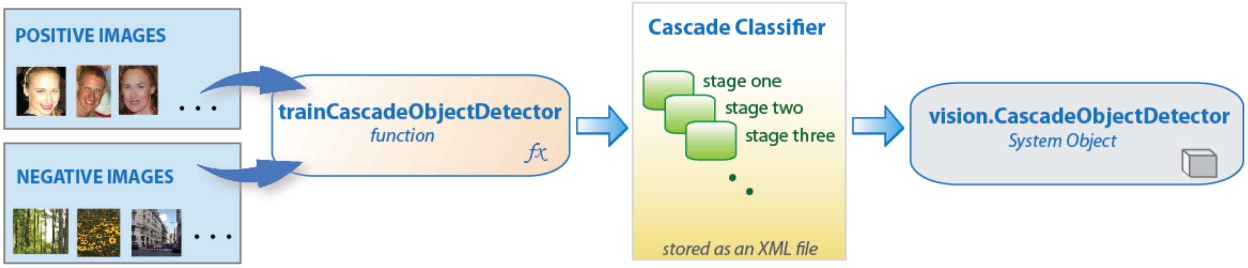
But among all these features we calculated, most of them are irrelevant. For example, consider the image below. Top row shows two good features. The first feature selected seems to focus on the property that the region of the eyes is often darker than the region of the nose and cheeks. The second feature selected relies on the property that the eyes are darker than the bridge of the nose. But the same windows applying on cheeks or any other place is irrelevant.



So how do we select the best features out of 160000+ features? This is accomplished using a concept called **Adaboost** which both selects the best features and trains the

classifiers that use them. This algorithm constructs a “strong” classifier as a linear combination of weighted simple “weak” classifiers. The process is as follows.
During the detection phase, a window of the target size is moved over the input image, and for each subsection of the image and Haar features are calculated.   You can see this in action in the video below. This difference is then compared to a learned threshold that separates non-objects from objects. Because each Haar feature is only a "weak classifier" (its detection quality is slightly better than random guessing) a large number of Haar features are necessary to describe an object with sufficient accuracy and are therefore organized into ***cascade classifiers*** to form a strong classifier.

Cascade Classifier



The cascade classifier consists of a collection of stages, where each stage is an ensemble of weak learners. The weak learners are simple classifiers called *decision stumps*. Each stage is trained using a technique called boosting. *Boosting* provides the ability to train a highly accurate classifier by taking a weighted average of the decisions made by the weak learners.

Each stage of the classifier labels the region defined by the current location of the sliding window as either positive or negative. *Positive* indicates that an object was found and *negative* indicates no objects were found. If the label is negative, the classification of this region is complete, and the detector slides the window to the next location. If the label is positive, the classifier passes the region to the next stage. The detector reports an object found at the current window location when the final stage classifies the region as positive.

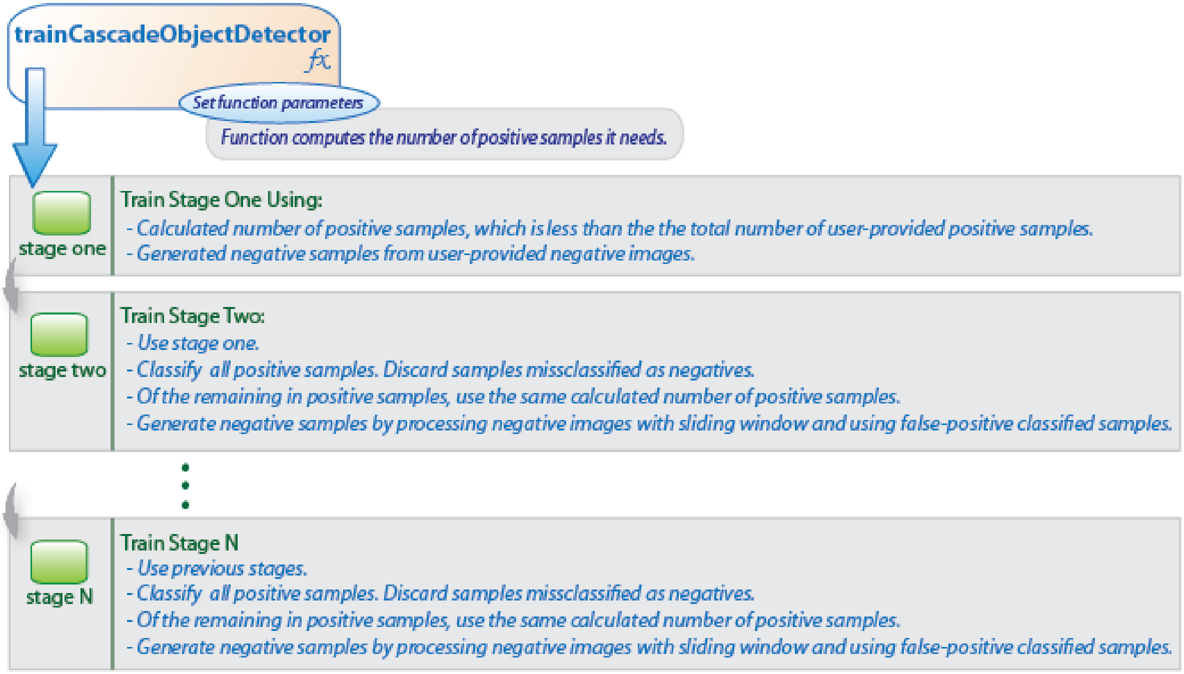
The stages are designed to reject negative samples as fast as possible. The assumption is that the vast majority of windows do not contain the object of interest. Conversely, true positives are rare and worth taking the time to verify.

- A *true positive* occurs when a positive sample is correctly classified.

- A *false positive* occurs when a negative sample is mistakenly classified as positive.
- A *false negative* occurs when a positive sample is mistakenly classified as negative.

To work well, each stage in the cascade must have a low false negative rate. If a stage incorrectly labels an object as negative, the classification stops, and you cannot correct the mistake. However, each stage can have a high false positive rate. Even if the detector incorrectly labels a nonobject as positive, you can correct the mistake in subsequent stages. Adding more stages reduces the overall false positive rate, but it also reduces the overall true positive rate.

Cascade classifier training requires a set of positive samples and a set of negative images. You must provide a set of positive images with regions of interest specified to be used as positive samples. You can use the Image Labeler to label objects of interest with bounding boxes. The Image Labeler outputs a table to use for positive samples. You also must provide a set of negative images from which the function generates negative samples automatically. To achieve acceptable detector accuracy, set the number of stages, feature type, and other function parameters.



Function computes the number of positive samples it needs.

Train Stage One Using:
- Calculated number of positive samples, which is less than the the total number of user-provided positive samples.
- Generated negative samples from user-provided negative images.

Train Stage Two:
- Use stage one.
- Classify all positive samples. Discard samples missclassified as negatives.
- Of the remaining in positive samples, use the same calculated number of positive samples.
- Generate negative samples by processing negative images with sliding window and using false-positive classified samples.

Train Stage N
- Use previous stages.
- Classify all positive samples. Discard samples missclassified as negatives.
- Of the remaining in positive samples, use the same calculated number of positive samples.
- Generate negative samples by processing negative images with sliding window and using false-positive classified samples.

The video below shows this in action.

Haar Cascade - Facial Detection IN ACTION

If a picture is worth a thousand words this would be a million words.   This is where it all comes together. The Ahh-hah moment.

This simple video helped crystalize for me how this algorithm works. Here are some observations:

- Notice how the algorithm moves the window systematically over the image, applying the Haar features as it is trying to detect the face. This is depicted by the green rectangles.
- Notice underneath the red boundary square, we see the classifier executing stages quickly discarding window frames that are clearly not a match (stages 1-25)
- To the right of the stage we see the how well it performed in identifying the face.
- Notice as it gets closer and closer to identifying the face, the number of stages increases into the 20s.   (around the 1 minute mark). This demonstrates the cascading effect where the early stages are discarding the input as it has identified them as irrelevant. As it gets closer to finding a face it pays closer attention.

Let me know if you have any questions or have any comments below.

I want to make sure I got this post right. It will be critical that you understand this before we go into the next section where we will implement a full **Custom Object Haar Cascade detector.**

NEXT STEPS

I don't know about you, but I find the best way to understand something is by doing it. Conceptually we now have an idea for how the machine learning Haar Cascade object detection works. Now lets build a real world custom Object Detector, train it, and see it in action. I have a really cool example for us! Click on the button below.

Face Detection using Haar Cascades

Goal

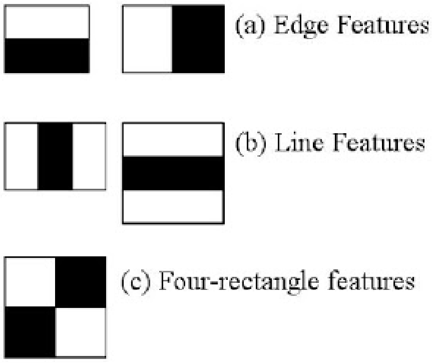
In this session,

- We will see the basics of face detection using Haar Feature-based Cascade Classifiers
- We will extend the same for eye detection etc.

Basics

Object Detection using Haar feature-based cascade classifiers is an effective object detection method proposed by Paul Viola and Michael Jones in their paper, "Rapid Object Detection using a Boosted Cascade of Simple Features" in 2001. It is a machine learning based approach where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images.

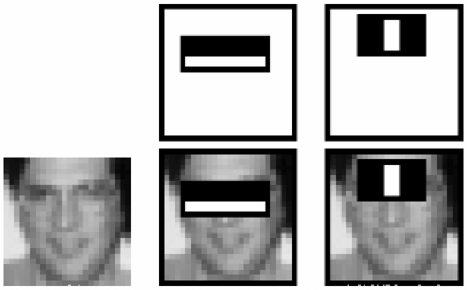
Here we will work with face detection. Initially, the algorithm needs a lot of positive images (images of faces) and negative images (images without faces) to train the classifier. Then we need to extract features from it. For this, Haar features shown in the below image are used. They are just like our convolutional kernel. Each feature is a single value obtained by subtracting sum of pixels under the white rectangle from sum of pixels under the black rectangle.



image

Now, all possible sizes and locations of each kernel are used to calculate lots of features. (Just imagine how much computation it needs? Even a 24x24 window results over 160000 features). For each feature calculation, we need to find the sum of the pixels under white and black rectangles. To solve this, they introduced the integral image. However large your image, it reduces the calculations for a given pixel to an operation involving just four pixels. Nice, isn't it? It makes things super-fast.

But among all these features we calculated, most of them are irrelevant. For example, consider the image below. The top row shows two good features. The first feature selected seems to focus on the property that the region of the eyes is often darker than the region of the nose and cheeks. The second feature selected relies on the property that the eyes are darker than the bridge of the nose. But the same windows applied to cheeks or any other place is irrelevant. So how do we select the best features out of 160000+ features? It is achieved by **Adaboost**.



image

For this, we apply each and every feature on all the training images. For each feature, it finds the best threshold which will classify the faces to positive and negative. Obviously, there will be errors or misclassifications. We select the features with minimum error rate, which means they are the features that most accurately classify the face and non-face images. (The process is not as simple as this. Each image is given an equal weight in the beginning. After each classification, weights of misclassified images are increased. Then the same process is done. New error rates are calculated. Also new weights. The process is continued until the required accuracy or error rate is achieved or the required number of features are found).

The final classifier is a weighted sum of these weak classifiers. It is called weak because it alone can't classify the image, but together with others forms a strong classifier. The paper says even 200 features provide detection with 95% accuracy. Their final setup had around 6000 features. (Imagine a reduction from 160000+ features to 6000 features. That is a big gain).

So now you take an image. Take each 24x24 window. Apply 6000 features to it. Check if it is face or not. Wow.. Isn't it a little inefficient and time consuming? Yes, it is. The authors have a good solution for that.

In an image, most of the image is non-face region. So it is a better idea to have a simple method to check if a window is not a face region. If it is not, discard it in a single shot, and don't process it again. Instead, focus on regions where there can be a face. This way, we spend more time checking possible face regions.

For this they introduced the concept of **Cascade of Classifiers**. Instead of applying all 6000 features on a window, the features are grouped into different stages of classifiers and applied one-by-one. (Normally the first few stages will contain very many fewer features). If a window fails the first stage, discard it. We don't consider the remaining features on it. If it passes, apply the second stage of features and continue the process. The window which passes all stages is a face region. How is that plan!

The authors' detector had 6000+ features with 38 stages with 1, 10, 25, 25 and 50 features in the first five stages. (The two features in the above image are actually obtained as the best two features from Adaboost). According to the authors, on average 10 features out of 6000+ are evaluated per sub-window.

So this is a simple intuitive explanation of how Viola-Jones face detection works. Read the paper for more details or check out the references in the Additional Resources section.

Haar-cascade Detection in OpenCV

OpenCV comes with a trainer as well as detector. If you want to train your own classifier for any object like car, planes etc. you can use OpenCV to create one. Its full details are given here: **Cascade Classifier Training**.

Here we will deal with detection. OpenCV already contains many pre-trained classifiers for face, eyes, smiles, etc. Those XML files are stored in the opencv/data/haarcascades/ folder. Let's create a face and eye detector with OpenCV.

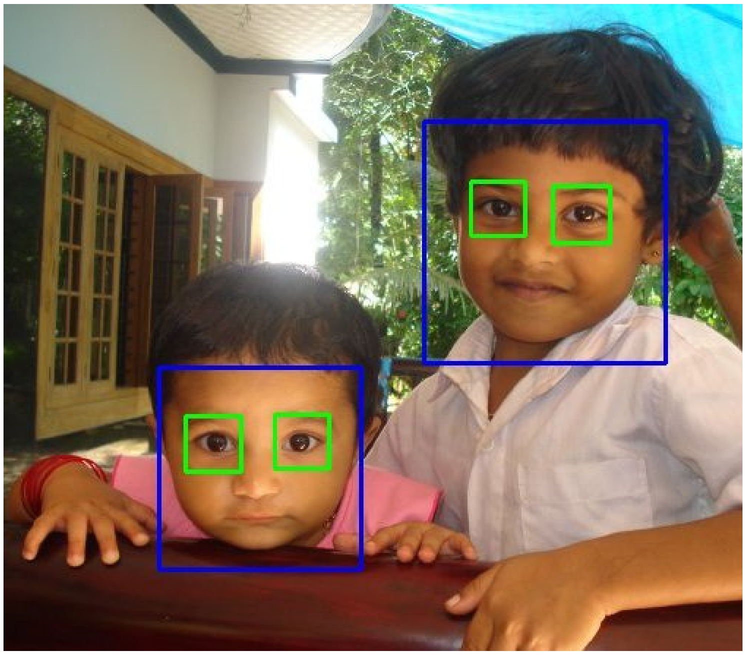
First we need to load the required XML classifiers. Then load our input image (or video) in grayscale mode.

```
import numpy as np
import cv2 as cv

face_cascade = cv.CascadeClassifier('haarcascade_frontalface_default.xml')
eye_cascade = cv.CascadeClassifier('haarcascade_eye.xml')

img = cv.imread('sachin.jpg')
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
```

Now we find the faces in the image. If faces are found, it returns the positions of detected faces as Rect(x,y,w,h). Once we get these locations, we can create a ROI for the face and apply eye detection on this ROI (since eyes are always on the face !!!).

```
faces = face_cascade.detectMultiScale(gray, 1.3, 5)
for (x,y,w,h) in faces:
    cv.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
    roi_gray = gray[y:y+h, x:x+w]
    roi_color = img[y:y+h, x:x+w]
    eyes = eye_cascade.detectMultiScale(roi_gray)
    for (ex,ey,ew,eh) in eyes:
        cv.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,255,0),2)

cv.imshow('img',img)
cv.waitKey(0)
cv.destroyAllWindows()
```

Result looks like below:



image

Additional Resources

1. Video Lecture on Face Detection and Tracking
2. An interesting interview regarding Face Detection by Adam Harvey

Exercises

Generated on Sat May 25 2019 03:40:17 for OpenCV by doxygen 1.8.12