PRE-REQUISITES FOR HAAR CASCADE CLASSIFIERS

- Boosting Algorithms (AdaBoost)
- Weak V/S Strong Learners

HAAR CASCADE CLASSIFIERS

A `Haar Classifier`, or a `Haar Cascade Classifier`, is a machine learning object detection program that identifies objects in an image and video.
It was proposed by `Paul Viola` and `Michael Jones` in 2001 under the name Rapid Object Detection Using a Boosted Cascade of Simple Features

Working of Haar Cascade Classifier

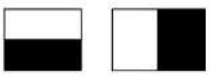
The algorithm can be understood in 4 steps:

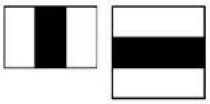
1. `Calculating Haar Features`
2. `Creating Integral Images`
3. `Using AdaBoost`
4. `Implementing Cascade Classifiers`

NOTE: **HAAR CASCADE REQUIRES A LOT OF POSITIVE AND NEGATIVE IMAGES IN ORDER TO TRAIN THE CLASSIFIER**

STEP 1: CALCULATING HAAR FEATURES

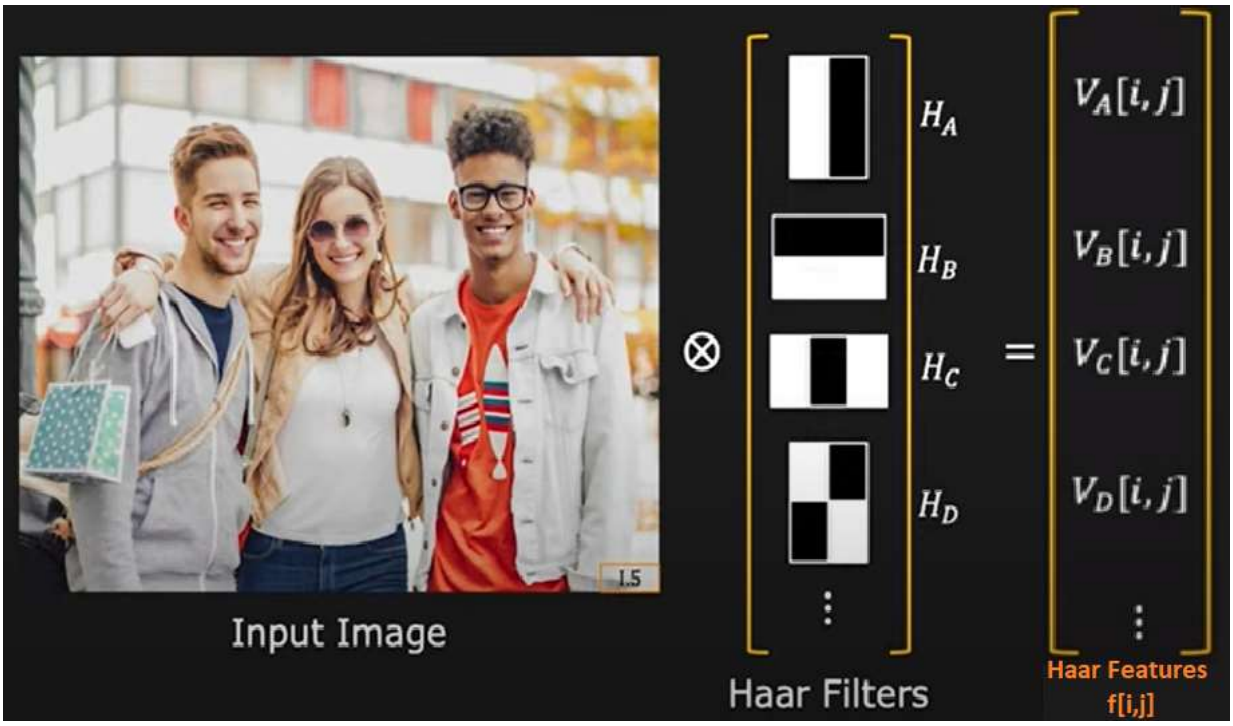
- A Haar feature is essentially calculations that are performed on adjacent rectangular regions at a specific location in a detection window.
- The calculation involves summing the pixel intensities in each region and calculating the differences between the sums.
- In layman terms, Haar features are calculated using Haar Filters which are based on Haar Wavelets which are essentially nothing but square functions. Some examples of Haar Features:


(a) Edge Features


(b) Line Features


(c) Four-rectangle features

- Haar Filters are basically 2 valued filters which makes it computationally very advantageous and is applied as a convolution on the image to get Haar Features.



- These features can be **difficult** to determine for a **large image**.
- This is where integral images come into play because the **number of operations** is **reduced** using the **INTEGRAL IMAGE**.

HAAR Features in Depth

- `Haar-like features` are `digital image features` used in `object recognition.`
- They owe their name to their intuitive similarity with Haar wavelets(Square Shaped Functions which allows the function to be represented in orthonormal basis i.e. in terms of unit vectors which are orthogonal to each other) and were used in the first real-time face detector.
- Historically, working with only **image intensities** (i.e., the RGB pixel values at each and every pixel of image) made the task of **feature calculation computationally expensive**. A publication by **Paul Viola and Michael Jones** discussed working with an `alternate feature set based on Haar wavelets` instead of the usual image intensities.
- They adapted the idea of using Haar wavelets and **developed** the so-called **Haar-like features**.
- A Haar-like 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. This difference is then used to categorize subsections of an image.
    - For example, with a human face, it is a common observation that among all faces the region of the eyes is darker than the region of the cheeks.
    - Therefore, a common Haar feature for face detection is a set of two adjacent rectangles that lie above the eye and the cheek region.
    - The position of these rectangles is defined relative to a detection window that acts like a bounding box to the target object (the face in this case)
- In the detection phase of this framework, a window of the target size is moved over the input image, and `for each subsection of the image` the **Haar-like feature is calculated**.
- This **difference** is then compared to a **learned threshold** that **separates non-objects from objects**.
- Because such a **Haar-like feature** is only a **weak learner or classifier** (its detection quality is slightly better than random guessing) a **large number** of **Haar-like features** are necessary to **describe an object with sufficient accuracy**.
- In this framework, the Haar-like features are therefore **organized** in something called a **classifier cascade** to form a **strong learner or classifier**.
- **The key advantage of a Haar-like feature over most other features is its calculation speed.**
- Due to the use of **Integral Images**, a Haar-like feature of any size can be calculated in constant time (approximately *60 microprocessor instructions for a 2-rectangle feature*).

Rectangular Haar Like Features

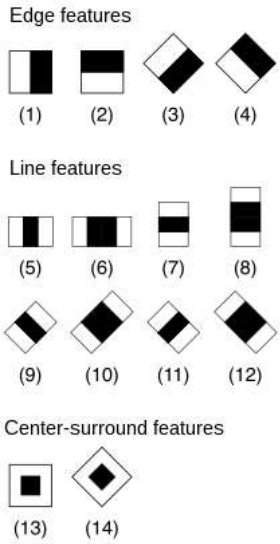
- A simple rectangular Haar-like feature can be defined as the difference of the sum of pixels of areas inside the rectangle, which can be at any position and scale within the original image.
- This modified feature set is called 2-rectangle feature. Viola and Jones also defined 3-rectangle features and 4-rectangle features.
- The values indicate certain characteristics of a particular area of the image.
- Each feature type can indicate the existence (or absence) of certain characteristics in the image, such as edges or changes in texture. For example, a 2-rectangle feature can indicate where the border lies between a dark region and a light region.

Tilted Haar Like Features

- Lienhart and Maydt introduced the concept of a tilted (45°) Haar-like feature.
- This was used to increase the dimensionality of the set of features in an attempt to improve the detection of objects in images.
- This was successful, as some of these features are able to describe the object in a better way.
- For example, a 2-rectangle tilted Haar-like feature can indicate the existence of an edge at 45°.

Rotated Haar Like Features

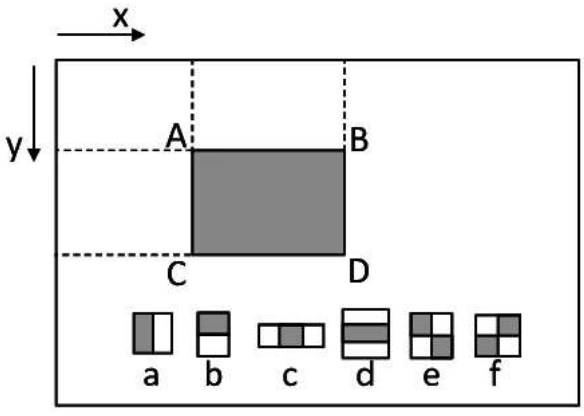
- Messom and Barczak extended the idea to a generic rotated Haar-like feature.
- Although the idea is sound mathematically, practical problems prevent the use of Haar-like features at any angle.



NOTE: In order to be fast, detection algorithms use low resolution images introducing rounding errors. For this reason rotated Haar-like features are not commonly used.

STEP 2: CREATING INTEGRAL IMAGES

- `Integral images` essentially `speed up` the `calculation` of these Haar features.
- Instead of computing at every pixel, it instead `creates sub-rectangles` and `creates array references` for `each` of those `sub-rectangles`.
- These are then used to compute the Haar features.



NOTE: Nearly all of the Haar features will be irrelevant when doing object detection, because the only features that are important are those of the object.

QUESTION: How do we determine the best features that represent an object from the hundreds of thousands of Haar features?
ANSWER: Using AdaBoost

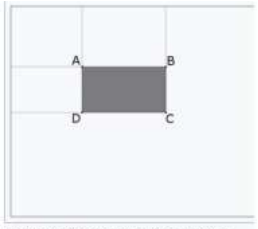
INTEGRAL IMAGES IN DETAIL

- One of the contributions of Viola and Jones was to use summed-area tables,which they called **integral images**.

- Integral images can be defined as two-dimensional lookup tables in the form of a matrix with the same size of the original image.
- Each element of the integral image contains the sum of all pixels located on the up-left region of the original image (in relation to the element's position).
- This allows to compute sum of rectangular areas in the image, at any position or scale, using only four lookups:

$$\text{sum} = I(C) + I(A) - I(B) - I(D).$$

- where points **A,B,C,D** belong to the *integral image* **I**, as shown in the figure.

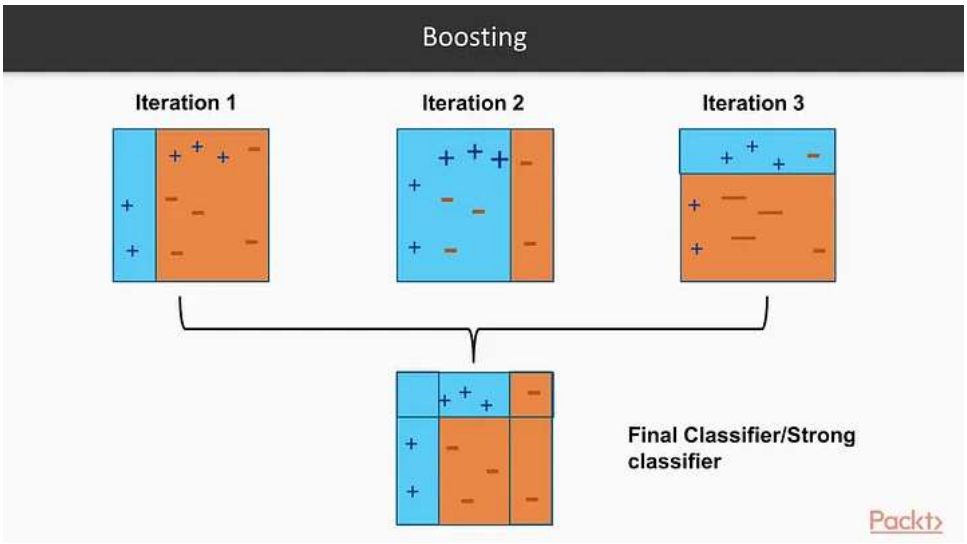


Finding the sum of the shaded rectangular area

- Each Haar-like feature may need more than four lookups, depending on how it was defined.
- Viola and Jones's 2-rectangle features need six lookups, 3-rectangle features need eight lookups, and 4-rectangle features need nine lookups.

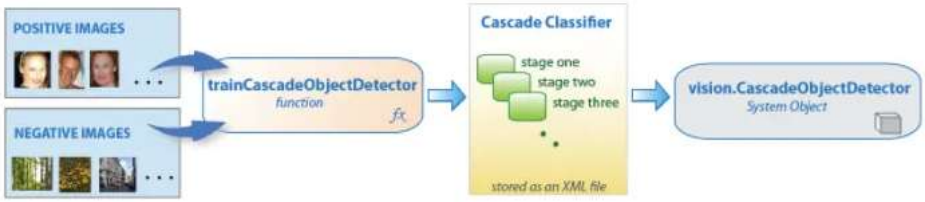
STEP 3: USING ADABOOST / ADABOOST TRAINING

- In layman terms AdaBoost essentially chooses the best features and trains the classifiers to use them.
- It uses a combination of `weak classifiers` to create a `strong classifiers` that the algorithm can use to detect objects.
- Weak learners are created by moving a window over the input image, and computing Haar features for each subsection of the image.
- This difference is compared to a learned threshold that separates non-objects from objects.
- Because these are **"weak classifiers"**, a large number of Haar features is needed for accuracy to form a strong classifier.



- The last step combines these weak learners into a strong learner using **cascading classifiers**.

STEP 4: IMPLEMENTING CASCADE CLASSIFIERS



- The cascade classifier is made up of series of stages, where each stage is a collection of weak learners.
- Weak Learners are trained using boosting, which allows for a highly accurate classifier from the mean prediction of all the weak learners.
- Based on this prediction, the classifier either decides to indicate an object was found (positive) or move on to the next region (negative).

- Stages are designed to reject negative samples as fast as possible, because a majority of the windows do not contain anything of interest.

  NOTE: It's important to maximize a LOW FALSE NEGATIVE RATE, because classifying an object as a non-object will severely impair your object detection algorithm.

APPLICATIONS OF HAARCASCADES

- **Facial Recognition**: Similar to how smartphones uses facial recognition, other electronic devices and security protocols can use Haar cascades to determine the validity of the user for secure login.
- **Robotics**: Robotic machines can "see" their surroundings to perform tasks using object recognition. For instance, this can be used to automate manufacturing tasks.
- **Autonomous Vehicles**: Autonomous vehicles require knowledge about their surroundings, and Haar cascades can help identify objects, such as pedestrians, traffic lights, and sidewalks, to produce more informed decisions and increase safety.
- **Image Search and Object Recognition**: Expanding off facial recognition, any variety of objects can be searched for by using a computer vision algorithm, such as Haar cascades.
- **Agriculture**: Haar classifiers can be used to determine whether harmful bugs are flying onto plants, reducing food shortages caused by pests.
- **Industrial Use**: Haar classifiers can be used to allow machines to pick up and recognize certain objects, automating many of the tasks that humans could previously only do.

BASIC IMPLEMENTATION OF HAARCASCADES ON IMAGES

```
import numpy as np
import cv2


  face_cascade = cv2.CascadeClassifier("files/haarcascade_frontalface_default.xml")
  eye_cascade = cv2.CascadeClassifier("files/haarcascade_eye.xml")


image = cv2.imread("data/cruella.jpg")
gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)


faces = face_cascade.detectMultiScale(gray,1.3,5)
for (x,y,w,h) in faces:
    img = cv2.rectangle(image,(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:
        cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,255,0),2)
cv2.imshow('img',image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

IMPLEMENTING HAARCASCADES FOR FACE DETECTION

```
  import cv2

  cap = cv2.VideoCapture(0)
  face_cascade = cv2.CascadeClassifier("files/haarcascade_frontalface_default.xml")
  eye_cascade = cv2.CascadeClassifier("files/haarcascade_eye.xml")
  while(True):
      ret,frame = cap.read()
      grey_frame = cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)

      faces = face_cascade.detectMultiScale(grey_frame,1.3,5)
      # Returns a list of tuples in which
      # each tuple will contain starting point (x,y) and width and height of the rectangle.

      # Scale Factor - 1.3 => specifies how much the image size is reduced at each image scale.
      # Image is being reduced by 30% in each pass

      # No of Neigbours - 5 => Parameter specifying how many neighbors each candidate rectangle should have.
      # Higher the no lesses will be the number of detections but their quality will be good

      if ret==False:
          continue
```

```
    for (x,y,w,h) in faces:
        cv2.rectangle(frame,(x,y),(x+w,y+h),(0,255,0),2)
        img = cv2.rectangle(frame,(x,y),(x+w,y+h),(0,255,0),2)
        cv2.rectangle(grey_frame,(x,y),(x+w,y+h),(0,255,0),2)
        roi_gray = grey_frame[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:
            cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,255,0),2)
            cv2.rectangle(roi_gray,(ex,ey),(ex+ew,ey+eh),(0,255,0),2)

    cv2.imshow("Video Frame",frame)
    cv2.imshow("Gray Frame",grey_frame)

    key_pressed = cv2.waitKey(1)
    if key_pressed==ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

Project: ATTENDANCE SYSTEM USING HAARCASCADES

Execute the following steps in order:

Sprint 1: Data Collection

1. Import all the necessary libraries required.
2. Load the haar filter for face and create the variables to store face data.
3. Use openCV video capture to connect with webcam and capture the data.
4. Convert the color frame to gray frame for easy processing.
5. Get the largest bounding box area from the detection made by the filter.
6. Extract the face region and resize it to the dimensions of `100x100`.
7. Save every 3rd frame so that you have enough time to change facial expression in order to have diverse data.
8. Convert the face data into numpy as array and save the file as `<yourName>.npy`

Sprint 2: Building ML Model to Identify the person

1. Import all the necessary libraries required.
2. Load the numpy files and filters.
3. Use openCV to get the video frames and convert each frame to grey frame.
4. Apply Haar Filter to detect the face
5. Use Machine Learning algorithm to classify the face
6. Store the name and the time in attendance.csv file

Resources

- Haar Cascade Classifier:
    - Haar Like Features - Wikipedia
    - Haar Cascades Explained - medium blog
    - Haar Cascades for Object Detection - Geeks for Geeks
    - Download Haar Cascade XML Files - Official Github
    - Haar Cascades official documentation - OpenCV
    - Computerphile - General discussion on Haar Cascades