# ICS2207 Report

By Gabriel Hili

Contents

[ICS2207 Report 1](#_Toc88402893)

[Viola-Jones Object Detection/Haar Cascade Technical Overview 3](#_Toc88402894)

[Developing Artifact 1 6](#_Toc88402895)

[Databases 6](#_Toc88402896)

[Cleaning 6](#_Toc88402897)

[Preparation 6](#_Toc88402898)

[Training 7](#_Toc88402899)

[Testing Script 9](#_Toc88402900)

[Developing Artifact 2 10](#_Toc88402901)

[Alternative Facial Detection Techniques 10](#_Toc88402902)

# Viola-Jones Object Detection/Haar Cascade Technical Overview

Viola-Jones Object Detection is a technique used to detect objects using a sliding window over an image. It was developed in 2001 by Paul Viola and Michael Jones. Haar features are used to recognize certain patterns commonly found in objects, (like how the nose is usually brighter than the eyes given a picture of a face).

To achieve object detection with Viola-Jones Object Detection Framework, means to train a Cascade Classifier for a face. A cascading classifier is a collection of weak classifiers which sequentially feed into each other. A weak classifier is composed of one Haar Feature and is able to detect small patterns of an object in an image, but not the whole object.

Hence, when these weak classifiers are chained together the whole image can be detected. A Haar Feature is a rectangular template for the brightness of pixels. In Voila-Jones when an area of pixels, has its average pixel brightness arranged in one of the following forms (where white mean bright, and black means dark), we say that a feature was recognized. A trained cascade classifier will detect objects by sliding a window across the image, with Haar Features in a certain position inside the window. If a good amount of these *features* are detected in the window, then the object was detected.

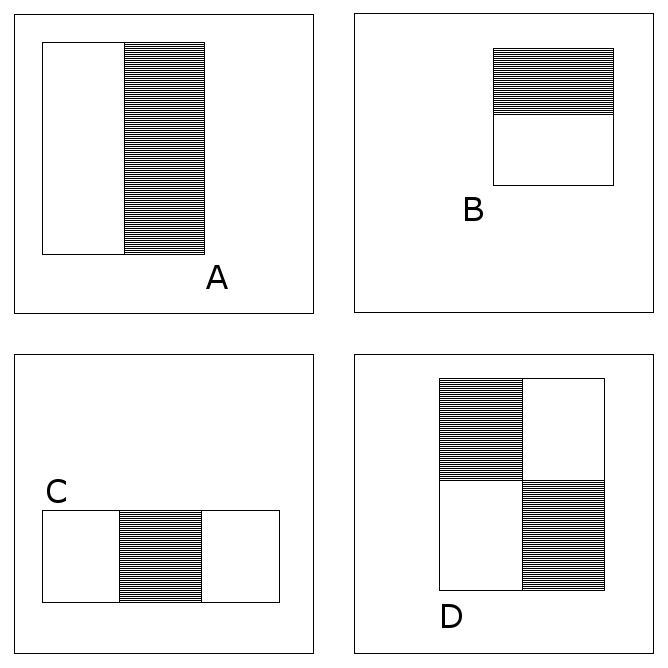


Figure 1: Different possible Haar Features.

The certainty of a given feature is the sum of the brightness of the pixels within white rectangles subtracted from the sum of the brightness of the pixels within black rectangles.

If a given weak classifier fails to find the Haar Feature in a given part of the image, the image is discarded as not being an object. Hence only when an image passes through all the weak classifiers, is the object detected.

Therefore to detect an object using a Cascading Classifier, which is the concatenation of several weak classifiers, each individual Haar features of that cascading classifier must be detected across the image.

In order for the cascading classifiers to find which features are the best in which position and size, the algorithm has to exhaustively search each possible configuration. These features can be of varying sizes, each rectangle could represent one pixel or half the image. Hence it becomes quite expensive to sum up every possible pixel brightness in a white rectangle subtracted from the sum of the brightness of the pixels within black rectangles for every possible different variation of a Haar Feature. To make the process faster an *integral image* is generated from the image being trained on.

An integral image is composed of values each corresponding to a pixel on the original image. A value in the integral image is the sum of all pixel values to the left and up of the corresponding pixel in the original image. This integral image is then used to calculate the sum over an area of pixels in the original image. The calculation will use the precomputed values in the integral image to determine the sum over an area of pixels, instead of linearly summing each pixel. This technique is much faster.

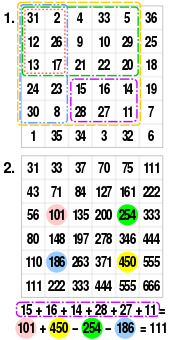


Figure 2: An example of an integral image being generated.

So given an input’s integral image, how does the Cascading Classifier learn that object’s features? Given a 24x24 image, there are 180,000+ possible feature configurations, as we have to consider the object can be of different sizes and position across the image. Only a small sample of these are actually useful. For example, the feature which says the eye region is darker than the forehead is useful while the one that says the forehead is darker than the inside of the mouth should not be considered, as generally this is not the case.

An exhaustive search for every combinations of features is infeasible. Viola-Jones uses a variant of the learning algorithm AdaBoost in order to detect the best features in a positive sample and add them to cascading classifier.

AdaBoost assigns a coefficient to every possible weak classifiers such that the sum training error of all the weak classifiers is as small as possible. To do this, this coefficient, or weight, is initially the same for all possible weak classifiers. For all weak classifiers, a strong classifier with only this single feature is trained, and the error is computed for every image passed, (if the feature fires on a negative sample then the error is increased, otherwise it’s decreased). After this process, the weights of the weak classifiers are updated. The lower the error, the higher this coefficient is, meaning that this feature will be more significant. This process will repeat until the sum of the weighted features is higher than half of the sum of the weights.

So after performing AdaBoost, we have obtained many different features, each with a value corresponding to how significant they are. Features with negligible values are discarded. Given an image that we wish to classify, the image will be passed through each of these features sequentially, starting with the highest significant one. If at any given moment, a feature does not fire on an image, we can immediately discard this image as not having our object. This greatly reduces the time for detection, as once a feature fails to fire on an image, since it is of higher significance than all the features coming after it, we know that it’s redundant to try the rest. This is why it is called a *cascading* classifier, as there are multiple small classifiers each feeding into each other.

Hence after training a cascading classifier using the Viola-Jones algorithm, given a trained cascading classifier and an input image, the integral image of the input image is computed. Then a sliding window of variant size which encapsulates any size the object can be in, is slid over the integral image. For every sliding window, the image encapsulated by this window is passed through the cascading classifier, and if the image passes through all of the features, the object is detected, and we can extract the bounding box of the object, since we know the position of our sliding window with respect to the image. If a feature fails to fire given an image, the image is discarded and the computation starts again for the next sliding window.

Viola-Jones Object Detection algorithm has seen a number of use-cases, specifically in the field of real-time facial detection. The algorithm is computationally cheap and is still the standard for facial detection despite the boom of deep learning and the fact that the algorithm is over 20 years old. Since the algorithm is fast, manufacturers do not have to increase hardware specifications for the devices running the algorithm which saves companies money. Moreover, it does not consume a lot of power due to it being computationally inexpensive, hence it is suitable for portable devices.

Digital Cameras use the Viola-Jones algorithm in-order to focus the lens of the camera on faces. Since the algorithm is fast, the camera’s display’s framerate remains real-time and power consumption is low.

# Developing Artifact 1

## Databases

For the positive samples, I used [CelebA](http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html), a dataset consisting of 200,000 celebrity faces. The images were of size 178x218 and the face’s eyes were fixed to an absolute position.

For the negative samples, I used [Caltech101](http://www.vision.caltech.edu/Image_Datasets/Caltech101/) and [Caltech256](http://www.vision.caltech.edu/Image_Datasets/Caltech256/), a dataset of objects from 101/256 categories respectively.

## Cleaning

I flattened both the directories of CelebA and Caltech using a shell script and placed them in a folder named pos and neg respectively. I manually looked through the negative dataset and removed any images with faces. Finally I created a shell script to rename all the images in the respective folder to a sequential index.

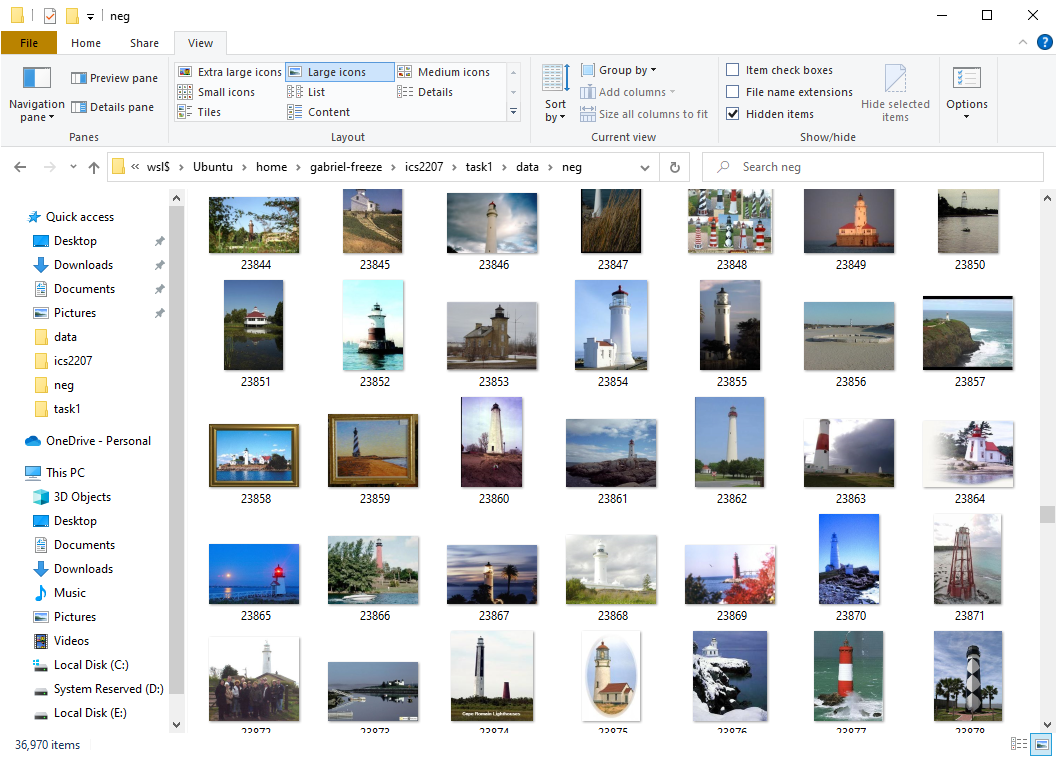


Figure 3: A snapshot of the *neg* folder.

## Preparation

I created a file called *info.dat* using a shell script which lists every positive image’s relative directory path and the bounding boxes for the face. Since the faces were all aligned by the position of the eyes, I overlayed some hundred faces on each other and took the average bounding box to be the bounding box for all the images, which I settled to be (x,y,w,h) = (30,57,127,137).

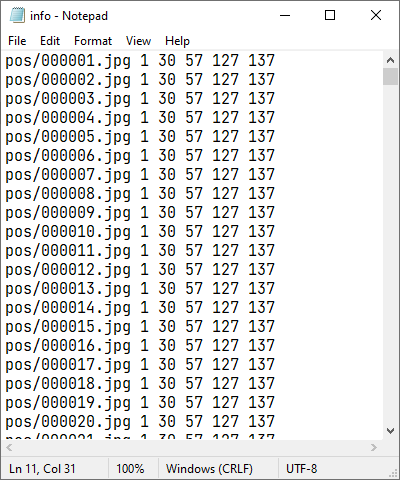


Figure 3: A snapshot of the *info.dat*



Figure 4: Overlay of several face images.

I created a file called *bg.txt* using a shell script which lists every negative images relative path directory. Both *bg.txt* and *info.dat* are needed to train the cascading classifier.

Using OpenCV, I had to transform the positive images into the binary format needed for OpenCV to process them. Using OpenCV’s *opencv\_createsamples* application I created a file called *pos.vec*. This file will be passed to the cascading classifier for training.

## Training

I used OpenCV’s *opencv\_traincascade* application to train the cascading classifier. I trained a number of different classifiers and chose the best one. I then passed them into a script, which performs the cascading classifier on each frame of a webcam and attempts to draw the bounding boxes across a face.

*Machine on which the cascading classifiers were trained:*

*Processor:* *AMD Ryzen 5 3600 6-Core Processor 3.59 GHz*

*RAM: 16GB*

*OS: Windows 10 Home 21H1*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Index** | **Positive Samples** | **Negative Samples** | **Acceptance Ratio Break Value** | **Max Number of Stages** | **Width and Height of Training Window** | **Remarks** |
| **1** | 2,000 | 1,000 | 10e-5 | 3 | (24,24) | My face and other faces were not being detected at all. I figured much would be so since the algorithm only took about 5 minutes to train and the False Alarm rate was well over 0.5. |
| **2** | 2,000 | 1,000 | 10e-5 | 12 | (24,24) | I increased the number of stages to 12 as 3 stages I thought 3 stages were little. Performance increased and the False Alarm rate was now hovering about 0.4 |
| **3** | 70,000 | 35,000 | 10e-5 | 12 | (24,24) | I was only using a small sample of my data before. I read that a 2:1 ratio for positive is to negative is the best approach, so I used all the negative images.  The model took 5 hours to train and the False Alarm Rate had started increasing again after stage 6 of training. Performance was worse. |
| **4** | 5,000 | 2,500 | 10e-5 | 20 | (24,24) | I decreased the dataset from fear of overfitting. I decided to increase the number of stages instead.  The algorithm did not take a long time to train and the False Alarm rate was around 0.3. Only 14 stages were used as on stage 14 the acceptance ratio had fallen below the threshold.  This classifier performed the best so far |
| **5** | 5,000 | 2,500 | NULL | 38 | (24,24) | I decided to increase the number of stages and remove the acceptance ratio break value. The algorithm took a longer time to train than before and the false alarm rate was more or less the same. |

In the end I chose Classifier 4 as it had performed the best with the least number of stages and training data.

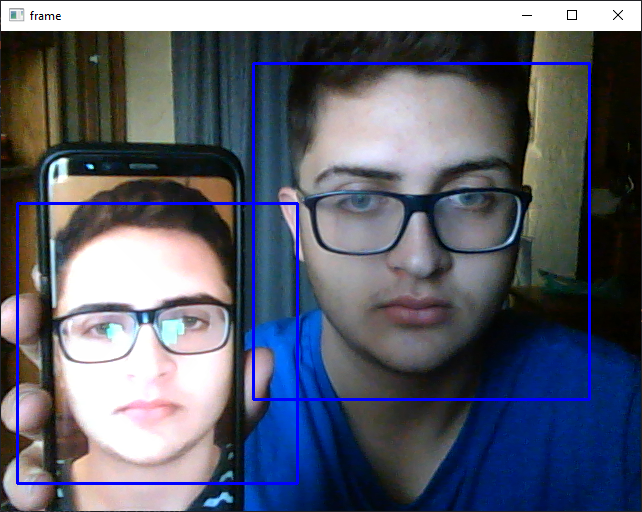


Figure 5: Multiple faces being detected using a cascade classifier.

## Testing Script

I first scale the current frame of the webcam by a factor of 4, grayscale it and equalize the brightness histogram. I detect faces across the processed face with parameters *scaleFactor=1.05* and *minNeighbours=3*. For each face detected, I draw the bounding boxes on the original frame. The testing script is *task1/face-recognition.py*

# Developing Artifact 2

I downloaded a [frontal face cascade](https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_default.xml) and an [eye cascade](https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_eye.xml). I then wrote a script that runs the face cascade on every frame of a webcam. The section of the frame with a face is passed to the eye cascade. Then all bounding boxes are drawn. The script is *task2/face-eye-detection.py*

# Alternative Facial Detection Techniques