VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

PROGRAMMING INTERGRATION PROJECT (CO3101)

FACE DETECTION

Class: CC01 – Group: 7

Lecturer: Ph.D Tran Tuan Anh
Students: Le Hoang Duy – 2152040
Vo Thien Nam – 2111817
Le Duc Hoang Nam – 2111795

HO CHI MINH CITY, JANUARY 2024

TABLE OF CONTENTS

1. Abstract

The integration of information technology has become pervasive in various aspects of daily life, streamlining tasks and enhancing efficiency. Computer systems play a pivotal role in this transformation, offering the ability to perform diverse functions, ultimately saving time and labor. An illustrative example is the application of facial recognition technology. Previously, security personnel in settings like supermarkets or airports had to manually scan surveillance camera feeds to identify suspects. Today, automated facial recognition systems have revolutionized this process, swiftly detecting human faces in images.

Human face detection is a critical component of facial recognition systems, simplifying the overall recognition problem by classifying input objects as either human faces or not. Once faces are detected, the system can then compare them with stored data to ascertain the identity of the individual, a common practice in recognizing celebrities or criminals.

To explore this further, this report delves into two primary face-detection methods:

1. **Classical Feature-Based Technique: Cascade Classifier**
   - Introduced by Viola and Jones, this method relies on the cascade classifier. Key concepts include Harr-like features, integral image techniques for rapid feature evaluation, AdaBoost for classifier construction by selecting crucial features, and the cascade of classifier technique, which accelerates detection by focusing on promising image regions.
2. **Deep Learning Method: Multi-task Cascade Convolutional Neural Network (MTCNN)**
   - A more recent approach that has achieved state-of-the-art results. MTCNN utilizes a Neural Network backbone, leveraging GPU capabilities for superior face detection performance compared to Viola and Jones' cascade face detector. The report explores Neural Networks, Convolutional Neural Networks, and the three stages proposed by MTCNN.

Moreover, we also discover briefly about 2 more modern algorithms, that is: **YOLO** and **HOG.**

The report covers the foundational principles behind these methods, including the Harr-like feature, integral image technique, AdaBoost, cascade of classifier technique for the cascade classifier, and the Neural Network and Convolutional Neural Network for MTCNN. Additionally, it provides insights into the three stages proposed by MTCNN. The ultimate aim is to enhance understanding and implementation within our project team.

2. Background

2.1 Cascade classifier

There are many methods to solve the problem of identifying human faces on 2D images based on different approaches. And perhaps two main approaches: feature-based methods that use hand-crafted filters to search for and detect faces, and image-based methods that learn holistically how to extract faces from the entire image.

The Haar-like – Adaboost method (abbreviated HA) by two authors Paul Viola and Michael J.Jones is a method of identifying human faces based on the feature-based approach. The feature-based approach means that the facial recognition system will learn facial patterns from a set of sample images. After this learning or training process is completed, the system will extract parameters to serve the identification process.

In general, the HA method is built based on the combination and assembly of 4 components, which are:

- **Haar-like features**: features are placed into image areas to calculate feature values. From these feature values, put into the Adaboost classifier, we will determine whether the image has a face or not.

- **Integral Image**: actually this is a tool that helps calculate Haar-like feature values faster.

- **Adaboost** (Adaptive Boost): classifier (filter) operates based on the principle of combining weak classifiers to create a strong classifier. Adaboost uses Haar-like feature values to classify images as faces or non-faces.

- **Cascade of Classifiers**: cascade classifier with each level being an Adaboost classifier, which increases classification speed.

Thus, the problem of identifying human faces in images is also the problem of classifying images into two classes: face or non-face.

Before dive into theory behind, there is small note that the HA method is performed on gray images. Each pixel will have a gray level value from 0 to 255 (8-bit color space). Thus, the HA method will not exploit facial color characteristics for identification but is

still very effective. Color images will be converted to gray images for identification. This conversion is quite simple, done with a conversion function, for example:

$$\mathbf{grayscale} = \mathbf{0.3} * \mathbf{R} + \mathbf{0.59} * \mathbf{G} + \mathbf{0.11} * \mathbf{B}$$

R,G,B is Red, Gree, Blue value of pixels in RGB 3 channel of the image.

After converting into a grayscale image, the image continues to be converted into an "integral image" (will be presented later) and in the first step of the recognition process, Haar-like features will work directly on the integral image.

2.1.1 Haar-like features

On an image, the face area is a collection of pixels that have different relationships compared to other image areas, these relationships create unique characteristics of the face. All human faces share the same characteristics after being converted to grayscale images, for example:

- The eye area will be darker than the cheek and cheek areas, meaning the gray level of this area is significantly higher than the other two areas.

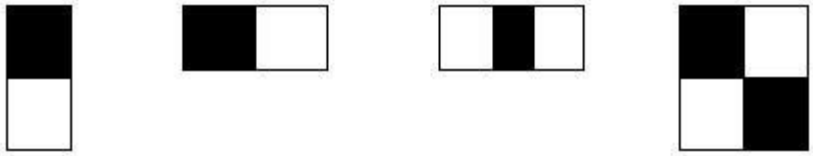
- The area in the middle of the bridge of the nose is also darker than the area on both sides of the nose…

And there are many other facial features and Haar-like features that rely on these features for identification. In general, Haar like features are not only used to identify faces but can be used to identify any object in the image (arms, legs, cars, objects, etc…). Because like faces, each object has unique features characterized by pixel regions, the job of Haar-like features is to calculate correlation values between those image regions.

This is only the initial step on how to apply features to classify child windows. Details on the classification of Adaboost and Cascade of Classifiers will be presented in the following section.

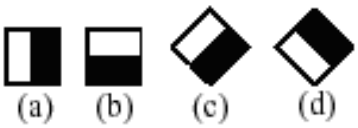
2.1.1.1 Some type of Haar-like features

Each Haar-like feature is a rectangular region divided into 2, 3 or 4 small rectangles conventionally distinguished by white and black.

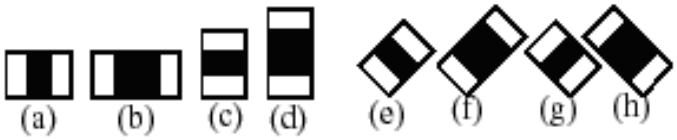


As shown in the figure above, there are 3 basic types of Haar-like features consisting of 2, 3 and 4 rectangles. Moreover, from the that basis we can extend to some more:

- Egde features:



- Line features:



The value of a Haar-like feature is the difference between the sum of the gray values of the pixels in the "black" region and the sum of the gray values of the pixels in the "white" region:

$$f(x) = Sum_{black\ rectangle}\ (pixel\ gray\ level) - Sum_{white\ rectangle}\ (pixel\ gray\ level)\ (1)$$

So when placed on an image area, the Haar-like feature will calculate and give the characteristic value f(x) of that image area.

2.1.1.2 Apply Haar-like features to detect faces in images

To detect faces, the system will create a sub-window of fixed size that scans the entire input image. Thus, there will be many sub-images corresponding to each sub-window, Haar-like features will be placed on these sub-windows to calculate the value of the feature. These values are then used by the classifier to confirm whether the frame is a face or not.

The image below is an example: the blue frame is a sub-window, a Haar-like feature with the size and placement as shown in the drawing.

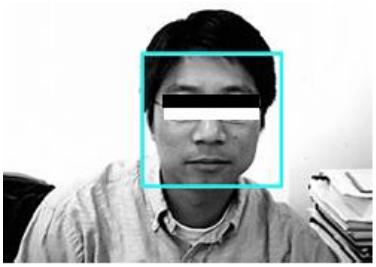


Figure 1

For each of the above features, a weak classifier $h_k(x)$ is defined as follows:

$$h_k(x) = \begin{cases} 1 \; if \; p_k f_k(x) < p_k \theta_k \\ 0 \; otherwise \end{cases} \quad (2)$$

With:

x: current 24x24 pixel sub-window of image

θ_k: threshold

f_k: Haar-like feature value

p_k: parity indicating the direction of the inequality sign

We understand the above formula simply as follows: when the value of the Haar-like feature k: f_k at sub-window x exceeds a threshold θ_k, the classifier $h_k(x)$ will conclude that sub-window x is a face. ($h_k(x)=1$), and if f_k does not exceed that threshold, it is not a face.

And in above example image, we see that the Haar-like k feature with size and position in the x sub-window on the image will have a very large f_k value (because the gray-pixel value in the eye area is much larger than the cheek area). This f_k value is much larger than most other f_k values when we move the sub-window x to other positions (scanning over the entire input image), and with the appropriate threshold θ_k, the classifier results $h_k(x)$ will give the sub-window in the above position a face but in other positions it will not.

Of course, there will be other locations on the input image that accidentally produce an f_k that exceeds the threshold and the weak classifier $h_k(x)$ concludes that it is a face, but we don't just use one Haar-like feature but use many features in different positions and sizes within the sub-window, reducing coincidence.

2.1.1.3 Haar-like feature quantity

Determining whether a sub-window is a face or not requires the use of many Haar-like features. For each size, feature type and position in the sub-window, we get a feature corresponding to a weak classifier x. So the complete set of features in a sub-window is very large. According to the calculations of the two authors, with a sub-window of basic size 24×24 pixels the full number of Haar-like features is more than 180,000. There are two issues raised here:

First, the value of each Haar-like feature is calculated by the total value of the black area pixels minus the total white area pixels, so that a large number of features will create a lot of calculation. This is not suitable for real-time response because the processing time is very long.

Second, among those hundreds of thousands of features, not all of them are useful for classification. If we do not find ways to eliminate useless features to focus on highly effective features, we will waste time processing in vain.

The following sections will seek to address these issues in turn.

2.1.2 Integral image

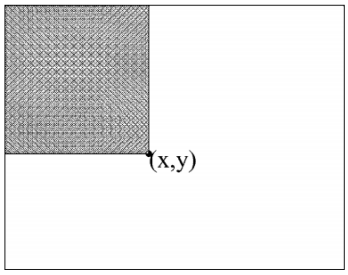
2.1.2.1 Definition

As stated above, the number of Haar-like features is very large and the amount of calculating the values of these features is very large. Therefore, integral images are introduced to quickly calculate features and reduce processing time.

The integral image is defined by the formula:

$$ii(x, y) = \sum_{x' \le x, y' \le y} i(x', y')$$

The value of the integral image at position (x,y) is the sum of the pixels in the rectangle defined by the upper left corner (0,0) and the lower right corner (x,y).



In fact, when converting an image into an integral image, we use the following recurrence formula:

$s(x,y) = s(x,y-1) + i(x,y)$ $\quad s(x,-1) = 0$

$ii(x,y) = ii(x-1,y)+s(x,y)$ $\quad ii(-1,y) = 0$

For example, converting a 2×2 image with gray values as below into an integral image: Computer Vision – The Integral Image | Computer Science: Source (wordpress.com)

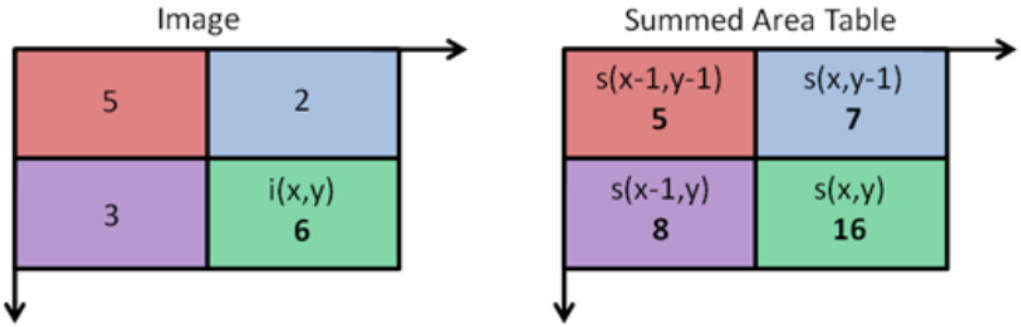
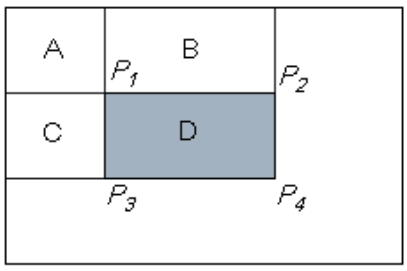


Figure 2

After converting the image into an integral image, calculating the value of Haar-like features will be very simple.

2.1.2.2 Apply for Haar-like feature calculation

To calculate the Haar-like feature value, we must calculate the total pixel value in a rectangular area on the image. For example, area D in the figure:



With: A, B, C, D is the total value of pixels in each area.

P1,P2,P3,P4 are the values of the integral image at the 4 vertices of D.

If it is a normal gray image, to calculate D we must sum all the pixel values in D. The larger the region D, the more additions there will be. But with an integral image, no matter what size the domain D is, D only needs to be calculated through 4 values at 4 vertices.

We have: $P1 = A, P2 = A + B, P3 = A + C, P4 = A + B + C + D$

Therefore, $P1+ P4–P2–P3= A + (A + B + C + D) – (A + B) – (A + C) = D$

So, $D = P1 + P4 – P2 – P3.$

When applied to calculating characteristic values we can see: The two-rectangle feature (edge feature) is calculated through 6 integrated pixel values. The three-rectangle feature (line feature) and center-surround feature are calculated through 8 integrated pixel values. The 4-rectangle feature (diagonal feature) is calculated through 9 integrated pixel values. Meanwhile, if calculated as original define, the values that need to be calculated are up to hundreds. This increases processing speed significantly.

2.1.3 Adaboost

2.1.3.1 Overview

Boosting technique: the basic principle of Boosting is to combine weak classifiers (or base classifiers) to create a strong classifier. These weak classifiers are even only slightly better than the random method. In this way, we say the classifier has been “boosted.”

Adaboost's improvement is that we will **assign each sample a weight**. The meaning of weight assignment is as follows:

In each iteration of the training process, when a weak classifier y_i has been built, we will update the weights for the samples. This update is carried out as follows: we will increase the weight of samples misclassified by the weak classifier y_i and decrease the weight of samples correctly classified by y_i. In this way, in the next iteration, we will build the weak classifier y_{i+1} in the following direction: **focusing on samples misclassified by the previous weak classifier.**

Finally, to get a strong classifier, we will linearly combine the found weak classifiers together. Each weak classifier will be given a weight corresponding to the goodness of that weak classifier.

2.1.3.2 Algorithm

Given example images $(x_1, t_1), \ldots, (x_n, t_n)$ with $t_i \in \{0, 1\}$

1. Initialize weights: $w_n^{(1)} = \frac{1}{N}$ with n = 1, 2, …, N.
2. For m = 1, …, M:

(a) Building a weak classifier h_m:

+ With each feature j, train a classifier h_j which is restricted to using a single feature. The error is evaluated:

$$E_j = \sum_{n=1}^{N} w_n^{(m)} I(h_m(x_n) \neq t_n) \qquad (1.0)$$

With $I(h_m(x_n) \neq t_n) = 1$ if $h_m(x_n) \neq t_n$ and = 0 otherwise.

+ Then choose the weak-classifier h_j with lowest error that is h_m.

(b) Update the weights:

+ Calculate:

$$\epsilon_m = \frac{\sum_{n=1}^{N} w_n^{(m)} I(h_m(x_n) \neq t_n)}{\sum_{n=1}^{N} w_n^{(m)}} \qquad (1.1)$$

and:

$$\alpha_m = ln \frac{1-\epsilon_m}{\epsilon_m} \qquad (1.2)$$

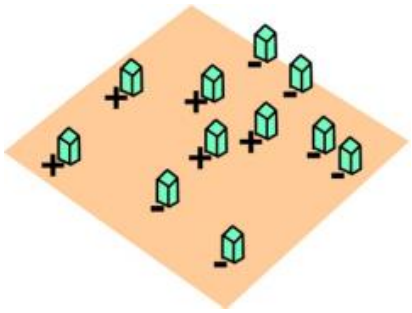
+ then update:

$$w_n^{(m+1)} = w_n^{(m)} \exp\{\alpha_m I(h_m(x_n) \neq t_n)\} \qquad (1.3)$$

3. The final strong classifier is:

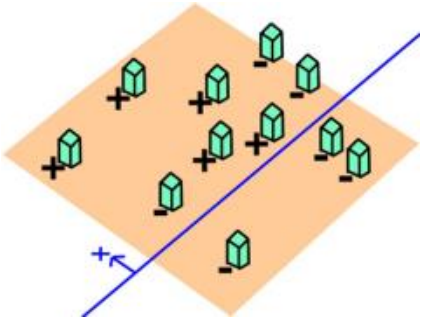
$$H_M(x) = sign[\sum_{m=1}^{M} \alpha_m h_m(x)] \qquad (1.4)$$

The algorithm starts by initializing the weights for the training samples. These weights are initialized to be equal. These weights tell the algorithm how important the sample is.



In each iteration, we do 2 things:

- Firstly: Find weak classifiers y_m based on the lowest error.



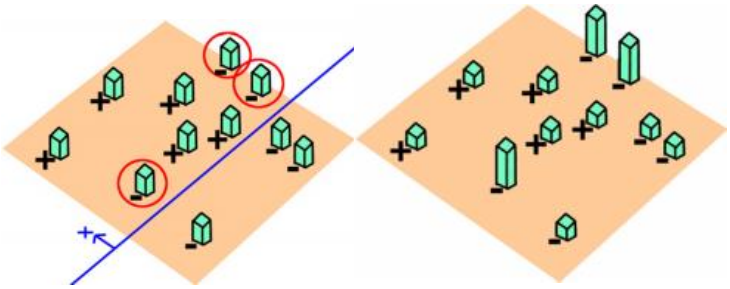
Formula for calculating error:

$$E_j = \sum_{n=1}^{N} w_n^{(m)} I(h_m(x_n) \neq t_n)$$

with $I(h_m(x_n) \neq t_n) = 1$ if $h_m(x_n) \neq t_n$ and = 0 otherwise.

This formula simply sums the weights of the misclassified samples.

- Sencondly: Update the weights according to the principle: we will increase the weight for samples that are currently misclassified and decrease the weight for samples that are currently correctly classified. In this way, in the next iteration we will build a weak classifier that focuses on solving samples misclassified by the previous weak classifier.



$$w_n^{(m+1)} = w_n^{(m)} \exp\{\alpha_m I(h_m(x_n) \neq t_n)\}$$

With:

$$\alpha_m = ln\frac{1-\epsilon_m}{\epsilon_m}$$

with:

$$\epsilon_m = \frac{\sum_{n=1}^{N} w_n^{(m)} I(h_m(x_n) \neq t_n)}{\sum_{n=1}^{N} w_n^{(m)}}$$

We see that if the sample is classified correctly, the weight does not change; and if the sample is misclassified then:

$$w_n^{(m+1)} = w_n^{(m)} \exp(\alpha_m) = w_n^{(m)} \frac{1-\epsilon_m}{\epsilon_m}$$

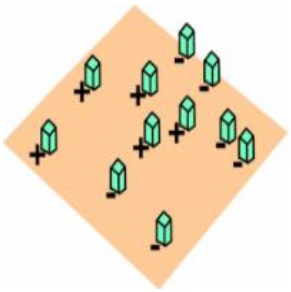
Clearly we have $\frac{1-\epsilon_m}{\epsilon_m} > 1$

Indeed, let's say if $\frac{1-\epsilon_m}{\epsilon_m} < 1$, that means the portion of wrong parts is greater than the right part. This contradicts the condition that the weak classifier is optimal (has the smallest error) because just by changing the direction of the classification plane, we immediately have a better weak classifier.

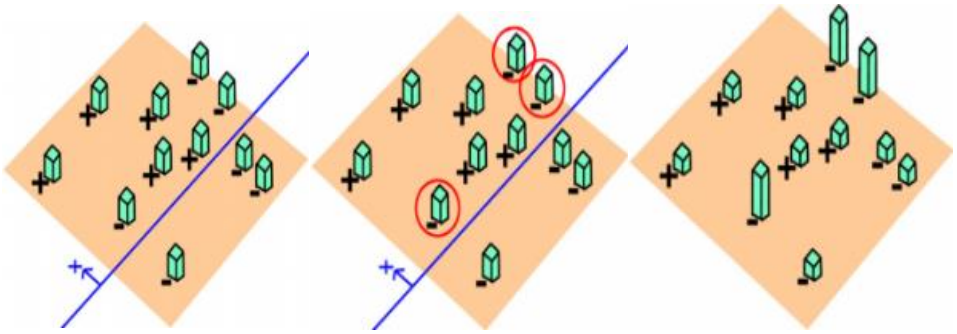
Then because $\frac{1-\epsilon_m}{\epsilon_m} > 1$ so $w_n^{(m+1)} > w_n^{(m)}$: The weight of misclassified samples is increased.

To visualize, let's observe the algorithm through the series of drawings below:

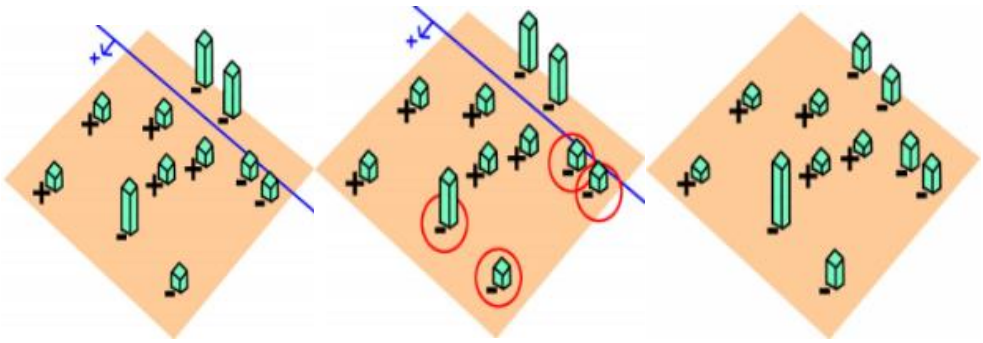
- Initialize weights for samples:



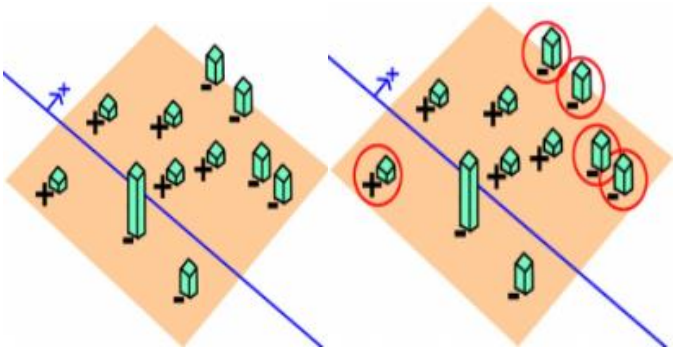
- First iteration:



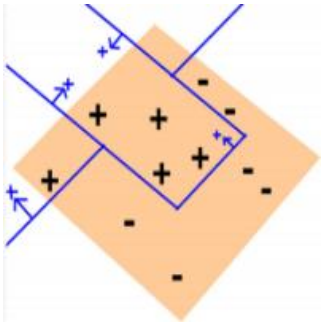
- Second iteration:



- Third iteration:



- Combine weak classifiers:



Finally, linearly combining the weak classifiers results in a strong classifier:

$$H_M(x) = sign[\sum_{m=1}^{M} \alpha_m h_m(x)]$$

It is found that the contribution level of each weak classifier to the strong classifier is determined by α_m:

$$\alpha_m = ln\frac{1-\epsilon_m}{\epsilon_m}$$

Clearly α_m inversely proportional to ϵ_m and ϵ_m represents the error corresponding to the classifier h_m. Thus, the lower the error level and the better the classification focus, the more likely it will contribute to the final classifier.

More about the stopping condition: in practice, people use a threshold value of the maximum false positive rate (max false positive) as the stopping condition. Through the iterations, the false recognition rate of the strong classifier will gradually decrease. At some point, this rate is smaller than the maximum false recognition rate and we will stop the algorithm.

2.1.4 Cascade of classifier

Visualize: https://www.youtube.com/watch?v=hPCTwxF0qf4

2.1.4.1 Definition

After understanding the Adaboost algorithm, our first thought is to use the Adaboost algorithm to train a strong classifier. Then, we spread sub windows, containing this strong classifier, throughout an input image to detect faces.

However, this method is costly. Suppose, using Adaboost we get a strong classifier consisting of 10 weak classifiers. If we do as above, in all sub-windows on the image we

will have to use all 10 weak classifiers. Meanwhile, we see: the windows that are actually faces are very few and the windows that are not faces can be eliminated by just using a strong classifier consisting of less than 10 weak classifiers.

Therefore, Viola and Jones proposed solution for this problem: using a cascade of classifiers.

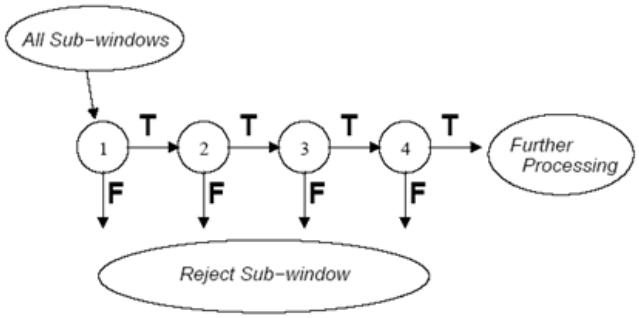
So what is cascade of classifiers?

We will have a series of classifiers, in which each classifier is built using the Adaboost algorithm. Now, we pass all the child windows through this sequence of classifiers:

- The first classifier will remove most of the non-face windows (nagative sub windows) and pass through the windows that are considered faces (positive sub windows). Here, the classifier is very simple and therefore the computational complexity is also very low. Of course, because it's that simple, among the windows that are recognized as faces, there will be a large number of windows that are incorrectly recognized (as not being faces aka false positive)

- The windows allowed to pass through by the first classifier will be considered by the subsequent classifier: if the classifier thinks it is not a face, we remove it; If the classifier thinks it's a face, we let it pass and move it to the next classifier.

- Later classifiers become more complex and require more computation. People call the sub-windows (samples) that the classifier cannot eliminate as hard-to-recognize samples. The deeper these patterns go in the classifier chain, the harder they are to identify. Only windows that pass through all classifiers can we decide that they are faces (highly possible)



In short, the chain of classifiers will handle incoming samples (sub-windows) according to the following principle: if a classifier thinks that it is not a human face, we immediately remove it; And if the classifier thinks it's a face, we move to the next classifier. If a sample passes all the classifiers, then we will decide that it is a face.

2.1.4.2 Training a Cascade of Classifiers

Firstly, we need to talk about two important measures: detection rate and false positive rate:

- Detection rate (true positive rate) = Number of regions recognized as faces and correct / Number of actual faces in the image.

- False positive rate = Number of regions recognized as faces and false / Number of regions recognized as faces

Consider a cascade consisting of K classifiers.

- The Detection rate of the cascade is calculated as follows:

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

$$with\ d_i is\ detection\ rate\ of\ classifier\ i(th)$$

- The False positive rate of the cascade is calculated as follows:

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

$$with\ f_i is\ false\ positive\ rate\ of\ classifier\ i(th)$$

Training algorithm:

- **Step 1**: Specify these values.
  + f: maximum accepted false positive rate of each classifier.
  + d: smallest acceptable detection rate of each classifier.
  + F_{target}: highest acceptable false positive rate of the cascade
- **Step 2**: Initialize.
  + P = set of samples that are faces (positive.)
  + N = set of non-face samples (negative.)

+ $F_0 = 1.0$; $D_0 = 1.0$

+ i = 0. //Index of the classifier in the cascade

- **Step 3**: While $F_i > F_{target}$:

+ i = i+1 (Classifier ith)

+ $n_i = 0$ (Number of weak classifiers of the ith classifier)

+ $F_i = F_{i-1}$

+ While $F_i > f * F_{i-1}$ (Cause we need later classifier have smaller threshold)

. $n_i = n_i + 1$

. Apply Adaboost algorithm: Using sample set P and N to train this ith classifier with n_i features (n_i weak-classifiers)

. Calculate F_i and D_i of current cascade via validation set.

. Decrease the threshold of the ith classifier until the current cascade achieves at least a detection rate $d * D_{i-1}$ (note that F_i also change)

+ N = 0

+ If $F_i > F_{target}$: Input these non-face samples into the current cascade and perform detection. Samples that are recognized as faces will be put into N again.

As mentioned, the processing principle of the cascade is that if a classifier says the sample is not a face, then remove it immediately, and if the classifier says it is a face, we pass it on to the classifier behind. Therefore, we see that in the algorithm, when building the cascade classifiers, the set P (corresponding to the set of face images) is always constant; The set N (corresponding to the set of non-face images) is changed: Suppose we have built a cascade to the ith classifier. We will put non-face images into the current cascade and perform detection; Images that are considered faces will be put into set N to continue building the i+1 classifier.

2.2 MTCNN

2.2.1 Why need to improve?

The difficulties in face detection mainly come from two aspects:

- The large visual variations of human faces in the cluttered backgrounds;
- The large search space of possible face positions and face sizes.

The former one requires the face detector to accurately address a binary classification problem while the latter one further imposes a time efficiency require.

Ever since the seminal work of Viola-Jones, the boosted cascade with simple features becomes the most popular and effective design for practical face detection. The simple nature of the features enable fast evaluation and quick early rejection of false positive detections. The original Viola-Jones face detector uses the Haar feature which is fast to evaluate yet discriminative enough for frontal faces. However, due to the simple nature of the Haar feature, it is relatively weak in the uncontrolled environment where faces are in varied poses, expressions under unexpected lighting. A number of improvements to the Viola-Jones face detector have been proposed in the past decade. Most of them follow the boosted cascade framework with more advanced features. However, the number of cascade stages required to achieve the similar detection accuracy can be reduced. Hence the overall computation may remain the same or even reduced because of fewer cascade stages.

This observation suggests that it is possible to apply more advanced features in a practical face detection solution as long as the false positive detections can be rejected quickly in the early stages. In recently, they propose to apply the Convolutional Neural Network (CNN) to face detection. Compared with the previous hand-crafted features, CNN can automatically learn features to capture complex visual variations by leveraging a large amount of training data and its testing phase can be easily parallelized on GPU cores for acceleration.

2.2.2 Neural Network

2.2.2.1 Definition

Artificial Neural Network (ANN) is an information processing model that imitates the information processing method of biological neural systems. It is made up of a large number of elements (neurons) connected to each other through links (weights) working as a unified whole to solve a specific problem. An artificial neural network is configured for a specific application (pattern recognition, data classification,...) through a learning process from a set of training samples. In essence, learning is the process of adjusting the weights of connections between neurons.

2.2.2.2 Key components of the Neural Network Architecture

The Neural Network architecture is made of individual units called neurons that mimic the biological behavior of the brain. Here are the various components of a neuron.

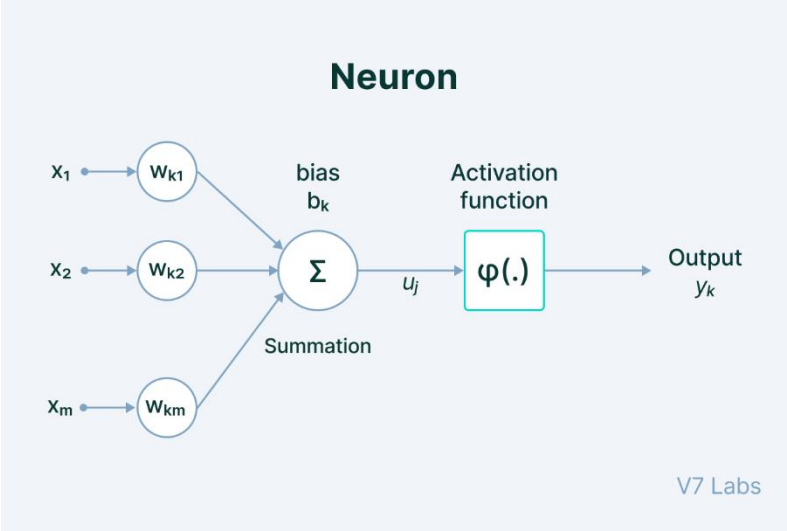


Figure 3

- **Input**: It is the set of features that are fed into the model for the learning process. For example, the input in object detection can be an array of pixel values pertaining to an image.

- **Weight**: Its main function is to give importance to those features that contribute more towards the learning. It does so by introducing scalar multiplication between the input value and the weight matrix. For example, a negative word would impact the decision of the sentiment analysis model more than a pair of neutral words.

- **Transfer function**: The job of the transfer function is to combine multiple inputs into one output value so that the activation function can be applied. It is done by a simple summation of all the inputs to the transfer function.

- **Activation Function**: It introduces non-linearity in the working of perception to consider varying linearity with the inputs. Without this, the output would just be a linear combination of input values and would not be able to introduce non-linearity in the network.

- **Bias**: The role of bias is to shift the value produced by the activation function. Its role is similar to the role of a constant in a linear function.

When multiple neurons are stacked together in a row, they constitute a layer, and multiple layers piled next to each other are called a multi-layer neural network. We've described the main components of this type of structure below.

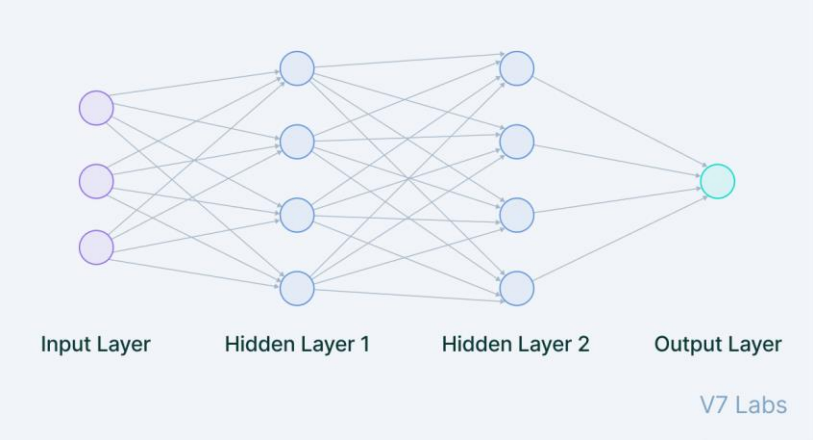


Figure 4

- **Input Layer**: The data that we feed to the model is loaded into the input layer from external sources like a CSV file or a web service. It is the only visible layer in the complete Neural Network architecture that passes the complete information from the outside world without any computation.

- **Hidden Layers**: The hidden layers are what makes deep learning what it is today. They are intermediate layers that do all the computations and extract the features from the data. There can be multiple interconnected hidden layers that account for searching different hidden features in the data. For example, in image processing, the first hidden layers are responsible for higher-level features like edges, shapes, or boundaries. On the other hand, the later hidden layers perform more complicated tasks like identifying complete objects (a car, a building, a person).

- **Output Layer**: The output layer takes input from preceding hidden layers and comes to a final prediction based on the model's learnings. It is the most important layer where we get the final result. In the case of classification/regression models, the output layer generally has a single node. However, it is completely problem-specific and dependent on the way the model was built.

2.2.3 Convolutional Neural Network

2.2.3.1 The problem of fully connected neural network with image processing

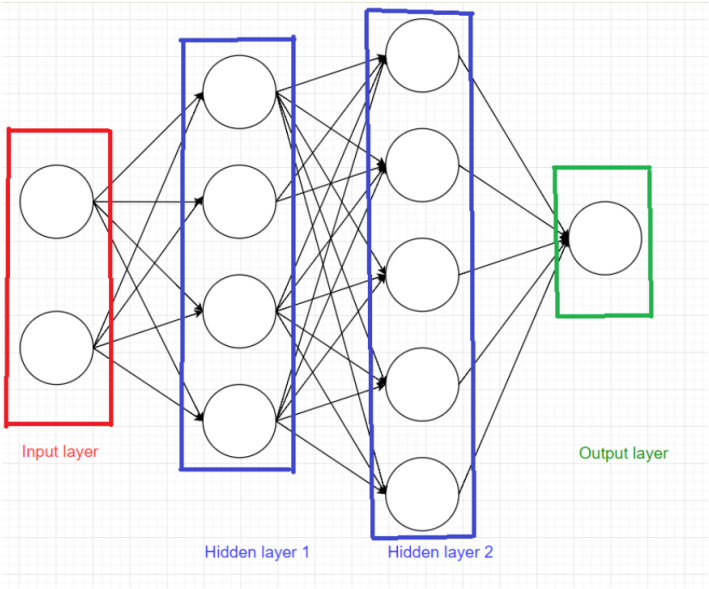


Figure 5

A computer sees an image as a matrix of numbers with (rows*columns*number of channels) shape. Any real-world image would be at least 64*64*3 pixels. So to represent the entire content of the image, it is necessary to pass into the input layer all the pixels (64*64*3 = 12288). This means the input layer now has 12288 nodes

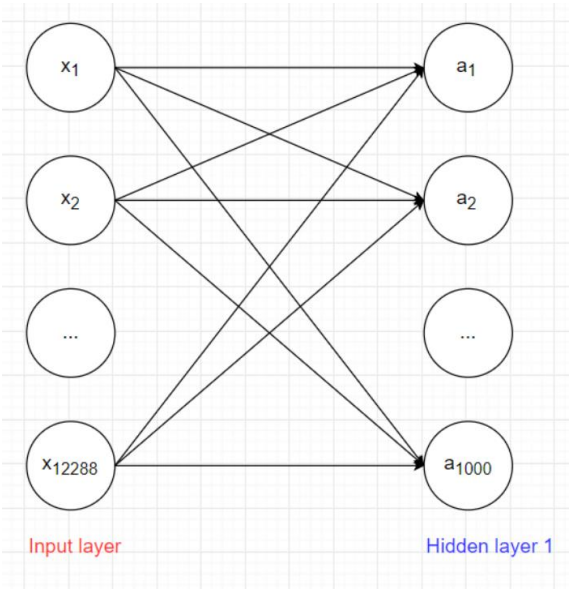


Figure 6

Suppose the number of nodes in hidden layer 1 is 1000. The number of weights W between the input layer and hidden layer 1 is 12288*1000 = 12288000, the number of biases is 1000 => the total number of parameters is: 12289000. That is just the number of parameters between the input layer and hidden layer 1, there are many more layers in the model, and if the image size increases, for example 512*512, the number of parameters increases extremely quickly => Need a better solution!!!

Dealing with such a huge amount of parameters requires many neurons and it may lead to overfitting. In contrast to that fully connected feedforward neural networks, convolutional neural networks look at one patch of an image at a time and move forward in this manner to derive complete information. It involves very few neurons with fewer parameters to scan an entire image to learn essential features.

2.2.3.2 Definition

Convolution neural network (also known as ConvNet or CNN) is a type of feed-forward neural network used in tasks like image analysis, natural language processing, and other complex image classification problems.

a. Convolutional Layer:

CNN works by comparing images piece by piece. Filters are spatially small along width and height but extend through the full depth of the input image. It is designed in such a manner that it detects a specific type of feature in the input image.

In the convolution layer, we move the filter/kernel to every possible position on the input matrix. Element-wise multiplication between the filter-sized patch of the input image and filter is done, which is then summed.

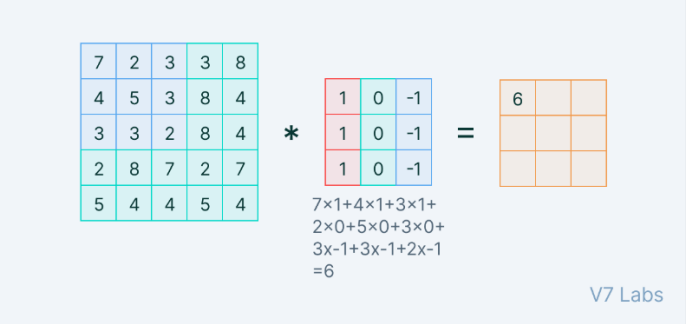


Figure 7

The translation of the filter to every possible position of the input matrix of the image gives an opportunity to discover that feature is present anywhere in the image. The generated resulting matrix is called the feature map. Convolution neural networks can learn from multiple features parallelly. In the final stage, we stack all the output feature maps along with the depth and produce the output.

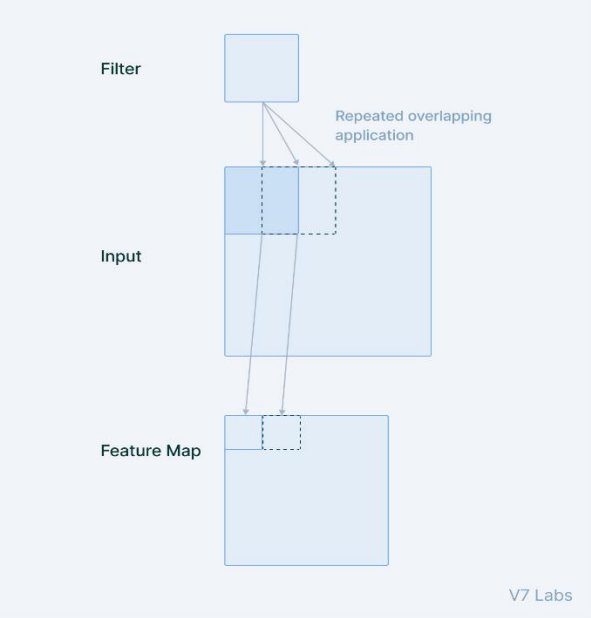


Figure 8

b. Spatial arrangement

We know that the images represented in a matrix of pixel values. The dimension increases depending on the size of the image. If all the neurons are connected to all previous neurons as in a fully connected layer, the number of parameters increases manifold.

To resolve this, we connect each neuron to only a patch of input data. This *spatial extent* (also known as the receptive field of the neuron) determines the size of the filter. So *spatial arrangement* is what governs the size of the neurons in the output volume and how they are arranged

Three hyperparameters that control the size of the output volume:

- **The depth**: The depth of the output volume is equal to the number of filters we use to look for different features in the image. The output volume has stacked activation/feature maps along with the depth, making it equal to the number of filters used.

- **Stride**: Stride refers to the number of pixels we slide while matching the filter with the input image patch. If the stride is one, we move the filters one pixel at a time. Higher the stride, smaller output volumes will be produced spatially.

- **Zero-padding**: It allows us to control the spatial size of the output volume by padding zeros around the border of the input data.

c. Pooling Layer

Pooling layers are added in between two convolution layers with the sole purpose of reducing the spatial size of the image representation.

The pooling layer has two hyperparameters:

- Window size

- Stride

From each window, we take either the maximum value or the average of the values in the window depending upon the type of pooling being performed. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, and later stacks them together.

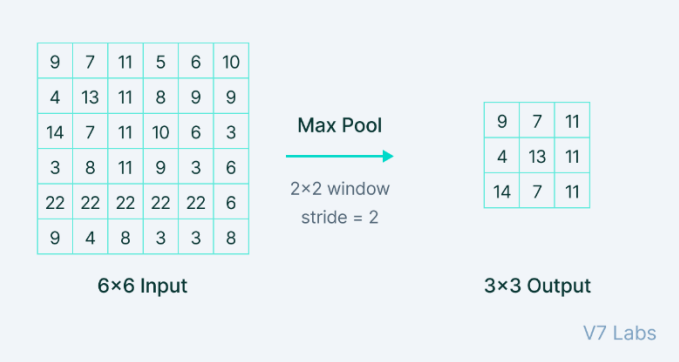


Figure 9

Types of Pooling:

- Max Pooling selects the maximum element from each of the windows of the feature map. Thus, after the max-pooling layer, the output would be a feature map containing the most dominant features of the previous feature map.

- Average Pooling computes the average of the elements present in the region of the feature map covered by the filter. It simply averages the features from the feature map.

d. Fully-Connected Layer

The Convolutional Layer, along with the Pooling Layer, forms a block in the Convolutional Neural Network. The number of such layers may be increased for capturing finer details depending upon the complexity of the task at the cost of more computational power.

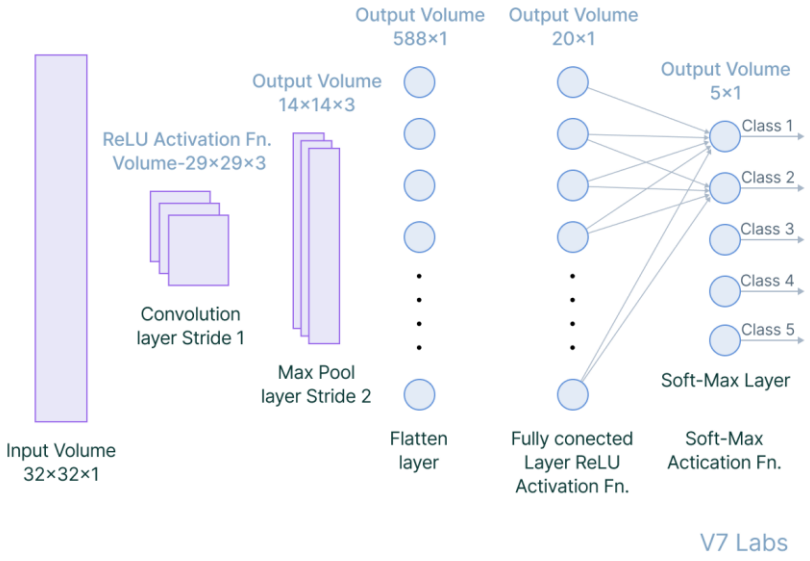


Figure 10

Having been able to furnish important feature extraction, we are going to flatten the final feature representation and feed it to a regular fully-connected neural network for image classification purposes.

2.2.3.3 How do Convolutional Neural Networks work?

A CNN has hidden layers of convolution layers that form the base of ConvNets. Like any other layer, a convolutional layer receives input volume, performs mathematical scalar product with the feature matrix (filter), and outputs the feature maps. Features refer to minute details in the image data like edges, borders, shapes, textures, objects, circles, etc.

At a higher level, convolutional layers detect these patterns in the image data with the help of filters. The higher-level details are taken care of by the first few convolutional layers. The deeper the network goes, the more sophisticated the pattern searching becomes.

For example, in later layers rather than edges and simple shapes, filters may detect specific objects like eyes or ears, and eventually a cat, a dog, and whatnot.

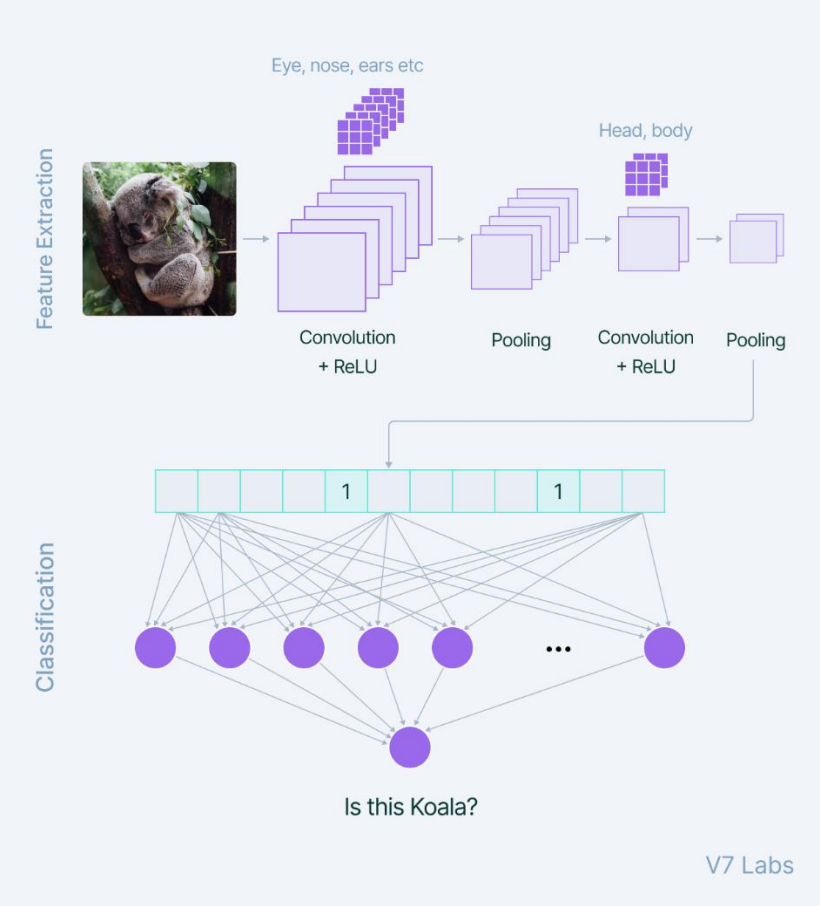


Figure 11

The first hidden layer in the network dealing with images is usually a convolutional layer.

When adding a convolutional layer to a network, we need to specify the number of filters we want the layer to have. A filter can be thought of as a relatively small matrix for which we decide the number of rows and columns this matrix has. The value of this feature matrix is initialized with random numbers. When this convolutional layer receives pixel values of input data, the filter will convolve over each patch of the input matrix.

The output of the convolutional layer is usually passed through the ReLU activation function to bring non-linearity to the model. It takes the feature map and replaces all the negative values with zero.

Then, the pooling layer is added in succession to the convolutional layer to reduce the dimensions. Pooling is a very important step in the ConvNet as reduces the computation and makes the model tolerant towards distortions and variations.

We take a window of say 2x2 and select either the maximum pixel value or the average of all pixels in the window and continue sliding the window. So, we take the feature map, perform a pooling operation, and generate a new feature map reduced in size.

That why we need the convolutional layer, it response for the feature extraction.

Then we apply a fully connected dense neural network by flattening feature matrix and predict according to the use case.

2.2.4 Architecture MTCNN

The MTCNN is just a model consists of 3 separate networks: the P-Net, the R-Net, and the O-Net.

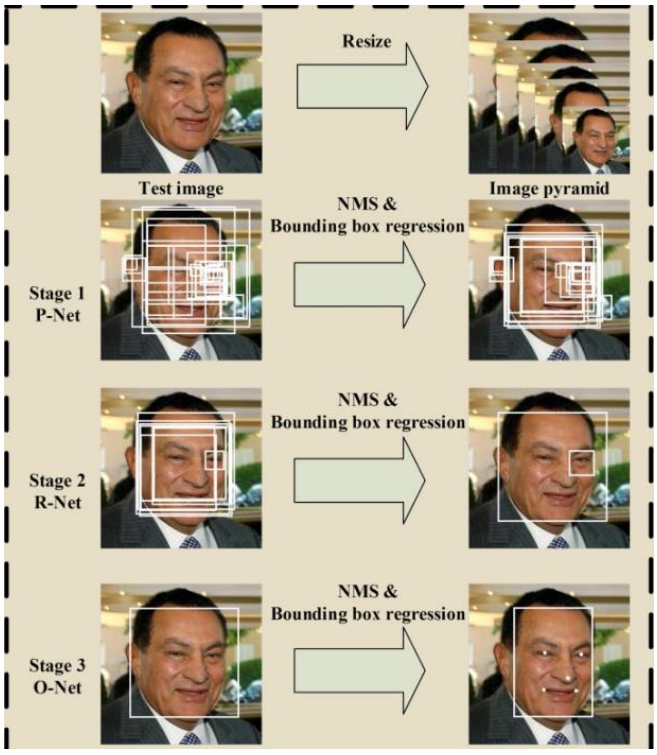


Figure 12

The overall pipeline of our approach is shown in above. Given an image, we initially resize it to different scales to build an image pyramid, which is the input of the following three-stage cascaded framework:

Stage 1: We exploit a fully convolutional network, called Proposal Network (P-Net), to obtain the candidate windows and their bounding box regression vectors. Then we use the estimated bounding box regression vectors to calibrate the candidates. After that, we employ non-maximum suppression (NMS) to merge highly overlapped candidates.

Stage 2: all candidates are fed to another CNN, called Refine Network (R-Net), which further rejects a large number of false candidates, performs calibration with bounding box regression, and NMS candidate merge.

Stage 3: This stage is similar to the second stage, but in this stage we aim to describe the face in more details. In particular, the network will output five facial landmarks' positions.

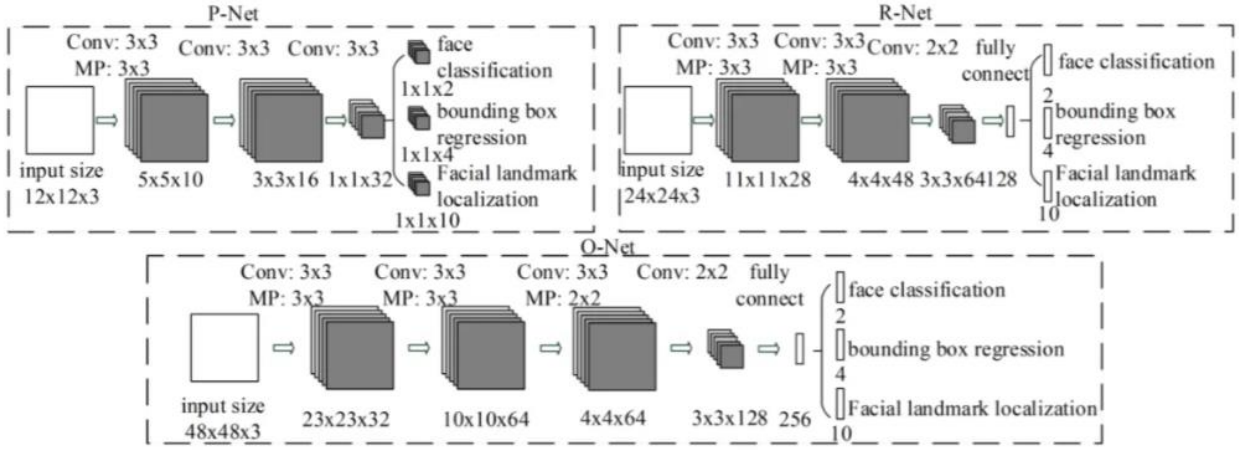


Figure 13

2.2.4.1 Stage 1: The Proposal Network (P-Net)

The first thing to do would be to pass in an image to the program. In this model, we want to create an image pyramid, in order to detect faces of all different sizes. In other words, we want to create different copies of the same image in different sizes to search for different sized faces within the image.

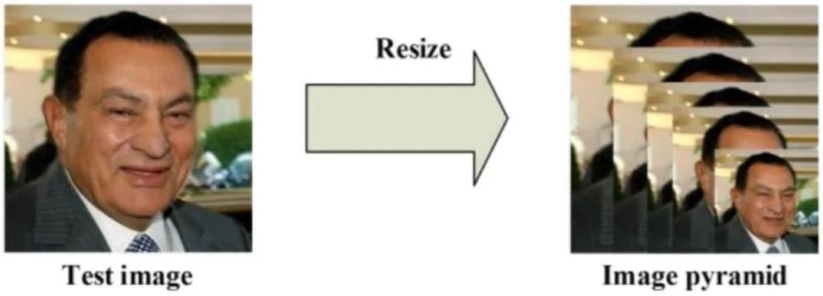


Figure 14

For each scaled copy, we have a 12 x 12 stage 1 **kernel (filter)** that will go through every part of the image, scanning for faces. It starts in the top left corner, a section of the image from (0,0) to (12,12). This portion of the image is passed to P-Net, which returns the coordinates of a bounding box if it notices a face. Then, it would repeat that process with sections (0+2a,0+2b) to (12+2a, 12+2b), **shifting (stride)** the 12 x 12 kernel 2 pixels right or down at a time.

Then come to the first stage. This is a fully convolutional network (FCN). The difference between a CNN and a FCN is that a fully convolutional network does not use a dense layer as part of the architecture. This Proposal Network is used to obtain candidate windows and their bounding box regression vectors.

Bounding box regression is a popular technique to predict the localization of boxes when the goal is detecting an object of some pre-defined class, in this case faces. After obtaining the bounding box vectors, some refinement is done to combine overlapping regions. The final output of this stage is all candidate windows after refinement to downsize the volume of candidates.

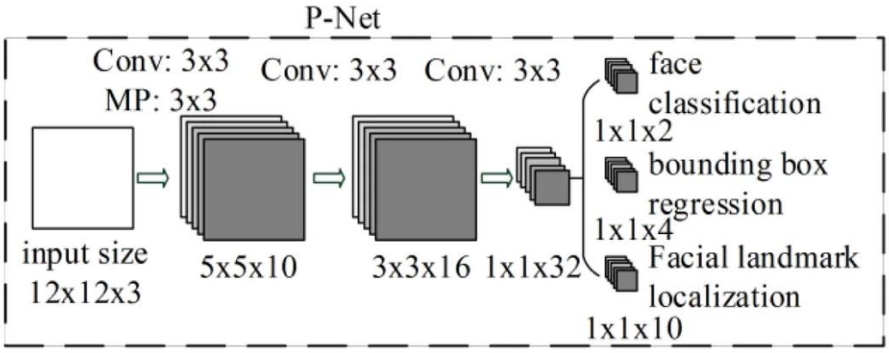


Figure 15

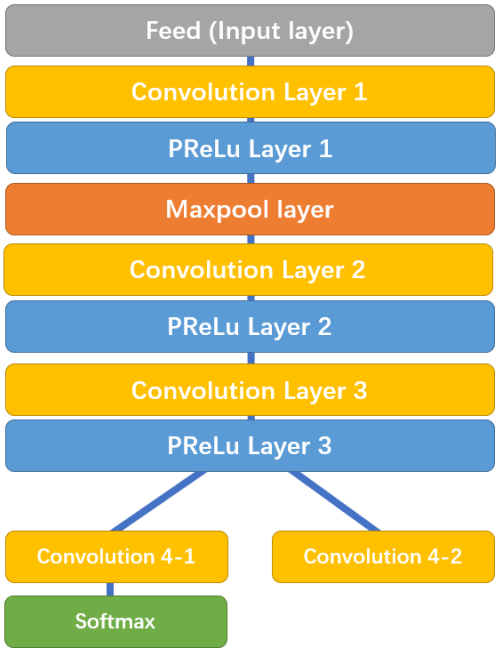


Figure 16

Note that after PReLu layer 3, it splits into 2 branches, convolution 4–1 outputs the probability of a face being in each bounding box, and convolution 4–2 outputs the coordinates of the bounding boxes.

2.2.4.2 Stage 2: The Refine Network (R-Net)

R-Net has a similar structure, but with even more layers. It takes the P-Net bounding boxes as its inputs, and refines its coordinates. Notice that this network is a CNN, not a FCN like the one before since there is a dense layer at the last stage of the network architecture. The R-Net further reduces the number of candidates, performs calibration with bounding box regression and employs non-maximum suppression (NMS) to merge overlapping candidates.

The R-Net outputs whether the input is a face or not, a 4 elements vector which is the bounding box for the face, and a 10 elements vector for facial landmark localization.

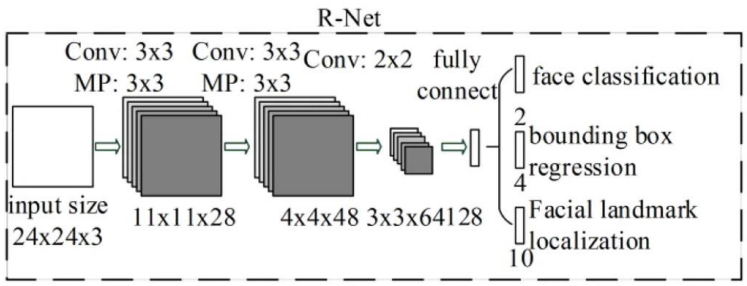


Figure 17

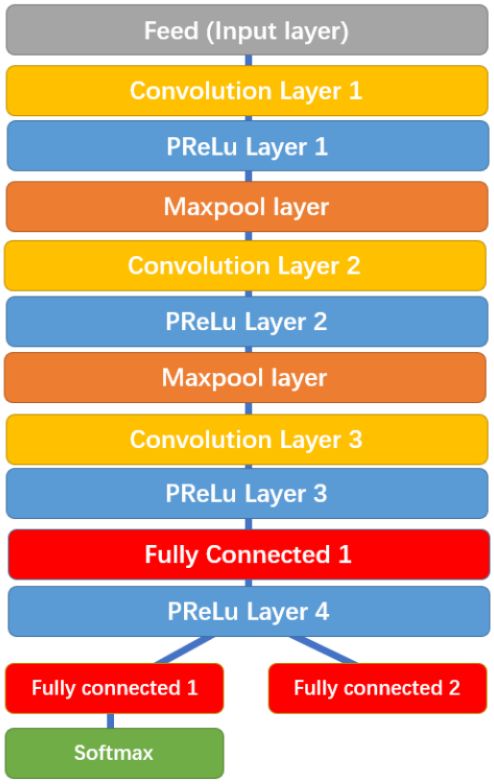


Figure 18

Similarly, R-Net splits into two layers in the end, giving out two outputs: the coordinates of the new bounding boxes and the machine's confidence in each bounding box.

2.2.4.3 Stage 3: O-Net

This stage is similar to the R-Net, but this Output Network aims to describe the face in more detail and output the five facial landmarks' positions for eyes, nose and mouth.

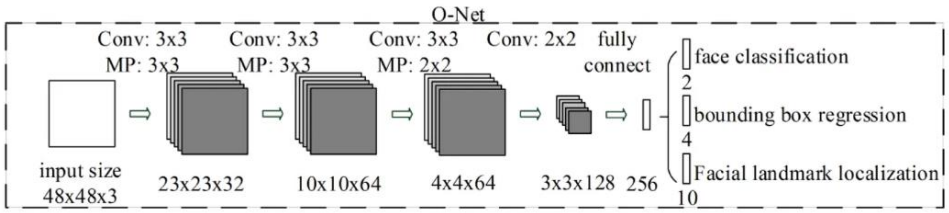


Figure 19

Figure 20

O-Net splits into 3 layers in the end, giving out 3 different outputs: the probability of a face being in the box, the coordinates of the bounding box, and the coordinates of the facial landmarks (locations of the eyes, nose, and mouth).

2.2.4.4 The Three Tasks of MTCNN

The Network's task is to output three things: face/non-face classification, bounding box regression, and facial landmark localization.

Face classification: this is a binary classification problem that uses cross-entropy loss:

$$L_i^{det} = -(y_i^{det} \log(p_i) + (1 - y_i^{det})(1 - \log(p_i)))$$

ground truth label $y_i^{det} \in \{0, 1\}$

probability produced by the network

Bounding box regression: the learning objective is a regression problem. For each candidate window, the offset between the candidate and the nearest ground truth is calculated. Euclidean loss is employed for this task:

$$L_i^{box} = \| \hat{y}_i^{box} - y_i^{box} \|_2^2$$

ground truth coordinate

target obtained from network

Facial Landmark localization: There are five landmarks: left eye, right eye, nose, left mouth corner and right mouth corner. The localization of facial landmarks is formulated as a regression problem, in which the loss function is Euclidean distance:

$$L_i^{landmark} = \| \hat{y}_i^{landmark} - y_i^{landmark} \|_2^2$$

2.3 YOLOv8 (You Only Live Once)

YOLOv8 or You Only Live Once v8 is a state-of-art algorithm that was released since May 2023. It's known as the latest version of YOLO family of algorithms that famous for their speed and accuracy.

In this project we will briefly go through the architecture of YOLO model and how it will be applied into our project.

2.3.1 Why YOLO different?

In consider to others model for applying object detection. Let's come in hand with the R-CNN (Region-based Convolutional Neural Network); this method first generate potential bounding boxes in an image and run a classifier on this proposed boxes. After classification, post-processing is used to refine the bounding boxes, which involves of hard and slow pipeline. On the other hand, YOLO present these processes with a simple convolutional neural network to sketch the bounding boxes and simultaneously calculate the class probabilities for those boxes as well. In simple words, it unified the separate components of object detection into a single neural network.

2.3.2 Architecture and works

The system divides the input images into an S x S grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object.

Each grid cell predicts B bounding boxes and confidence scores for these boxes. If no object exists in that cell, the confidence scores should be zero. Otherwise, we want the confidence scores to equal the intersection over union (IOU) between the predicted box and the ground truth.

Each bounding box consists of 5 predictions: x,y,w,h and the confidence. The (x,y) coordinates represents the center of the box relative to the bounds of grid cell.

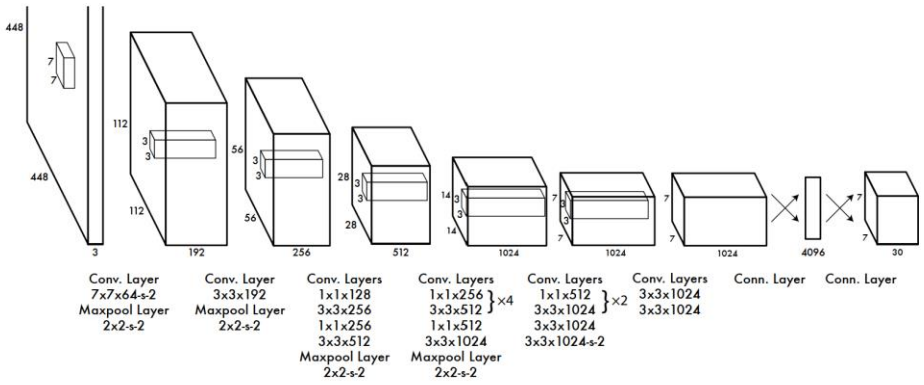


Figure 3: The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224 input image) and then double the resolution for detection.

The system contains 24 convolutional layers for features extracting and 2 fully connected to refine for calculating the ouput probabilities and coordinates.

A loss function also in accounted through layers in the system:

$$\begin{aligned}
&\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p}_i(c) \right)^2 \qquad (3)
\end{aligned}$$

2.4 HOG (Histogram of Oriented Gradients)

HOG combined with SVM (Support Vector Machine) is widely used to object detection. Firstly, there was used ready HOG algorithm from OpenCV package to check if chosen approach make sense. Finally, HOG algorithm was implemented and used with SVM model.

The main idea of HOG is to use histograms of gradients directions as a feature descriptor. Algorithm steps:

1. Find Intensity Gradient of the Image:

Filter smoothened image with a Sobel filter in both horizontal and vertical direction to get first derivatives g_x and g_y. Then count edge gradient magnitude g and direction θ for each pixel as follows:

$$g = \sqrt{g_x^2 + g_y^2}$$

$$\theta = \arctan \frac{g_y}{g_x}$$

2. Calculate histograms of gradients in chosen cells.

Divide image to equally size cells. In our case 64x64 px images where split to 64 8x8 cells. Now in each cell calculate histograms of gradients values with respect to gradients degrees. There were chosen 9 splits on angles 0, 20, 40, 60, 80, 100, 120, 140 and 160 degrees. Example:

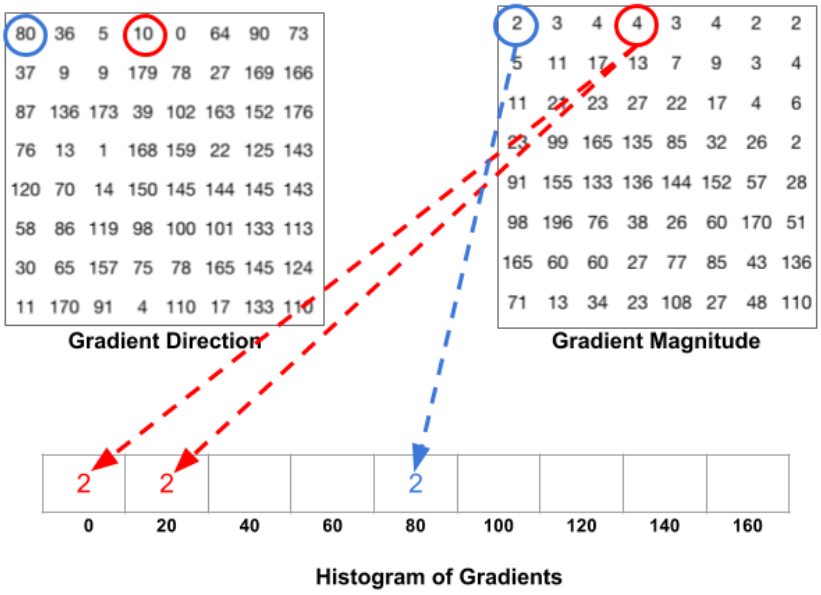


Figure 21

Gradient magnitude in first pixel is 2 and direction is 80 degrees so whole value (2) is attached to 80 degrees in histogram. On the other hand, in pixel number 4 there is magnitude=4 and direction=10 therefore value 4 is split equally between angles 20 and 40 in the histogram.

3. Block normalization.

Initialize block size as a multiplicity of cell size. In our case it is 16x16 px. Now for all histograms obtained in previous step, their values are normalized within block. Results are attached to final feature list. Afterwards block slides about one step (in our case 8 pixels) and operation is repeated. Whole process is running in a loop for a whole image. Results are returned as a final feature vector. Example:

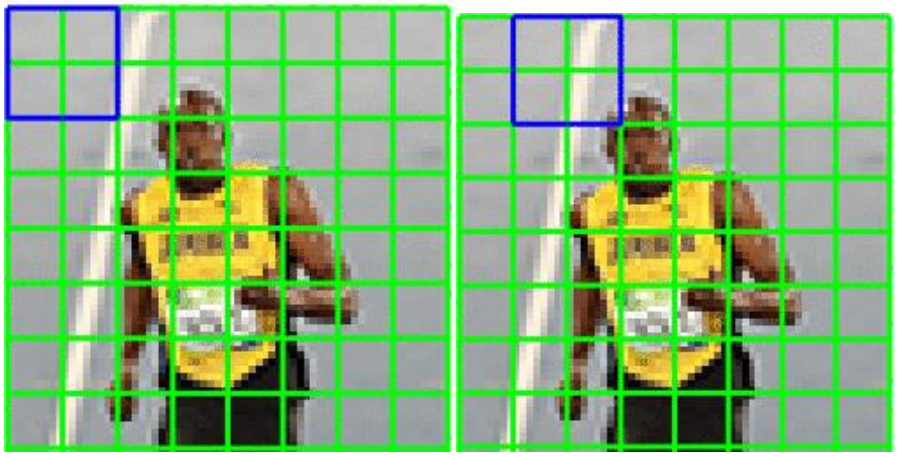


Figure 22

In the first image block covers 4 cells, histograms of those cells are normalized and obtained values are added to final results list. In the second image we can see that window slided about one step and whole operation is repeated.

3. Implementation

3.1. Haar-like Cascade Classifier

```
import cv2 as cv
from google.colab.patches import cv2_imshow


classifier= cv.CascadeClassifier('/content/gdrive/MyDrive/ Face_data/
test_folder/haarcascade_frontalface_default.xml')
pixels=cv.imread('/content/gdrive/MyDrive/ Face_data/
test_folder/test1.jpg')
pixels2=cv.imread('/content/gdrive/MyDrive/ Face_data/
test_folder/test2.jpg')
bboxes=classifier.detectMultiScale(pixels)
bboxes1=classifier.detectMultiScale(pixels2)
for box in bboxes:
  x,y,width,height=box
  x2,y2=x+width,y+height
  cv.rectangle(pixels, (x,y),(x2,y2),(0,0,255),2)


cv2_imshow( pixels)

for box in bboxes1:
  x,y,width,height=box
x2,y2=x+width,y+height
  cv.rectangle(pixels2, (x,y),(x2,y2),(0,0,255),2)

cv2_imshow( pixels2)

cv.destroyAllWindows()
```

The code snippet above shows the implementation of the Haar-like Cascade classifier using opencv as cv2 library. The code load the classifier named CascadeClassifer from cv2

as the Haar-like Cascade Classifier. The classifier then find boxes through iterations of the images and show the results.

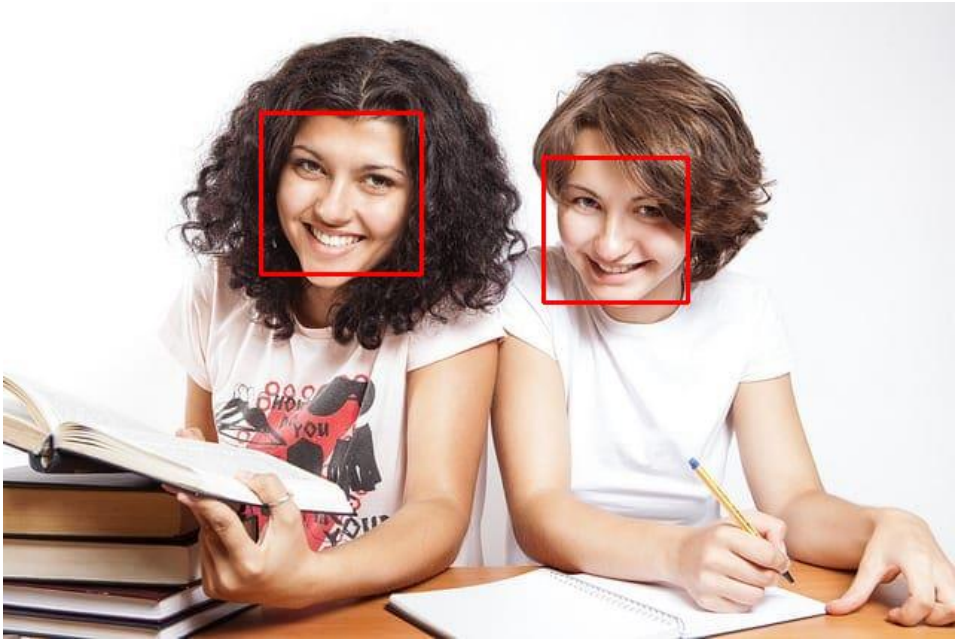


Figure 22

Figure 23

3.2. MTCNN

```
from mtcnn.mtcnn import MTCNN
import cv2
from matplotlib import pyplot
from matplotlib.patches import Rectangle
from matplotlib.patches import Circle
def draw_images_in_boxes(fileName, result_list):
  data = pyplot.imread(fileName)
  pyplot.imshow(data)

  axis = pyplot.gca()

  for result in result_list:
    x,y,width,height = result['box']

    rect = Rectangle((x,y),width,height,fill=False,color='red')

    axis.add_patch(rect)
    for _,value in result['keypoints'].items():
      dot = Circle(value, radius=2,color ='purple')
      axis.add_patch(dot)
  pyplot.show()

pixels = pyplot.imread('/content/gdrive/MyDrive/ Face_data/
test_folder/test1.jpg')

detector = MTCNN()

faces = detector.detect_faces(pixels)
fileName = '/content/gdrive/MyDrive/ Face_data/ test_folder/test1.jpg'
draw_images_in_boxes(fileName, faces)
```

The code snippet like the previous applied another classifier, which is MTCNN by loading the MTCNN model from built-library.

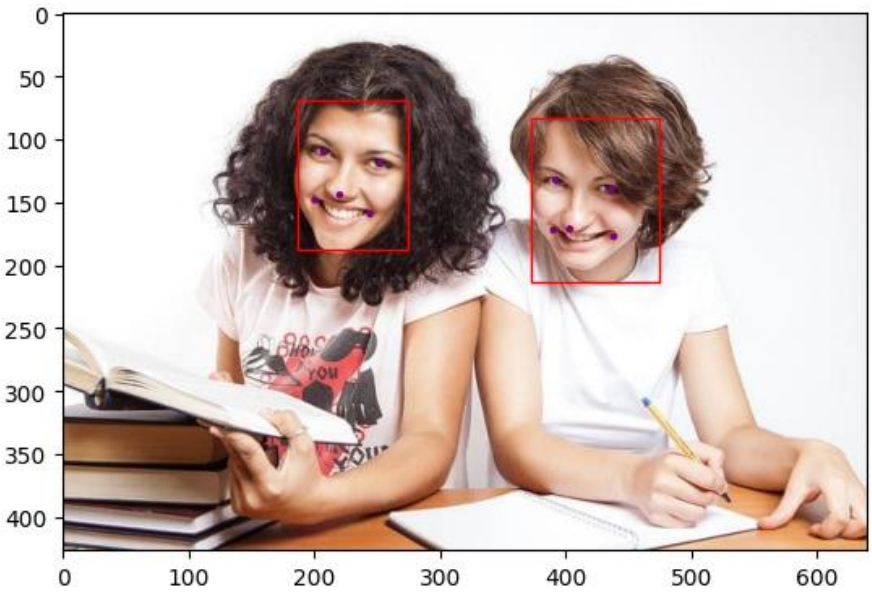


Figure 24

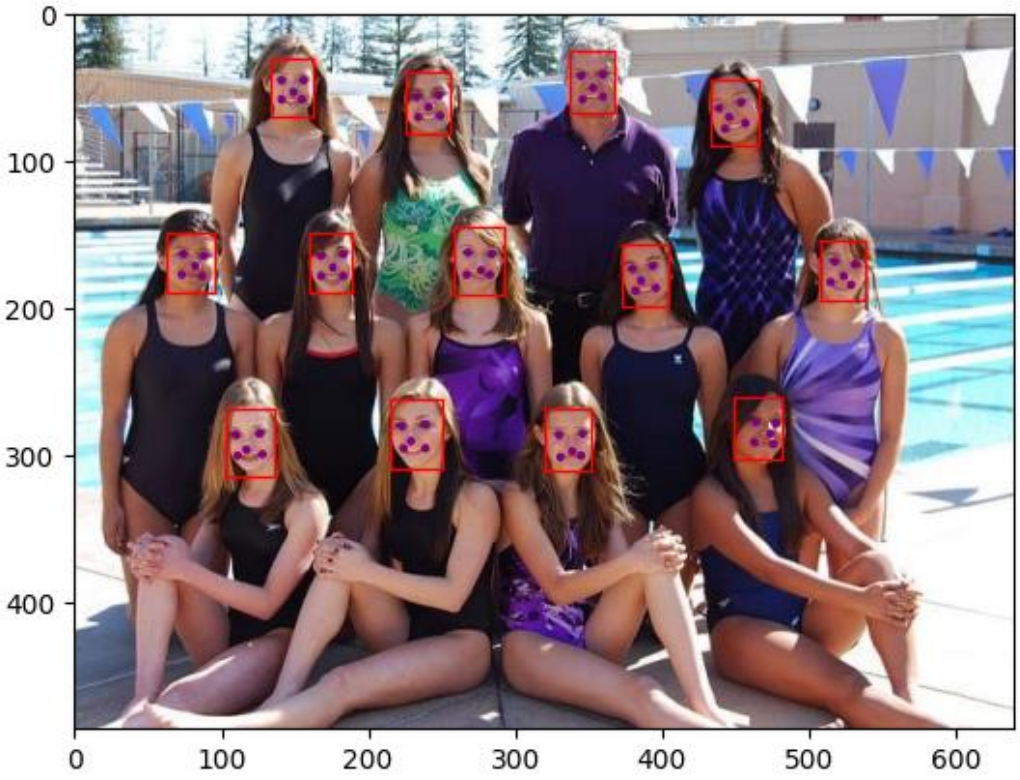


Figure 25

The classifier also shown the landmarks on faces. And also in the second image, the detector has recognized the girl on the left most of the image that has been ignored by the first detector.

3.3. YOLOv8

We used the YOLO model loaded from the **ultralytics** library:

```
 !pip install roboflow

from roboflow import Roboflow
rf = Roboflow(api_key="IYi8e2X7hj437sWLrWpe")
project = rf.workspace("fisrt-one").project("face_detection-tvmir")
dataset = project.version(1).download("yolov8")
```

And also the labled data that we prepared from a webpage called roboflow. The data then be separated into train, test and validate dataset.

```
!yolo task=detect mode=train model=yolov8m.pt
data={dataset.location}/data.yaml epochs=30 imgsz=640



Ultralytics YOLOv8.0.234 🚀 Python-3.10.12 torch-2.1.0+cu121 CUDA:0 (Tesla
T4, 15102MiB)
Model summary (fused): 218 layers, 25840339 parameters, 0 gradients, 78.7
GFLOPs
val: Scanning /content/face_detection-1/valid/labels.cache... 13 images, 0
backgrounds, 0 corrupt: 100% 13/13 [00:00<?, ?it/s]
                 Class     Images  Instances      Box(P          R
mAP50  mAP50-95): 100% 1/1 [00:01<00:00,  1.76s/it]
                   all         13         17          1      0.706
0.971       0.89
Speed: 0.2ms preprocess, 48.0ms inference, 0.0ms loss, 69.2ms postprocess
per image
Results saved to runs/detect/val3
💡 Learn more at https://docs.ultralytics.com/modes/val
!yolo task=detect mode=val
model=/content/runs/detect/train7/weights/best.pt
data={dataset.location}/data.yaml
```

```
!yolo task=detect mode=predict
model=/content/runs/detect/train7/weights/best.pt conf=0.5
source={dataset.location}/test/images
!yolo task=detect mode=predict
model=/content/runs/detect/train7/weights/best.pt conf=0.5
source={dataset.location}/test/images
import glob
from IPython.display import Image, display

for image_path in glob.glob('/content/runs/detect/predict3/*.jpg'):
    display(Image(filename=image_path, height=400))
    print('\n')
```

Examples:

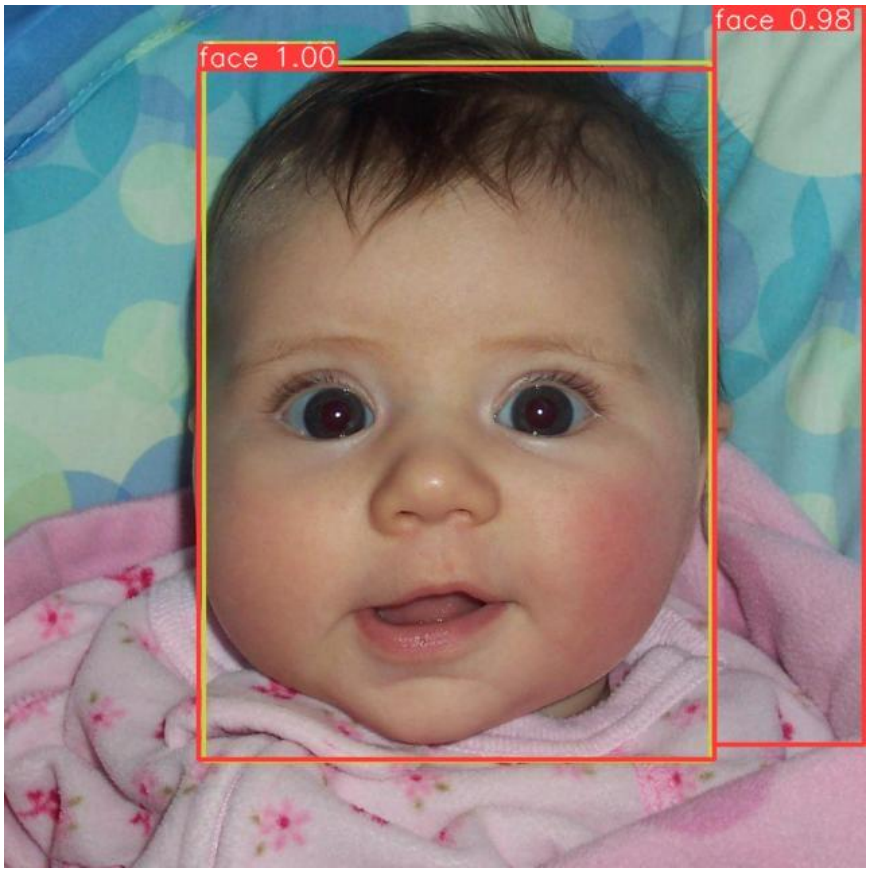


Figure 26

Figure 27

Figure 28

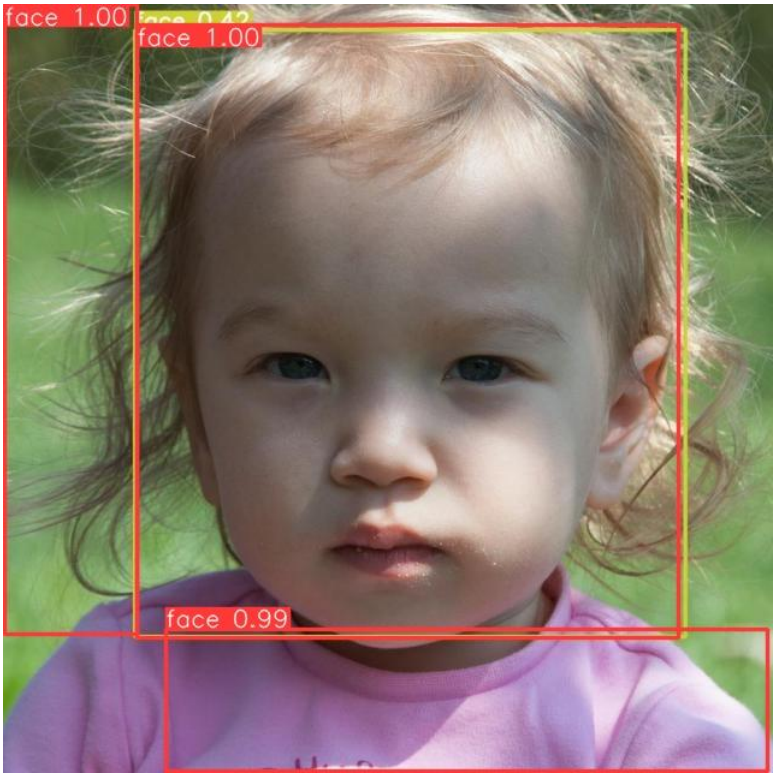


Figure 29

Figure 30

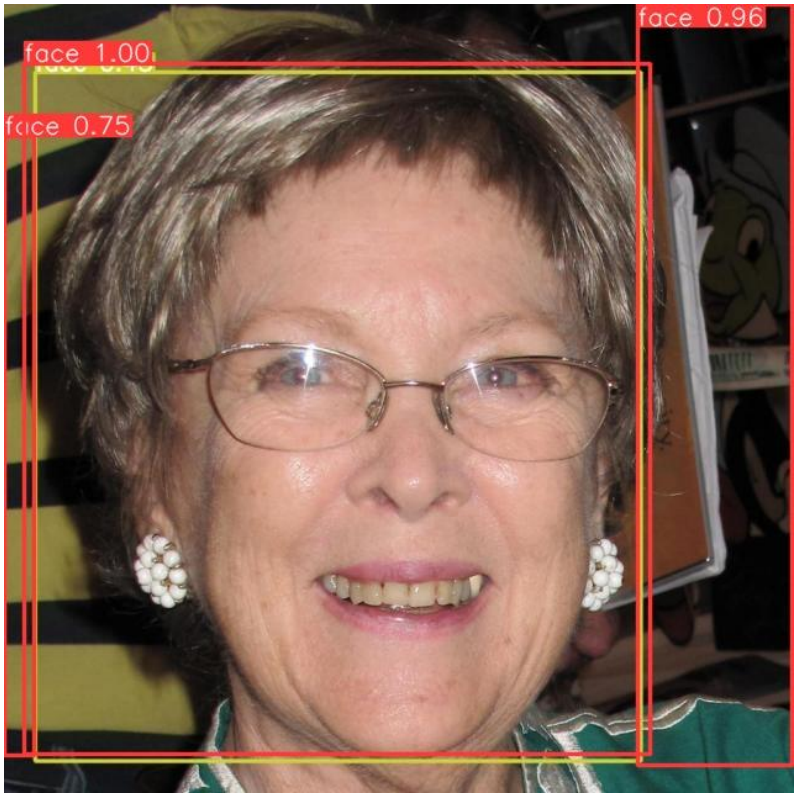


Figure 31

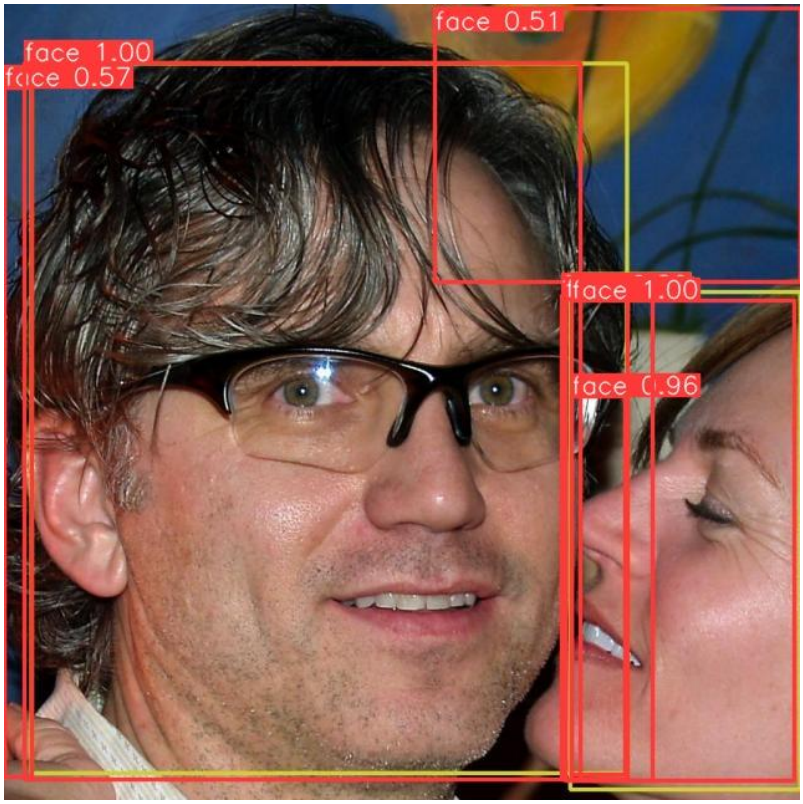


Figure 32

The above images are the results after trained the YOLOv8 detector with the training dataset.

3.4. HOG

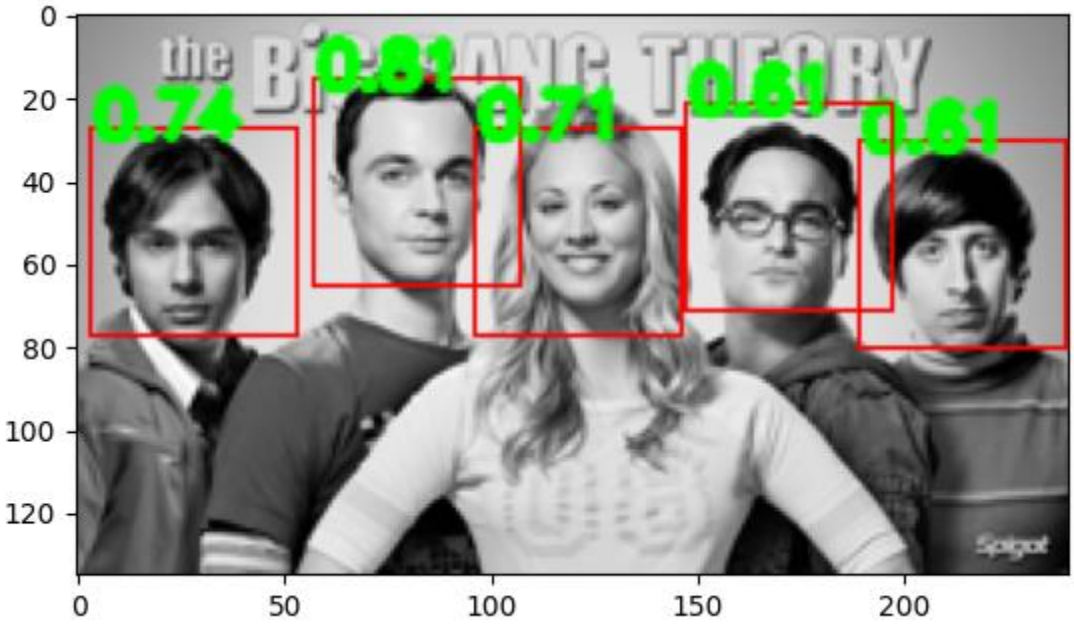


Figure 33

The above image after applying the HOG in detect faces .

References

[1]. Paul Viola, Michael Jones. Rapid Object Detection using a Boosted Cascade of Simple Features. paper.dvi (cmu.edu)

[2]. Satya Mallick. Histogram of Oriented Gradients explained using OpenCV, 2016. URL https://learnopencv.com/histogram-of-oriented-gradients/. Last accessed 10 January 2023.

[3]. Mrinal Tyagi. Viola Jones Algorithm and Haar Cascade Classifier. Viola Jones Algorithm and Haar Cascade Classifier | by Mrinal Tyagi | Towards Data Science

[4]. Phạm Hồng Ngự. Nhận dạng đối tượng sử dụng thuật toán AdaBoost, HOT. Đề tài: Nhận dạng đối tượng sử dụng thuật toán AdaBoost, HOT | PDF (slideshare.net)

[5]. Ben Mauss. Haar-like Features: Seeing in Black and White. Haar-like Features: Seeing in Black and White | by BenMauss | Level Up Coding (gitconnected.com)

[6]. Rohan Chaudhury. Adaboost classifier for face detection using viola jones algorithm. Adaboost classifier for face detection using viola jones algorithm | by Rohan Chaudhury | Medium

[7]. Vincent T. The Viola-Jones Face Detection Algorithm. The Viola-Jones Face Detection Algorithm | by Vincent T. | 0xCODE | Medium

[8]. A Convolutional Neural Network Cascade for Face Detection. A convolutional neural network cascade for face detection | IEEE Conference Publication | IEEE Xplore

[9]. Joint face detection and alignment using multi-task cascaded convolutional networks. [1604.02878] Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks (arxiv.org)

[10].