Lab Manual - DD2427 Image Based Classification and Recognition

Babak Rasolzadeh, Josephine Sullivan

May 2, 2013

Face Detection

Real-time face detection in multi-scale images with an attentional *cascade of boosted classifiers.*

In this project you will explore the machine learning method called Adaboost by implementing it for the computer vision task of real-time face detection in images. Real-time performance is achieved by exploiting a so called *attentional cascade*. The final classifier/detector should be capable of detecting upright frontal faces observed in reasonable lighting conditions.

Face detection is an important problem in image processing. It could potentially be the first step in many applications – marking areas of interest for better quality encoding for television broadcasting, content-based representation (in MPEG-4), face recognition, face tracking and gender recognition. In fact for this latter task computer-based algorithms out-perform humans.

During the past decade, many methods and techniques have been gradually developed and applied to solve the problem. These include vector quantization with multiple codebooks, face templates and Principal Component Analysis (PCA). The latter technique is directly related to Eigenfaces and Fisherfaces. Here we will develop a face detection system based on the well-known work of Paul Viola and Michael Jones Viola and Jones [2001a]. This basically involves the interpretation of Haar-like features in a boosted cascade, see paper on the course homepage.

Theory and Background

There is a loosely defined difference between the process of face detection and that of face recognition. The former assumes that each *instance* of a face belongs to a more general class of objects called *faces*. Thus, the process

of face recognition is a more specific process where you try to identify which specific instance of a face you are looking at. Of course this is a sliding scale since one could go on and say that a specific face, e.g. the face of Bill Gates, can have different instances as well (i.e. happy, sad, tired etc.). That would be an even more specific recognition process. In other words, the difference between detection and recognition depends strongly on what class levels one has defined.
However, for the sake of the argument that there is indeed a difference between the two, one could say that at the detection level one needs to identify a set of generalized features that apply for the class we are trying to identify. The localization of such features can be accomplished by a number of common methods. There are basically four different approaches to the problem of face detection:

1. **Knowledge-based methods**: Rules are encoded based on the human knowledge of the defining features of a human face. A majority of these rules capture the relationship between features Tang et al. [1999]; Peer and Solina [1999].

2. **Feature invariant methods**: Algorithms designed to find structural features of a face that are invariant to the common problems of pose, occlusion, expression, image conditions and rotation Viola and Jones [2001b]; Lienhart and Maydt [2002]; Menser [1999].

3. **Template matching methods**: Given a sample set a corresponding standard facial pattern set is produced. The relation between the sample image and the defined pattern set is computed and used to provide inference Saber and Tekalp [1998]; Shin et al. [2000].

4. **Appearance-based methods**: Similar to template matching methods. The goal here is to achieve higher accuracy through larger variation in training data, since one uses statistics without any prior model assumptions Rowley et al. [1998]; Roth et al. [2000].

In this project, the focus will be on a specific appearance-based method, namely the Viola-Jones face detector. This technique relies on the use of simple Haar-like features that are evaluated quickly through the use of a new image representation.

Over-complete Haar-like features with boosted cascade has proven to be an effective approach to visual object detection, capable of processing images extremely rapidly and achieving high detection rates with very low false alarms. The effectiveness of this method comes from four key contributions. The first one is a set of simple masks which are similar to Haar filters. The second part is an image representation called *Integral image* which allows

these features to be computed very quickly. The third contribution is a learning algorithm, based on *Adaboost* which selects a small number of features from a large set and yields extremely effective classifier. The last one is a method for combining increasingly more complex classifier in a *Cascade structure* which allow background region of an image to be quickly discarded while spending more computation on promising object-like regions. This is sometimes referred to as an *attentional cascade* since it spends more computational effort on the more plausible target regions.

The lab project

Getting started

Your task is to code up, from scratch, a face detector. This set of notes guides you through the solution you should initially follow. Fairly explicit instructions are given. To ensure the code you've written is correct, there are many debugging checks set up along the way. Please use them. If you want to follow your own design I suggest, first implement the solution described here and then incrementally change things to your own design. In fact, this is expected if you hope to get a high grade for the lab. It is important to note that the implementation described here is computationally quite efficient, however, due to the just-in-time compilation of *Matlab* it is slow to run on lots of data. But it is an easily understood implementation and lends itself to easy debugging. Also it is not hard to convert the implementation to a more *Matlab* efficient one and this document will explain how to do this once you have a version up and running.

Material to download

Training images: To get started download TrainingImages.tar.gz, the database of face and non-face images, from the course website. Under the directory `TrainingImages` you will find the images of face (`FACES`) and non-face (`NFACES`) images.

Test images: When you have a face detector up and running, you will want to use it to detect faces of all sizes in large images. Initially, you will perform these tasks on the set of images contained in TestImages.tar.gz; available at the course website.

Debugging information: Throughout this lab you will verify the correctness of your code by checking your output against previously computed results DebugInfo.tar.gz; once again available at the course website.

Task I - Integral image and feature computation

Initially, you will write the functions to compute the Haar-like features on which the face detector is based, see figure 2. Let $I(x, y)$ represent the intensity of a pixel at (x, y) and $B(x, y, w, h)$ the sum of the pixel intensities in a rectangular region. This rectangular region has width w and height h

pixels and (x, y) are the coordinates of its top left coordinate. So formally

$$B(x, y, w, h) = \sum_{x'=x}^{x+w-1} \sum_{y'=y}^{y+h-1} I(x', y') \tag{1}$$

Each Haar-like feature is formed from adding and subtracting sums from

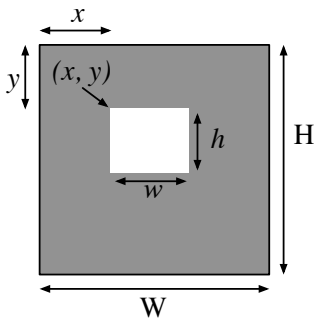
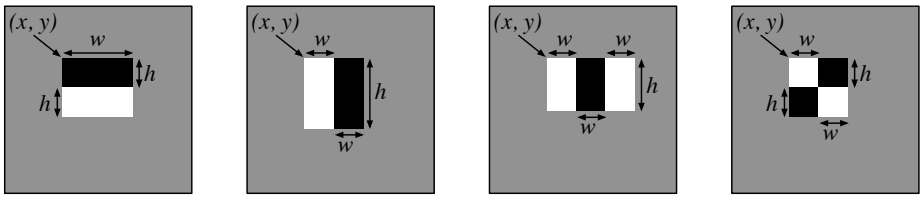


Figure 1: The features used are constructed from the sum of pixel intensities within rectangular regions. In this work a rectangular region is parametrized by the coordinates of its top left hand corner and its width and height.

different rectangular regions. So for instance features of type I and type II respectively have the form

$$B(x, y, w, h) - B(x, y + h, w, h) \quad \text{and} \quad B(x + w, y, w, h) - B(x, y, w, h)$$

These Haar-like features can be computed quickly from the image's integral



(a) **Type I** (b)**Type II** (c) **Type III** (d) **Type IV**

Figure 2: The four type of features used in the *Viola &Jones* system and consequently this lab. The figure shows how they are parametrized by the four postive integers (x, y, w, h). The sum of pixels in white rectangles are subtracted from those in the black rectangles.

image. To refresh your memory, the integral image is defined as

$$I'(x, y) = \sum_{y'=1}^{y} \sum_{x'=1}^{x} I(x', y') \tag{2}$$

Without further ado then let's get started with the lab. As a word of advice, you should create a separate directory (subdirectories) to contain your code.

2.1 Initial image processing

To begin you will write a function `LoadImage.m`. It will take as input the filename of an image and return two arrays. The first corresponds to a normalized version of the pixel data of the image and the second to its integral image. This function will contain three parts.

Program 1: `function [im, ii_im] = LoadIm(im_fname)`

Read in image: The `Matlab` function `imread` can be used to do this. *Remember to cast the loaded data to* `double`.

Image normalization: You will want your face detector to have some invariance to illumination changes. Thus you have to normalize the pixel values by applying this transformation to your image:

$$I(x,y) = \frac{I(x,y) - \mu}{\sigma} \tag{3}$$

where μ is the average intensity value of the image and σ is the standard deviation of the pixel intensity values. (`Matlab` functions `mean` and `std` can compute these values.) *Note you will run into problems if σ equals zero. In this case you can either decide not to divide by σ or add a small value to σ.*

Compute the integral image: This can be done efficiently using the `Matlab` function `cumsum`.

Sanity Check:
Is data in `im` normalized? Check that the average intensity of `im` is 0 and that its standard deviation is 1.

Does `ii_im` contain the correct values? For instance check that `ii_im(y, x)` equals `sum(sum(im(1:y, 1:x)))` for different values of `x` and `y`.

Debug Point: Run your function `LoadImage.m` on the the image `face00001.bmp` from the `FACES` directory. The matrices you calculate should match those in `DebugInfo/debuginfo1.mat`. Check this with

```
>> dinfo1 = load('DebugInfo/debuginfo1.mat');
>> eps = 1e-6;
>> s1 = sum(abs(dinfo1.im(:)  - im(:))  > eps)
>> s2 = sum(abs(dinfo1.ii_im(:)  - ii_im(:))  > eps)
```

If everything is correct then `s1` and `s2` should both be equal to zero. When you display `im` and `ii_im`, they should look as in figure 3.

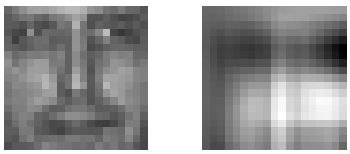


image **integral image**

Figure 3: The original image and its integral image when computed from the original image normalized.

2.2 Computation of the Haar-like features

You now have the code to load an image, normalize it and compute its integral image. The next stage is to *efficiently* compute the Haar-like features. The steps required to write this code are presented here.

2.2.1 Sum of pixel values within a rectangular region

The first step is to write a function `ComputeBoxSum.m` that computes the sum of the pixel values within a rectangular area using the integral image. A rectangular region is defined by 4 numbers `(x, y, w, h)` as shown in figure 1. Note this function is a short one.

Program 2: `function A = ComputeBoxSum(ii_im, x, y, w, h)`

Use the integral image, `ii_im`, to compute the sum of the pixel values in the original `im` in the rectangular region defined by `(x, y, w, h)`, as in equation (1).

Sanity Check:
Compute $B(x, y, w, h)$ using `sum(sum(im(y:y+h-1, x:x+w-1)))`. Check your function `ComputeBoxSum(ii_im, x, y, w, h)` produces the same output.

The function `ComputeBoxSum.m` forms the basis for the evaluation of the four features types used in this lab. And these are what you will now compute.

2.2.2 Feature computations

Each feature is defined by its type and four numbers `(x, y, w, h)`. `(x,y)` is the coordinate of the upper left corner of the feature. `w` is the width and `h` is the height of the sub rectangular regions from which the feature is constructed. So in this case type I features have total width `w` and total height 2`h` while type IV features have total width 2`w` and total height 2`h`, see figure 2. Your task is to write the four different functions to compute these four different features which take the integral image and `(x, y, w, h)` as input.

Program 3: `function f = FeatureTypeI(ii_im, x, y, w, h)`

Use the integral image and the function `ComputeBoxSum` to compute

$$F_1(x, y, w, h) = B(x, y, w, h) - B(x, y + h, w, h)$$

Sanity Check:
Compute $F_1(x, y, w, h)$ from `im` using the function `sum` and check your function `FeatureTypeI(ii_im, x, y, w, h)` produces the same results.

Program 4: `function f = FeatureTypeII(ii_im, x, y, w, h)`

Use the integral image and the function `ComputeBoxSum` to compute $F_2(x, y, w, h)$.

Sanity Check:
Compute $F_2(x, y, w, h)$ from `im` using the function `sum` and check your function `FeatureTypeII(ii_im, x, y, w, h)` produces the same results.

Program 5: `function f = FeatureTypeIII(ii_im, x, y, w, h)`

Use the integral image and the function `ComputeBoxSum` to compute $F_3(x, y, w, h)$.

Sanity Check:
Compute $F_3(x, y, w, h)$ from `im` using the function `sum` and check your function `FeatureTypeIII(ii_im, x, y, w, h)` produces the same results.

Program 6: `function f = FeatureTypeIV(ii_im, x, y, w, h)`

Use the integral image and the function `ComputeBoxSum` to compute $F_4(x, y, w, h)$.

Sanity Check:
Compute $F_4(x, y, w, h)$ from `im` using the function `sum` and check your function `FeatureTypeIV(ii_im, x, y, w, h)` produces the same results.

Debug Point: Using the integral image, `ii_im`, computed from the image `face00001.bmp`, check your newly written functions with the following code. Note that you are checking the ouput of your function with values previously calculated.

```
>> dinfo2 = load('DebugInfo/debuginfo2.mat');
>> x = dinfo2.x; y = dinfo2.y; w = dinfo2.w; h = dinfo2.h;
>> abs(dinfo2.f1 - FeatureTypeI(ii_im, x, y, w, h)) > eps
>> abs(dinfo2.f2 - FeatureTypeII(ii_im, x, y, w, h)) > eps
>> abs(dinfo2.f3 - FeatureTypeIII(ii_im, x, y, w, h)) > eps
>> abs(dinfo2.f4 - FeatureTypeIV(ii_im, x, y, w, h)) > eps
```

2.3 Enumerate all features

At this point you have written code to calculate each feature type of a certain size and at a certain position. Now you have to enumerate all the different possible position and sizes of a feature type that can be computed within the `19×19` image patch. The latter sentence means the support of the entire feature must be included entirely within the image. Thus, for example, features of type II can have starting positions and sizes enumerated by:

```
for w = 1:floor(W/2)-2
 for h = 1:H-2
  for x = 2:W-2*w
   for y = 2:H-h
    ...........
```

Figure 4 displays a small subset of all the possible type II features. Now write a function `EnumAllFeatures.m` which enumerates all the features given the width `W` and height `H` of the image.

Program 7: `function all_ftypes = EnumAllFeatures(W, H)`

Write a function that enumerates all the features. Keep a record of these features in the matrix `all_ftypes`. `all_ftypes` will have size `nf`$\times$`5` where `nf` is the number of features. Each row is a description of the feature and has the form `(type, x, y, w, h)` where `type` is either `1, 2, 3, 4` corresponding to the feature type. While the rest of the numbers are the starting position and size of the feature. **Tip** allocate the memory for `all_ftypes` at the start. Note you can declare an array that has too many rows and trim it at the end when you know the exact number of features you have declared.

Sanity Check:
Check the limits of the `for` loops used to define all the different features and that all the features have support within the `19`$\times$`19` image. Do this by checking that for every feature defined `x+w-1` $\leq$ `W` and `y+h-1` $\leq$ `H`. `nf` should have a value around `32,746`.

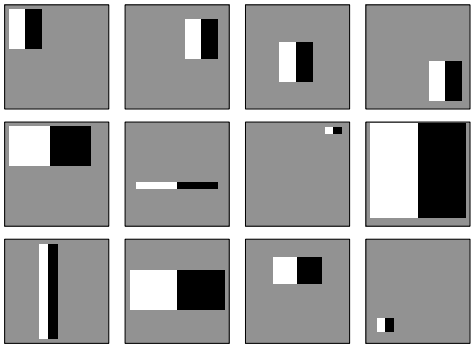


Figure 4: A small subset of different possible features of type II.

Program 8: `function fs = ComputeFeature(ii_ims, ftype)`

The inputs of this function are a cell array `ii_ims` and the parameters of a feature `ftype`. Each element of `ii_ims` is an integral image. Now write the code to extract the feature defined by `ftype` from each integral image defined in `ii_ims`. The output is stored in an array `fs` of size `1`$\times$`length(ii_ims)`.

Debug Point: Load and compute the integral image for the first 100 images in the directory `'FACES'`, (i.e. `face00001.bmp`, ... , `face00100.bmp`). Store the integral images in a cell array `ii_ims`. Check your newly written function produces the same output as a version we have written.

```
>> dinfo3 = load('DebugInfo/debuginfo3.mat');
>> ftype = dinfo3.ftype;
>> sum(abs(dinfo3.fs - ComputeFeature(ii_ims, ftype)) > eps)
```

2.4 Vectorize your code

You have now written debugged code that computes all the features. However, in its current form the code is not very efficient computationally (w.r.t. *Matlab*). Therefore, in this subsection you will write code to vectorize the operations. You are aiming to perform the feature computation as a matrix multiplication. That is if given an integral image `ii_im` then the computation of a feature can be written as

```
ii_im(:)'  * ftype_vec
```

where `ftype_vec` is a column vector. To achieve this first write a function `VecBoxSum.m` that computes the correct `b_vec` to compute the sum of the pixel intensities in a rectangle defined by `(x, y, w, h)` in the form just described.

Program 9: `function b_vec = VecBoxSum(x, y, w, h, W, H)`

If `W` and `H` are the dimensions of the integral image then this function returns the column vector `b_vec` which will be zeros except for 4 elements such that

`ii_im(:)'*b_vec` equals `ComputeBoxSum(ii_im, x, y, w, h)`

Sanity Check:
Given the integral image `ii_im` of `face00001.bmp`, calculate

```
>> b_vec = VecBoxSum(x, y, w, h, 19, 19);
>> A1 = ii_im(:)'  * b_vec
>> A2 = ComputeBoxSum(ii_im, x, y, w, h)
```

for differing values of `x,y,w` and `h`. Check each `A1` equals `A2`.

Given the ability to compute the correct `b_vec` to apply to an integral image to compute the sum of the pixels in rectangular image, it is then easy to compute the `b_vec` needed to be applied to an integral image to compute features defined by `ftype`. This is what you should do in the function `VecFeature.m` that you will write next.

Program 10: `function ftype_vec = VecFeature(ftype, W, H)`

In this function, calculate the column vector required to compute the response for a feature defined by `ftype`. So for instance features of type I can be calculated with

```
ftype_vec=VecBoxSum(x, y, w, h, W, H)-VecBoxSum(x, y+h, w,
h, W, H)
```

Notice how this is analagous to the functions you wrote in previous subsections. Note that you can use the *Matlab* `switch` function to check which feature type is defined by `ftype(1)`.

Sanity Check:
Given the integral image `ii_im` of `face00001.bmp`, calculate

```
>> ftype_vec = VecFeature([1, x, y, w, h], 19, 19);
>> A1 = ii_im(:)'  * ftype_vec
>> A2 = FeatureTypeI(ii_im, x, y, w, h)
```

for differing values of `x,y,w` and `h` and check `A1` equals `A2`. Try similar calculations for the different feature types 2, 3 and 4.

You can now generate `ftype_vec` for any feature type. The next task is to compute a feature matrix `fmat`. The columns of this matrix are the `ftype_vec` for each feature defined by `all_ftypes`. Write the function `VecAllFeatures.m` to compute this.

Program 11: `function fmat = VecAllFeatures(all_ftypes, W, H)`

This function will generate the column vectors used to generate each feature defined in `all_ftypes`. It will return an array `fmat` of size `W*H` $\times$ `nf` where each column corresponds to a feature. Note all you need to do is call `VecFeature` the appropriate number of times and store the output in `fmat`.

Your final task in this section is to replicate the function `ComputeFeature.m` with a new function `VecComputeFeature.m`. This time round the inputs are an array `ii_ims` containing the integral images you want to process and `ftype_vec` the column vector defining your feature.

Program 12: `function fs = VecComputeFeature(ii_ims, ftype_vec)`

The inputs of this function are an array `ii_ims` of size `ni` $\times$ `W*H` and `ftype_vec` is a column vector containing the description of a feature. Each row of `ii_ims` is an integral image. Now write the code to extract the feature defined by `ftype_vec` from each integral image defined in `ii_ims`. The output is stored in an array `fs` of size `size(ii_ims, 1)`$\times$`1`.

Sanity Check:
Given the integral image `ii_im` of the first 100 face images `face00001.bmp`, `face00002.bmp`, ... and check that if you calculate

```
>> fs1 = VecComputeFeature(ii_ims, fmat(:, 1));
>> fs2 = ComputeFeature(ii_ims1, all_ftypes(1, :));
```

then `fs1` and `fs2` are equal. Note that `ii_ims1` contains the same data as `ii_ims` but just in a cell array and the images are stored as 2d arrays. Try similar calculations for different features.

Note you will probably not use this function again as it is only one line of code, it will be better to directly write this line into any function.

2.5 Extract features and training data

A list of `ni` numbers randomly chosen images in the directory `'FACES'` can be obtained with the following code:

```
face_fnames = dir('FACES');
aa = 3:length(face_fnames);
a = randperm(length(aa));
fnums = aa(a(1:ni));
```

These numbers correspond to the images you will use for training so for instance the first image will be:

```
im_fname = ['FACES/', face_fnames(fnums(1)).name];
```

The next function you will write is to load all the images you will use for training and also to compute `fmat` for the features you have defined.

Program 13: `function LoadSaveImData(dirname, ni, im_sfn)`

Choose images: Randomly pick `ni` images in the directory `dirname` as described in the text.

Load data: For each image use the function `LoadIm` to load it and compute its integral image. Then store each integral image as a row in an array called `ii_ims`.

Save the image data: Save the details of the training image data in the file `im_sfn`:

```
save(im_sfn, 'dirname', 'fnums', 'ii_ims');
```

Sanity Check:
Run your newly written function on the `'FACES'` directory with `ni = 100`. Check that the code runs smoothly.

Program 14: `function ComputeSaveFData(all_ftypes, f_sfn)`

Compute feature vectors: Use `VecAllFeatures` to compute `fmat` from `all_ftypes` and set the values of `W` and `H` to `19`.

Save the feature info: Save the details of the features you will use in the file `f_sfn`:

```
save(f_sfn, 'fmat', 'all_ftypes', 'W', 'H');
```

Sanity Check:
Check that the code runs smoothly.

Debug Point: Set `dirname` to the `FACES` directory. Read in the file `debuginfo4.mat` and follow the instructions below to check the output of your newly written functions.

```
>> dinfo4 = load('DebugInfo/debuginfo4.mat');
>> ni = dinfo4.ni;
>> all_ftypes = dinfo4.all_ftypes;
>> im_sfn = 'FaceData.mat';
>> f_sfn = 'FeaturesToMat.mat';
>> rng(dinfo4.jseed);
>> LoadSaveImData(dirname, ni, im_sfn);
>> ComputeSaveFData(all_ftypes, f_sfn);
```

Then check that `dinfo4.fmat` equals the `fmat` you calculated and similarly for `dinfo4.ii_ims`.

You are now ready to write the final function in this section. Basically, in this function you compute and save the training data extracted from both the face images and the non-face images.

Program 15: `function GetTrainingData(all_ftypes, np, nn)`

Write a function which calls `LoadSaveImData` twice - once for data extracted from the face images and the second time round for the non-face images. `np` is the number of face images used and `nn` is the number of non-face images used. In the two calls to `LoadSaveImData`, set `im_sfn` to `FaceData.mat` and `NonFaceData.mat` respectively. The function should also call `ComputeSaveFData` and set `f_sfn` to `FeaturesToUse.mat`.

Now to create the data required for training the face detector run the following code:

```
>> dinfo5 = load('DebugInfo/debuginfo5.mat');
>> np = dinfo5.np;
>> nn = dinfo5.nn;
>> all_ftypes = dinfo5.all_ftypes;
>> rng(dinfo5.jseed);
>> GetTrainingData(all_ftypes, np, nn);
```

Note that it may take upto several minutes for the function `GetTrainingData` to complete, depending on the speed of your machine and the effeciency of your code. Once it has completed then load your saved files into *Matlab* with the following commands:

```
>> Fdata = load('FaceData.mat');
>> NFdata = load('NonFaceData.mat');
>> FTdata = load('FeaturesToUse.mat');
```

`Fdata`, `NFdata` and `FTdata` are structures and contain the data you have just saved. So, for instance, the name of the directory containing the face training images is accessed with `Fdata.dirname`, while the integral image data is accessed with `Fdata.ii_ims`.

If you have successfully reached this point then you are ready to get this part of the lab, **Check I**, signed off by one of the Teaching Assistants (TA). See the accompanying form to see what the TA will expect you to demonstrate and answer questions on.

Task II - Boosting to learn a strong classifier

Boosting is a process of forming a strong strong classifier through the linear combination of weak ones. In the context of Viola-Jones face detection, a binary classification task, the weak classifiers are derived from the extracted set of features.

The details of the *AdaBoost* algorithm are given in algorithm 1. The core idea behind the use of AdaBoost is the application of a weight distribution to the training examples and the modification of the distribution during each iteration of the algorithm. At the beginning the weight distribution is flat, but after each iteration of the algorithm each of the weak classifiers returns a classification on each of the sample-images. If the classification is correct the weight on that image is reduced (seen as an easier sample), otherwise there is no change to its weight. Therefore, weak classifiers that manage to classify difficult sample-images (i.e. with high weights) are given higher weighting in the final strong classifier.

Now let's go and implement the *AdaBoost* algorithm to build a face detector.

3.1 Defining & learning a weak classifier

At this moment you have extracted many Haar-like features from many training images. How can these simple features be used to build weak classifiers from which we will build the strong classifier ? We choose the weak classifiers to have a very simple form. Let $\mathbf{f} = (f^1, \ldots, f^N)$ be the vector of feature values extracted from one image. Then a weak classifier $h(\cdot)$ with parameters $\mathbf{\Theta} = (j, p, \theta)$ can be defined as

$$h(\mathbf{f}\,;\,\mathbf{\Theta}) = h(\mathbf{f}\,;\,j, p, \theta) = \begin{cases} 1 & \text{if } p\, f^j < p\, \theta \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

where f^j is the jth coordinate of $\mathbf{f}$. This is the type of weak classifier you will use in the lab. However, you are free to define another form of weak classifier when you make changes to the default detector. In this subsection you will write code to automatically set the parameters p, θ associated with the weak classifiers of this form when there is a weight associated with misclassifying each training algorithm. Algorithm 2 describes a very simple way to do this.

Before finding the parameters of a weak classifier we will ensure that you understand what we is meant when referring to feature responses. Remember, the structure `FTdata` contains the matrix `fmat`. Each column of this matrix corresponds to a column vector which when multiplied with the integral image of an image (represented as a row vector) produces the value of applying a Haar-like feature to the original image. While the structures

Algorithm 1 AdaBoost

Input: A set of feature vectors $\{\mathbf{f}_1, \ldots, \mathbf{f}_n\}$ extracted from example images and associated labels $\{y_1, \ldots, y_n\}$ where $y_i \in \{0, 1\}$. $y_i = 0$ denotes a negative example and $y_i = 1$ a positive one. m is the number of negative examples. A postive integer T which defines the number of weak classifiers used in the final strong classifier.

Output: A set of parameters $\{\boldsymbol{\Theta}_1, \ldots, \boldsymbol{\Theta}_T\}$ associated with the weak classifier $h(\cdot)$ and a set of weights $\alpha_1, \ldots, \alpha_T$ which define a stong classifier of the form:

$$H(\mathbf{f}) = \begin{cases} 1 & \text{if } \left(\sum_{t=1}^{T} \alpha_t \, h(\mathbf{f}; \boldsymbol{\Theta}_t)\right) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Steps of Algorithm:

Initialize the n weights to:

$$w_{1,i} = \begin{cases} (2m)^{-1} & \text{if } y_i = 0 \\ (2(n-m))^{-1} & \text{otherwise} \end{cases} \tag{5}$$

for $t = 1, \ldots, T$ **do**

- Normalize the weights so they sum to one: $w_{t,i} = \frac{w_{t,i}}{\sum_j w_{t,j}}$.
- For each feature j - the jth coordinate of the feature vectors $\mathbf{f}_i$ - train a weak classifier h restricted to using this feature that tries to minimize the error

$$\epsilon_j = \sum_i w_{t,i} \, | \, h(\mathbf{f}_i; j, \boldsymbol{\theta}_j) - y_i \, | \tag{6}$$

- Choose the weak classifier with the lowest error: $j^* = \arg\min_j \epsilon_j$
- Set $\boldsymbol{\Theta}_t = (j^*, \boldsymbol{\theta}_{j^*})$ and $\epsilon_t = \epsilon_{j^*}$.
- Update the weights:

$$w_{t+1,i} = w_{t,i} \, \beta_t^{1-|\, h(\mathbf{f}_i; \boldsymbol{\Theta}_t) - y_i \,|}, \quad \text{with } \beta_t = \frac{\epsilon_t}{1 - \epsilon_t}. \tag{7}$$

- Set $\alpha_t = \log \frac{1}{\beta_t}$.

end for

`Fdata` and `NFdata` contain the integral images extracted from the face and non-face training images. Using these integral images and one column of `fmat`, say `fmat(:, 12028)`, one can compute the feature responses for all the images for this feature type with a simple matrix multiplication - remember `VecComputeFeatures.m`. Using this data create a set of responses `fs` for the positive and for the negative examples. Next use `hist` to compute the histogram of the feature responses from the face images and from the non-face images. Display the histograms on the same figure. You should plot curves that look like those in figure 5.

Now your task is to write the function `LearnWeakClassifier` that implements algorithm 2. It takes as input the vector of weights associated with each training image, a vector containing the value of a particular feature extracted from each training image and a vector of the labels associated with each training image. The output are then the learnt parameters of the weak classifier and its associated error.

Program 16: `function [theta, p, err] = LearnWeakClassifier(ws, fs, ys)`

Compute the threshold and parity as described in algorithm 2.

Sanity Check:
As stated before the structure `FTdata` contains the feature array `fmat` while the structures `Fdata` and `NFdata` contain the integral images extracted from the face and non-face training images. Using these integral images and one column of `fmat`, say `fmat(:, 12028)`, compute the feature responses for one `ftype`. Use this data to create `fs` and `ys`. Then set the weights `ws` as they are initialized in algorithm 2. Using this input run your newly written function to compute `theta` and `p`. You should get values similar to `theta = -3.6453` and `p = 1`.

Next use `hist` to compute the histogram of the feature responses from the face images and from the non-face images. Display the histograms on the same figure as well as the line $x = \theta$, see figure 5. You can repeat this process for different features and check that your function produces sensible results.

3.2 Display functions

Before proceeding to write a program to implement the boosting algorithm, you will write a couple of functions used for display purposes. These will be extremely useful when debugging your boosting implementation and interpreting its output. The first function is to make an image representing a feature, as in figure 2, defined by the vector `ftype`.

Algorithm 2 Simple weak classifier

Input: A set of feature vectors $\{\mathbf{f}_1, \ldots, \mathbf{f}_n\}$ extracted from training images and associated labels $\{y_1, \ldots, y_n\}$ where $y_i \in \{0, 1\}$. An integer j indicating which feature - coordinate of the $\mathbf{f}_i$'s - to base the classifier on and a set of non-negative weights $\{w_1, \ldots, w_n\}$ associated with each feature vector (image) that sum to one.

Output: $\boldsymbol{\theta} = (p, \theta)$ and $\epsilon > 0$. θ is a threshold value and $p \in \{-1, 1\}$ is a parity value. Together they define a weak classifier of the form:

$$h(\mathbf{f}; j, p, \theta) = \begin{cases} 1 & \text{if } p\, f^j < p\, \theta \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

where f^j is the jth coordinate of $\mathbf{f}$. ϵ is the value of the error associated with this classifier when applied to the training data.

Steps of Algorithm:

- Compute the weighted mean of the postive examples and negative examples

$$\mu_{\mathrm{P}} = \frac{\sum_{i=1}^n w_i\, f_i^j\, y_i}{\sum_{i=1}^n w_i\, y_i}, \quad \mu_{\mathrm{N}} = \frac{\sum_{i=1}^n w_i\, f_i^j\, (1 - y_i)}{\sum_{i=1}^n w_i\, (1 - y_i)} \tag{10}$$

- Set the threshold to $\theta = \frac{1}{2}(\mu_{\mathrm{P}} + \mu_{\mathrm{N}})$.

- Compute the error associated with the two possible values of the parity

$$\epsilon_{-1} = \sum_{i=1}^n w_i\, |y_i - h(\mathbf{f}_i; j, -1, \theta)|, \quad \epsilon_1 = \sum_{i=1}^n w_i\, |y_i - h(\mathbf{f}_i; j, 1, \theta)| \tag{11}$$

- Set $p^* = \arg \min_{p \in \{-1,1\}} \epsilon_p$ and then $\epsilon = \epsilon_{p^*}$.

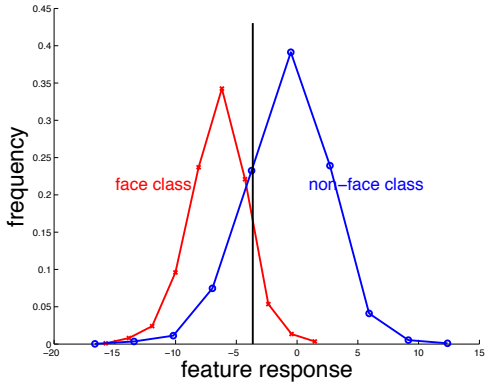


Figure 5: **A simple weak classifier.** The blue curve is the histogram of the feature responses for the non-face training images. The red curve those for the face images. The black line is the threshold value chosen using algorithm 2. Marked on the figure are the regions, defined by the threshold, where feature responses will be classified as face /non-face.

Program 17: `function fpic = MakeFeaturePic(ftype, W, H)`

Create a matrix, `fpic`, of zeros of size `(H, W)`. From the information in `ftype`, set the appropriate pixels to `1` and to `-1`.

Sanity Check:
Run

```
fpic = MakeFeaturePic([4, 5, 5, 5, 5], 19, 19);
```

and then display `fpic` via `imagesc`. The feature should appear as in figure 6(a).

The strong classifier, though, consists of a weighted sum of the weak classifiers. Thus the second display function you have to write takes as input the array defining each feature, a vector `chosen_f` of postive integers that correspond to the features used in the classifier and the weights `alphas` associated with each feature/weak classifier.

Program 18: `function cpic = MakeClassifierPic(all_ftypes, chosen_f, alphas, ps, W, H)`

Create a matrix, `cpic`, of zeros of size `(H, W)`. For each feature in `chosen_f` create its picture via

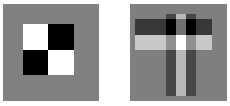
```
MakeFeaturePic(all_ftypes(chosen_f(i), :), W, H);
```

Then set `cpic` as a weighted sum of these newly create pictures. The weights are equal to `alphas .* ps`.

Sanity Check:
Run

```
cpic = MakeClassifierPic(all_ftypes, [5192, 12765], [1.8725,1.467],
                         [1,-1]);
```

and then display `cpic` via `imagesc`. You may have to compensate for potential negative numbers in `cpic`. The image representation of the classifier should appear as in figure 6(b).



(a) (b)

Figure 6: Example output of the feature and classifier display.

3.3 Implement the Boosting algorithm

You are now ready to write the code to implement the *AdaBoost* algorithm to produce a face detector. Before we start we introduce the concept of a structure (`struct`) in *Matlab* as the function `BoostingAlg` returns a structure, `Cparams`, containing the parameters of the strong classifier and those of the associated weak classifiers. The structure `Cparams` contains the following fields:

```
Cparams.alphas, Cparams.Thetas, Cparams.fmat, Cparams.all_ftypes
```

where `alphas` represents the α_t's in equation (4) and `Thetas` represents the $\boldsymbol{\Theta}_t$'s in algorithm 1 which are the parameters of the weak classifiers. Thus `alphas` is vector of length `T` and `Thetas` is an array of size `T`$\times$`3` where the first column represents the features chosen, the second column the thresholds of the weak classifiers and the third column the associated parities. The other fields have already been introduced.

Program 19: `function Cparams = BoostingAlg(Fdata, NFdata, FTdata, T)`

Implement the boosting algorithm as described in algorithm 1. The inputs to this function are the training data obtained from the positive and negative images and the number of weak classifiers T to include in the final strong classifier. The output is then the structure representing the final classifier. Remember during training you have to learn the parameters for each weak classifier (which takes the weight of each training example into account) and then choose the one with lowest error. So you have to use the array of the integral images and the appropriate column of `fmat` to generate the feature responses for each feature. This whole process is repeated `T` times.

Sanity Check:
While debugging and writing this function only use a fraction of the features defined in `FTdata.fmat` as otherwise things will run very slowly. I suggest just use the first `1000` features defined in `FTdata.fmat` to begin with and run the command

```
Cparams = BoostingAlg(Fdata, NFdata, FTdata, 3);
```

Then use the function `MakeFeaturePic` to display the 3 different features selected and `MakeClassifierPic` to display the learned classifier. I got the results in figure 7.

Once you have written this command and think you have passed the sanity check then you should do a more exact check. Remember this is just using the first `1000` features defined in `FTdata.fmat`.

Debug Point: To check the output of your code, run the following commands

```
>> dinfo6 = load('DebugInfo/debuginfo6.mat');
>> T = dinfo6.T;
>> Cparams = BoostingAlg(Fdata, NFdata, FTdata, T);
>> sum(abs(dinfo6.alphas - Cparams.alphas)>eps)
>> sum(abs(dinfo6.Thetas(:)  - Cparams.Thetas(:))>eps)
```

If you have successfully passed this latest check then update `BoostingAlg` to use all the features defined in `FTdata.fmat`. Before you run this function on all this data try and optimize your code slightly so that it runs relatively efficiently in the inner most loop. Now just run `BoostingAlg` with `T` set to 1. The feature my code selected is shown in figure 8. If this seems to be working then run this debug check and go get yourself a cup of coffee! It may take from 10-30 minutes to run depending on your machine and the efficiency of your code.

Figure 7: **The initial 3 features chosen by boosting (left to right) and the final strong classifier**. The final strong classifier in this example consists of 3 features and is the rightmost figure. The first feature chosen by boosting is the leftmost one. These are the features chosen when you use a very small pool features for training.

```
>> dinfo7 = load('DebugInfo/debuginfo7.mat');
>> T = dinfo7.T;
>> Cparams = BoostingAlg(Fdata, NFdata, FTdata, T);
>> sum(abs(dinfo7.alphas - Cparams.alphas)>eps)
>> sum(abs(dinfo7.Thetas(:)  - Cparams.Thetas(:))>eps)
```

Once you have computed `Cparams`, save it using the command `save`.

Figure 8: **The initial features chosen by boosting (left to right) and the final strong classifier**. The final strong classifier in this example consists of 10 features and is the rightmost figure. The first feature chosen by boosting is the leftmost one.

If you have successfully reached this point then you are ready to get this part of the lab, **Check II**, signed off by one of the Teaching Assistants (TA). See the accompanying form to see what the TA will expect you to demonstrate and answer questions on.

Task III - Classifier evaluation

Congratulations you have constructed your first boosted face detector! But is it any good? In this part of the lab you will investigate how good it is. You will do this via the ROC-curve (Receiver Operator Characteristic).

However, before computing the ROC-curve you have to write a funtion that can apply your strong classifier.

Program 20: `function sc = ApplyDetector(Cparams, ii_im)`

This function applies your strong classifier to a test image of size 19×19. It takes as input the parameters of your classifier `Cparams` and the integral image, `ii_im`, computed from a normalized version of your test image. It extracts each feature used in the strong classifier from the test image and then computes a weighted sum of the weak classifier outputs. That is the function returns the score

$$\sum_{t=1}^{T} \alpha_t \, h(\mathbf{f}; \boldsymbol{\Theta}_t)$$

Sanity Check:
Run your new function on the image `face00001.bmp`. I got a score of around `9.1409`.

Now we introduce some concepts which are used in the definition of the ROC-curve. Look at table 1 to review the definitions of true-positive, false-positive etc. From these definitions the definition of *false positive rate* and

Label	Predicted Class	True Class
true-positive (tp)	Positive	Postive
false-positive (fp)	Positive	Negative
true-negative (tn)	Negative	Negative
false-negative (fn)	Negative	Positive

Table 1: A classifier predicts the class of a test example. If the true class is known then the test example can be labelled according to the above table.

true positive rate are based

$$\text{true positive rate} = \text{tpr} = \frac{n_{\text{tp}}}{n_{\text{tp}} + n_{\text{fn}}} \tag{12}$$

$$\text{false positive rate} = \text{fpr} = \frac{n_{\text{fp}}}{n_{\text{tn}} + n_{\text{fp}}} \tag{13}$$

where n_{tp} is the number of true-positives etc. The number of true-positives and false-positives will vary depending on the threshold applied to the final strong classifier. The ROC-curve is a way to summarize this variation. It is a curve that plots **fpr Vs tpr** as the threshold varies from $-\infty$ to $+\infty$. (NOTE the default `AdaBoost` threshold is designed to yield a low error rate on the training data.) From this curve you can ascertain what loss in classifier specifity you will have to endure for a required accuracy. With this

knowledge you can write the function to compute the ROC curve on the training data you didn't use when learning your classifier.

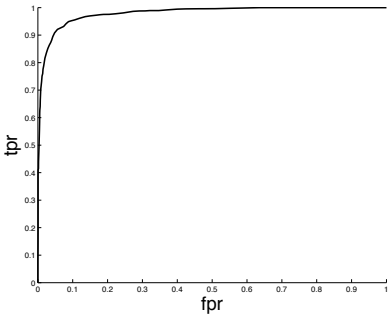


Figure 9: **ROC curve computed from the images omitted from training**. The classifier usesd consisted of 10 weak classifiers.

Program 21: `function ComputeROC(Cparams, Fdata, NFdata)`

Get test images: From `Fdata` and `NFdata` list the images that were used for training. Given this information then you can choose the face and non-face images that were not used for training as those to be used for testing. The command `setdiff` may help.

Apply detector to each test image: Run the learnt detector, using `ApplyDetector`, on each test image. Keep a record of the score of each image and its label (face/non-face).

Compute *true* and *false* positive rates: Choose a threshold to apply to the recorded scores. This predicts a labelling for each image. Check how this corresponds with the ground truth and from this compute the *true positive rate* and the *false positive rate*.

Plot the ROC curve: Let the threshold vary at fixed intervals from the minimum score value to the largest. For each threshold value keep a record of the *true positive rate* and the *false positive rate*. Then plot the *false positive rate* values Vs the *true positive rate* values.

Sanity Check:
Values of `fpr` and `tpr` for large threshold values ?

Values of `fpr` and `tpr` for small threshold values ?

Check the shape of the ROC curve. Run your function. The ROC curve you plot should look something like the one in figure 9.

Debug Point: You have created the ROC-curve for your detector. Now choose the threshold of your detector such that you get a true positive rate of above 70% on the test examples. This may seem like a low number but this is to ensure a relatively low false positive rate. This threshold value should be around 6.5. Add an extra field `Cparams.thresh` to the `Cparams` structure to retain the value of the overall threshold.

If you have successfully reached this point then you are ready to get this part of the lab, **Check III**, signed off by one of the Teaching Assistants (TA). See the accompanying form to see what the TA will expect you to demonstrate and answer questions on.

Task IV - Face detection in an image

You have now learnt a classifier via boosting that detects faces. The next step is to apply this classifier to an image and see if it detects the faces it may or may not contain. Now the function `ApplyDetector.m` will only classify faces occupying subwindows of size 19×19 whose pixel data has been normalized to have mean 0 and standard deviation 1. However, even if an image contains a face of size 19×19. You will have to try every, or almost every, possible subwindow of the image to detect the face, see figure 10. You now have to write such a function `ScanImageFixedSize.m` whose inputs are the parameters of the detector and the pixel data of the image to be processed. The output will be the parameters of the bounding boxes (sub-windows) classified as faces. This will be an array of size `nd`$\times$`4` where `nd` will be the number of face detections.

Figure 10: The sliding window of `ScanImageFixedSize.m` will traverse different locations in the large image.

It is important to note that the variance and the mean pixel intensity of the sub-window defined by (x, y, L, L) can be computed quickly using a pair of

integral images as

$$\mu = \frac{1}{L^2}\sum_{x'=x}^{x+L-1}\sum_{y'=y}^{y+L-1} I(x',y'), \quad \sigma^2 = \frac{1}{L^2-1}\left(\sum_{x'=x}^{x+L-1}\sum_{y'=y}^{y+L-1} I^2(x',y') - L^2\,\mu^2\right)$$

So the mean of the sub-window can be computed from the integral image of `im` while the sum of squared pixel values can be computed using the integral image of the image squared (i.e. `im .* im`). **Remember** if you calculate the sum of pixel intensities in an rectangular region where the pixel values have not been normalized then

$$B(x,y,w,h) = \sum_{x'=x}^{x+w-1}\sum_{y'=y}^{y+h-1} I(x',y')$$

while this sum if the pixel data has been normalized is

$$B^n(x,y,w,h) = \sum_{x'=x}^{x+w-1}\sum_{y'=y}^{y+h-1}\left[\frac{I(x',y')-\mu}{\sigma}\right] = \frac{1}{\sigma}B(x,y,w,h) - \frac{wh}{\sigma}\mu$$

We introduce the superscript n to signify a quantity has been computed from normalized data. Thus for features of type I

$$\begin{aligned} F_1^n(x,y,w,h) &= B^n(x,y,w,h) - B^n(x,y+h,w,h) \\ &= \frac{1}{\sigma}(B(x,y,w,h) - B(x,y+h,w,h)) = \frac{1}{\sigma}F_1(x,y,w,h) \end{aligned}$$

Repeat these calculations for the other feature types and write down the expression for each one. With this in mind you can adjust you features extracted from a non-normalized image very easily. You must do this or the weak classifiers you learnt cannot be applied sensibly to your image.

Program 22: `function dets = ScanImageFixedSize(Cparams, im)`

Do image processing: If necessary convert `im` to grayscale. Compute its square and compute the two necessary integral images.

Adapt `ApplyDetector.m`: We want to apply the detector to an arbitrary sub-window of size 19×19 of the large image. Remember the pixel data in the sub-window is not necessarily normalized thus you have to compensate for this fact as described in the text. Also remember that the top-left corner of the sub-window is now not necessarily at $(1, 1)$.

Search the image Write nested `for` loops to vary the top-left corner of the sub-window to be classified and keep a record of the sub-windows classified as faces in the array `dets` which has size `nd`×`4`. Each row contains the parameters of the face sub-window.

Sanity Checks:
Is the normalization correct?

Does this function replicate previous performance? If you run this function on one of the small training images you should get the same result as when you run `LoadIm` and then the original `ApplyDetector`.

5.1 Display the detection results

From the `TestImages` subdirectory load the image `one_chris.png` and scan the image for faces of size 19×19 using your newly written function.

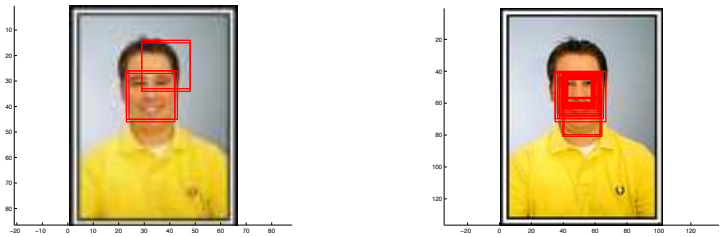
```
>> im = 'TestImages/one_chris.png';
>> dets = ScanImageFixedSize(Cparams, im);
```

Now, of course, you would like to see the output of your detector. Thus you have to write a function that takes the bounding box information contained in `dets` and displays the rectangles on top of the image. The *Matlab* function `rect` can be used for this purpose.

Program 23: `function DisplayDetections(im, dets)`

Use *Matlab*'s plotting and image display functions to show the bounding boxes of the face detections.

Sanity Check:
Running this function after `ScanImageFixedSize` on the image `one_chris.png` you should get something similar the results in figure 11 (a).



(a) **search over one scale** (b) **search over multiple scales**

Figure 11: **Results of face detection using the learnt strong classifier** The left image shows the results of the original strong classifier applied to an image. Every `19` $\times$ `19` patch is examined labelled as face or non-face. A threshold of 6.5 was used. While the right image, which is bigger than the left one, is searched over a range of scales. A threshold of 8 was applied.

5.2 Integration of multiple detections (*Optional*)

As you've probably noticed your detector is insensitive to small changes in translation and thus multiple detections occur around each face and false positive. However, you would probably like to have only one final detection per face (and per false positive). Therefore, it is useful to prune the detected sub-windows so that overlapping detections are summarized by one detection.

This can be achieved in many different ways and there is no obvious one correct solution. One solution is to partition the detections into disjoint subsets. Two detections are in the same subset if their regions intersect. Each partition yields a single final detection. The corners of the final bounding regions are the average of the corners of all detections in the set.

Program 24: `function fdets = PruneDetections(dets)`

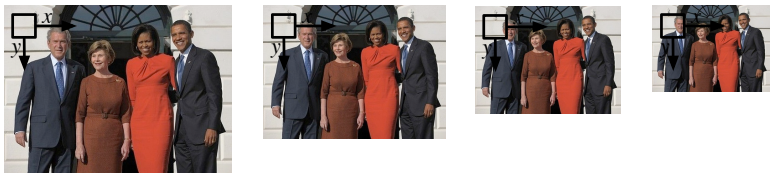
Find overlapping detections: Let `nd` be the number of detetections. Create a matrix `D` of zeros of size `nd`×`nd`. Then set `D(i,j) = 1` if the `i`th detection overlaps the `j`th detection (the function `rectint` may be of use).

Find the connected components: Use the *Matlab* function `graphconncomp` to partition the detections into those that overlap. With this information it is possible, as described in the text, to reduce overlapping detections into one detection.

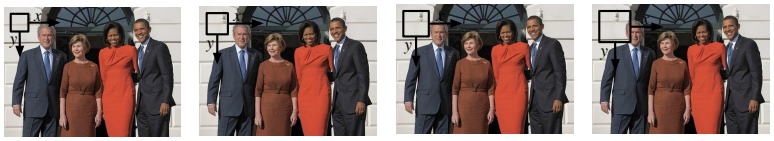
Sanity Check:
Check before and after pictures: Display the detections before you run this function and then afterwards. Visually inspect if the function has performed the expected task.

5.3 Face detection over multiple scales

Obviously, not all faces in images are of size 19×19. Thus the detector needs to be scanned across an image at multiple scales in addition to multiple locations. Scaling can be achieved in two ways. Option one is to scale the image and look for 19×19 faces in the re-scaled image, see figure 12(a). While the second option is to scale the detector, rather than scaling the image, see figure 12(b). Features for the latter detector can be computed with the same cost at any scale. Remember though, in this case, you have to normalize the feature value calculated with respect to the scale change so the learned thresholds of the weak detectors are meaningful. Write the function `ScanImageOverScale.m` which takes as input the parameters of the detector, the image, the minimum and maximum value of the scale factor and the step size defining which intermediary scale factors will be searched over. The output will be the bounding boxes corresponding to the final face detections.



(a) **Multi-scale search option one**



(b) **Multi-scale search option two**

Figure 12: **Options for performing the multi-scale search. (a)** Keep the size of the detector constant and apply it to scaled versions of the image. **(b)** Keep the size of the image constant and scale the classifiers's window.

Program 25: `function dets = ScanImageOverScale(Cparams, im, min_s, max_s, step_s)`

Implement a multi-scale search: Decide how you'll implement the multi-scale and then write the appropriate code. I would suggest you resize the image using `imresize` for each scale you check as then you can re-use the function `ScanImageFixedSize` on each of these rescaled images as the size of your classifier window remains constant. The other option would probably require more work to implement. Also remember when you find a detection, at a certain location and scale, record and save what this bounding would correspond to in the original size of the input image.

Combine detections: If necessary adapt the function `PruneDetections.m` to ensure that overlapping detections are combined into one detection.

Sanity Check:
Create images with large faces: Use the *Matlab* function `imresize` to upscale the image `one_chris.png` by a factor of 1.2. Then run your new function and check if you can still detect the face it contains.

Debug Point: Run your function `ScanImageOverScale` on the image `big_one_chris.png` and plot the detections. I used the following settings `min_s=.6`, `max_s = 1.3` and `step_s = .06`. Your results should be similar to those in figure 11(b).

If you have successfully reached this point then you are ready to get this part of the lab, **Check IV**, signed off by one of the Teaching Assistants (TA). See the accompanying form to see what the TA will expect you to

demonstrate and answer questions on.

Task V

Now you should build an accurate strong classifier. This task involves involve running the function `BoostingAlg` with `T` set to say around 100. As you know `BoostingAlg` can be slow to run. Thus before calling it with `T`≈100 you may need to optimize your code. You can you use the *Matlab* command `profile` to analyse how much time is spent by *Matlab* on each part of your code.

Some ideas for speeding up your boosting algorithm

- There is an overhead associated with function calls to user-defined functions. Thus for instance when calling `FeatureTypeI` 1,000,000 times, the majority of the time will be spent on function call overheads as opposed to the calculations executed in `FeatureTypeI`. Thus you can remove function calls and paste them into the main function. This is, in general, not good programming practice, but in the world of *Matlab*....
- You can turn `fmat` into a `sparse` matrix. Using this representation will speed up the matrix multiplication you have to perform when computing the feature responses.

Once you have built an accurate classifier the next task will be to run it on the images contained in the directory `TestImages`.

If you have successfully reached this point then you are ready to get this part of the lab, **Check V**, signed off by one of the Teaching Assistants (TA). See the accompanying form to see what the TA will expect you to demonstrate and answer questions on.

Task VI - Train and Test for Real

Now it's over to you! You now have an implementation which you have debugged pretty thoroughly. And you have passed the lab. However, you have not thrown a lot of data at the training process and it is pretty slow to run due to the just-in-time compilation of `Matlab`. You ran a detector that was trained using ≈4000 positive examples and ≈8000 negative examples using the very simple classifier described in the notes. The list of possible issues to be investigated or improved are endless. Here are some suggestions:

2,000 3,000?

- Use all the images in the database for training. If you do this, do you get better performance on images with lots of clutter?
- Continuing in the same theme, you could potentially artificially generate more training examples by perturbing the existing training examples with random rotations, scalings and translations. If you do this and then train your classifier including these *new* images, what happens to the performance? Also it is very easy to extend the set of non-face training images. Find pictures with no faces and take random `19×19` patches from these images.
- You could run your detector on the George Bush images and see how well your detector performs.
- There are many databases of faces publicly available. You could exploit these for training. It is also very easy to collect a very large of negative training examples. A common approach is to apply your face detector to lots of images that do not contain faces. Record all the false positive hits and add these patches to your negative set, then retrain your classifier using this new training data.
- A better weak classifier Gambs et al. [2007]; Rasolzadeh et al. [2006]. You could use a random forest or a decision tree as the weak classifier.
- Speed up the training process Pham and Cham [2007].
- There is a great correlation amongst the responses from similar features. Could this be potentially exploited to speed up training?
- Speeding up the training process could allow you to add extra feature types to your set of weak classifiers. You could test if widening your feature pool improves your classification rate.
- You could perhaps use some of the functions, coded in VLFeat, to compute different image patch descriptors - such as SIFT, HOG - and build weak classifiers based on these descriptors.

Please note I don't expect you to investigate all of the issues in the list or those you do at any great length. To have an *initial* investigation into one of these issues would be interesting to hear about at the poster session. At the very least run your learnt detector on a range of images containing faces and look and record its output. And, of course, this list is by no means exhaustive. You could potentially come up with your own ideas.

References

S. Gambs, B. Kégl, and E. Aïmeur. Privacy-preserving boosting. *Data Mining and Knowledge Discovery*, 14(1):131–170, 2007.

R. Lienhart and J. Maydt. An extended set of Haar-like features for rapid object detection. In *Proceedings International Conference on Image Processing*, volume 1, pages 900–903, 2002.

B. Menser. Segmentation of human faces in color images using connected operators. In *Proceedings International Conference on Image Processing*, pages III:632–636, 1999.

P. Peer and F. Solina. An automatic human face detection method. In *Proceedings of the Computer Vision Winter Workshop*, pages 122–130, 1999.

M.-T. Pham and T.-J. Cham. Fast training and selection of Haar features using statistics in boosting-based face detection. In *Proceedings of the International Conference on Computer Vision*, October 2007.

B. Rasolzadeh, L. Petersson, and N. Pettersson. Response binning: Improved weak classifiers for boosting. In *IEEE Intelligent Vehicles Symposium*, Tokyo, Japan, June 2006.

D. Roth, M.-H. Yang, and N. Ahuja. A SNoW-based face detector. In *Advances in Neural Information Processing Systems 12*, pages 855–861, 2000.

H. Rowley, S. Baluja, and T. Kanade. Neural network-based face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20:22–38, 1998.

E. Saber and A. M. Tekalp. Frontal-view face detection and facial feature extraction using color, shape and symmetry based cost functions. *Pattern Recognition Letters*, 19(8):669–680, 1998.

J.-I. Shin, H.-S. Kim, W.-S. Kang, and S.-H. Park. Face detection using template matching and ellipse fitting. *IEICE Transactions on Information and Systems*, E83-D(11):2008–2011, 2000.

J. Tang, S. Kawato, and J. Ohya. A face recognition system based on wavelet transform and neural network. In *Proceedings of the International Conference on Wavelet Analysis and its Applications*, 1999.

P. Viola and M. Jones. Robust real-time object detection. In *Second International Workshop on Statistical Learning and Computational Theories of Vision Modeling, Learning, Computing and Sampling*, July 2001a.

P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the Conference on Computer vision and Pattern Recognition*, 2001b.