Lab Manual - DD2427 Image Based Classification and Recognition

Babak Rasolzadeh, Josephine Sullivan

April 28, 2015

Face Detection

Real-time face detection in multi-scale images with a 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. 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 [2001]. This basically involves the interpretation of Haar-like features in a boosted cascade, see paper on the course homepage.

The Lab project: A Basic Face Detector

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

To get started go to the Lab project webpage and download the following:

Training images: `TrainingImages.tar.gz`, the database of face and non-face images. 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`.

Debugging information: Throughout this lab you will verify the correctness of your code by checking your output against previously computed results `DebugInfo.tar.gz`.

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 different rectangular regions. For example 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

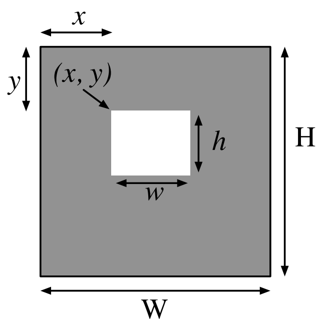
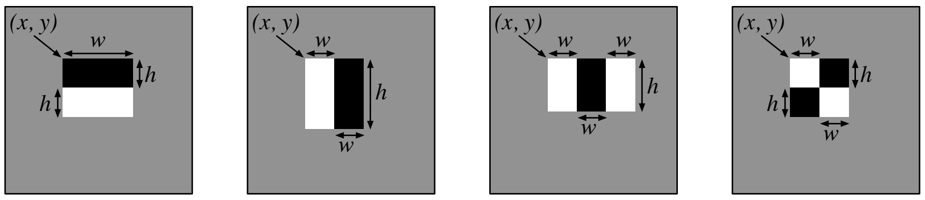


Figure 1: The features used are constructed from the sum of pixel intensities within rectangular regions. In this work a rectangular region is parametrized by the coordinates of its top left hand corner and its width and height.



(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 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.

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 for an image. Our aim is to extract all these features with one matrix multiplication. That is if given an integral image `ii_im` then all the features can be computed as

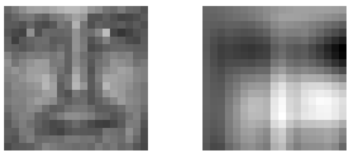


Figure 3: The original image and its integral image when computed from the original normalized image.

```
fmat * ii_im(:)
```

where `fmat` is an `nf`×`np` matrix (`nf` is the number of features computed and `np` is the number of pixels in the image). The steps to calculate `fmat` are now explained.

2.2.1 Sum of pixel values within a rectangular region

A rectangular region is defined by 4 numbers `(x, y, w, h)` as shown in figure 1. There are two ways you can compute the sum of the pixel intensities in this region:

1. You can explicitly compute the sum as in equation (1) from the original normalized image using the *Matlab* function `sum`. The problem with this approach is that it is very slow. It is much too slow for this project.
2. Exploit the fact that you have pre-computed the integral image. Use the integral image to compute the box sum by indexing it four times and computing a couple of subtractions and additions. This approach is much more computationally efficient than the first approach.

Your task is to write a function `VecBoxSum.m` that returns a vector `b_vec` (of size `1`× `np`) such that

```
b_vec * ii_im(:)
```

computes the sum of the pixel values within a rectangular area.

Program 2: `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 row vector `b_vec` which will be all zeros except for 4 of its entries s.t.

```
b_vec * ii_im(:)
```

equals the sum of the pixels intensities in the rectangular region defined by `(x, y, w, h)`. To find the coordinates (in 1d) of the four non-zero entries, first calculate the 2d coordinates you have to access in the 2d array `ii_im` to compute the box sum. Once you have these 2d coordinates then it is relatively easy to transform them into the corresponding 1d coordinates when `ii_im` is turned into a 1d vector with the command `ii_im(:)`.
There are boundary cases that you will have to take care of (or explicitly state that you do not compute because you will exclude them from the set of features you extract from the image). This is when for your input variables either `x=1` or `y=1`.
Sanity Check:
Given the integral image `ii_im` of `face00001.bmp`, calculate

```
>> b_vec = VecBoxSum(x, y, w, h, 19, 19);
>> A1 = b_vec * ii_im(:)
>> A2 = sum(sum(im(y:y+h-1, x:x+w-1)))
```

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

Being able to compute the sums of pixel intensity in a box forms the basis for the evaluation of the four features types used in this lab. And this is the next step

2.2.2 Feature computations

Each feature is defined by its type (1, 2, 3 or 4) 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 function `VecFeature.m` that when given the parameters of the feature type (number and size) returns a row vector `ftype_vec` s.t.

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

computes the feature on the original image `im`.

Program 3: `function ftype_vec = VecFeature(ftype, W, H)`

In this function, calculate the row vector required to compute the response for a feature defined by `ftype`. Remember features of type I can be calculated as

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

therefore the corresponding row vector needed to compute this feature with a matrix multiplication is

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

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 = ftype_vec * ii_im(:)
>> b0 = sum(sum(im(y:y+h-1, x:x+w-1)));
>> b1 = sum(sum(im(y+h:y+2*h-1, x:x+w-1)));
>> A2 = b0 - b1
```

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.

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');
>> fs = dinfo2.fs; W=19; H=19;
>> abs(fs(1) - VecFeature(dinfo2.ftypes(1, :), W, H)* ii_im(:))  > eps
>> abs(fs(2) - VecFeature(dinfo2.ftypes(2, :), W, H)* ii_im(:))  > eps
>> abs(fs(3) - VecFeature(dinfo2.ftypes(3, :), W, H)* ii_im(:))  > eps
>> abs(fs(4) - VecFeature(dinfo2.ftypes(4, :), W, H)* ii_im(:))  > eps
```

2.3 Enumerate all features

At this point you have written code that allows you 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 4: `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`×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. 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`×`19` image. Do this by checking that for every feature defined `x+w-1` ≤ `W` and `y+h-1` ≤ `H`. `nf` should have a value around `32,746`.

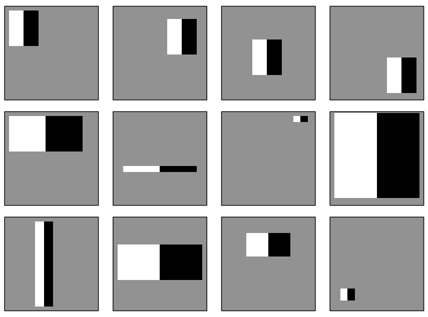


Figure 4: A small subset of different possible features of type II.

The next task is to compute a feature matrix `fmat`. The rows of this matrix are the `ftype_vec` for each feature defined by `all_ftypes`. Write the function `VecAllFeatures.m` to compute this.

Program 5: `function fmat = VecAllFeatures(all_ftypes, W, H)`

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

Debug Point: Use the integral image, `ii_im`, computed from the image `face00001.bmp`, check your newly written function with:

```
>> dinfo3 = load('DebugInfo/debuginfo3.mat');
>> fmat = VecAllFeatures(dinfo3.all_ftypes, W, H);
>> sum(abs(dinfo3.fs - fmat * ii_im(:))  > eps)
```

2.4 Extract features and training data

You can obtain a list of the images in the directory `'FACES'` with:

```
face_fnames = dir('FACES/*.bmp');
```

The names of the first image you will use for training can be accessed as:

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

The next function you will write is to load all the images or a subset of the images in a directory.

Program 6: `function ii_ims = LoadImDataDir(dirname, varargin)`

List the images: List the images in the directory `dirname`. If you don't want to load all the images (because you are debugging etc.), you can send in an extra number, which says I want to load the first `ni` images, after `im_sfn`. (Use a with variable-length input argument list.) You can access this number in your function with `varargin{1}`.

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

Sanity Check:
Run your newly written function on the `'FACES'` directory with `ni = 100`. 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;
>> ii_ims = LoadImDataDir(dirname, ni);
>> fmat = VecAllFeatures(dinfo4.all_ftypes, 19, 19);
```

Then check that `dinfo4.fmat` equals the `fmat` you calculated and similarly for `dinfo4.ii_ims`. Also check that `dinfo4.fs` equals `fmat * 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 7: `function SaveTrainingData(all_ftypes, train_inds, s_fn)`

Write a function which calls `LoadImDataDir` twice - once to load all the face images and the second time round to load all the non-face images. You should concatenate the two arrays returned by the two calls to `LoadImDataDir` to create an array `ii_ims` of size $361\times$`(n)` which contains the integral images for all the images (both face and non-face) in the database. To accompany the array `ii_ims` you should create a vector `ys` of length `n` that contains the ground-truth label (`+1` for faces and `-1` for non-faces) for each image in `ii_ims`. After you have loaded the images you should then compute `fmat` by calling the function `VecAllFeatures` with `all_ftypes`. The input vector `train_inds` contains the indices of images you will use for training (that is `ii_ims(:, train_inds)`). Having read in all the images and computed `fmat`, you should save to the file `s_fn` all the data you will need for training: (`W` and `H` correspond to the width and height of the original images.)

```
save(s_fn, 'ii_ims', 'ys', 'fmat', 'all_ftypes', 'W', 'H');
```

To create the data required for training the face detector run the code:

```
>> dinfo5 = load('DebugInfo/debuginfo5.mat');
>> train_inds = dinfo5.train_inds;
>> all_ftypes = dinfo5.all_ftypes;
>> SaveTrainingData(all_ftypes, train_inds, 'training_data.mat');
```

It may take several minutes for the function `SaveTrainingData` to complete, depending on the speed of your machine and the effeciency of your code. Once it has completed, load your saved file into *Matlab* with the command:

```
>> Tdata = load('training_data.mat');
```

`Tdata` is a structure and contains the data you have just saved. So, for instance, the integral image data is accessed with `Tdata.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 forms a strong classifier from a linear combination of weak ones. In the context of Viola-Jones face detection, a binary classification task, the weak classifiers are derived from the set of extracted features.

The details of the *AdaBoost* algorithm are given in algorithm 1. The core idea behind AdaBoost is the weight distribution maintained over the training examples and the modification of this distribution during each iteration of the algorithm. At the beginning the weight distribution is flat, but after each iteration the chosen weak classifier returns a classification for each of the training-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 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. In the mathematical description of the algorithm we denote the feature vector extracted when all the Haar-like filters are applied to an image $\mathbf{x}$ as $\mathbf{f}_\mathbf{x} = (f_1(\mathbf{x}), \ldots, f_N(\mathbf{x}))$ where N is the total number of features extracted. Then one weak classifier $h(\cdot)$ with parameters $\boldsymbol{\Theta} = (j, p, \theta)$

$$h(\mathbf{f}_\mathbf{x}\,;\,\boldsymbol{\Theta}) = h(\mathbf{f}_\mathbf{x}\,;\,j, p, \theta) = \begin{cases} +1 & \text{if } p\,f_j(\mathbf{x}) < p\,\theta \\ -1 & \text{otherwise} \end{cases} \tag{8}$$

where $f_j(\mathbf{x})$ is the response of the jth feature when applied to image $\mathbf{x}$. 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

Algorithm 1 AdaBoost

Input: A set of feature vectors $\{\mathbf{f}_{\mathbf{x}_1}, \ldots, \mathbf{f}_{\mathbf{x}_n}\}$ extracted from each example image $\mathbf{x}_i$ and associated labels $\{y_1, \ldots, y_n\}$ where $y_i \in \{-1, 1\}$. $y_i = -1$ 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:

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

```
Steps of Algorithm:
  Initialize the $n$ weights to:

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

  for $t = 1, \ldots, T$ do
    • Normalize the weights so they sum to one: $w_i^{(t)} = w_i^{(t)} / (\sum_j w_j^{(t)})$.
    • For each feature $j$ train a weak classifier $h$ that just uses the $j$th feature and tries to minimize the error

      $$\epsilon_j = .5 \sum_i w_i^{(t)} \, | \, h(\mathbf{f}_{\mathbf{x}_i}; j, \boldsymbol{\theta}_j) - y_i \, | \tag{6}$$

    • Choose the weak classifier with the lowest error: $j^* = \arg\min_j \epsilon_j$
    • Set $\boldsymbol{\Theta}_t = (j^*, \boldsymbol{\theta}_{j^*})$ and $\epsilon_t = \epsilon_{j^*}$.
    • Set $\alpha_t = \frac{1}{2} \log \left( \frac{1-\epsilon_t}{\epsilon_t} \right)$.
    • Update the weights:

      $$w_i^{(t+1)} = w_i^{(t)} \, \exp\{-\alpha_t \, y_t \, h(\mathbf{f}_{\mathbf{x}_i}; \boldsymbol{\Theta}_t)\} \tag{7}$$

  end for
```

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 is meant when referring to feature responses. Remember, the structure `Tdata` contains the matrix `fmat`. Each row of this matrix corresponds to a row vector which when multiplied with the integral image of an image (represented as a column vector) produces the result of applying a particular Haar-like feature to the original image. `Tdata` also contains the integral images extracted from the face and non-face training images. Using these integral images and one row of `fmat`, say `fmat(12028, :)`, one can compute the feature responses for all the images for this feature type with a simple matrix multiplication. From this data create a vector of responses `fs = fmat(12028, :)*ii_ims` for the training data. Using the labels `Tdata.ys` and `hist` one can 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 outputs are then the learnt parameters of the weak classifier and its associated error.

Program 8: `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 `Tdata` contains the feature array `fmat` and the integral image data and the ground truth labels. Using these integral images and one row of `fmat`, say `fmat(12028, :)`, compute the feature responses for one `ftype`. Use this data to create `fs`. 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 = -3.7698` 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.

Algorithm 2 Simple weak classifier

Input: A set of feature responses $\{f_j(\mathbf{x}_1), \ldots, f_j(\mathbf{x}_n)\}$ extracted by applying the feature f_j to each training image $\mathbf{x}_i$ and associated labels $\{y_1, \ldots, y_n\}$ where $y_i \in \{-1, 1\}$. A set of non-negative weights $\{w_1, \ldots, w_n\}$ associated with each 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:

$$g(f_j(\mathbf{x})\,;\,p,\theta) = \begin{cases} +1 & \text{if } p\,f_j(\mathbf{x}) < p\,\theta \\ -1 & \text{otherwise} \end{cases} \tag{9}$$

ϵ is the value of the error associated with this classifier when applied to the training data.

(The parameters $\boldsymbol{\theta} = (p, \theta)$ along with j will then be the parameters of the weak classifier defined in equation (8).)

Steps of Algorithm:

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

$$\mu_{\text{P}} = \frac{\sum_{i=1}^{n} w_i\, f_j(\mathbf{x}_i)\,(1+y_i)}{\sum_{i=1}^{n} w_i\,(1+y_i)}, \quad \mu_{\text{N}} = \frac{\sum_{i=1}^{n} w_i\, f_j(\mathbf{x}_i)\,(1-y_i)}{\sum_{i=1}^{n} w_i\,(1-y_i)} \tag{10}$$

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

- Compute the error associated with the two possible parity values

$$\epsilon_{-1} = .5 \sum_{i=1}^{n} w_i\, |y_i - g(f_j(\mathbf{x}_i)\,;\,-1, \theta)|, \tag{11}$$

$$\epsilon_{1} = .5 \sum_{i=1}^{n} w_i\, |y_i - g(f_j(\mathbf{x}_i)\,;\,+1, \theta)| \tag{12}$$

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

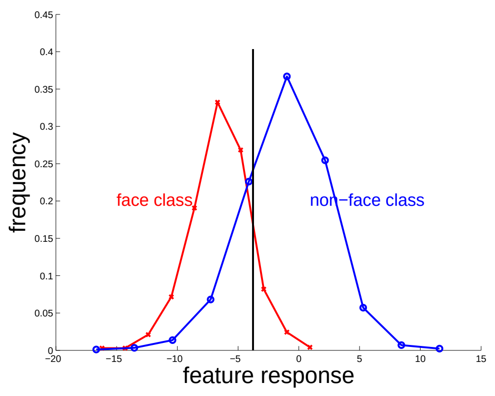


Figure 5: **A simple weak classifier.** The blue curve is the histogram of the feature responses for the non-face training images. The red curve those for the face images. The black line is the threshold value chosen using algorithm 2.

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`.

Program 9: `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` and compare to 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 10: `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).

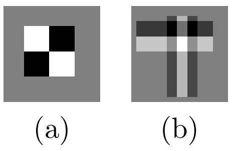


Figure 6: Example output of the feature and classifier display.

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. `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 $\mathbf{\Theta}_t$'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`×`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 11: `function Cparams = BoostingAlg(Tdata, 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(Tdata, 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 your code, run the following commands

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

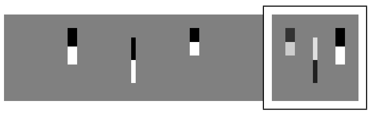


Figure 7: **The initial 3 features chosen by boosting (left to right) and the final strong classifier**. The final strong classifier in this example consists of 3 features and is the rightmost figure. The first feature chosen by boosting is the leftmost one. These are the features chosen from a very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

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 to run 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 work, then run this debug check and go get yourself a cup of coffee! It may take several minutes to run. (Depends on your machine and the efficiency of your code.)

```
>> dinfo7 = load('DebugInfo/debuginfo7.mat');
>> T = dinfo7.T;
>> Cparams = BoostingAlg(Tdata, 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`.

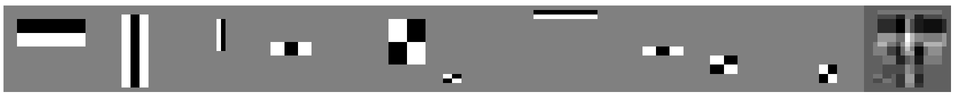


Figure 8: **The initial features chosen by boosting (left to right) and the final strong classifier**. The final strong classifier in this example consists of 10 features and is the rightmost figure. The first feature chosen by boosting is the leftmost one.

If you have successfully reached this point, 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.

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 12: `function scs = ApplyDetector(Cparams, ii_ims)`

This function applies your strong classifier to a set of test images. It takes as input the parameters of your classifier `Cparams` and an array of integral images, `ii_ims`, computed from a normalized version of your test images, such that each integral image is a column of `ii_ims`. It extracts each feature used in the strong classifier from the test images and then computes a weighted sum of the weak classifier outputs. The function returns for each test image the score

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

Sanity Check:
Run your function on `face00001.bmp`. I got a score of around `2.7669`.

We now introduce some concepts needed to define the ROC-curve. Table 1 reviews the definitions of true-positive, false-positive etc. Given these

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.

definitions the *false positive rate* and *true positive rate* are:

$$\text{true positive rate} = \text{tpr} = \frac{n_{\text{tp}}}{n_{\text{tp}} + n_{\text{fn}}} \tag{13}$$

$$\text{false positive rate} = \text{fpr} = \frac{n_{\text{fp}}}{n_{\text{tn}} + n_{\text{fp}}} \tag{14}$$

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 $+\infty$. (The default `AdaBoost` threshold is designed to yield a low error rate on the training data.) This curve tells you the loss in specifity you will suffer for a required accuracy. You can now write the function to compute the ROC curve on the training data you didn't use when learning your classifier.

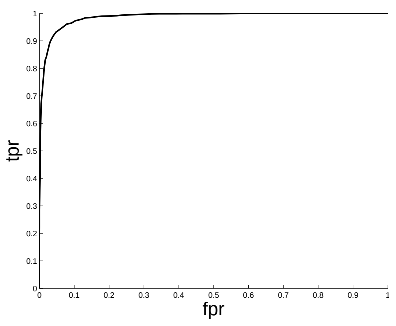


Figure 9: **ROC curve computed from the images omitted from training**. The classifier usesd consisted of 10 weak classifiers.

Program 13: `function ComputeROC(Cparams, Tdata)`

Get test images: From `Tdata.train_inds` you can see which images in `Tdata.ii_ims` you used for training. For computing the ROC curve you all images in `Tdata.ii_ims` not used for training as the test images. You can use the function `setdiff` to get the indices of the test images.

Apply detector to each test image: Run `ApplyDetector` on the test images and record their scores.

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 1.75. 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 - an `nd`$\times$`4` array where `nd` is the number of face detections. It is important to note that the variance and the mean

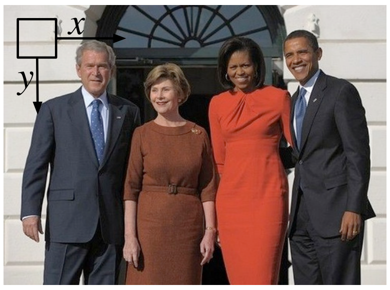


Figure 10: The sliding window of `ScanImageFixedSize.m` will traverse different locations in the large image.

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 14: `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 19×19 sub-window we grab from 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.

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.

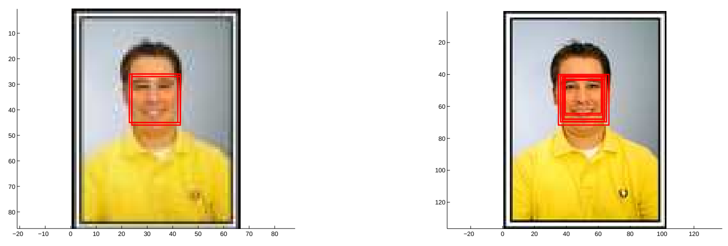
```
>> im = 'TestImages/one_chris.png';
>> dets = ScanImageFixedSize(Cparams, im);
```

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 15: `function DisplayDetections(im, dets)`

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

Sanity Check:
Displaying the results of `ScanImageFixedSize` on `one_chris.png` you should get results similar to 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 × 19` patch is examined labelled as face or non-face. A threshold of 1.75 was used. While the right image, which is bigger than the left one, is searched over a range of scales. A threshold of 3 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). You can 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 *significantly*. Frequently, what significantly means is the following. Let A and B correspond to the bounding boxes of two face detections then they are considered as being generated by the same face if

$$\frac{\text{area}(A \cap B)}{\text{area}(A \cup B)} > \rho \tag{15}$$

where $0 < \rho < 1$ and usually ρ is set relatively high. After computing these partitions 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 or it is the corners of the bounding box with the highest response.

Program 16: `function fdets = PruneDetections(dets)`

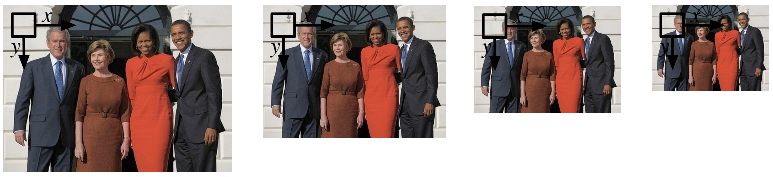
Find overlapping detections: Let `nd` be the number of detetections. Create a matrix `D` of zeros of size `nd`×`nd`. Then set `D(i,j) = 1` if the `i`th detection overlaps the `j`th detection according to equation (15) (the function `rectint` may be of use).

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.

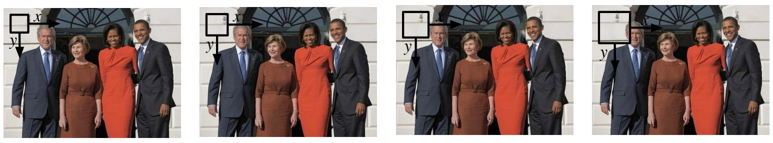
Sanity Check:
Check before and after pictures: Display the detections before you run this function and then afterwards. Visually inspect if the function has performed the expected task.

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 17: `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. Use the *Matlab* command `profile` to analyze 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.

Optional Part II of Lab

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 $\approx$4000 positive examples and $\approx$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 lists of some suggestions (by no means exhaustive).

Improve the performance of your basic classifier

- Increase the amount of training data you use for training and see how much performance increases on test images. Some options are
  - Use all the images in the database for training.
  - Artificially generate more positive training examples by perturbing the existing positive training examples with random rotations, scalings and translations.
  - Perform *hard-negative mining*. That is given images containing no faces, run your trained detector on these images. Record all the false positive hits and add these patches to your negative set, then retrain your classifier using this augmented negative training data.
  - There are many databases of faces publicly available. You could exploit these for training and testing.
- Increase the pool of features you extract from each image patch by apply Haar features to not just the grayscale intensity image but also to the gradient magnitude image. For some inspiration check out the papers by Dollar et al. [2009] and Benenson et al. [2013]. You could also include HOG, SIFT, LBP type features etc. But these are expensive to compute so there would probably be a
- A better weak classifier. For example Yyu could use a decision tree of depth > 1 as the weak classifier, see the reference Benenson et al. [2013] for more ideas.

Improve speed of detection or training times

- Implement a cascade of classifiers as in the original Viola & Jones to speed up detection.
- Speed up the training process Pham and Cham [2007].
- There is a large correlation amongst the responses from similar features. Could this be potentially exploited to speed up training?

Compare performance of boosted classifier to other classifiers

If are interested in the other classifiers and features that we learnt about during the course, you could explore how well they perform for face detection in terms of either accuracy, speed of training, speed of detection, or a combination of these factors. Here are some ideas:

- Using the VLFeat package you could build a feature descriptor of an image patch from a subset of its Haar feature, RGB, HOG, SIFT, and LBP descriptors. Then train a linear classifier trained with an linear kernel SVM or with logistic regression, or a non-linear classifier such as a random forest or a kernel SVM. Note if you have a very large dimensional feature vector then it is best to perform some form of feature selection before you train an SVM classifier. You could perhaps use boosting or random forest to perform this feature selection and then train an SVM classifier just using these selected features. Packages like `libsvm` and `liblinear` also allow one to train with L_1 regularization which promotes a sparse weight vector which corresponds to performing feature selection.
- There are several software packages available to train ConvNets. These include `cuda-convet` (C++), `torch` (lua), `Caffe` (C++) and perhaps the most accessible because it is written in Matlab is MatConvNet:CNNs for MATLAB. You could use one of these packages to train a ConvNet to perform face classification and see how well it performs.

References

R. Benenson, M. Mathias, T. Tuytelaars, and L. V. Gool. Seeking the Strongest Rigid Detector. In *Proceedings of the Conference on Computer vision and Pattern Recognition*, 2013.

P. Dollar, Z. Tu, P. Perona, and S. Belongie. Integral Channel Features. In *Proceedings of the British Machine Vision Conference*, 2009.

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.

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 2001.