Implementation of "Robust Real-time Object Detection" (Viola, Jones)

Advisor: Dr. Rita Osadchy, University of Haifa

Prepared by: Daniel Heilper

Natan Silnitsky

TOC

Abstract

We present an efficient implementation of the Viola& Jones paper:
Our implementation is written in Matlab.
The code can easily be migrated to C++ performance libraries (IPP, OpenCV, SSE), using the same code structure, in order to produce a commercial product.

Key Features:

- Use of box of windows for simultaneous operations.
- Fast threshold determination & weight updating
- Fast integral image calculation.
- Pre processing of all features application on all of the windows
- Time complexity is substantially reduced in expense of memory complexity.
- Final Detector comprised of several partial cascades built on top of each other.
- Image testing is done using multiple scales of the detector up to a ceiling of 384X288.
- Images with Sizes above the ceiling are downsized.

“Robust Real-time Object Detection”

In 2001 Viola & Jones presented a framework for robust and extremely rapid object detection, in particular face detection.
The Framework Comprised of the following key elements:

- Haar-Like Features coupled with Integral Image
- A Learning Algorithm based on AdaBoost
- Fast “Clutter” discarding using a cascade of classifiers

Haar-Like Features

Their procedure classifies images based on the value of simple features, reminiscent of Haar basis functions. The value of a two-rectangle feature is the difference between the sum of the pixels within two rectangular regions. The regions have the same size and shape and are horizontally or vertically adjacent (see Figure 1). A three-rectangle feature computes the sum within two outside rectangles subtracted from the sum in a center rectangle. Finally a four-rectangle feature computes the difference between diagonal pairs of rectangles.

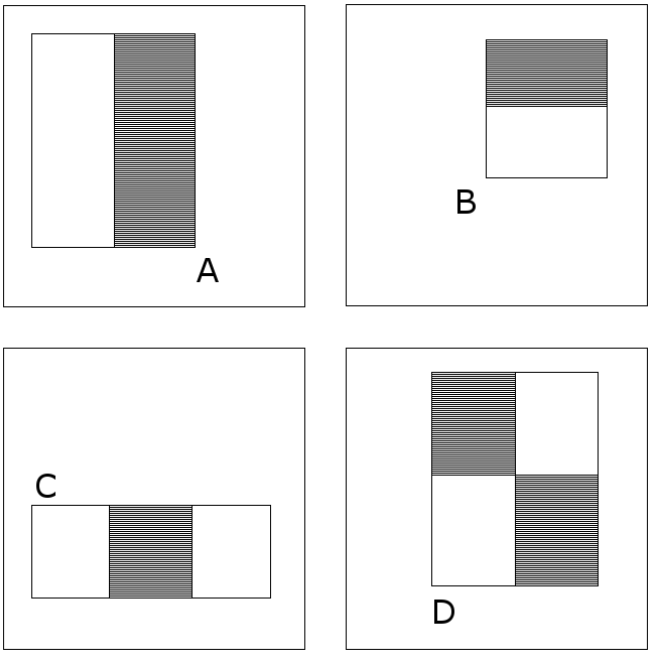


Figure 1: The different Feature Types.

Integral Image

Rectangle features can be computed very rapidly using an intermediate representation for the image known as the integral image.
The integral image at location x, y contains the sum of the pixels above and to the left of , inclusive:

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

where ii(x, y) is the integral image and i(x, y) is the original image.

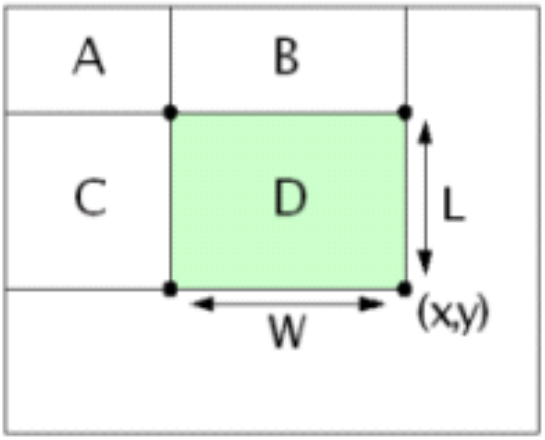


Figure 3: The sum of the pixels within rectangle D can be computed with four array references. D = D' – C' – B' + A' (' means lower right corner).
4 + 1

AdaBoost

In its original form, the AdaBoost learning algorithm is used to boost the classification performance of a simple learning algorithm (e.g., it might be used to boost the performance of a simple perceptron). It does this by combining a collection of weak classification functions to form a stronger classifier. The learner is called weak because we do not expect even the best classification function to classify the training data well. In order for the weak learner to be boosted, it is called upon to solve a sequence of learning problems. After the first round of learning, the examples are re-weighted in order to emphasize those which were incorrectly classified by the previous weak classifier. The final strong classifier takes the form of a perceptron, a weighted combination of weak classifiers followed by a threshold.
One practical method for completing this analogy is to restrict the weak learner to the set of classification functions each of which depend on a single feature. In support of this goal, the weak learning algorithm is designed to select the single rectangle feature which best separates the positive and negative examples. For each feature, the weak learner determines the optimal threshold classification function, such that the minimum number of examples are misclassified. A weak classifier ($h_j(x)$) thus consists of a feature (f_j), a threshold (θ_j) and a parity (p_j) indicating the direction of the inequality sign:

$$h_j(x) = \begin{cases} 1 \text{ if } p_j f_j(x) < p_j \theta_j \\ 0 \text{ otherwise} \end{cases}$$

Cascade

The Goal of Constructing a cascade of classifiers is to achieve increased detection performance while radically reducing computation time. The key insight is that smaller,

and therefore more efficient, boosted classifiers can be constructed which reject many of the negative sub-windows while detecting almost all positive instances. Simpler classifiers are used to reject the majority of sub-windows before more complex classifiers are called upon to achieve low false positive rates.

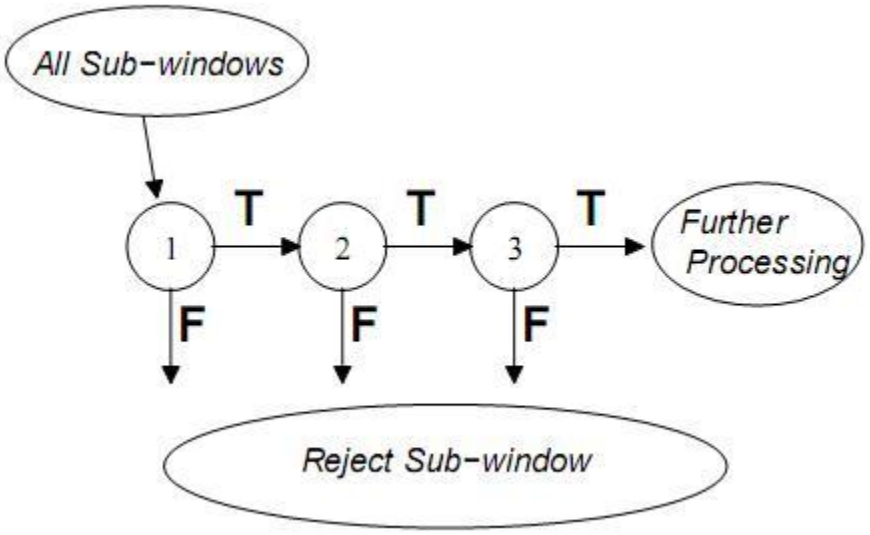


Figure 4: General concept of cascade

Our Approach

Use of Matlab

We chose to implement the algorithm (both Training & Testing) in Matlab. There are several good reasons to choose Matlab here. First, vector & matrix operations are basic operation in Matlab. Second, Matlab makes redundant the need for memory management (allocation, deallocation). Third, it has very good I/O capabilities vis-à-vis Images. Fourth, the Matlab environment is optimized for vector & matrix debugging (easy access to complex structures).
On the other hand, Matlab is an interpreted language, and is inherently slow.
In view of this we decided to migrate time consuming parts of the code to c, if running time turned out to be slow.
Any high time complexity task will inherently be slow both in Matlab & native languages, therefore we focused more on reducing algorithmic time complexity than lower complexity constants.

Order of Implementation

After careful reading of the paper, we mapped out all of the required functions.
We still felt we hadn't acquired the whole "picture". As a result we decided to implement the Adaboost algorithm using the bottom-up approach, meaning we implemented the more basic functions first and only then did we build the main Adaboost function.
We decided on a top-down approach for the cascade implementation, as we felt we understood the algorithm well in advance.

Finding Feature threshold

After some deliberations and fine-tuning we designed the following full-search algorithm:

1. Retrieve values of computation of the feature on the positive & negative examples and put them in an array.
2. Sort the values array.
3. Create 2 additional arrays, positive and negative. Scan the retrieved values array. Fill the arrays by Accumulating units of 1 multiplied by the value's weights in the following fashion:
4. For a positive value,
   - 4.1. in the positive array, add the previous element's content and add 1 multiplied by the value's weight.
   - 4.2. in the negative array just add the previous element's content
5. For a negative value,
   - 5.1. in the negative array, add the previous element's content and add 1 multiplied by the value's weight.
   - 5.2. in the positive array just add the previous element's content
6. The value index with the best threshold will be the index where the absolute difference between the arrays content is greatest.

Polarity is the side of the threshold which has more positive examples, meaning polarity is positive if more positive examples occur on the left side of the threshold, and polarity is negative if there is less on that side.

Quick feature error calculation is achieved by logical vector operation (xor) between the classified vector and the sorted label vector (the "Y").

Example (Disregarding the fact that each example has a weight)

Pos examples feature application values

Value	9	5	4	10	1
Classification	1	1	1	1	1

Neg examples feature application values

Value	10	15	12	14	8
Classification	0	0	0	0	0

Sorted examples feature application values

Value	1	4	5	8	9	10	10	12	14	15
Classification	1	1	1	0	1	1	0	0	0	0

Positive examples accumulation array

accumulation	1	2	3	3	4	5	5	5	5	5

Negative examples accumulation array

accumulation	0	0	0	1	1	1	2	3	4	5

Absolute difference between the arrays

abs-diff	1	2	3	2	3	4	3	2	1	0
threshold	1	4	5	8	9	10	10	12	14	15

The chosen threshold is **10**, polarity is positive (because more positive examples are on the left of the threshold)

Reducing Time complexity

While implementing the training and testing algorithm, our goal was to reduce time complexity as much as possible.
We achieved this by building data structures such that in Matlab it was is possible to compute operation on them in 1 function call and avoid loops. The overhead is an initial construction of these data structures.
Data Structure Examples follows:

"Boxing" the examples

Training Examples are 19X19 frames.
During Training, every features needs to be applied on all integral images of the training examples.
In order to perform fast Integral Image computation, and to perform fast application of features on the examples, we decide to put all of the example frames in a 3D box.
We use the 3D box in Testing as well, putting all of the sliding windows of the test image in it.

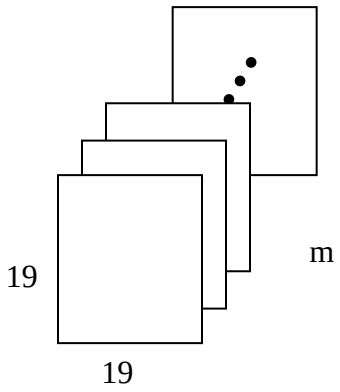


Figure 6: A 3D box with *m* training examples in it

Once all of the examples or sliding windows are in the box, simultaneous computation of each frame's Integral Image is as easy as 2 lines of code in matlab (no loops involved):
box = cumsum(box,1);
box = cumsum(box,2);

In addition, computing a feature's application simultaneously on all examples or sliding windows becomes much faster, because the Integral Image values of the rectangle vertices of all of the examples can be retrieved simultaneously when they are structured in a box (i.e. colon operation in Matlab).

Padding the image examples with zeros (as the 1st row & 1st column) made the integral image calculations work on all of the image pixels, during creation of the box, and during the simultaneous retrieval of values from it.

Calculating Feature application values only Once

In training, Feature application values are computed before we start the learning process for all training examples simultaneously. The result is a matrix with m rows representing m features & n columns representing n examples. Each matrix element is a feature applied on a training example.
Because of this data structure we only compute the feature application for each training example once during the entire training process (adaboost + cascade).
All iterations of the Adaboost process use the same feature application values, including threshold determination & misclassification calculation. Even when cascade training process discards parts of the negative examples that are definitely not considered as a face, there is no need for any new calculations. Instead of discarding the irrelevant negative examples, the corresponding irrelevant feature application values are discarded.

Simultaneous Misclassification Calculation & Examples Weights Update

Updating Weights Adaboost stage is performed as such that misclassification calculation of the example set & the weights updating of the example set are done at once.
Pseudo Code:

```
- sort examples values
- apply threshold
- xor with labels
- update weights for every misclassified example
```

Reduce correctness testing time

When we finished writing the code, we decided we will not immediately jump to check if we got nice results on test images, but write unit tests instead, i.e. methodically test for bugs and correctness of each of the functions we wrote.
In order for these unit tests to be independent of each other (so there will be no "running error") we simulated new inputs for each of the functions.
Only after we had finished this, we started testing to see if our implementation gives meaningful results. The thought behind doing extensive unit testing was to significantly reduce functional testing time and to focus our minds first on simple bugs and in later stage on more complex errors in implementation.

Results

Our final detector is a cascade of 9 classifiers with a total of 206 features.
The training set was comprised of Faces & Non-Faces examples of size 19X19.
The first 5 classifiers were trained on 2429 faces & 4548 non-faces from the *CBCL face database* using the cascade method (we set the detection rate to be 99% and the false positive rate to be 0.25%).

The next 3 classifiers were trained independently on 4500 Non-faces examples each, that were taken from the previous partial cascade resulting false alarms on random images from the web, which had no faces in them (we set the detection rate to be 99.7% and the false positive rate to be 0.1%).

.

The final 9th classifier was trained in the same way as the previous three, only this time the non-faces examples were taken from the partial cascade comprising 8 classifiers resulting false alarms (we again set the detection rate to be 99.7% and the false positive rate to be 0.1%).

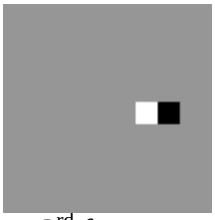
Detecting is done multiple scales up to a ceiling of 384X288. Images sized above this ceiling are downsized.

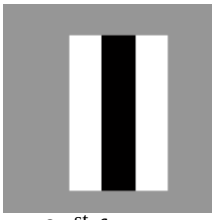
The final cascade:

stage	1st cascade					2nd group			Last one
Number of features	5	8	9	10	6	38	35	47	48

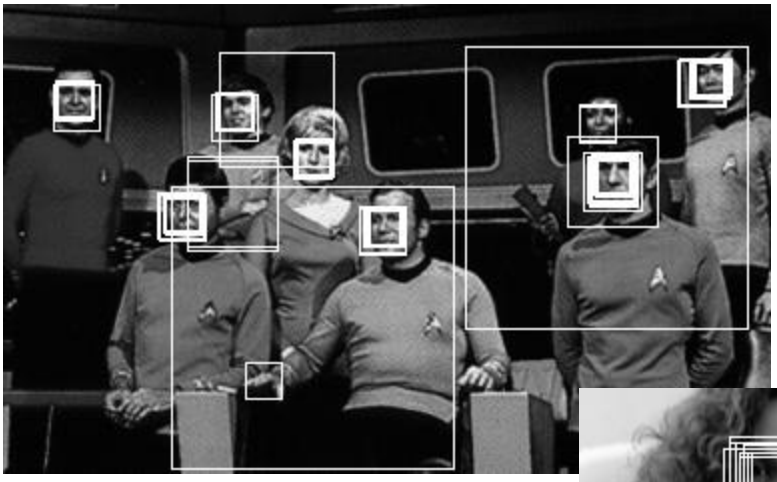

1st feature

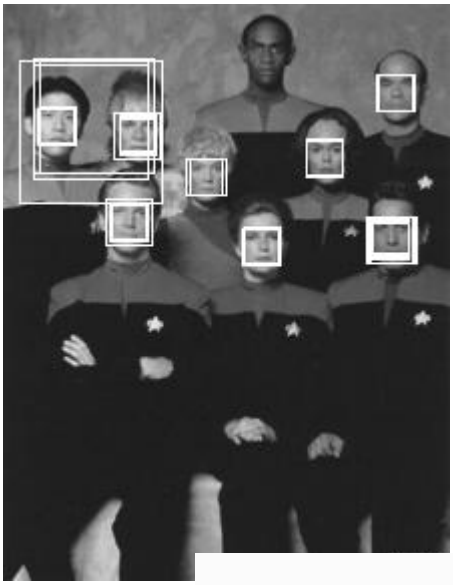

2nd feature

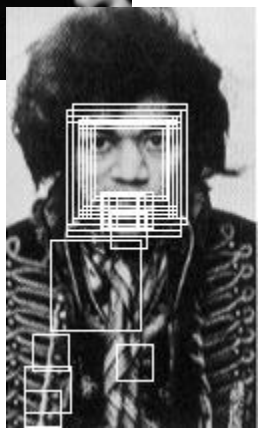

3rd feature


201st feature

Few image examples:







Results Table for (Rowley, Baluja, Kanade; CMU) dataset

Computer details

Image Name	Detection Rate(%)	Detected/ Overall	Detection Rate(%) *Frontal, No Facial Hair, 19X19 and above, Closed Mouth, Open Eyes*	Detected/ Overall *Frontal, No Facial Hair, 19X19 and above, Closed Mouth, Open Eyes*	False alarms	Elapsed Time (seconds)
Weighted Average(*)	**49**		**63**		**2.7**	**8.11**
Aerosmith-double.tiff	60	3/5	100	2/2	1	11.33
baseball.tiff	0	0/1		0/0	0	6.35
Blues-double.tiff	0	0/4	0	0/3	0	10.06
boat.tiff([7])	0	0/2		0/0	9	11.57
book.tiff([1])	0	0/2		0/0	0	2.32
cnn1085.tiff	0	0/1		0/0	0	8.18
cnn1160.tiff([8])	0	0/2		0/0	0	7.76
cnn1260.tiff	100	1/1	100	1/1	0	7.88
cnn1630.tiff	100	1/1	100	1/1	1	8.50
cnn1714.tiff	100	1/1		0/0	1	8.40
cnn2020.tiff	0	0/1	0	0/1	0	7.08
cnn2221.tiff	0	0/1		0/0	1	8.13
cnn2600.tiff	100	1/1	100	1/1	0	8.40
ds9.tiff	57	4/7	67	4/6	1	11.26
eugene.tiff	0	0/1		0/0	0	5.24
hendrix1-bigger.tiff([2])	0	0/25		0/0	32	10.23
hendrix2.tiff	100	1/1	80	1/1	8	2.80
henry.tiff	0	0/1	0	0/1	0	11.19
judybats.tiff([3])	80	4/5	100	4/4	5	10.68
kaari1.tiff	100	1/1	100	1/1	1	5.50
kaari2.tiff	100	1/1	100	1/1	0	5.35
kaari-stef.tiff	50	1/2	100	1/1	0	10.36
knex0.tiff	100	1/1	100	1/1	4	8.25
knex20.tiff	NAN			0/0	1	7.94
knex37.tiff	0	0/1		0/0	0	7.50
knex42.tiff	NAN			0/0	1	8.12
lacrosse.tiff	0	0/1	0	0/1	1	2.57
life2100.tiff	0	0/3		0/0	1	8.20

life7422.tiff	0	0/1		0/0	0	7.28
mom-baby.tiff	50	1/2		0/0	0	6.53
music-groups-double.tiff	63	5/8	83	5/6	6	12.31
next.tiff	67	4/6	80	4/5	6	10.34
original1.tiff	100	8/8	100	8/8	5	10.20
original2.tiff	0	0/7	0	0/6		6.26
Pace-university-double.tiff	13	1/8	20	1/5	0	7.85
people.tiff	25	3/12	33	3/9	11	13.21
plays.tiff ([4])	0	0/4	0	0/1	1	6.14
puneet.tiff([5])	7	1/14		0/0	5	10.59
shumeet.tiff	100	1/1	100	1/1	2	10.83
tammy.tiff	0	0/1	0	0/1	2	5.55
voyager2.tiff	89	8/9	89	8/9	2	5.69
voyager.tiff([6])	0	0/9		0/0	3	6.73

(*) – Weighted for amount of faces, disregarding (numbered) images

([1]) – Original size is 277X801, faces after downsize are 10X10

([2]) – Original image size 814X820, faces' size is 15X15

([3]) – Original image size is 716X684, one face's size is 13X13 after downsize.

([4]) – Original image size is 539X734, 3 faces' size is 12X12 after downsize.

([5]) – Original image size is 580X380, many faces sizes are smaller than 19X19 after downsize.

([6]) – Original image size is 468X720, 8 faces' size is 13X13 after downsize.

Analysis

Unnecessary recalculation (which we discarded)

After analyzing initial training results we've noticed superfluous recalculation of features in Adaboost.
Cascade calls Adaboost iteratively with the *numOfFeatures* argument increasing incrementally in order to achieve a lower false alarm rate.
In our original implementation, in every call to Adaboost, the process found the best features in the amount of *numOfFeatures* all over again, even though it calculated (*numOfFeatures – 1)* in the previous time it was called by cascade.
The reason for this inefficiency was our original bottom up approach (see "Order of Implementation") which focused on Adaboost only, not realizing the strong coupling with cascade.

Overfitting

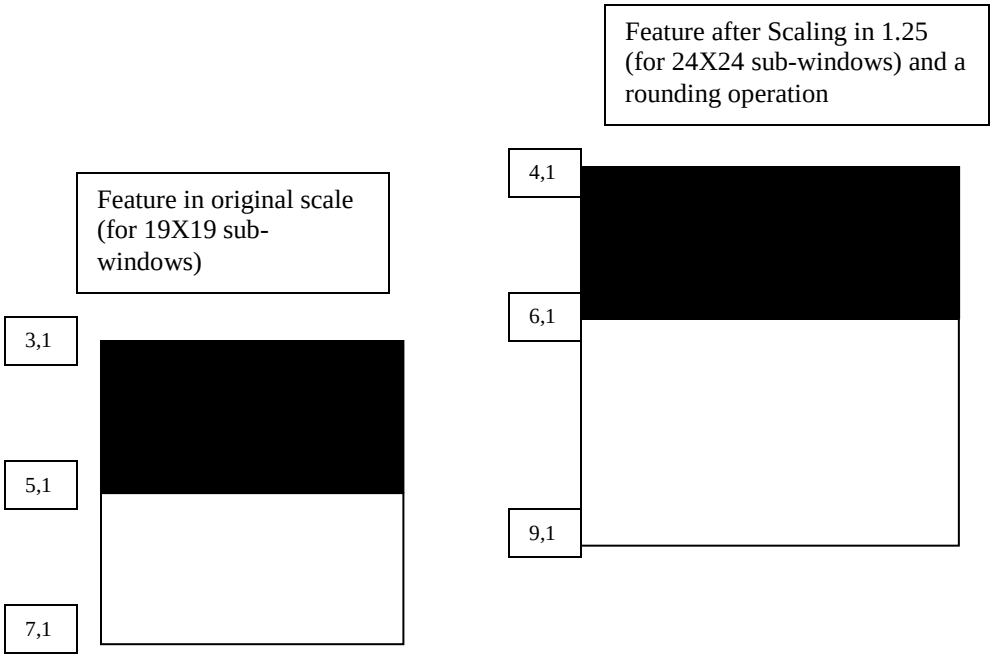
While experimenting with training parameters, we slowly raised detection rate and lowered false positive rate. After some point (99.9% detection rate & 0.01% false positive rate) less faces from the CMU dataset were detected, while not improving false alarm rate.

Using partial cascade to find false alarm examples

We settled on 99% detection rate and 0.25% false positive rate. After further examination of the paper we decided to stop the current cascade and start new ones which use false alarms from the old ones. False alarm rate improved significantly

Rounding of Scaling Operation

After 3 rounds of cascade and another round of negative set collecting from false alarms, And another cascade the false alarm rate was sufficient for the base scale of our detector (19X19), but bad for higher scales.
We came up with the idea the problem lies with the way we scale up the features, particularly with rounding of small rectangle features (rounding effects small number much more proportionally than for larger numbers). As a result rectangles were not congruent.
We solved this problem by scaling up the size of rectangle sides instead of scaling up integral image indices.
For example, the following feature's rectangles will not be congruent after scaling up:


Feature after Scaling in 1.25 (for 24X24 sub-windows) and a rounding operation
4,1
6,1
9,1
Feature in original scale (for 19X19 sub-windows)
3,1
5,1
7,1

Memory limitations

We were able to use only the following maximal set sizes in order to avoid Matlab memory limitations,
2429 faces
4548 Non Faces
21252 features (out of a possible 53130 features)

This limitation means that at the current way the code is designed (time complexity optimizations) we will not be able to achieve the same detection rate as Viola & Jones.

A 64bit Matlab version is needed + 6 GB of memory.
An alternative is to rewrite the code in a way that reduces memory needs (obviously in the expense of run-time)

Downsizing

The memory limitations mentioned above meant that we couldn't scale the detector for a size bigger than 384X288.
As a result, in testing, first we downsize the image to this size (while keeping image proportions), than we allow as many detector size increases as possible.
Another result is that perfectly good sized faces in the scaled down images will become smaller than 19X19 in size and thus will not be detected.

Further Work:

Overcoming downsizing problems

A possible way to work around the downsize problem for small faces is to scale the detector down. False alarm rate may go up slightly,

Merging overlapping detections

In order to determine more precisely detection rate and false positive rate, it is crucial to be able to merge overlapping detections of multiple scales into a single detection window. At this stage, it seems as a non trivial task.

Reducing further discretization errors caused by scaling detectors

For a single feature, we adjusted the scaling such that they will still preserve a Haar-like shape for each scaling. However, there are at least two more possible fine tuning adjustments:

1. Compensating the deficiency of geometrical similarity to the original rectangles that builds the feature in its basic size.
2. Compensating the feature change of relative origin within a scaled window.

The above discretization errors have greater weight in the first few scaling stages, while as the scaling is bigger, it becomes more negligible. Future work on examinations and testing of appropriate linear interpolation methods could be considered.

Easy vector migration to SSE & openCV/IPP

During the Matlab implementation phase, we worked on the task of migration to C++ (later we dimmed the migration unnecessary), including enabling work with mex & with visual studio compiler. As for mex-mechanism, we managed to transfer all types of data that we defined and used in our Matlab implementation. As for the C++ coding, further work can be done on how to use OpenCV and IPP data built in structures and API for "Boxing" operations, in order to preserve our code structure.

References

[1] P. Viola & M. Jones. Robust Real-time Object Detection. In *SECOND INTERNATIONAL WORKSHOP ON STATISTICAL AND COMPUTATIONAL THEORIES OF VISION*, VANCOUVER, CANADA, JULY 13, 2001.