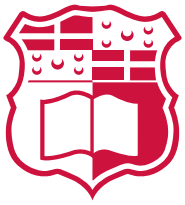

L-Università ta' Malta

ICS3206 - Assignment
Fast Frontal Face and Eye Detection using Viola-Jones Object Detection

Daniel Magro

January 2019

Statement of Completion

Item	Completed
Viola-Jones technical discussion	Yes
Artefact 1	Yes
Artefact 2	Yes
Comparison to alternative algorithm	Yes
Experiments and their evaluation	Yes
Overall conclusions	Yes

Viola Jones - Technical Discussion

All information in this section was adapted from [1], [2], [3] and [4].
All images were copied or adapted from [3] and [4].

Basic Terminology:
Negative or Background images are images which **do not** contain the object being detected
Positive images are images which **do** contain the object being detected

In Viola-Jones Object Detection, an image is converted to grayscale, and scanned using 'Haar Features'. These features are very effective at detecting edges and lines. The Viola Jones cascade looks for specific features in 'sub-windows' of the whole image, of varying size. Classification of these sub-windows is carried out with the use of a "detection cascade".

A detection cascade is a 'chain' of stages, where each stage looks for certain features in an image, contributing to whether an image contains the object in question. If it does, the image is passed on to the next stage, if not that image is immediately disregarded and considered not to contain the object. Each stage looks for a number of features in the image. The features in the earlier stages are chosen specifically because they offer the most differentiation between negative images and positive images. The earlier stages of the detection cascade are designed to eliminate negative images as early on as possible, with minimal computational power. Such a design is one of the factors that makes Viola-Jones object detection so fitting for real-time

applications. The thresholds of these features are adjusted to be rather low such that false negatives almost never happen.

In Viola-Jones Object Detection, rectangular features are used to classify images. Three main types of rectangular features are used, two-rectangle features, three-rectangle features and four-rectangle features. Two-rectangle features are primarily used for vertical or horizontal edge detection, three-rectangle features are suited for line detection, and four-rectangle features are used for diagonal edge detection.
Each feature calculates the sum of pixels in its individual rectangles, and uses those numbers to detect the presence of edges or boundaries.
For example, eyebrows are very often darker than the forehead above them, so a two-rectangle feature such as that in Figure 1 is very effective at de-

Figure 1: Horizontal Edge Feature

tecting an eyebrow. The detection of this feature alone, of course, does not conclude that a face has been detected, rather it needs to be detected alongside other features.
The bridges of noses are almost always brighter than the eyes on either side of them, so a three-rectangle feature such as that in Figure 2 may be used to

Figure 2: Vertical Line Feature

detect it. Similarly, when someone is smiling, their lips are darker than the

white teeth in between, so a horizontal three-rectangle feature such as that in Figure 3 can be used to detect a mouth.

Figure 3: Horizontal Line Feature

During training, a very large number of different types of rectangle features are applied in different sizes and areas of the image, in order to determine which features are the best at separating the positive images from the negative images. For a 24x24 image, around 180,000 features can be generated, which will be tested across the given dataset.
For example, during training it will probably be found that a horizontal two-rectangle feature is much more effective at detecting an eye than a four-rectangle feature will be at detecting a mouth, as it is fairly unlikely that a mouth will be darker in two opposite corners and lighter in the others.

These best features will be placed in the first stage of a detection cascade.
The cascade is essentially a degenerate decision tree, which passes what it thinks are faces to the next stage, and immediately disregards what it almost certainly knows aren't faces. The number of false negatives is minimised as the threshold is usually set to be lenient. Thus the detection cascade can very quickly consider a region of an image as not being a face with very high certainty. The only time that all stages in the cascade need to be considered is when a face is detected, for all other cases, an area is likely to be disregarded very early on in the tree if it is a negative. This allows the algorithm to choose whether to spend more time further scanning a sub-section of an image or to move on, with just a few arithmetic operations (the calculation of features will be described later on).

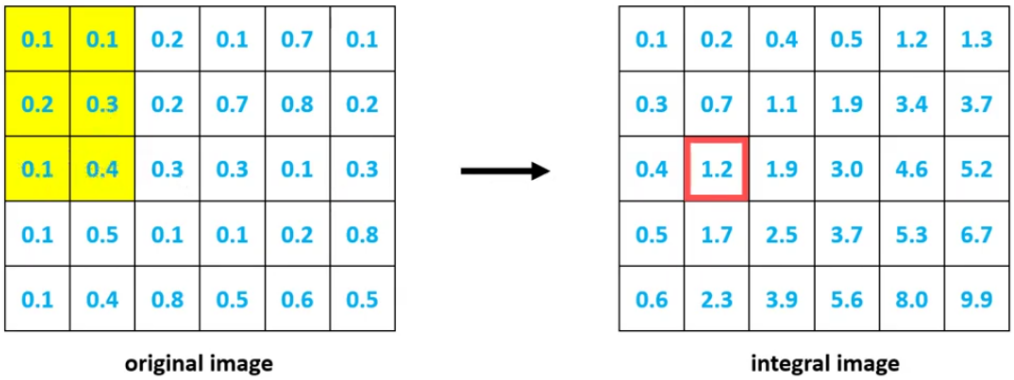

Figure 4: Original Image to Integral Image

The process of calculating these features is conceptually a very easy task. This simply involves summing all of the pixels in one rectangle, and deducting the sum of all the pixels in the other rectangle.
The time complexity of this operation for an $n \times n$ pixel rectangle, however, is $O(n^2)$ since all pixels in the region need to be added.
Considering that around 6,000 features are typically used on one frame for face detection, and that there are at least 15 frames every second, this computation becomes a problem.

P. Viola and M. Jones solve this problem by applying the **Integral Image**, as an intermediate form of the image. The Integral Image has the same shape as the original image, however each pixel, instead of storing the original pixel value, now stores the sum of all the pixels in the area to its left and above it, as shown in Figure 4.

Since each pixel now contains the sum of the pixels to its left and above it, this turns the computation of the sum of an area of pixels from an $O(n^2)$ operation to an $O(1)$ operation.
For example, in Figure 5, computing the sum of the yellow area can be done using 4 'pixels' from the integral image, i.e. the *3.7*, which contains the pixels to its left and above it, reducing the *1.7* which contains the unwanted pixels to the left and the *0.5* which contains the unwanted pixels above, and adding the *0.2* to make up for the pixels that were deducted twice.

For a 1000x1000 pixel area, computation using summation on the original image would take a million addition operations and array accesses, using the integral image this would take just 4 array accesses and 3 arithmetic operations. The creation of the integral image does of course take time, but

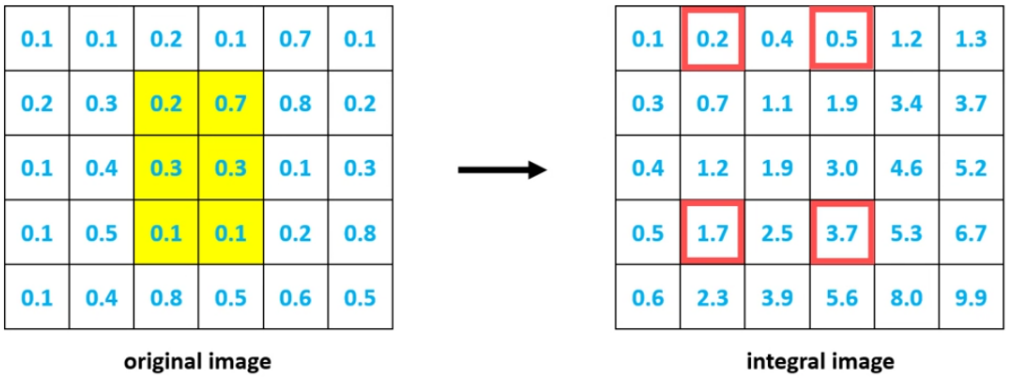

Figure 5: Computing the Sum of Pixels in an Integral Image

considering that 6,000 features will likely be calculated on the image, it is much more sensible to run an $O(n^2)$ operation once followed by 6,000 $O(1)$ operations rather than 6,000 $O(n^2)$ operations.

Artefact 1

To train the cascade, a large number of examples (images) of the object needing to be detected, positive images, and images where the object to be detected is not present, negative or background images, are required.

The positive training images, pictures of faces in this case, were obtained from [5a]. This data set contains 202,599 cropped images of faces of celebrities. This dataset was particularly appealing as it had a very good number of images (more than 200,000), all images were cropped to the celebrities' heads, and there is a very good mix of different facial features, ages, ethnicities, lighting, angles and poses.

The negative images were obtained from various different sources[6a-g], providing the dataset with negative images from many different domains. The most important feature of a negative image is, of course, that it does not contain the object to be detected. In this case, all of these image data sets were chosen as they definitely do not contain any people in them (home objects, dogs, flowers, food, cars, aeroplanes etc.)

A python file named *TrainingHaar.py* was created, which contains a number of functions that are needed to create the necessary inputs to the OpenCV 'traincascade' function.

A function was written named *normalise_neg_images()* which iterates through every individual image in every downloaded dataset, loads the image to memory in grayscale, resizes the image to 250x250, and saves the modified image to disk in the *neg* directory, with an arbitrary name (an integer which increments with each image).

When that process is done, a function named *create_bg_txt()* creates the *bg.txt* file, which is a parameter to the *OpenCV traincascade* program. The function goes through all the normalised images in the *neg* directory, and for each image, inserts a line at the end of the bg.txt file with the relative path of the individual negative image, such as 'neg/1.jpg'. Special attention must be paid that Unix line endings are used for Open CV to read the files correctly.

A similar function, *create_info_txt()*, creates the *info.txt* file. This file is very similar to the bg.txt file, however apart from the relative path, the number of objects in the image is stated, along with the bounding rectangle for each object. In the CelebA dataset being used in this assignment, every positive image contains exactly one face. Furthermore, all the images are 178x218 and the images are cropped to contain just the face, thus the bounding rectangle will be the entire image. Thus, the info.txt file will have entries as follows: 'pos/000001.jpg 1 0 0 178 218', where 1 is the number of faces, 0 0 are the x y start of the bounding rectangle and 178 218 are the width and height of the bounding rectangle. All entries will end in 1 0 0 178 218, as explained above.

Once the info.txt file is generated, the vector file, another parameter to the traincascade function, needs to be generated. This can be accomplished by running the following command in a command line.
opencv_createsamples -info info.txt -num 202599 -w 24 -h 24 -vec faces.vec
'-info' specifies the relative path (and name) of the info.txt file containing information about where each positive image is stored, the number of objects in the image and their location. '-num' states the number of (positive) images in the info.txt file, and '-w' and '-h' specify the final dimensions of each training image in the vector file.

This creates a vector representation of the positive training images, which is optimised for the training process.

Once all these are generated, the training process can be started with the command:
opencv_traincascade -data trained_cascade -vec faces.vec -bg bg.txt -numPos 150000 -numNeg 75000 -numStages 15 -w 24 -h 24
'-data' specifies the name of the folder in which the resulting stages and final cascade will be stored. '-vec' denotes the name of the vector file created by the createsamples command. '-bg' specifies the name of the text file containing the path to each negative image (created by my *create_bg_txt()* function). '-numPos' states the number of positive images to be used during training, whereas '-numNeg' refers to the number of negative images to be used. '-numStages' specifies the number of cascade stages to be trained. '-w' and '-h' specify the width and height of the training images respectively, the value of these should be the same as was used in *createsamples*.

When the *traincascade* command is issued, the cascade will start training. The final cascade will be stored in the directory pointed to by the '-data' parameter, in this case the *trained_cascade* folder, and will be named 'cascade.xml'. In the same folder, there will be a file for each stage of the cascade, this allows training to either resume from a certain stage (keeping all other parameters constant), or for a cascade with x stages to be instantly generated from a cascade with y stages (where $x < y$).
Training in this case, for 150,000 24x24 positive images, 75,000 negative images and 15 stages took about 48 hours.

How a generated 'cascade.xml' file can be used will be discussed in the following section.

All information in this section was adapted from: [7] and [8].

Artefact 2

The Python file *PreTrainedCascade.py* contains the general process of applying a cascade to an image in a couple of different applications, these being finding faces and eyes in a static image, either one entered by the user or a

random one chosen by the system, or detecting faces and eyes from a webcam stream.

The general process is as follows:
First the cascades are loaded using the **cv2.CascadeClassifier** command, as follows:
haar_eye_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_eye.xml')

Next, the image in which faces and eyes will be detected is loaded using **cv2.imread**, with the path of the image provided as a parameter.
Then, the image is converted to grayscale using **cv2.cvtColor**, with the original image and 'cv2.COLOR_BGR2GRAY' as parameters.

Next, faces are detected in the image using the cascade that was loaded earlier and the **detectMultiScale** command, with the grayscale image as a parameter.
This command can also receive optional parameters such as 'scaleFactor', useful since during training faces are of a fixed size (eg. 24x24) and faces in images may be larger, with this parameter images get rescaled to make larger faces detectable by the cascade
and 'minNeighbors' which specifies how many neighbouring rectangles each candidate rectangle should have; higher values results in less false positives but possibly higher false negatives.

When a face is detected by the detectMultiScale command, it is stored in the *faces* array. Each entry in the faces array stores the bottom left x,y co ordinates and the width and height of the face. A loop then goes over every entry in the *faces* array, and draws a green rectangle over the specified coordinates on the original coloured image.

After the green rectangle is drawn, the region of interest of the image is calculated, i.e. the area of the image in which the face was found is stored temporarily as a 'sub-image'. The eye cascade is then run on this 'sub-image' to find eyes within the face. Similarly to the face cascade, it stores the bottom left x, y coordinates and width and height of each eye in the *eyes* array. A nested loop goes through the *eyes* array and draws a red rectangle over each detected eye at the specified coordinates on the coloured image.

When this is done, the image with the green (face) and red (eyes) rectangles is displayed to the user, who can then press any button to exit.

If instead of images, a camera stream or a video is the source, the same process is applied, only that the video is taken on a frame by frame basis. Viola-Jones is perfect for such real-time applications as it is very lightweight and optimised, as described in a previous section.

The described process performs detection using pre-trained cascades. To perform detection with a cascade that I trained, the same process can be used, only instead specifying a cascade that I trained rather than a pre-trained one in the **cv2.CascadeClassifier** command.

All Pre-Trained HaarCascades were obtained from [9].
All information in this section was adapted from: [10], [8] and [11].

Comparison to Alternative Algorithm

One alternative algorithm to the Viola-Jones algorithm for real-time face detection is Local Binary Patterns (LBP).

LBPs scan a grayscale image in 3x3 pixel blocks.
For each 3x3 pixel block, the intensity of the centre pixel is compared to that of the surrounding 8 pixels. If the intensity of a surrounding pixel is greater or equal to that of the centre pixel, that surrounding pixel is assigned a **1**, if it is lower it is assigned a **0**.
These 8 one-bit values are then converted into a single byte, starting from the top left bit and moving clockwise (any order can be used as long as it is consistent throughout the whole image). This byte now represents a decimal number in the encoding.
(This description assumes LBP with radius = 1 and neighbours = 8, LBPs can have different radii or number of neighbours, this is simply one way it can be done)
This process is explained graphically in Figure 6.

The 3x3 binary window obtained after thresholding represents the edges in the image. Every time consecutive bits in the string of bits change from 0

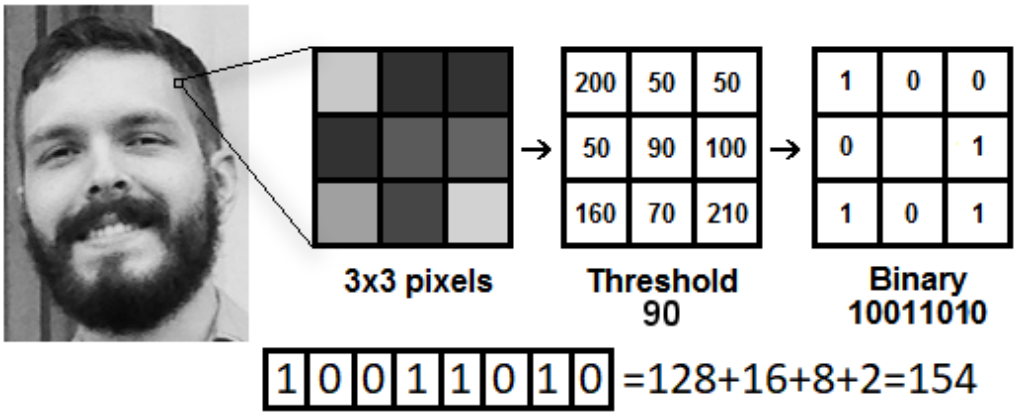

Figure 6: Explanation of how an image is scanned by LBP
Adapted from: `https://github.com/kelvins/lbph`

to 1 or 1 to 0, indicates that there is a transition from a dark area to a light area, i.e. there is an edge.

This thresholding process is applied to every pixel in the image, i.e. every pixel is compared to the 8 surrounding pixels, and is then substituted by the resulting decimal number in the new encoding of the image.
When the encoding is finished, a histogram is created, with the decimal value on the x-axis (0-255), and the number of occurrences of that decimal value in the image on the y-axis.
The histogram is essentially a 'feature descriptor' of the original image.

Images can also be divided into 'sections', and a histogram will be generated for each of these sections. The individual histograms will be concatenated, resulting in a larger histogram to compare. This can often result in higher detection rates.

During training, the algorithm will extract the histogram of a large number of images containing faces (positives) and images not containing faces (negatives). This allows the algorithm to learn which values are most relevant for detecting faces and which are 'noise'. This will also produce a model histogram, to which histograms will be compared when the algorithm is run for detection.

During detection, the histogram of the image will be calculated, and its dis-

tance will be calculated from the learned histogram according to some distance metric (eg. Euclidean Distance). The shorter the distance, the higher the confidence that a face has been detected.

LBPs have been shown to have around a 5% higher correct rate than Haar Features. LBPs also take significantly less time to train than Haar features, due to their small feature set.
Furthermore, LBPs are "illumination invariant", i.e. if the entire image is brighter or darker, faces can be detected just as well with the same process, as the intensity of the centre pixel (threshold) will increase by as much as the surrounding pixels, so the resulting byte will be identical.
Another feature of LBP is that not only can it detect faces, but it has proved capable of recognising one face from another, simply by looking for different features in the histogram.

All information in this section was obtained from: [12], [13], [14], [15] and [16].
[16] is one instance where this algorithm is implemented in a real-time system, showing that it is, in fact, suitable for real-time applications.

Experiments and their Evaluation

In order to evaluate the cascades' performance, the python file **Evaluation.py** was created.

This script first creates a DataFrame to store the results of evaluation, with a row for each cascade, and a column for the 'true positives', 'true negatives', 'false positives', 'false negatives', 'precision', 'recall' and 'FMeasure' associated with each cascade.

Then, 2 pre-trained frontal face cascades are loaded to serve as a benchmark, along with several cascades that I trained.

The relative path of the evaluation dataset is specified. This dataset was obtained from the 'person' folder from the Natural Images dataset. These images were, of course, not used during training.[5b]

The directory of the evaluation images is specified, in this case the folder 'evaluation'. A loop then iterates over every image in the specified directory and loads the image to memory in grayscale.
A nested loop then iterates over every cascade, and detects the faces in the current image using the current cascade.
All the evaluation images contain just one face, thus if one face is detected, it is considered a true positive, any more faces are considered false positives. If no faces are detected, it is considered a false negative.
When a result for a cascade is computed, it is stored in the 'results' DataFrame, in the cascade's respective row under the relevant column.

One fault with this approach is that just because a face is found in an image that is known to contain one face, it doesn't necessarily mean that the correct face was found, for example the face might not have been detected and the shoulder was detected as a face. With this evaluation method, this would be marked as a true positive (i.e. instead of a true positive it should be marked as a false positive and a false negative).
I believe this is a reasonable assumption however, as with Viola-Jones it is rather unlikely for false negatives to occur, given the way the detection cascade is constructed.

Once the *true positives*, *true negatives*, *false positives* and *false negatives* have been computed for each cascade, the *precision*, *recall* and *F-Measure* are also computed for each cascade. Having such metrics helps evaluate the performance of each individual cascade more effectively.

The findings of the evaluation will be discussed in the next section.

The cascades were also qualitatively tested by loading them instead of the pre-trained cascades in *PreTrainedCascade.py*, and manually checking how well each could detect my face. All models could detect my face decently well, only some detected more false positives than others, whereas others didn't detect my face at certain angles where others could. These are discussed in the next section.

Overall Conclusions

	true positives	true negatives	false positives	false negatives	precision	recall	FMeasure
pre-trained_default	978	0	9	8	0.990881	0.991886	0.991384
pre-trained_alt	980	0	3	6	0.996948	0.993915	0.995429
pos150k_neg75k_stages8	984	0	2343	2	0.295762	0.997972	0.456295
pos150k_neg75k_stages10	745	0	518	241	0.589865	0.755578	0.662517
pos150k_neg75k_stages12	218	0	28	768	0.886179	0.221095	0.353896
pos150k_neg75k_stages15	9	0	0	977	1.000000	0.009128	0.018090
pos75k_neg150k_stages8	984	0	2450	2	0.286546	0.997972	0.445249
pos75k_neg150k_stages10	723	0	493	263	0.594572	0.733266	0.656676
pos75k_neg150k_stages12	199	0	26	787	0.884444	0.201826	0.328654
pos75k_neg150k_stages15	8	0	0	978	1.000000	0.008114	0.016097
pos150k_neg150k_stages10	654	0	316	332	0.674227	0.663286	0.668712
pos50k_neg100k_stages10	770	0	550	216	0.583333	0.780933	0.667823
pos25k_neg50k_stages10	601	0	321	385	0.651844	0.609533	0.629979

Figure 7: Cascade Evaluation Results

Figure 7 shows the results of the evaluation procedure described in the previous section.

The pre-trained cascades that were used as a benchmark performed excellently, both reporting over 99% precision, recall and F-Measure.

Next, the cascades trained with 150k positive images, 75k negative images (2:1) and 8, 10, 12 and 15 stages was evaluated.
The cascade with 8 stages had a 99% recall, however just a 30% precision. This suggests that the cascade was too lenient, and was picking up an excess of false positives. 8 stages, for this amount of training images proved to be not enough to fit the data properly.
The cascades with 12 and 15 stages had excellent precision, however their recall was terrible. This suggests that for this many training examples, 12 stages are far too many, and the cascade has over fit the data.
The sweet-spot seemed to be 10 stages for this many training examples. The cascade obtained 59% precision and 75.6% recall, resulting in an F-Measure of 0.662. This is a rather respectable result for a cascade that took just two days to train on a home computer.

Then, the cascades trained with 75k positive images, 150k negative images (1:2) and 8, 10, 12 and 15 stages was evaluated.
The cascade with 8 stages again performed terribly, as it was under fitting the data and was too lenient and had terrible precision. The cascades with 12 and 15 stages were over fitting the data, and had unsatisfactory recall.
The cascade with 10 stages was again the sweet-spot. The cascade obtained

59.4% precision and 73.3% recall, resulting in an F-Measure of 0.657. This again is a a rather respectable performance.

At this point it was evident that for this amount of training data, 10 stages was the right choice.
The impact on performance that the ratio of positive to negative examples wasn't very clear, however it seems that 2 pos : 1 neg results in better recall, whereas 1 pos : 2 neg results in higher precision. Thus the choice of what ratio to use is application dependent.

The cascade with 150k positive : 150k negative images and 10 stages was evaluated next.
This cascade had a relatively high precision of 67.4%, and a decent recall of 66.3%. The F-Measure obtained by this cascade was of 0.669, the highest of all the cascades I trained. This cascade also has precision and recall levels that are very close to each other, with no imbalance in their scores. To put things into perspective though, this model was trained with 300k images, whereas the others used 3/4 of that (225k images).

To observe how the performance of the cascade changes as less training data is provided, two cascades were trained with 50k pos : 100k neg and 25k pos : 50k neg images, both with 10 stages.
The former maintained a very good 58.3% precision and a 78% recall, with an F-Measure of 0.668. This score arguably beats that obtained when using 150k pos:75k neg or 75k pos:150k neg and 10 stages.
The latter surprisingly saw a precision of 65.2% and a recall of 61%, with an F-Measure of 0.630.

It would be interesting to experiment not only with different positive image sizes (20x20, 24x24 etc.), pos:neg ratios or number of stages, but also with the parameters of the **detectMultiScale** command. The *scaleFactor* parameter can be adjusted to find different sized objects than what the cascade was trained for, or the *minNeighbors* parameter can be adjusted to change how many neighbouring rectangles each candidate rectangle should have, higher values may lead to less false positives but possibly more false negatives.

The dataset used for the positive images, while containing a plethora of diverse cropped and aligned images, was cropped slightly large, meaning that the positive training images didn't contain just a face, rather the entire

head. This might have thrown the cascade slightly off at times.

All in all, it is impressive how such an algorithm has stood the test of time, and is still being used in modern applications today, for example to locate a face which will be used for facial recognition with high efficiency, rather than scanning an entire image.

References

[1] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001, Kauai, HI, USA, 2001, pp. I-I. doi: 10.1109/CVPR.2001.990517

[2] M. Pound, "Detecting Faces (Viola Jones Algorithm)", YouTube - Computerphile, Oct 2018, url: https://youtu.be/uEJ71VlUmMQ

[3] B. Holczer, "Computer Vision - Haar-Features", YouTube - Balazs Holczer, Feb 2018, url: https://youtu.be/F5rysk51txQ

[4] B. Holczer, "Computer Vision - Integral Images", YouTube - Balazs Holczer, Feb 2018, url: https://youtu.be/x41KFOFGnUE

[5] Positive image datasets:

(a) "*Large-scale CelebFaces Attributes (CelebA) Dataset*", MMLAB, `http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html`

(b) 'person' folder of "Natural Images" dataset, Kaggle, `https://www.kaggle.com/prasunroy/natural-images`

[6] Negative image datasets:

(a) "Home Objects dataset", Caltech, `http://www.vision.caltech.edu/pmoreels/Datasets/Home_Objects_06/`

(b) "Face Negatives", originally from `http://face.urtho.net/`, however were obtained from `https://github.com/handaga/tutorial-haartraining/tree/master/data/negatives`

(c) "Dog Breed Identification", Kaggle, `https://www.kaggle.com/c/dog-breed-identification/data`

(d) "Flowers Recognition", Kaggle, `https://www.kaggle.com/alxmamaev/flowers-recognition`

(e) "Food Images (Food-101)", Kaggle, https://www.kaggle.com/kmader/food41#images.zip

(f) "Stanford Cars Dataset", Kaggle, https://www.kaggle.com/jessicali9530/stanford-cars-dataset

(g) "Natural Images", Kaggle, https://www.kaggle.com/prasunroy/natural-images (all folders except 'person' folder)

[7] "Cascade Classifier Training", OpenCV Documentation, https://docs.opencv.org/3.4/dc/d88/tutorial_traincascade.html

[8] "Creating your own Haar Cascade OpenCV Python Tutorial", pythonprogramming.net, https://pythonprogramming.net/haar-cascade-object-detection-python-opencv-tut

[9] "Pre-Trained HaarCascades", GitHub - opencv, https://github.com/opencv/opencv/tree/master/data/haarcascades

[10] "Face Detection using OpenCV and Python: A Beginners Guide", R. Raja, SuperDataScience, https://www.superdatascience.com/opencv-face-detection/

[11] "CascadeClassifier::detectMultiScale", OpenCV Documentation, https://www.docs.opencv.org/2.4/modules/objdetect/doc/cascade_classification.html

[12] T. Ahonen, A. Hadid and M. Pietikainen, "Face description with local binary patterns: Application to face recognition". IEEE Transactions on Pattern Analysis & Machine Intelligence, (12), pp.2037-2041, 2006.

[13] M. Valstar, "Faces & the Local Binary Pattern", YouTube - Computerphile, Oct 2015, https://www.youtube.com/watch?v=wpAwdsubl1w

[14] J. Chang-Yeon, "Face Detection using LBP features," 2008.

[15] L. Zhang, R. Chu, S. Xiang, S. Liao, and S. Z. Li, "Face detection based on multi-block lbp representation," in Advances in Biometrics (S.-W. Lee and S. Z. Li, eds.), (Berlin, Heidelberg), pp. 11-18, Springer Berlin Heidelberg, 2007.

[16] I. Das, I. Gangopadhyay and A. Chatterjee, "FACE DETECTION AND RECOGNITION USING HAAR CLASSIFIER AND LBP HISTOGRAM," International Journal of Advanced Research in Computer Science, vol. 9, (2), pp. 592-598, 2018.