Face Detection Using OpenCV and Haar Cascades Classifiers

Presentation · March 2020

DOI: 10.13140/RG.2.2.26708.83840

CITATIONS
0

READS
2,688

2 authors:

Sidra Mehtab
NSHM Knowledge Campus
59 PUBLICATIONS **124** CITATIONS
SEE PROFILE

Jaydip Sen
Praxis Business School
327 PUBLICATIONS **3,585** CITATIONS
SEE PROFILE

Face Detection Using OpenCV and Haar Cascades Classifiers

Master of Science (Data Science & Analytics) Batch 2018 – 2020
Minor Project Presentation

By
Sidra Mehtab
(Reg. No: 182341810028)
Under the Supervision of Prof. Jaydip Sen

NSHM Knowledge Campus, Kolkata, INDIA
Affiliated to
Maulana Abul Kalam Azad University of Technology, Kolkata, INDIA

Objective of the Work

- The primary objective of this work is to develop a framework for face detection using Haar Feature-based Cascade Classifiers.

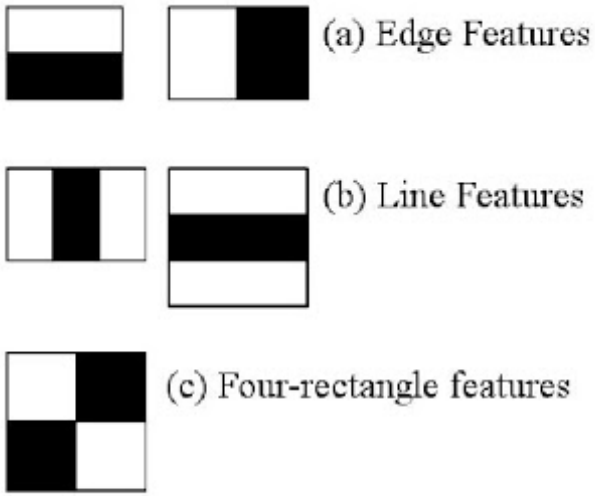
- Based on the framework developed, it will be extended for the detection of other features of a human face like eyes and nose.

Outline

- Fundamentals
- Cascades of classifiers
- Face detection in an image
- Eyes detection in a face
- Nose detection in a face
- Results
  - Face detection
  - Eyes detection
  - Nose detection
- Conclusion
- References

Fundamentals

- Object detection using Haar feature-based cascade classifiers is an effective object detection method proposed by Paul Viola and Michael Jones in their seminal paper titled “Rapid Object Detection using a Boosted Cascade of Simple Features” in 2001. It is a machine learning-based approach where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images.
- The detection algorithm needs a lot of positive images (images of faces) and negative images (images without faces) to train the classifier. Then we need to extract features from it. For this, Haar features shown in the images are used.
- They are just like our convolutional kernel. Each feature is a single value obtained by subtracting the sum of pixels under the white rectangle from the sum of pixels under the black rectangle.

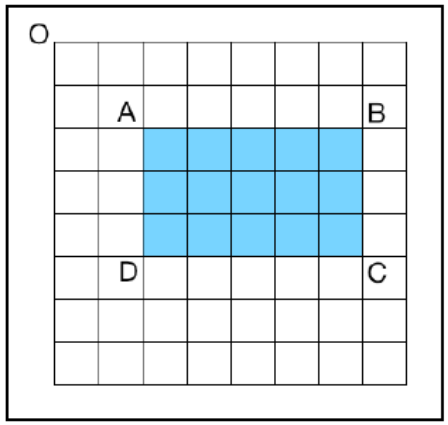


Fundamentals (contd…)

- Now all possible sizes and locations of each kernel are used to calculate plenty of features. Even a 24*24 window results in over 160000 features.
- For each feature calculation, we need to find the sum of pixels under white and black rectangles. To solve this problem, Viola and Jones introduced the concept of “integral images.” It simplifies the calculation of the sum of pixels, however large may be the number of pixels, to an operation involving just four pixels. This makes the computation fast.
- Consider the following image. If we want to compute the sum of the rectangle ABCD, we don’t need to go through each pixel in that rectangular area.
- Let’s say OC indicates the area of the rectangle formed by the top left corner O and the point C on the diagonally opposite corners of the rectangle. To calculate the are of the rectangle ABCD, we can use the following:

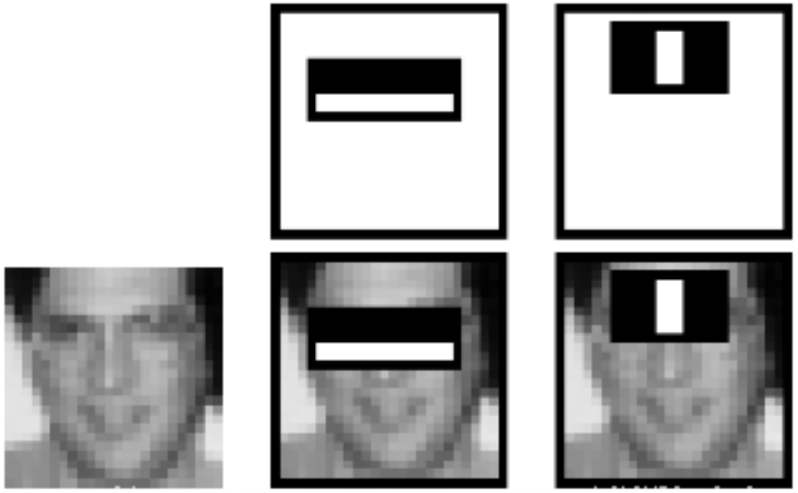
Area of the rectangle ABCD = OC –(OB + OD – OA)

- We don’t have to iterate through anything or re-compute any rectangle areas. All the values on the right-hand side of the equation are already available as they were computed during the earlier cycles.



Fundamentals (contd…)

- Most of the features extracted, however, are irrelevant.
- In the image below, the top row shows two good features. The first feature selected seems to focus on the property that the region of the eyes is often darker than the region of the nose and cheeks. The second feature selected relies on the property that the eyes are darker than the bridge of the nose. However, the same windows applying on cheeks or any other place is irrelevant.
- How do we select the best features out of 160000+ features? It is achieved by Adaboost.



Fundamentals (contd…)

- We apply each feature on all the training images. For each feature, it finds the best threshold which will classify the faces to positive and negative. However, there will be misclassifications too.
- We select the features with a minimum error rate, which means they are the features that best classifies the face and non-face images.
- The above process is carried out as follows. Each image is given an equal weight at the start. After each classification step, weights of misclassified images are increased. Then again the same process is repeated, new error rates and new weights are computed. The process is continued until the required accuracy or error rate is achieved or the required number of features is found.
- The final classifier is a weighted sum of these weak classifiers. The classifiers are called weak because they individually cannot classify the image but together can form a very powerful classifier.
- The original work of Viola and Jones involved 200 features and that yielded 95% accuracy. The final setup had around 6000 features (a reduction from 160000+ features)

Fundamentals (contd…)

- So now we take an image. Take each 24*24 window. Apply 6000 features to it. Then, check if it is a face or not. Of course, a very inefficient and time-consuming proposition. However, Viola and Jones had a smarter proposition.
- In an image, most of the image region is a non-face region. Hence, it is a better idea to have a simple method to check if a window is not a face region. If it is not, we discard it straight away.
- In order to implement the above concept, Viola and Jones introduced the concept of **Cascade of Classifiers**.
- Instead of applying all the 6000 features on a window, we group the features into different stages of classifiers and apply one-by-one. Usually, the first few stages will contain very few number of features.
- If a window fails in the first stage, we discard it and don't consider remaining features on it.
- If a window passes the first stage, we apply the second stage of features and continue with the process.
- A window that passes all stages is a face region.
- The detector of Viola and Jones has 6000+ features with 38 stages with 1, 10, 25, 15, and 50 features in the first five stages. On average, 10 features out of 6000+ are evaluated per sub-window.

Cascade of Classifiers

- OpenCV comes with a lot of pre-trained classifiers. For instance, there are classifiers for smile, eyes, face, nose, etc.
- These come in the form of XML files and are located in the location: opencv/data/haarcascades/ folder.
- We have downloaded the XML files from this link and placed them in the "data" folder in the same working directory as the "jupyter" notebook.
- For "Face detection" we have used the "detectMultiScale" module of the classifier. The function returns a rectangle with coordinates (x, y, w, h) around the detected face.
- The function has two important parameters that have to be tuned according to the data – "scaleFactor" and "minNeighbors". We will discuss their respective roles when we discuss our implementation.

Face Detection in an Image

- We first loaded the Haar cascade file of the frontal face using the CascadeClassifier method of cv2 module using the following command:

```
face_cascade  = cv2.CascadeClassifier('haar_cascade_files/Haarcascade_frontalface_default.xml')
```

- Next, we read the RGB image "kids.jpeg" in which faces are to be detected using the imread function of the cv2 module.

```
img = cv2.imread("kids.jpeg", 1)
```

- Next, we converted the RGB image into a grayscale image using the function cvtColor defined in the module cv2. The function cvtColor needs two mandatory parameters "src" – the image whose color space is to be changed, and "code" – the color space conversion code. We have used cv2.COLOR_BGR2GRAY, as we need to convert the source BGR image into a grayscale image.

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

- Next, we run the face detector on the grayscale image using the following line with "scaleFactor" value 1.3 (default value) and the minNeighbors value 20.

```
faces = face_cascade.detectMultiScale(gray, 1.3, 20)
```

Face Detection in an Image (contd…)

- The first parameter “scaleFactor” of the detectMultiScale function determines a trade-off between detection accuracy and speed of detection. The detection window starts at size “minSize”, and after testing all windows of that size, the window is scaled up the “scaleFactor”, and re-tested, and so on until the window reaches or exceeds the “maxSize”. If the “scaleFactor” is large, (e.g., 2.0), there will be fewer steps, so detection will be faster, but we may miss objects whose size is between two tested scales. We usually don’t change the default value, which is 1.3.

- The second parameter known as the “minNeighbors” is changed based on the requirements. The higher the values of the second parameter(minNeighbors), less will be the number of false positives and less error will be in terms of false detection of faces. However, there is a chance of missing some unclear face traces as well.

- Next, we iterate through the detected faces and draw rectangles around them, so that the faces are clearly identified and the detected faces are displayed.

```
for (x, y, w, h) in faces:
    cv2.rectangle(img, (x, y), (x+w, y+h), (50, 255, 255), 4)
# Display the output
cv2.imshow('img', img)
```

Eyes Detection in a Face

- We first loaded the Haar cascade files of “frontal face” and “eye” using the CascadeClassifier method of the cv2 module using the following command:

```
face_cascade  = cv2.CascadeClassifier(‘haar_cascade_files/Haarcascade_frontalface_default.xml’)

eye_cascade = cv2.CascadeClassifier('haar_cascade_files/haarcascade_eye.xml')
```

- Next, we read the RGB image “kids.jpeg” in which faces are to be detected using the imread function of cv2 module.

```
img = cv2.imread(“kids.jpeg”, 1)
```

- Next, we converted the RGB image into a grayscale image using the function cvtColor defined in the module cv2. The function cvtColor needs two mandatory parameters “src” – the image whose color space is to be changed, and “code” – the color space conversion code. We have used cv2.COLOR_BGR2GRAY, as we need to convert the source BGR image into a grayscale image.

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

- Next, we run the face detector on the grayscale image using the following line with “scaleFactor” value 1.3 (default value) and the minNeighbors value 20 in an outer “for loop”.

```
faces = face_cascade.detectMultiScale(gray, 1.3, 20)
```

Eyes Detection in a Face (contd…)

- In an inner “for” loop we detected the eyes for each “face” detected in the outer “for” loop.

```
cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,255,0),4)
```

- The code of the two for loops together is as follows:

```
faces = face_cascade.detectMultiScale(gray, 1.3, 20)
for (x,y,w,h) in faces:
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
    roi_gray = gray[y:y+h, x:x+w]
    roi_color = img[y:y+h, x:x+w]
    # noses = nose_cascade.detectMultiScale(roi_gray)
    eyes = eye_cascade.detectMultiScale(roi_gray)
    for (ex,ey,ew,eh) in eyes:
        cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,255,0),4)
cv2.imshow('img',img)
```

Nose Detection in a Face

- We first loaded the Haar cascade files of frontal face and nose using the CascadeClassifier method of the cv2 module using the following command:

```
face_cascade  = cv2.CascadeClassifier('haar_cascade_files/Haarcascade_frontalface_default.xml')

nose_cascade = cv2.CascadeClassifier('haar_cascade_files/haarcascade_mcs_nose.xml')
```

- Next, we read the RGB image "kids.jpeg" in which faces are to be detected using the imread function of cv2 module.

```
img = cv2.imread("kids.jpeg", 1)
```

- Next, we converted the RGB image into a grayscale image using the function cvtColor defined in the module cv2. The function cvtColor needs two mandatory parameters "src" – the image whose color space is to be changed, and "code" – the color space conversion code. We have used cv2.COLOR_BGR2GRAY, as we need to convert the source BGR image into a grayscale image.

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

- Next, we run the face detector on the grayscale image using the following line with "scaleFactor" value 1.3 (default value) and the "minNeighbors" value of 20 in an outer "for loop".

```
faces = face_cascade.detectMultiScale(gray, 1.3, 20)
```

Nose Detection in a Face (contd…)

- In an inner “for” loop we detected the nose for each “face” detected in the outer “for” loop.

```
cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,255,0),2)
```

- The code of the two “for loops” together is as follows:

```
faces = face_cascade.detectMultiScale(gray, 1.3, minNeighbors=20)
for (x,y,w,h) in faces:
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
    roi_gray = gray[y:y+h, x:x+w]
    roi_color = img[y:y+h, x:x+w]
    noses = nose_cascade.detectMultiScale(roi_gray, minNeighbors=10)
    for (ex,ey,ew,eh) in noses:
        cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,255,0),2)
cv2.imshow('Nose Found',img)
```

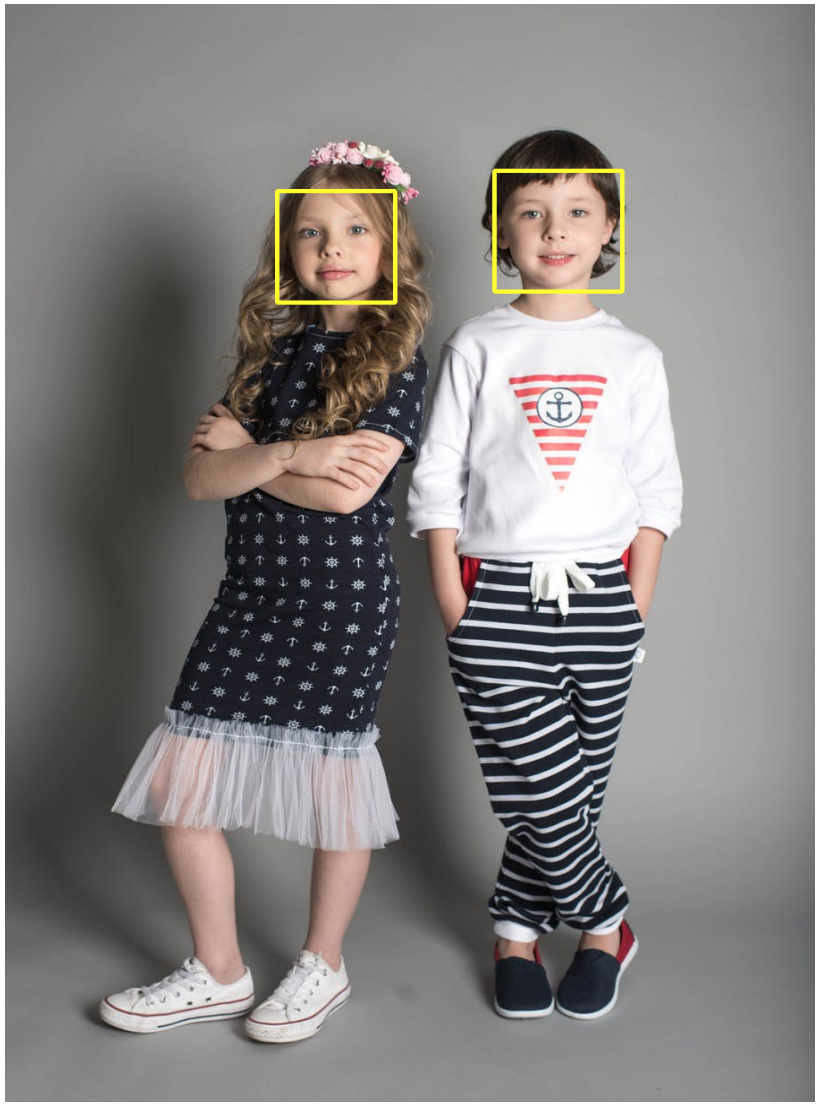
Results : Face Detection

Input Color Image

Results : Face Detection

Converted Grayscale Image

Results : Face Detection



Output Image after Face Detection

Results : Eyes Detection

Output Image after Eyes Detection

Results : Nose Detection

Output Image after Nose Detection

Conclusion

- In this work, we have discussed the concept of face detection using OpenCV in Python using Haar Cascade.
- We used the rich library set of OpenCV for a robust face detection from a sample image. For training the model with the feature set of a face, we used the “Haar frontal face” XML file.
- We later extended our model to detect eyes and nose in the same input image. We used “haar_eyes” and “haar_mcs_nose” XML files for this purpose.
- Our model could successfully detect all faces, eyes, and noses in the input image with 100% detection accuracy and in real-time detection speed.

References

1. Viola, P.. & Jones, M. (2001). 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),* December 8-14, 2001, Kauai, HI, USA.
2. Kirby, M., Sirovich, L. (1990) Application of the Karhunen-Loeve procedure for the characterization of human faces. *IEEE Transaction of Pattern Analysis and Machine Intelligence*, Vol 12, No 1, January 1990., pp. 103 – 108.
3. Liao, S., Jain, A.K., Li, S. Z. (2016). A fast and accurate unconstrained face detector. *IEEE Transaction of Pattern Analysis and Machine Intelligence*, Vol 38, No 2, pp. 211 – 123.
4. Luo, D., Wen, G., Li, D., Hu, Y., and Huna, E. (2018). Deep learning-based face detection using iterative bounding-box regression. *Multimedia Tools Applications*. DOI: https://doi.or/10.1007/s11042-018-56585.
5. Mingxing, J., Junqiang, D., Tao, C., Ning, Y., Yi, J., and Zhen, Z. (2013). An improved detection algorithm of face with combining AdaBoost and SVM. *Proceedings of the 25th Chinese Control and Decision Conference*, pp. 2459-2463.
6. Ren, Z., Yang, S., Zou, F., Yang, F., Luan, C., and Li, K. (2017). A face tracking framework based on convolutional neural networks and Kalman filter. *Proceedings of the 8th IEEE International Conference on Software Engineering and Services Science*, pp. 410-413.
7. Zhang, H., Xie, Y., Xu, C. (2011). A classifier training method for face detection based on AdaBoost. *Proceedings of the International Conference on Transportation, Mechanical, and Electrical Engineering*, pp. 731-734.
8. Zou, L., Kamata, S. (2010). Face detection in color images based on skin color models. *Proceedings of IEEE Region 10 Conferences* , pp. 681-686.

Thank You!

Questions?

email: smehtab@acm.org