DETECTING AND TRACKING FACES USING OPENCV

Preface

Computer vision is found everywhere in modern technology. OpenCV for Python enables us to run computer vision algorithms in real time. With the advent of powerful machines, we are getting more processing power to work with. Using this technology, we can seamlessly integrate our computer vision applications into the cloud. Web developers can develop complex applications without having to reinvent the wheel
The following document provides a detailed and easy to understand, explanation and implementation of Using Haar cascades to Detect and track faces.

Terminology

There are terms that are frequently used throughout this paper that need to be clarified

Haar cascades :When we say Haar cascades, we are actually talking about cascade classifiers based on Haar features.

OpenCV :OpenCV provides a nice face detection framework. We just need to load the cascade file and use it to detect the faces in an image

opencv Brief History

OpenCV (Open Source Computer Vision) is a library of programming functions mainly aimed at realtime computer vision , originally developed by Intel research center in Nizhny Novgorod (Russia), later supported by Willow Garage and now maintained by Itseez. The library is crossplatform and free for use under the opensource BSD license .

opencv has C++, C, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and Mac OS. OpenCV leans mostly towards realtime vision applications and takes advantage of MMX and SSE instructions when available.

Using Haar cascades to detect things

When we say Haar cascades, we are actually talking about cascade classifiers based on Haar features. To understand what this means, we need to take a step back and understand why we need this in the first place. Back in 2001, Paul Viola and Michael Jones came up with a very effective object detection method in their seminal paper. It has become one of the major landmarks in the field of machine learning.

In their paper, they have described a machine learning technique where a boosted cascade of simple classifiers is used to get an overall classifier that performs really well. This way,we can circumvent the process of building a single complex classifier that performs with high accuracy. The reason this is so amazing is because building a robust single-step classifier is a computationally intensive process. Besides, we need a lot of training data to build such a classifier. The model ends up becoming complex and the performance might not be up to the mark.

Let's say we want to detect an object like, say, a pineapple. To solve this, we need to build a machine learning system that will learn what a pineapple looks like. It should be able to tell us if an unknown image contains a pineapple or not. To achieve something like this, we need to train our system. In the realm of machine learning, we have a lot of methods available to train a system. It's a lot like training a dog, except that it won't fetch the ball for you! To train our system, we take a lot of pineapple and non-pineapple images, and then feed them into the system. Here, pineapple images are called positive images and the non-pineapple images are called negative images

As far as the training is concerned, there are a lot of routes available. But all the traditional techniques are computationally intensive and result in complex models. We cannot use these models to build a real time system. Hence, we need to keep the classifier simple. But if we keep the classifier simple, it will not be accurate. The trade off between speed and accuracy is common in machine learning. We overcome this problem by building a set of simple classifiers and then cascading them together to form a unified classifier that's robust. To make sure that the overall classifier works well, we need to get creative in the cascading step. This is one of the main reasons why the Viola-Jones method is so effective.

Coming to the topic of face detection, let's see how to train a system to detect faces. If we want to build a machine learning system, we first need to extract features from all the images. In our case, the machine learning algorithms will use these features to learn what a face looks like. We use Haar features to build our feature vectors. Haar features are simple summations and differences of patches across the image. We do this at multiple image sizes to make sure our system is scale invariant.

Once we extract these features, we pass it through a cascade of classifiers. We just check all the different rectangular sub-regions and keep discarding the ones that don't have faces in them. This way, we arrive at the final answer quickly to see if a given rectangle contains a face or not.

What are integral images?

If we want to compute Haar features, we will have to compute the summations of many different rectangular regions within the image. If we want to effectively build the feature set, we need to compute these summations at multiple scales. This is a very expensive process! If we want to build a real time system, we cannot spend so many cycles in computing these sums. So we use something called integral images.

To compute the sum of any rectangle in the image, we don’t need to go through all the elements in that rectangular area. Let’s say AP indicates the sum of all the elements in the rectangle formed by the top left point and the point P in the image as the two diagonally opposite corners. So now, if we want to compute the area of the rectangle ABCD, we can use the following formula:

Area of the rectangle ABCD = AC – (AB + AD - AA)

extracting Haar

features includes computing the areas of a large number of rectangles in the image at multiple scales. A lot of those computations are repetitive and the overall process is very slow. In fact, it is so slow that we cannot afford to run anything in real time. That’s the reason we use this formulation! The good thing about this approach is that we don’t have to recalculate anything. All the values for the areas on the right hand side of this equation are already available. So we just use them to compute the area of any given rectangle and extract the features.

Detecting and tracking faces:

OpenCV provides a nice face detection framework. We just need to load the cascade file and use it to detect the faces in an image.

Let’s see how to do it:

```
import cv2

import numpy as np

face_cascade =

cv2.CascadeClassifier('./cascade_files/haarcascade_frontalface_alt.xml')

cap = cv2.VideoCapture(0)

scaling_factor = 0.5

while True:

ret, frame = cap.read()

frame = cv2.resize(frame, None, fx=scaling_factor, fy=scaling_factor,
interpolation=cv2.INTER_AREA)

gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

face_rects = face_cascade.detectMultiScale(gray, 1.3, 5)

for (x,y,w,h) in face_rects:

cv2.rectangle(frame, (x,y), (x+w,y+h), (0,255,0), 3)

cv2.imshow('Face Detector', frame)

c = cv2.waitKey(1)

if c == 27:

break

cap.release()

cv2.destroyAllWindows()
```

Understanding it better

We need a classifier model that can be used to detect the faces in an image. OpenCV provides an xml file that can be used for this purpose. We use the function CascadeClassifier to load the xml file. Once we start capturing the input frames from the webcam, we convert it to grayscale and use the function detectMultiScale to get the bounding boxes for all the faces in the current image. The second argument in this function specifies the jump in the scaling factor. As in, if we don't find an image in the current scale, the next size to check will be, in our case, 1.3 times bigger than the current size. The last parameter is a threshold that specifies the number of adjacent rectangles needed to keep the current rectangle. It can be used to increase the robustness of the face detector.

Fun with faces

Now that we know how to detect and track faces, let’s have some fun with it. When we capture a video stream from the webcam, we can overlay funny masks on top of our faces

Let’s look at the code to see how to overlay the skull mask on top of the face in the input video stream:

```
import cv2

import numpy as np

face_cascade =

cv2.CascadeClassifier('./cascade_files/haarcascade_frontalface_alt.xml')

face_mask = cv2.imread('mask.png')

h_mask, w_mask = face_mask.shape[:2]

if face_cascade.empty():

raise IOError('Unable to load the face cascade classifier xml file')

cap = cv2.VideoCapture(0)

scaling_factor = 0.5

while True:
```

```
ret, frame = cap.read()

frame = cv2.resize(frame, None, fx=scaling_factor, fy=scaling_factor,

interpolation=cv2.INTER_AREA)

gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

face_rects = face_cascade.detectMultiScale(gray, 1.3, 5)

for (x,y,w,h) in face_rects:

if h > 0 and w > 0:

# Adjust the height and weight parameters depending on the

sizes and the locations. You need to play around with these to make sure

you get it right.

h, w = int(1.4*h), int(1.0*w)

y -= 0.1*h

# Extract the region of interest from the image

frame_roi = frame[y:y+h, x:x+w]

face_mask_small = cv2.resize(face_mask, (w, h),

interpolation=cv2.INTER_AREA)

# Convert color image to grayscale and threshold it

gray_mask = cv2.cvtColor(face_mask_small, cv2.COLOR_BGR2GRAY)

ret, mask = cv2.threshold(gray_mask, 180, 255,

cv2.THRESH_BINARY_INV)

# Create an inverse mask

mask_inv = cv2.bitwise_not(mask)
```

```
# Use the mask to extract the face mask region of interest

masked_face = cv2.bitwise_and(face_mask_small, face_mask_small,

mask=mask)

# Use the inverse mask to get the remaining part of the image

masked_frame = cv2.bitwise_and(frame_roi, frame_roi,

mask=mask_inv)

# add the two images to get the final output

frame[y:y+h, x:x+w] = cv2.add(masked_face, masked_frame)

cv2.imshow('Face Detector', frame)

c = cv2.waitKey(1)

if c == 27:

break

cap.release()

cv2.destroyAllWindows()
```

Under the hood

Just like before, we first load the face cascade classifier xml file. The face detection steps work as usual. We start the infinite loop and keep detecting the face in every frame. Once we know where the face is, we need to modify the coordinates a bit to make sure the mask fits properly. This

manipulation process is subjective and depends on the mask in question.

Different masks require different levels of adjustments to make it look more natural. We extract the region-of-interest from the input frame in the following line:

```
frame_roi = frame[y:y+h, x:x+w]
```

Now that we have the required region-of-interest, we need to overlay the mask on top of this. So we resize the input mask to make sure it fits in this region-of-interest. The input mask has a white background. So if we just overlay this on top of the region-of-interest, it will look unnatural because of the white background. We need to overlay only the skullmask pixels and the remaining area should be transparent

So in the next step, we create a mask by thresholding the skull image. Since the background is white, we threshold the image so that any pixel with an intensity value greater than 180 becomes 0, and everything else becomes 255

As far as the frame regionof- interest is concerned, we need to black out everything in this mask region. We can do that by simply using the inverse of the mask we just created. Once we have the masked versions of the skull image and the input region-of-interest, we just add them up to get the final image.

Conclusion :

The above Document provides you with the information of how to use opencv to detect and track faces ,to do fun activity with faces by adding mask to the face when it was detected

References:

http://www.cs.ubc.ca/~lowe/425/slides/13ViolaJones.pdf

Book :OpenCV with Python By Example Prateek Joshi