Raspberry Pi: Facial landmarks + drowsiness detection with OpenCV and dlib

by **Adrian Rosebrock** on October 23, 2017 in **dlib**, **Facial Landmarks**, **Raspberry Pi**

Today's blog post is the long-awaited tutorial on **real-time drowsiness detection on the Raspberry Pi!**

Back in May I wrote a (laptop-based) drowsiness detector that can be used to detect if the driver of a motor vehicle was getting tired and potentially falling asleep at the wheel.

The driver drowsiness detector project was inspired by a conversation I had with my Uncle John, a long haul truck driver who has witnessed a more than a few accidents due to fatigued drivers.

The post was really popular and a lot of readers got value out of it…

…but the method was not optimized for the Raspberry Pi!

Since then readers have been requesting me to write a followup blog post that covers the necessary optimizations to run the drowsiness detector on the Raspberry Pi.

I caught up with my Uncle John a few weeks ago and asked him what he would think of a small computer that could be mounted inside his truck cab to help determine if he was getting tired at the wheel.

He wasn't crazy about the idea of being monitored by a camera his entire work day (and I don't necessarily blame him either — I wouldn't want to be monitored all the time either). But he *did* eventually concede that a device like this, and ideally less invasive, would certainly help avoid accidents due to fatigued drivers.

To learn more about these facial landmark optimizations and how to run our drowsiness detector on the Raspberry Pi, *just keep reading!*

Looking for the source code to this post?
Jump right to the downloads section.

Raspberry Pi: Facial landmarks + drowsiness detection with OpenCV and dlib

Today's tutorial is broken into four parts:

1. Discussing the tradeoffs between Haar cascades and HOG + Linear SVM detectors.
2. Examining the TrafficHAT used to create the alarm that will sound if a driver/user gets tired.
3. Implementing dlib facial landmark optimizations so we can deploy our drowsiness detector to the Raspberry Pi.
4. Viewing the results of our optimized driver drowsiness detection algorithm on the Raspberry Pi.

Before we get started I would ***highly encourage*** you to read through my previous tutorial on *Drowsiness detection with OpenCV*.

While I'll be reviewing the code in its entirety here, you should still read the previous post as I discuss the actual Eye Aspect Ratio (EAR) algorithm in more detail.

The EAR algorithm is responsible for detecting driver drowsiness.

Haar cascades: less accurate, but faster than HOG

The major optimization we need to run our driver drowsiness detection algorithm on the Raspberry Pi is to swap out the default dlib HOG + Linear SVM face detector and replace it with OpenCV's Haar cascade face detector.

While HOG + Linear SVM detectors tend to be significantly more accurate than Haar cascades, the cascade method is also *much* faster than HOG + Linear SVM detection algorithms.

A complete review of both HOG + Linear SVM and Haar cascades work is outside the scope of this blog post, but I would encourage you to:

1. Read this post on *Histogram of Oriented Gradients and Object Detection* where I discuss the pros and cons of HOG + Linear SVM and Haar cascades.
2. Work through the *PyImageSearch Gurus* course where I demonstrate how to implement your own custom HOG + Linear SVM object detectors from scratch.

The Raspberry Pi TrafficHAT

In our previous tutorial on drowsiness detection I used my laptop to execute driver drowsiness detection code — this enabled me to:

1. Ensure the drowsiness detection algorithm would run in real-time due to the faster hardware.
2. Use the laptop speaker to sound an alarm by playing a .WAV file.

The Raspberry Pi does not have a speaker so we cannot play any loud alarms to wake up the driver…

…but the Raspberry Pi is a highly versatile piece of hardware that includes a large array of hardware add-ons.

One of my favorites is the TrafficHAT:

Figure 1: The Raspberry Pi 3 with TrafficHat board containing button, buzzer, and lights.

The TrafficHAT includes:

- Three LED lights
- A button
- A loud buzzer (which we'll be using as our alarm)

This kit is an excellent starting point to get some exposure to GPIO. If you're just getting started as well, be sure to take a look at the TrafficHat.

You don't have to use the TrafficHAT of course; any other piece of hardware that emits a loud noise will do.

Another approach I like to do is just plug a 3.5mm audio cable in the audio jack, and then set up text to speech using `espeak` (a package available via `apt-get`). Using this method you could have your Pi say *"WAKEUP WAKEUP!"* when you're drowsy. I'll leave this as an exercise for you to implement if you so choose.

However, for the sake of this tutorial I will be using the TrafficHAT. **You can buy your own TrafficHAT here**.

And from there you can install the required Python packages you need to use the TrafficHAT via `pip`. But first, ensure you're in your appropriate virtual environment on your Pi. I have a thorough explanation on

virtual environments on this previous post.

Here are the installation steps upon opening a terminal or SSH connection:

```
Raspberry Pi: Facial landmarks + drowsiness
1 $ workon cv
2 $ pip install RPi.GPIO
3 $ pip install gpiozero
```

From there, if you want to check that everything is installed properly in your virtual environment you may run the Python interpreter directly:

```
Raspberry Pi: Facial landmarks + drowsiness
1 $ workon cv
2 $ python
3 >>> import RPi.GPIO
4 >>> import gpiozero
5 >>> import numpy
6 >>> import dlib
7 >>> import cv2
8 >>> import imutils
```

Note: *I've made the assumption that the virtual environment you are using* ***already*** *has the above packages installed in it. My* `cv` *virtual environment has NumPy, dlib, OpenCV, and imutils already installed, so by using* `pip` *to install the* `RPi.GPIO` *and* `gpiozero` *to install the respective GPIO packages, I'm able to access all six libraries from within the same environment. You may* `pip install` *each of the packages (except for OpenCV). To install an optimized OpenCV on your Raspberry Pi, then just follow this previous post. If you are having trouble getting dlib installed, please follow this guide.*

The driver drowsiness detection algorithm is identical to the one we implemented in our previous tutorial.

To start, we will apply OpenCV's Haar cascades to detect the face in an image, which boils down to finding the bounding box *(x, y)*-coordinates of the face in the frame.

Given the bounding box the face we can apply dlib's facial landmark predictor to obtain **68 salient points** used to localize the eyes, eyebrows, nose, mouth, and jawline:

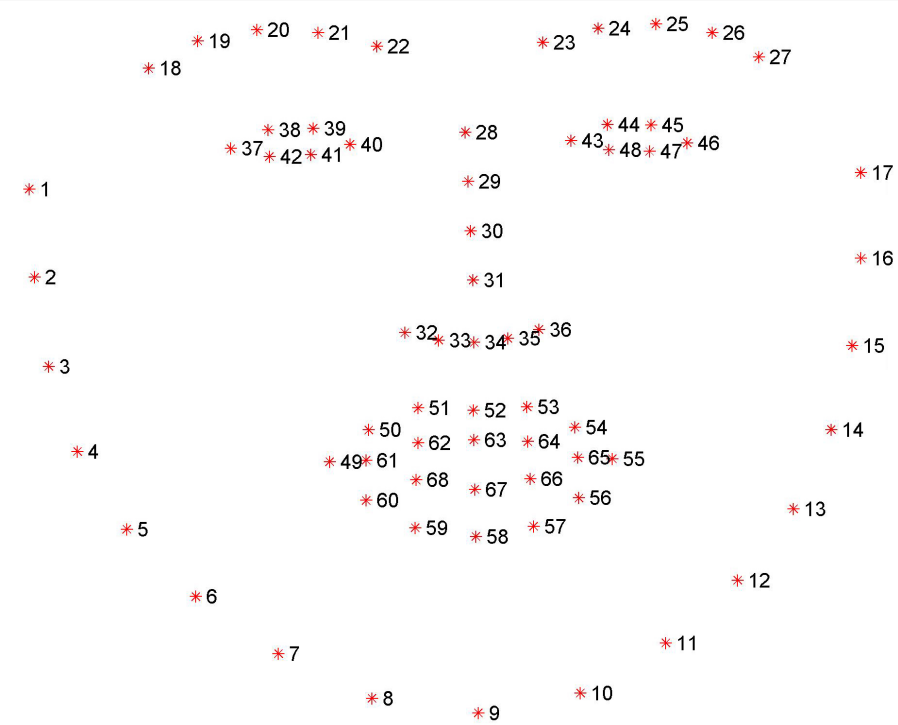

Figure 2: Visualizing the 68 facial landmark coordinates from the iBUG 300-W dataset.

As I discuss in this tutorial, dlib's 68 facial landmarks are *indexable* which enables us to extract the various facial structures using simple Python array slices.

Given the facial landmarks associated with an eye, we can apply the *Eye Aspect Ratio (EAR)* algorithm which was introduced by Soukupová and Čech's in their 2017 paper, *Real-Time Eye Blink Detection suing Facial Landmarks*:

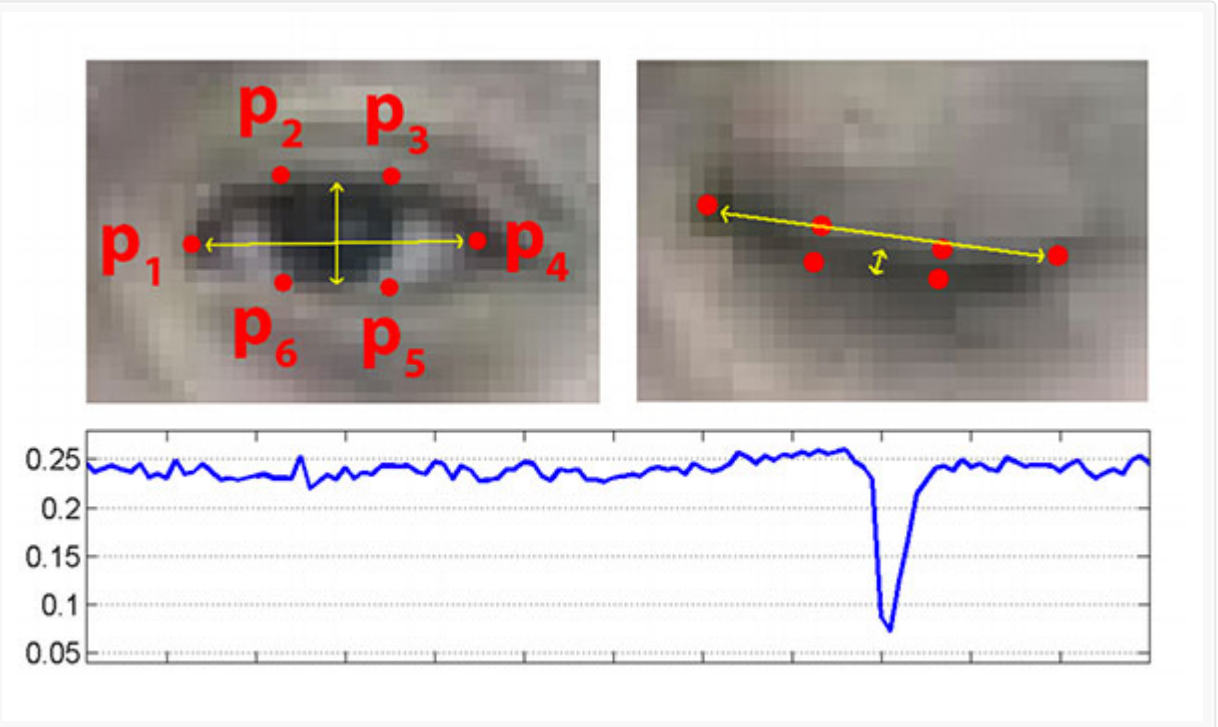

Figure 3: *Top-left:* A visualization of eye landmarks when then the eye is open. *Top-right:*Eye landmarks when the eye is closed. *Bottom:* Plotting the eye aspect ratio over time. The dip in the eye aspect ratio indicates a blink

(Image credit: Figure 1 of Soukupová and Čech).

On the *top-left* we have an eye that is fully open and the eye facial landmarks plotted. Then on the *top-right* we have an eye that is closed. The *bottom* then plots the eye aspect ratio over time. As we can see, the eye aspect ratio is constant (indicating that the eye is open), then rapidly drops to close to zero, then increases again, indicating a blink has taken place.

You can read more about the blink detection algorithm and the eye aspect ratio in this post dedicated to blink detection.

In our drowsiness detector case, we'll be monitoring the eye aspect ratio to see if the value *falls* but *does not increase again*, thus implying that the driver/user has closed their eyes.

Once implemented, our algorithm will start by localizing the facial landmarks on extracting the eye regions:

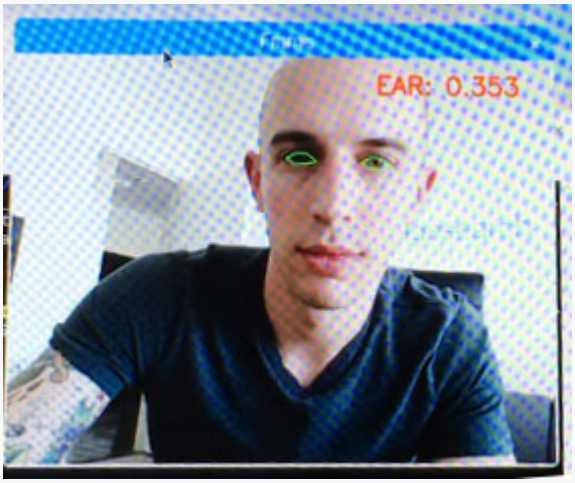

Figure 4: Me with my eyes open — I'm not drowsy, so the Eye Aspect Ratio (EAR) is high.

We can then monitor the eye aspect ratio to determine if the eyes are closed:

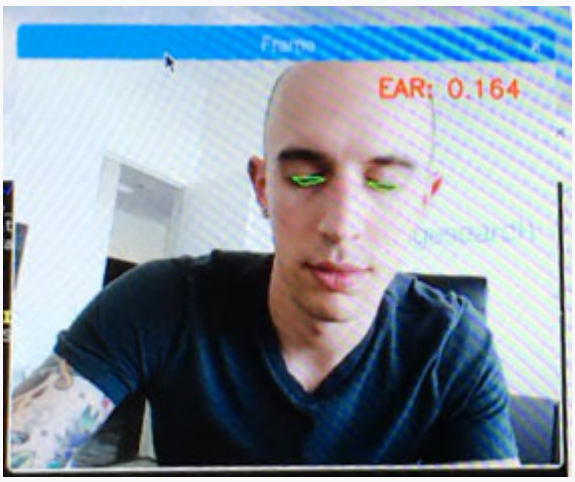

Figure 5: The EAR is low because my eyes are closed — I'm getting drowsy.

And then finally raising an alarm if the eye aspect ratio is below a pre-defined threshold for a sufficiently long amount of time (indicating that the driver/user is tired):

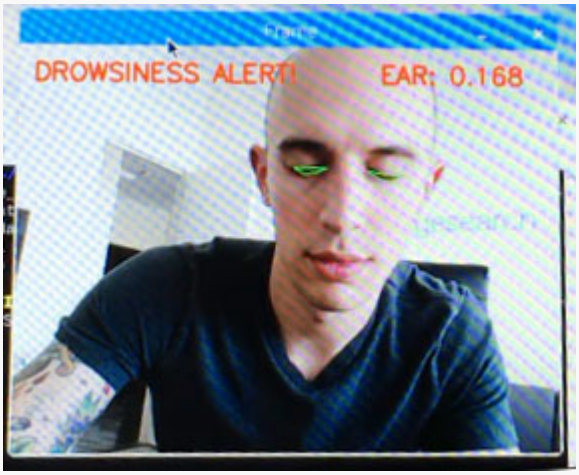

Figure 6: My EAR has been below the threshold long enough for the drowsiness alarm to come on.

In the next section, we'll implement the optimized drowsiness detection algorithm detailed above on the Raspberry Pi using OpenCV, dlib, and Python.

A real-time drowsiness detector on the Raspberry Pi with OpenCV and dlib

Open up a new file in your favorite editor or IDE and name it `pi_drowsiness_detection.py` . From there, let's get started coding:

```
Raspberry Pi: Facial landmarks + drowsiness                    pencv
1 # import the necessary packages
2 from imutils.video import VideoStream
3 from imutils import face_utils
4 import numpy as np
5 import argparse
6 import imutils
7 import time
8 import dlib
9 import cv2
```

Lines 1-9 handle our imports — make sure you have each of these installed in your virtual environment.

From there let's define a distance function:

```
Raspberry Pi: Facial landmarks + drowsiness                    pencv
11 def euclidean_dist(ptA, ptB):
12     # compute and return the euclidean distance between the two
13     # points
14     return np.linalg.norm(ptA - ptB)
```

On **Lines 11-14** we define a convenience function for calculating the Euclidean distance using NumPy. Euclidean is arguably the most well known and must used distance metric. The Euclidean distance is normally described as the distance between two points "as the crow flies".

Now let's define our **Eye Aspect Ratio (EAR) function** which is used to compute the ratio of distances between the vertical eye landmarks and the distances between the horizontal eye landmarks:

Raspberry Pi: Facial landmarks + drowsiness detection with OpenCV

```
16 def eye_aspect_ratio(eye):
17     # compute the euclidean distances between the two sets of
18     # vertical eye landmarks (x, y)-coordinates
19     A = euclidean_dist(eye[1], eye[5])
20     B = euclidean_dist(eye[2], eye[4])
21
22     # compute the euclidean distance between the horizontal
23     # eye landmark (x, y)-coordinates
24     C = euclidean_dist(eye[0], eye[3])
25
26     # compute the eye aspect ratio
27     ear = (A + B) / (2.0 * C)
28
29     # return the eye aspect ratio
30     return ear
```

The return value will be approximately constant when the eye is open and will decrease towards zero during a blink. If the eye is closed, the eye aspect ratio will remain constant at a much smaller value.

From there, we need to parse our command line arguments:

Raspberry Pi: Facial landmarks + drowsiness detection with OpenCV

```
32 # construct the argument parse and parse the arguments
33 ap = argparse.ArgumentParser()
34 ap.add_argument("-c", "--cascade", required=True,
35     help = "path to where the face cascade resides")
36 ap.add_argument("-p", "--shape-predictor", required=True,
37     help="path to facial landmark predictor")
38 ap.add_argument("-a", "--alarm", type=int, default=0,
39     help="boolean used to indicate if TrafficHat should be used")
40 args = vars(ap.parse_args())
```

We have defined two required arguments and one optional one on **Lines 33-40**:

- `--cascade` : The path to the Haar cascade XML file used for face detection.
- `--shape-predictor` : The path to the dlib facial landmark predictor file.
- `--alarm` : A boolean to indicate if the TrafficHat buzzer should be used when drowsiness is detected.

Both the `--cascade` and `--shape-predictor` files are available in the ***"Downloads"*** section at the end of the post.

If the `--alarm` flag is set, we'll set up the TrafficHat:

Raspberry Pi: Facial landmarks + drowsiness detection with OpenCV

```
42 # check to see if we are using GPIO/TrafficHat as an alarm
43 if args["alarm"] > 0:
44     from gpiozero import TrafficHat
45     th = TrafficHat()
46     print("[INFO] using TrafficHat alarm...")
```

As shown in **Lines 43-46** if the argument supplied is greater than 0, we'll import the TrafficHat function to handle our buzzer alarm.

Let's also define a set of important configuration variables:

```
Raspberry Pi: Facial landmarks + drowsiness detection with OpenCV and Python
48 # define two constants, one for the eye aspect ratio to indicate
49 # blink and then a second constant for the number of consecutive
50 # frames the eye must be below the threshold for to set off the
51 # alarm
52 EYE_AR_THRESH = 0.3
53 EYE_AR_CONSEC_FRAMES = 16
54
55 # initialize the frame counter as well as a boolean used to
56 # indicate if the alarm is going off
57 COUNTER = 0
58 ALARM_ON = False
```

The two constants on **Lines 52 and 53** define the EAR threshold and number of consecutive frames eyes must be closed to be considered drowsy, respectively.

Then we initialize the frame counter and a boolean for the alarm (**Lines 57 and 58**).

From there we'll load our Haar cascade and facial landmark predictor files:

```
Raspberry Pi: Facial landmarks + drowsiness detection with OpenCV and Python
60 # load OpenCV's Haar cascade for face detection (which is faster
61 # dlib's built-in HOG detector, but less accurate), then create
62 # facial landmark predictor
63 print("[INFO] loading facial landmark predictor...")
64 detector = cv2.CascadeClassifier(args["cascade"])
65 predictor = dlib.shape_predictor(args["shape_predictor"])
```

Line 64 differs from the face detector initialization from our previous post on drowsiness detection — here we use a faster detection algorithm (Haar cascades) while sacrificing accuracy. Haar cascades are faster than dlib's face detector (which is HOG + Linear SVM-based) making it a great choice for the Raspberry Pi.

There are no changes to **Line 65** where we load up dlib's `shape_predictor` while providing the path to the file.

Next, we'll initialize the indexes of the facial landmarks for each eye:

```
Raspberry Pi: Facial landmarks + drowsiness detection with OpenCV and Python
67 # grab the indexes of the facial landmarks for the left and
68 # right eye, respectively
69 (lStart, lEnd) = face_utils.FACIAL_LANDMARKS_IDXS["left_eye"]
70 (rStart, rEnd) = face_utils.FACIAL_LANDMARKS_IDXS["right_eye"]
```

Here we supply array slice indexes in order to extract the eye regions from the set of facial landmarks.

We're now ready to start our video stream thread:

```
Raspberry Pi: Facial landmarks + drowsiness detection with OpenCV and Python
72 # start the video stream thread
73 print("[INFO] starting video stream thread...")
74 vs = VideoStream(src=0).start()
75 # vs = VideoStream(usePiCamera=True).start()
76 time.sleep(1.0)
```

If you are using the PiCamera module, be sure to *comment out* **Line 74**and *uncomment* **Line 75** to switch the video stream to the Raspberry Pi camera. Otherwise if you are using a USB camera, you can leave this unchanged.

We have one second sleep so the camera sensor can warm up.

From there let's loop over the frames from the video stream:

```
Raspberry Pi: Facial landmarks + drowsiness                    pencv
78 # loop over frames from the video stream
79 while True:
80     # grab the frame from the threaded video file stream, resize
81     # it, and convert it to grayscale
82     # channels)
83     frame = vs.read()
84     frame = imutils.resize(frame, width=450)
85     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
86
87     # detect faces in the grayscale frame
88     rects = detector.detectMultiScale(gray, scaleFactor=1.1,
89         minNeighbors=5, minSize=(30, 30),
90         flags=cv2.CASCADE_SCALE_IMAGE)
```

The beginning of this loop should look familiar if you've read the previous post. We read a frame, resize it (for efficiency), and convert it to grayscale (**Lines 83-85**).

Then we detect faces in the grayscale image with our detector on **Lines 88-90**.

Now let's loop over the detections:

```
Raspberry Pi: Facial landmarks + drowsiness                    pencv
92     # loop over the face detections
93     for (x, y, w, h) in rects:
94         # construct a dlib rectangle object from the Haar casca
95         # bounding box
96         rect = dlib.rectangle(int(x), int(y), int(x + w),
97             int(y + h))
98
99         # determine the facial landmarks for the face region, t
100        # convert the facial landmark (x, y)-coordinates to a N
101        # array
102        shape = predictor(gray, rect)
103        shape = face_utils.shape_to_np(shape)
```

Line 93 begins a lengthy for-loop which is broken down into several code blocks here. First we extract the coordinates and width + height of the `rects` detections. Then, on **Lines 96 and 97** we construct a dlib `rectangle` object using the information extracted from the Haar cascade bounding box.

From there, we determine the facial landmarks for the face region (**Line 102**) and convert the facial landmark *(x, y)*-coordinates to a NumPy array.

Given our NumPy array, `shape` , we can extract each eye's coordinates and compute the EAR:

```
Raspberry Pi: Facial landmarks + drowsiness                    pencv
105        # extract the left and right eye coordinates, then use
```

```
106         # coordinates to compute the eye aspect ratio for both
107         leftEye = shape[lStart:lEnd]
108         rightEye = shape[rStart:rEnd]
109         leftEAR = eye_aspect_ratio(leftEye)
110         rightEAR = eye_aspect_ratio(rightEye)
111
112         # average the eye aspect ratio together for both eyes
113         ear = (leftEAR + rightEAR) / 2.0
```

Utilizing the indexes of the eye landmarks, we can slice the `shape` array to obtain the *(x, y)*-coordinates each eye (**Lines 107 and 108**).

We then calculate the EAR for each eye on **Lines 109 and 110**.

Soukupová and Čech recommend averaging both eye aspect ratios together to obtain a better estimation (**Line 113**).

This next block is strictly for visualization purposes:

```
Raspberry Pi: Facial landmarks + drowsiness                    penCV
115         # compute the convex hull for the left and right eye, t
116         # visualize each of the eyes
117         leftEyeHull = cv2.convexHull(leftEye)
118         rightEyeHull = cv2.convexHull(rightEye)
119         cv2.drawContours(frame, [leftEyeHull], -1, (0, 255, 0),
120         cv2.drawContours(frame, [rightEyeHull], -1, (0, 255, 0)
```

We can visualize each of the eye regions on our frame by using `cv2.drawContours` and supplying the `cv2.convexHull` calculation of each eye (**Lines 117-120**). These few lines are great for debugging our script but aren't necessary if you are making an embedded product with no screen.

From there, we will check our Eye Aspect Ratio (`ear`) and frame counter (`COUNTER`) to see if the eyes are closed, while sounding the alarm to alert the drowsy driver if needed:

```
Raspberry Pi: Facial landmarks + drowsiness                    penCV
122         # check to see if the eye aspect ratio is below the bli
123         # threshold, and if so, increment the blink frame count
124         if ear < EYE_AR_THRESH:
125             COUNTER += 1
126
127             # if the eyes were closed for a sufficient number o
128             # frames, then sound the alarm
129             if COUNTER >= EYE_AR_CONSEC_FRAMES:
130                 # if the alarm is not on, turn it on
131                 if not ALARM_ON:
132                     ALARM_ON = True
133
134                     # check to see if the TrafficHat buzzer sho
135                     # be sounded
136                     if args["alarm"] > 0:
137                         th.buzzer.blink(0.1, 0.1, 10,
138                             background=True)
139
140                 # draw an alarm on the frame
141                 cv2.putText(frame, "DROWSINESS ALERT!", (10, 30
142                     cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255),
143
144         # otherwise, the eye aspect ratio is not below the blin
```

```
145             # threshold, so reset the counter and alarm
146             else:
147                 COUNTER = 0
148                 ALARM_ON = False
```

On **Line 124** we check the `ear` against the `EYE_AR_THRESH` — if it is less than the threshold (eyes are closed), we increment our `COUNTER` (**Line 125**) and subsequently check it to see if the eyes have been closed for enough consecutive frames to sound the alarm (**Line 129**).

If the alarm isn't on, we turn it on for a few seconds to wake up the drowsy driver. This is accomplished on **Lines 136-138**.

Optionally (if you're implementing this code with a screen), you can draw the alarm on the frame as I have done on **Lines 141 and 142**.

That brings us to the case where the `ear` wasn't less than the `EYE_AR_THRESH` — in this case we reset our `COUNTER` to 0 and make sure our alarm is turned off (**Lines 146-148**).

We're almost done — in our last code block we'll draw the EAR on the `frame`, display the `frame`, and do some cleanup:

```
Raspberry Pi: Facial landmarks + drowsiness                    penCV
150             # draw the computed eye aspect ratio on the frame to he
151             # with debugging and setting the correct eye aspect rat
152             # thresholds and frame counters
153             cv2.putText(frame, "EAR: {:.3f}".format(ear), (300, 30)
154                 cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
155
156         # show the frame
157         cv2.imshow("Frame", frame)
158         key = cv2.waitKey(1) & 0xFF
159
160         # if the `q` key was pressed, break from the loop
161         if key == ord("q"):
162             break
163
164 # do a bit of cleanup
165 cv2.destroyAllWindows()
166 vs.stop()
```

If you're integrating with a screen or debugging you may wish to display the computed eye aspect ratio on the frame as I have done on **Lines 153 and 154**. The frame is displayed to the actual screen on **Lines 157 and 158**.

The program is stopped when the 'q' key is pressed on a keyboard (**Lines 157 and 158**).

You might be thinking, *"I won't have a keyboard hooked up in my car!"* Well, if you're debugging using your webcam and your computer at your desk, you certainly do. If you want to use the button on the TrafficHAT to turn on/off the drowsiness detection algorithm, that is perfectly fine — the first reader to post the solution in the comments to using the button to turn on and off the drowsiness detector with the Pi deserves an ice cold craft beer or a hot artisan coffee.

Finally, we clean up by closing any open windows and stopping the video stream (**Lines 165 and 166**).

Drowsiness detection results

To run this program on your own Raspberry Pi, be sure to use the ***"Downloads"*** section at the bottom of this post to grab the source code, face detection Haar cascade, and dlib facial landmark detector.

I didn't have enough time to wire everything up in my car and record the screen while as I did previously. It would have been quite challenging to record the Raspberry Pi screen while driving as well.

Instead, I'll demonstrate at my desk — you can then take this implementation and use it inside your own car for drowsiness detection as you see fit.

You can see an image of my setup below:

Figure 7: My desk setup for coding, testing, and debugging the Raspberry Pi Drowsiness Detector.

To run the program, simply execute the following command:

```
Raspberry Pi: Facial landmarks + drowsiness d                    enCV
$ python pi_detect_drowsiness.py --cascade haarcascade_frontalfac
    --shape-predictor shape_predictor_68_face_landmarks.dat --ala
```

I have included a video of myself demoing the real-time drowsiness detector on the Raspberry Pi below:

Our Raspberry Pi 3 is able to accurately determine if I'm getting "drowsy". We were able to accomplish this using our optimized code.

Disclaimer: *I do not advise that you rely upon the hobbyist Raspberry Pi and this code to keep you awake at the wheel if you are in fact drowsy while driving. The best thing to do is to pull over and rest; walk around; or have a coffee/soda. Have fun with this project and show it off to your friends, but do not risk your life or that of others.*

How do I run this program automatically when the Pi boots up?

This is a common question I receive. I have a blog post covering the answer here: *Running a Python + OpenCV script on reboot*.

Summary

In today's blog post, we learned how to optimize facial landmarks on the Raspberry Pi by swapping out a HOG + Linear SVM-based face detector for a Haar cascade.

Haar cascades, while less accurate, are significantly faster than HOG + Linear SVM detectors.

Given the detections from the Haar cascade we were able to construct a `dlib.rectangle` object corresponding to the bounding box *(x, y)*-coordinates in the image. This object was fed into dlib's facial landmark predictor which in turn gives us the set of localized facial landmarks on the face. From there, we applied the same algorithm we used in our previous post to detect drowsiness in a video stream.

I hope you enjoyed this tutorial!

To be notified when new blog posts are published here on the PyImageSearch blog, *be sure to enter your email address in the form below* — I'll be sure to notify you when new content is released!

Downloads:

If you would like to download the code and images used in this post, please enter your email address in the form below. Not only will you get a .zip of the code, I'll also send you a **FREE 11-page Resource Guide**on Computer Vision and Image Search Engines, including **exclusive techniques** that I don't post on this blog! Sound good? If so, enter your email address and I'll send you the code immediately!

Email address:

Your email address

DOWNLOAD THE CODE!

Resource Guide (it's totally free).

detection, **dlib**, **face detection**, **face parts**, **face regions**, **facial landmarks**, **raspberry pi**, **real-time**, **video**, **video stream**

34 Responses to *Raspberry Pi: Facial landmarks + drowsiness detection with OpenCV and dlib*

Mike October 23, 2017 at 11:07 am #

REPLY

Great article! Do you plan an article (or series) on low light environment face/eye blink detection. Followed your guide recently, but really excited to know how to raise the detection quality on low-light environments/low quality video stream.

Adrian Rosebrock October 23, 2017 at 12:23 pm #

REPLY

It's always easier to write code for (reliable) computer vision algorithms for higher quality video streams than try to write code that compensates for a poor environment. If you're running into situations where you are considering writing code for a poor environment I would encourage you to first examine the environment and see if you can update to make it higher quality.

Mike October 24, 2017 at 7:57 am #

REPLY

Due to business requirements I can't force our clients to shot themselves only in good-to-process conditions. They could be using our software anywhere they want, so… I'd like to read of any approaches available to solve this problem. Just as an idea for you for future publications.

Adrian Rosebrock October 24, 2017 at 10:33 am #

REPLY

Other readers have suggested infrared cameras and infrared lights. I would expect that solution to solve the problem when it is dark outside. There are other “poor conditions” such as reflection and glare which you would need to overcome too. This blog post will get you started but it isn’t intended to be a solution that you can sell.

Petri K. October 24, 2017 at 1:47 am #

REPLY

Raspberry Pi has “night vision” camera boards. They have IR LED spotlights and some of the cameras come without IR filter. Your eyes are not able to see the infrared light, but the camera is. Add light to low light and create higher quality video stream…

There is also IR webcams available and it is possible to use infrared light with some of the standard non IR webcam. Most of the webcams have IR blocking filter, but some of them doesn’t filter properly. (And it is possible to remove the filters in some cases. Use google for this.)

Maybe this could help you?

(The articles are excellent! Thank you Adrian!)

fariborz October 23, 2017 at 11:31 am #

REPLY

WooooooooooooooooooooooooooooooooW

That is great

now this is what i need

very very thank you Adrian

Adrian Rosebrock October 23, 2017 at 12:21 pm #

REPLY

Thanks Fariborz, I’m glad you enjoyed the tutorial 🙂

Some Guy October 23, 2017 at 11:39 am #

REPLY

Hi Dr. Rosebrock, great article as usual! Thank you for the good consistent content. I'm learning a lot 🙂

Adrian Rosebrock October 23, 2017 at 12:21 pm #

REPLY

Thank you 🙂

rohit October 23, 2017 at 9:37 pm #

REPLY

Hi Adrian,
Thanks for posting this.
In this post from May '17 about running dlib on a raspberry pi, you mention that a Raspberry Pi3 is not fast enough to do dlib's face landmark detection in realtime.

https://www.pyimagesearch.com/2017/05/01/install-dlib-raspberry-pi/

Since the drowsiness detection also uses dlib's face landmarks, does it have similar performance issues as you mention in your older post? Or have you figured out some optimizations for RPi3 to improve performance?

Thanks,
Rohit

Adrian Rosebrock October 24, 2017 at 7:17 am #

REPLY

Hi Rohit — please see the section entitled "Haar cascades: less accurate, but faster than HOG". This is where our big speedup comes from.

pochao October 23, 2017 at 10:14 pm #

REPLY

Logitech webcam is better than Pi camera?

Adrian Rosebrock October 24, 2017 at 7:16 am #

REPLY

It depends on how you define "better". What is your use case? How do you intend on deploying it? Both cameras can be good for different reasons. The Raspberry Pi camera module is

cheaper but the Logitech C920 is technically “better” for many uses. It is nice being able to connect the camera directly to the Pi though.

arash allahari October 24, 2017 at 1:40 am #

REPLY

oh Come on man i just wrote this idea two weeks ago in C++ obviously ideas could go beyond pacific through continents

but good news for me is i optimized it with an awesome idea and now i can process drowsiness with almost 30 frame per second from 1 megapixel image stream in Raspberry Pi
WOOOOW…..

I beg u Dr. Rosebrock do not publish such ideas, image processing fans and researchers will get it with just a hint

Adrian Rosebrock October 24, 2017 at 7:14 am #

REPLY

Hey Arash — I actually wrote the original drowsiness detection tutorial way back in May. Secondly, I tend to write blog posts 2-3 weeks ahead of time before they are actually published. I’m not sure what your point is — you would prefer I note publish tutorials?

Melrick Nicolas October 24, 2017 at 3:25 am #

REPLY

How to download updated imutils?

Adrian Rosebrock October 24, 2017 at 7:08 am #

REPLY

I would suggest using “pip”:

```
$ pip install --upgrade imutils
```

If you are using a Python virtual environment please make sure you activate it before installing/upgrading.

Marcus Souza October 24, 2017 at 11:07 am #

REPLY

Hey Adrian,

Thanks for sharing!

As always a great job !!

I tested with webcom and verified a great performance in the identification of drowsiness, with a processing load of 70%. Perhaps there is something that can be improved to reduce PLOAD, perhaps by altering Haarcascade, perhaps by using the one haarcascade_eye.xml or similar, targeting only the eye area. I wanted you to share your opinion with us. Can you comment on the subject?

Thanks for all help, Adrian

Adrian Rosebrock October 25, 2017 at 8:23 am #

REPLY

In order to apply drowsiness detection we need to detect the entire face — this enables us to localize the eyes. We could use a Haar cascade to detect eyes but the problem is that we need to train a facial landmark detector for just the eyes. That wouldn't do much to improve processing speed.

Raghu October 24, 2017 at 11:32 am #

REPLY

Hi Adrian,

I'm impressed with the tutorial!

Please let me know what Operating System used in the Raspberry Pi 3.

Adrian Rosebrock October 24, 2017 at 2:50 pm #

REPLY

Hi Raghu — Raspbian is the official operating system and the one used. You can download it here.

Marcus Souza October 24, 2017 at 1:29 pm #

REPLY

Hey Adrian,

First thank you for sharing this great edition !!

Doing some tests I found the following error in the code, when I used the "PICamera", I got the following TypeError:

from:
vs = VideoStream(usePicamera=True).start()

to:
vs = VideoStream(usePiCamera=True).start()

This corrects the following failure:

vs = VideoStream(usePicamera=True).start()
TYpeError: __init__() got an unexpected keyword argument
'usePicamera'

Thanks,
Marvin

Adrian Rosebrock October 24, 2017 at 2:34 pm #

REPLY

Thank you Marvin — you are correct. I've updated the post, and I'll update the download soon. Thanks for bringing this to my attention.

Adrian Rosebrock October 31, 2017 at 8:54 am #

REPLY

I have now updated the code download as well. Thanks again!

Yoni October 25, 2017 at 6:59 am #

REPLY

Hey!

So there's something which bothers me here:

Your original article used something like HOG+SVM and a sliding window for detection.
I got to say, that face detector that you have provided does work most of the time (~75%).
However, doesn't RCNN (or faster RCNN etc,whatever you get the point) just work better than pre-deep learning techniques? I mean,that's what Justin from stanford claims (https://www.youtube.com/watch?v=nDPWywWRIRo&index=11&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&t=1950s).

Is that really the case? If so, when should I NOT prefer RCNN & Why?

Adrian Rosebrock October 25, 2017 at 8:21 am #

REPLY