https://www.pyimagesearch.com/2017/10/16/raspberry-pi-deep-learning-object-detection-with-opencv/

Raspberry Pi: Deep learning object detection with OpenCV

by Adrian Rosebrock on October 16, 2017 in Deep Learning, Object Detection, OpenCV, Raspberry Pi

A few weeks ago I demonstrated how to perform real-time object detection using deep learning and OpenCV on a standard laptop/desktop.

After the post was published I received a number of emails from PyImageSearch readers who were curious if the Raspberry Pi could *also* be used for real-time object detection.

The short answer is "kind of"...

...but only if you set your expectations accordingly.

Even when applying our optimized OpenCV + Raspberry Pi install the Pi is only capable of **getting up to ~0.9 frames per second** when applying deep learning for object detection with Python and OpenCV.

Is that fast enough?

Well, that depends on your application.

If you're attempting to detect objects that are quickly moving through your field of view, likely

not.

But if you're monitoring a low traffic environment with slower moving objects, the Raspberry Pi could indeed be fast enough.

In the remainder of today's blog post we'll be reviewing two methods to perform deep learning-based object detection on the Raspberry Pi.

Looking for the source code to this post?

Jump right to the downloads section.

Raspberry Pi: Deep learning object detection with OpenCV

Today’ s blog post is broken down into two parts.

In the first part, we’ ll benchmark the Raspberry Pi for real-time object detection using OpenCV and Python. This benchmark will come from the exact code we used for our laptop/desktop deep learning object detector from a few weeks ago.

I’ ll then demonstrate how to use multiprocessing to create an alternate method to object detection using the Raspberry Pi. This method may or may not be useful for your particular application, but at the very least it will give you an idea on different methods to approach the problem.

Object detection and OpenCV benchmark on the Raspberry Pi

The code we’ ll discuss in this section is is identical to our previous post on *Real-time object detection with deep learning and OpenCV*; therefore, I will not be reviewing the code exhaustively.

For a deep dive into the code, please see the original post.

Instead, we’ ll simply be using this code to benchmark the Raspberry Pi for deep learning-based object detection.

To get started, open up a new file, name it `real_time_object_detection.py` , and insert the following code:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
# import the necessary packages
from imutils.video import VideoStream
from imutils.video import FPS
import numpy as np
import argparse
import imutils
import time
import cv2
```

We then need to parse our command line arguments:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-p", "--prototxt", required=True,
```

```
13          help="path to Caffe 'deploy' prototxt file")
14  ap.add_argument("-m", "--model", required=True,
15          help="path to Caffe pre-trained model")
16  ap.add_argument("-c", "--confidence", type=float, default=0.2,
17          help="minimum probability to filter weak detections")
18  args = vars(ap.parse_args())
```

Followed by performing some initializations:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
20  # initialize the list of class labels MobileNet SSD was trained to
21  # detect, then generate a set of bounding box colors for each class
22  CLASSES = ["background", "aeroplane", "bicycle", "bird", "boat",
23          "bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
24          "dog", "horse", "motorbike", "person", "pottedplant", "sheep",
25          "sofa", "train", "tvmonitor"]
26  COLORS = np.random.uniform(0, 255, size=(len(CLASSES), 3))
27
28  # load our serialized model from disk
29  print("[INFO] loading model...")
30  net = cv2.dnn.readNetFromCaffe(args["prototxt"], args["model"])
```

We initialize `CLASSES` , our class labels, and corresponding `COLORS` , for on-frame text and bounding boxes (**Lines 22-26**), followed by loading the serialized neural network model (**Line 30**).

Next, we' ll initialize the video stream object and frames per second counter:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
32  # initialize the video stream, allow the camera sensor to warm up,
33  # and initialize the FPS counter
34  print("[INFO] starting video stream...")
35  vs = VideoStream(src=0).start()
36  # vs = VideoStream(usePiCamera=True).start()
37  time.sleep(2.0)
38  fps = FPS().start()
```

Wwe initialize the video stream and allow the camera warm up for 2.0 seconds (**Lines 35-37**).

On **Line 35** we initialize our `VideoStream` using a USB camera If you are using the Raspberry Pi camera module you' ll want to comment out **Line 35** and uncomment **Line 36** (which will enable you to access the Raspberry Pi camera module via the `VideoStream` class).

From there we start our `fps` counter on **Line 38**.

We are now ready to loop over frames from our input video stream:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
40  # loop over the frames from the video stream
41  while True:
42          # grab the frame from the threaded video stream and resize it
```

```
43          # to have a maximum width of 400 pixels
44          frame = vs.read()
45          frame = imutils.resize(frame, width=400)
46
47          # grab the frame dimensions and convert it to a blob
48          (h, w) = frame.shape[:2]
49          blob = cv2.dnn.blobFromImage(cv2.resize(frame, (300, 300)),
50                  0.007843, (300, 300), 127.5)
51
52          # pass the blob through the network and obtain the detections and
53          # predictions
54          net.setInput(blob)
55          detections = net.forward()
```

Lines 41-55 simply grab and resize a `frame`, convert it to a `blob`, and pass the `blob` through the neural network, obtaining the `detections` and bounding box predictions.

From there we need to loop over the `detections` to see what objects were detected in the `frame`:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
57          # loop over the detections
58          for i in np.arange(0, detections.shape[2]):
59                  # extract the confidence (i.e., probability) associated with
60                  # the prediction
61                  confidence = detections[0, 0, i, 2]
62
63                  # filter out weak detections by ensuring the `confidence` is
64                  # greater than the minimum confidence
65                  if confidence > args["confidence"]:
66                          # extract the index of the class label from the
67                          # `detections`, then compute the (x, y)-coordinates of
68                          # the bounding box for the object
69                          idx = int(detections[0, 0, i, 1])
70                          box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
71                          (startX, startY, endX, endY) = box.astype("int")
72
73                          # draw the prediction on the frame
74                          label = "{}: {:.2f}%".format(CLASSES[idx],
75                                  confidence * 100)
76                          cv2.rectangle(frame, (startX, startY), (endX, endY),
77                                  COLORS[idx], 2)
78                          y = startY - 15 if startY - 15 > 15 else startY + 15
79                          cv2.putText(frame, label, (startX, y),
80                                  cv2.FONT_HERSHEY_SIMPLEX, 0.5, COLORS[idx], 2)
```

On **Lines 58-80**, we loop over our `detections`. For each detection we examine the `confidence` and ensure the corresponding probability of the detection is above a predefined threshold. If it is, then we extract the class label and compute *(x, y)* bounding box coordinates. These coordinates will enable us to draw a bounding box around the object in the image along with the associated class label.

From there we'll finish out the loop and do some cleanup:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
82          # show the output frame
83          cv2.imshow("Frame", frame)
84          key = cv2.waitKey(1) & 0xFF
85
86          # if the `q` key was pressed, break from the loop
87          if key == ord("q"):
88                  break
89
90          # update the FPS counter
91          fps.update()
92
93  # stop the timer and display FPS information
94  fps.stop()
95  print("[INFO] elapsed time: {:.2f}".format(fps.elapsed()))
96  print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))
97
98  # do a bit of cleanup
99  cv2.destroyAllWindows()
100 vs.stop()
```

Lines 82-91 close out the loop — we show each frame, `break` if 'q' key is pressed, and update our `fps` counter.

The final terminal message output and cleanup is handled on **Lines 94-100**.

Now that our brief explanation of `real_time_object_detection.py` is finished, let's examine the results of this approach to obtain a baseline.

Go ahead and use the ***"Downloads"*** section of this post to download the source code and pre-trained models.

From there, execute the following command:

Raspberry Pi: Deep learning object detection with OpenCV

Shell

```
1   $ python real_time_object_detection.py \
2           --prototxt MobileNetSSD_deploy.prototxt.txt \
3           --model MobileNetSSD_deploy.caffemodel
4   [INFO] loading model...
5   [INFO] starting video stream...
6   [INFO] elapsed time: 54.70
7   [INFO] approx. FPS: 0.90
```

As you can see from my results we are obtaining ~0.9 frames per second throughput using this method and the Raspberry Pi.

Compared to the 6-7 frames per second using our laptop/desktop we can see that the Raspberry Pi is *substantially* slower.

That's not to say that the Raspberry Pi is unusable when applying deep learning object detection, but you need to set your expectations on what's

realistic (even when applying our OpenCV + Raspberry Pi optimizations).

Note: *For what it's worth, I could only obtain* ***0.49 FPS*** *when* ***NOT*** *using our optimized OpenCV + Raspberry Pi install — that just goes to show you how much of a difference NEON and VFPV3 can make.*

A different approach to object detection on the Raspberry Pi

Using the example from the previous section we see that calling `net.forward()` is a ***blocking operation*** — the rest of the code in the `while` loop is not allowed to complete until `net.forward()` returns the `detections`.

So, what if `net.forward()` was not a blocking operation?

Would we able to obtain a faster frames per second throughput?

Well, that's a loaded question.

No matter what, it will take approximately a little over a second for `net.forward()` to complete using the Raspberry Pi and this particular architecture — that cannot change.

But what we *can* do is create a separate process that is solely responsible for applying the deep learning object detector, thereby unblocking the main thread of execution and allow our `while` loop to continue.

Moving the predictions to separate process will give the *illusion* that our Raspberry Pi object detector is running faster than it actually is, when in reality the `net.forward()` computation is still taking a little over one second.

The only problem here is that our output object detection predictions will lag behind what is currently being displayed on our screen. If you detecting *fast-moving objects* you may miss the detection entirely, or at the very least, the object will be out of the frame before you obtain your detections from the neural network.

Therefore, this approach should only be used for *slow-moving objects* where we can tolerate lag.

To see how this multiprocessing method works, open up a new file, name it `pi_object_detection.py`, and insert the following code:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
1  # import the necessary packages
```

```
from imutils.video import VideoStream
from imutils.video import FPS
from multiprocessing import Process
from multiprocessing import Queue
import numpy as np
import argparse
import imutils
import time
import cv2
```

For the code walkthrough in this section, I'll be pointing out and explaining the *differences* (there are quite a few) compared to our non-multprocessing method.

Our imports on **Lines 2-10** are mostly the same, but notice the imports of `Process` and `Queue` from Python's multiprocessing package.

Next, I'd like to draw your attention to a new function, `classify_frame`:

```
Raspberry Pi: Deep learning object detection with OpenCV
Python
```

```
def classify_frame(net, inputQueue, outputQueue):
	# keep looping
	while True:
		# check to see if there is a frame in our input queue
		if not inputQueue.empty():
			# grab the frame from the input queue, resize it, and
			# construct a blob from it
			frame = inputQueue.get()
			frame = cv2.resize(frame, (300, 300))
			blob = cv2.dnn.blobFromImage(frame, 0.007843,
				(300, 300), 127.5)

			# set the blob as input to our deep learning object
			# detector and obtain the detections
			net.setInput(blob)
			detections = net.forward()

			# write the detections to the output queue
			outputQueue.put(detections)
```

Our new `classify_frame` function is responsible for our multiprocessing — later on we'll set it up to run in a child process.

The `classify_frame` function takes three parameters:

- `net`: the neural network object.
- `inputQueue`: our FIFO (first in first out) queue of frames for object detection.
- `outputQueue`: our FIFO queue of detections which will be processed in the main thread.

This child process will loop continuously until the parent exits and effectively terminates the child.

In the loop, if the `inputQueue` contains a `frame`, we grab it, and then pre-process it and create a `blob` (**Lines 16-22**), just as we have done in the previous script.

From there, we send the `blob` through the neural network (**Lines 26-27**) and place the `detections` in an `outputQueue` for processing by the parent.

Now let's parse our command line arguments:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
32 # construct the argument parse and parse the arguments
33 ap = argparse.ArgumentParser()
34 ap.add_argument("-p", "--prototxt", required=True,
35         help="path to Caffe 'deploy' prototxt file")
36 ap.add_argument("-m", "--model", required=True,
37         help="path to Caffe pre-trained model")
38 ap.add_argument("-c", "--confidence", type=float, default=0.2,
39         help="minimum probability to filter weak detections")
40 args = vars(ap.parse_args())
```

There is no difference here — we are simply parsing the same command line arguments on **Lines 33-40**.

Next we initialize some variables just as in our previous script:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
42 # initialize the list of class labels MobileNet SSD was trained to
43 # detect, then generate a set of bounding box colors for each class
44 CLASSES = ["background", "aeroplane", "bicycle", "bird", "boat",
45         "bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
46         "dog", "horse", "motorbike", "person", "pottedplant", "sheep",
47         "sofa", "train", "tvmonitor"]
48 COLORS = np.random.uniform(0, 255, size=(len(CLASSES), 3))
49
50 # load our serialized model from disk
51 print("[INFO] loading model...")
52 net = cv2.dnn.readNetFromCaffe(args["prototxt"], args["model"])
```

This code is the same — we initialize class labels, colors, and load our model.

Here's where things get different:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
54 # initialize the input queue (frames), output queue (detections),
55 # and the list of actual detections returned by the child process
56 inputQueue = Queue(maxsize=1)
57 outputQueue = Queue(maxsize=1)
58 detections = None
```

On **Lines 56-58** we initialize an `inputQueue` of frames, an `outputQueue` of detections, and a `detections` list.

Our `inputQueue` will be populated by the parent and processed by the child — it is the input to the child process. Our `outputQueue` will be populated by the child, and processed by the parent — it is output from the child process. Both of these queues trivially have a size of *one* as our neural network will only be applying object detections to one frame at a time.

Let's initialize and start the child process:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
60  # construct a child process *indepedent* from our main process of
61  # execution
62  print("[INFO] starting process...")
63  p = Process(target=classify_frame, args=(net, inputQueue,
64          outputQueue,))
65  p.daemon = True
66  p.start()
```

It is very easy to construct a child process with Python's multiprocessing module — simply specify the `target` function and `args` to the function as we have done on **Lines 63 and 64**.

Line 65 specifies that `p` is a daemon process, and **Line 66** kicks the process off.

From there we'll see some more familiar code:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
68  # initialize the video stream, allow the cammera sensor to warmup,
69  # and initialize the FPS counter
70  print("[INFO] starting video stream...")
71  vs = VideoStream(src=0).start()
72  # vs = VideoStream(usePiCamera=True).start()
73  time.sleep(2.0)
74  fps = FPS().start()
```

Don't forget to change your video stream object to use the PiCamera if you desire by switching which line is commented (**Lines 71 and 72**).

Once our `vs` object and `fps` counters are initialized, we can loop over the video frames:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
76  # loop over the frames from the video stream
77  while True:
78          # grab the frame from the threaded video stream, resize it, and
79          # grab its dimensions
80          frame = vs.read()
81          frame = imutils.resize(frame, width=400)
82          (fH, fW) = frame.shape[:2]
```

On **Lines 80-82**, we read a frame, resize it, and extract the width and height.

Next, we'll work our our queues into the flow:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
84          # if the input queue *is* empty, give the current frame to
85          # classify
86          if inputQueue.empty():
87                  inputQueue.put(frame)
88
89          # if the output queue *is not* empty, grab the detections
90          if not outputQueue.empty():
91                  detections = outputQueue.get()
```

First we check if the `inputQueue` is empty — if it is empty, we put a frame in the `inputQueue` for processing by the child (**Lines 86 and 87**). Remember, the child process is running in an infinite loop, so it will be processing the `inputQueue` in the background.

Then we check if the `outputQueue` is not empty — if it is not empty (something is in it), we grab the `detections` for processing here in the parent (**Lines 90 and 91**). When we call `get()` on the `outputQueue`, the detections are returned and the `outputQueue` is now momentarily empty.

If you are unfamiliar with Queues or if you want a refresher, see this documentation.

Let's process our detections:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
93          # check to see if our detectios are not None (and if so, we'll
94          # draw the detections on the frame)
95          if detections is not None:
96                  # loop over the detections
97                  for i in np.arange(0, detections.shape[2]):
98                          # extract the confidence (i.e., probability) associated
99                          # with the prediction
100                         confidence = detections[0, 0, i, 2]
101
102                         # filter out weak detections by ensuring the `confidence`
103                         # is greater than the minimum confidence
104                         if confidence < args["confidence"]:
105                                 continue
106
107                         # otherwise, extract the index of the class label from
108                         # the `detections`, then compute the (x, y)-coordinates
109                         # of the bounding box for the object
110                         idx = int(detections[0, 0, i, 1])
111                         dims = np.array([fW, fH, fW, fH])
112                         box = detections[0, 0, i, 3:7] * dims
113                         (startX, startY, endX, endY) = box.astype("int")
114
115                         # draw the prediction on the frame
116                         label = "{}: {:.2f}%".format(CLASSES[idx],
117                                 confidence * 100)
```

```
118                         cv2.rectangle(frame, (startX, startY), (endX, endY),
119                                 COLORS[idx], 2)
120                         y = startY - 15 if startY - 15 > 15 else startY + 15
121                         cv2.putText(frame, label, (startX, y),
122                                 cv2.FONT_HERSHEY_SIMPLEX, 0.5, COLORS[idx], 2)
```

If our `detections` list is populated (it is not `None`), we loop over the detections as we have done in the previous section's code.

In the loop, we extract and check the `confidence` against the threshold (**Lines 100-105**), extract the class label index (**Line 110**), and draw a box and label on the frame (**Lines 111-122**).

From there in the while loop we'll complete a few remaining steps, followed by printing some statistics to the terminal, and performing cleanup:

Raspberry Pi: Deep learning object detection with OpenCV

Python

```
124         # show the output frame
125         cv2.imshow("Frame", frame)
126         key = cv2.waitKey(1) & 0xFF
127
128         # if the `q` key was pressed, break from the loop
129         if key == ord("q"):
130                 break
131
132         # update the FPS counter
133         fps.update()
134
135 # stop the timer and display FPS information
136 fps.stop()
137 print("[INFO] elapsed time: {:.2f}".format(fps.elapsed()))
138 print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))
139
140 # do a bit of cleanup
141 cv2.destroyAllWindows()
142 vs.stop()
```

In the remainder of the loop, we display the frame to the screen (**Line 125**) and capture a key press and check if it is the quit key at which point we break out of the loop (**Lines 126-130**). We also update our `fps` counter.

To finish out, we stop the `fps` counter, print our time/FPS statistics, and finally close windows and stop the video stream (**Lines 136-142**).

Now that we're done walking through our new multiprocessing code, let's compare the method to the single thread approach from the previous section.

Be sure to use the ***"Downloads"*** section of this blog post to download the source code + pre-trained MobileNet SSD neural network. From there, execute the following command:

Raspberry Pi: Deep learning object detection with OpenCV

```shell
$ python pi_object_detection.py \
        --prototxt MobileNetSSD_deploy.prototxt.txt \
        --model MobileNetSSD_deploy.caffemodel
[INFO] loading model...
[INFO] starting process...
[INFO] starting video stream...
[INFO] elapsed time: 48.55
[INFO] approx. FPS: 27.83
```

Here you can see that our `while` loop is capable of processing 27 frames per second. However, this throughput rate is an illusion — the neural network running in the background is still only capable of processing 0.9 frames per second.

Note: *I also tested this code on the Raspberry Pi camera module and was able to obtain 60.92 frames per second over 35 elapsed seconds.*

The difference here is that we can obtain real-time *throughput* by displaying each new input frame in real-time and then drawing any previous `detections` on the current frame.

Once we have a new set of `detections` we then draw the new ones on the frame.

This process repeats until we exit the script. **The downside is that we see substantial lag.** There are clips in the above video where we can see that all objects have clearly left the field of view…

…however, our script still reports the objects as being present.

Therefore, you should ***consider only using this approach when***:

1. Objects are slow moving and the previous detections can be used as an approximation to the new location.
2. Displaying the actual frames themselves in real-time is paramount to user experience.

Summary

In today’ s blog post we examined using the Raspberry Pi for object detection using deep learning, OpenCV, and Python.

As our results demonstrated we were able to get up to **0.9 frames per second**, which is not fast enough to constitute real-time detection. That said,

given the limited processing power of the Pi, 0.9 frames per second is still reasonable for some applications.

We then wrapped up this blog post by examining an alternate method to deep learning object detection on the Raspberry Pi by using multiprocessing. Whether or not this second approach is suitable for you is again highly dependent on your application.

If your use case involves low traffic object detection where the objects are slow moving through the frame, then you can certainly consider using the Raspberry Pi for deep learning object detection. However, if you are developing an application that involves many objects that are fast moving, you should instead consider faster hardware.

Thanks for reading and enjoy!

And if you’ re interested in studying deep learning in more depth, be sure to take a look at my new book, *Deep Learning for Computer Vision with Python*. Whether this is the first time you’ ve worked with machine learning and neural networks or you’ re already a seasoned deep learning practitioner, my new book is engineered from the ground up to help you reach expert status. **Just click here to start your journey to deep learning mastery**.