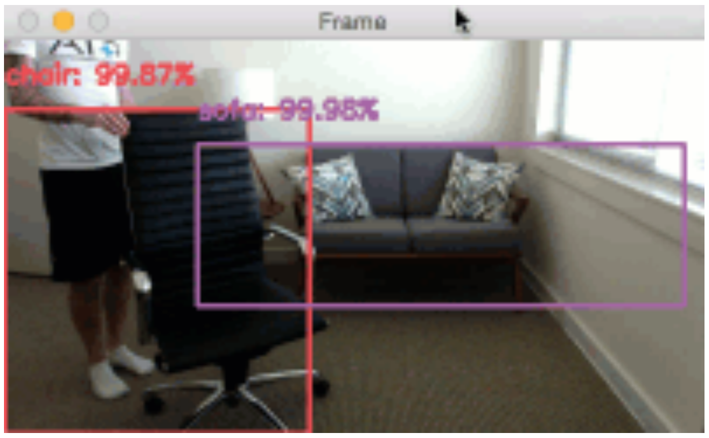
pyimagesearch.com

Real-time object detection with deep learning and OpenCV

Adrian Rosebrock

13-16 minutes



Today's blog post was inspired by PyImageSearch reader, Emmanuel. Emmanuel emailed me after last week's tutorial on object detection with deep learning + OpenCV and asked:

> "Hi Adrian,
>
> I really enjoyed last week's blog post on object detection with deep learning and OpenCV, thanks for putting it together and for making deep learning with OpenCV so accessible.
>
> I want to apply the same technique to real-time video.
>
> What is the best way to do this?
>
> How can I achieve the most efficiency?

> If you could do a tutorial on real-time object detection with deep learning and OpenCV I would really appreciate it."

Great question, thanks for asking Emmanuel.

Luckily, extending our previous tutorial on object detection with deep learning and OpenCV to ***real-time video streams*** is fairly straightforward — we simply need to combine some efficient, boilerplate code for real-time video access and then add in our object detection.

By the end of this tutorial you'll be able to apply deep learning-based object detection to real-time video streams using OpenCV and Python — to learn how, *just keep reading*.

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

Real-time object detection with deep learning and OpenCV

Today's blog post is broken into two parts.

In the first part we'll learn how to extend last week's tutorial to apply real-time object detection using deep learning and OpenCV to work with video streams and video files. This will be accomplished using the highly efficient VideoStream class discussed in this tutorial.

From there, we'll apply our deep learning + object detection code to actual video streams and measure the FPS processing rate.

Object detection in video with deep learning and OpenCV

To build our deep learning-based real-time object detector with OpenCV we'll need to (1) access our webcam/video stream in an efficient manner and (2) apply object detection to each frame.

To see how this is done, open up a new file, name it real_time_object_detection.py and insert the following code:

```
# 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 begin by importing packages on **Lines 2-8**. For this tutorial, you will need imutils and OpenCV 3.3.

To get your system set up, simply install OpenCV using the relevant instructions for your system (while ensuring you're following any Python virtualenv commands).

Note: *Make sure to download and install opencv and and opencv-contrib releases for OpenCV 3.3. This will ensure that the deep neural network (dnn) module is installed. You* ***must***

have OpenCV 3.3 (or newer) to run the code in this tutorial.

Next, we'll parse our command line arguments:

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

Compared to last week, we don't need the image argument since we're working with streams and videos — other than that the following arguments remain the same:

- --prototxt : The path to the Caffe prototxt file.
- --model : The path to the pre-trained model.
- --confidence : The minimum probability threshold to filter weak detections. The default is 20%.

We then initialize a class list and a color set:

```
# initialize the list of class labels MobileNet SSD was trained to
# detect, then generate a set of bounding box colors for each class
```

```
CLASSES = ["background", "aeroplane", "bicycle", "bird",
"boat",
"bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
"dog", "horse", "motorbike", "person", "pottedplant", "sheep",
"sofa", "train", "tvmonitor"]
COLORS = np.random.uniform(0, 255, size=(len(CLASSES),
3))
```

On **Lines 22-26** we initialize CLASS labels and corresponding random COLORS . For more information on these classes (and how the network was trained), please refer to last week's blog post.

Now, let's load our model and set up our video stream:

```
# load our serialized model from disk
print("[INFO] loading model...")
net = cv2.dnn.readNetFromCaffe(args["prototxt"],
args["model"])
# initialize the video stream, allow the cammera sensor to
warmup,
# and initialize the FPS counter
print("[INFO] starting video stream...")
vs = VideoStream(src=0).start()
time.sleep(2.0)
fps = FPS().start()
```

We load our serialized model, providing the references to our prototxt and model files on **Line 30** — notice how easy this is in OpenCV 3.3.

Next let's initialize our video stream (this can be from a video file or a camera). First we start the VideoStream (**Line 35**), then we wait for the camera to warm up (**Line 36**), and finally we start the frames per second counter (**Line 37**). The VideoStream and FPS classes are part of my imutils package.

Now, let's loop over each and every frame (for speed purposes, you could skip frames):

```
# loop over the frames from the video stream
while True:
# grab the frame from the threaded video stream and resize it
# to have a maximum width of 400 pixels
frame = vs.read()
frame = imutils.resize(frame, width=400)
# grab the frame dimensions and convert it to a blob
(h, w) = frame.shape[:2]
blob = cv2.dnn.blobFromImage(cv2.resize(frame, (300, 300)),
0.007843, (300, 300), 127.5)
# pass the blob through the network and obtain the
```

```
51 detections and
52 # predictions
53 net.setInput(blob)
54 detections = net.forward()
```

First, we read a frame (**Line 43**) from the stream, followed by resizing it (**Line 44**).

Since we will need the width and height later, we grab these now on **Line 47**. This is followed by converting the frame to a blob with the dnn module (**Lines 48 and 49**).

Now for the heavy lifting: we set the blob as the input to our neural network (**Line 53**) and feed the input through the net (**Line 54**) which gives us our detections .

At this point, we have detected objects in the input frame. It is now time to look at confidence values and determine if we should draw a box + label surrounding the object– you'll recognize this code block from last week:

```
55 # loop over the detections
56 for i in np.arange(0, detections.shape[2]):
57 # extract the confidence (i.e., probability) associated with
58 # the prediction
59 confidence = detections[0, 0, i, 2]
60 # filter out weak detections by ensuring the `confidence` is
61 # greater than the minimum confidence
```

```

if confidence > args["confidence"]:
# extract the index of the class label from the
# `detections`, then compute the (x, y)-coordinates of
# the bounding box for the object
idx = int(detections[0, 0, i, 1])
box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
(startX, startY, endX, endY) = box.astype("int")
# draw the prediction on the frame
label = "{}: {:.2f}%".format(CLASSES[idx],
confidence * 100)
cv2.rectangle(frame, (startX, startY), (endX, endY),
COLORS[idx], 2)
y = startY - 15 if startY - 15 > 15 else startY + 15
cv2.putText(frame, label, (startX, y),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, COLORS[idx], 2)

```

We start by looping over our detections , keeping in mind that multiple objects can be detected in a single image. We also apply a check to the confidence (i.e., probability) associated with each detection. If the confidence is high enough (i.e. above the threshold), then we'll display the prediction in the terminal as well as draw the prediction on the image with text and a colored bounding box. Let's break it down line-by-line:

Looping through our detections , first we extract the confidence value (**Line 60**).

If the confidence is above our minimum threshold (**Line 64**), we extract the class label index (**Line 68**) and compute the bounding box coordinates around the detected object (**Line 69**).

Then, we extract the *(x, y)*-coordinates of the box (**Line 70**) which we will will use shortly for drawing a rectangle and displaying text.

We build a text label containing the CLASS name and the confidence (**Lines 73 and 74**).

Let's also draw a colored rectangle around the object using our class color and previously extracted *(x, y)*-coordinates (**Lines 75 and 76**).

In general, we want the label to be displayed above the rectangle, but if there isn't room, we'll display it just below the top of the rectangle (**Line 77**).

Finally, we overlay the colored text onto the frame using the *y*-value that we just calculated (**Lines 78 and 79**).

The remaining steps in the frame capture loop involve (1) displaying the frame, (2) checking for a quit key, and (3) updating our frames per second counter:

```
# show the output frame

cv2.imshow("Frame", frame)
```

```
key = cv2.waitKey(1) & 0xFF
# if the `q` key was pressed, break from the loop
if key == ord("q"):
break
# update the FPS counter
fps.update()
```

The above code block is pretty self-explanatory — first we display the frame (**Line 82**). Then we capture a key press (**Line 83**) while checking if the 'q' key (for "quit") is pressed, at which point we break out of the frame capture loop (**Lines 86 and 87**).

Finally we update our fps counter (**Line 90**).

If we break out of the loop ('q' key press or end of the video stream), we have some housekeeping to take care of:

```
# stop the timer and display FPS information
fps.stop()
print("[INFO] elapsed time: {:.2f}".format(fps.elapsed()))
print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))
# do a bit of cleanup
cv2.destroyAllWindows()
vs.stop()
```

When we've exited the loop, we stop the fps counter (**Line 93**) and print information about the frames per second to our terminal (**Lines 94 and 95**).

We close the open window (**Line 98**) followed by stopping the video stream (**Line 99**).

If you've made it this far, you're probably ready to give it a try with your webcam — to see how it's done, let's move on to the next section.

Real-time deep learning object detection results

To see our real-time deep-learning based object detector in action, make sure you use the ***"Downloads"*** section of this guide to download the example code + pre-trained Convolutional Neural Network.

From there, open up a terminal and execute the following command:

```
$ python real_time_object_detection.py \
--prototxt MobileNetSSD_deploy.prototxt.txt \
--model MobileNetSSD_deploy.caffemodel
[INFO] loading model...
[INFO] starting video stream...
[INFO] elapsed time: 55.07
[INFO] approx. FPS: 6.54
```

Provided that OpenCV can access your webcam you should see the output video frame with any detected objects. I have included sample results of applying deep learning object detection to an example video below:

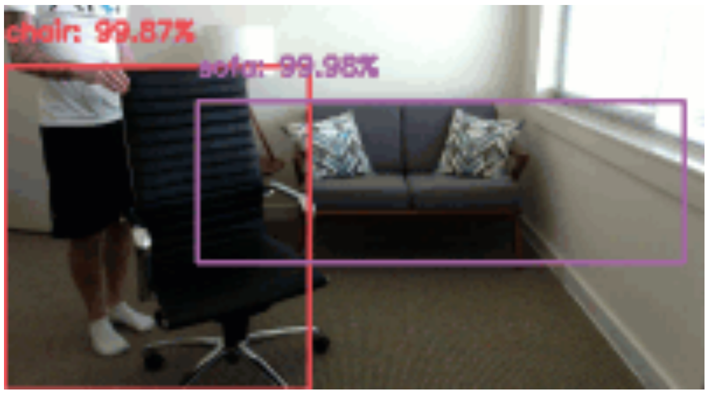


Figure 1: *A short clip of real-time object detection with deep learning and OpenCV + Python.*

Notice how our deep learning object detector can detect not only myself (a *person*), but also the *sofa* I am sitting on and the *chair* next to me — ***all in real-time!***

The full video can be found below:

Summary

In today's blog post we learned how to perform real-time object detection using deep learning + OpenCV + video streams.

We accomplished this by combing two separate tutorials:

1. Object detection with deep learning and OpenCV
2. Efficient, threaded video streams with OpenCV

The end result is a deep learning-based object detector that can process approximately 6-8 FPS (depending on the speed of your system, of course).

Further speed improvements can be obtained by:

1. Applying skip frames.
2. Swapping different variations of MobileNet (that are faster, but

less accurate).

3. Potentially using the quantized variation of SqueezeNet (I haven't tested this, but imagine it would be faster due to smaller network footprint).

In future blog posts we'll be discussing deep learning object detection methods in more detail.

In the meantime, be sure to take a look at my book, *Deep Learning for Computer Vision with Python*, where I'll be reviewing object detection frameworks such as Faster R-CNNs and Single Shot Detectors!

If you're interested in studying deep learning for computer vision and image classification tasks, you just can't beat this book — click here to learn more.

Downloads:

Resource Guide (it's totally free).

Enter your email address below to get my **free 11-page Image Search Engine Resource Guide PDF**. Uncover **exclusive**

techniques that I don't publish on this blog and start building image search engines of your own!

cnn, coco, convolutional neural network, deep learning, machine learning, opencv 3, pascal voc, real-time, video, video stream