PYIMAGESEARCH

Click here to download the source code to this post

DEEP LEARNING (HTTPS://PYIMAGESEARCH.COM/CATEGORY/DEEP-LEARNING/)

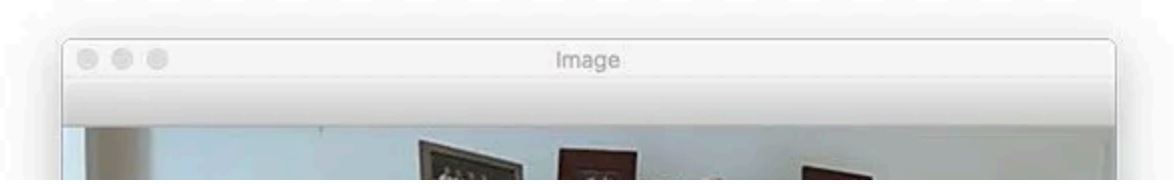
OBJECT DETECTION (HTTPS://PYIMAGESEARCH.COM/CATEGORY/OBJECT-DETECTION/)

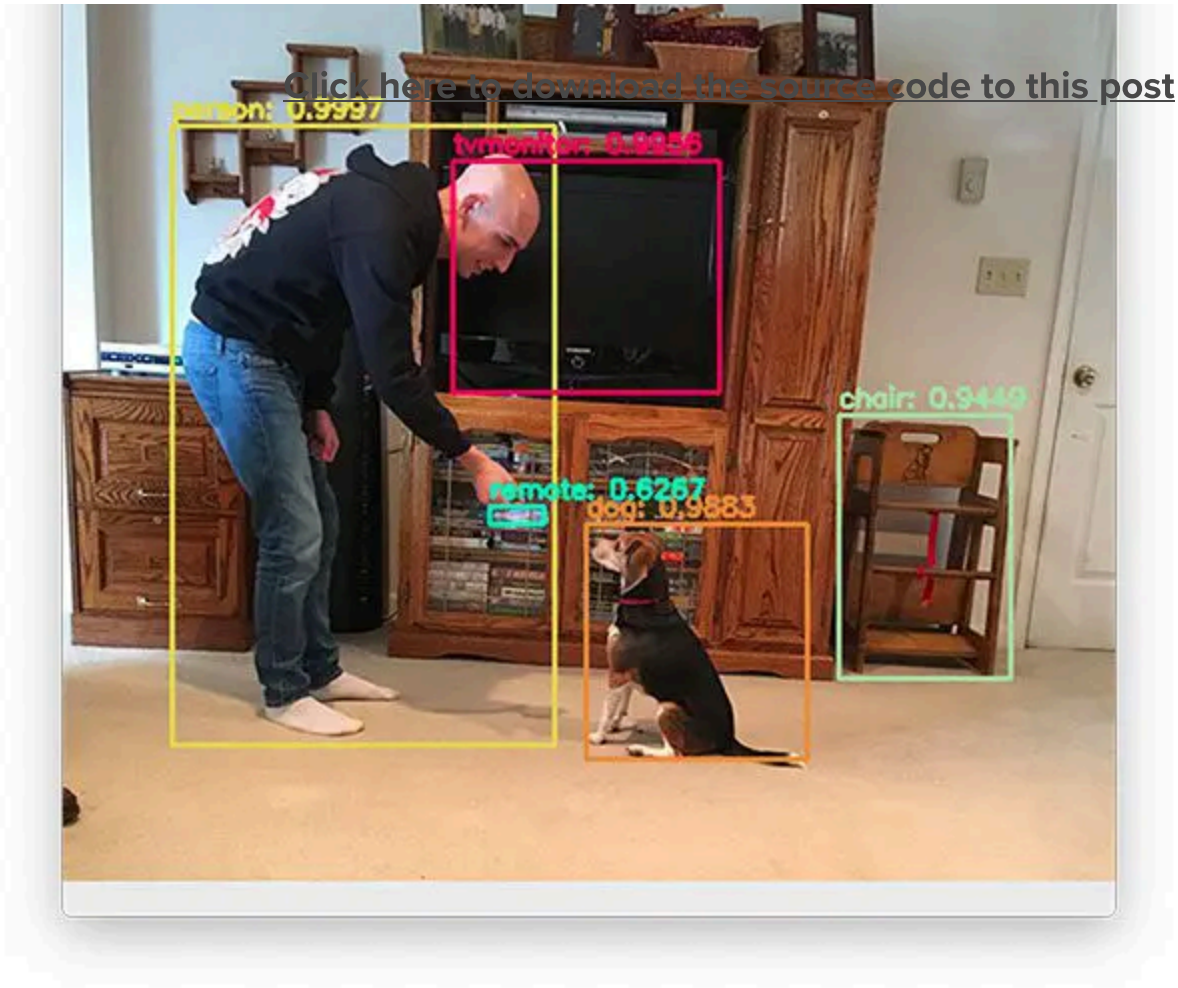
TUTORIALS (HTTPS://PYIMAGESEARCH.COM/CATEGORY/TUTORIALS/)

YOLO object detection with OpenCV

by **Adrian Rosebrock (https://pyimagesearch.com/author/adrian/)** *on* **November 12, 2018**

In this tutorial, you'll learn how to use the YOLO object detector to detect objects in both images and video streams using Deep Learning, OpenCV, and Python.





(https://pyimagesearch.com/wp-content/uploads/2018/11/yolo_living_room_output.jpg)

By applying object detection, you'll not only be able to determine *what* is in an image but also *where* a given object resides!

We'll start with a brief discussion of the YOLO object detector, including how the object detector works.

From there we'll use OpenCV, Python, and deep learning to:

1. Apply the YOLO object detector to images
2. Apply YOLO to video streams

We'll wrap up the tutorial by discussing some of the limitations and drawbacks of the YOLO object detector, including some of my personal tips and suggestions.

YOLO object detector, including some of my personal tips and suggestions.

Click here to download the source code to this post

A dataset with annotated objects is critical for understanding and implementing YOLO object detection. It aids in building a model that can detect and classify various objects in images or videos.

Roboflow (https://roboflow.com/?ref=pyimagesearch) has free tools for each stage of the computer vision pipeline that will streamline your workflows and supercharge your productivity.

Sign up or Log in to your **Roboflow account (https://roboflow.com/?ref=pyimagesearch)** to access state of the art dataset libaries and revolutionize your computer vision pipeline.

You can start by choosing your own datasets or using our **PyimageSearch's assorted library of useful datasets (https://universe.roboflow.com/pyimagesearch?ref=pyimagesearch)**.

Bring data in any of 40+ formats to **Roboflow (https://roboflow.com/?ref=pyimagesearch)**, train using any state-of-the-art model architectures, deploy across multiple platforms (API, NVIDIA, browser, iOS, etc), and connect to applications or 3rd party tools.

To learn how to use YOLO for object detection with OpenCV, *just keep reading!*

- **Update July 2021:** Added section on YOLO v4 and YOLO v5, including how they can be incorporated into OpenCV and PyTorch projects.

Click here to download the source code to this post

Looking for the source code to this post?

JUMP RIGHT TO THE DOWNLOADS SECTION →

YOLO Object detection with OpenCV

In the rest of this tutorial we'll:

- Discuss the YOLO object detector model and architecture
- Utilize YOLO to detect objects in images
- Apply YOLO to detect objects in video streams
- Discuss some of the limitations and drawbacks of the YOLO object detector

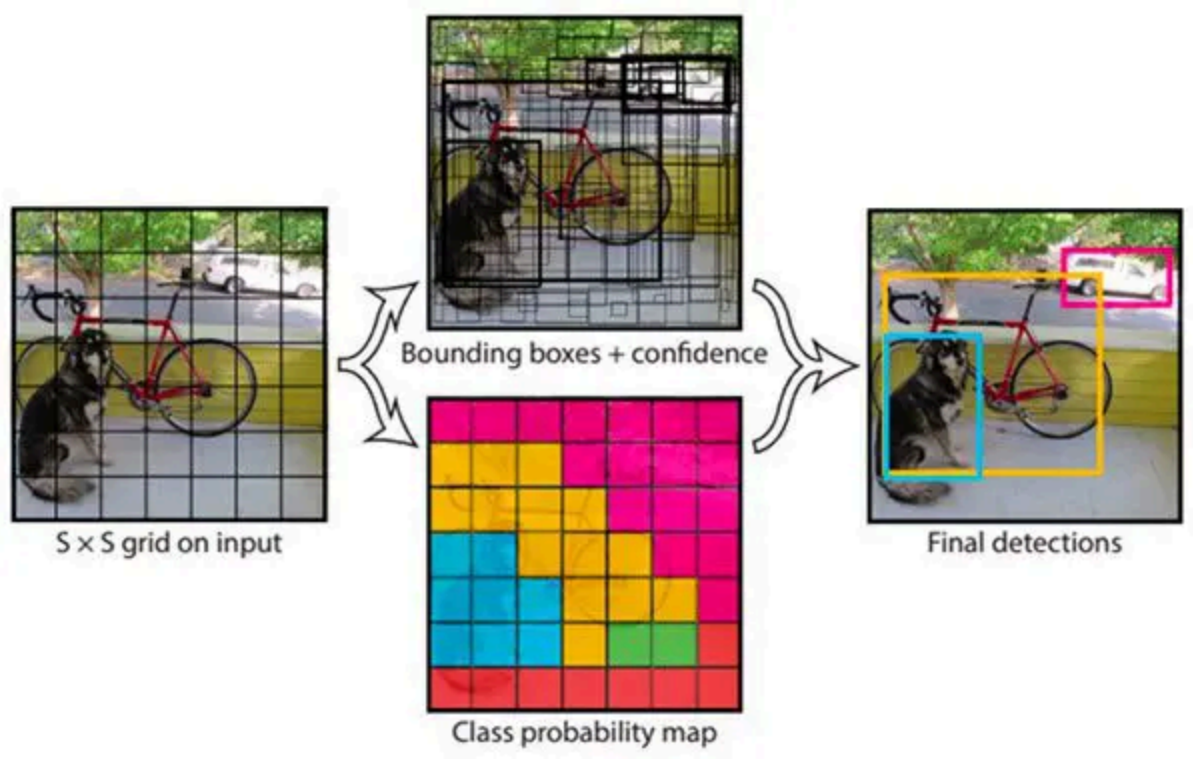
- Discuss some of the limitations and drawbacks of the YOLO object detector

Click here to download the source code to this post

Let's dive in!

Note: This post was last updated on February 5th, 2022 to update images, references, and formatting. Enjoy!

What is the YOLO object detector?



(https://pyimagesearch.com/wp-content/uploads/2018/11/yolo_design.jpg)

Figure 1: A simplified illustration of the YOLO object detector pipeline (**source (https://arxiv.org/abs/1506.02640)**). We'll use YOLO with OpenCV in this blog post.

When it comes to deep learning-based object detection, there are three primary object detectors you'll encounter:

- R-CNN and their variants, including the original R-CNN, Fast R- CNN, and Faster R-

CNN

Click here to download the source code to this post

- Single Shot Detector (SSDs)
- YOLO

R-CNNs are one of the first deep learning-based object detectors and are an example of a ***two-stage detector.***

1. In the first R-CNN publication, ***Rich feature hierarchies for accurate object detection and semantic segmentation*** **(https://arxiv.org/abs/1311.2524)**, (2013) Girshick et al. proposed an object detector that required an algorithm such as **Selective Search (http://www.huppelen.nl/publications/selectiveSearchDraft.pdf)** (or equivalent) to propose candidate bounding boxes that could contain objects.
2. These regions were then passed into a CNN for classification, ultimately leading to one of the first deep learning-based object detectors.

The problem with the standard R-CNN method was that it was *painfully slow* and not a complete end-to-end object detector.

Girshick et al. published a second paper in 2015, entitled ***Fast R- CNN*** **(https://arxiv.org/abs/1504.08083)**. The Fast R-CNN algorithm made considerable improvements to the original R-CNN, namely increasing accuracy and reducing the time it took to perform a forward pass; however, the model still relied on an external region proposal algorithm.

It wasn't until Girshick et al.'s follow-up 2015 paper, ***Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*** **(https://arxiv.org/abs/1506.01497)**, that R-CNNs became a true end-to-end deep learning object detector by removing the Selective Search requirement and instead relying on a Region Proposal Network (RPN)

Selective Search requirement and instead relying on a Region Proposal Network (RPN) that is (1) fully convolutional and (2) can predict the object bounding boxes and "objectness" scores (i.e., a score quantifying how likely it is a region of an image may contain an image). The outputs of the RPNs are then passed into the R-CNN component for final classification and labeling.

Click here to download the source code to this post

While R-CNNs tend to be very accurate, the biggest problem with the R-CNN family of networks is their speed — they were incredibly slow, obtaining only 5 FPS on a GPU.

To help increase the speed of deep learning-based object detectors, both Single Shot Detectors (SSDs) and YOLO use a ***one-stage detector strategy***.

These algorithms treat object detection as a regression problem, taking a given input image and simultaneously learning bounding box coordinates and corresponding class label probabilities.

In general, single-stage detectors tend to be less accurate than two-stage detectors *but are significantly faster.*

YOLO is a great example of a single stage detector.

First introduced in 2015 by Redmon et al., their paper, ***You Only Look Once: Unified, Real-Time Object Detection* (https://arxiv.org/abs/1506.02640)**, details an object detector capable of super real-time object detection, obtaining **45 FPS** on a GPU.

Note: *A smaller variant of their model called "Fast YOLO" claims to achieve 155 FPS on a GPU.*

YOLO has gone through a number of different iterations, including ***YOLO9000: Better, Faster, Stronger* (https://arxiv.org/abs/1612.08242)** (i.e., YOLOv2), capable of detecting over 9,000 object detectors.

Redmon and Farhadi are able to achieve such a large number of object detections by performing joint training for both object detection and classification. Using joint training the authors trained YOLO9000 simultaneously on both the ImageNet classification dataset and COCO detection dataset. The result is a YOLO model, called YOLO9000, that can predict detections for object classes that don't have labeled detection data.

Click here to download the source code to this post

While interesting and novel, YOLOv2's performance was a bit underwhelming given the title and abstract of the paper.

On the 156 class version of COCO, YOLO9000 achieved 16% mean Average Precision (mAP), and yes, while YOLO can detect 9,000 separate classes, the accuracy is not quite what we would desire.

Redmon and Farhadi recently published a new YOLO paper, ***YOLOv3: An Incremental Improvement* (https://arxiv.org/abs/1804.02767)** (2018). YOLOv3 is significantly larger than previous models but is, in my opinion, the best one yet out of the YOLO family of object detectors.

We'll be using YOLOv3 in this blog post, in particular, YOLO trained on the COCO dataset.

The COCO dataset consists of 80 labels, including, but not limited to:

- People
- Bicycles
- Cars and trucks
- Airplanes
- Stop signs and fire hydrants
- Animals, including cats, dogs, birds, horses, cows, and sheep, to name a few
- Kitchen and dining objects, such as wine glasses, cups, forks, knives, spoons, etc.
- *and much more!*

- *... and much more!*

Click here to download the source code to this post

You can find a full list of what YOLO trained on the COCO dataset can detect using this link (https://github.com/pjreddie/darknet/blob/master/data/coco.names).

I'll wrap up this section by saying that any academic needs to read Redmon's YOLO papers and tech reports — not only are they novel and insightful, they are incredibly entertaining as well.

But seriously, if you do nothing else today **read the YOLOv3 tech report (https://arxiv.org/pdf/1804.02767.pdf)**.

It's only 6 pages and one of those pages is just references/citations.

Furthermore, the tech report is honest in a way that academic papers rarely, if ever, are.

Project structure

Let's take a look at today's project layout. You can use your OS's GUI (Finder for OSX, Nautilus for Ubuntu), but you may find it easier and faster to use the `tree` command in your terminal:

```
YOLO Object Detection with OpenCV
$ tree
.
├── images
│   ├── baggage_claim.jpg
│   ├── dining_table.jpg
│   ├── living_room.jpg
│   └── soccer.jpg
├── output
│   ├── airport_output.avi
│   ├── car_chase_01_output.avi
│   ├── car_chase_02_output.avi
│   └── overpass_output.avi
├── videos
│   ├── airport.mp4
│   ├── car_chase_01.mp4
│   ├── car_chase_02.mp4
│   └── overpass.mp4
├── yolo-coco
│   ├── coco.names
│   ├── yolov3.cfg
```

```
21. |   |   └── yolov3.weights
22. |   ├── yolo.py
23. |   └── yolo_video.py
24. |
25. |   4 directories, 19 files
```

Click here to download the source code to this post

Our project today consists of 4 directories and two Python scripts.

The directories (in order of importance) are:

- `yolo-coco/` : The YOLOv3 object detector pre-trained (on the COCO dataset) model files. These were trained by the **Darknet team (https://pjreddie.com/darknet/yolo/)**.
- `images/` : This folder contains four static images which we'll perform object detection on for testing and evaluation purposes.
- `videos/` : After performing object detection with YOLO on images, we'll process videos in real time. This directory contains four sample videos for you to test with.
- `output/` : Output videos that have been processed by YOLO and annotated with bounding boxes and class names can go in this folder.

We're reviewing two Python scripts — `yolo.py` and `yolo_video.py` . The first script is for images and then we'll take what we learn and apply it to video in the second script.

Are you ready?

YOLO object detection in images

Let's get started applying the YOLO object detector to images!

Open up the `yolo.py` file in your project and insert the following code:

```
    YOLO Object Detection with OpenCV
1. | # import the necessary packages
2. | import numpy as np
3. | import argparse
4. | import time
5. | import cv2
```

```
import os

# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required=True,
   help="path to input image")
ap.add_argument("-y", "--yolo", required=True,
   help="base path to YOLO directory")
ap.add_argument("-c", "--confidence", type=float, default=0.5,
   help="minimum probability to filter weak detections")
ap.add_argument("-t", "--threshold", type=float, default=0.3,
   help="threshold when applying non-maxima suppression")
args = vars(ap.parse_args())
```

Click here to download the source code to this post

All you need installed for this script is OpenCV 3.4.2+ with Python bindings. You can find my **OpenCV installation tutorials here (https://pyimagesearch.com/opencv-tutorials-resources-guides/)**, just keep in mind that OpenCV 4 is in beta right now — you may run into issues installing or running certain scripts since it's not an official release. For the time being I recommend going for OpenCV 3.4.2+. You can actually be up and running in less than 5 minutes **with pip (https://pyimagesearch.com/2018/09/19/pip-install-opencv/)** as well.

First, we import our required packages — as long as OpenCV and NumPy are installed, your interpreter will breeze past these lines.

Now let's parse four command line arguments. Command line arguments are processed at runtime and allow us to change the inputs to our script from the terminal. If you aren't familiar with them, I encourage you to read more in my **previous tutorial (https://pyimagesearch.com/2018/03/12/python-argparse-command-line-arguments/)**. Our command line arguments include:

- `--image` : The path to the input image. We'll detect objects in this image using YOLO.
- `--yolo` : The base path to the YOLO directory. Our script will then load the required YOLO files in order to perform object detection on the image.
- `--confidence` : Minimum probability to filter weak detections. I've given this a

default value of 50% (`0.5`), but you should feel free to experiment with this value.

Click here to download the source code to this post

- `--threshold` : This is our non-maxima suppression threshold with a default value of `0.3` . You can read more about **non-maxima suppression here (https://pyimagesearch.com/2014/11/17/non-maximum-suppression-object-detection-python/)**.

After parsing, the `args` variable is now a dictionary containing the key-value pairs for the command line arguments. You'll see `args` a number of times in the rest of this script.

Let's load our class labels and set random colors for each:

```
YOLO Object Detection with OpenCV
20. | # load the COCO class labels our YOLO model was trained on
21. | labelsPath = os.path.sep.join([args["yolo"], "coco.names"])
22. | LABELS = open(labelsPath).read().strip().split("\n")
23. |
24. | # initialize a list of colors to represent each possible class label
25. | np.random.seed(42)
26. | COLORS = np.random.randint(0, 255, size=(len(LABELS), 3),
27. |    dtype="uint8")
```

Here we load all of our class `LABELS` (notice the first command line argument, `args["yolo"]` being used) on **Lines 21 and 22**. Random `COLORS` are then assigned to each label on **Lines 25-27**.

Let's derive the paths to the YOLO weights and configuration files followed by loading YOLO from disk:

```
YOLO Object Detection with OpenCV
29. | # derive the paths to the YOLO weights and model configuration
30. | weightsPath = os.path.sep.join([args["yolo"], "yolov3.weights"])
31. | configPath = os.path.sep.join([args["yolo"], "yolov3.cfg"])
32. |
33. | # load our YOLO object detector trained on COCO dataset (80 classes)
34. | print("[INFO] loading YOLO from disk...")
35. | net = cv2.dnn.readNetFromDarknet(configPath, weightsPath)
```

To load YOLO from disk on **Line 35**, we'll take advantage of OpenCV's DNN function called `cv2.dnn.readNetFromDarknet` . This function requires both a `configPath` and

`weightsPath` which are established via command line arguments on **Lines 30 and 31**.

Click here to download the source code to this post

I cannot stress this enough: you'll need at least OpenCV 3.4.2 to run this code as it has the updated `dnn` module required to load YOLO.

Let's load the image and send it through the network:

```
      YOLO Object Detection with OpenCV
37. | # load our input image and grab its spatial dimensions
38. | image = cv2.imread(args["image"])
39. | (H, W) = image.shape[:2]
40. |
41. | # determine only the *output* layer names that we need from YOLO
42. | ln = net.getLayerNames()
43. | ln = [ln[i[0] - 1] for i in net.getUnconnectedOutLayers()]
44. |
45. | # construct a blob from the input image and then perform a forward
46. | # pass of the YOLO object detector, giving us our bounding boxes and
47. | # associated probabilities
48. | blob = cv2.dnn.blobFromImage(image, 1 / 255.0, (416, 416),
49. |    swapRB=True, crop=False)
50. | net.setInput(blob)
51. | start = time.time()
52. | layerOutputs = net.forward(ln)
53. | end = time.time()
54. |
55. | # show timing information on YOLO
56. | print("[INFO] YOLO took {:.6f} seconds".format(end - start))
```

In this block we:

- Load the input `image` and extract its dimensions (**Lines 38 and 39**).
- Determine the output layer names from the YOLO model (**Lines 42 and 43**).
- Construct a `blob` from the image (**Lines 48 and 49**). Are you confused about what a blob is or what the `cv2.dnn.blobFromImage` does? Give **this blog post (https://pyimagesearch.com/2017/11/06/deep-learning-opencvs-blobfromimage-works/)** a read.

Now that our blob is prepared, we'll

- Perform a forward pass through our YOLO network (**Lines 50 and 52**)
- Show the inference time for YOLO (**Line 56**)

- Show the inference time for YOLO (**Line 56**)

Click here to download the source code to this post

What good is object detection unless we visualize our results? Let's take steps now to filter and visualize our results.

But first, let's initialize some lists we'll need in the process of doing so:

```
YOLO Object Detection with OpenCV
58. | # initialize our lists of detected bounding boxes, confidences, and
59. | # class IDs, respectively
60. | boxes = []
61. | confidences = []
62. | classIDs = []
```

These lists include:

- `boxes` : Our bounding boxes around the object.
- `confidences` : The confidence value that YOLO assigns to an object. Lower confidence values indicate that the object might not be what the network thinks it is. Remember from our command line arguments above that we'll filter out objects that don't meet the `0.5` threshold.
- `classIDs` : The detected object's class label.

Let's begin populating these lists with data from our YOLO `layerOutputs` :

```
YOLO Object Detection with OpenCV
64. | # loop over each of the layer outputs
65. | for output in layerOutputs:
66. |     # loop over each of the detections
67. |     for detection in output:
68. |         # extract the class ID and confidence (i.e., probability) of
69. |         # the current object detection
70. |         scores = detection[5:]
71. |         classID = np.argmax(scores)
72. |         confidence = scores[classID]
73. |
74. |         # filter out weak predictions by ensuring the detected
75. |         # probability is greater than the minimum probability
76. |         if confidence > args["confidence"]:
77. |             # scale the bounding box coordinates back relative to the
78. |             # size of the image, keeping in mind that YOLO actually
79. |             # returns the center (x, y)-coordinates of the bounding
80. |             # box followed by the box's width and height
```

```
80. |             # box followed by the boxes' width and height
81. |             box = detection[0:4] * np.array([W, H, W, H])
82. |             (centerX, centerY, width, height) = box.astype("int")
83. |
84. |             # use the center (x, y)-coordinates to derive the top and
85. |             # and left corner of the bounding box
86. |             x = int(centerX - (width / 2))
87. |             y = int(centerY - (height / 2))
88. |
89. |             # update our list of bounding box coordinates, confidences,
90. |             # and class IDs
91. |             boxes.append([x, y, int(width), int(height)])
92. |             confidences.append(float(confidence))
93. |             classIDs.append(classID)
```

Click here to download the source code to this post

There's a lot here in this code block — let's break it down.

In this block, we:

- Loop over each of the `layerOutputs` (beginning on **Line 65**).
- Loop over each `detection` in `output` (a nested loop beginning on **Line 67**).
- Extract the `classID` and `confidence` (**Lines 70-72**).
- Use the `confidence` to filter out weak detections (**Line 76**).

Now that we've filtered out unwanted detections, we're going to:

- Scale bounding box coordinates so we can display them properly on our original image (**Line 81**).
- Extract coordinates and dimensions of the bounding box (**Line 82**). YOLO returns bounding box coordinates in the form: `(centerX, centerY, width, and height)` .
- Use this information to derive the top-left *(x, y)*-coordinates of the bounding box (**Lines 86 and 87**).
- Update the `boxes` , `confidences` , and `classIDs` lists (**Lines 91-93**).

With this data, we're now going to apply what is called "non-maxima suppression":

```
YOLO Object Detection with OpenCV
# apply non-maxima suppression to suppress weak, overlapping bounding
# boxes
idxs = cv2.dnn.NMSBoxes(boxes, confidences, args["confidence"],
    args["threshold"])
```

Click here to download the source code to this post

YOLO does not apply **non-maxima suppression (https://pyimagesearch.com/2014/11/17/non-maximum-suppression-object-detection-python/)** for us, so we need to explicitly apply it.

Applying non-maxima suppression suppresses significantly overlapping bounding boxes, keeping only the most confident ones.

NMS also ensures that we do not have any redundant or extraneous bounding boxes.

Taking advantage of OpenCV's built-in DNN module implementation of NMS, we perform non-maxima suppression on **Lines 97 and 98**. All that is required is that we submit our bounding `boxes`, `confidences`, as well as both our confidence threshold and NMS threshold.

If you've been reading this blog, you might be wondering why we didn't use my **imutils implementation of NMS (https://github.com/jrosebr1/imutils/blob/master/imutils/object_detection.py#L4)**. The primary reason is that the `NMSBoxes` function is now working in OpenCV. Previously it failed for some inputs and resulted in an error message. Now that the `NMSBoxes` function is working, we can use it in our own scripts.

Let's draw the boxes and class text on the image!

```
YOLO Object Detection with OpenCV
# ensure at least one detection exists
if len(idxs) > 0:
    # loop over the indexes we are keeping
    for i in idxs.flatten():
        # extract the bounding box coordinates
        (x, y) = (boxes[i][0], boxes[i][1])
        (w, h) = (boxes[i][2], boxes[i][3])

        # draw a bounding box rectangle and label on the image
        color = [int(c) for c in COLORS[classIDs[i]]]
        cv2.rectangle(image, (x, y), (x + w, y + h), color, 2)
```

```
111. |         text = "{}: {:.4f}".format(LABELS[classIDs[i]], confidences[i])
112. |         cv2.putText(image, text, (x, y - 5), cv2.FONT_HERSHEY_SIMPLEX,
113. |             0.5, color, 2)
114. |
115. |  # show the output image
116. |  cv2.imshow("Image", image)
117. |  cv2.waitKey(0)
```

Click here to download the source code to this post

Assuming at least one detection exists (**Line 101**), we proceed to loop over `idxs` determined by non-maxima suppression.

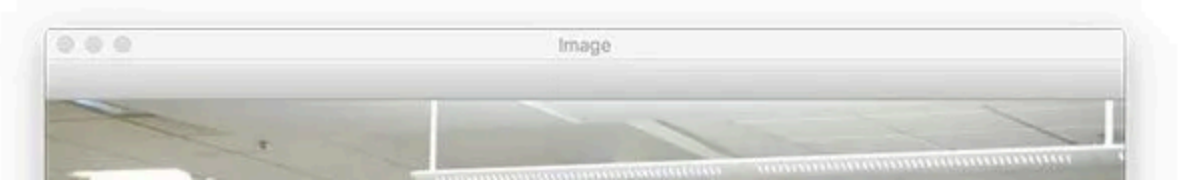
Then, we simply draw the bounding box and text on `image` using our random class colors (**Lines 105-113**).

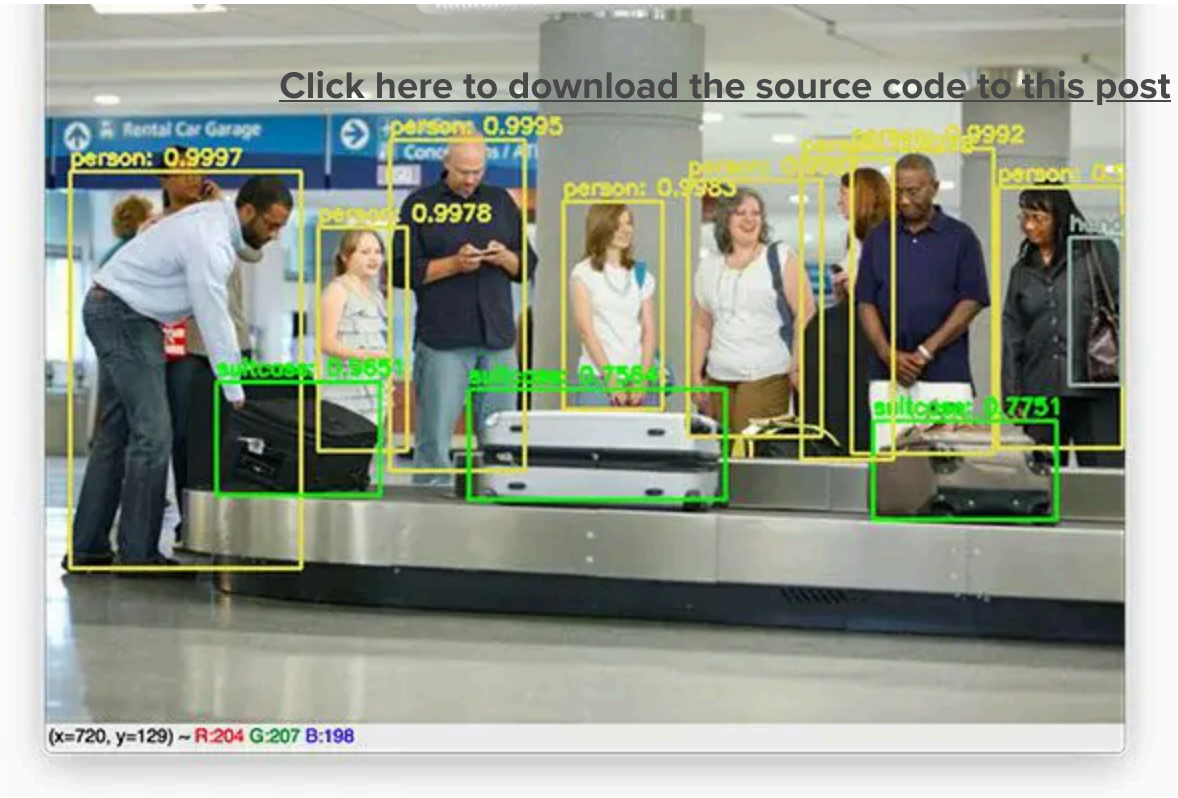
Finally, we display our resulting image until the user presses any key on their keyboard (ensuring the window opened by OpenCV is selected and focused).

To follow along with this guide, make sure you use the ***"Downloads"*** section of this tutorial to download the source code, YOLO model, and example images.

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

```
YOLO Object Detection with OpenCV
1. |  $ python yolo.py --image images/baggage_claim.jpg --yolo yolo-coco
2. |  [INFO] loading YOLO from disk...
3. |  [INFO] YOLO took 0.347815 seconds
```





(https://pyimagesearch.com/wp-content/uploads/2018/11/yolo_baggage_claim_output.jpg)

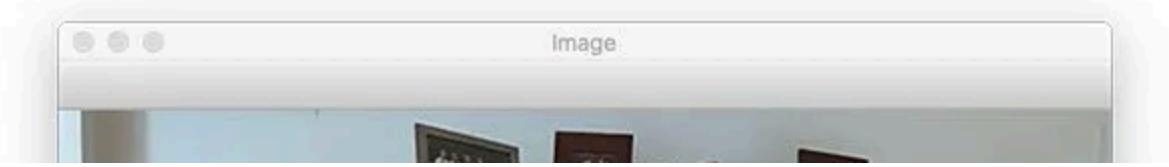
Figure 2: YOLO with OpenCV is used to detect people and baggage in an airport.

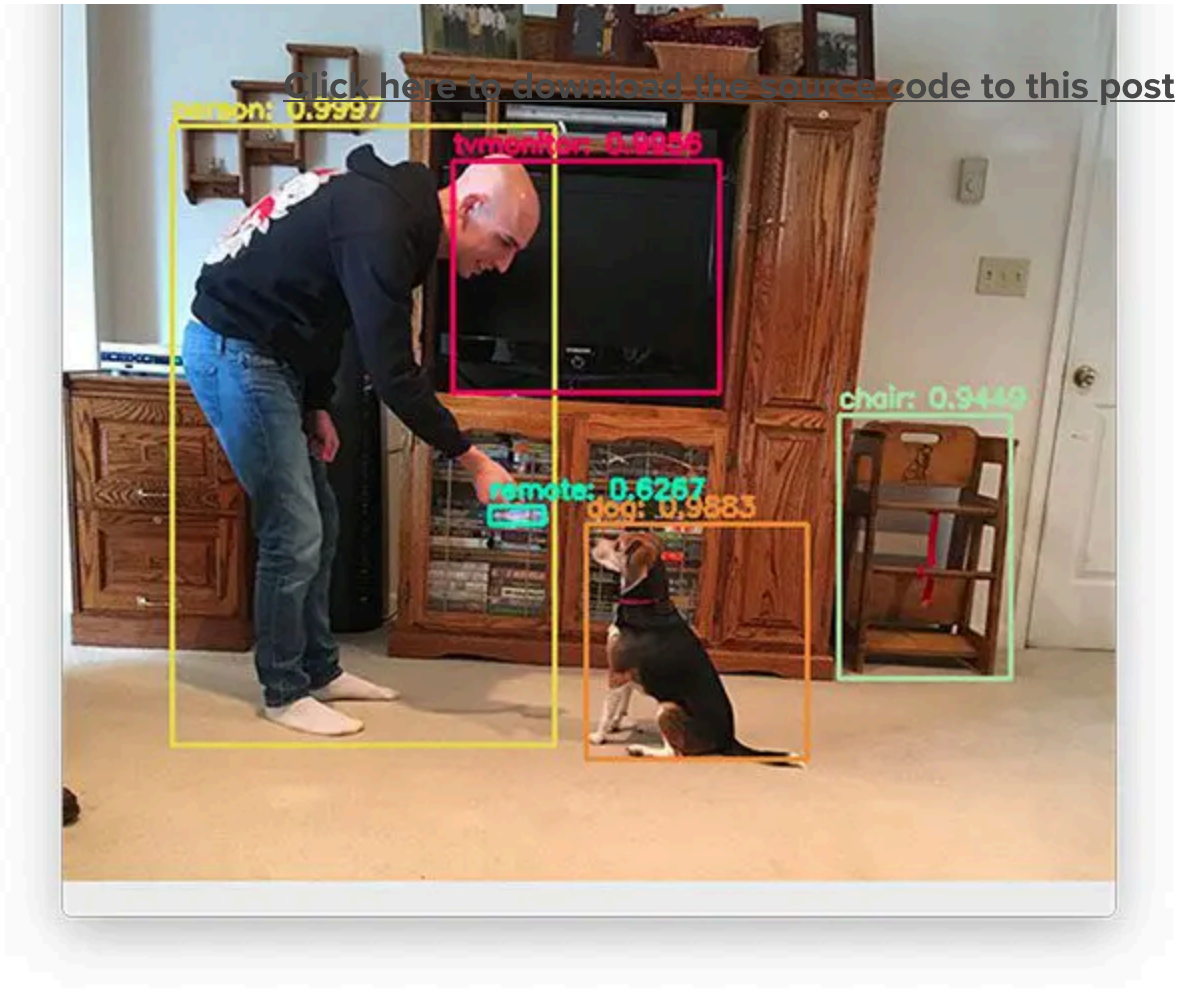
Here you can see that YOLO has not only detected each person in the input image, but also the suitcases as well!

Furthermore, if you take a look at the right corner of the image you'll see that YOLO has also detected the handbag on the lady's shoulder.

Let's try another example:

```
YOLO Object Detection with OpenCV
$ python yolo.py --image images/living_room.jpg --yolo yolo-coco
[INFO] loading YOLO from disk...
[INFO] YOLO took 0.340221 seconds
```





(https://pyimagesearch.com/wp-content/uploads/2018/11/yolo_living_room_output.jpg)

Figure 3: YOLO object detection with OpenCV is used to detect a person, dog, TV, and chair. The remote is a false-positive detection but looking at the ROI you could imagine that the area does share resemblances to a remote.

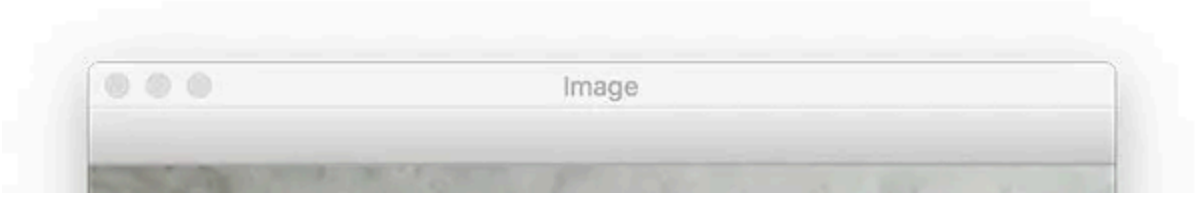
The image above contains a person (myself) and a dog (Jemma, the family beagle).

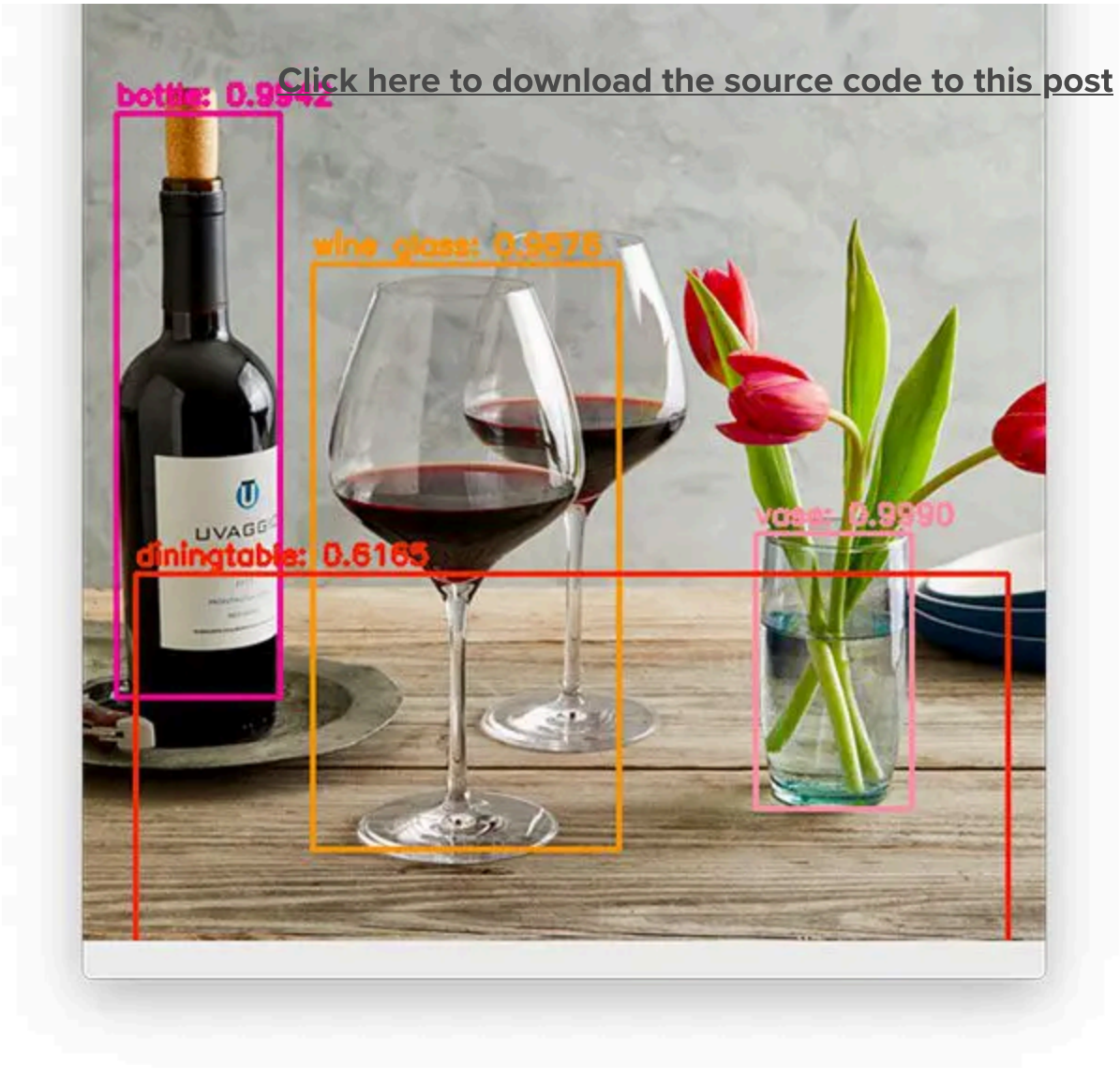
YOLO also detects the TV monitor and a chair as well. I'm particularly impressed that YOLO was able to detect the chair given that it's handmade, old fashioned "baby high chair".

Interestingly, YOLO thinks there is a "remote" in my hand. It's actually not a remote — it's the reflection of glass on a VHS tape; however, if you stare at the region it actually does look like it could be a remote.

The following example image demonstrates a limitation and weakness of the YOLO object detector: **Click here to download the source code to this post**

```
YOLO Object Detection with OpenCV
$ python yolo.py --image images/dining_table.jpg --yolo yolo-coco
[INFO] loading YOLO from disk...
[INFO] YOLO took 0.362369 seconds
```





(https://pyimagesearch.com/wp-content/uploads/2018/11/yolo_dining_table_output.jpg)

Figure 4: YOLO and OpenCV are used for object detection of a dining room table.

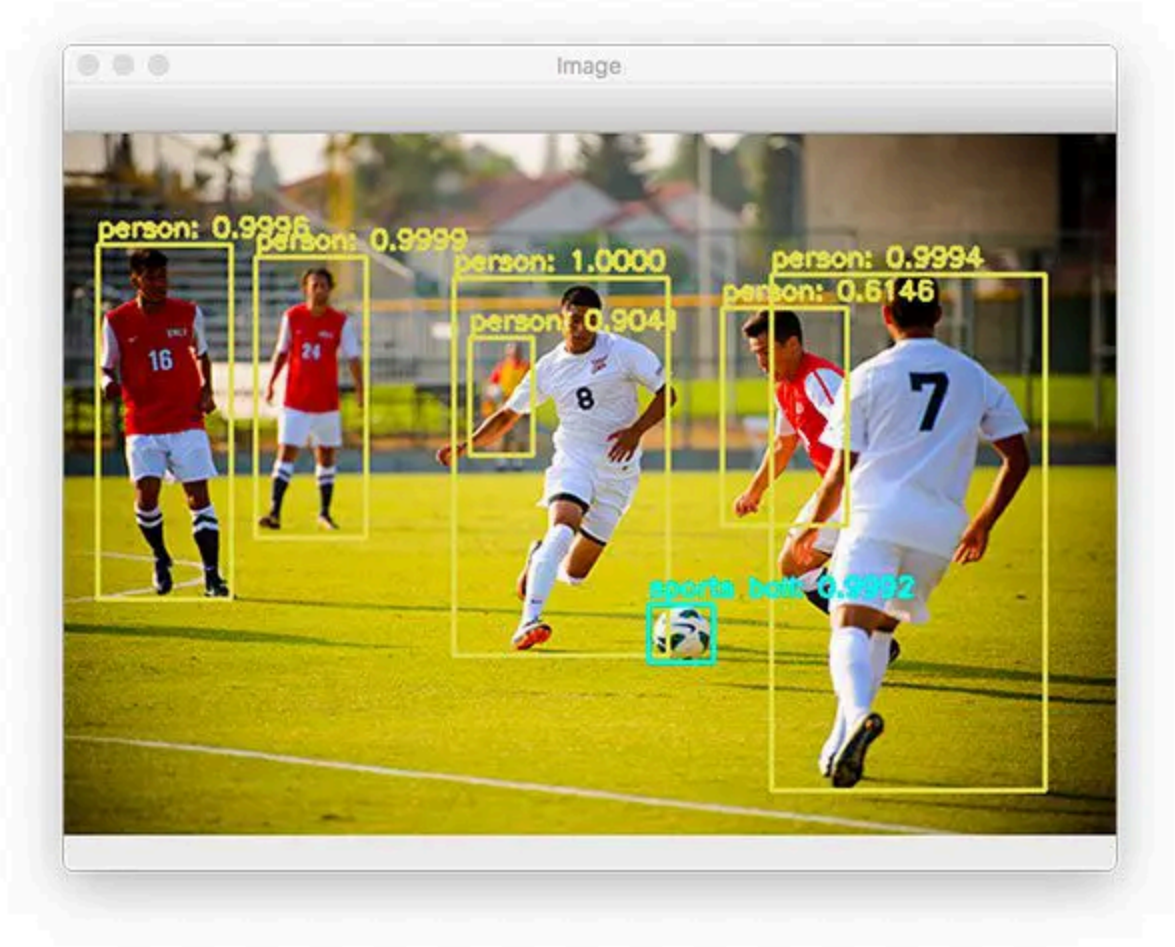
While both the wine bottle, dining table, and vase are correctly detected by YOLO, only one of the two wine glasses is properly detected.

We discuss why YOLO struggles with objects close together in the *"Limitations and drawbacks of the YOLO object detector"* section below.

Let's try one final image:

```
YOLO Object Detection with OpenCV
$ python yolo.py --image images/soccer.jpg --yolo yolo-coco
[INFO] loading YOLO from disk...
[INFO] YOLO took 0.345656 seconds
```

Click here to download the source code to this post



(https://pyimagesearch.com/wp-content/uploads/2018/11/yolo_soccer_output.jpg)

Figure 5: Soccer players and a soccer ball are detected with OpenCV using the YOLO object detector.

YOLO is able to correctly detect each of the players on the pitch, including the soccer ball itself. Notice the person in the background who is detected despite the area being highly blurred and partially obscured.

YOLO object detection in video streams

Now that we've learned how to apply the YOLO object detector to single images, let's also utilize YOLO to perform object detection in input video files as well.

Open up the `yolo_video.py` file and insert the following code:

Click here to download the source code to this post

```
YOLO Object Detection with OpenCV
 1. # import the necessary packages
 2. import numpy as np
 3. import argparse
 4. import imutils
 5. import time
 6. import cv2
 7. import os
 8.
 9. # construct the argument parse and parse the arguments
10. ap = argparse.ArgumentParser()
11. ap.add_argument("-i", "--input", required=True,
12.    help="path to input video")
13. ap.add_argument("-o", "--output", required=True,
14.    help="path to output video")
15. ap.add_argument("-y", "--yolo", required=True,
16.    help="base path to YOLO directory")
17. ap.add_argument("-c", "--confidence", type=float, default=0.5,
18.    help="minimum probability to filter weak detections")
19. ap.add_argument("-t", "--threshold", type=float, default=0.3,
20.    help="threshold when applying non-maxima suppression")
21. args = vars(ap.parse_args())
```

We begin with our imports and command line arguments.

Notice that this script doesn't have the `--image` argument as before. To take its place, we now have two video-related arguments:

- `--input` : The path to the *input* video file.
- `--output` : Our path to the *output* video file.

Given these arguments, you can now use videos that you record of scenes with your smartphone or videos you find online. You can then process the video file producing an annotated output video. Of course if you want to use your webcam to process a live

video stream, that is possible too. Just find examples on PyImageSearch where the `VideoStream` class from `imutils.video` is utilized and make some minor changes.

Moving on, the next block is *identical* to the block from the YOLO image processing

script:

Click here to download the source code to this post

```
YOLO Object Detection with OpenCV
# load the COCO class labels our YOLO model was trained on
labelsPath = os.path.sep.join([args["yolo"], "coco.names"])
LABELS = open(labelsPath).read().strip().split("\n")

# initialize a list of colors to represent each possible class label
np.random.seed(42)
COLORS = np.random.randint(0, 255, size=(len(LABELS), 3),
    dtype="uint8")

# derive the paths to the YOLO weights and model configuration
weightsPath = os.path.sep.join([args["yolo"], "yolov3.weights"])
configPath = os.path.sep.join([args["yolo"], "yolov3.cfg"])

# load our YOLO object detector trained on COCO dataset (80 classes)
# and determine only the *output* layer names that we need from YOLO
print("[INFO] loading YOLO from disk...")
net = cv2.dnn.readNetFromDarknet(configPath, weightsPath)
ln = net.getLayerNames()
ln = [ln[i[0] - 1] for i in net.getUnconnectedOutLayers()]
```

Here we load labels and generate colors followed by loading our YOLO model and determining output layer names.

Next, we'll take care of some video-specific tasks:

```
YOLO Object Detection with OpenCV
# initialize the video stream, pointer to output video file, and
# frame dimensions
vs = cv2.VideoCapture(args["input"])
writer = None
(W, H) = (None, None)

# try to determine the total number of frames in the video file
try:
    prop = cv2.cv.CV_CAP_PROP_FRAME_COUNT if imutils.is_cv2() \
        else cv2.CAP_PROP_FRAME_COUNT
    total = int(vs.get(prop))
    print("[INFO] {} total frames in video".format(total))

# an error occurred while trying to determine the total
# number of frames in the video file
except:
    print("[INFO] could not determine # of frames in video")
    print("[INFO] no approx. completion time can be provided")
    total = -1
```

In this block, we:

- Open a file pointer to the video file for reading frames in the upcoming loop (**Line 45**). **Click here to download the source code to this post**
- Initialize our video `writer` and frame dimensions (**Lines 46 and 47**).
- Try to determine the `total` number of frames in the video file so we can estimate how long processing the entire video will take (**Lines 50-61**).

Now we're ready to start processing frames one by one:

```
YOLO Object Detection with OpenCV
63. | # loop over frames from the video file stream
64. | while True:
65. |     # read the next frame from the file
66. |     (grabbed, frame) = vs.read()
67. |
68. |     # if the frame was not grabbed, then we have reached the end
69. |     # of the stream
70. |     if not grabbed:
71. |         break
72. |
73. |     # if the frame dimensions are empty, grab them
74. |     if W is None or H is None:
75. |         (H, W) = frame.shape[:2]
```

We define a `while` loop (**Line 64**) and then we grab our first frame (**Line 66**).

We make a check to see if it is the last frame of the video. If so we need to `break` from the `while` loop (**Lines 70 and 71**).

Next, we grab the frame dimensions if they haven't been grabbed yet (**Lines 74 and 75**).

Next, let's perform a forward pass of YOLO, using our current `frame` as the input:

```
YOLO Object Detection with OpenCV
77. |     # construct a blob from the input frame and then perform a forward
78. |     # pass of the YOLO object detector, giving us our bounding boxes
79. |     # and associated probabilities
80. |     blob = cv2.dnn.blobFromImage(frame, 1 / 255.0, (416, 416),

81. |         swapRB=True, crop=False)
82. |     net.setInput(blob)
83. |     start = time.time()
84. |     layerOutputs = net.forward(ln)
85. |     end = time.time()
86. |
```

```
87. |     # initialize our lists of detected bounding boxes, confidences,
88. |     # and class IDs, respectively
89. |     boxes = []
90. |     confidences = []
91. |     classIDs = []
```

Click here to download the source code to this post

Here we construct a `blob` and pass it through the network, obtaining predictions. I've surrounded the forward pass operation with time stamps so we can calculate the elapsed time to make predictions on one frame — this will help us estimate the time needed to process the entire video.

We'll then go ahead and initialize the same three lists we used in our previous script: `boxes`, `confidences`, and `classIDs`.

This next block is, again, *identical* to our previous script:

```
    YOLO Object Detection with OpenCV
93. |     # loop over each of the layer outputs
94. |     for output in layerOutputs:
95. |         # loop over each of the detections
96. |         for detection in output:
97. |             # extract the class ID and confidence (i.e., probability)
98. |             # of the current object detection
99. |             scores = detection[5:]
100. |            classID = np.argmax(scores)
101. |            confidence = scores[classID]
102. |
103. |            # filter out weak predictions by ensuring the detected
104. |            # probability is greater than the minimum probability
105. |            if confidence > args["confidence"]:
106. |                # scale the bounding box coordinates back relative to
107. |                # the size of the image, keeping in mind that YOLO
108. |                # actually returns the center (x, y)-coordinates of
109. |                # the bounding box followed by the boxes' width and
110. |                # height
111. |                box = detection[0:4] * np.array([W, H, W, H])
112. |                (centerX, centerY, width, height) = box.astype("int")
113. |
114. |                # use the center (x, y)-coordinates to derive the top
115. |                # and and left corner of the bounding box
116. |                x = int(centerX - (width / 2))
117. |                y = int(centerY - (height / 2))
118. |
119. |                # update our list of bounding box coordinates,
120. |                # confidences, and class IDs
121. |                boxes.append([x, y, int(width), int(height)])
122. |                confidences.append(float(confidence))
123. |                classIDs.append(classID)
```

In this code block, we:

In this code block, we:

- Loop over output layers and detections (**Lines 94-96**)

- Extract the `classID` and filter out weak predictions (**Lines 99-105**).
- Compute bounding box coordinates (**Lines 111-117**).
- Update our respective lists (**Lines 121-123**).

Next, we'll apply non-maxima suppression and begin to proceed to annotate the frame:

```
YOLO Object Detection with OpenCV
125. |     # apply non-maxima suppression to suppress weak, overlapping
126. |     # bounding boxes
127. |     idxs = cv2.dnn.NMSBoxes(boxes, confidences, args["confidence"],
128. |         args["threshold"])
129. |
130. |     # ensure at least one detection exists
131. |     if len(idxs) > 0:
132. |         # loop over the indexes we are keeping
133. |         for i in idxs.flatten():
134. |             # extract the bounding box coordinates
135. |             (x, y) = (boxes[i][0], boxes[i][1])
136. |             (w, h) = (boxes[i][2], boxes[i][3])
137. |
138. |             # draw a bounding box rectangle and label on the frame
139. |             color = [int(c) for c in COLORS[classIDs[i]]]
140. |             cv2.rectangle(frame, (x, y), (x + w, y + h), color, 2)
141. |             text = "{}: {:.4f}".format(LABELS[classIDs[i]],
142. |                 confidences[i])
143. |             cv2.putText(frame, text, (x, y - 5),
144. |                 cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)
```

You should recognize these lines as well. Here we:

- Apply NMS using the `cv2.dnn.NMSBoxes` function (**Lines 127 and 128**) to suppress weak, overlapping bounding boxes. You can read more about **non-maxima suppression here (https://pyimagesearch.com/2014/11/17/non-maximum-suppression-object-detection-python/)**.
- Loop over the `idxs` calculated by NMS and draw the corresponding bounding boxes + labels (**Lines 131-144**).

Let's finish out the script:

Let's finish out the script:


```
YOLO Object Detection with OpenCV
146. |     # check if the video writer is None
147. |     if writer is None:
148. |         # initialize our video writer
149. |         fourcc = cv2.VideoWriter_fourcc(*"MJPG")
150. |         writer = cv2.VideoWriter(args["output"], fourcc, 30,
151. |             (frame.shape[1], frame.shape[0]), True)
152. |
153. |         # some information on processing single frame
154. |         if total > 0:
155. |             elap = (end - start)
156. |             print("[INFO] single frame took {:.4f} seconds".format(elap))
157. |             print("[INFO] estimated total time to finish: {:.4f}".format(
158. |                 elap * total))
159. |
160. |     # write the output frame to disk
161. |     writer.write(frame)
162. |
163. |  # release the file pointers
164. |  print("[INFO] cleaning up...")
165. |  writer.release()
166. |  vs.release()
```

To wrap up, we simply:

- Initialize our video `writer` if necessary (**Lines 147-151**). The `writer` will be initialized on the first iteration of the loop.
- Print out our estimates of how long it will take to process the video (**Lines 154-158**).
- Write the `frame` to the output video file (**Line 161**).
- Cleanup and release pointers (**Lines 165 and 166**).

To apply YOLO object detection to video streams, make sure you use the ***"Downloads"*** section of this blog post to download the source, YOLO object detector, and example videos.

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

```
YOLO Object Detection with OpenCV
1. |  $ python yolo_video.py --input videos/car_chase_01.mp4 \
2. |     --output output/car_chase_01.avi --yolo yolo-coco
```

```
[INFO] loading YOLO from disk...
[INFO] 583 total frames in video
[INFO] single frame took 0.3500 seconds
[INFO] estimated total time to finish: 204.0238
[INFO] cleaning up...
```

Click here to download the source code to this post

When you execute the commands above you will see a GIF excerpt from **a car chase video I found on YouTube (https://www.youtube.com/watch?v=7K3dZmrcriU)**. Why not share it with me on twitter @pyimagesearch?

In the video/GIF, you can see not only the vehicles being detected, but people, as well as the traffic lights, are detected too!

The YOLO object detector is performing quite well here. Let's try a different video clip from the same car chase video:

```
YOLO Object Detection with OpenCV
$ python yolo_video.py --input videos/car_chase_02.mp4 \
    --output output/car_chase_02.avi --yolo yolo-coco
[INFO] loading YOLO from disk...
[INFO] 3132 total frames in video
[INFO] single frame took 0.3455 seconds
[INFO] estimated total time to finish: 1082.0806
[INFO] cleaning up...
```

When you run the commands above you will see a suspect on the run, we have used OpenCV and YOLO object detection to find the person! Share your output with me again on Twitter @pyimagesearch.

YOLO is once again able to detect people.

At one point the suspect is actually able to make it back to their car and continue the chase — let's see how YOLO performs there as well:

```
YOLO Object Detection with OpenCV
$ python yolo_video.py --input videos/car_chase_03.mp4 \
    --output output/car_chase_03.avi --yolo yolo-coco
[INFO] loading YOLO from disk...
[INFO] 749 total frames in video
[INFO] single frame took 0.3442 seconds
[INFO] estimated total time to finish: 257.8418
[INFO] cleaning up...
```

As a final example, run the code above and you'll see you can use YOLO as a starting

As a final example, run the code above and you'll see you can use YOLO as a starting point to building a traffic counter. Can you think of a great application that uses traffic counting?

Click here to download the source code to this post

I can — how about peak traffic detectors to notify people when to start a commute using traffic counting. Another option is to use traffic counting in order to measure the number of inbound customers to a brick and mortar business.

Bingo! There's a business waiting to be launched right there. Be sure to run the script below and see the output.

```
YOLO Object Detection with OpenCV
$ python yolo_video.py --input videos/overpass.mp4 \
   --output output/overpass.avi --yolo yolo-coco
[INFO] loading YOLO from disk...
[INFO] 812 total frames in video
[INFO] single frame took 0.3534 seconds
[INFO] estimated total time to finish: 286.9583
[INFO] cleaning up...
```

I've put together a full video of YOLO object detection examples below:

Click here to download the source code to this post

Credits for video and audio:

- Car chase video posted on **YouTube by Quaker Oats (https://www.youtube.com/watch?v=7K3dZmrcriU)**.
- Overpass video on **YouTube by Vlad Kiraly (https://www.youtube.com/watch?v=nt3D26Irkho)**.
- "White Crow" on the **FreeMusicArchive by XTaKeRuX (https://freemusicarchive.org/music/XTaKeRuX/Empty_Grave/White_Crow)**.

Limitations and drawbacks of the YOLO object detector

Arguably the largest limitation and drawback of the YOLO object detector is that:

1. It does not always handle small objects well
2. It *especially* does not handle objects grouped close together

The reason for this limitation is due to the YOLO algorithm itself:

- The YOLO object detector divides an input image into an *SxS* grid where each cell in the grid predicts only a single object.

- If there exist multiple, small objects in a single cell then YOLO will be unable to detect them, ultimately leading to missed object detections.

Click here to download the source code to this post

Therefore, if you know your dataset consists of many small objects grouped close together then you should not use the YOLO object detector.

In terms of small objects, Faster R-CNN tends to work the best; however, it's also the slowest.

SSDs can also be used here; however, SSDs can also struggle with smaller objects (but not as much as YOLO).

SSDs often give a nice tradeoff in terms of speed and accuracy as well.

It's also worth noting that YOLO ran slower than SSDs in this tutorial. In my previous tutorial on **OpenCV object detection (https://pyimagesearch.com/2017/09/11/object-detection-with-deep-learning-and-opencv/)**, we utilized an SSD — a single forward pass of the SSD took ~0.03 seconds.

However, from this tutorial, we know that a forward pass of the YOLO object detector took ≈0.3 seconds, *approximately, an order of magnitude slower!*

If you're using the pre-trained deep learning object detectors OpenCV supplies you may want to consider using SSDs over YOLO. From my personal experience, I've rarely encountered situations where I needed to use YOLO over SSDs:

- I have found SSDs much easier to train and their performance in terms of accuracy almost always outperforms YOLO (at least for the datasets I've worked with).

- YOLO may have excellent results on the COCO dataset; however, I have not found that same level of accuracy for my own tasks.

I therefore, tend to use the following guidelines when picking an object detector for a

I, therefore, tend to use the following guidelines when picking an object detector for a given problem:

Click here to download the source code to this post

1. If I know I need to detect small objects and speed is not a concern, I tend to use Faster R-CNN.
2. If speed is absolutely paramount, I use YOLO.
3. If I need a middle ground, I tend to go with SSDs.

In most of my situations I end up using SSDs or RetinaNet — both are a great balance between the YOLO/Faster R-CNN.

Alternative YOLO object detection models

We utilized YOLO v3 inside this tutorial to perform YOLO object detection with OpenCV.

Joseph Redmon, the creator of the YOLO object detector, **has ceased working on YOLO due to privacy concerns and misuse in military applications (https://twitter.com/pjreddie/status/1230524770350817280)**; however, other researchers in the computer vision and deep learning community have continued his work.

While these are not "official" YOLO models (in the sense that Joseph Redmon did not create them nor does he endorse them), you will find publications and references to both YOLO v4 and YOLO v5 online:

1. **YOLO v4: Optimal Speed and Accuracy of Object Detection (https://arxiv.org/abs/2004.10934)**
2. **YOLO v5: PyTorch compatible (https://github.com/ultralytics/yolov5)**

If you use the PyTorch deep learning library, then definitely check out YOLO v5 — the library makes it super easy to train custom YOLO models; however, the output YOLO v5 models are *not* directly compatible with OpenCV (i.e., you'll need to write additional code to make predictions on images/frames if you're using OpenCV and YOLO v5

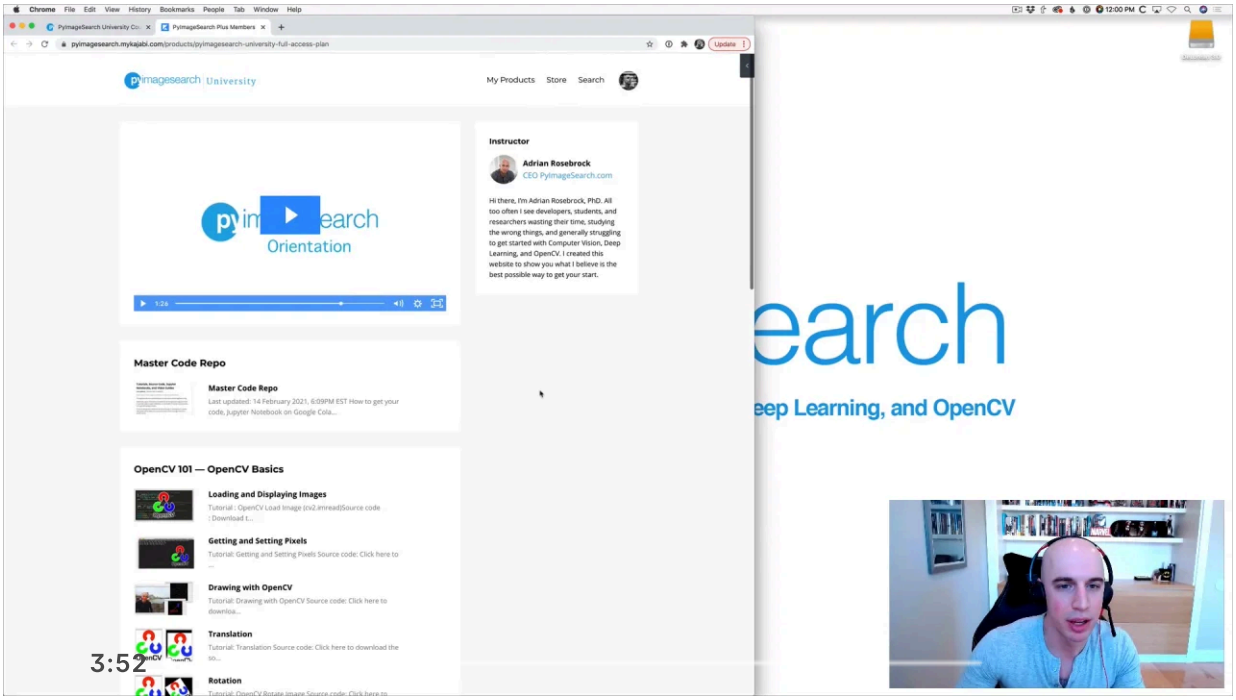
code to make predictions on images/frames if you're using OpenCV and YOLO v3 together).

Click here to download the source code to this post

YOLO v4, on the other hand, *is* compatible with OpenCV using the same code provided in this tutorial. All you need to do is provide the YOLO v4 weights and configuration files. **This tutorial (https://dsbyprateekg.blogspot.com/2020/08/how-to-use-opencv-python-with-darknets.html)** will help you get started using YOLO v4 with OpenCV.

What's next? We recommend PyImageSearch University

[Click here to download the source code to this post](https://pyimagesearch.com/pyimagesearch-university/?utm_source=blogPost&utm_medium=bottomBanner&utm_campaign=What%27s%20next%3F%20I%20recommend).



Course information:

86+ total classes • 115+ hours hours of on-demand code walkthrough videos • Last updated: July 2025

★★★★★ 4.84 (128 Ratings) • 16,000+ Students Enrolled

I strongly believe that if you had the right teacher you could *master* computer vision and deep learning.

Do you think learning computer vision and deep learning has to be time-consuming, overwhelming, and complicated? Or has to involve complex mathematics and equations? Or requires a degree in computer science?

Click here to download the source code to this post

That's *not* the case.

All you need to master computer vision and deep learning is for someone to explain things to you in *simple, intuitive* terms. *And that's exactly what I do*. My mission is to change education and how complex Artificial Intelligence topics are taught.

If you're serious about learning computer vision, your next stop should be PyImageSearch University, the most comprehensive computer vision, deep learning, and OpenCV course online today. Here you'll learn how to *successfully* and *confidently* apply computer vision to your work, research, and projects. Join me in computer vision mastery.

Inside PyImageSearch University you'll find:

✓ **86+ courses** on essential computer vision, deep learning, and OpenCV topics

✓ **86 Certificates** of Completion

✓ **115+ hours hours** of on-demand video

✓ **Brand new courses released *regularly***, ensuring you can keep up with state-of-the-art techniques

✓ **Pre-configured Jupyter Notebooks in Google Colab**

✓ Run all code examples in your web browser — works on Windows, macOS, and Linux (no dev environment configuration required!)

Summary

In this tutorial we learned how to perform YOLO object detection using Deep Learning, OpenCV, and Python.

We then briefly discussed the YOLO architecture followed by implementing Python code to:

1. Apply YOLO object detection to single images
2. Apply the YOLO object detector to video streams

On my machine with a 3GHz Intel Xeon W processor, a single forward pass of YOLO took ≈0.3 seconds; however, **using a Single Shot Detector (SSD) from a previous tutorial (https://pyimagesearch.com/2017/09/11/object-detection-with-deep-learning-and-opencv/)**, resulted in only 0.03 second detection, *an order of magnitude faster!*

For real-time deep learning-based object detection on your CPU with OpenCV and Python, you may want to consider using the SSD.

Click here to download the source code to this post

If you are interested in training your own deep learning object detectors on your own custom datasets, be sure to refer to my book, ***Deep Learning for Computer Vision with Python*** **(https://pyimagesearch.com/deep-learning-computer-vision-python-book/)**, where I provide detailed guides on how to successfully train your own detectors.

I hope you enjoyed today's YOLO object detection tutorial!

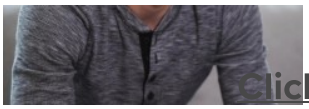
To download the source code to today's post, and be notified when future PyImageSearch blog posts are published, *just enter your email address in the form below*.

Download the Source Code and FREE 17-page Resource Guide

Enter your email address below to get a .zip of the code and a **FREE 17-page Resource Guide on Computer Vision, OpenCV, and Deep Learning.** Inside you'll find my hand-picked tutorials, books, courses, and libraries to help you master CV and DL!

About the Author

Hi there, I'm Adrian Rosebrock, PhD. All too often I see developers, students, and researchers wasting their time,



studying the wrong things, and generally struggling to get started with Computer Vision, Deep Learning, and OpenCV. I created this website to show you what I believe is the best possible way to get your start.

Click here to download the source code to this post

Similar articles

KICKSTARTER

It's time. The Deep Learning for Computer Vision with Python Kickstarter is LIVE.

January 18, 2017

(https://pyimagesearch.com/2017/01/18/its-time-the-deep-learning-for-computer-vision-with-python-kickstarter-is-live/)

DEEP LEARNING KERAS AND TENSORFLOW TUTORIALS

Smile detection with OpenCV, Keras, and TensorFlow

July 14, 2021

(https://pyimagesearch.com/2021/07/14/smile-detection-with-opencv-keras-and-tensorflow/)

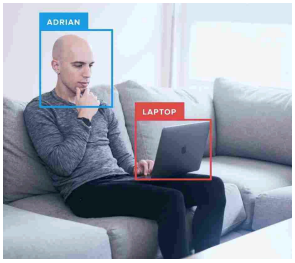
FACE APPLICATIONS TUTORIALS

Click here to download the source code to this post

How to build a custom face recognition dataset

June 11, 2018

(https://pyimagesearch.com/2018/06/11/how-to-build-a-custom-face-recognition-dataset/)



You can learn Computer Vision, Deep Learning, and OpenCV.

Get your FREE 17 page Computer Vision, OpenCV, and Deep Learning Resource Guide PDF. Inside you'll find our hand-picked tutorials, books, courses, and libraries to help you master CV and DL.

Topics

Deep Learning (https://pyimagesearch.com/category/deep-

Machine Learning and Computer Vision (https://pyimagesearch.com/category/machine-learning/)

learning/)

Dlib Library (https://pyimagesearch.com/category/dlib/)

Embedded/IoT and Computer Vision (https://pyimagesearch.com/category/embedded/)

Face Applications (https://pyimagesearch.com/category/faces/)

Image Processing (https://pyimagesearch.com/category/image-processing/)

Interviews (https://pyimagesearch.com/category/interviews/)

Keras & Tensorflow (https://pyimagesearch.com/category/keras-and-tensorflow/)

OpenCV Install Guides (https://pyimagesearch.com/opencv-tutorials-resources-guides/)

Click here to download the source code to this post

Medical Computer Vision (https://pyimagesearch.com/category/medical/)

Optical Character Recognition (OCR) (https://pyimagesearch.com/category/optical-character-recognition-ocr/)

Object Detection (https://pyimagesearch.com/category/object-detection/)

Object Tracking (https://pyimagesearch.com/category/object-tracking/)

OpenCV Tutorials (https://pyimagesearch.com/category/opencv/)

Raspberry Pi (https://pyimagesearch.com/category/raspberry-pi/)

Books & Courses

PyImageSearch University (https://pyimagesearch.com/pyimagesearch-university/)

FREE CV, DL, and OpenCV Crash Course (https://pyimagesearch.com/free-opencv-computer-vision-deep-learning-crash-course/)

Practical Python and OpenCV (https://pyimagesearch.com/practical-python-opencv/)

PyImageSearch

Affiliates (https://pyimagesearch.com/affiliates/)

Get Started (https://pyimagesearch.com/start-here/)

About (https://pyimagesearch.com/about/)

Consulting (https://pyimagesearch.com/consulting-2/)

Coaching (https://pyimagesearch.com/consult-adrian/)

Deep Learning for Computer Vision with Python (https://pyimagesearch.com/deep-learning-computer-vision-python-book/)

PyImageSearch Gurus Course (https://pyimagesearch.com/pyimagesearch-gurus/)

Raspberry Pi for Computer Vision (https://pyimagesearch.com/raspberry-pi-for-computer-vision/)

Click here to download the source code to this post

FAQ (https://pyimagesearch.com/faqs/)

YouTube (https://pyimagesearch.com/youtube/)

Blog (https://pyimagesearch.com/topics/)

Contact (https://pyimagesearch.com/contact/)

Privacy Policy (https://pyimagesearch.com/privacy-policy/)

(https://www.facebook.com/pyimagesearch) (https://twitter.com/PyImageSearch) (https://www.linkedin.com/company/pyimagesearch/) (https://www.youtube.com/channel/UCoQK7OVcIVy-nV4m-SMCk_Q/videos)