# Object detection with deep learning and OpenCV

In the first part of today’s post on object detection using deep learning we’ll discuss Single Shot Detectors and MobileNets.

When combined together these methods can be used for super fast, real-time object detection on resource constrained devices (including the Raspberry Pi, smartphones, etc.)

From there we’ll discover how to use OpenCV’s dnn  module to load a pre-trained object detection network.

This will enable us to pass input images through the network and obtain the output bounding box (x, y)-coordinates of each object in the image.

Finally we’ll look at the results of applying the MobileNet Single Shot Detector to example input images.

In a future blog post we’ll extend our script to work with real-time video streams as well

## Single Shot Detectors for object detection

**Figure 1:** Examples of object detection using Single Shot Detectors (SSD) from Liu et al.

When it comes to deep learning-based object detection there are three primary object detection methods that you’ll likely encounter:

* [Faster R-CNNs](https://arxiv.org/abs/1506.01497) (Girshick et al., 2015)
* [You Only Look Once (YOLO)](https://arxiv.org/abs/1506.02640) (Redmon and Farhadi, 2015)
* [Single Shot Detectors (SSDs)](https://arxiv.org/abs/1512.02325) (Liu et al., 2015)

Faster R-CNNs are likely the most “heard of” method for object detection using deep learning; however, the technique can be difficult to understand (especially for beginners in deep learning), hard to implement, and challenging to train.

Furthermore, even with the “faster” implementation R-CNNs (where the “R” stands for “Region Proposal”) the algorithm can be quite slow, on the order of 7 FPS.

If we are looking for pure speed then we tend to use YOLO as this algorithm is much faster, capable of processing 40-90 FPS on a Titan X GPU. The super fast variant of YOLO can even get up to 155 FPS.

The problem with YOLO is that it leaves much accuracy to be desired.

SSDs, originally developed by Google, are a balance between the two. The algorithm is more straightforward (and I would argue better explained in the original seminal paper) than Faster R-CNNs.

We can also enjoy a much faster FPS throughput than [Girshick et al](https://arxiv.org/abs/1506.01497). at 22-46 FPS depending on which variant of the network we use. SSDs also tend to be more accurate than YOLO. To learn more about SSDs, please refer to [Liu et al](https://arxiv.org/abs/1512.02325).

## MobileNets: Efficient (deep) neural networks

**Figure 2:** (Left) Standard convolutional layer with batch normalization and ReLU. (Right) Depthwise separable convolution with depthwise and pointwise layers followed by batch normalization and ReLU (figure and caption from Liu et al.).

When building object detection networks we normally use an existing network architecture, such as VGG or ResNet, and then use it inside the object detection pipeline. The problem is that these network architectures can be very large in the order of 200-500MB.

Network architectures such as these are unsuitable for resource constrained devices due to their sheer size and resulting number of computations.

Instead, we can use [MobileNets](https://arxiv.org/abs/1704.04861) (Howard et al., 2017), another paper by Google researchers. We call these networks “MobileNets” because they are designed for resource constrained devices such as your smartphone. MobileNets differ from traditional CNNs through the usage of depthwise separable convolution (**Figure 2** above).

The general idea behind depthwise separable convolution is to split convolution into two stages:

1. A 3×3 depthwise convolution.
2. Followed by a 1×1 pointwise convolution.

This allows us to actually reduce the number of parameters in our network.

The problem is that we sacrifice accuracy — MobileNets are normally not as accurate as their larger big brothers…

…but they are much more resource efficient.

For more details on MobileNets please see [Howard et al](https://arxiv.org/abs/1704.04861).

## Combining MobileNets and Single Shot Detectors for fast, efficient deep-learning based object detection

If we combine both the MobileNet architecture and the Single Shot Detector (SSD) framework, we arrive at a fast, efficient deep learning-based method to object detection.

The model we’ll be using in this blog post is a Caffe version of the [original TensorFlow implementation](https://github.com/Zehaos/MobileNet) by Howard et al. and was trained by chuanqi305 ([see GitHub](https://github.com/chuanqi305/MobileNet-SSD)).

The MobileNet SSD was first trained on the [COCO dataset](http://cocodataset.org/) (Common Objects in Context) and was then fine-tuned on PASCAL VOC reaching 72.7% mAP (mean average precision).

We can therefore detect 20 objects in images (+1 for the background class), including airplanes, bicycles, birds, boats, bottles, buses, cars, cats, chairs, cows, dining tables, dogs, horses, motorbikes, people, potted plants, sheep, sofas, trains, and tv monitors.

## Deep learning-based object detection with OpenCV

In this section we will use the MobileNet SSD + deep neural network ( dnn ) module in OpenCV to build our object detector.

I would suggest using the ***“Downloads”*** code at the bottom of this blog post to download the source code + trained network + example images so you can test them on your machine.

Let’s go ahead and get started building our deep learning object detector using OpenCV.

Open up a new file, name it deep\_learning\_object\_detection.py , and insert the following code:

Object detection with deep learning and OpenCV Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | # import the necessary packages  import numpy as np  import argparse  import cv2    # 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("-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()) |

On **Lines 2-4** we import packages required for this script — the dnn module is included in cv2 , again, making hte assumption that you’re using OpenCV 3.3.

Then, we parse our command line arguments (**Lines 7-16**):

* --image : The path to the input image.
* --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%.

Again, example files for the first three arguments are included in the ***“Downloads”*** section of this blog post. I urge you to start there while also supplying some query images of your own.

Next, let’s initialize class labels and bounding box colors:

Object detection with deep learning and OpenCV Python

|  |  |
| --- | --- |
| 18  19  20  21  22  23  24 | # 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)) |

**Lines 20-23** build a list called CLASSES containing our labels. This is followed by a list, COLORS which contains corresponding random colors for bounding boxes (**Line 24**).

Now we need to load our model:

Object detection with deep learning and OpenCV Python

|  |  |
| --- | --- |
| 26  27  28 | # load our serialized model from disk  print("[INFO] loading model...")  net = cv2.dnn.readNetFromCaffe(args["prototxt"], args["model"]) |

The above lines are self-explanatory, we simply print a message and load our model  (**Lines 27 and 28**).

Next, we will load our query image and prepare our blob , which we will feed-forward through the network:

Object detection with deep learning and OpenCV Python

|  |  |
| --- | --- |
| 30  31  32  33  34  35  36  37 | # load the input image and construct an input blob for the image  # by resizing to a fixed 300x300 pixels and then normalizing it  # (note: normalization is done via the authors of the MobileNet SSD  # implementation)  image = cv2.imread(args["image"])  (h, w) = image.shape[:2]  blob = cv2.dnn.blobFromImage(cv2.resize(image, (300, 300)), 0.007843,  (300, 300), 127.5) |

Taking note of the comment in this block, we load our image  (**Line 34**), extract the height and width (**Line 35**), and calculate a 300 by 300 pixel blob  from our image (**Line 36**).

Now we’re ready to do the heavy lifting — we’ll pass this blob through the neural network:

Object detection with deep learning and OpenCV Python

|  |  |
| --- | --- |
| 38  39  40  41  42 | # pass the blob through the network and obtain the detections and  # predictions  print("[INFO] computing object detections...")  net.setInput(blob)  detections = net.forward() |

On **Lines 41 and 42** we set the input to the network and compute the forward pass for the input, storing the result as detections . Computing the forward pass and associated detections could take awhile depending on your model and input size, but for this example it will be relatively quick on most CPUs.

Let’s loop through our detections  and determine what and where the objects are in the image:

Object detection with deep learning and OpenCV Python

|  |  |
| --- | --- |
| 44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67 | # loop over the detections  for i in np.arange(0, detections.shape[2]):  # extract the confidence (i.e., probability) associated with the  # prediction  confidence = detections[0, 0, i, 2]    # filter out weak detections by ensuring the `confidence` is  # 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")    # display the prediction  label = "{}: {:.2f}%".format(CLASSES[idx], confidence \* 100)  print("[INFO] {}".format(label))  cv2.rectangle(image, (startX, startY), (endX, endY),  COLORS[idx], 2)  y = startY - 15 if startY - 15 > 15 else startY + 15  cv2.putText(image, 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 48**).

If the confidence  is above our minimum threshold (**Line 52**), we extract the class label index (**Line 56**) and compute the bounding box around the detected object (**Line 57**).

Then, we extract the (x, y)-coordinates of the box (**Line 58**) which we will will use shortly for drawing a rectangle and displaying text.

Next, we build a text label  containing the CLASS  name and the confidence  (**Line 61**).

Using the label, we print it to the terminal (**Line 62**), followed by drawing a colored rectangle around the object using our previously extracted (x, y)-coordinates (**Lines 63 and 64**).

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 65**).

Finally, we overlay the colored text onto the image  using the y-value that we just calculated (**Lines 66 and 67**).

The only remaining step is to display the result:

Object detection with deep learning and OpenCV Python

|  |  |
| --- | --- |
| 69  70  71 | # show the output image  cv2.imshow("Output", image)  cv2.waitKey(0) |

We display the resulting output image to the screen until a key is pressed (**Lines 70 and 71**).

## OpenCV and deep learning object detection results

To download the code + pre-trained network + example images, be sure to use the ***“Downloads”*** section at the bottom of this blog post.

From there, unzip the archive and execute the following command:

Object detection with deep learning and OpenCV Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $ python deep\_learning\_object\_detection.py \  --prototxt MobileNetSSD\_deploy.prototxt.txt \  --model MobileNetSSD\_deploy.caffemodel --image images/example\_01.jpg  [INFO] loading model...  [INFO] computing object detections...  [INFO] loading model...  [INFO] computing object detections...  [INFO] car: 99.78%  [INFO] car: 99.25% |

**Figure 3:** Two Toyotas on the highway recognized with near-100% confidence using OpenCV, deep learning, and object detection.

Our first result shows cars recognized and detected with near-100% confidence.

In this example we detect an airplane using deep learning-based object detection:

Object detection with deep learning and OpenCV Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $ python deep\_learning\_object\_detection.py \  --prototxt MobileNetSSD\_deploy.prototxt.txt \  --model MobileNetSSD\_deploy.caffemodel --image images/example\_02.jpg  [INFO] loading model...  [INFO] computing object detections...  [INFO] loading model...  [INFO] computing object detections...  [INFO] aeroplane: 98.42% |

**Figure 4:** An airplane successfully detected with high confidence via Python, OpenCV, and deep learning.

The ability for deep learning to detect and localize obscured objects is demonstrated in the following image, where we see a horse (and it’s rider) jumping a fence flanked by two potted plants:

Object detection with deep learning and OpenCV Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $ python deep\_learning\_object\_detection.py \  --prototxt MobileNetSSD\_deploy.prototxt.txt \  --model MobileNetSSD\_deploy.caffemodel --image images/example\_03.jpg  [INFO] loading model...  [INFO] computing object detections...  [INFO] horse: 96.67%  [INFO] person: 92.58%  [INFO] pottedplant: 96.87%  [INFO] pottedplant: 34.42% |

**Figure 5:** A person riding a horse and two potted plants are successfully identified despite a lot of objects in the image via deep learning-based object detection.

In this example we can see a beer bottle is detected with an impressive 100% confidence:

Object detection with deep learning and OpenCV Python

|  |  |
| --- | --- |
| 1  2  3  4  5 | $ python deep\_learning\_object\_detection.py --prototxt MobileNetSSD\_deploy.prototxt.txt \  --model MobileNetSSD\_deploy.caffemodel --image images/example\_04.jpg  [INFO] loading model...  [INFO] computing object detections...  [INFO] bottle: 100.00% |

**Figure 6:** Deep learning + OpenCV are able to correctly detect a beer bottle in an input image.

Followed by another horse image which also contains a dog, car, and person:

Object detection with deep learning and OpenCV Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $ python deep\_learning\_object\_detection.py \  --prototxt MobileNetSSD\_deploy.prototxt.txt \  --model MobileNetSSD\_deploy.caffemodel --image images/example\_05.jpg  [INFO] loading model...  [INFO] computing object detections...  [INFO] car: 99.87%  [INFO] dog: 94.88%  [INFO] horse: 99.97%  [INFO] person: 99.88% |

**Figure 7:** Several objects in this image including a car, dog, horse, and person are all recognized.

Finally, a picture of me and Jemma, the family beagle:

Object detection with deep learning and OpenCV Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $ python deep\_learning\_object\_detection.py \  --prototxt MobileNetSSD\_deploy.prototxt.txt \  --model MobileNetSSD\_deploy.caffemodel --image images/example\_06.jpg  [INFO] loading model...  [INFO] computing object detections...  [INFO] dog: 95.88%  [INFO] person: 99.95% |

**Figure 8:** Me and the family beagle are corrected as a “person” and a “dog” via deep learning, object detection, and OpenCV. The TV monitor is not recognized.

Unfortunately the TV monitor isn’t recognized in this image which is likely due to (1) me blocking it and (2) poor contrast around the TV. That being said, we have demonstrated excellent object detection results using OpenCV’s dnn  module.

# Summary

In today’s blog post we learned how to perform object detection using deep learning and OpenCV.

Specifically, we used both MobileNets + Single Shot Detectors along with OpenCV 3.3’s brand new (totally overhauled) dnn  module to detect objects in images.

As a computer vision and deep learning community we owe a lot to the contributions of Aleksandr Rybnikov, the main contributor to the dnn  module for making deep learning so accessible from within the OpenCV library. You can find Aleksandr’s original OpenCV example script [here](https://github.com/opencv/opencv/blob/master/samples/dnn/mobilenet_ssd_python.py) — I have modified it for the purposes of this blog post.

In a future blog post I’ll be demonstrating how we can modify today’s tutorial to work with real-time video streams, thus enabling us to perform deep learning-based object detection to videos. We’ll be sure to leverage efficient frame I/O to increase the FPS throughout our pipeline as well.

Bottom of Form

×

### Almost there...

To start your free 17-day computer vision and deep learning crash course, confirm your email address by clicking the link in the email I just sent you.