Custom object training and detection with YOLOv3, Darknet and OpenCV

Vino Mahendran Follow
Nov 15, 2019 · 5 min read

Photo by Jessica Ruscello on Unsplash

> *YOLO is a state-of-the-art, real-time object detection system. It looks at the whole image at test time so its predictions are informed by global context in the image. It also makes predictions with a single network evaluation which makes it extremely fast when compared to R-CNN and Fast R-CNN.*

This paper gives more details about how YOLO achieves the performance improvement.

Darknet

We will use Darknet, an open source neural network framework to train the detector. Download and build darknet

```
git clone https://github.com/pjreddie/darknet
cd darknet
make
```

Once that's successful, To test the build we can download pre trained YOLO weights and perform detection with the test image.

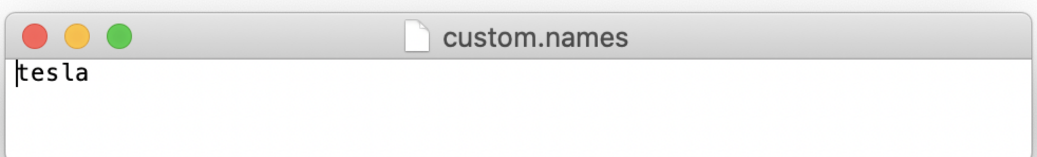
```
./darknet detector test cfg/coco.data cfg/yolov3.cfg
yolov3.weights data/dog.jpg
```

For training with custom objects, let us create the following required files and directories

```
custom_data/custom.names
custom_data/images
custom_data/train.txt
custom_data/test.txt
```

In this example we will train to detect Tesla cars for which we should collect the images and place it in `custom_data/images` directory.

Labels of our objects should be saved in `custom_data/custom.names` file, each line in the file corresponds to an object. In our case since we have only one object class, the file should contain the following



Annotation

After we collect the images containing our custom object, we will need to annotate them. For YOLOv3, each image should have a corresponding text file with the same file name as that of the image in the same directory.

In our case text files should be saved in `custom_data/images` directory. For e.g. `image1.jpg` should have a text file `image1.txt`.

Each row in the text file corresponds to a single bounding box of the object and should have the following information

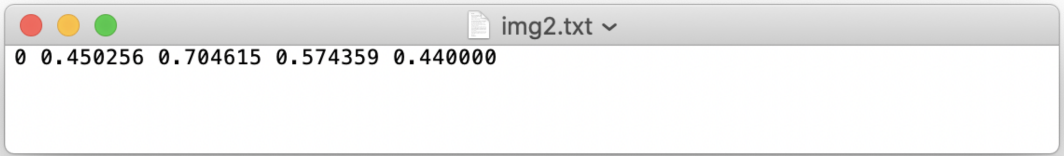
```
<object-class-id> <x-centre> <y-centre> <width> <height>
```

where

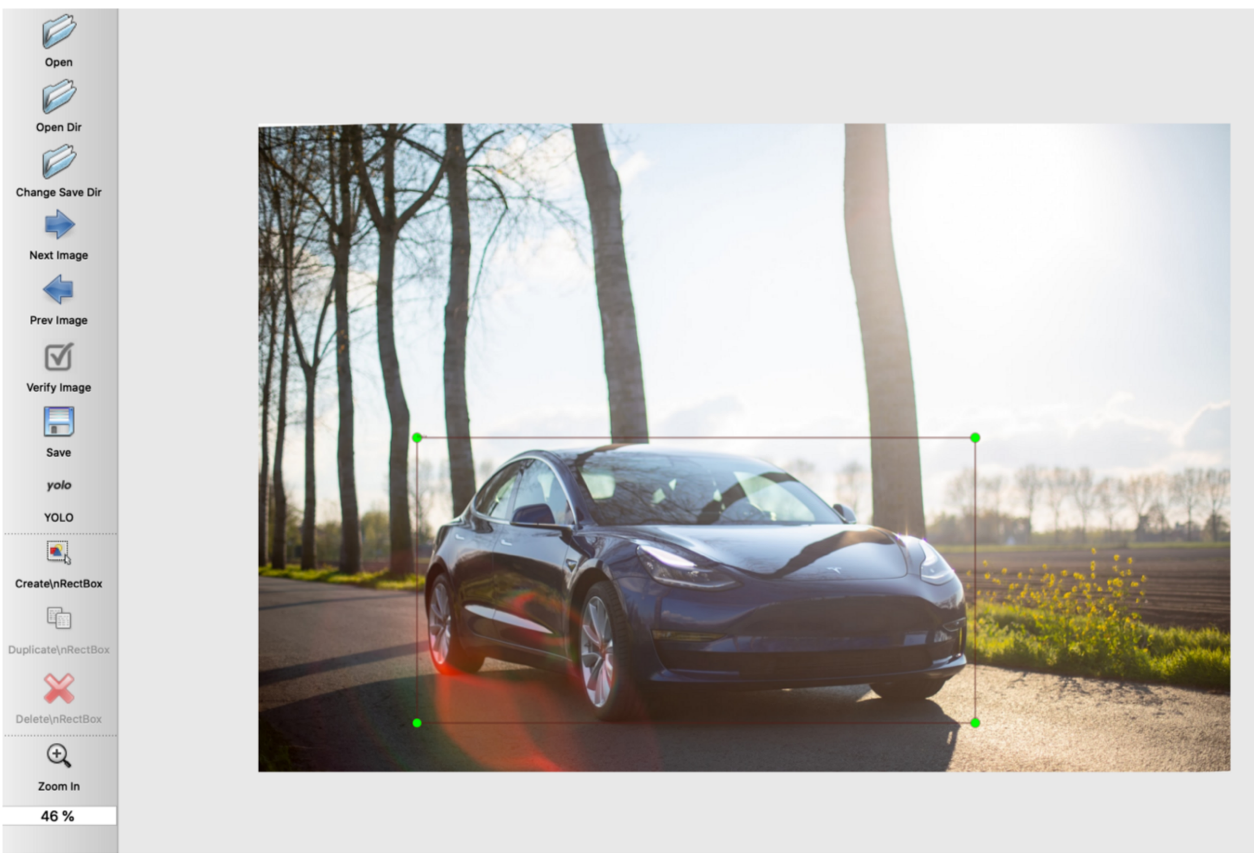
```
<object-class-id> an integer from 0 to (classes - 1)
corresponding to the classes in the
custom_data/custom.names file

height, width - Actual height and width of the image
x, y - centre coordinates of the bounding box
h, w - height and width of the bounding box

<x-centre> : x / width
<y-centre> : y / height
<width> : w / width
<height> : h / height
```



This can also be done with LabelImg, a graphical image annotation tool which creates `.txt` files for the images in YOLO format.



Train and Test sets

We can then randomly split the annotated images into train and test sets in the ratio of 80:20

`custom_data/train.txt` Each row in the file should have the location of train dataset.

`custom_data/test.txt` Each row in the file should have the location of test dataset.

train.txt

```
custom_data/images/image230.jpg
custom_data/images/image279.jpg
custom_data/images/image30.jpg
custom_data/images/image214.jpg
custom_data/images/image216.jpg
custom_data/images/image3.jpg
custom_data/images/image273.jpg
custom_data/images/image293.jpg
custom_data/images/image369.jpg
custom_data/images/image327.jpg
custom_data/images/image348.jpg
custom_data/images/image28.jpg
custom_data/images/image257.jpg
custom_data/images/image237.jpg
custom_data/images/image160.jpg
```

Pre-trained weights

To train our object detector we can use the existing pre trained weights that are already trained on huge data sets. From here we can download the pre trained weights to the root directory.

YOLO data file

Create a `detector.data` file in the `custom_data` directory which should contain information regarding the train and test data sets

```
classes=1
train=custom_data/train.txt
valid=custom_data/test.txt
names=custom_data/custom.names
backup=backup/
```

backup is the location where newly trained weights would be saved.

Configurations

Based on the required performance we can select the YOLOv3 configuration file. For this example we will be using `yolov3.cfg`. We can duplicate the file from `cfg/yolov3.cfg` to `custom_data/cfg/yolov3-custom.cfg`

While training the images, weights of the neural networks are updated iteratively. We may use huge training sets which makes it resource consuming to update the weights for the entire training set in a single iteration. To use a small set of images to iteratively update the weights, `batch` param is set. By default it is set to 64.

The maximum number of iterations for which our network should be trained is set with the param `max_batches=4000`. Also update `steps=3200,3600` which is 80%, 90% of `max_batches`.

We will need to update the `classes` and `filters` params of `[yolo]` and `[convolutional]` layers that are just before the `[yolo]` layers.

In this example since we have a single class (tesla) we will update the `classes` param in the `[yolo]` layers to 1 at `line numbers: 610, 696, 783`

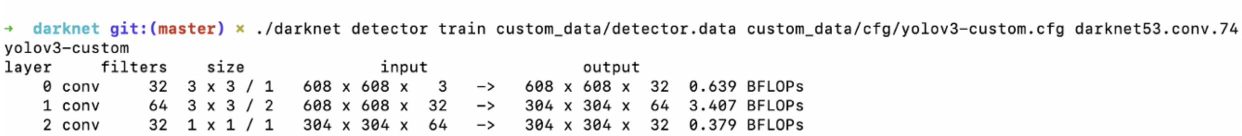
Similarly we will need to update the `filters` param based on the classes count `filters=(classes + 5) * 3`. For a single class we should set `filters=18` at `line numbers: 603, 689, 776`

All the configuration changes are made to `custom_data/cfg/yolov3-custom.cfg`

Training

With all the required files and annotated images we can start our training

```
./darknet detector train custom_data/detector.data
custom_data/cfg/yolov3-custom.cfg darknet53.conv.74
```



```
➜  darknet git:(master) ✗ ./darknet detector train custom_data/detector.data custom_data/cfg/yolov3-custom.cfg darknet53.conv.74
yolov3-custom
layer     filters    size              input                output
    0 conv     32  3 x 3 / 1   608 x 608 x   3   ->   608 x 608 x  32  0.639 BFLOPs
    1 conv     64  3 x 3 / 2   608 x 608 x  32   ->   304 x 304 x  64  3.407 BFLOPs
    2 conv     32  1 x 1 / 1   304 x 304 x  64   ->   304 x 304 x  32  0.379 BFLOPs
    3 conv     64  3 x 3 / 1   304 x 304 x  32   ->   304 x 304 x  64  3.407 BFLOPs
```

```
  4 res    1                 304 x 304 x  64   ->   304 x 304 x  64
  5 conv    128  3 x 3 / 2   304 x 304 x  64   ->   152 x 152 x 128  3.407 BFLOPs
  6 conv     64  1 x 1 / 1   152 x 152 x 128   ->   152 x 152 x  64  0.379 BFLOPs
  7 conv    128  3 x 3 / 1   152 x 152 x  64   ->   152 x 152 x 128  3.407 BFLOPs
  8 res    5                 152 x 152 x 128   ->   152 x 152 x 128
  9 conv     64  1 x 1 / 1   152 x 152 x 128   ->   152 x 152 x  64  0.379 BFLOPs
 10 conv    128  3 x 3 / 1   152 x 152 x  64   ->   152 x 152 x 128  3.407 BFLOPs
 11 res    8                 152 x 152 x 128   ->   152 x 152 x 128
 12 conv    256  3 x 3 / 2   152 x 152 x 128   ->    76 x  76 x 256  3.407 BFLOPs
 13 conv    128  1 x 1 / 1    76 x  76 x 256   ->    76 x  76 x 128  0.379 BFLOPs
 14 conv    256  3 x 3 / 1    76 x  76 x 128   ->    76 x  76 x 256  3.407 BFLOPs
 15 res   12                  76 x  76 x 256   ->    76 x  76 x 256
 16 conv    128  1 x 1 / 1    76 x  76 x 256   ->    76 x  76 x 128  0.379 BFLOPs
 17 conv    256  3 x 3 / 1    76 x  76 x 128   ->    76 x  76 x 256  3.407 BFLOPs
 18 res   15                  76 x  76 x 256   ->    76 x  76 x 256
 19 conv    128  1 x 1 / 1    76 x  76 x 256   ->    76 x  76 x 128  0.379 BFLOPs
 20 conv    256  3 x 3 / 1    76 x  76 x 128   ->    76 x  76 x 256  3.407 BFLOPs
 21 res   18                  76 x  76 x 256   ->    76 x  76 x 256
```

We can continue training until the loss reaches a certain threshold. By default, weights for the custom detector is saved for every 100 iterations until 1000 iterations and then continues to save for every 10000 iterations. This behaviour can be modified by updating the condition at line 138 of `examples/detector.c` file.

Once the training is complete we can use the generated weights to perform detection.

Detection with OpenCV

We can perform detection with OpenCV DNN as it is a fast DNN implementation for CPU.

```
import argparse

import cv2
import numpy as np

parser = argparse.ArgumentParser(add_help=False)
parser.add_argument("--image", default='samples/image.jpg', help="image for
parser.add_argument("--config", default='cfg/yolov3.cfg', help="YOLO config
parser.add_argument("--weights", default='yolov3.weights', help="YOLO weight
```

```
parser.add_argument("--names", default='data/coco.names', help="class names
args = parser.parse_args()

CONF_THRESH, NMS_THRESH = 0.5, 0.5

# Load the network
net = cv2.dnn.readNetFromDarknet(args.config, args.weights)
net.setPreferableBackend(cv2.dnn.DNN_BACKEND_OPENCV)
net.setPreferableTarget(cv2.dnn.DNN_TARGET_CPU)

# Get the output layer from YOLO
layers = net.getLayerNames()
output_layers = [layers[i[0] - 1] for i in net.getUnconnectedOutLayers()]

# Read and convert the image to blob and perform forward pass to get the bou
img = cv2.imread(args.image)
height, width = img.shape[:2]

blob = cv2.dnn.blobFromImage(img, 0.00392, (416, 416), swapRB=True, crop=Fal
net.setInput(blob)
layer_outputs = net.forward(output_layers)

class_ids, confidences, b_boxes = [], [], []
for output in layer_outputs:
    for detection in output:
        scores = detection[5:]
        class_id = np.argmax(scores)
        confidence = scores[class_id]

        if confidence > CONF_THRESH:
            center_x, center_y, w, h = (detection[0:4] * np.array([width, he

            x = int(center_x - w / 2)
            y = int(center_y - h / 2)

            b_boxes.append([x, y, int(w), int(h)])
            confidences.append(float(confidence))
            class_ids.append(int(class_id))

```

```
# Perform non maximum suppression for the bounding boxes to filter overlappi
indices = cv2.dnn.NMSBoxes(b_boxes, confidences, CONF_THRESH, NMS_THRESH).fl

# Draw the filtered bounding boxes with their class to the image
with open(args.names, "r") as f:
    classes = [line.strip() for line in f.readlines()]
colors = np.random.uniform(0, 255, size=(len(classes), 3))

for index in indices:
```

We can specify `--image`, `--config`, `--weights` and `--names` params as per our training to perform predictions for our custom object.

. . .

Francium Tech is a technology company laser focused on delivering top quality software of scale at extreme speeds. Numbers and Size of the data don't scare us. If you have any requirements or want a free health check of your systems or architecture, feel free to shoot an email to contact@francium.tech, we will get in touch with you!

Yolov3 Darknet Opencv Deep Learning Object Detection