A brief guide on how to set up and use YOLO V3

Yungyu Song

July 2019

1 Introduction

This is a documentation describing the different steps and processes that are required for a proper manipulation of YOLO V3 on CentOS7 Linux environment. YOLO V3 is an object recognition engine which was developed in order to allow a real-time object detection, through the use of a special CNN structure. For more information on YOLO V3 engine itself, please refer to the links provided in the reference section.

This was written in order to help any beginners in this domain and act as an illuminating manual. If the following instructions are not sufficient, or if the reader comes across any errors that are not introduced in this document, it would be advisable to search further using Google. To those who are reading this, I sincerely hope that this would offer you insight on how to approach the deep world of object recognition. Without further ado, *Sapere Aude!*

2 Installation of necessary elements

2.1 Installation of OpenCV

First and foremost, it is essential to begin the approach to YOLO V3 by installing OpenCV, a library designed for computer vision. This will be essential later on when using YOLO V3 to perform object recognition within a video file. For convenience and the stability of OpenCV, it is advisable to download OpenCV version 3.4.0. The code allowing the software's download is provided in the appendix, along with essential descriptions for the code lines.

This is perhaps the most complicated part of the document, as the make process of OpenCV is prone to produce insufferable amount of errors. Due to the variety of complications that can be produced in this stage, this guide is unfortunately unable to provide a solution for all of them. However, the problems can be more or less avoided by keeping in mind the following instructions.

1. Make sure that the correct OpenCV version is declared during the installation process, by using git checkout command. This line will set the correct version (3.4.0) and is absolutely essential for a successful installation.

2. Choose as little options as possible for cmake. By doing so, it is possible to reduce the chances of error. It is highly likely that only the minimum settings will be required for a proper operation of OpenCV.

3. Next to the make command it is possible to note the -j*any_integer*. The integer following the -j is the number of CPU that the PC possesses, hence it is imperative for the user to choose an integer value that is lower than the number of CPU provided by the PC.

2.2 Installation of DarkNet

Once the installation of OpenCV is complete, it would be advisable for the reader to now install the most important element of this document, DarkNet; the software corresponds to the open-source neural network in C, which contains the YOLO V3 engine and other object recognition engines. The installation process of DarkNet is far from being complicated, and the user will simply need to follow the instructions given in the link: https://pjreddie.com/darknet/install/. It is absolutely necessary that the user compiles DarkNet with both CUDA and OpenCV, as this will allow the PC to use its GPU when processing heavy files such as videos or high-resolution images. To do so, simply edit lines 1 and 3 of Makefile in /darknet directory, changing from GPU=0 to GPU=1 and OPENCV=0 to OPENCV=1. Note that this instruction assumes that the user's PC already has CUDA installed.

Once the Makefile is edited and make command is applied on Linux terminal, the user may notice that darknet.exe is generated in /darknet directory. This indeed corresponds to the main body of DarkNet, used for both training datasets as well as producing outputs after going through object recognition. Once this .exe file is generated, it would be advisable for the user to go through the tutorials (provided in the url given above) to understand how DarkNet works.

2.2.1 Potential error

When the user tries to run darknet for the first time, it is possible that the terminal returns an error stating OpenCV was not found or a library from opencv was not found. This is due to the wrong settings of PKG_CONFIG_PATH and LD_LIBRARY_PATH, usually both being empty at first. To change this, entre the following commands in the Linux terminal.

```
$ export PKG_CONFIG_PATH=/usr/local/lib64/pkgonfig/:/usr/local/cuda-8.0/pkgconfig/
$ export LD_LIBRARY_PATH=/usr/lib64/:/usr/local/cuda-8.0/lib64/:/usr/local/lib64/
```

2.3 Installation of Anaconda & virtual environment setup

Unlike the two previous installation processes, this subsection is not exactly mandatory. However, having a python virtual environment from Anaconda certainly allows the user to manage data (video results and so on) from DarkNet with far more ease.

For this subsection, it is possible to begin by installing Miniconda, a lightweight version of Anaconda. The process is extremely simple, involving only the two following lines on a Linux terminal for the installation.

```
$ wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
$ sh Miniconda3-latest-Linux-x86_64.sh
```

This should properly install the latest version of Miniconda, which can be checked by the command conda -V on the Linux terminal. Once this is completed, create a virtual environment and activate it using the following lines.

```
$ conda create -n name_of_your_virtual_environment python=python_version_to_use -y
$ source activate name_of_your_virtual_environment
```

The virtual environment behaves similarly to the conda terminal, allowing the users to perform convenient installation of different modules using pip-install (this may cause proxy errors, in which case it is advisable to contact the network administrator), as well as running .py files later on. The modules that can be downloaded using pip-install that are necessary for YOLO V3 would be utils, matplotlib, tensorflow, numpy, argparse, imutils, pyTorch, time, cv2 (opencv-python), and finally os. Make sure that these modules are installed before running relevant .py files.

3 YOLO V3 on custom data

Once the installation processes and the tutorials for DarkNet are complete, it is possible to directly move onto the use of YOLO V3 with customized data sets. Put simply, the goal of this section would be to create a .weights file produced as output of machine learning training, and use it to then obtain an object recognition engine that befits the purpose of the user.

Firstly, prepare the dataset so that it fits to the data format required by YOLO V3. The dataset will be composed of two major elements, the images files (.jpg/.png) and the text files (.txt). The image files of course contains the image to be trained for the custom object detector, and the text files denote the elements YOLO V3 will detect that can be found in the images. Special care must be taken for the format of the text files' content, as the YOLO V3 specifically requires the content of the text files to be under the following structure: [category number] [object centre in X] [object centre in Y] [object width in X] [object width in Y].

The following task to complete is to then create a 'train.txt' file and 'test.txt' file, two .txt files that describes the dataset's location, allowing the image detector to access it. This can be executed by running a python script that may be called process.py, which is attached in the appendix.

Finally, the core structure of YOLO V3 must be modified, by creating 3 files to specify how and what to train. The 3 files that to be formed under the /darknet/cfg/ directory are: obj.data, obj.names, and yolo-obj.cfg. The file /darknet/cfg/obj.data specifies the trainig information such as the train.txt and test.txt file locations, number of classes, and the location where the .weights files will be saved. The content of the file should look like the following.

```
classes=no._of_classes_that_should_be_detected
train=location_of_train.txt_file
valid=location_of_test.txt_file
names=location_of_obj.names_file
backup=backup/ #This will automatically generate backup file in /darknet/ directory
and will be the location where the .weights and .backup file will be saved.
```

On the other hand, /darknet/cfg/obj.names file is extremely simple to create, simply enumerate every object category/class in order, keeping in mind that every new category should be written on a new line.

The last file to modify is the /darknet/cfg/yolo-obj.cfg. This is in fact the most complicated and important file as it describes the convolutional neural network (CNN) structure used for training. When creating this file, it is advisable to first copy and paste from /darknet/cfg/yolov3.cfg. The lines that must be edited are described below.

- Change the batch size to batch=64 (recommended value).
- Change the subdivisions number to subdivisions=8 or higher (lower values may overuse the GPU).
- Change the max_batches value to max_batches=2000*classes
- Change the line steps to 80% and 90% of the max_batches values
- Change the classes values that can be found in the [yolo] layers
- Change the filter values in the convolutional layer right before each [yolo] layers to be equal to filters=(classes+5)*3

For a more detailed explanation, please refer to the following github repository: https://github.com/AlexeyAB/darknet#how-to-train-to-detect-your-custom-objects.

Note that the file names and locations specified in this section can be freely modified to suit the user's convenience, but please remember to change the codes provided in the instruction appropriately.

4 Training & results

After following through all the provided instructions, the user should now be able to begin training for a custom dataset. To begin, first download a set of convolutional weights to start training.

```
$ wget https://pjreddie.com/media/files/darknet53.conv.74
```

Then the two following lines will begin the training process.

```
$ cd darknet
$ ./darknet detector train cfg/obj.data cfg/yolo-obj.cfg darknet53.conv.74
```

Throughout the course of the training, the .weights file will be created once every 100 iterations until 1000 iterations is reached, and then once every 10000 iterations. This can simply be changed by modifying line 138 of /darknet/examples/detector.c to produce .weights file at a rate the user desires. Once the .weights file are created, simply run the object detection engine on an image/video with reference to the newly created .weights file to view the output.

Once this step is over, it is possible for the user to further optimize the object detection engine by modifying its core structure. As mentioned previously, the yolo-obj.cfg file contains the core structure of the CNN; it is thus possible to modify various variables on this file (most notably the learning rate, the beta-decay value, batch size, and so on) to investigate how their changes affect the performance of the detector. Once changes are made on the .cfg file, please do not forget to run the two commands above in order to create new weight files.

5 Use of Python for YOLO V3

The problem with the use of demo function of darknet.exe is that the method to save the output video file once object detection is performed is implicit: python can thus be used to circumvent the problem. One can approach this solution by creating a file (home directory) with the following structure.

- videos folder, storing videos to go through the detection engine
- output folder, which will be used to store video outputs after object detection is performed
- yolo-obj folder, which will store the files required for object detection to run: obj.names, yolo-obj.cfg and the .weights file obtained previously
- yolo_video.py, allowing python to run yolo on a video file

The yolo_video.py file is provided in the appendix. With the correct files and codes, open the Linux terminal then activate the python virtual environment. Once opened, type the command in the terminal.

```
$ python yolo_video.py --input videos/name_of_video.mp4 --output output/name_of_output
.avi
```

The results of training can also be viewed, by using the instructions provided in the link: https://github.com/ultralytics/yolov3/wiki/Train-Custom-Data.

6 References

- OpenCV installation instruction (Japanese): https://qiita.com/usk81/items/98e54e2463e9d8a11415
- DarkNet installation instruction and tutorials (English): https://pjreddie.com/darknet/yolo/

7 Appendix

7.1 OpenCV installation commands

```
$ sudo su -

#installation of the required packages
$ yum install -y git gcc bzip2 bzip2-devel openssl openssl-devel readline readline
-devel sqlite-devel
$ yum install -y cmake libjpeg-devel libtiff-devel libpng-devel jasper-devel
$ yum install -y mesa-libGL-devel libXt-devel libgphoto2-devel nasm libtheora-devel
$ yum install -y autoconf automake gcc-c++ libtool yasm openal-devel blas blas-devel
atlas atlas-devel lapack lapack-devel
$ yum install -y tbb-devel

#download opencv and opencv_contrib
$ cd /usr/local/src
$ git clone https://github.com/opencv/opencv.git
$ git clone https://github.com/opencv/opencv_contrib.git
$ cd opencv_contrib
$ git checkout -b 3.4.0 refs/tags/3.4.0
$ cd ../opencv/
$ git checkout -b 3.4.0 refs/tags/3.4.0

#make process of OpenCV under directory build
$ mkdir build
$ cd build
$ cmake -D CMAKE_BUILD_TYPE=Release -D CMAKE_INSTALL_PREFIX=/usr/local
$ make -j8
$ make install
$ echo /usr/local/lib > /etc/ld.so.conf.d/opencv.conf
$ ldconfig -v
```

7.2 Process.py

```
import glob
import os

#current directory
current_dir=os.path.dirname(os.path.abspath('location_of_dataset_files')

#Directory where the data will reside, relative to 'darknet.exe'
path_data='location_of_data'

#Percentage of images used for the test set
percentage_test=10 #recommended value

#Create and/or truncate train.txt and test.txt
file_train=open('train.txt', 'w')
file_test=open('test.txt', 'w')

#Populate train.txt and test.txt
counter=1
```

```
index_test=round(100/percentage_test)
#detects only .jpg files
for pathAndFilename in glob.iglob(os.path.join(current_dir, "*.jpg")):
    title, ext=os.path.splitext(os.path.basename(pathAndFilename))

    if counter==index_test:
        counter=1
        file_test.write(path_data+title+'.jpg'+"\n")
    else:
        file_train.write(path_data+title+'.jpg'+"\n")
        counter = counter+1
```

7.3 Yolo_video.py

```
# import the necessary packages
import numpy as np
import argparse
import imutils
import time
import cv2
import os

# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--input", required=True,
    help="path to input video")
ap.add_argument("-o", "--output", required=True,
    help="path to output video")
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 applyong non-maxima suppression")
args = vars(ap.parse_args())

# load the custom data class labels the YOLO model was trained on
labelsPath = os.path.sep.join([args["yolo"], "obj.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"], ".weights_file"])
configPath = os.path.sep.join([args["yolo"], "yolov-obj.cfg"])

# load the YOLO object detector trained on custom dataset
# 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()]

# 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

# loop over frames from the video file stream
while True:
    # read the next frame from the file
    (grabbed, frame) = vs.read()
    # if the frame was not grabbed, then we have reached the end
    # of the stream
    if not grabbed:
        break

    # if the frame dimensions are empty, grab them
    if W is None or H is None:
        (H, W) = frame.shape[:2]

    # construct a blob from the input frame and then perform a forward
    # pass of the YOLO object detector, giving us our bounding boxes
    # and associated probabilities
    blob = cv2.dnn.blobFromImage(frame, 1 / 255.0, (416, 416),
        swapRB=True, crop=False)
    net.setInput(blob)
    start = time.time()
    layerOutputs = net.forward(ln)
    end = time.time()

    # initialize our lists of detected bounding boxes, confidences,
    # and class IDs, respectively
    boxes = []
    confidences = []
    classIDs = []
```

```
# loop over each of the layer outputs
for output in layerOutputs:
    # loop over each of the detections
        for detection in output:
        # extract the class ID and confidence (i.e., probability)
        # of the current object detection
        scores = detection[5:]
        classID = np.argmax(scores)
        confidence = scores[classID]

        # filter out weak predictions by ensuring the detected
        # probability is greater than the minimum probability
        if confidence > args["confidence"]:
            # scale the bounding box coordinates back relative to
            # the size of the image, keeping in mind that YOLO
            # actually returns the center (x, y)-coordinates of
            # the bounding box followed by the boxes' width and
            # height
            box = detection[0:4] * np.array([W, H, W, H])
            (centerX, centerY, width, height) = box.astype("int")

            # use the center (x, y)-coordinates to derive the top
            # and and left corner of the bounding box
            x = int(centerX - (width / 2))
            y = int(centerY - (height / 2))

            # update our list of bounding box coordinates,
            # confidences, and class IDs
            boxes.append([x, y, int(width), int(height)])
            confidences.append(float(confidence))
            classIDs.append(classID)

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

# 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 frame
        color = [int(c) for c in COLORS[classIDs[i]]]
        cv2.rectangle(frame, (x, y), (x + w, y + h), color, 2)
        text = "{}: {:.4f}".format(LABELS[classIDs[i]],
            confidences[i])
        cv2.putText(frame, text, (x, y - 5),
                cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)

    # check if the video writer is None
    if writer is None:
        # initialize our video writer
        fourcc = cv2.VideoWriter_fourcc(*"MJPG")
        writer = cv2.VideoWriter(args["output"], fourcc, 30,
            (frame.shape[1], frame.shape[0]), True)

        # some information on processing single frame
        if total > 0:
            elap = (end - start)
            print("[INFO] single frame took {:.4f} seconds".format(elap))
            print("[INFO] estimated total time to finish: {:.4f}".format(
                elap * total))

    # write the output frame to disk
    writer.write(frame)

# release the file pointers
print("[INFO] cleaning up...")
writer.release()
vs.release()
```