Chapter 13

A Tutorial of Object Detection

13.1 Learning Objectives

In the previous chapter, we described algorithms and architectures of YOLO. In this chapter, we will present a tutorial to demonstrate how to implement YOLO v3 from scratch using Pytorch framework with a pre-trained model (weights of ConvNet). This tutorial is based on the blog at https://blog.paperspace.com/how-to-implement-a-yolo-object-detector-in-pytorch/ . The purpose of this tutorial is twofold: 1) to give a step-by-step guide to implement YOLO v3 in Python using PyTorch framework; and 2) to practice the concepts learned in the previous chapter, so that we can really understand the principle of object detection in computer vision.

To make the most of this tutorial, one should read the code line-by-line with the help of the explanations in the text, and try to understand the function of each code block. After this tutorial, you will be able to

- Implement YOLO v3 from scratch on images or videos
- Modify the code to customize your own YOLO v3 implementation

13.2 YOLO v3 Architecture

To make the text self-contented in this tutorial, we revisit the YOLO v2 architecture, shown in Fig.1. The feature extraction ConvNet is composed of a series of 1x1 or 3x3 convolutional layers with 2-layer skip connections occasionally (ResNet). Two upsample layers are used to generate three scale feature map resolutions (grid sizes). The detailed specification of each layer can be seen in Fig.1.

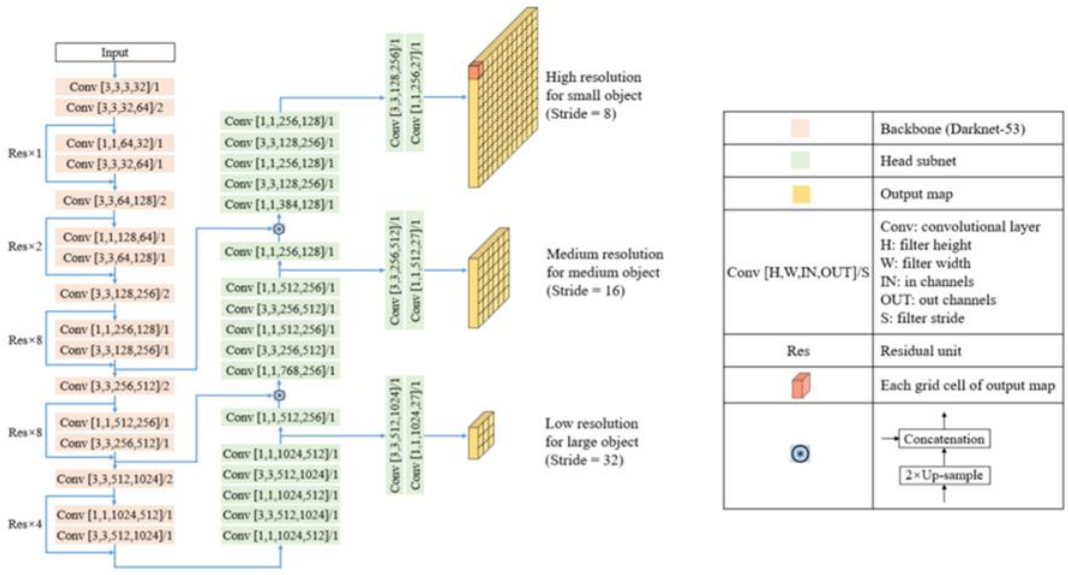

Fig. 1 Architecture of YOLO v2 network

13.3 File Structure

This tutorial was inspired by the github [1] at https://github.com/ayooshkathuria/YOLO_v3_tutorial_from_scratch. We download the files from the github, and unzip to the working directory *YOLO_v3_tutorial*. In YOLO_v3_tutorial folder, as shown in Fig.2, there are 8 files: darknet.py, detect.py, util.py, video.py, dog-cycle-car.png, pallete, yolov3.weights, README.md, and three folders: cfg, data, imgs. The folder /cfg stores configuration file yolov3.cfg. There are two files COCO.names and VOC.names in folder /data for the names of classes. Folder imgs stores some images. The file *darknet.py* defines Darknet-based ConvNet for the detection. The file util.py defines some basic functions which will be used in the main file detect.py, mainly for preparing inputs and post-processing the output tensors from the ConvNet. Similar to detect.py, video.py detect objects on video frames, instead of images. Another important file is yolov3.weights that is a pre-trained model weights file downloaded at https://pjreddie.com/media/files/yolov3.weights. Other files and folders were created for a testing purpose during the tutorial.

Since the code provided in the following text is only for an explanation purpose and thus may be incomplete, one is strongly recommended to download all files directly from the website of the textbook, instead of copying and pasting from the following text.

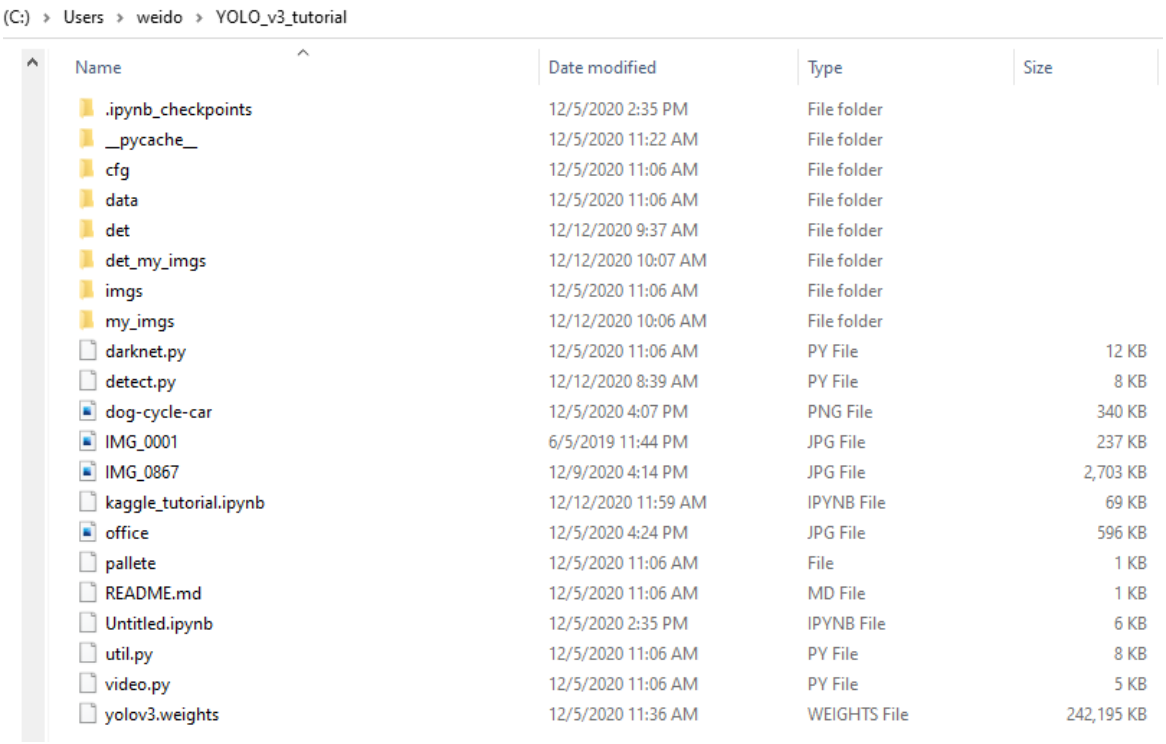

Fig.2 File structure for the tutorial

13.4 Configuration File (yolov3.cfg)

We create a configuration file to describe the layout of a neural network, block by block, and then develop a function parse_cfg to parse and construct PyTorch modules for the blocks present in the config file. The config file (yolov3.cfg) corresponding to YOLO v3 [2] can be downloaded at https://github.com/pjreddie/darknet/blob/master/cfg/yolov3.cfg. In this section, we will explain the typical blocks in yolov3.cfg. The config file consists of a sequence of blocks. Each block describes a layer. Note that each layer is explicitly specified by a block and the order of the blocks matters. It is helpful to refer to Fig.1 for understanding the config file.

Examples of blocks in config file are shown in Fig.3. The block **[net]** describes the information on the network input and training parameters. It isn't used in the forward pass of YOLO. The block **[convolutional]** defines the structure of a conv layer, including the number of filters, filter size, stride, zero-padding, batch normalization and activation settings. The number of input channels for a layer is automatically specified by the output of the previous layer. A **[shourtcut]** block is a skip connection: *from* parameter is -3, which means the output of the shortcut layer is obtained by adding feature maps from the previous (i.e. the first layer backwards) and the 3rd layer backwards from the shortcut layer.

```
[net]
# Testing
batch=1
subdivisions=1
# Training
# batch=64
# subdivisions=16
width= 416
height = 416
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

learning_rate=0.001
burn_in=1000
max_batches = 500200
policy=steps
steps=400000,450000
scales=.1,.1

[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky

[shortcut]
from=-3
activation=linear

[route]
layers = -4

[route]
layers = -1, 61

[upsample]
stride=2

[yolo]
mask = 6,7,8
anchors = 10,13,  16,30,  33,23,  30,61,
62,45,  59,119,  116,90,  156,198,
373,326
classes=80
num=9
jitter=.3
ignore_thresh = .5
truth_thresh = 1
random=1
```

Fig.3 Block examples of configuration file (note that this is not a complete config file for YOLO v3)

The **[route]** layer has an attribute *layers* which can have either one, or two values. When *layers* attribute has only one value, it outputs the feature maps of the layer indexed by the value. In our example, it is -4, so the layer will output feature map from the 4th layer backwards from the Route layer. When *layers* has two values, it returns the concatenated feature maps of the layers indexed by it's values. In our example it is -1, 61, and the layer will output feature maps from the previous layer (-1) and the 61st layer, concatenated along the depth dimension. The **[upsample]** layer upsamples the feature map in the previous layer by a factor of stride using bilinear upsampling.

The **[yolo]** layer corresponds to the detection layer, which processes the output feature map tensor for detection. The *anchors* describes 9 anchors, but only the anchors which are indexed by attributes of the *mask* tag are used. Here, the value of mask is 6, 7, 8, which means the last three anchors are used. In total, we have detection layers at 3 grid scales, making up for a total of 9 anchors with 3 anchors per scale. Other parameters, such as *classes*, *num* and so on, are defined.

13.5 darknet.py

First, we add the required imports at the top of the darknet.py:

```
# in darknet.py
from __future__ import division
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import numpy as np
from util import *
```

13.5.1 Parsing the configuration file

The purpose of the function *parse_cfg* is to parse the cfg, and store every block as a dict. The attributes of the blocks and their values are stored as key-value pairs in the dictionary. As we parse through the cfg, we keep appending these dicts, denoted by the variable block in our code, to a list *blocks*. Our function will return this list *blocks*.

```
# in darknet
def parse_cfg(cfgfile):
    """
    Takes a configuration file

    Returns a list of blocks. Each blocks describes a block in the neural
    network to be built. Block is represented as a dictionary in the list

    """

    file = open(cfgfile, 'r')
    lines = file.read().split('\n')                        # store the lines in a list
    lines = [x for x in lines if len(x) > 0]               # get rid of the empty lines
    lines = [x for x in lines if x[0] != '#']              # get rid of comments
    lines = [x.rstrip().lstrip() for x in lines]           # get rid of fringe whitespaces

    block = {}
    blocks = []
```

```
    for line in lines:
        if line[0] == "[":               # This marks the start of a new block
            if len(block) != 0:          # If block is not empty, implies it is storing values of previous block.
                blocks.append(block)     # add it the blocks list
                block = {}               # re-init the block
            block["type"] = line[1:-1].rstrip()
        else:
            key,value = line.split("=")
            block[key.rstrip()] = value.lstrip()
    blocks.append(block)

    return blocks
```

13.5.2 Creating the building blocks

Now we will use the list returned by function parse_cfg to construct PyTorch modules specified in the config file. There are 5 types of layers in the config file: *convolutional*, *upsample*, *shortcut*, *route*, and *yolo*. PyTorch provides pre-built layers for types *convolutional* and *upsample*. We will have to write our own modules for the other three types by extending the *nn.Module* class.

The *create_modules* function takes a list *blocks* returned by the parse_cfg function, and returns net_info and module_list. net_info stores information about the network.

```
# in darknet.py
class EmptyLayer(nn.Module):
    def __init__(self):
        super(EmptyLayer, self).__init__()

class DetectionLayer(nn.Module):
    def __init__(self, anchors):
        super(DetectionLayer, self).__init__()
        self.anchors = anchors

def create_modules(blocks):
    net_info = blocks[0]     #Captures the information about the input and pre-processing
    module_list = nn.ModuleList()
    prev_filters = 3
    output_filters = []

    for index, x in enumerate(blocks[1:]):
        module = nn.Sequential()

        #check the type of block
        #create a new module for the block
        #append to module_list

        #If it's a convolutional layer
        if (x["type"] == "convolutional"):
            #Get the info about the layer
            activation = x["activation"]
            try:
                batch_normalize = int(x["batch_normalize"])
                bias = False
```

```
        except:
            batch_normalize = 0
            bias = True

        filters= int(x["filters"])
        padding = int(x["pad"])
        kernel_size = int(x["size"])
        stride = int(x["stride"])

        if padding:
            pad = (kernel_size - 1) // 2
        else:
            pad = 0

        #Add the convolutional layer
        conv = nn.Conv2d(prev_filters, filters, kernel_size, stride, pad, bias = bias)
        module.add_module("conv_{0}".format(index), conv)

        #Add the Batch Norm Layer
        if batch_normalize:
            bn = nn.BatchNorm2d(filters)
            module.add_module("batch_norm_{0}".format(index), bn)

        #Check the activation.
        #It is either Linear or a Leaky ReLU for YOLO
        if activation == "leaky":
            activn = nn.LeakyReLU(0.1, inplace = True)
            module.add_module("leaky_{0}".format(index), activn)

        #If it's an upsampling layer
        #We use Bilinear2dUpsampling
    elif (x["type"] == "upsample"):
        stride = int(x["stride"])
        upsample = nn.Upsample(scale_factor = 2, mode = "nearest")
        module.add_module("upsample_{}".format(index), upsample)

    #If it is a route layer
    elif (x["type"] == "route"):
        x["layers"] = x["layers"].split(',')
        #Start  of a route
        start = int(x["layers"][0])
        #end, if there exists one.
        try:
            end = int(x["layers"][1])
        except:
            end = 0
        #Positive anotation
        if start > 0:
            start = start - index
        if end > 0:
            end = end - index
        route = EmptyLayer()
        module.add_module("route_{0}".format(index), route)
        if end < 0:
```

```
            filters = output_filters[index + start] + output_filters[index + end]
        else:
            filters= output_filters[index + start]

    #shortcut corresponds to skip connection
    elif x["type"] == "shortcut":
        shortcut = EmptyLayer()
        module.add_module("shortcut_{}".format(index), shortcut)

    #Yolo is the detection layer
    elif x["type"] == "yolo":
        mask = x["mask"].split(",")
        mask = [int(x) for x in mask]

        anchors = x["anchors"].split(",")
        anchors = [int(a) for a in anchors]
        anchors = [(anchors[i], anchors[i+1]) for i in range(0, len(anchors),2)]
        anchors = [anchors[i] for i in mask]

        detection = DetectionLayer(anchors)
        module.add_module("Detection_{}".format(index), detection)

    module_list.append(module)
    prev_filters = filters
    output_filters.append(filters)

return (net_info, module_list)
```

we can test the code by running the following two lines after running darknet.py.

```
blocks = parse_cfg("cfg/yolov3.cfg")
print(create_modules(blocks))
```

The output looks like:

```
({'type': 'net', 'batch': '1', 'subdivisions': '1', 'width': '416', 'height': '416', 'channels': '3', 'momentum': '0.9', 'decay': '0.0005',
'angle': '0', 'saturation': '1.5', 'exposure': '1.5', 'hue': '.1', 'learning_rate': '0.001', 'burn_in': '1000', 'max_batches': '500200',
'policy': 'steps', 'steps': '400000,450000', 'scales': '.1,.1'}, ModuleList(
  (0): Sequential(
    (conv_0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (batch_norm_0): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (leaky_0): LeakyReLU(negative_slope=0.1, inplace=True)
  )
  (1): Sequential(
    (conv_1): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (batch_norm_1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (leaky_1): LeakyReLU(negative_slope=0.1, inplace=True)
  )
  (2): Sequential(
    (conv_2): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (batch_norm_2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (leaky_2): LeakyReLU(negative_slope=0.1, inplace=True)
  )
  (3): Sequential(
```

```
…..
(106): Sequential(
   (Detection_106): DetectionLayer()
  )
))
```

13.5.3 Define Network

In the **darknet.py** file, we add the following class, ***Darknet***, a subclass of nn.Module. The network is initialized with blocks, net_info and module_list, which were generated by function parse_cfg and create_modules.

A) Forward pass

The forward pass of the network is implemented by overriding the *forward* method of the nn.Module class. *forward* serves two purposes. First, to calculate the output, and second, to transform the output detection feature maps in a way that it can be processed easier (such as transforming them such that detection maps across multiple scales can be concatenated, which otherwise isn't possible as they are of different dimensions). *forward* takes three arguments, *self*, the input *x* and *CUDA*, which if true, would use GPU to accelerate the forward pass.

The iteration begins with self.blocks[1:] instead of self.blocks since the first element of self.blocks is a net block which isn't a part of the forward pass. Since route and shortcut layers need output maps from previous layers, we cache the output feature maps of every layer in a dict outputs. The keys are the indices of the layers, and the values are the feature maps.

In the iteration loop, it is straightforward for *convolutional* or *upsample* layers. For *route* layers, we have to concatenate two feature maps using the *torch.cat* function with the second argument as 1. This is because we want to concatenate the feature maps along the depth. (In PyTorch, input and output of a convolutional layer has the format [B, C, H, W]. The depth corresponds to the channel dimension). For *shourcut* layers, the output is obtained by adding the previous output to the backward layer (defined by parameter *from*) output.

The *yolo* layers will re-organize the output feature maps from the feature extractor ConvNet. The output of feature extractor ConvNet is a tensor that contains the bounding box attributes along the depth of the feature map: [batch, H, W, Depth]. For example, for the 13x13 grid scale and 80 classes, the feature map is [batch, 13, 13, 255] where each grid cell contains 3 bounding boxes. For a different grid scale, the tensor has a different dimension. This 4-D format is inconvenient for output processing such as thresholding by an object confidence, adding grid offsets to centers, and applying anchors etc. We use a function *predict_transform* to convert the 4-D tensor to a 3-D tensor [batch, HxWx3, 5+C] that is more convenient for post-processing. The function *predic_transform* is defined in util.py. *predict_transform* takes in 5 parameters; prediction (3D tensor), inp_dim (input image dimension), anchors, num_classes, and an optional CUDA flag. First, predict_transform function takes a feature map (say, [batch, 13, 13, 255]) and turns it to a tensor: [batch, 13x13x3, 85], where each row of the tensor (i.e. [batch, box, :]) corresponds to attributes of a bounding box, in the following order shown in Fig.4.

```
# in util.py
def predict_transform(prediction, inp_dim, anchors, num_classes, CUDA = True):
   batch_size = prediction.size(0)
   stride =  inp_dim // prediction.size(2)
```

```
grid_size = inp_dim // stride
bbox_attrs = 5 + num_classes
num_anchors = len(anchors)

prediction = prediction.view(batch_size, bbox_attrs*num_anchors, grid_size*grid_size)
prediction = prediction.transpose(1,2).contiguous()
prediction = prediction.view(batch_size, grid_size*grid_size*num_anchors, bbox_attrs)
anchors = [(a[0]/stride, a[1]/stride) for a in anchors]

#Sigmoid the  centre_X, centre_Y. and object confidencce
prediction[:,:,0] = torch.sigmoid(prediction[:,:,0])
prediction[:,:,1] = torch.sigmoid(prediction[:,:,1])
prediction[:,:,4] = torch.sigmoid(prediction[:,:,4])

#Add the center offsets
grid = np.arange(grid_size)
a,b = np.meshgrid(grid, grid)
x_offset = torch.FloatTensor(a).view(-1,1)
y_offset = torch.FloatTensor(b).view(-1,1)

if CUDA:
    x_offset = x_offset.cuda()
    y_offset = y_offset.cuda()
x_y_offset = torch.cat((x_offset, y_offset), 1).repeat(1,num_anchors).view(-1,2).unsqueeze(0)

prediction[:,:,:2] += x_y_offset

#log space transform height and the width
anchors = torch.FloatTensor(anchors)
if CUDA:
    anchors = anchors.cuda()
anchors = anchors.repeat(grid_size*grid_size, 1).unsqueeze(0)
prediction[:,:,2:4] = torch.exp(prediction[:,:,2:4])*anchors
prediction[:,:,5: 5 + num_classes] = torch.sigmoid((prediction[:,:, 5 : 5 + num_classes]))
prediction[:,:,:4] *= stride
return prediction
```

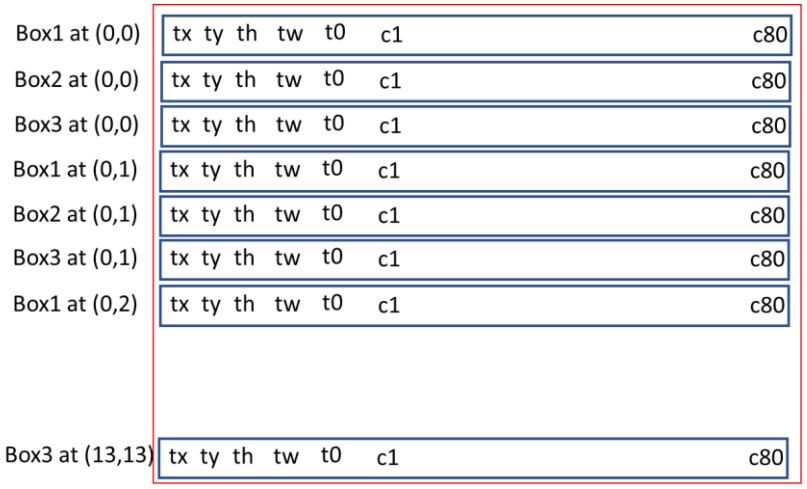

Fig.4 Re-organize the feature map

Then, predict_transform function performs the following coordinate transformation and applies the sigmoid function to the object confidence t0 and conditional class probabilities c1,…, c80. Finally, predict_transform returns the result: a tensor prediction [batch, 13x13x3, 85] after the sigmoid function and exponential function are applied. The details about predic_transform can be found in utils.py.

$$
\begin{aligned}
b_x &= \sigma(t_x) + c_x \\
b_y &= \sigma(t_y) + c_y \\
b_w &= p_w e^{t_w} \\
b_h &= p_h e^{t_h} \\
Pr(\text{object}) * IOU(b, \text{object}) &= \sigma(t_o)
\end{aligned}
$$

where

t_x, t_y, t_w, t_h are predictions made by YOLO.

c_x, c_y is the top left corner of the grid cell of the anchor.

p_w, p_h are the width and height of the anchor.

c_x, c_y, p_w, p_h are normalized by the image width and height.

b_x, b_y, b_w, b_h are the predicted boundary box.

$\sigma(t_o)$ is the box confidence score.

```
# in darknet.py
class Darknet(nn.Module):
    def __init__(self, cfgfile):
        super(Darknet, self).__init__()
        self.blocks = parse_cfg(cfgfile)
        self.net_info, self.module_list = create_modules(self.blocks)

    def forward(self, x, CUDA):
        modules = self.blocks[1:]
        outputs = {}   #We cache the outputs for the route layer

        write = 0
        for i, module in enumerate(modules):
            module_type = (module["type"])

            if module_type == "convolutional" or module_type == "upsample":
                x = self.module_list[i](x)

            elif module_type == "route":
                layers = module["layers"]
                layers = [int(a) for a in layers]

                if (layers[0]) > 0:
                    layers[0] = layers[0] - i

                if len(layers) == 1:
                    x = outputs[i + (layers[0])]

                else:
                    if (layers[1]) > 0:
```

```
                layers[1] = layers[1] - i

            map1 = outputs[i + layers[0]]
            map2 = outputs[i + layers[1]]
            x = torch.cat((map1, map2), 1)


        elif  module_type == "shortcut":
            from_ = int(module["from"])
            x = outputs[i-1] + outputs[i+from_]

        elif module_type == 'yolo':
            anchors = self.module_list[i][0].anchors
            #Get the input dimensions
            inp_dim = int (self.net_info["height"])

            #Get the number of classes
            num_classes = int (module["classes"])

            #Transform
            x = x.data
            x = predict_transform(x, inp_dim, anchors, num_classes, CUDA)
            if not write:              #if no collector has been intialised.
                detections = x
                write = 1

            else:
                detections = torch.cat((detections, x), 1)

        outputs[i] = x

    return detections
```

The three *yolo* layers in the *forward* function deliver three detection tensors: [batch, 13x13x3, 85], [batch, 26x26x3, 85] and [batch, 52x52x3, 85], for three grid scales, respectively. The *forward* function finally concatenates the three tensors along dim=1. The variable *write* is initially set to zero to initialize the concatenation. The resulting tensor is called detections: [batch, 13x13x3+26x26x3+52x52x3, 85] = [batch, 10647, 85]. The function *forward* returns this tensor: *detections*.

Testing the forward pass

To test the forward pass, we write a function, called get_test_input(), to prepare one image data for the model input. This function is located in darknet.py.

```
# in darknet.py
def get_test_input():
    img = cv2.imread("dog-cycle-car.png")
    img = cv2.resize(img, (416,416))          #Resize to the input dimension
    img_ =  img[:,:,::-1].transpose((2,0,1))  # BGR -> RGB | H X W C -> C X H X W
    img_ = img_[np.newaxis,:,:,:]/255.0       #Add a channel at 0 (for batch) | Normalise
    img_ = torch.from_numpy(img_).float()     #Convert to float
    img_ = Variable(img_)                     # Convert to Variable
    return img_
```

Before we run the following testing code, we save an image “dog-cycle-car.png” into the working directory.

```
model = Darknet("cfg/yolov3.cfg")
inp = get_test_input()
pred = model(inp, torch.cuda.is_available())
print (pred)
pred.size()
```

```
tensor([[[1.6210e+01, 1.7484e+01, 1.1186e+02,  ..., 4.8894e-01,
          5.9853e-01, 5.0372e-01],
         [1.9692e+01, 1.5895e+01, 1.7866e+02,  ..., 4.5781e-01,
          3.9541e-01, 5.6321e-01],
         [1.7326e+01, 1.8186e+01, 5.5756e+02,  ..., 4.7512e-01,
          4.5709e-01, 5.0930e-01],
         ...,
         [4.1234e+02, 4.1282e+02, 7.8555e+00,  ..., 5.7244e-01,
          4.5277e-01, 5.7574e-01],
         [4.1179e+02, 4.1197e+02, 2.0278e+01,  ..., 4.7002e-01,
          5.4624e-01, 4.6539e-01],
         [4.1166e+02, 4.1298e+02, 2.9698e+01,  ..., 4.7733e-01,
          5.0632e-01, 4.8636e-01]]])
torch.Size([1, 10647, 85])
```

The shape of the output tensor is 1 x 10647 x 85. The first dimension is the batch size which is simply 1 because we have used a single image. For each image in a batch, we have a 10647 x 85 table. The row of each of this table represents a bounding box.

B) Load weights

Initially our network has random weights. To get the network work correctly, we need to train the network to search for optimal weights. Training a deep network requires a large dataset and careful training settings. For many general applications, pre-trained models are usually available in open sources. In this tutorial, we select to download the yolov3 weights from https://pjreddie.com/media/files/yolov3.weights, then save it to the working directory. This model was trained based on the MSCOCO dataset.

The weights file, yolov3.weights, is a binary file that contains weights stored in a serial fashion. The weights belong to either a batch norm layer or a convolutional layer. They are stored exactly in the same order as they appear in the config file, yolov3.cfg. When a batch norm layer appears in a convolutional block (it is the case for all conv layers except the ones for output), there is no bias. However, when there is no batch norm layer, bias “weights” is present in the weights file.

Now let’s write a function to load weights for the model. It is a member function of the Darknet class. It takes one argument that is the path of the weights file.

```
# in darknet.py
def load_weights(self, weightfile):
    #Open the weights file
    fp = open(weightfile, "rb")
```

```
#The first 5 values are header information
# 1. Major version number
# 2. Minor Version Number
# 3. Subversion number
# 4,5. Images seen by the network (during training)
header = np.fromfile(fp, dtype = np.int32, count = 5)
self.header = torch.from_numpy(header)
self.seen = self.header[3]

weights = np.fromfile(fp, dtype = np.float32)

ptr = 0
for i in range(len(self.module_list)):
   module_type = self.blocks[i + 1]["type"]

   #If module_type is convolutional load weights
   #Otherwise ignore.

   if module_type == "convolutional":
      model = self.module_list[i]
      try:
         batch_normalize = int(self.blocks[i+1]["batch_normalize"])
      except:
         batch_normalize = 0

      conv = model[0]

      if (batch_normalize):
         bn = model[1]

         #Get the number of weights of Batch Norm Layer
         num_bn_biases = bn.bias.numel()

         #Load the weights
         bn_biases = torch.from_numpy(weights[ptr:ptr + num_bn_biases])
         ptr += num_bn_biases

         bn_weights = torch.from_numpy(weights[ptr: ptr + num_bn_biases])
         ptr  += num_bn_biases

         bn_running_mean = torch.from_numpy(weights[ptr: ptr + num_bn_biases])
         ptr  += num_bn_biases

         bn_running_var = torch.from_numpy(weights[ptr: ptr + num_bn_biases])
         ptr  += num_bn_biases

         #Cast the loaded weights into dims of model weights.
         bn_biases = bn_biases.view_as(bn.bias.data)
         bn_weights = bn_weights.view_as(bn.weight.data)
         bn_running_mean = bn_running_mean.view_as(bn.running_mean)
         bn_running_var = bn_running_var.view_as(bn.running_var)

         #Copy the data to model
         bn.bias.data.copy_(bn_biases)
```

```
            bn.weight.data.copy_(bn_weights)
            bn.running_mean.copy_(bn_running_mean)
            bn.running_var.copy_(bn_running_var)

        else:
            #Number of biases
            num_biases = conv.bias.numel()

            #Load the weights
            conv_biases = torch.from_numpy(weights[ptr: ptr + num_biases])
            ptr = ptr + num_biases

            #reshape the loaded weights according to the dims of the model weights
            conv_biases = conv_biases.view_as(conv.bias.data)

            #Finally copy the data
            conv.bias.data.copy_(conv_biases)

        #Let us load the weights for the Convolutional layers
        num_weights = conv.weight.numel()

        #Do the same as above for weights
        conv_weights = torch.from_numpy(weights[ptr:ptr+num_weights])
        ptr = ptr + num_weights

        conv_weights = conv_weights.view_as(conv.weight.data)
        conv.weight.data.copy_(conv_weights)
```

The first 5 int32 values in the weights file constitute the header of the file. The rest of bits represent the weights in a specified order. The weights are stored as 32-bit floats. In the iteration loop over the weights file, we load the weights into the modules of the network. Note that a variable *ptr* is used to keep track of where the operations happen in the weights array. Now we can load weights in Darknet model by calling load_weights function

```
model = Darknet("cfg/yolov3.cfg")
model.load_weights("yolov3.weights")
```

13.6 Object Confidence Thresholding and Non-Max Suppression

So far, we have created a model which outputs detections tensor with a dimension: [batch, 10647, 85], where batch is the number of images in a batch, 10647 is the total number of bounding boxes predicted per image, and 85 is the number of attributes per bounding box. However, most of the bounding boxes don't include any object or they redundantly detect the same object. Thus, we need to apply objectness score thresholding and non-max suppression to 10647 boxes, to select the bounding boxes that truly indicate the final detection and localization. To do this, we create a function, ***write_results***, in the file **util.py**. The function takes as input: prediction (output tensor from darkent), confidence (objectness score threshold), num_classes (80 in our case) and nms_conf (the NMS IoU threshold), and returns output (the final bounding boxes).

```
# in util.py
def write_results(prediction, confidence, num_classes, nms_conf = 0.4):
    conf_mask = (prediction[:,:,4] > confidence).float().unsqueeze(2)
```

```
prediction = prediction*conf_mask

box_corner = prediction.new(prediction.shape)
box_corner[:,:,0] = (prediction[:,:,0] - prediction[:,:,2]/2)
box_corner[:,:,1] = (prediction[:,:,1] - prediction[:,:,3]/2)
box_corner[:,:,2] = (prediction[:,:,0] + prediction[:,:,2]/2)
box_corner[:,:,3] = (prediction[:,:,1] + prediction[:,:,3]/2)
prediction[:,:,:4] = box_corner[:,:,:4]

batch_size = prediction.size(0)
write = False

for ind in range(batch_size):
   image_pred = prediction[ind]        #image Tensor
  #confidence threshholding
  #NMS

   max_conf, max_conf_score = torch.max(image_pred[:,5:5+ num_classes], 1)
   max_conf = max_conf.float().unsqueeze(1)
   max_conf_score = max_conf_score.float().unsqueeze(1)
   seq = (image_pred[:,:5], max_conf, max_conf_score)
   image_pred = torch.cat(seq, 1)

   non_zero_ind =  (torch.nonzero(image_pred[:,4]))
   try:
      image_pred_ = image_pred[non_zero_ind.squeeze(),:].view(-1,7)
   except:
      continue

   if image_pred_.shape[0] == 0:
      continue

   #Get the various classes detected in the image
   img_classes = unique(image_pred_[:,-1])  # -1 index holds the class index


   for cls in img_classes:
      #perform NMS
      #get the detections with one particular class
      cls_mask = image_pred_*(image_pred_[:,-1] == cls).float().unsqueeze(1)
      class_mask_ind = torch.nonzero(cls_mask[:,-2]).squeeze()
      image_pred_class = image_pred_[class_mask_ind].view(-1,7)

      #sort the detections such that the entry with the maximum objectness
      #confidence is at the top
      conf_sort_index = torch.sort(image_pred_class[:,4], descending = True )[1]
      image_pred_class = image_pred_class[conf_sort_index]
      idx = image_pred_class.size(0)   #Number of detections

      for i in range(idx):
         #Get the IOUs of all boxes that come after the one we are looking at
         #in the loop
         try:
            ious = bbox_iou(image_pred_class[i].unsqueeze(0), image_pred_class[i+1:])
```

```
            except ValueError:
                break

            except IndexError:
                break

            #Zero out all the detections that have IoU > treshhold
            iou_mask = (ious < nms_conf).float().unsqueeze(1)
            image_pred_class[i+1:] *= iou_mask

            #Remove the non-zero entries
            non_zero_ind = torch.nonzero(image_pred_class[:,4]).squeeze()
            image_pred_class = image_pred_class[non_zero_ind].view(-1,7)

        batch_ind = image_pred_class.new(image_pred_class.size(0), 1).fill_(ind)      #Repeat the batch_id for as
many detections of the class cls in the image
        seq = batch_ind, image_pred_class

        if not write:
            output = torch.cat(seq,1)
            write = True
        else:
            out = torch.cat(seq,1)
            output = torch.cat((output,out))

    try:
        return output
    except:
        return 0
```

We will explain the details of function *write_results*. Please refer to the above code when you read the following explanations in this section.

13.6.1 Object confidence thresholding

The prediction tensor has a format [batch, 10647, 85], which contains information about batchx10647 bounding boxes. The index 4 in each bounding box corresponding the object confidence (objectness score). For each of the bounding box having an objectness score less than a threshold, we set all attributes of the bounding box to zero:

13.6.2 Non-max suppression (NMS)

To calculate the IoU of two boxes, we transform the bounding box in [center_x, center_y, height, width] format to [top-left x, top-left y, right-bottom x, right-bottom y].

The number of true detections in every image may be different. For example, a batch of size 3 where images 1, 2 and 3 have 5, 2, 4 true detections respectively. Therefore, confidence thresholding and NMS have to be done for one image at once. This means, we cannot vectorize the operations involved, and must loop over the first dimension of prediction (containing indexes of images in a batch).

A flag, *write*=***False***, is used to indicate that we have not initialize output, a tensor to be used to collect final bounding boxes across the entire batch. As soon as the first final bounding box is collected, *write* will be set to **True**.

Once inside the loop, let's clean things up a bit. Notice each bounding box row has 85 attributes, out of which 80 are the class scores. At this point, we're only concerned with the class score having the maximum value. So, we remove the 80 class scores from each bounding box, and instead add the index of the class having the maximum values, as well the class score of that class. Then the bounding boxes with objectness score below a threshold are removed. The try-except block is there to handle situations where we get no detections. In that case, we use continue to skip the rest of the loop body for this image. Since non-max suppression (NMS) will be applied across the image one time for each class detected, we need a variable, *img_classes*, to save the class indexes detected in the image before NMS is applied. Since there can be multiple true detections of the same class, we use a function called *unique* to get classes present in any given image. Note that *unique* is defined in **util.py**.

```
# in util.py
def unique(tensor):
    tensor_np = tensor.cpu().numpy()
    unique_np = np.unique(tensor_np)
    unique_tensor = torch.from_numpy(unique_np)

    tensor_res = tensor.new(unique_tensor.shape)
    tensor_res.copy_(unique_tensor)
    return tensor_res
```

Then, we perform NMS in class-wise, that is, for each detected class (in variable *img_classes*), an NMS is performed. Thus, NMS is implemented for a particular class within an iteration loop over *img_classes*. image_pred_class is the detections for a particular class with a descending sort of objectness scores, that is, the maximum objectness socre is at the top, as shown in Fig.5.

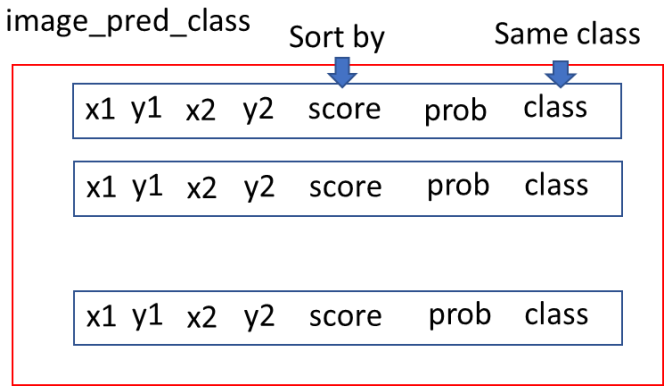

Fig.5 predicted bounding boxes for one class in one image after sorted by objectness scores

NMS algorithm on a class can be described as two steps:

1) Among the output tensor, discard all bounding boxes with pc less than a threshold (e.g. 0.6) (done in Section 13.5.1)
2) do the followings while there are any remaining bounding boxes:
    a) Among these remaining boxes, pick the box with the largest pc, and output this box as a prediction, and then remove this box from “remaining bounding boxes”.

    b) Discard any remaining box with IOU≥ a threshold (e.g. 0.5) with the output box in the step a)

In the iteration loop: *for i in range idx*, we use a function *bbox_iou* to calculate the IoUs of the i-th bounding box in *image_pred_class* with each of the bounding boxes with indices higher than *i*. *bbox_iou* function is defined in **util.py**.

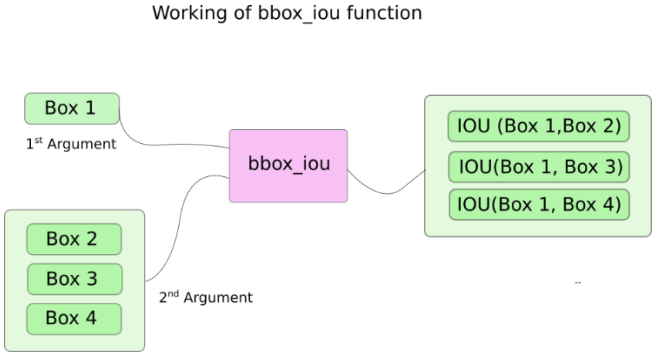

Fig. 6 bbox_iou function to calculate IoUs

```
# in util.py
def bbox_iou(box1, box2):
    """
    Returns the IoU of two bounding boxes


    """
    #Get the coordinates of bounding boxes
    b1_x1, b1_y1, b1_x2, b1_y2 = box1[:,0], box1[:,1], box1[:,2], box1[:,3]
    b2_x1, b2_y1, b2_x2, b2_y2 = box2[:,0], box2[:,1], box2[:,2], box2[:,3]

    #get the corrdinates of the intersection rectangle
    inter_rect_x1 =  torch.max(b1_x1, b2_x1)
    inter_rect_y1 =  torch.max(b1_y1, b2_y1)
    inter_rect_x2 =  torch.min(b1_x2, b2_x2)
    inter_rect_y2 =  torch.min(b1_y2, b2_y2)

    #Intersection area
    inter_area = torch.clamp(inter_rect_x2 - inter_rect_x1 + 1, min=0) * torch.clamp(inter_rect_y2 -
inter_rect_y1 + 1, min=0)

    #Union Area
    b1_area = (b1_x2 - b1_x1 + 1)*(b1_y2 - b1_y1 + 1)
    b2_area = (b2_x2 - b2_x1 + 1)*(b2_y2 - b2_y1 + 1)

    iou = inter_area / (b1_area + b2_area - inter_area)

    return iou
```

Every iteration, if any of the bounding boxes having indices greater than *i* has an IoU (with box indexed by *i*) larger than the threshold *nms_thresh*, then that particular box is eliminated. we have put the line of code to compute the *ious* in a try-catch block. This is because the loop is designed to run *idx* iterations (number of rows in image_pred_class). However, as we proceed with the loop, a number of bounding boxes may be removed from *image_pred_class*. This means, even if one value is removed from *image_pred_class*, we cannot have *idx* iterations. Hence, we might try to index a value that is out of bounds (IndexError), or the slice *image_pred_class[i+1:]* may return

an empty tensor, assigning which triggers a ValueError. At that point, we can ascertain that NMS can remove no further bounding boxes, and we break out of the loop.

Note that a loop that iterates over classes is required to perform NMS for each detected class. At the end of function *write_results*, it outputs a tensor of shape D x 8, as shown in Fig.7. Here D is the total number of true detections in all of images, each represented by a row. Each detection has 8 attributes, namely, index of the image in the batch to which the detection belongs to, 4 corner coordinates, objectness score, the score of class with maximum conditional probability, and the index of that class.

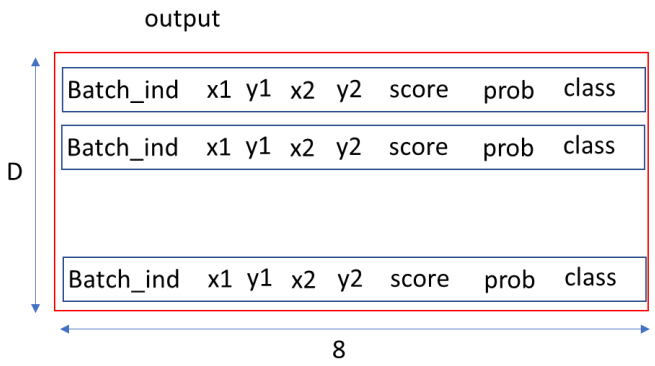

Fig. 7 True detections after NMS for one batch

13.7 Put all Together: detect.py

In the previous sections, we have implemented all components required for YOLO v3, including config file, darknet architecture, weight loading, and post-processes (thresholding and NMS). In this section, we will put all things together to form a pipeline, involving the reading images, making predictions, drawing bounding boxes on images, and saving them to the disk. The main file is detect.py located in the working directory.

13.7.1 Imports

At the top of detect.py, we import all required packages.

```
# in detect.py
from __future__ import division
import time
import torch
import torch.nn as nn
from torch.autograd import Variable
import numpy as np
import cv2
from util import *
import argparse
import os
import os.path as osp
from darknet import Darknet
import pickle as pkl
import pandas as pd
import random
```

13.7.2 Creating command line arguments

We create a function, *arg_parse()*, at the beginning of detect.py, so that we can run the detection by command line with arguments passed to it. For example, we can run it in Jupyter notebook as

[1] %run detect.py

[2] %run detect.py --images dog-cycle-car.png --det det

```
Loading network.....
Network successfully loaded
C:\Users\weido\YOLO_v3_tutorial\dog-cycle-car.png predicted in  1.944 seconds
Objects Detected:    bicycle truck dog
----------------------------------------------------------
SUMMARY
----------------------------------------------------------
Task                 : Time Taken (in seconds)

Reading addresses    : 0.000
Loading batch        : 0.022
Detection (1 images) : 1.947
Output Processing    : 0.000
Drawing Boxes        : 0.029
Average time_per_img : 1.998
----------------------------------------------------------
```

or run it on terminal as

```
(mypytorch) C:\Users\weido\YOLO_v3_tutorial>python detect.py --images dog-cycle-car.png --det det
Loading network.....
Network successfully loaded
C:\Users\weido\YOLO_v3_tutorial\dog-cycle-car.png predicted in  2.812 seconds
Objects Detected:    bicycle truck dog
----------------------------------------------------------
SUMMARY
----------------------------------------------------------
Task                 : Time Taken (in seconds)

Reading addresses    : 0.001
Loading batch        : 0.076
Detection (1 images) : 2.841
Output Processing    : 0.000
Drawing Boxes        : 0.026
Average time_per_img : 2.946
----------------------------------------------------------
```

```
# in detect.py
def arg_parse():
    """
    Parse arguements to the detect module

    """
```

```
    parser = argparse.ArgumentParser(description='YOLO v3 Detection Module')

    parser.add_argument("--images", dest = 'images', help =
                        "Image / Directory containing images to perform detection upon",
                        default = "imgs", type = str)
    parser.add_argument("--det", dest = 'det', help =
                        "Image / Directory to store detections to",
                        default = "det", type = str)
    parser.add_argument("--bs", dest = "bs", help = "Batch size", default = 1)
    parser.add_argument("--confidence", dest = "confidence", help = "Object Confidence to filter predictions",
default = 0.5)
    parser.add_argument("--nms_thresh", dest = "nms_thresh", help = "NMS Threshhold", default = 0.4)
    parser.add_argument("--cfg", dest = 'cfgfile', help =
                        "Config file",
                        default = "cfg/yolov3.cfg", type = str)
    parser.add_argument("--weights", dest = 'weightsfile', help =
                        "weightsfile",
                        default = "yolov3.weights", type = str)
    parser.add_argument("--reso", dest = 'reso', help =
                        "Input resolution of the network. Increase to increase accuracy. Decrease to increase speed",
                        default = "416", type = str)

    return parser.parse_args()
args = arg_parse()
images = args.images
batch_size = int(args.bs)
confidence = float(args.confidence)
nms_thesh = float(args.nms_thresh)
start = 0
CUDA = torch.cuda.is_available()
```

Amongst these, important flags are *images* (used to specify the input image or directory of images), *det* (Directory to save detections to), *reso* (Input image's resolution, can be used for speed - accuracy tradeoff), *cfg* (alternative configuration file) and *weightfile*.

13.7.3 Loading the network

In the working directory, create a folder *data*. Download the file coco.names from https://github.com/pjreddie/darknet/tree/master/data, and save it to folder *data*. Coco.names contains the names of the objects in the COCO dataset. Now in detect.py, we load the class file.

```
# in detect.py
Num_classes = 80
classes = load_classes("data/coco.names")
```

Note that the function load_classes is defined in **util.py** that returns a dictionary which maps the index of every class to a string of its name.

```
# in util.py
def load_classes(namesfile):
    fp = open(namesfile, "r")
    names = fp.read().split("\n")[:-1]
    return names
```

Now, we create the network and load yolov3 weights.

```
# in detect.py
#Set up the neural network
print("Loading network.....")
model = Darknet(args.cfgfile)
model.load_weights(args.weightsfile)
print("Network successfully loaded")

model.net_info["height"] = args.reso
inp_dim = int(model.net_info["height"])
assert inp_dim % 32 == 0
assert inp_dim > 32

#If there's a GPU availible, put the model on GPU
if CUDA:
   model.cuda()

#Set the model in evaluation mode
model.eval()
```

13.7.4 Read images

First, we store the paths of the image/images to a list called *imlist*. If the directory to save the detections, defined by the det flag, doesn't exist, create it.

```
# in detect.py
read_dir = time.time()
#Detection phase
try:
   imlist = [osp.join(osp.realpath('.'), images, img) for img in os.listdir(images)]
except NotADirectoryError:
   imlist = []
   imlist.append(osp.join(osp.realpath('.'), images))
except FileNotFoundError:
   print ("No file or directory with the name {}".format(images))
   exit()

if not os.path.exists(args.det):
   os.makedirs(args.det)
```

Then, we use OpenCV to load the images. Since OpenCV loads an image as a numpy array, with BGR as the order of the color channels, and PyTorch's input format is [batch, channels, H, W], with the channel order of RGB, we need to write a function *pre_image* in **util.py** to transform the numpy array into PyTorch input format. Before applying *pre_image*, we need a function *letterbox_image* (defined in **util.py**) to resize the image, keep the aspect ratio consistent, and pad the left out areas with the color (128, 128,128).

```
# in util.py
def letterbox_image(img, inp_dim):
   '''resize image with unchanged aspect ratio using padding'''
   img_w, img_h = img.shape[1], img.shape[0]
   w, h = inp_dim
   new_w = int(img_w * min(w/img_w, h/img_h))
```

```
    new_h = int(img_h * min(w/img_w, h/img_h))
    resized_image = cv2.resize(img, (new_w,new_h), interpolation = cv2.INTER_CUBIC)

    canvas = np.full((inp_dim[1], inp_dim[0], 3), 128)

    canvas[(h-new_h)//2:(h-new_h)//2 + new_h,(w-new_w)//2:(w-new_w)//2 + new_w,  :] = resized_image

    return canvas

def prep_image(img, inp_dim):
    """
    Prepare image for inputting to the neural network.

    Returns a Variable
    """
    img = (letterbox_image(img, (inp_dim, inp_dim)))
    img = img[:,:,::-1].transpose((2,0,1)).copy()
    img = torch.from_numpy(img).float().div(255.0).unsqueeze(0)
    return img
```

In addition to the transformed image, we also maintain a list of original images, and *im_dim_list*, a list containing the dimensions of the original images. Then batches of images are created.

```
# in detect.py
load_batch = time.time()
loaded_ims = [cv2.imread(x) for x in imlist]

im_batches = list(map(prep_image, loaded_ims, [inp_dim for x in range(len(imlist))]))
im_dim_list = [(x.shape[1], x.shape[0]) for x in loaded_ims]
im_dim_list = torch.FloatTensor(im_dim_list).repeat(1,2)


leftover = 0
if (len(im_dim_list) % batch_size):
    leftover = 1

if batch_size != 1:
    num_batches = len(imlist) // batch_size + leftover
    im_batches = [torch.cat((im_batches[i*batch_size : min((i +  1)*batch_size,
                len(im_batches))]))  for i in range(num_batches)]
```

13.7.5 Detection loop

Through the detection loop iterating over the batches, we generate the prediction tensors of all the images in all batches. For each batch, a prediction tensor [D,8] is generated by *write_results*. The iteration loop concatenates the tensors across all batches.

In the tensor returned by write_results, one of the attributes is the index of the image in batch. In the iteration loop, we transform this index to the index of the image in imlist, the list containing addresses of all images.

```
# in detect.py
write = 0

if CUDA:
    im_dim_list = im_dim_list.cuda()

start_det_loop = time.time()
for i, batch in enumerate(im_batches):
#load the image
    start = time.time()
    if CUDA:
        batch = batch.cuda()
    with torch.no_grad():
        prediction = model(Variable(batch), CUDA)

    prediction = write_results(prediction, confidence, num_classes, nms_conf = nms_thesh)

    end = time.time()

    if type(prediction) == int:

        for im_num, image in enumerate(imlist[i*batch_size: min((i +  1)*batch_size, len(imlist))]):
            im_id = i*batch_size + im_num
            print("{0:20s} predicted in {1:6.3f} seconds".format(image.split("/")[-1], (end - start)/batch_size))
            print("{0:20s} {1:s}".format("Objects Detected:", ""))
            print("----------------------------------------------------------")
        continue

    prediction[:,0] += i*batch_size    #transform the atribute from index in batch to index in imlist

    if not write:                      #If we have't initialised output
        output = prediction
        write = 1
    else:
        output = torch.cat((output,prediction))

    for im_num, image in enumerate(imlist[i*batch_size: min((i +  1)*batch_size, len(imlist))]):
        im_id = i*batch_size + im_num
        objs = [classes[int(x[-1])] for x in output if int(x[0]) == im_id]
        print("{0:20s} predicted in {1:6.3f} seconds".format(image.split("/")[-1], (end - start)/batch_size))
        print("{0:20s} {1:s}".format("Objects Detected:", " ".join(objs)))
        print("----------------------------------------------------------")

    if CUDA:
        torch.cuda.synchronize()
```

13.7.6 Draw bounding boxes on images

First, we use a try-except black to check whether there has been a single detection made or not. If no, exit the program. Since the input size of the network is not the same as the original sizes of the image in general, we need to transform the attributes of each bounding box for the original dimensions of images, so that we can draw the bounding boxes on the images.

The predictions contained in our output tensor are predictions on the padded resized image, and not the original image. We first need to transform the co-ordinates of the boxes to be measured with respect to boundaries of the area on the padded image that contains the original image. We then undo the rescaling to get the co-ordinates of the bounding box on the original image. Finally, we clip any bounding box that has boundaries outside the image to the edges of the image.

To plot the bounding boxes in different colors, we download file *pallete* (https://github.com/ayooshkathuria/YOLO_v3_tutorial_from_scratch/raw/master/pallete) to the working directory. This is a pickled file that contains many colors to randomly choose from.

```
# in detect.py
try:
   output
except NameError:
   print ("No detections were made")
   exit()

im_dim_list = torch.index_select(im_dim_list, 0, output[:,0].long())

scaling_factor = torch.min(inp_dim/im_dim_list,1)[0].view(-1,1)

output[:,[1,3]] -= (inp_dim - scaling_factor*im_dim_list[:,0].view(-1,1))/2
output[:,[2,4]] -= (inp_dim - scaling_factor*im_dim_list[:,1].view(-1,1))/2

output[:,1:5] /= scaling_factor

for i in range(output.shape[0]):
   output[i, [1,3]] = torch.clamp(output[i, [1,3]], 0.0, im_dim_list[i,0])
   output[i, [2,4]] = torch.clamp(output[i, [2,4]], 0.0, im_dim_list[i,1])

output_recast = time.time()
class_load = time.time()
colors = pkl.load(open("pallete", "rb"))

draw = time.time()

def write(x, results):
   c1 = tuple(x[1:3].int())
   c2 = tuple(x[3:5].int())
   img = results[int(x[0])]
   cls = int(x[-1])
   color = random.choice(colors)
   label = "{0}".format(classes[cls])
   cv2.rectangle(img, c1, c2,color, 10)
   t_size = cv2.getTextSize(label, cv2.FONT_HERSHEY_PLAIN, 1 , 1)[0]
   c2 = c1[0] + t_size[0] + 3, c1[1] + t_size[1] + 4
   cv2.rectangle(img, c1, c2,color, -1)
   cv2.putText(img, label, (c1[0], c1[1] + t_size[1] + 4), cv2.FONT_HERSHEY_PLAIN, 1, [225,255,255], 1);
   return img

list(map(lambda x: write(x, loaded_ims), output))

#det_names = pd.Series(imlist).apply(lambda x: "{}/det_{}".format(args.det,x.split("/")[-1]))
det_names = pd.Series(imlist).apply(lambda x: "{}/det_{}".format(args.det,x.split("\\")[-1]))
```

```
list(map(cv2.imwrite, det_names, loaded_ims))

end = time.time()
```

The function *write(x, results)* draws two rectangles with a color of a random choice from colors to the images for each bounding box specified by *results*. One rectangle is defined by a bounding box while another is a filled rectangle located at the top left corner of the bounding box to display the class name. The function finally returns img, which is an image with bounding boxes and class labels.

Once we have *write(x, results)* function, we can modify the images inside *loaded_img* inplace. Each image is saved as its original name prefixed by "*det_*" to the folder */det*.

13.7.7 Printing time summary

At the end, we print a summary containing the execution time for each major part of code. This is useful when we compare how different hyperparameters effect the speed of the detector. Hyperparameters such as batch size, objectness confidence and NMS threshold, (passed with bs, confidence, nms_thresh flags respectively) can be set while executing the script detect.py on the command line.

```
print("SUMMARY")
print("----------------------------------------------------------")
print("{:25s}: {}".format("Task", "Time Taken (in seconds)"))
print()
print("{:25s}: {:2.3f}".format("Reading addresses", load_batch - read_dir))
print("{:25s}: {:2.3f}".format("Loading batch", start_det_loop - load_batch))
print("{:25s}: {:2.3f}".format("Detection (" + str(len(imlist)) +  " images)", output_recast - start_det_loop))
print("{:25s}: {:2.3f}".format("Output Processing", class_load - output_recast))
print("{:25s}: {:2.3f}".format("Drawing Boxes", end - draw))
print("{:25s}: {:2.3f}".format("Average time_per_img", (end - load_batch)/len(imlist)))
print("----------------------------------------------------------")

torch.cuda.empty_cache()
```

13.7.8 Test detect.py

To test the object detector, we can run detect.py either on terminal or in Juptor notebook, with appropriate arguments. For example, to detect object on one image, we just need to specify the image path for the argument *--images*, the the result will be saved to folder */det*.

```
(mypytorch) C:\Users\weido\YOLO_v3_tutorial>python detect.py --images dog-cycle-car.png --det det

%run detect.py --images dog-cycle-car.png  --det det
```

If we want to do the detection on many images, we first create a folder (e.g. *my_imgs*) to store the images, then run

```
%run detect.py --images my_imgs  --det det_my_imgs
```

The resulting images will be saved to the folder det_my_imgs, with prefix det_ for each image. Fig.8 shows some examples of the results. Note that the fist is incorrectly recognized as a sports ball in (b) image, and the giraffe is detected twice.

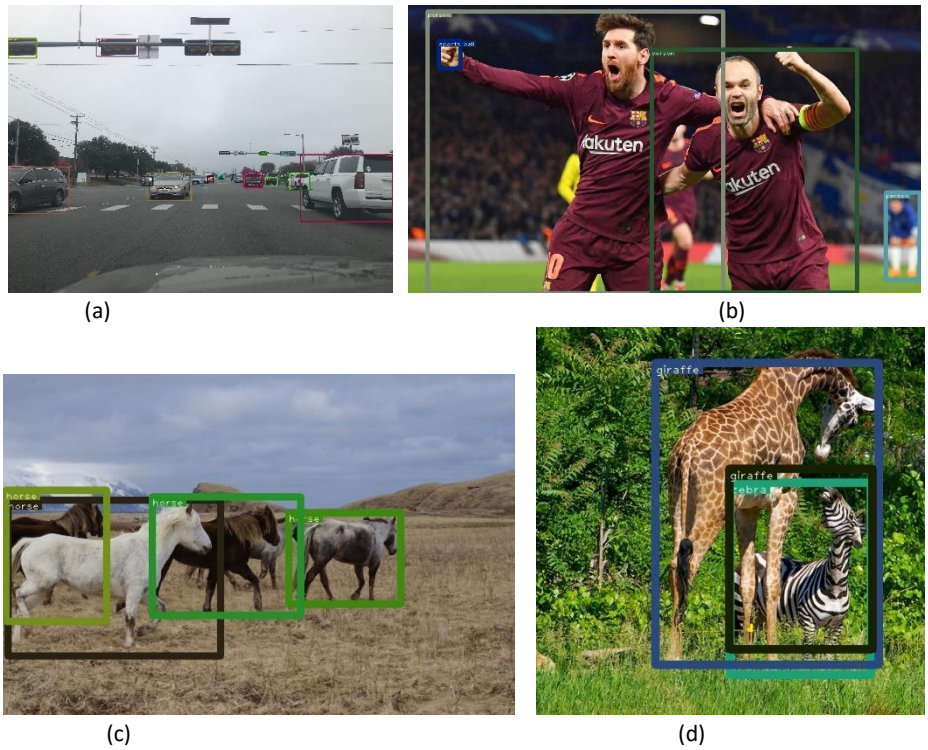

Fig.8 Some examples: (a) 6 cars, 2 trucks and 2 traffic lights detected; (b) two persons detected, and 1 sports ball falsely detected; (c) 4 horses detected; (d) 1 zebra, and 1 giraffe detected twice.

13.7.9 Test video.py

Similarly, we can use the following file video.py to detect objects on video frames or web camera. Note that CPU may not deliver the frames fast enough for a smooth video play. For example, my CPU processed the video at 0.6 Frames/second.

```
[1] %run video.py --video MVI_0805.mp4
```

```
# video.py
from __future__ import division
import time
import torch
import torch.nn as nn
from torch.autograd import Variable
import numpy as np
import cv2
from util import *
import argparse
import os
```

```
import os.path as osp
from darknet import Darknet
import pickle as pkl
import pandas as pd
import random

def arg_parse():
    """
    Parse arguements to the detect module

    """
    parser = argparse.ArgumentParser(description='YOLO v3 Detection Module')
    parser.add_argument("--bs", dest = "bs", help = "Batch size", default = 1)
    parser.add_argument("--confidence", dest = "confidence", help = "Object Confidence to filter predictions",
default = 0.5)
    parser.add_argument("--nms_thresh", dest = "nms_thresh", help = "NMS Threshhold", default = 0.4)
    parser.add_argument("--cfg", dest = 'cfgfile', help =
                        "Config file",
                        default = "cfg/yolov3.cfg", type = str)
    parser.add_argument("--weights", dest = 'weightsfile', help =
                        "weightsfile",
                        default = "yolov3.weights", type = str)
    parser.add_argument("--reso", dest = 'reso', help =
                        "Input resolution of the network. Increase to increase accuracy. Decrease to increase speed",
                        default = "416", type = str)
    parser.add_argument("--video", dest = "videofile", help = "Video file to      run detection on", default =
"video.avi", type = str)

    return parser.parse_args()

args = arg_parse()
batch_size = int(args.bs)
confidence = float(args.confidence)
nms_thesh = float(args.nms_thresh)
start = 0
CUDA = torch.cuda.is_available()

num_classes = 80
classes = load_classes("data/coco.names")

#Set up the neural network
print("Loading network.....")
model = Darknet(args.cfgfile)
model.load_weights(args.weightsfile)
print("Network successfully loaded")

model.net_info["height"] = args.reso
inp_dim = int(model.net_info["height"])
assert inp_dim % 32 == 0
assert inp_dim > 32

#If there's a GPU availible, put the model on GPU
if CUDA:
    model.cuda()
```

```
#Set the model in evaluation mode
model.eval()

def write(x, results):
    c1 = tuple(x[1:3].int())
    c2 = tuple(x[3:5].int())
    img = results
    cls = int(x[-1])
    color = random.choice(colors)
    label = "{0}".format(classes[cls])
    cv2.rectangle(img, c1, c2,color, 1)
    t_size = cv2.getTextSize(label, cv2.FONT_HERSHEY_PLAIN, 1 , 1)[0]
    c2 = c1[0] + t_size[0] + 3, c1[1] + t_size[1] + 4
    cv2.rectangle(img, c1, c2,color, -1)
    cv2.putText(img, label, (c1[0], c1[1] + t_size[1] + 4), cv2.FONT_HERSHEY_PLAIN, 1, [225,255,255], 1);
    return img

#Detection phase
videofile = args.videofile #or path to the video file.
cap = cv2.VideoCapture(videofile)

#cap = cv2.VideoCapture(0) #  for webcam

assert cap.isOpened(), 'Cannot capture source'

frames = 0
start = time.time()

while cap.isOpened():
    ret, frame = cap.read()

    if ret:
        img = prep_image(frame, inp_dim)
#       cv2.imshow("a", frame)
        im_dim = frame.shape[1], frame.shape[0]
        im_dim = torch.FloatTensor(im_dim).repeat(1,2)

        if CUDA:
            im_dim = im_dim.cuda()
            img = img.cuda()

        with torch.no_grad():
            output = model(Variable(img, volatile = True), CUDA)
        output = write_results(output, confidence, num_classes, nms_conf = nms_thesh)

        if type(output) == int:
            frames += 1
            print("FPS of the video is {:5.4f}".format( frames / (time.time() - start)))
            cv2.imshow("frame", frame)
            key = cv2.waitKey(1)
            if key & 0xFF == ord('q'):
                break
            continue
```

```
    im_dim = im_dim.repeat(output.size(0), 1)
    scaling_factor = torch.min(416/im_dim,1)[0].view(-1,1)

    output[:,[1,3]] -= (inp_dim - scaling_factor*im_dim[:,0].view(-1,1))/2
    output[:,[2,4]] -= (inp_dim - scaling_factor*im_dim[:,1].view(-1,1))/2

    output[:,1:5] /= scaling_factor

    for i in range(output.shape[0]):
        output[i, [1,3]] = torch.clamp(output[i, [1,3]], 0.0, im_dim[i,0])
        output[i, [2,4]] = torch.clamp(output[i, [2,4]], 0.0, im_dim[i,1])

    classes = load_classes('data/coco.names')
    colors = pkl.load(open("pallete", "rb"))

    list(map(lambda x: write(x, frame), output))

    cv2.imshow("frame", frame)
    key = cv2.waitKey(1)
    if key & 0xFF == ord('q'):
        break
    frames += 1
    print(time.time() - start)
    print("FPS of the video is {:5.2f}".format( frames / (time.time() - start)))
else:
    break
```

We can move the project to Google Colab at achieve a speed of 14 Frames/second. First, we copy the entire project folder, *\YOLO_v3_tutorial* to Google drive, *\My drive\Colab Notebooks*. Then we launch Colab at https://colab.research.google.com/ , create a new notebook and mount Drive. There are two ways to mount Drive: 1) click the icon "mount Drive" in the "Table of contents"; 2) use statements in the notebook:

```
from google.colab import drive
drive.mount('/content/drive')
```

The later one requires a pass code sent from Google.

After the Drive is mounted, we can see the folder *\drive\My drive\Cola Notebooks\YOLO_v3_tutorial* in the directory *\content*, as shown in Fig.9. In the created notebook, we can run detect.py for images and video.py for video files.

```
%cd drive/My\ Drive/Colab\ Notebooks/YOLO_v3_tutorial/
%run detect.py --images office.jpg --det det
%run video.py --video MVI_0805.MP4
```

Please note that we modify the video.py as follows, so that it saves the video with the detection results into another video file *filename.mp4*, instead of real-time displaying. An example output is shown in Fig.10.

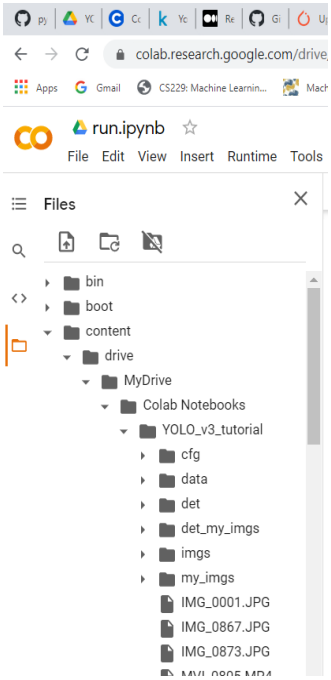

Fig.9 folder structure in Colab.

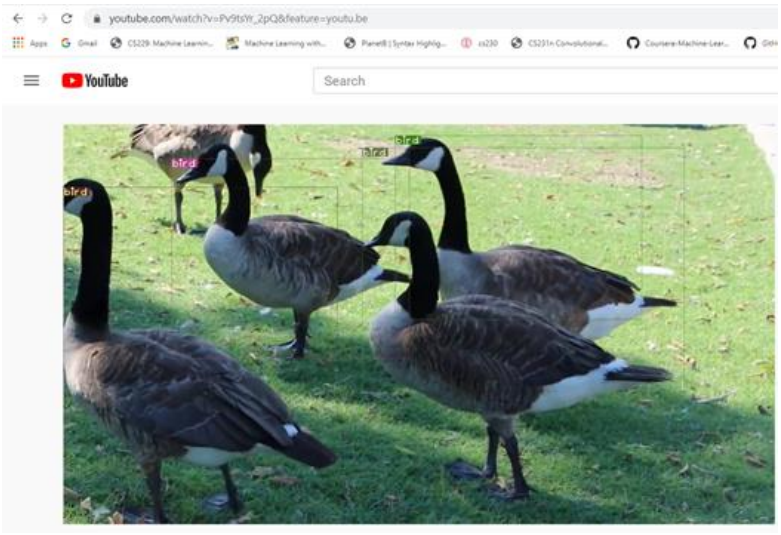

Fig.10 the resulting video posted at YouTube

https://www.youtube.com/watch?v=Pv9tsYr_2pQ&feature=youtu.be

```
from __future__ import division
import time
import torch
import torch.nn as nn
from torch.autograd import Variable
import numpy as np
import cv2
from util import *
import argparse
import os
import os.path as osp
from darknet import Darknet
import pickle as pkl
import pandas as pd
import random


def arg_parse():
    """
    Parse arguements to the detect module

    """

    parser = argparse.ArgumentParser(description='YOLO v3 Detection Module')
    parser.add_argument("--bs", dest = "bs", help = "Batch size", default = 1)
```

```
    parser.add_argument("--confidence", dest = "confidence",
              help = "Object Confidence to filter predictions", default = 0.5)
    parser.add_argument("--nms_thresh", dest = "nms_thresh",
                        help = "NMS Threshhold", default = 0.4)
    parser.add_argument("--cfg", dest = 'cfgfile', help =
                        "Config file",
                        default = "cfg/yolov3.cfg", type = str)
    parser.add_argument("--weights", dest = 'weightsfile', help =
                        "weightsfile",
                        default = "yolov3.weights", type = str)
    parser.add_argument("--reso", dest = 'reso', help =
                "Input resolution of the network. Increase to increase accuracy.
               Decrease to increase speed",default = "416", type = str)
    parser.add_argument("--video", dest = "videofile",
                        help = "Video file to     run detection on",
                        default = "video.avi", type = str)

    return parser.parse_args()

args = arg_parse()
batch_size = int(args.bs)
confidence = float(args.confidence)
nms_thesh = float(args.nms_thresh)
start = 0
CUDA = torch.cuda.is_available()


num_classes = 80
classes = load_classes("data/coco.names")


#Set up the neural network
print("Loading network.....")
model = Darknet(args.cfgfile)
model.load_weights(args.weightsfile)
print("Network successfully loaded")

model.net_info["height"] = args.reso
inp_dim = int(model.net_info["height"])
assert inp_dim % 32 == 0
assert inp_dim > 32

#If there's a GPU availible, put the model on GPU
if CUDA:
    model.cuda()

#Set the model in evaluation mode
model.eval()


def write(x, results):
    c1 = tuple(x[1:3].int())
    c2 = tuple(x[3:5].int())
    img = results
    cls = int(x[-1])
    color = random.choice(colors)
    label = "{0}".format(classes[cls])
    cv2.rectangle(img, c1, c2,color, 1)
    t_size = cv2.getTextSize(label, cv2.FONT_HERSHEY_PLAIN, 2 , 2)[0]
    c2 = c1[0] + t_size[0] + 3, c1[1] + t_size[1] + 4
    cv2.rectangle(img, c1, c2,color, -1)
    cv2.putText(img, label, (c1[0], c1[1] + t_size[1] + 4),
                        cv2.FONT_HERSHEY_PLAIN, 2, [225,255,255], 2);
```

```
    return img

#Detection phase

videofile = args.videofile #for path to the video file.

cap = cv2.VideoCapture(videofile)

#cap = cv2.VideoCapture(0) #  for webcam
#################
frame_width = int(cap.get(3))
frame_height = int(cap.get(4))

size = (frame_width, frame_height)

# Below VideoWriter object will create
# a frame of above defined The output
# is stored in 'filename.avi' file.
result = cv2.VideoWriter('filename.mp4',
                         cv2.VideoWriter_fourcc(*'MP4V'),
                         15, size)
#################
assert cap.isOpened(), 'Cannot capture source'

frames = 0
start = time.time()

while cap.isOpened():
    ret, frame = cap.read()

    if ret and frames <400:
        img = prep_image(frame, inp_dim)
#        cv2.imshow("a", frame)
        im_dim = frame.shape[1], frame.shape[0]
        im_dim = torch.FloatTensor(im_dim).repeat(1,2)

        if CUDA:
            im_dim = im_dim.cuda()
            img = img.cuda()

        with torch.no_grad():
            #output = model(Variable(img, volatile = True), CUDA)
            output = model(Variable(img), CUDA)
        output = write_results(output, confidence, num_classes,
                    nms_conf = nms_thesh)

        if type(output) == int:
            frames += 1
            print("FPS of the video is {:5.4f}".format(
                    frames / (time.time() - start)))
            cv2.imshow("frame", frame)
            key = cv2.waitKey(1)
            if key & 0xFF == ord('q'):
                break
            continue


        im_dim = im_dim.repeat(output.size(0), 1)
        scaling_factor = torch.min(416/im_dim,1)[0].view(-1,1)

        output[:,[1,3]] -= (inp_dim - scaling_factor*im_dim[:,0].view(-1,1))/2
        output[:,[2,4]] -= (inp_dim - scaling_factor*im_dim[:,1].view(-1,1))/2
```

```
        output[:,1:5] /= scaling_factor

        for i in range(output.shape[0]):
            output[i, [1,3]] = torch.clamp(output[i, [1,3]], 0.0, im_dim[i,0])
            output[i, [2,4]] = torch.clamp(output[i, [2,4]], 0.0, im_dim[i,1])



        classes = load_classes('data/coco.names')
        colors = pkl.load(open("pallete", "rb"))

        list(map(lambda x: write(x, frame), output))
        result.write(frame)
        frames += 1

        print(time.time() - start)
        print(frames)
        print("FPS of the video is {:5.2f}".format(
                     frames / (time.time() - start)))
    else:
        break
result.release()
```

Summary

This chapter gives a tutorial of implementation of YOLO v3 with a model of pre-trained weights. After revisiting the architecture of YOLO v3, we described the file structure for the tutorial, which specifies what folders and files are required to implement YOLO v3. Then we explained all files line-by-line or block-by-block, including configuration file, darknet.py, util.py, detect.py.

Specifically, the configuration file defines the architecture of YOLO v3. The Python file darknet.py converts the config file *yolov3.cfg* to PyTorch model *Darknet* and loads the pre-trained weights *yolov3.weights*. The file *util.py* includes reusable functions for darknet.py and detect.py, such as object score thresholding and non-max suppression. The main file detect.py puts all components together, including specifying the command line, instantiating the model with pre-trained weights, preparing input images, looping detection, drawing bounding boxes, and saving results.

However, the model implemented in this tutorial is limited to the pre-defined 80 classes. If we are only interested in a particular set of objects in an object detection task, which are not included in COCO training dataset, we need to create our own dataset, and re-train the network using our dataset [3] [4]. Three examples are listed here: 1) detecting different pieces of chess [5]; 2) detecting playing cards [6]; 3) detecting students with raising hands [7]. It would be very beneficial to follow through these examples for a further understanding, and to help us develop some special relevant applications.

References

[1] https://github.com/ayooshkathuria/YOLO_v3_tutorial_from_scratch

[2] https://github.com/pjreddie/darknet/blob/master/cfg/yolov3.cfg

[3] Prabhat Kumar Sahu, A Guide To Build Your Own Custom Object Detector Using YoloV3, https://medium.com/analytics-vidhya/custom-object-detection-with-yolov3-8f72fe8ced79

[4] https://github.com/eriklindernoren/PyTorch-YOLOv3

[5] Joseph Nelson, Training a YOLOv3 Object Detection Model with a Custom Dataset (chess), https://towardsdatascience.com/training-a-yolov3-object-detection-model-with-a-custom-dataset-4981fa480af0

[6] Philip Xu, Conant Raj Kumar, Grace Bramley-Simmons, "Bridge bidding via playing card detection using YOLO v3", https://cs.brown.edu/research/pubs/theses/capstones/2020/xu.philip.pdf.

[7] Lei Luo, Implementing YOLO-V3 Using PyTorch,

http://leiluoray.com/2018/11/10/Implementing-YOLOV3-Using-PyTorch/