

超详细的 **Pytorch** 版 **yolov3** 代码中文注释 汇总

GiantPandaCV 公众号

GiantPandaCV-逍遥王可爱

2020-06-14



Contents

0. 序言	2
第一部分: darknet.py	2
第二部分: utils.py	13
第三部分: detect.py	23
第四部分: video.py	33

0. 序言

版权声明: 此份电子书整理自公众号「GiantPandaCV」, 版权所有 GiantPandaCV, 禁止任何形式的转载, 禁止传播、商用, 违者必究! GiantPandaCV 公众号由专注于技术的一群 90 后创建, 专注于机器学习、深度学习、计算机视觉、图像处理等领域。半年以来已更新 **242** 篇原创技术文章。我们编写了《从零开始学习 YOLOv3》、《从零开始学习 SSD》、《Faster R-CNN 原理和代码讲解》、《多目标跟踪快速入门》等系列原创电子书, 关注后回复对应关键字即可免费领取。每天更新一到两篇相关推文, 希望在传播知识、分享知识的同时能够启发你。欢迎扫描下方二维码关注我们的公众号。



第一部分: darknet.py

一个 yolov3 的 pytorch 版快速实现工程的见 github:

https://github.com/ayoozhkathuria/YOLO_v3_tutorial_from_scratch

该工程的作者写了个入门教程，在这个教程中，作者使用 PyTorch 实现基于 YOLO v3 的目标检测器，该教程一共有五个部分，虽然并没有含有训练部分。链接：

<https://blog.paperspace.com/how-to-implement-a-yolo-object-detector-in-pytorch/>

这个教程已经有了全的翻译版本，分为上下两个部分，上部分的链接：

<https://www.jiqizhixin.com/articles/2018-04-23-3>

下部分的链接：从零开始 PyTorch 项目：YOLO v3 目标检测实现

有了上面这些教程，我自然不会重复之前的工作，而是给出每个程序每行代码最详细全面的小白入门注释，不论基础多差都能看懂，注释到每个语句每个变量是什么意思，只有把工作做细到这个程度，才是真正对我们这些小白有利（大神们请忽略，这只是给小白们看的。）本篇是系列教程的第一篇，详细阐述程序 darknet.py。

话不多说，先看 darknet.py 代码的超详细注释。

```
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 *
```



```
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))    #img 是【h,w,channel】，这里的
    ↪ img[:, :, :-1] 是将第三个维度 channel 从 opencv 的 BGR 转化为 pytorch 的 RGB,
    ↪ 然后 transpose((2,0,1)) 的意思是将
    ↪ [height,width,channel]->[channel,height,width]
    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_

def parse_cfg(cfgfile):
    """
    输入：配置文件路径
    返回值：列表对象，其中每一个元素为一个字典类型对应于一个要建立的神经网络模块（层）
```

```

"""
# 加载文件并过滤掉文本中多余内容
file = open(cfgfile, 'r')
lines = file.read().split('\n') # store the lines
↪ in a list 等价于 readlines
lines = [x for x in lines if len(x) > 0] # 去掉空行
lines = [x for x in lines if x[0] != '#'] # 去掉以 # 开头的注释行
lines = [x.rstrip().lstrip() for x in lines] # 去掉左右两边的空格
↪ (rstrip 是去掉右边的空格, lstrip 是去掉左边的空格)
# cfg 文件中的每个块用 [] 括起来最后组成一个列表, 一个 block 存储一个块的内容, 即每个层
↪ 用一个字典 block 存储。
block = {}
blocks = []

for line in lines:
    if line[0] == "[": # 这是 cfg 文件中一个层 (块) 的开始
        ↪
        if len(block) != 0: # 如果块内已经存了信息, 说明是上一个块的信息
            ↪ 还没有保存
            blocks.append(block) # 那么这个块 (字典) 加入到 blocks 列表中去
            block = {} # 覆盖掉已存储的 block, 新建一个空白块存储
        ↪ 描述下一个块的信息 (block 是字典)
        block["type"] = line[1:-1].rstrip() # 把 cfg 的 [] 中的块名作为键
        ↪ type 的值
        else:
            key,value = line.split("=") # 按等号分割
            block[key.rstrip()] = value.lstrip() # 左边是 key(去掉右空格), 右边是
            ↪ value(去掉左空格), 形成一个 block 字典的键值对
            blocks.append(block) # 退出循环, 将最后一个未加入的 block 加进去
            # print('\n\n'.join([repr(x) for x in blocks]))
    return blocks

# 配置文件定义了 6 种不同 type
# 'net': 相当于超参数, 网络全局配置的相关参数
# {'convolutional', 'net', 'route', 'shortcut', 'upsample', 'yolo'}

# cfg = parse_cfg("cfg/yolov3.cfg")
# print(cfg)

```

```

class EmptyLayer(nn.Module):
    """

```

为 *shortcut layer / route layer* 准备，具体功能不在此实现，在 *Darknet* 类的 *forward* 函数中有体现

```
"""
def __init__(self):
    super(EmptyLayer, self).__init__()
```

class DetectionLayer(nn.Module):

'''yolo 检测层的具体实现，在特征图上使用锚点预测目标区域和类别，功能函数在 *predict_transform* 中'''

```
def __init__(self, anchors):
    super(DetectionLayer, self).__init__()
    self.anchors = anchors
```

def create_modules(blocks):

net_info = *blocks*[0] # *blocks*[0] 存储了 *cfg* 中 [*net*] 的信息，它是一个字典，
 ↪ 获取网络输入和预处理相关信息
module_list = nn.ModuleList() # *module_list* 用于存储每个 *block*，每个 *block*
 ↪ 对应 *cfg* 文件中一个块，类似 [*convolutional*] 里面就对应一个卷积块
prev_filters = 3 # 初始值对应于输入数据 3 通道，用来存储我们需要持续追踪被应用卷积
 ↪ 层的卷积核数量（上一层的卷积核数量（或特征图深度））
output_filters = [] # 我们不仅需要追踪前一层的卷积核数量，还需要追踪之前每个层。随
 ↪ 着不断地迭代，我们将每个模块的输出卷积核数量添加到 *output_filters* 列表上。

for index, x **in** enumerate(blocks[1:]): # 这里，我们迭代 *block*[1:] 而不是
 ↪ *blocks*，因为 *blocks* 的第一个元素是一个 *net* 块，它不属于前向传播。
 module = nn.Sequential() # 这里每个块用 *nn.sequential()* 创建为了一个
 ↪ *module*，一个 *module* 有多个层

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

```
if (x["type"] == "convolutional"):
    ''' 1. 卷积层 '''
    # 获取激活函数/批归一化/卷积层参数（通过字典的键获取值）
    activation = x["activation"]
    try:
        batch_normalize = int(x["batch_normalize"])
        bias = False # 卷积层后接 BN 就不需要 bias
    except:
        batch_normalize = 0
        bias = True # 卷积层后无 BN 层就需要 bias
```

```

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

# 开始创建并添加相应层
# Add the convolutional layer
# nn.Conv2d(self, in_channels, out_channels, kernel_size,
    ↪ stride=1, padding=0, bias=True)
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
# 给定参数负轴系数 0.1
if activation == "leaky":
    activn = nn.LeakyReLU(0.1, inplace = True)
    module.add_module("leaky_{0}".format(index), activn)

elif (x["type"] == "upsample"):
    '''
    2. upsampling layer
    没有使用 Bilinear2dUpsampling
    实际使用的为最近邻插值
    '''
    stride = int(x["stride"])# 这个 stride 在 cfg 中就是 2, 所以下面的
    ↪ scale_factor 写 2 或者 stride 是等价的
    upsample = nn.Upsample(scale_factor = 2, mode = "nearest")
    module.add_module("upsample_{0}".format(index), upsample)

# route layer -> Empty layer

```

```

# route 层的作用: 当 layer 取值为正时, 输出这个正数对应的层的特征, 如果 layer 取
↪ 值为负数, 输出 route 层向后退 layer 层对应层的特征
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: 正值
    if start > 0:
        start = start - index
    if end > 0: # 若 end>0, 由于 end= end - index, 再执行 index + end 输出
        ↪ 的还是第 end 层的特征
        end = end - index
    route = EmptyLayer()
    module.add_module("route_{0}".format(index), route)
    if end < 0: # 若 end<0, 则 end 还是 end, 输出 index+end(而 end<0) 故
        ↪ index 向后退 end 层的特征。
        filters = output_filters[index + start] +
↪ output_filters[index + end]
    else: # 如果没有第二个参数, end=0, 则对应下面的公式, 此时若 start>0, 由于
        ↪ start = start - index, 再执行 index + start 输出的还是第 start 层
        ↪ 的特征; 若 start<0, 则 start 还是 start, 输出 index+start(而
        ↪ start<0) 故 index 向后退 start 层的特征。
        filters= output_filters[index + start]

#shortcut corresponds to skip connection
elif x["type"] == "shortcut":
    shortcut = EmptyLayer() # 使用空的层, 因为它还要执行一个非常简单的操作
↪ (加)。没必要更新 filters 变量, 因为它只是将前一层的特征图添加到后面的层上而已。
    module.add_module("shortcut_{0}".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)# 锚点, 检测, 位置回归, 分类, 这个
↪ 类见 predict_transform 中
        module.add_module("Detection_{}".format(index), detection)

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

    return (net_info, module_list)

class Darknet(nn.Module):
    def __init__(self, cfgfile):
        super(Darknet, self).__init__()
        self.blocks = parse_cfg(cfgfile) # 调用 parse_cfg 函数
        self.net_info, self.module_list = create_modules(self.blocks)# 调用
↪ create_modules 函数

    def forward(self, x, CUDA):
        modules = self.blocks[1:] # 除了 net 块之外的所有, forward 这里用的是
↪ blocks 列表中的各个 block 块字典
        outputs = {} #We cache the outputs for the route layer

        write = 0#write 表示我们是否遇到第一个检测。write=0, 则收集器尚未初始化,
↪ write=1, 则收集器已经初始化, 我们只需要将检测图与收集器级联起来即可。
        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

```



```

# 如果只有一层时。从前面的 if (layers[0]) > 0: 语句中可知, 如果
↪ layer[0]>0, 则输出的就是当前 layer[0] 这一层的特征, 如果
↪ layer[0]<0, 输出就是从 route 层 (第 i 层) 向后退 layer[0] 层那
↪ 一层得到的特征
if len(layers) == 1:
    x = outputs[i + (layers[0])]
# 第二个元素同理
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) # 第二个参数设为 1, 这是因为我们
↪ 希望将特征图沿 anchor 数量的维度级联起来。

elif module_type == "shortcut":
    from_ = int(module["from"])
    x = outputs[i-1] + outputs[i+from_] # 求和运算, 它只是将前一层的特
↪ 征图添加到后面的层上而已

elif module_type == 'yolo':
    anchors = self.module_list[i][0].anchors
    # 从 net_info(实际就是 blocks[0], 即 [net]) 中 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 # 这里得到的是预测的 yolo 层 feature map
    # 在 util.py 中的 predict_transform() 函数利用 x(是传入 yolo 层的
    ↪ feature map), 得到每个格子所对应的 anchor 最终得到的目标
    # 坐标与宽高, 以及出现目标的得分与每种类别的得分。经过
    ↪ predict_transform 变换后的 x 的维度是 (batch_size,
    ↪ grid_size*grid_size*num_anchors, 5+ 类别数量)
    x = predict_transform(x, inp_dim, anchors, num_classes, CUDA)

    if not write: #if no collector has been
    ↪ initialised. 因为一个空的 tensor 无法与一个有数据的 tensor 进行
    ↪ concatenate 操作,

```

```

        detections = x # 所以 detections 的初始化在有预测值出来时才进行,
        write = 1 # 用 write = 1 标记, 当后面的分数出来后, 直接
        ↪ concatenate 操作即可。

    else:
        '''
        变换后 x 的维度是 (batch_size,
        ↪ grid_size*grid_size*num_anchors, 5+ 类别数量), 这里是在维度 1 上进行
        ↪ concatenate, 即按照
            anchor 数量的维度进行连接, 对应教程 part3 中的 Bounding Box
        ↪ attributes 图的行进行连接。yolov3 中有 3 个 yolo 层, 所以
            对于每个 yolo 层的输出先用 predict_transform() 变成每行为一个
        ↪ anchor 对应的预测值的形式 (不看 batch_size 这个维度, x 剩下的
            维度可以看成是一个二维 tensor), 这样 3 个 yolo 层的预测值按照每个方框对
        ↪ 应的行的维度进行连接。得到了这张图处所有 anchor 的预测值, 后面的 NMS 等操作可以一次完成
        '''
        detections = torch.cat((detections, x), 1)# 将在 3 个不同
        ↪ level 的 feature map 上检测结果存储在 detections 里

    outputs[i] = x

    return detections

# blocks = parse_cfg('cfg/yolov3.cfg')
# x,y = create_modules(blocks)
# print(y)

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)# 这里读取 first
    ↪ 5 values 权重
    self.header = torch.from_numpy(header)
    self.seen = self.header[3]

    weights = np.fromfile(fp, dtype = np.float32)# 加载 np.ndarray 中的剩余
    ↪ 权重, 权重是以 float32 类型存储的

```

```
ptr = 0
for i in range(len(self.module_list)):
    module_type = self.blocks[i + 1]["type"] # blocks 中的第一个元素是网
    ↪ 络参数和图像的描述，所以从 blocks[1] 开始读入

    #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"])
            ↪ # 当有 bn 层时, "batch_normalize" 对应值为 1
        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 将从 weights 文件中得到的权重
    ↪ bn_biases 复制到 model 中 (bn.bias.data)
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:# 如果 batch_normalize 的检查结果不是 True, 只需要加载卷积层的偏
    ↪ 置项
    #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)
```

总的来说, darknet.py 程序包含函数 parse_cfg 输入配置文件路径返回一个列表, 其中每一个元素为一个字典类型对应于一个要建立的神经网络模块 (层), 而函数 create_modules 用来创建网络层级, 而 Darknet 类的 forward 函数就是实现网络前向传播函数了, 还有个 load_weights 用来导入预训练的网络权重参数。当然, forward 函数中需要产生需要的预测输出形式, 因此需要变换输出即

函数 `predict_transform` 在文件 `util.py` 中，我们在 `Darknet` 类别的 `forward` 中使用该函数时，将导入该函数。下一篇就要详细注释 `util.py` 了。

第二部分: `utils.py`

本篇接着上一篇去解释 `util.py`。这个程序包含了 `predict_transform` 函数（`Darknet` 类中的 `forward` 函数要用到），`write_results` 函数使我们的输出满足 `objectness` 分数阈值和非极大值抑制（NMS），以得到「真实」检测结果。还有 `prep_image` 和 `letterbox_image` 等图片预处理函数等（前者用来将 `numpy` 数组转换成 `PyTorch` 需要的输入格式，后者用来将图片按照纵横比进行缩放，将空白部分用 (128,128,128) 填充）。话不多说，直接看 `util.py` 的注释。

```
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
import cv2

def unique(tensor):# 因为同一类别可能会有多个真实检测结果，所以我们使用 unique 函数来去
    ↪ 除重复的元素，即一类只留下一个元素，达到获取任意给定图像中存在的类别的目的。
    tensor_np = tensor.cpu().numpy()
    unique_np = np.unique(tensor_np)#np.unique 该函数是去除数组中的重复数字，并进行
    ↪ 排序之后输出
    unique_tensor = torch.from_numpy(unique_np)
    # 复制数据
    tensor_res = tensor.new(unique_tensor.shape)# new(args, *kwargs) 构建 [相
    ↪ 同数据类型] 的新 Tensor
    tensor_res.copy_(unique_tensor)
    return tensor_res

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 coordinates 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
# Intersection area 这里没有对 inter_area 为负的情况进行判断, 后面计算出来的 IOU
↪ 就可能是负的
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

```

```

def predict_transform(prediction, inp_dim, anchors, num_classes, CUDA =
↪ True):
    """
    在特征图上进行多尺度预测, 在 GRID 每个位置都有三个不同尺度的锚
    ↪ 点。predict_transform() 利用一个 scale 得到的 feature map 预测得到的每个 anchor
    ↪ 的属性 (x,y,w,h,s,s_cls1,s_cls2...), 其中 x,y,w,h
    ↪ 是在网络输入图片坐标系下的值, s 是方框含有目标的置信度得分, s_cls1,s_cls_2 等是方框所含
    ↪ 目标对应每类的概率。输入的 feature map(prediction 变量)
    ↪ 维度为 (batch_size, num_anchors*bbox_attrs, grid_size, grid_size), 类似于一个
    ↪ batch 彩色图片 BxCxHxW 存储方式。参数见 predict_transform() 里面的变量。
    ↪ 并且将结果的维度变换成 (batch_size, grid_size*grid_size*num_anchors, 5+ 类别数
    ↪ 量) 的 tensor, 同时得到每个方框在网络输入图片 (416x416) 坐标系下的 (x,y,w,h) 以及方
    ↪ 框含有目标的得分以及每个类的得分。
    """
    batch_size = prediction.size(0)
    # stride 表示的是整个网络的步长, 等于图像原始尺寸与 yolo 层输入的 feature mapr 尺寸
    ↪ 相除, 因为输入图像是正方形, 所以用高相除即可
    stride = inp_dim // prediction.size(2) # 416//13=32
    # feature map 每条边格子的数量, 416//32=13
    grid_size = inp_dim // stride
    # 一个方框属性个数, 等于 5+ 类别数量
    bbox_attrs = 5 + num_classes

```

```

# anchors 数量
num_anchors = len(anchors)
# 输入的 prediction 维度为 (batch_size, num_anchors * bbox_attrs,
    ↪ grid_size, grid_size), 类似于一个 batch 彩色图片 BxCxHxW
# 存储方式, 将它的维度变换成 (batch_size, bbox_attrs*num_anchors,
    ↪ grid_size*grid_size)
prediction = prediction.view(batch_size, bbox_attrs*num_anchors,
    ↪ grid_size*grid_size)
#contiguous: view 只能用在 contiguous 的 variable 上。如果在 view 之前用了
    ↪ transpose, permute 等, 需要用 contiguous() 来返回一个 contiguous copy。
prediction = prediction.transpose(1,2).contiguous()
# 将 prediction 维度转换成 (batch_size, grid_size*grid_size*num_anchors,
    ↪ bbox_attrs)。不看 batch_size,
# (grid_size*grid_size*num_anchors, bbox_attrs) 相当于将所有 anchor 按行排列,
    ↪ 即一行对应一个 anchor 属性, 此时的属性仍然是 feature map 得到的值
prediction = prediction.view(batch_size,
    ↪ grid_size*grid_size*num_anchors, bbox_attrs)
# 锚点的维度与 net 块的 height 和 width 属性一致。这些属性描述了输入图像的维度, 比
    ↪ feature map 的规模大 (二者之商即是步幅)。因此, 我们必须使用 stride 分割锚点。变换
    ↪ 后的 anchors 是相对于最终的 feature map 的尺寸
anchors = [(a[0]/stride, a[1]/stride) for a in anchors]

#Sigmoid the tx, ty. and object confidence.tx 与 ty 为预测的坐标偏移值
prediction[:, :, 0] = torch.sigmoid(prediction[:, :, 0])
prediction[:, :, 1] = torch.sigmoid(prediction[:, :, 1])
prediction[:, :, 4] = torch.sigmoid(prediction[:, :, 4])

# 这里生成了每个格子的左上角坐标, 生成的坐标为 grid x grid 的二维数组, a, b 分别对应这
    ↪ 个二维矩阵的 x,y 坐标的数组, a,b 的维度与 grid 维度一样。每个 grid cell 的尺寸
    ↪ 均为 1, 故 grid 范围是 [0,12] (假如当前的特征图 13*13)
grid = np.arange(grid_size)
a,b = np.meshgrid(grid, grid)
#x_offset 即 cx,y_offset 即 cy, 表示当前 cell 左上角坐标
x_offset = torch.FloatTensor(a).view(-1,1)#view 是 reshape 功能, -1 表示自
    ↪ 适应
y_offset = torch.FloatTensor(b).view(-1,1)

if CUDA:
    x_offset = x_offset.cuda()
    y_offset = y_offset.cuda()
# 这里的 x_y_offset 对应的是最终的 feature map 中每个格子的左上角坐标, 比如有 13 个
    ↪ 格子, 刚 x_y_offset 的坐标就对应为 (0,0),(0,1)···(12,12) .view(-1, 2) 将
    ↪ tensor 变成两列, unsqueeze(0) 在 0 维上添加了一维。

```

```

x_y_offset = torch.cat((x_offset, y_offset),
↪ 1).repeat(1,num_anchors).view(-1,2).unsqueeze(0)

prediction[:, :, :2] += x_y_offset# $bx = \text{sigmoid}(tx) + cx, by = \text{sigmoid}(ty) + cy$ 

anchors = torch.FloatTensor(anchors)

if CUDA:
    anchors = anchors.cuda()
    # 这里的 anchors 本来是一个长度为 6 的 list(三个 anchors 每个 2 个坐标), 然后在 0
    ↪ 维上 (行) 进行了  $grid\_size * grid\_size$  个复制, 在 1 维 (列) 上
    # 一次复制 (没有变化), 即对每个格子都得到三个 anchor。Unsqueeze(0) 的作用是在数组上添
    ↪ 加一维, 这里是在第 0 维上添加的。添加  $grid\_size$  是为了之后的公式  $bw = pw * e^{tw}$  的
    ↪  $tw$ 。
    anchors = anchors.repeat(grid_size*grid_size, 1).unsqueeze(0)
    # 对网络预测得到的矩形框的宽高的偏差值进行指数计算, 然后乘以 anchors 里面对应的宽高 (这
    ↪ 里的 anchors 里面的宽高是对应最终的 feature map 尺寸  $grid\_size$ ),
    # 得到目标的方框的宽高, 这里得到的宽高是相对于在 feature map 的尺寸
    prediction[:, :, 2:4] = torch.exp(prediction[:, :, 2:4])*anchors# 公式
    ↪  $bw = pw * e^{tw}$  及  $bh = ph * e^{th}$ ,  $pw$  为 anchorbox 的长度
    # 这里得到每个 anchor 中每个类别的得分。将网络预测的每个得分用 sigmoid() 函数计算得到
    prediction[:, :, 5: 5 + num_classes] = torch.sigmoid((prediction[:, :, 5 :
    ↪ 5 + num_classes]))

    prediction[:, :, :4] *= stride# 将相对于最终 feature map 的方框坐标和尺寸映射回输
    ↪ 入网络图片 (416x416), 即将方框的坐标乘以网络的 stride 即可

return prediction

'''
必须使我们的输出满足 objectness 分数阈值和非极大值抑制 (NMS), 以得到后文所提到的「真实」
↪ 检测结果。要做到这一点就要用 write_results 函数。
函数的输入为预测结果、置信度 (objectness 分数阈值)、num_classes (我们这里是 80) 和
↪ nms_conf (NMS IoU 阈值)。
write_results() 首先将网络输出方框属性 (x,y,w,h) 转换为在网络输入图片 (416x416)
↪ 坐标系中, 方框左上角与右下角坐标 (x1,y1,x2,y2), 以方便 NMS 操作。
然后将方框含有目标得分低于阈值的方框去掉, 提取得分最高的那个类的得分 max_conf, 同时返回这
↪ 个类对应的序号 max_conf_score,
然后进行 NMS 操作。最终每个方框的属性为 (ind,x1,y1,x2,y2,s,s_cls,index_cls), ind
↪ 是这个方框所属图片在这个 batch 中的序号,

```



```

    x1,y1 是在网络输入图片 (416x416) 坐标系中, 方框左上角的坐标; x2,y2 是在网络输入图片
    ↪ (416x416) 坐标系中, 方框右下角的坐标。
    s 是这个方框含有目标的得分, s_cls 是这个方框中所含目标最有可能的类别的概率得分,
    ↪ index_cls 是 s_cls 对应的这个类别所对应的序号。
    '''
def write_results(prediction, confidence, num_classes, nms_conf = 0.4):
    # confidence: 输入的预测 shape=(1,10647, 85)。conf_mask: shape=(1,10647) =>
    ↪ 增加一维度之后 (1, 10647, 1)
    conf_mask = (prediction[:, :, 4] > confidence).float().unsqueeze(2) # 我们的
    ↪ 预测张量包含有关 Bx10647 边界框的信息。对于含有目标的得分小于 confidence 的每个方框,
    ↪ 它对应的含有目标的得分将变成 0, 即 conf_mask 中对应元素为 0。而保留预测结果中置信度大
    ↪ 于给定阈值的部分 prediction 的 conf_mask
    prediction = prediction*conf_mask # 小于置信度的条目值全为 0, 剩下部分不变。
    ↪ conf_mask 中含有目标的得分小于 confidence 的方框所对应的含有目标的得分为 0,
    # 根据 numpy 的广播原理, 它会扩展成与 prediction 维度一样的 tensor, 所以含有目标的得
    ↪ 分小于 confidence 的方框所有的属性都会变为 0, 故如果没有检测任何有效目标, 则返回值
    ↪ 为 0

    '''
    保留预测结果中置信度大于阈值的 bbox
    下面开始为 nms 准备
    '''

    # prediction 的前五个数据分别表示 (Cx, Cy, w, h, score), 这里创建一个新的数组, 大
    ↪ 小与 predicton 的大小相同
    box_corner = prediction.new(prediction.shape)
    '''
    我们可以将我们的框的 (中心 x, 中心 y, 高度, 宽度) 属性转换成 (左上角 x, 左上角 y, 右
    ↪ 下角 x, 右下角 y)
    这样做用每个框的两个对角坐标能更轻松地计算两个框的 IoU
    '''
    box_corner[:, :, 0] = (prediction[:, :, 0] - prediction[:, :, 2]/2) # x1 = Cx -
    ↪ w/2
    box_corner[:, :, 1] = (prediction[:, :, 1] - prediction[:, :, 3]/2) # y1 = Cy -
    ↪ h/2
    box_corner[:, :, 2] = (prediction[:, :, 0] + prediction[:, :, 2]/2) # x2 = Cx +
    ↪ w/2
    box_corner[:, :, 3] = (prediction[:, :, 1] + prediction[:, :, 3]/2) # y2 = Cy +
    ↪ h/2
    prediction[:, :, 4] = box_corner[:, :, 4] # 计算后的新坐标复制回去

    batch_size = prediction.size(0) # 第 0 个维度是 batch_size
    # output = prediction.new(1, prediction.size(2)+1) # shape=(1,85+1)

```

```

write = False # 拼接结果到 output 中最后返回

# 对每一张图片得分的预测值进行 NMS 操作, 因为每张图片的目标数量不一样, 所以有效得分的方框
    ↪ 的数量不一样, 没法将几张图片同时处理, 因此一次只能完成一张图的置信度阈值的设置和 NMS,
    ↪ 不能将所涉及的操作向量化.
# 所以必须在预测的第一个维度上 (batch 数量) 上遍历每张图片, 将得分低于一定分数的去掉, 对
    ↪ 剩下的方框进行进行 NMS
for ind in range(batch_size):
    image_pred = prediction[ind] # 选择此 batch 中第 ind 个图像的预
    ↪ 测结果, image_pred 对应一张图片中所有方框的坐标 (x1,y1,x2,y2) 以及得分, 是一个二维
    ↪ tensor 维度为 10647x85

    # 最大值索引, 最大值, 按照 dim=1 方向计算
    max_conf, max_conf_score = torch.max(image_pred[:,5:5+ num_classes],
    ↪ 1)# 我们只关心有最大值的类别分数, prediction[:, 5:] 表示每一分类的分数, 返回每一行中
    ↪ 所有类别的得分最高的那个类的得分 max_conf, 同时返回这个类对应的序号 max_conf_score
    # 维度扩展 max_conf: shape=(10647->15) => (10647->15,1) 添加一个列的维度,
        ↪ max_conf 变成二维 tensor, 尺寸为 10647x1
    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)# 我们移除了每一行的这
    ↪ 80 个类别分数, 只保留 bbox4 个坐标以及 objectness 分数, 转而增加了有最大值的类别分数
    ↪ 及索引。
    # 将每个方框的 (x1,y1,x2,y2,s) 与得分最高的这个类的分数 s_cls(max_conf) 和对
        ↪ 应类的序号 index_cls(max_conf_score) 在列维度上连接起来,
    # 即将 10647x5,10647x1,10647x1 三个 tensor 在列维度进行 concatenate 操作,
        ↪ 得到一个 10647x7 的 tensor, (x1,y1,x2,y2,s,s_cls,index_cls)。
    image_pred = torch.cat(seq, 1)# shape=(10647, 5+1+1=7)
    #image_pred[:,4] 是长度为 10647 的一维 tensor, 维度为 4 的列是置信度分数。假
        ↪ 设有 15 个框含有目标的得分非 0, 返回 15x1 的 tensor
    non_zero_ind = (torch.nonzero(image_pred[:,4]))#torch.nonzero 返回的
    ↪ 是索引, 会让 non_zero_ind 是个 2 维 tensor
    try:# try-except 模块的目的是处理无检测结果的情况.non_zero_ind.squeeze() 将
        ↪ 15x1 的 non_zero_ind 去掉维度为 1 的维度, 变成长度为 15 的一维 tensor, 相
        ↪ 当于一个列向量,
        # image_pred[non_zero_ind.squeeze(),:] 是在 image_pred 中找到
            ↪ non_zero_ind 中非 0 目标得分的行的所有元素 (image_pred 维度
            # 是 10647x7, 找到其中的 15 行), 再用 view(-1,7) 将它变为 15x7 的
            ↪ tensor, 用 view() 确保第二个维度必须是 7.
        image_pred_ = image_pred[non_zero_ind.squeeze(),:].view(-1,7)
    except:
        continue

```

```

if image_pred_.shape[0] == 0: # 当没有检测到目标时, 我们使用 continue 来跳
    ↪ 过对本图像的循环, 即进行下一次循环。
    continue

    # 获取当前图像检测结果中出现的所有类别
    img_classes = unique(image_pred_[:-1]) # pred_[:, -1] 是一个 15x7 的
    ↪ tensor, 最后一列保存的是每个框里面物体的类别, -1 表示取最后一列。
    # 用 unique() 除去重复的元素, 即一类只留下一个元素, 假设这里最后只剩下了 3 个元素,
    ↪ 即只有 3 类物体。

    # 按照类别执行 NMS

    for cls in img_classes:
        # 一旦我们进入循环, 我们要做的第一件事就是提取特定类别 (用变量 cls 表示) 的检测
        ↪ 结果, 分离检测结果中属于当前类的数据 -1: index_cls, -2: s_cls
        '''
            本句是将 image_pred_ 中属于 cls 类的预测值保持不变, 其余的全部变成 0。
        ↪ image_pred_[:, -1] == cls, 返回一个与 image_pred_
            行数一样的一维 tensor, 这里长度为 15. 当 image_pred_ 中的最后一个元素 (物体
        ↪ 类别索引) 等于第 cls 类时, 返回的 tensor 对应元素为 1,
            否则为 0. 它与 image_pred_ 相乘时, 先扩展为 15x7 的 tensor (似乎这里还没有
        ↪ 变成 15x7 的 tensor), 为 0 元素一行全部为 0, 再与
            image_pred_ 相乘, 属于 cls 这类的方框对应预测元素不变, 其它类的为
        ↪ 0. unsqueeze(1) 添加了列这一维, 变成 15x7 的二维 tensor。
        '''
        cls_mask = image_pred_*(image_pred_[:-1] ==
        ↪ cls).float().unsqueeze(1)
        class_mask_ind = torch.nonzero(cls_mask[:, -2]).squeeze()
        ↪ # cls_mask[:, -2] 为 cls_mask 倒数第二列, 是物体类别分数。
        # cls_mask 本身为 15x7, cls_mask[:, -2] 将 cls_mask 的倒数第二列取出来, 此时是 1 维
        ↪ tensor, torch.nonzero(cls_mask[:, -2]) 得到的是非零元素的索引,
        # 将返回一个二维 tensor, 这里是 4x2, 再用 squeeze() 去掉长度为 1 的维度 (这里是第二维),
        ↪ 得到一维 tensor (相当于一列)。
        image_pred_class = image_pred_[class_mask_ind].view(-1, 7) # 从
        ↪ prediction 中取出属于 cls 类别的所有结果, 为下一步的 nms 的输入。
        # 找到 image_pred_ 中对应 cls 类的所有方框的预测值, 并转换为二维张量。这里 4x7。
        ↪ image_pred_[class_mask_ind] 本身得到的数据就是 4x7, view(-1, 7) 是为了确保第二维
        ↪ 为 7
        ''' 到此步 prediction_class 已经存在了我们需要进行非极大值抑制的数据 '''
        # 开始 nms
        # 按照 score 排序, 由大到小
        # 最大值最上面

```

```

        conf_sort_index = torch.sort(image_pred_class[:,4], descending =
↪ True )[1]# # 这里的 sort() 将返回两个 tensor，第一个是每个框含有有目标的分数由低到高
↪ 排列，第二个是现在由高到底的 tensor 中每个元素在原来的序号。[0] 是排序结果，[1] 是排序
↪ 结果的索引
        image_pred_class = image_pred_class[conf_sort_index]# 根据排序后的
↪ 索引对应出的 bbox 的坐标与分数，依然为 4x7 的 tensor
        idx = image_pred_class.size(0)    #detections 的个数

        ''' 开始执行 " 非极大值抑制" 操作'''
        for i in range(idx):
            # 对已经有序的结果，每次开始更新后索引加一，挨个与后面的结果比较

            try:# image_pred_class[i].unsqueeze(0)，为什么要加
↪ unsqueeze(0)? 这里 image_pred_class 为 4x7 的 tensor，
↪ image_pred_class[i] 是一个长度为 7 的 tensor，要变成 1x7 的
↪ tensor，在第 0 维添加一个维度。
                ious = bbox_iou(image_pred_class[i].unsqueeze(0),
↪ image_pred_class[i+1:])# 这句话的作用是计算第 i 个方框和 i+1 到最终的所有方框的
↪ IOU。

                except ValueError:
                    '''
                    在 for i in range(idx): 这个循环中，因为有一些框（在
↪ image_pred_class 对应一行）会被去掉，image_pred_class 行数会减少，
                    这样在后面的循环中，idx 序号会超出 image_pred_class 的行数的范围，出现
↪ ValueError 错误。
                    所以当抛出这个错误时，则跳出这个循环，因为此时已经没有更多可以去掉的方框了。
                    '''
                    break

                except IndexError:
                    break

            iou_mask = (ious < nms_conf).float().unsqueeze(1)# 计算出需要保
↪ 留的 item（保留 ious < nms_conf 的框）而 ious < nms_conf 得到的是 torch.uint8
↪ 类型，用 float() 将它们转换为 float 类型。因为要与 image_pred_class[i+1:] 相乘，
↪ 故长度为 7 的 tensor，要变成 1x7 的 tensor，需添加一个维度。
            image_pred_class[i+1:] *= iou_mask # 将 iou_mask 与比序号 i 大的
↪ 框的预测值相乘，其中 IOU 大于阈值的框的预测值全部变成 0。得出需要保留的框

            # 开始移除

```

```

        non_zero_ind =
        ↪ torch.nonzero(image_pred_class[:,4]).squeeze()#torch.nonzero 返回的是索引，
        ↪ 是 2 维 tensor。将经过 iou_mask 掩码后的每个方框含有目标的得分非 0 的方框的索引提取
        ↪ 出来，non_zero_ind 经 squeeze 后为一维 tensor，含有目标的得分非 0 的索引
        image_pred_class =
        ↪ image_pred_class[non_zero_ind].view(-1,7)# 得到含有目标的得分非 0 的方框的预测
        ↪ 值 (x1, y1, x2, y2, s, s_cls,index_cls), 为 1x7 的 tensor
        # 当前类的 nms 执行完之后，下一次循环将对剩下的方框中得分第 i+1 高的方框进行
        ↪ NMS 操作，因为刚刚已经对得分第 1 到 i 高的方框进行了 NMS 操作。直到最后
        ↪ 一个方框循环完成为止
        # 在每次进行 NMS 操作的时候，预测值 tensor 中都会有一些行（对应某些方框）被去
        ↪ 掉。接下来是保存结果。
        # new() 创建了一个和 image_pred_class 类型相同的 tensor，tensor 行数等
        ↪ 于 cls 这个类别所有的方框经过 NMS 剩下的方框的个数，即
        ↪ image_pred_class 的行数，列数为 1。
        # 再将生成的这个 tensor 所有元素赋值为这些方框所属图片对应于 batch 中的序号
        ↪ ind(一个 batch 有多张图片同时测试)，用 fill_(ind) 实现
        batch_ind = image_pred_class.new(image_pred_class.size(0),
        ↪ 1).fill_(ind)
        seq = batch_ind, image_pred_class
        # 我们没有初始化我们的输出张量，除非我们有要分配给它的检测结果。一旦其被初始化，
        ↪ 我们就将后续的检测结果与它连接起来。我们使用 write 标签来表示张量是否初始
        ↪ 化了。在类别上迭代的循环结束时，我们将所得到的检测结果加入到张量输出中。
        if not write:
        # 将 batch_ind, image_pred_class 在列维度上进行连接，image_pred_class
        ↪ 每一行存储的是 (x1,y1,x2,y2,s,s_cls,index_cls)，现在在第一列增加了
        ↪ 一个代表这个行对应方框所属图片在一个 batch 中的序号 ind
            output = torch.cat(seq,1)
            write = True
        else:
            out = torch.cat(seq,1)
            output = torch.cat((output,out))

    try:# 在该函数结束时，我们会检查输出是否已被初始化。如果没有，就意味着在该 batch 的任意
        ↪ 图像中都没有单个检测结果。在这种情况下，我们返回 0。
        return output
    except:# 如果所有的图片都没有检测到方框，则在前面不会进行 NMS 等操作，不会生成 output，
        ↪ 此时将在 except 中返回 0
        return 0
# 最终返回的 output 是一个 batch 中所有图片中剩下的方框的属性，一行对应一个方框，属性为
        ↪ (x1,y1,x2,y2,s,s_cls,index_cls)，
# ind 是这个方框所属图片在这个 batch 中的序号，x1,y1 是在网络输入图片 (416x416) 坐
        ↪ 标系中，方框左上角的坐标；x2,y2 是在网络输入

```

图片 (416x416) 坐标系中, 方框右下角的坐标。 s 是这个方框含有目标的得分 s_cls 是这个方框中所含目标最有可能的类别的概率得分, $index_cls$ 是 s_cls 对应的这个类别所对应的序号

def letterbox_image(img, inp_dim):

"""

$letterbox_image()$ 将图片按照纵横比进行缩放, 将空白部分用 (128,128,128) 填充, 调整图像尺寸

具体而言, 此时某个边正好可以等于目标长度, 另一边小于等于目标长度
将缩放后的数据拷贝到画布中心, 返回完成缩放

"""

$img_w, img_h = img.shape[1], img.shape[0]$

$w, h = inp_dim$ # inp_dim 是需要 $resize$ 的尺寸 (如 416*416)

取 $\min(w/img_w, h/img_h)$ 这个比例来缩放, 缩放后的尺寸为 new_w, new_h , 即保证较长的边缩放后正好等于目标长度 (需要的尺寸), 另一边的尺寸缩放后还没有填满。

$new_w = \text{int}(img_w * \min(w/img_w, h/img_h))$

$new_h = \text{int}(img_h * \min(w/img_w, h/img_h))$

$resized_image = cv2.resize(img, (new_w, new_h), interpolation =$

$cv2.INTER_CUBIC)$ # 将图片按照纵横比不变来缩放为 $new_w \times new_h$, 768 x 576 的图片
缩放成 416x312., 用了双三次插值

创建一个画布, 将 $resized_image$ 数据拷贝到画布中心。

$canvas = np.full((inp_dim[1], inp_dim[0], 3), 128)$ # 生成一个我们最终需要的图片尺寸 $h \times w \times 3$ 的 $array$, 这里生成 416x416x3 的 $array$, 每个元素值为 128

将 $w \times h \times 3$ 的 $array$ 中对应 $new_w \times new_h \times 3$ 的部分 (这两个部分的中心应该对齐) 赋值为刚刚由原图缩放得到的数组, 得到最终缩放后图片

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

return canvas

def prep_image(img, inp_dim): # $prep_image$ 用来将 $numpy$ 数组转换成 $PyTorch$ 需要的输入格式。即 (3, 416, 416)

"""

为神经网络准备输入图像数据

返回值: 处理后图像, 原图, 原图尺寸

"""

$img = (letterbox_image(img, (inp_dim, inp_dim)))$ # $letterbox_image()$ 将图片按照纵横比进行缩放, 将空白部分用 (128,128,128) 填充

$img = img[:, :, ::-1].transpose((2, 0, 1)).copy()$ # img 是 $[h, w, channel]$, 这里的 $img[:, :, ::-1]$ 是将第三个维度 $channel$ 从 $opencv$ 的 BGR 转化为 $pytorch$ 的 RGB ,

然后 $transpose((2, 0, 1))$ 的意思是将

$[height, width, channel] \rightarrow [channel, height, width]$

```

    img = torch.from_numpy(img).float().div(255.0).unsqueeze(0)#
    ↪ from_numpy() 将 ndarray 数据转换为 tensor 格式, div(255.0) 将每个元素除以
    ↪ 255.0, 进行归一化, unsqueeze(0) 在 0 维上添加了一维,
# 从 3x416x416 变成 1x3x416x416, 多出来的一维表示 batch。这里就将图片变成了 BxCxHxW 的
    ↪ pytorch 格式
    return img

def load_classes(namesfile): #load_classes 会返回一个字典——将每个类别的索引映射到
    ↪ 其名称的字符串
    """
    加载类名文件
    :param namesfile:
    :return: 元组, 包括类名数组和总类的个数
    """
    fp = open(namesfile, "r")
    names = fp.read().split("\n")[:-1]
    return names

```

第三部分: detect.py

本篇是第三篇, 主要是对 detect.py 的注释。在这一部分, 我们将为我们的检测器构建输入和输出流程。这涉及到从磁盘读取图像, 做出预测, 使用预测结果在图像上绘制边界框, 然后将它们保存到磁盘上。我们将引入一些命令行标签, 以便能使用该网络的各种超参数进行一些实验。注意代码中有一处错误我进行了修改。源代码在计算 scaling_factor 时, 用的 scaling_factor = torch.min(416/im_dim_list,1)[0].view(-1,1) 显然不对, 应该使用用户输入的 args.reso 即改为 scaling_factor = torch.min(int(args.reso)/im_dim_list,1)[0].view(-1,1)

接下来就开始吧。

```

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():
```

```
    """
```

```
    检测模块的参数转换
```

```
    """
```

```
    # 创建一个 ArgumentParser 对象, 格式: 参数名, 目标参数 (dest 是字典的 key), 帮助信
    ↪ 息, 默认值, 类型
```

```
    parser = argparse.ArgumentParser(description='YOLO v3 检测模型')
```

```
    parser.add_argument("--images", dest = 'images', help =
        " 待检测图像目录",
        default = "imgs", type = str) # images 是所有测试图片所
```

```
    ↪ 在的文件夹
```

```
    parser.add_argument("--det", dest = 'det', help = #det 保存检测结果的目录
        " 检测结果保存目录",
        default = "det", type = str)
```

```
    parser.add_argument("--bs", dest = "bs", help = "Batch size, 默认为 1",
    ↪ default = 1)
```

```
    parser.add_argument("--confidence", dest = "confidence", help = " 目标检测
    ↪ 结果置信度阈值", default = 0.5)
```

```
    parser.add_argument("--nms_thresh", dest = "nms_thresh", help = "NMS 非极
    ↪ 大值抑制阈值", default = 0.4)
```

```
    parser.add_argument("--cfg", dest = 'cfgfile', help =
        " 配置文件",
        default = "cfg/yolov3.cfg", type = str)
    parser.add_argument("--weights", dest = 'weightsfile', help =
        " 模型权重",
        default = "yolov3.weights", type = str)
```

```
    parser.add_argument("--reso", dest = 'reso', help =
        " 网络输入分辨率. 分辨率越高, 则准确率越高; 反之亦然.",
        default = "416", type = str) #reso 输入图像的分辨率, 可用于
```

```
    ↪ 在速度与准确度之间的权衡
```

```
    parser.add_argument("--scales", dest="scales", help=" 缩放尺度用于检测",
    ↪ default="1,2,3", type=str)
```

```
    return parser.parse_args()# 返回转换好的结果
```

```
args = arg_parse()# args 是一个 namespace 类型的变量, 即 argparse.Namespace, 可以
    ↪ 像 easydict 一样使用, 就像一个字典, key 来索引变量的值
```

```
# Namespace(bs=1, cfgfile='cfg/yolov3.cfg', confidence=0.5,det='det',
```

```
    ↪ images='imgs', nms_thresh=0.4, reso='416', weightsfile='yolov3.weights')
```



```
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()# GPU 环境是否可用

num_classes = 80# coco 数据集有 80 类
classes = load_classes("data/coco.names") # 将类别文件载入到我们的程序中,
    ↪ coco.names 文件中保存的是所有类别的名字, load_classes() 返回一个列表 classes, 每个
    ↪ 元素是一个类别的名字

# 初始化网络并载入权重
print(" 载入神经网络...")
model = Darknet(args.cfgfile)# Darknet 类中初始化时得到了网络结构和网络的参数信息, 保
    ↪ 存在 net_info, module_list 中
model.load_weights(args.weightsfile)# 将权重文件载入, 并复制给对应的网络结构 model
    ↪ 中
print(" 模型加载成功.")
# 网络输入数据大小
model.net_info["height"] = args.reso # model 类中 net_info 是一个字典。'' height''
    ↪ 是图片的宽高, 因为图片缩放到 416x416, 所以宽高一样大
inp_dim = int(model.net_info["height"]) #inp_dim 是网络输入图片尺寸 (如
    ↪ 416*416)
assert inp_dim % 32 == 0 # 如果设定的输入图片的尺寸不是 32 的位数或者不大于 32, 抛出异常
assert inp_dim > 32

# 如果 GPU 可用, 模型切换到 cuda 中运行
if CUDA:
    model.cuda()

model.eval()# 变成测试模式, 这主要是对 dropout 和 batch normalization 的操作在训练和
    ↪ 测试的时候是不一样的

read_dir = time.time() #read_dir 是一个用于测量时间的检查点, 开始计时
# 加载待检测图像列表
```

```

try: # 从磁盘读取图像或从目录读取多张图像。图像的路径存储在一个名为 imlist 的列表中,imlist
    ↪ 列表保存了 images 文件中所有图片的完整路径, 一张图片路径对应一个元素。
        #osp.realpath('.') 得到了图片所在文件夹的绝对路径, images 是测试图片文件夹,
        ↪ listdir(images) 得到了 images 文件夹下面所有图片的名字。
        # 通过 join() 把目录(文件夹)的绝对路径和图片名结合起来, 就得到了一张图片的完整路径
        imlist = [osp.join(osp.realpath('.'), images, img) for img in
    ↪ os.listdir(images)]
except NotADirectoryError: # 如果上面的路径有错, 只得到 images 文件夹绝对路径即可
    imlist = []
    imlist.append(osp.join(osp.realpath('.'), images))
except FileNotFoundError:
    print ("No file or directory with the name {}".format(images))
    exit()
# 存储结果目录
if not os.path.exists(args.det): # 如果保存检测结果的目录(由 det 标签定义)不存在, 就
    ↪ 创建一个
        os.makedirs(args.det)

load_batch = time.time() # 开始载入图片的时间。 load_batch - read_dir 得到读取所有图
    ↪ 片路径的时间
loaded_ims = [cv2.imread(x) for x in imlist] # 使用 OpenCV 来加载图像, 将所有的图
    ↪ 片读入, 一张图片的数组在 loaded_ims 列表中保存为一个元素

# 加载全部待检测图像
# loaded_ims 和 [inp_dim for x in range(len(imlist))] 是两个列表, loaded_ims 是
    ↪ 所有图片数组的列表, [inp_dim for x in range(len(imlist))] 遍历 imlist 长度 (即
    ↪ 图片的数量) 这么多次, 每次返回值是图片需 resize 的输入尺寸 inp_dim (如 416)
im_batches = list(map(prepare_image, loaded_ims, [inp_dim for x in
    ↪ range(len(imlist)))) #map 函数将对应的元素作为参数传入 prepare_image 函数, 最终的所
    ↪ 有结果也会组成一个列表 (im_batches), 是 BxCxHxW
im_dim_list = [(x.shape[1], x.shape[0]) for x in loaded_ims] # 除了转换后的图像,
    ↪ 我们也会维护一个列表 im_dim_list 用于保存原始图片的维度。一个元素对应一张图片的宽
    ↪ 高, opencv 读入的图片矩阵对应的是 HxWxC
# 将 im_dim_list 转换为 floatTensor 类型的 tensor, 此时维度为 11x2, (因为本例测试集一
    ↪ 共 11 张图片) 并且每个元素沿着第二维 (列的方向) 进行复制, 最终变成 11x4 的 tensor。一
    ↪ 行的元素为 (W,H,W,H), 对应一张图片原始的宽、高, 且重复了一次。(W,H,W,H) 主要是在后面计
    ↪ 算 x1,y1,x2,y2 各自对应的缩放系数时好对应上。
im_dim_list = torch.FloatTensor(im_dim_list).repeat(1,2)#repeat(*size), 沿着
    ↪ 指定维度复制数据, size 维度必须和数据本身维度要一致

leftover = 0 # 创建 batch, 将所有测试图片按照 batch_size 分成多个 batch
if (len(im_dim_list) % batch_size): # 如果测试图片的数量不能被 batch_size 整除,
    ↪ leftover=1

```

```

    leftover = 1
# 如果 batch_size 不等于 1, 则将一个 batch 的图片作为一个元素保存在 im_batches 中, 按照
↪ if 语句里面的公式计算。如果 batch_size=1, 则每一张图片作为一个元素保存在 im_batches
↪ 中
if batch_size != 1:
# 如果 batch_size 不等于 1, 则 batch 的数量 = 图片数量//batch_size + leftover(测试
↪ 图片的数量不能被 batch_size 整除, leftover=1, 否则为 0)。本例有 11 张图片, 假设
↪ batch_size=2, 则 batch 数量 =6
    num_batches = len(imlist) // batch_size + leftover
# 前面的 im_batches 变量将所有的图片以 BxCxHxW 的格式保存。而这里将一个 batch 的所有图片
↪ 在 B 这个维度 (第 0 维度) 上进行连接, torch.cat() 默认在 0 维上进行连接。将这个连接后
↪ 的 tensor 作为 im_batches 列表的一个元素。
# 第 i 个 batch 在前面的 im_batches 变量中所对应的元素就是 i*batch_size: (i +
↪ 1)*batch_size, 但是最后一个 batch 如果用 (i + 1)*batch_size 可能会超过图片数量的
↪ len(im_batches) 长度, 所以取 min((i + 1)*batch_size, len(im_batches))
↪
    im_batches = [torch.cat((im_batches[i*batch_size : min((i +
↪ 1)*batch_size,
                                len(im_batches))])) for i in range(num_batches)]

# The Detection Loop
write = 0

if CUDA:
    im_dim_list = im_dim_list.cuda()
# 开始计时, 计算开始检测的时间。start_det_loop - load_batch 为读入所有图片并将它们分成不
↪ 同 batch 的时间
start_det_loop = time.time()
# enumerate 返回 im_batches 列表中每个 batch 在 0 维连接成一个元素的 tensor 和这个
↪ tensor 在 im_batches 中的序号。本例子中 batch 只有一张图片
for i, batch in enumerate(im_batches):
#load the image
    start = time.time()
    if CUDA:
        batch = batch.cuda()
    # 取消梯度计算
    with torch.no_grad():
# Variable(batch) 将图片生成一个可导 tensor, 现在已经不再支持这种写法, Autograd
↪ automatically supports Tensors with requires_grad set to True.
# prediction 是一个 batch 所有图片通过 yolov3 模型得到的预测值, 维度为
↪ 1x10647x85, 三个 scale 的图片每个 scale 的特征图大小为 13x13, 26x26, 52x52,
↪ 一个元素看作一个格子, 每个格子有 3 个 anchor, 将一个 anchor 保存为一行,

```

```

# 所以 prediction 一共有 (13x13+26x26+52x52)x3=10647 行, 一个 anchor 预测
↪ (x,y,w,h,s,s_cls1,s_cls2...s_cls_80), 一共有 85 个元素。所以 prediction
↪ 的维度为 Bx10647x85, 加为这里 batch_size 为 1, 所以 prediction 的维度为
↪ 1x10647x85
prediction = model(Variable(batch), CUDA)
# 结果过滤. 这里返回了经过 NMS 后剩下的方框, 最终每个方框的属性为
↪ (ind,x1,y1,x2,y2,s,s_cls,index_cls) ind 是这个方框所属图片在这个 batch 中
↪ 的序号, x1,y1 是在网络输入图片 (416x416) 坐标系中, 方框左上角的坐标; x2,y2 是方
↪ 框右下角的坐标。
# s 是这个方框含有目标的得分, s_cls 是这个方框中所含目标最有可能的类别的概率得分,
↪ index_cls 是 s_cls 对应的这个类别在所有类别中所对应的序号。这里 prediction 维
↪ 度是 3x8, 表示有 3 个框
prediction = write_results(prediction, confidence, num_classes, nms_conf
↪ = nms_thesh)

end = time.time()
# 如果从 write_results() 返回的一个 batch 的结果是一个 int(0), 表示没有检测到目
↪ 标, 此时用 continue 跳过本次循环
if type(prediction) == int:
# 在 imlist 中, 遍历一个 batch 所有的图片对应的元素 (即每张图片的存储位置和名字), 同时
↪ 返回这张图片在这个 batch 中的序号 im_num
for im_num, image in enumerate(imlist[i*batch_size: min((i +
↪ 1)*batch_size, len(imlist))]):
    # 计算图片在 imlist 中所对应的序号, 即在所有图片中的序号
    im_id = i*batch_size + im_num
    # 打印图片运行的时间, 用一个 batch 的平均运行时间来表示。.3f 就表示
↪ 保留三位小数点的浮点
    print("{0:20s} predicted in {1:6.3f}
    ↪ seconds".format(image.split("/")[-1], (end -
    ↪ start)/batch_size))
    # 输出本次处理图片所有检测到的目标的名字
    print("{0:20s} {1:s}".format("Objects Detected:", ""))
    print("-----")
    continue
# prediction[:,0] 取出了每个方框在所在图片在这个 batch(第 i 个 batch) 中的序号,
↪ 加上 i*batch_size, 就将 prediction 中每个框 (一行) 的第一个元素 (维度 0) 变成了这个
↪ 框所在图片在 imlist 中的序号, 即在所有图片中的序号
prediction[:,0] += i*batch_size
# 这里用一个 write 标志来标记是否是第一次得到输出结果, 因为每次的结果要进行
↪ torch.cat() 操作, 而一个空的变量不能与 tensor 连接, 所以第一次将它赋值给
↪ output, 后面就直接进行 cat() 操作
if not write:                                     #If we have't initialised output
    output = prediction

```

```

        write = 1
    else:
        # output 将每个 batch 的输出结果在 0 维进行连接, 即在行维度上连接, 每行表示一个检测方
        ↪ 框的预测值。最终本例子中的 11 张图片检测得到的结果 output 维度为 34 x 8
        output = torch.cat((output, prediction))
        # 在 imlist 中, 遍历一个 batch 所有的图片对应的元素 (即每张图片的存储位置加名字), 同时
        ↪ 返回这张图片在这个 batch 中的序号 im_num
        for im_num, image in enumerate(imlist[i*batch_size: min((i +
        ↪ 1)*batch_size, len(imlist))]):
            im_id = i*batch_size + im_num# 计算图片在 imlist 中所对应的序号, 即在所有图
            ↪ 片中的序号
            # objs 列表包含了本次处理图片中所有检测得到的方框所包含目标的类别名称。每个元素对应
            ↪ 一个检测得到的方框所包含目标的类别名称。for x in output 遍历 output 中的每
            ↪ 一行 (即一个方框的预测值) 得到 x, 如果这个方
            # 框所在图片在所有图片中的序号等于本次处理图片的序号, 则用 classes[int(x[-1])]
            ↪ 找到这个方框包含目标类别在 classes 中对应的类的名字。
            objs = [classes[int(x[-1])]] for x in output if int(x[0]) == im_id#
            ↪ classes 在之前的语句 classes = load_classes("data/coco.names") 中就是为了把类
            ↪ 的序号转为字符名字
            print("{0:20s} predicted in {1:6.3f}"
            ↪ seconds".format(image.split("/")[-1], (end -
            ↪ start)/batch_size))# 打印本次处理图片运行的时间, 用一个 batch 的平均运行
            ↪ 时间来表示。.3f 就表示保留三位小数点的浮点
            print("{0:20s} {1:s}".format("Objects Detected:", " ".join(objs)))#
            ↪ 输出本次处理图片所有检测到的目标的类别名字
            print("-----")

    if CUDA:
        torch.cuda.synchronize() # 保证 gpu 和 cpu 同步, 否则, 一旦 GPU 工作排队了
        ↪ 并且 GPU 工作还远未完成, 那么 CUDA 核就将控制返回给 CPU (异步调用)。
    # 对所有的输入的检测结果
    try:
        # check whether there has been a single detection has been made or not
        output
    except NameError:
        print (" 没有检测到任何目标")
        exit() # 当所有图片都有没检测到目标时, 退出程序
    # 最后输出 output_recast - start_det_loop 计算的是从开始检测, 到去掉低分, NMS 操作的时
    ↪ 间。
    output_recast = time.time()
    # 前面 im_dim_list 是一个 4 维 tensor, 一行的元素为 (W,H,W,H), 对应一张图片原始的宽、高,
    ↪ 且重复了一次。(W,H,W,H) 主要是在后面计算 x1,y1,x2,y2 各自对应的缩放系数时好对应上。

```

```

# 本例中 im_dim_list 维度为 11x4.index_select() 就是在 im_dim_list 中查找 output
    ↳ 中每行所对应方框所在图片在所有图片中的序号对应 im_dim_list 中的那一行，最终得到的
    ↳ im_dim_list 的行数应该与 output 的行数相同。
# 因此这样做后本例中此时 im_dim_list 维度 34x4
im_dim_list = torch.index_select(im_dim_list, 0, output[:,0].long())#
    ↳ pytorch 切片 torch.index_select(data, dim, indices)
"""

应该将方框的坐标转换为相对于填充后的图片中包含原始图片区域的计算方式。min(416/im_dim_list,
    ↳ 1), 416 除以 im_dim_list 中的每个元素，然后在得到的 tensor 中的第 1 维（每行）去找
    ↳ 到最小的元素(torch.min()) 返回一个
有两个 tensor 元素的 tuple，第一个元素就是找到最小的元素的结果，这里没有给定 keepdim=True
    ↳ 的标记，所以得到的最小元素的 tensor 会比原来减小一维，
另一个是每个最小值在每行中对应的序号。torch.min(416/im_dim_list, 1)[0] 得到长度为 34
    ↳ 的最小元素构成的 tensor，通过 view(-1, 1)
变成了维度为 34x1 的 tensor。这个 tensor，即 scaling_factor 的每个元素就对应一张图片缩
    ↳ 放成 416 的时候所采用的缩放系数
注意了!!! Scaling_factor 在进行计算的时候用的 416，如果是其它的尺寸，这里不应该固定为
    ↳ 416，在开始检测时 util.py 里所用的缩放系数就是用的 min(w/img_w, h/img_h)

"""

#scaling_factor = torch.min(416/im_dim_list,1)[0].view(-1,1)# 这是源代码，下面
    ↳ 是我修改的代码
scaling_factor = torch.min(int(args.reso)/im_dim_list,1)[0].view(-1,1)
# 将相对于输入网络图片（416x416）的方框属性变换成原图按照纵横比不变进行缩放后的区域的坐标。
#scaling_factor*img_w 和 scaling_factor*img_h 是图片按照纵横比不变进行缩放后的图片，
    ↳ 即原图是 768x576 按照纵横比长边不变缩放到了 416x372。
# 经坐标换算，得到的坐标还是在输入网络的图片（416x416）坐标系下的绝对坐标，但是此时已经是相
    ↳ 对于 416x372 这个区域的坐标了，而不再相对于（0,0）原点。
output[:,[1,3]] -= (inp_dim - scaling_factor*im_dim_list[:,0].view(-
    ↳ 1,1))/2#x1=x1-(416-scaling_factor*img_w)/2,x2=x2-
    ↳ (416-scaling_factor*img_w)/2
output[:,[2,4]] -= (inp_dim -
    ↳ scaling_factor*im_dim_list[:,1].view(-1,1))/2#y1=y1-
    ↳ (416-scaling_factor*img_h)/2,y2=y2-(416-scaling_factor*img_h)/2

# 将方框坐标（x1,y1,x2,y2）映射到原始图片尺寸上，直接除以缩放系数即可。output[:,1:5] 维
    ↳ 度为 34x4，scaling_factor 维度是 34x1。相除时会利用广播性质将 scaling_factor 扩
    ↳ 展为 34x4 的 tensor
output[:,1:5] /= scaling_factor # 缩放至原图大小尺寸

```

```

# 如果映射回原始图片中的坐标超过了原始图片的区域, 则 x1,x2 限定在 [0,img_w] 内, img_w 为
→ 原始图片的宽度。如果 x1,x2 小于 0.0, 令 x1,x2 为 0.0, 如果 x1,x2 大于原始图片宽度,
→ 令 x1,x2 大小为图片的宽度。
# 同理, y1,y2 限定在 0,img_h] 内, img_h 为原始图片的高度。clamp() 函数就是将第一个输入对
→ 数的值限定在后面两个数字的区间
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])

class_load = time.time()# 开始载入颜色文件的时间
# 绘图
colors = pkl.load(open("pallete", "rb"))# 读入包含 100 个颜色的文件 pallete, 里面
→ 是 100 个三元组序列

draw = time.time() # 开始画方框的文字的时间

# x 为映射到原始图片中一个方框的属性 (ind,x1,y1,x2,y2,s,s_cls,index_cls), results
→ 列表保存了所有测试图片, 一个元素对应一张图片
def write(x, results):

    c1 = tuple(x[1:3].int())# c1 为方框左上角坐标 x1,y1
    c2 = tuple(x[3:5].int()) # c2 为方框右下角坐标 x2,y2
    img = results[int(x[0])]*# 在 results 中找到 x 方框所对应的图片, x[0] 为方框所在
→ 图片在所有测试图片中的序号
    cls = int(x[-1])
    color = random.choice(colors) # 随机选择一个颜色, 用于后面画方框的颜色
    label = "{0}".format(classes[cls])# label 为这个框所含目标类别名字的字符串
    cv2.rectangle(img, c1, c2,color, 1)# 在图片上画出 (x1,y1,x2,y2) 矩形, 即我们检
→ 测到的目标方框
    t_size = cv2.getTextSize(label, cv2.FONT_HERSHEY_PLAIN, 1 , 1)[0] # 得到
→ 一个包含目标名字字符的方框的宽高
    c2 = c1[0] + t_size[0] + 3, c1[1] + t_size[1] + 4 # 得到包含目标名字的方框
→ 右下角坐标 c2, 这里在 x,y 方向上分别加了 3、4 个像素
    cv2.rectangle(img, c1, c2,color, -1) # 在图片上画一个实心方框, 我们将会在方框内放置
→ 目标类别名字
    cv2.putText(img, label, (c1[0], c1[1] + t_size[1] + 4),
→ cv2.FONT_HERSHEY_PLAIN, 1, [225,255,255], 1); # 在图片上写文字, (c1[0],
→ c1[1] + t_size[1] + 4) 为字符串的左下角坐标
    return img

# 开始逐条绘制 output 中结果。将每个框在对应图片上画出来, 同时写上方框所含目标名字。map 函数
→ 将 output 传递给 map() 中参数是函数的那个参数, 每次传递一行。

```



```

# 而 lambda 中 x 就是 output 中的一行，维度为 1x8。loaded_ims 列表保存了所有图片内容数
    ↪ 组，一个元素对应一张图片，原地修改了 loaded_ims 之中的图像，使之还包含了目标类别名字。
list(map(lambda x: write(x, loaded_ims), output))
# 将带有方框的每张测试图片重新命名。det_names 是一个 series 对象，类似于一个列表，
    ↪ pd.Series(imlist) 返回一个 series 对象。
# 对于 imlist 这个列表（保存的是所有测试图片的绝对路径 + 名字，一个元素对应一张图片路径加名
    ↪ 字），生成的 series 对象包含两列，一列是每个 imlist 元素的索引，一列是 imlist 元素。
# apply() 函数将这个 series 对象传递给 apply() 里面的函数，以遍历的方式进行。apply() 返
    ↪ 回结果是经过 apply() 里面的函数返回每张测试图片将要保存的文件路径，这里依然是一个
    ↪ series 对象
# x 是 Series() 返回的对象中的一个元素，即一张图片的绝对路径加名字，args.det 是即将保存图片
    ↪ 的文件夹（默认 det），返回"det/det_ 图片名", x.split("/")[-1] 中的 "/" 是
    ↪ linux 下文件路径分隔符
det_names = pd.Series(imlist).apply(lambda x:
    ↪ "{}/{}/det_{}".format(args.det, x.split("/")[-1]))# 每张图像都以「det_」加上图像
    ↪ 名称的方式保存。我们创建了一个地址列表，这是我们保存我们的检测结果图像的位置。

list(map(cv2.imwrite, det_names, loaded_ims))# 保存标注了方框和目标类别名字的图片。
    ↪ det_names 对应所有测试图片的保存路径，loaded_ims 对应所有标注了方框和目标名字的图片数
    ↪ 组

end = time.time()

print("SUMMARY")
print("-----")
print("{:25s}: {}".format("Task", "Time Taken (in seconds)"))
print()
print("{:25s}: {:.2.3f}".format("Reading addresses", load_batch - read_dir))#
    ↪ 读取所有图片路径的时间
print("{:25s}: {:.2.3f}".format("Loading batch", start_det_loop -
    ↪ load_batch))# 读入所有图片，并将图片按照 batch size 分成不同 batch 的时间
# 从开始检测到去掉低分，NMS 操作得到 output 的时间。
print("{:25s}: {:.2.3f}".format("Detection (" + str(len(imlist)) + "
    ↪ images)", output_recast - start_det_loop))
# 这里 output 映射回原图的时间
print("{:25s}: {:.2.3f}".format("Output Processing", class_load -
    ↪ output_recast))
print("{:25s}: {:.2.3f}".format("Drawing Boxes", end - draw))# 画框和文字的时间
print("{:25s}: {:.2.3f}".format("Average time_per_img", (end -
    ↪ load_batch)/len(imlist)))# 从开始载入图片到所有结果处理完成，平均每张图片所消耗时间
print("-----")

```



```
torch.cuda.empty_cache()
```

第四部分: **video.py**

本篇介绍如何让检测器在视频或者网络摄像头实时工作。我们将引入一些命令行标签,以便能使用该网络的各种超参数进行一些实验。这个代码是 **video.py**, 代码整体上很像 **detect.py**, 只有几处变化, 只是我们不会在 **batch** 上迭代, 而是在视频的帧上迭代。

注意代码中有一处错误我进行了修改。源代码在计算 **scaling_factor** 时, 用的 **scaling_factor = torch.min(416/im_dim,1)[0].view(-1,1)** 显然不对, 应该使用用户输入的 **args.reso** 即改为 **scaling_factor = torch.min(int(args.reso)/im_dim,1)[0].view(-1,1)**

接下来就开始吧。

```
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():
```

```
    """
```

```
    视频检测模块的参数转换
```

```
    """
```

```
    # 创建一个 ArgumentParser 对象, 格式: 参数名, 目标参数 (dest 是字典的 key), 帮助信  
    ↪ 息, 默认值, 类型
```

```
    parser = argparse.ArgumentParser(description='YOLO v3 检测模型')
```

```
    parser.add_argument("--bs", dest = "bs", help = "Batch size, 默认为 1",  
    ↪ default = 1)
```

```
    parser.add_argument("--confidence", dest = "confidence", help = " 目标检测  
    ↪ 结果置信度阈值", default = 0.5)
```

```

    parser.add_argument("--nms_thresh", dest = "nms_thresh", help = "NMS 非极大值抑制阈值", default = 0.4)
    parser.add_argument("--cfg", dest = 'cfgfile', help = "配置文件", default = "cfg/yolov3.cfg", type = str)
    parser.add_argument("--weights", dest = 'weightsfile', help = "模型权重", default = "yolov3.weights", type = str)
    parser.add_argument("--reso", dest = 'reso', help = "网络输入分辨率。分辨率越高，则准确率越高；反之亦然", default = "416", type = str)
    parser.add_argument("--video", dest = "videofile", help = "待检测视频目录", default = "video.avi", type = str)

    return parser.parse_args()

```

```

args = arg_parse()# args 是一个 namespace 类型的变量，即 argparse.Namespace，可以像 easydict 一样使用，就像一个字典，key 来索引变量的值
# Namespace(bs=1, cfgfile='cfg/yolov3.cfg', confidence=0.5, det='det', images='imgs', nms_thresh=0.4, reso='416', weightsfile='yolov3.weights')
batch_size = int(args.bs)
confidence = float(args.confidence)
nms_thesh = float(args.nms_thresh)
start = 0
CUDA = torch.cuda.is_available()# GPU 环境是否可用

```

```

num_classes = 80# coco 数据集有 80 类
classes = load_classes("data/coco.names")# 将类别文件载入到我们的程序中，
# coco.names 文件中保存的是所有类别的名字，load_classes() 返回一个列表 classes，每个元素是一个类别的名字

```

```

# 初始化网络并载入权重
print(" 载入神经网络....")
model = Darknet(args.cfgfile)# Darknet 类中初始化时得到了网络结构和网络的参数信息，保存在 net_info, module_list 中
model.load_weights(args.weightsfile)# 将权重文件载入，并复制给对应的网络结构 model 中
print(" 模型加载成功.")
# 网络输入数据大小

```

```
model.net_info["height"] = args.reso# model 类中 net_info 是一个字典。'' height''
    ↳ 是图片的宽高，因为图片缩放到 416x416，所以宽高一样大
inp_dim = int(model.net_info["height"])#inp_dim 是网络输入图片尺寸（如 416*416）
assert inp_dim % 32 == 0 # 如果设定的输入图片的尺寸不是 32 的位数或者不大于 32，抛出异常
assert inp_dim > 32

# 如果 GPU 可用，模型切换到 cuda 中运行
if CUDA:
    model.cuda()

# 变成测试模式，这主要是对 dropout 和 batch normalization 的操作在训练和测试的时候是不
    ↳ 一样的
model.eval()

# 要在视频或网络摄像头运行这个检测器，代码基本可以保持不变，只是我们不会在 batch 上迭代，而
    ↳ 是在视频的帧上迭代。
# 将方框和文字写在图片上
def write(x, results):
    c1 = tuple(x[1:3].int())# c1 为方框左上角坐标 x1,y1
    c2 = tuple(x[3:5].int())# c2 为方框右下角坐标 x2,y2
    img = results
    cls = int(x[-1])
    color = random.choice(colors)# 随机选择一个颜色，用于后面画方框的颜色
    label = "{0}".format(classes[cls])#label 为这个框所含目标类别名字的字符串
    cv2.rectangle(img, c1, c2,color, 1)# 在图片上画出 (x1,y1,x2,y2) 矩形，即我们检
    ↳ 测到的目标方框
    t_size = cv2.getTextSize(label, cv2.FONT_HERSHEY_PLAIN, 1, 1)[0]# 得到一
    ↳ 个包含目标名字字符的方框的宽高
    c2 = c1[0] + t_size[0] + 3, c1[1] + t_size[1] + 4# 得到包含目标名字的方框右下
    ↳ 角坐标 c2，这里在 x,y 方向上分别加了 3、4 个像素
    cv2.rectangle(img, c1, c2,color, -1)# 在图片上画一个实心方框，我们将在方框内放置
    ↳ 目标类别名字
    cv2.putText(img, label, (c1[0], c1[1] + t_size[1] + 4),
    ↳ cv2.FONT_HERSHEY_PLAIN, 1, [225,255,255], 1);# 在图片上写文字，(c1[0],
    ↳ c1[1] + t_size[1] + 4) 为字符串的左下角坐标
    return img

#Detection phase

videofile = args.videofile #or path to the video file.
```

```

cap = cv2.VideoCapture(videofile) # 用 OpenCV 打开视频

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

# 当没有打开视频时抛出错误
assert cap.isOpened(), 'Cannot capture source'
# frames 用于统计图片的帧数
frames = 0
start = time.time()

fourcc = cv2.VideoWriter_fourcc('M','J','P','G')
fps = 24
savedPath = './det/savevideo.avi' # 保存的地址和视频名
ret, frame = cap.read()
videoWriter = cv2.VideoWriter(savedPath, fourcc, fps, (frame.shape[1],
    ↪ frame.shape[0])) # 最后为视频图片的形状

while cap.isOpened():# ret 指示是否读入了一张图片, 为 true 时读入了一帧图片
    ret, frame = cap.read()

    if ret:
        # 将图片按照比例缩放缩放, 将空白部分用 (128,128,128) 填充, 得到为 416x416 的图
        ↪ 片。并且将 HxWxC 转换为 CxHxW
        img = prep_image(frame, inp_dim)
        #cv2.imshow("a", frame)
        # 得到图片的 W,H, 是一个二元素 tuple. 因为我们不必再处理 batch, 而是一次只处理一
        ↪ 张图像, 所以很多地方的代码都进行了简化。
        # 因为一次只处理一帧, 故使用一个元组 im_dim 替代 im_dim_list 的张量。
        im_dim = frame.shape[1], frame.shape[0]
# 先将 im_dim 变成长度为 2 的一维行 tensor, 再在 1 维度 (列这个维度) 上复制一次, 变成
↪ 1x4 的二维行 tensor[W,H,W,H], 展开成 1x4 主要是在后面计算 x1,y1,x2,y2 各自对应的
↪ 缩放系数时好对应上。
        im_dim = torch.FloatTensor(im_dim).repeat(1,2)#repeat() 可能会改变
↪ tensor 的维度。它对 tensor 中对应 repeat 参数对应的维度上进行重复给定的次数, 如果
↪ tensor 的维度小于 repeat() 参数给定的维度, tensor 的维度将变成和 repeat() 一致。这
↪ 里 repeat(1,2), 表示在第一维度上重复一次, 第二维上重复两次, repeat(1,2) 有 2 个元素,
↪ 表示它给定的维度有 2 个, 所以将长度为 2 的一维行 tensor 变成了维度为 1x4 的二维
↪ tensor

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

```

```
# 只进行前向计算, 不计算梯度
with torch.no_grad():
# 得到每个预测方框在输入网络图片 (416x416) 坐标系中的坐标和宽高以及目标得分以及各个类别得分
    ↪ (x,y,w,h,s,s_cls1,s_cls2...)
# 并且将 tensor 的维度转换成 (batch_size, grid_size*grid_size*num_anchors, 5+ 类别数量)
    ↪ 别数量)
    output = model(Variable(img, volatile = True), CUDA)
# 将方框属性转换成 (ind,x1,y1,x2,y2,s,s_cls,index_cls), 去掉低分, NMS 等操作, 得到在输入网络坐标系中的最终预测结果
    ↪ 作, 得到在输入网络坐标系中的最终预测结果
    output = write_results(output, confidence, num_classes, nms_conf =
    ↪ nms_thesh)

# output 的正常输出类型为 float32, 如果没有检测到目标时 output 元素为 0, 此时为
    ↪ int 型, 将会用 continue 进行下一次检测
if type(output) == int:
# 每次迭代, 我们都会跟踪名为 frames 的变量中帧的数量。然后用这个数字除以自第一帧
    ↪ 以来过去的时间, 得到视频的帧率。
    frames += 1
    print("FPS of the video is {:.4f}".format( frames / (time.time()
    ↪ - start)))
# 我们不再使用 cv2.imwrite 将检测结果图像写入磁盘, 而是使用 cv2.imshow 展示画有
    ↪ 边界框的帧。
    cv2.imshow("frame", frame)
    key = cv2.waitKey(1)
# 如果用户按 Q 按钮, 就会让代码中断循环, 并且视频终止。
    if key & 0xFF == ord('q'):
        break
    continue

#im_dim 一行对应一个方框所在图片尺寸。在 detect.py 中一次测试多张图片, 所以对应的
    ↪ im_dim_list 是找到每个方框对应的图片的尺寸。
# 而这里每次只有一张图片, 每个方框所在图片的尺寸一样, 只需将图片的尺寸的行数重复方框的数量次数
    ↪ 即可
    im_dim = im_dim.repeat(output.size(0), 1)
# 得到每个方框所在图片缩放系数
#scaling_factor = torch.min(416/im_dim,1)[0].view(-1,1)# 这是源代码, 下
    ↪ 面是我修改的代码
    scaling_factor = torch.min(int(args.reso)/im_dim,1)[0].view(-1,1)
# 将方框的坐标 (x1,y1,x2,y2) 转换为相对于填充后的图片中包含原始图片区域 (如
    ↪ 416*312 区域) 的计算方式。
    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
        # 将坐标映射回原始图片
        output[:, 1:5] /= scaling_factor
        # 将超过了原始图片范围的方框坐标限定在图片范围之内
        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])

# coco.names 文件中保存的是所有类别的名字, load_classes() 返回一个列表
↪ classes, 每个元素是一个类别的名字
classes = load_classes('data/coco.names')
# 读入包含 100 个颜色的文件 pallete, 里面是 100 个三元组序列
colors = pkl.load(open("pallete", "rb"))
# 将每个方框的属性写在图片上
list(map(lambda x: write(x, frame), output))

cv2.imshow("frame", frame)

videoWriter.write(frame)                # 每次循环, 写入该帧
key = cv2.waitKey(1)
# 如果有按键输入则返回按键值编码, 输入 q 返回 113
if key & 0xFF == ord('q'):
    break
# 统计已经处理过的帧数
frames += 1
print(time.time() - start)
print("FPS of the video is {:.2f}".format( frames / (time.time() -
↪ start)))
else:
    videoWriter.release()                # 结束循环的时候释放
    break

```