所需环境： torch == 1.2.0

训练步骤

1、数据集使用VOC格式。将标签文件放在VOCdevkit文件夹下的VOC2007文件夹下的Annotation中。将图片文件放在VOCdevkit文件夹下的VOC2007文件夹下的JPEGImages中
2、在训练前利用voc2yolo4.py文件生成对应的txt。
3、再运行根目录下的voc_annotation.py，运行前需要将classes改成你自己的classes。注意不要使用中文标签，文件夹中不要有空格！

```
classes = ["aeroplane", "bicycle", "bird", "boat", "bottle", "bus", "car",
"cat", "chair", "cow", "diningtable", "dog", "horse", "motorbike",
"person", "pottedplant", "sheep", "sofa", "train", "tvmonitor"]
```

4、此时会生成对应的2007_train.txt，每一行对应其图片位置及其真实框的位置。
5、在训练前需要务必在model_data下新建一个txt文档，文档中输入需要分的类，在train.py中将classes_path指向该文件，示例如下：

```
classes_path = 'model_data/new_classes.txt'
```

model_data/new_classes.txt文件内容为：

```
cat
dog
...
```

6、修改utils/config.py里面的classes，使其为要检测的类的个数。
7、运行train.py即可开始训练。

训练时注意:

utils\config.py中的classes的数量需要修改,若使用聚类算法，先验框的尺寸也在此调整。
train.py中训练集和验证集的划分，关系到后面计算MAP。

使用自己的权重进行图片预测或摄像头预测:

1、在yolo.py文件里面，在如下部分修改model_path和classes_path使其对应训练好的文件；**model_path对应logs文件夹下面的权值文件，classes_path是model_path对应分的类**。

```
_defaults = {
    "model_path": 'model_data/yolo_weights.pth',
    "anchors_path": 'model_data/yolo_anchors.txt',
    "classes_path": 'model_data/coco_classes.txt,
    "score" : 0.5,
    "iou" : 0.3,
    # 显存比较小可以使用416x416
    # 显存比较大可以使用608x608
```

```
    "model_image_size" : (416, 416)
}
```

```
2、如进行图片预测：运行predict.py后，输入图片地址即可。
3、如进行视频预测：运行video.py。
```

计算MAP：

```
1、运行get_dr_txt.py，此时会在input\detection-results中生成预测结果的txt
文件。每个txt对应一个图片
2、运行get_gt_txt.py，此时会在input\ground-truth中生成真实标签的txt文件。
每个txt对应一个图片
3、运行get_map.py，此时会在results中生成所需的结果。
```

基础知识： 卷积后尺寸计算： 输入大小为w*w* 卷积核大小*f*f 步长s 填充像素数p 卷积后的尺寸为 n=(w-f+2p)/s+1

YOLOV3笔记： 1 首先将图像调整到416*416*的大小。为了防止图像失真（长宽比不是*1：1*的话），会将空白部分用灰色填充。 *2* 将图像分别分成*13*13、26*26、52*52的网格。不同尺度的网格用来检测不同尺寸的物体。 3 每个网格点负责右下角区域的预测，只要物体中心点落在这个区域里，这个物体就由这个网格来确定。

YOLOV3实现过程！！！ 注意：最后一个数是通道数，但在实际的代码中，通道数在batch_size后面的一个。 1 主干特征提取网络DarkNet-53 ship 1: iuput (batch_size, 416, 416, 3) ship 2: conv2D 32*33 (batch_size, 208, 80, 64) ship 3: Residual Block 1*64 (batch_size, 208, 280, 64) ship 4: Residual Block 2*128 (batch_size, 104, 104, 128) ship 5: Residual Block 8*256 (batch_size, 52, 52, 256) ->concat ship 6: Residual Block 8*512 (batch_size, 26, 26, 512) ->concat ship 7: Residual Block 4*1024 (batch_size, 13, 13, 1024) ->concat 2 特征金字塔 对ship 7的输出的特征图（13，13，1024）进行五次卷积，结果记为out 1。这个结果有两个走向。 走向1： 对out 1进行分类和回归预测，实际上是两次卷积，一次3*3的卷积，一次1*1的卷积。最后得到（13，13，75）->(13, 13 ,25*3) ->(13, 13, 3*(20+1+4))。3代表三个先验框，20代表20个类别的置信度，1代表是否有物体，4代表预测框的坐标。 走向2： 上采样，与ship 6的结果进行连接。连接的结果记为out 2。这个结果有两个走向。 对out 2进行预测（同上面走向1）得到（26，26，75） out 2再进行上采样，与ship 5的结果连接，再进行预测，得到（52, 52, 75）

主要部分代码的结构：

```
darknet.py：定义了主干特征提取网络DarkNet-53的结构,最后将主干网络存储在变量
model中。
yolo3.py：定义了特征金字塔部分的结构，将最后预测的结果保存out0 out1 out2
中。out0是最大尺度的结果，out2是最小尺度的结果。
utils/utils.py：解码，对先验框进行调整。
utils\config.py：原始先验框的设定，跟nets\yolo_training.py密切相关
predict.py: 对单张图片进行预测
nets\yolo_training.py：定义训练阶段的结构
train.py：启动训练，定义训练过程
yolo.py：定义预测阶段的结构

iou部分:utils\utils.py
LOSS部分：train.py
lr和Batch_size部分：train.py
```

下面是部分代码的注释

In [ ]:

```
#-----------------------------------#

##############darknet.py##############

######主干特征提取网络DarkNet-53######

#-----------------------------------#
import torch
import torch.nn as nn
import math
from collections import OrderedDict

# 基本的darknet块
class BasicBlock(nn.Module):
    def __init__(self, inplanes, planes):
        '''
        1*1卷积后再3*3卷积是为了减少参数。1*1卷积后通道数会下降，3*3后通道数又会上升
        '''
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(inplanes, planes[0], kernel_size=1,
                               stride=1, padding=0, bias=False) # 1*1卷积32通道 下降
        self.bn1 = nn.BatchNorm2d(planes[0]) # 标准化
        self.relu1 = nn.LeakyReLU(0.1) # 激活函数

        self.conv2 = nn.Conv2d(planes[0], planes[1], kernel_size=3,
                               stride=1, padding=1, bias=False) # 3*3卷积64通道 扩张
        self.bn2 = nn.BatchNorm2d(planes[1]) # 标准化
        self.relu2 = nn.LeakyReLU(0.1) # 激活函数

    def forward(self, x):
        # 残差块
        residual = x

        # 两组卷积+标准化+激活函数
        out = self.conv1(x) # 第一组
        out = self.bn1(out)
        out = self.relu1(out)

        out = self.conv2(out) # 第二组
        out = self.bn2(out)
        out = self.relu2(out)

        #将输出和残差边相加，这样就完成了一个前向传播的残差
        out += residual
        return out


class DarkNet(nn.Module):
    def __init__(self, layers):
        super(DarkNet, self).__init__()
        '''
        网络的初始化
        '''
        self.inplanes = 32 # 卷积的通道数
        self.conv1 = nn.Conv2d(3, self.inplanes, kernel_size=3, stride=1, padding=1,
        self.bn1 = nn.BatchNorm2d(self.inplanes) # nn.BatchNorm2d()函数是进行数据的归
        self.relu1 = nn.LeakyReLU(0.1) # 表示使用LeakyReLU激活函数,后面的参数表示x<0时

        self.layer1 = self._make_layer([32, 64], layers[0]) # 这里的_make_layer代表残
        self.layer2 = self._make_layer([64, 128], layers[1])
        self.layer3 = self._make_layer([128, 256], layers[2])
        self.layer4 = self._make_layer([256, 512], layers[3])
```

```
        self.layer5 = self._make_layer([512, 1024], layers[4])

        self.layers_out_filters = [64, 128, 256, 512, 1024]

        # 进行权值初始化
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()

    def _make_layer(self, planes, blocks):
        '''
        残差块
        planes: 通道数

        '''
        layers = []
        # 下采样，步长为2，卷积核大小为3
        layers.append(("ds_conv", nn.Conv2d(self.inplanes, planes[1], kernel_size=3,
                                stride=2, padding=1, bias=False))) # 卷积
        layers.append(("ds_bn", nn.BatchNorm2d(planes[1]))) # 标准化
        layers.append(("ds_relu", nn.LeakyReLU(0.1))) # 激活函数
        # 加入darknet模块
        self.inplanes = planes[1]
        for i in range(0, blocks): # block规定了堆叠残差块的循环次数，对应101行model
            layers.append(("residual_{}".format(i), BasicBlock(self.inplanes, planes)
        return nn.Sequential(OrderedDict(layers))

    def forward(self, x):
        x = self.conv1(x) # 88 89 90这三行是主干网络中第一个卷积块的卷积、标准化和激活
        x = self.bn1(x)
        x = self.relu1(x)

        x = self.layer1(x) # 第一个残差块
        x = self.layer2(x) # 第二个残差块
        out3 = self.layer3(x) # 特征金字塔的一个输出 52*52*256
        out4 = self.layer4(out3) # 特征金字塔的一个输出 26*26*512
        out5 = self.layer5(out4) # 特征金字塔的一个输出 13*13*1024

        return out3, out4, out5

def darknet53(pretrained, **kwargs):
    model = DarkNet([1, 2, 8, 8, 4]) # 这里的1，2，8，8，4对应的是主干网络中残差块的使
    if pretrained: # 载入预训练
        if isinstance(pretrained, str):
            model.load_state_dict(torch.load(pretrained))
        else:
            raise Exception("darknet request a pretrained path. got [{}]".format(pret
    return model
```

In [ ]:

```
#-------------------------------------#

##############yolo3.py###############

############特征金字塔部分############

#-------------------------------------#
import torch
import torch.nn as nn
from collections import OrderedDict
from nets.darknet import darknet53
```

```
def conv2d(filter_in, filter_out, kernel_size):
    '''
    定义一个卷积块，包括一次卷积，一次标准化和一个激活函数
    '''
    pad = (kernel_size - 1) // 2 if kernel_size else 0
    return nn.Sequential(OrderedDict([
        ("conv", nn.Conv2d(filter_in, filter_out, kernel_size=kernel_size, stride=1, 
        ("bn", nn.BatchNorm2d(filter_out)),
        ("relu", nn.LeakyReLU(0.1)),
    ]))

def make_last_layers(filters_list, in_filters, out_filter):
    '''
    最后的那七次卷积
    '''
    m = nn.ModuleList([
        conv2d(in_filters, filters_list[0], 1), # 1*1卷积调整通道数
        conv2d(filters_list[0], filters_list[1], 3), # 3*3卷积提取特征
        conv2d(filters_list[1], filters_list[0], 1), # 1*1卷积调整通道数
        conv2d(filters_list[0], filters_list[1], 3), # 3*3卷积提取特征
        conv2d(filters_list[1], filters_list[0], 1), # 1*1卷积调整通道数
        conv2d(filters_list[0], filters_list[1], 3), # 下面这两个卷积是分类预测和回归
        nn.Conv2d(filters_list[1], out_filter, kernel_size=1,
                                        stride=1, padding=0, bias=True)
    ])
    return m

class YoloBody(nn.Module):
    def __init__(self, config):
        super(YoloBody, self).__init__()
        self.config = config
        #  backbone
        self.backbone = darknet53(None) # 将darknet.py中获得的主干网络的结构保存在.ba

        out_filters = self.backbone.layers_out_filters
        #  last_layer0 3*(5+num_classes)=3*(5+20)=3*(4+1+20)=75 这部分是处理out5的特
        final_out_filter0 = len(config["yolo"]["anchors"][0]) * (5 + config["yolo"]["
        self.last_layer0 = make_last_layers([512, 1024], out_filters[-1], final_out_f

        #  embedding1 75 这部分是处理out4的特征层
        final_out_filter1 = len(config["yolo"]["anchors"][1]) * (5 + config["yolo"]["
        self.last_layer1_conv = conv2d(512, 256, 1) # 用1*1的卷积调整通道数
        self.last_layer1_upsample = nn.Upsample(scale_factor=2, mode='nearest') # 第
        # 此处已经获得26,26,256的特征层
        self.last_layer1 = make_last_layers([256, 512], out_filters[-2] + 256, final_

        #  embedding2 75 这部分是处理out3的特征层
        final_out_filter2 = len(config["yolo"]["anchors"][2]) * (5 + config["yolo"]["
        self.last_layer2_conv = conv2d(256, 128, 1) # 1*1卷积调整通道数
        self.last_layer2_upsample = nn.Upsample(scale_factor=2, mode='nearest') # 第
        # 此处已经获得52,52,128的特征层
        self.last_layer2 = make_last_layers([128, 256], out_filters[-3] + 128, final_


    def forward(self, x):
        def _branch(last_layer, layer_in): # 因为特征金字塔那七次卷积是在一起的，但结
            for i, e in enumerate(last_layer):
                layer_in = e(layer_in)
                if i == 4:
                    out_branch = layer_in # 将特征金字塔部分那五次卷积的结果保存在out_
            return layer_in, out_branch # 将特征金字塔部分回归预测和分类预测的结果保
        #  backbone
        x2, x1, x0 = self.backbone(x) # 获取主干特征提取网络
        '''
```

```
        x2: Out3对应的特征层
        x1: Out4对应的特征层
        x0: Out5对应的特征层
        '''
        #  yolo branch 0
        out0, out0_branch = _branch(self.last_layer0, x0) # 将五次卷积和最后两次卷积结

        #  yolo branch 1
        x1_in = self.last_layer1_conv(out0_branch) # 1*1卷积调整通道数
        x1_in = self.last_layer1_upsample(x1_in) # 上采样
        x1_in = torch.cat([x1_in, x1], 1) # 不同尺度的特征层进行堆叠
        out1, out1_branch = _branch(self.last_layer1, x1_in) # 将五次卷积和最后两次卷

        #  yolo branch 2
        x2_in = self.last_layer2_conv(out1_branch) # 1*1卷积调整通道数
        x2_in = self.last_layer2_upsample(x2_in) # 上采样
        x2_in = torch.cat([x2_in, x2], 1) # 不同尺度的特征层进行堆叠
        out2, _ = _branch(self.last_layer2, x2_in) # 将五次卷积和最后两次卷积结果分开
        return out0, out1, out2
```

In [ ]:

```
#------------------------------------#

###########utils\utils.py############

###########解码 这里使用CIOU##########

#------------------------------------#
from __future__ import division
import os
import math
import time
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import numpy as np

from PIL import Image, ImageDraw, ImageFont

class DecodeBox(nn.Module):
    def __init__(self, anchors, num_classes, img_size):
        super(DecodeBox, self).__init__()
        self.anchors = anchors
        self.num_anchors = len(anchors)
        self.num_classes = num_classes
        self.bbox_attrs = 5 + num_classes
        self.img_size = img_size

    def forward(self, input):
        # 注释以13*13*75的特征层为例
        # input为batch_size,3*(1+4+num_classes),13,13  中间的3代表三个先验框，1代表是
        # 解码主要是计算上面那个4
        batch_size = input.size(0) # 图片的数量
        input_height = input.size(2) # 特征层的宽
        input_width = input.size(3) # 特征层的高

        # 计算步长，也就是特征层上的特征点对应原图上多少个像素
        stride_h = self.img_size[1] / input_height # 416/13=32
        stride_w = self.img_size[0] / input_width # 416/13=32
        # 归一到特征层上
        # 先把先验框的尺寸调整到特征层对应的大小
        # 计算出先验框在特征层上对应的宽高
        scaled_anchors = [(anchor_width / stride_w, anchor_height / stride_h) for anc
```

```
        # 对预测结果进行resize，调整了几个参数的顺序。batch_size,3*(5+num_classes),13,
        prediction = input.view(batch_size, self.num_anchors,
                                self.bbox_attrs, input_height, input_width).permute(0,

        # 先验框的中心位置的调整参数
        x = torch.sigmoid(prediction[..., 0]) # sigmoid将中心位置调整到0和1之间
        y = torch.sigmoid(prediction[..., 1]) # sigmoid将中心位置调整到0和1之间 这样i
        # 先验框的宽高调整参数
        w = prediction[..., 2]  # Width
        h = prediction[..., 3]  # Height

        # 获得置信度，是否有物体
        conf = torch.sigmoid(prediction[..., 4])
        # 种类置信度
        pred_cls = torch.sigmoid(prediction[..., 5:])  # Cls pred.

        FloatTensor = torch.cuda.FloatTensor if x.is_cuda else torch.FloatTensor
        LongTensor = torch.cuda.LongTensor if x.is_cuda else torch.LongTensor

        # 生成网格，先验框中心，网格左上角 batch_size,3,13,13
        grid_x = torch.linspace(0, input_width - 1, input_width).repeat(input_width,
            batch_size * self.num_anchors, 1, 1).view(x.shape).type(FloatTensor)
        grid_y = torch.linspace(0, input_height - 1, input_height).repeat(input_heigh
            batch_size * self.num_anchors, 1, 1).view(y.shape).type(FloatTensor)

        # 生成先验框的宽高
        anchor_w = FloatTensor(scaled_anchors).index_select(1, LongTensor([0]))
        anchor_h = FloatTensor(scaled_anchors).index_select(1, LongTensor([1]))
        anchor_w = anchor_w.repeat(batch_size, 1).repeat(1, 1, input_height * input_w
        anchor_h = anchor_h.repeat(batch_size, 1).repeat(1, 1, input_height * input_w

        # 计算调整后的先验框中心与宽高
        pred_boxes = FloatTensor(prediction[..., :4].shape)
        pred_boxes[..., 0] = x.data + grid_x # 调整先验框的中心，x的调整参数直接加上对
        pred_boxes[..., 1] = y.data + grid_y # 调整先验框的中心，y的调整参数直接加上对
        pred_boxes[..., 2] = torch.exp(w.data) * anchor_w # 宽的调整系数取对数乘以先验
        pred_boxes[..., 3] = torch.exp(h.data) * anchor_h # 高的调整系数取对数乘以先验

        # 用于将输出调整为相对于416x416的大小
        _scale = torch.Tensor([stride_w, stride_h] * 2).type(FloatTensor)
        output = torch.cat((pred_boxes.view(batch_size, -1, 4) * _scale,
                            conf.view(batch_size, -1, 1), pred_cls.view(batch_size, -
        return output.data

def letterbox_image(image, size):
    '''
    算法要求图片为正方形。如416*416。如果是长方形的图片，需要将图片固定到这个大小
    为了不让长方形的图片失真，则需要添加灰条。
    '''
    iw, ih = image.size # 获取图片像素的宽和高
    w, h = size # 获取网络要求的像素大小
    scale = min(w/iw, h/ih) # 计算缩放比例
    nw = int(iw*scale)
    nh = int(ih*scale)

    image = image.resize((nw,nh), Image.BICUBIC) # 调整图片尺寸
    new_image = Image.new('RGB', size, (128,128,128)) # 新图像中添加灰条
    new_image.paste(image, ((w-nw)//2, (h-nh)//2)) # 将RESIZE后的图片贴在新图片对应的
    return new_image

def yolo_correct_boxes(top, left, bottom, right, input_shape, image_shape):
    '''
    目前框框的位置是相对于有灰条图片左上角的位置。去掉灰条要转换为原图的左上角的位置。
    '''
    new_shape = image_shape*np.min(input_shape/image_shape)
```

```
    offset = (input_shape-new_shape)/2./input_shape
    scale = input_shape/new_shape

    box_yx = np.concatenate(((top+bottom)/2,(left+right)/2),axis=-1)/input_shape
    box_hw = np.concatenate((bottom-top,right-left),axis=-1)/input_shape

    box_yx = (box_yx - offset) * scale
    box_hw *= scale

    box_mins = box_yx - (box_hw / 2.)
    box_maxes = box_yx + (box_hw / 2.)
    boxes =  np.concatenate([
        box_mins[:, 0:1],
        box_mins[:, 1:2],
        box_maxes[:, 0:1],
        box_maxes[:, 1:2]
    ],axis=-1)
    print(np.shape(boxes))
    boxes *= np.concatenate([image_shape, image_shape],axis=-1)
    return boxes

def bbox_iou(b1, b2, x1y1x2y2=True):
    """
        计算IOU
    """
# 求出预测框左上角右下角
    b1_xy = b1[..., :2]
    b1_wh = b1[..., 2:4]
    b1_wh_half = b1_wh/2.
    b1_mins = b1_xy - b1_wh_half
    b1_maxes = b1_xy + b1_wh_half

    # 求出真实框左上角右下角
    b2_xy = b2[..., :2]
    b2_wh = b2[..., 2:4]
    b2_wh_half = b2_wh/2.
    b2_mins = b2_xy - b2_wh_half
    b2_maxes = b2_xy + b2_wh_half

    # 求真实框和预测框所有的iou
    intersect_mins = torch.max(b1_mins, b2_mins)
    intersect_maxes = torch.min(b1_maxes, b2_maxes)
    intersect_wh = torch.max(intersect_maxes - intersect_mins, torch.zeros_like(inter
    intersect_area = intersect_wh[..., 0] * intersect_wh[..., 1]
    b1_area = b1_wh[..., 0] * b1_wh[..., 1]
    b2_area = b2_wh[..., 0] * b2_wh[..., 1]
    union_area = b1_area + b2_area - intersect_area
    iou = intersect_area / torch.clamp(union_area,min = 1e-6)

    # 计算中心的差距
    center_distance = torch.sum(torch.pow((b1_xy - b2_xy), 2), axis=-1)

    # 找到包裹两个框的最小框的左上角和右下角
    enclose_mins = torch.min(b1_mins, b2_mins)
    enclose_maxes = torch.max(b1_maxes, b2_maxes)
    enclose_wh = torch.max(enclose_maxes - enclose_mins, torch.zeros_like(intersect_m

    # 计算对角线距离
    enclose_diagonal = torch.sum(torch.pow(enclose_wh,2), axis=-1)
    ciou = iou - 1.0 * (center_distance) / torch.clamp(enclose_diagonal,min = 1e-6)

    v = (4 / (math.pi ** 2)) * torch.pow((torch.atan(b1_wh[..., 0]/torch.clamp(b1_wh
    alpha = v / torch.clamp((1.0 - iou + v),min=1e-6)
    iou = ciou - alpha * v
```

```
    return iou


def non_max_suppression(prediction, num_classes, conf_thres=0.5, nms_thres=0.4):
    '''
    非极大抑制
    '''
    # 解码时框框的位置是中心加宽高组成的，现在转化为求左上角和右下角
    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]

    output = [None for _ in range(len(prediction))]
    for image_i, image_pred in enumerate(prediction): # 对这组要检测的图片进行循环
        # 利用置信度进行第一轮筛选
        conf_mask = (image_pred[:, 4] >= conf_thres).squeeze()
        image_pred = image_pred[conf_mask]

        if not image_pred.size(0):
            continue

        # 获得种类及其置信度
        class_conf, class_pred = torch.max(image_pred[:, 5:5 + num_classes], 1, keepd

        # 获得的内容为(x1, y1, x2, y2, obj_conf, class_conf, class_pred)
        detections = torch.cat((image_pred[:, :5], class_conf.float(), class_pred.flo

        # 获得种类
        unique_labels = detections[:, -1].cpu().unique()

        if prediction.is_cuda:
            unique_labels = unique_labels.cuda()

        for c in unique_labels: # 进行遍历 完成非极大抑制的操作
            # 获得某一类初步筛选后全部的预测结果
            detections_class = detections[detections[:, -1] == c]
            # 按照存在物体的置信度排序
            _, conf_sort_index = torch.sort(detections_class[:, 4], descending=True)
            detections_class = detections_class[conf_sort_index]
            # 进行非极大抑制
            max_detections = []
            while detections_class.size(0):
                # 取出这一类置信度最高的，一步一步往下判断，判断重合程度是否大于nms_th
                max_detections.append(detections_class[0].unsqueeze(0))
                if len(detections_class) == 1:
                    break
                ious = bbox_iou(max_detections[-1], detections_class[1:]) # IOU计算
                detections_class = detections_class[1:][ious < nms_thres]
            # 堆叠
            max_detections = torch.cat(max_detections).data
            # Add max detections to outputs
            output[image_i] = max_detections if output[image_i] is None else torch.c
                (output[image_i], max_detections))

    return output
```

In [ ]:

```
#----------------------------------#

########nets\yolo_training.py########

##########定义训练阶段的结构##########
```

```
#-------------------------------------#

import cv2
from random import shuffle
import numpy as np
import torch
import torch.nn as nn
import math
import torch.nn.functional as F
from matplotlib.colors import rgb_to_hsv, hsv_to_rgb
from PIL import Image
from utils.utils import bbox_iou
from new_code import focal_loss2 as focal

def jaccard(_box_a, _box_b):
    b1_x1, b1_x2 = _box_a[:, 0] - _box_a[:, 2] / 2, _box_a[:, 0] + _box_a[:, 2] / 2
    b1_y1, b1_y2 = _box_a[:, 1] - _box_a[:, 3] / 2, _box_a[:, 1] + _box_a[:, 3] / 2
    b2_x1, b2_x2 = _box_b[:, 0] - _box_b[:, 2] / 2, _box_b[:, 0] + _box_b[:, 2] / 2
    b2_y1, b2_y2 = _box_b[:, 1] - _box_b[:, 3] / 2, _box_b[:, 1] + _box_b[:, 3] / 2
    box_a = torch.zeros_like(_box_a)
    box_b = torch.zeros_like(_box_b)
    box_a[:, 0], box_a[:, 1], box_a[:, 2], box_a[:, 3] = b1_x1, b1_y1, b1_x2, b1_y2
    box_b[:, 0], box_b[:, 1], box_b[:, 2], box_b[:, 3] = b2_x1, b2_y1, b2_x2, b2_y2
    A = box_a.size(0)
    B = box_b.size(0)
    max_xy = torch.min(box_a[:, 2:].unsqueeze(1).expand(A, B, 2),
                       box_b[:, 2:].unsqueeze(0).expand(A, B, 2))
    min_xy = torch.max(box_a[:, :2].unsqueeze(1).expand(A, B, 2),
                       box_b[:, :2].unsqueeze(0).expand(A, B, 2))
    inter = torch.clamp((max_xy - min_xy), min=0)

    inter = inter[:, :, 0] * inter[:, :, 1]
    # 计算先验框和真实框各自的面积
    area_a = ((box_a[:, 2]-box_a[:, 0]) *
              (box_a[:, 3]-box_a[:, 1])).unsqueeze(1).expand_as(inter)  # [A,B]
    area_b = ((box_b[:, 2]-box_b[:, 0]) *
              (box_b[:, 3]-box_b[:, 1])).unsqueeze(0).expand_as(inter)  # [A,B]
    # 求IOU
    union = area_a + area_b - inter
    return inter / union  # [A,B]

def clip_by_tensor(t,t_min,t_max):
    t=t.float()

    result = (t >= t_min).float() * t + (t < t_min).float() * t_min
    result = (result <= t_max).float() * result + (result > t_max).float() * t_max
    return result

def MSELoss(pred,target):
    return (pred-target)**2

def BCELoss(pred,target):
    epsilon = 1e-7
    pred = clip_by_tensor(pred, epsilon, 1.0 - epsilon)
    output = -target * torch.log(pred) - (1.0 - target) * torch.log(1.0 - pred)
    return output

class YOLOLoss(nn.Module):
    '''
    定义损失函数
    '''
    def __init__(self, anchors, num_classes, img_size, cuda):
        super(YOLOLoss, self).__init__()
        self.anchors = anchors
```

```
        self.num_anchors = len(anchors)
        self.num_classes = num_classes
        self.bbox_attrs = 5 + num_classes
        self.feature_length = [img_size[0]//32,img_size[0]//16,img_size[0]//8]
        self.img_size = img_size

        self.ignore_threshold = 0.5 # 与gt的iou超过ignore_threshold则被忽略，低于igno
        self.lambda_xy = 1.0 # 下面的四个参数是各LOSS的权重
        self.lambda_wh = 1.0
        self.lambda_conf = 1.0
        self.lambda_cls = 1.0
        self.cuda = cuda

    def forward(self, input, targets=None):
        # input为bs,3*(5+num_classes),13,13

        # 一共多少张图片
        bs = input.size(0)
        # 特征层的高
        in_h = input.size(2)
        # 特征层的宽
        in_w = input.size(3)

        # 计算步长
        # 每一个特征点对应原来的图片上多少个像素点
        # 如果特征层为13x13的话，一个特征点就对应原来的图片上的32个像素点
        stride_h = self.img_size[1] / in_h
        stride_w = self.img_size[0] / in_w

        # 把先验框的尺寸调整成特征层大小的形式
        # 计算出先验框在特征层上对应的宽高
        scaled_anchors = [(a_w / stride_w, a_h / stride_h) for a_w, a_h in self.anch

        # bs,3*(5+num_classes),13,13 -> bs,3,13,13,(5+num_classes)
        prediction = input.view(bs, int(self.num_anchors/3),
                                self.bbox_attrs, in_h, in_w).permute(0, 1, 3, 4, 2).c

        # 对prediction预测进行调整
        x = torch.sigmoid(prediction[..., 0])  # Center x
        y = torch.sigmoid(prediction[..., 1])  # Center y
        w = prediction[..., 2]  # Width
        h = prediction[..., 3]  # Height
        conf = torch.sigmoid(prediction[..., 4])  # Conf
        pred_cls = torch.sigmoid(prediction[..., 5:])  # Cls pred.

        # 找到哪些先验框内部包含物体
        mask, noobj_mask, tx, ty, tw, th, tconf, tcls, box_loss_scale_x, box_loss_scal
                                                                          self.get_t



        noobj_mask = self.get_ignore(prediction, targets, scaled_anchors, in_w, in_h,
        if self.cuda:
            box_loss_scale_x = (box_loss_scale_x).cuda()
            box_loss_scale_y = (box_loss_scale_y).cuda()
            mask, noobj_mask = mask.cuda(), noobj_mask.cuda()
            tx, ty, tw, th = tx.cuda(), ty.cuda(), tw.cuda(), th.cuda()
            tconf, tcls = tconf.cuda(), tcls.cuda()
        box_loss_scale = 2 - box_loss_scale_x*box_loss_scale_y

        #  losses.
        loss_x = torch.sum(BCELoss(x, tx) / bs * box_loss_scale * mask) # 先验框中心的
        loss_y = torch.sum(BCELoss(y, ty) / bs * box_loss_scale * mask)
        loss_w = torch.sum(MSELoss(w, tw) / bs * 0.5 * box_loss_scale * mask) # 先验框
        loss_h = torch.sum(MSELoss(h, th) / bs * 0.5 * box_loss_scale * mask)
```

```
        loss_conf = torch.sum(BCELoss(conf, mask) * mask / bs) + \
                    torch.sum(BCELoss(conf, mask) * noobj_mask / bs) # 先验框置信度LO

        loss_cls = torch.sum(BCELoss(pred_cls[mask == 1], tcls[mask == 1])/bs) # 类别

        loss = loss_x * self.lambda_xy + loss_y * self.lambda_xy + \
                loss_w * self.lambda_wh + loss_h * self.lambda_wh + \
                loss_conf * self.lambda_conf + loss_cls * self.lambda_cls # 将LOSS进行
        # print(loss, loss_x.item() + loss_y.item(), loss_w.item() + loss_h.item(),
        #         loss_conf.item(), loss_cls.item(), \
        #         torch.sum(mask),torch.sum(noobj_mask))
        return loss, loss_x.item(), loss_y.item(), loss_w.item(), \
                loss_h.item(), loss_conf.item(), loss_cls.item()

    def get_target(self, target, anchors, in_w, in_h, ignore_threshold):
        # 计算一共有多少张图片
        bs = len(target)
        # 获得先验框
        anchor_index = [[0,1,2],[3,4,5],[6,7,8]][self.feature_length.index(in_w)]
        subtract_index = [0,3,6][self.feature_length.index(in_w)]
        # 创建全是0或者全是1的阵列
        mask = torch.zeros(bs, int(self.num_anchors/3), in_h, in_w, requires_grad=Fal
        noobj_mask = torch.ones(bs, int(self.num_anchors/3), in_h, in_w, requires_gra

        tx = torch.zeros(bs, int(self.num_anchors/3), in_h, in_w, requires_grad=Fals
        ty = torch.zeros(bs, int(self.num_anchors/3), in_h, in_w, requires_grad=Fals
        tw = torch.zeros(bs, int(self.num_anchors/3), in_h, in_w, requires_grad=Fals
        th = torch.zeros(bs, int(self.num_anchors/3), in_h, in_w, requires_grad=Fals
        tconf = torch.zeros(bs, int(self.num_anchors/3), in_h, in_w, requires_grad=Fa
        tcls = torch.zeros(bs, int(self.num_anchors/3), in_h, in_w, self.num_classes,

        box_loss_scale_x = torch.zeros(bs, int(self.num_anchors/3), in_h, in_w, requi
        box_loss_scale_y = torch.zeros(bs, int(self.num_anchors/3), in_h, in_w, requi
        for b in range(bs):
            for t in range(target[b].shape[0]):
                # 计算出在特征层上的点位
                gx = target[b][t, 0] * in_w
                gy = target[b][t, 1] * in_h

                gw = target[b][t, 2] * in_w
                gh = target[b][t, 3] * in_h

                # 计算出属于哪个网格
                gi = int(gx)
                gj = int(gy)

                # 计算真实框的位置
                gt_box = torch.FloatTensor(np.array([0, 0, gw, gh])).unsqueeze(0)

                # 计算出所有先验框的位置
                anchor_shapes = torch.FloatTensor(np.concatenate((np.zeros((self.num_
                                                                  np.array(anchors)),
                # 计算重合程度
                anch_ious = bbox_iou(gt_box, anchor_shapes)

                # 判断找到的先验框是否属于这个特征层
                best_n = np.argmax(anch_ious)
                if best_n not in anchor_index:
                    continue
                # 编码
                if (gj < in_h) and (gi < in_w):
                    best_n = best_n - subtract_index
                    # 判定哪些先验框内部真实的存在物体
                    noobj_mask[b, best_n, gj, gi] = 0
```

```
                    mask[b, best_n, gj, gi] = 1
                    # 计算先验框中心调整参数
                    tx[b, best_n, gj, gi] = gx - gi
                    ty[b, best_n, gj, gi] = gy - gj
                    # 计算先验框宽高调整参数
                    tw[b, best_n, gj, gi] = math.log(gw / anchors[best_n+subtract_ind
                    th[b, best_n, gj, gi] = math.log(gh / anchors[best_n+subtract_ind
                    # 用于获得xywh的比例
                    box_loss_scale_x[b, best_n, gj, gi] = target[b][t, 2]
                    box_loss_scale_y[b, best_n, gj, gi] = target[b][t, 3]
                    # 物体置信度
                    tconf[b, best_n, gj, gi] = 1
                    # 种类
                    tcls[b, best_n, gj, gi, int(target[b][t, 4])] = 1
                else:
                    print('Step {0} out of bound'.format(b))
                    print('gj: {0}, height: {1} | gi: {2}, width: {3}'.format(gj, in_h
                    continue

        return mask, noobj_mask, tx, ty, tw, th, tconf, tcls, box_loss_scale_x, box_l

    def get_ignore(self,prediction,target,scaled_anchors,in_w, in_h,noobj_mask):
        bs = len(target)
        anchor_index = [[0,1,2],[3,4,5],[6,7,8]][self.feature_length.index(in_w)]
        scaled_anchors = np.array(scaled_anchors)[anchor_index]
        # print(scaled_anchors)
        # 先验框的中心位置的调整参数
        x = torch.sigmoid(prediction[..., 0])
        y = torch.sigmoid(prediction[..., 1])
        # 先验框的宽高调整参数
        w = prediction[..., 2]  # Width
        h = prediction[..., 3]  # Height

        FloatTensor = torch.cuda.FloatTensor if x.is_cuda else torch.FloatTensor
        LongTensor = torch.cuda.LongTensor if x.is_cuda else torch.LongTensor

        # 生成网格，先验框中心，网格左上角
        grid_x = torch.linspace(0, in_w - 1, in_w).repeat(in_w, 1).repeat(
            int(bs*self.num_anchors/3), 1, 1).view(x.shape).type(FloatTensor)
        grid_y = torch.linspace(0, in_h - 1, in_h).repeat(in_h, 1).t().repeat(
            int(bs*self.num_anchors/3), 1, 1).view(y.shape).type(FloatTensor)

        # 生成先验框的宽高
        anchor_w = FloatTensor(scaled_anchors).index_select(1, LongTensor([0]))
        anchor_h = FloatTensor(scaled_anchors).index_select(1, LongTensor([1]))

        anchor_w = anchor_w.repeat(bs, 1).repeat(1, 1, in_h * in_w).view(w.shape)
        anchor_h = anchor_h.repeat(bs, 1).repeat(1, 1, in_h * in_w).view(h.shape)

        # 计算调整后的先验框中心与宽高
        pred_boxes = FloatTensor(prediction[..., :4].shape)
        pred_boxes[..., 0] = x.data + grid_x
        pred_boxes[..., 1] = y.data + grid_y
        pred_boxes[..., 2] = torch.exp(w.data) * anchor_w
        pred_boxes[..., 3] = torch.exp(h.data) * anchor_h

        for i in range(bs):
            pred_boxes_for_ignore = pred_boxes[i]
            pred_boxes_for_ignore = pred_boxes_for_ignore.view(-1, 4)

            if len(target[i]) > 0:
                gx = target[i][:, 0:1] * in_w
                gy = target[i][:, 1:2] * in_h
                gw = target[i][:, 2:3] * in_w
                gh = target[i][:, 3:4] * in_h
```

```
                gt_box = torch.FloatTensor(np.concatenate([gx, gy, gw, gh],-1)).type(

                anch_ious = jaccard(gt_box, pred_boxes_for_ignore)
                for t in range(target[i].shape[0]):
                    anch_iou = anch_ious[t].view(pred_boxes[i].size()[:3])
                    noobj_mask[i][anch_iou>self.ignore_threshold] = 0
                # print(torch.max(anch_ious))
        return noobj_mask


def rand(a=0, b=1):
    return np.random.rand()*(b-a) + a


class Generator(object):
    def __init__(self,batch_size,
                 train_lines, image_size,
                 ):

        self.batch_size = batch_size
        self.train_lines = train_lines
        self.train_batches = len(train_lines)
        self.image_size = image_size

    def get_random_data(self, annotation_line, input_shape, jitter=.1, hue=.1, sat=1.
        '''r实时数据增强的随机预处理'''
        line = annotation_line.split()
        image = Image.open(line[0])
        iw, ih = image.size
        h, w = input_shape
        box = np.array([np.array(list(map(int,box.split(',')))) for box in line[1:]])

        # resize image
        new_ar = w/h * rand(1-jitter,1+jitter)/rand(1-jitter,1+jitter)
        scale = rand(.25, 2)
        if new_ar < 1:
            nh = int(scale*h)
            nw = int(nh*new_ar)
        else:
            nw = int(scale*w)
            nh = int(nw/new_ar)
        image = image.resize((nw,nh), Image.BICUBIC)

        # place image
        dx = int(rand(0, w-nw))
        dy = int(rand(0, h-nh))
        new_image = Image.new('RGB', (w,h), (128,128,128))
        new_image.paste(image, (dx, dy))
        image = new_image

        # flip image or not
        flip = rand()<.5
        if flip: image = image.transpose(Image.FLIP_LEFT_RIGHT)

        # distort image
        hue = rand(-hue, hue)
        sat = rand(1, sat) if rand()<.5 else 1/rand(1, sat)
        val = rand(1, val) if rand()<.5 else 1/rand(1, val)
        x = cv2.cvtColor(np.array(image,np.float32)/255, cv2.COLOR_RGB2HSV)
        x[..., 0] += hue*360
        x[..., 0][x[..., 0]>1] -= 1
        x[..., 0][x[..., 0]<0] += 1
        x[..., 1] *= sat
        x[..., 2] *= val
        x[x[:,:, 0]>360, 0] = 360
```

```
        x[:, :, 1:][x[:, :, 1:]>1] = 1
        x[x<0] = 0
        image_data = cv2.cvtColor(x, cv2.COLOR_HSV2RGB)*255

        # correct boxes
        box_data = np.zeros((len(box),5))
        if len(box)>0:
            np.random.shuffle(box)
            box[:, [0,2]] = box[:, [0,2]]*nw/iw + dx
            box[:, [1,3]] = box[:, [1,3]]*nh/ih + dy
            if flip: box[:, [0,2]] = w - box[:, [2,0]]
            box[:, 0:2][box[:, 0:2]<0] = 0
            box[:, 2][box[:, 2]>w] = w
            box[:, 3][box[:, 3]>h] = h
            box_w = box[:, 2] - box[:, 0]
            box_h = box[:, 3] - box[:, 1]
            box = box[np.logical_and(box_w>1, box_h>1)] # discard invalid box
            box_data = np.zeros((len(box),5))
            box_data[:len(box)] = box
        if len(box) == 0:
            return image_data, []

        if (box_data[:,:4]>0).any():
            return image_data, box_data
        else:
            return image_data, []

    def generate(self, train=True):
        '''
        读取需要训练的图片并进行处理
        '''
        while True:
            shuffle(self.train_lines)
            lines = self.train_lines
            inputs = []
            targets = []
            for annotation_line in lines:
                img,y=self.get_random_data(annotation_line,self.image_size[0:2]) # ge

                if len(y)!=0:
                    boxes = np.array(y[:,:4],dtype=np.float32)
                    boxes[:,0] = boxes[:,0]/self.image_size[1] # 下面四行是对数据进行
                    boxes[:,1] = boxes[:,1]/self.image_size[0]
                    boxes[:,2] = boxes[:,2]/self.image_size[1]
                    boxes[:,3] = boxes[:,3]/self.image_size[0]
                    # gr框是左上右下的格式，我们要转换为中心+宽高的格式
                    boxes = np.maximum(np.minimum(boxes,1),0)
                    boxes[:,2] = boxes[:,2] - boxes[:,0]
                    boxes[:,3] = boxes[:,3] - boxes[:,1]

                    boxes[:,0] = boxes[:,0] + boxes[:,2]/2
                    boxes[:,1] = boxes[:,1] + boxes[:,3]/2
                    y = np.concatenate([boxes,y[:,-1:]],axis=-1)
                img = np.array(img,dtype = np.float32)

                inputs.append(np.transpose(img/255.0,(2,0,1))) # 对图片进行归一化，进
                targets.append(np.array(y,dtype = np.float32))
                if len(targets) == self.batch_size: # 如果处理的图片数已经等于batchsi
                    tmp_inp = np.array(inputs)
                    tmp_targets = np.array(targets)
                    inputs = []
                    targets = []
                    yield tmp_inp, tmp_targets # 返回处理的图片和对应的框框
```

In [ ]:
```
#-----------------------------------#

###############yolo.py###############

###########定义预测阶段的结构##########

#-----------------------------------#


#-------------------------------------#
#       创建YOLO类
#-------------------------------------#
import cv2
import numpy as np
import colorsys
import os
import torch
import torch.nn as nn
from nets.yolo3 import YoloBody
import torch.backends.cudnn as cudnn
from PIL import Image,ImageFont, ImageDraw
from torch.autograd import Variable
from utils.config import Config
from utils.utils import non_max_suppression, bbox_iou, DecodeBox,letterbox_image,yol

#--------------------------------------------#
#   使用自己训练好的模型预测需要修改2个参数
#   model_path和classes_path都需要修改！
#--------------------------------------------#
class YOLO(object):
    #
    _defaults = {
        "model_path": 'logs\Epoch1-Total_Loss63.1416-Val_Loss15.9550.pth',
        "classes_path": 'model_data/voc_classes.txt',
        "model_image_size" : (416, 416, 3),
        "confidence": 0.5,
        "cuda": True
    }

    @classmethod
    def get_defaults(cls, n):
        if n in cls._defaults:
            return cls._defaults[n]
        else:
            return "Unrecognized attribute name '" + n + "'"

    #---------------------------------------------------#
    #   初始化YOLO
    #---------------------------------------------------#
    def __init__(self, **kwargs):
        self.__dict__.update(self._defaults)
        self.class_names = self._get_class()
        self.config = Config
        self.generate()
    #---------------------------------------------------#
    #   获得所有的分类
    #---------------------------------------------------#
    def _get_class(self): # 载入目标包含的类数
        classes_path = os.path.expanduser(self.classes_path)
        with open(classes_path) as f:
            class_names = f.readlines()
        class_names = [c.strip() for c in class_names]
        return class_names
    #---------------------------------------------------#
    #   获得所有的分类
```

```
    #-------------------------------------------------#
    def generate(self):
        self.config["yolo"]["classes"] = len(self.class_names)
        self.net = YoloBody(self.config)

        # 加快模型训练的效率
        print('Loading weights into state dict...')
        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu') # 利用
        state_dict = torch.load(self.model_path, map_location=device) # 载入权重文件
        self.net.load_state_dict(state_dict)
        self.net = self.net.eval()

        if self.cuda:
            os.environ["CUDA_VISIBLE_DEVICES"] = '0'
            self.net = nn.DataParallel(self.net)
            self.net = self.net.cuda()

        self.yolo_decodes = []
        for i in range(3):
            self.yolo_decodes.append(DecodeBox(self.config["yolo"]["anchors"][i], sel


        print('{} model, anchors, and classes loaded.'.format(self.model_path))
        # 画框设置不同的颜色
        hsv_tuples = [(x / len(self.class_names), 1., 1.)
                      for x in range(len(self.class_names))]
        self.colors = list(map(lambda x: colorsys.hsv_to_rgb(*x), hsv_tuples))
        self.colors = list(
            map(lambda x: (int(x[0] * 255), int(x[1] * 255), int(x[2] * 255)),
                self.colors))

    #-------------------------------------------------#
    #   检测图片
    #-------------------------------------------------#
    def detect_image(self, image):
        image_shape = np.array(np.shape(image)[0:2])
        # 图片处理
        crop_img = np.array(letterbox_image(image, (self.model_image_size[0],self.mod
        photo = np.array(crop_img,dtype = np.float32)
        photo /= 255.0 # 归一化
        photo = np.transpose(photo, (2, 0, 1)) # 在pytorch中通道数在第一个，所以在这
        photo = photo.astype(np.float32) # 转换数据类型
        images = []
        images.append(photo)

        images = np.asarray(images)
        images = torch.from_numpy(images) # 将numpy转换成tenor类型
        if self.cuda:
            images = images.cuda()
        # 放入网络中进行预测并画框
        with torch.no_grad():
            outputs = self.net(images) # 图片放入网络中
            output_list = []
            for i in range(3): # 特征层解码，因为特征金字塔有三个尺度的输出，所以要循
                output_list.append(self.yolo_decodes[i](outputs[i])) # 解码：调整先验
            output = torch.cat(output_list, 1) # 将预测结果堆叠起来
            batch_detections = non_max_suppression(output, self.config["yolo"]["class
                                                   conf_thres=self.confidence,
                                                   nms_thres=0.3) # non_max_suppress
        try :
            batch_detections = batch_detections[0].cpu().numpy()
        except:
            return image
        top_index = batch_detections[:,4]*batch_detections[:,5] > self.confidence # 
        top_conf = batch_detections[top_index,4]*batch_detections[top_index,5] # 下面
```

```
        top_label = np.array(batch_detections[top_index,-1],np.int32)
        top_bboxes = np.array(batch_detections[top_index,:4])
        top_xmin, top_ymin, top_xmax, top_ymax = np.expand_dims(top_bboxes[:,0],-1),n

        # 去掉灰条
        '''
        目前框框的位置是相对于有灰条图片左上角的位置。去掉灰条要转换为原图的左上角的位
        yolo_correct_boxes函数就是完成这样的坐标变换
        '''
        boxes = yolo_correct_boxes(top_ymin,top_xmin,top_ymax,top_xmax,np.array([self.

        font = ImageFont.truetype(font='model_data/simhei.ttf',size=np.floor(3e-2 * n

        thickness = (np.shape(image)[0] + np.shape(image)[1]) // self.model_image_siz
        # 下面的代码就是用来画图的
        for i, c in enumerate(top_label):
            predicted_class = self.class_names[c] # 获得类的名称
            score = top_conf[i] # 获得得分
            # 获得位置信息
            top, left, bottom, right = boxes[i]
            top = top - 5
            left = left - 5
            bottom = bottom + 5
            right = right + 5

            top = max(0, np.floor(top + 0.5).astype('int32'))
            left = max(0, np.floor(left + 0.5).astype('int32'))
            bottom = min(np.shape(image)[0], np.floor(bottom + 0.5).astype('int32'))
            right = min(np.shape(image)[1], np.floor(right + 0.5).astype('int32'))

            # 画框框
            label = '{} {:.2f}'.format(predicted_class, score)
            draw = ImageDraw.Draw(image)
            label_size = draw.textsize(label, font)
            label = label.encode('utf-8')
            print(label)

            if top - label_size[1] >= 0:
                text_origin = np.array([left, top - label_size[1]])
            else:
                text_origin = np.array([left, top + 1])

            for i in range(thickness):
                draw.rectangle(
                    [left + i, top + i, right - i, bottom - i],
                    outline=self.colors[self.class_names.index(predicted_class)])
            draw.rectangle(
                [tuple(text_origin), tuple(text_origin + label_size)],
                fill=self.colors[self.class_names.index(predicted_class)])
            draw.text(text_origin, str(label,'UTF-8'), fill=(0, 0, 0), font=font) # 在
            del draw
        return image
```

In [ ]:

```
#-----------------------------------#

##############train.py###############

########启动训练，定义训练过程########

#-----------------------------------#



#-------------------------------------#
#       对数据集进行训练
```

```
#-------------------------------------#
import os
import numpy as np
import time
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torch.backends.cudnn as cudnn
from utils.config import Config
from torch.utils.data import DataLoader
from utils.dataloader import yolo_dataset_collate, YoloDataset
from nets.yolo_training import YOLOLoss,Generator
from nets.yolo3 import YoloBody
from tqdm import tqdm

def get_lr(optimizer):
    for param_group in optimizer.param_groups:
        return param_group['lr']

def fit_ont_epoch(net,yolo_losses,epoch,epoch_size,epoch_size_val,gen,genval,Epoch,cu
    '''
    用于训练的函数
    '''
    total_loss = 0
    val_loss = 0
    start_time = time.time()
    with tqdm(total=epoch_size,desc=f'Epoch {epoch + 1}/{Epoch}',postfix=dict,mininte
        for iteration, batch in enumerate(gen):
            if iteration >= epoch_size:
                break
            images, targets = batch[0], batch[1]
            with torch.no_grad():
                if cuda:
                    images = Variable(torch.from_numpy(images).type(torch.FloatTensor
                    targets = [Variable(torch.from_numpy(ann).type(torch.FloatTensor)
                else:
                    images = Variable(torch.from_numpy(images).type(torch.FloatTensor
                    targets = [Variable(torch.from_numpy(ann).type(torch.FloatTensor)
            optimizer.zero_grad() # 梯度清零
            outputs = net(images) # 输出预测结果
            losses = []
            for i in range(3): # 对结果计算loss
                loss_item = yolo_losses[i](outputs[i], targets)
                losses.append(loss_item[0]) # 将三个特征层的LOSS叠加起来
            loss = sum(losses)
            loss.backward() # 反向梯度计算
            optimizer.step()

            total_loss += loss
            waste_time = time.time() - start_time

            pbar.set_postfix(**{'total_loss': total_loss.item() / (iteration + 1),
                                'lr'        : get_lr(optimizer),
                                'step/s'    : waste_time})
            pbar.update(1)

            start_time = time.time()

    print('Start Validation') # 后面是对验证集LOSS的计算
    with tqdm(total=epoch_size_val, desc=f'Epoch {epoch + 1}/{Epoch}',postfix=dict,mi
        for iteration, batch in enumerate(genval):
            if iteration >= epoch_size_val:
                break
```

```
            images_val, targets_val = batch[0], batch[1]

            with torch.no_grad():
                if cuda:
                    images_val = Variable(torch.from_numpy(images_val).type(torch.Flo
                    targets_val = [Variable(torch.from_numpy(ann).type(torch.FloatTen
                else:
                    images_val = Variable(torch.from_numpy(images_val).type(torch.Flo
                    targets_val = [Variable(torch.from_numpy(ann).type(torch.FloatTen
                optimizer.zero_grad()
                outputs = net(images_val)
                losses = []
                for i in range(3):
                    loss_item = yolo_losses[i](outputs[i], targets_val)
                    losses.append(loss_item[0])
                loss = sum(losses)
                val_loss += loss
            pbar.set_postfix(**{'total_loss': val_loss.item() / (iteration + 1)})
            pbar.update(1)

    print('Finish Validation')
    print('Epoch:'+ str(epoch+1) + '/' + str(Epoch))
    print('Total Loss: %.4f || Val Loss: %.4f ' % (total_loss/(epoch_size+1),val_loss

    print('Saving state, iter:', str(epoch+1))
    torch.save(model.state_dict(), 'logs/Epoch%d-Total_Loss%.4f-Val_Loss%.4f.pth'%((e

#----------------------------------------------------#
#   检测精度mAP和pr曲线计算参考视频
#   https://www.bilibili.com/video/BV1zE411u7Vw
#----------------------------------------------------#
if __name__ == "__main__":
    # 参数初始化
    annotation_path = '2007_train.txt' # 获取训练所需目标信息
    model = YoloBody(Config) # 创建yolo的模型
    Cuda = True
    #-------------------------------#
    #   Dataloder的使用
    #-------------------------------#
    Use_Data_Loader = True

    #-------------------------------------------#
    #   权值文件的下载请看README
    #-------------------------------------------#
    print('Loading weights into state dict...') # 这一部分是载入预训练的权重
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model_dict = model.state_dict()
    pretrained_dict = torch.load("model_data/yolo_weights.pth", map_location=device)
    pretrained_dict = {k: v for k, v in pretrained_dict.items() if np.shape(model_di
    model_dict.update(pretrained_dict)
    model.load_state_dict(model_dict)
    print('Finished!')

    net = model.train()

    if Cuda: # 设置cuda参数
        net = torch.nn.DataParallel(model)
        cudnn.benchmark = True
        net = net.cuda()

    # 建立loss函数
    yolo_losses = []
    for i in range(3): # 因为yolov3的网络会输出三个有效特征层，所以LOSS函数要循环三次
        yolo_losses.append(YOLOLoss(np.reshape(Config["yolo"]["anchors"],[-1,2]),
                                    Config["yolo"]["classes"], (Config["img_w"], Confi
```

```
# 0.1用于验证，0.9用于训练
val_split = 0.1 # 验证集占0.1
with open(annotation_path) as f:
    lines = f.readlines()
np.random.seed(10101)
np.random.shuffle(lines)
np.random.seed(None)
num_val = int(len(lines)*val_split)
num_train = len(lines) - num_val


#------------------------------------------------------#
#   主干特征提取网络特征通用，冻结训练可以加快训练速度
#   也可以在训练初期防止权值被破坏。
#   Init_Epoch为起始世代
#   Freeze_Epoch为冻结训练的世代
#   Epoch总训练世代
#   提示OOM或者显存不足请调小Batch_size
#------------------------------------------------------#
if True:
    # 最开始使用1e-3的学习率可以收敛的更快
    lr = 1e-3
    Batch_size = 8
    Init_Epoch = 0 # 初始训练位于的代数
    Freeze_Epoch = 25 # 冻结模型训练要持续多少代

    optimizer = optim.Adam(net.parameters(),lr) # 定义模型训练使用的优化器
    lr_scheduler = optim.lr_scheduler.StepLR(optimizer,step_size=1,gamma=0.95) #

    if Use_Data_Loader:
        train_dataset = YoloDataset(lines[:num_train], (Config["img_h"], Config["
        val_dataset = YoloDataset(lines[num_train:], (Config["img_h"], Config["img
        gen = DataLoader(train_dataset, batch_size=Batch_size, num_workers=4, pin
                                drop_last=True, collate_fn=yolo_dataset_collate)
        gen_val = DataLoader(val_dataset, batch_size=Batch_size, num_workers=4,pi
                                drop_last=True, collate_fn=yolo_dataset_collate)
    else:
        gen = Generator(Batch_size, lines[:num_train],
                        (Config["img_h"], Config["img_w"])).generate() # 生成训练
        gen_val = Generator(Batch_size, lines[num_train:],
                        (Config["img_h"], Config["img_w"])).generate() # 生成验证

    epoch_size = num_train//Batch_size # 每个世代训练的步长
    epoch_size_val = num_val//Batch_size # 每个世代验证的步长
    #------------------------------------#
    #   冻结一定部分训练
    #------------------------------------#
    for param in model.backbone.parameters(): # 冻结模型
        param.requires_grad = False

    for epoch in range(Init_Epoch,Freeze_Epoch): # 开始训练
        fit_ont_epoch(net,yolo_losses,epoch,epoch_size,epoch_size_val,gen,gen_val,
        lr_scheduler.step()

if True: # 解冻之后的训练
    lr = 1e-4
    Batch_size = 4
    Freeze_Epoch = 25
    Unfreeze_Epoch = 50

    optimizer = optim.Adam(net.parameters(),lr)
    lr_scheduler = optim.lr_scheduler.StepLR(optimizer,step_size=1,gamma=0.95)
    if Use_Data_Loader:
        train_dataset = YoloDataset(lines[:num_train], (Config["img_h"], Config["
```

```
            val_dataset = YoloDataset(lines[num_train:], (Config["img_h"], Config["img
            gen = DataLoader(train_dataset, batch_size=Batch_size, num_workers=4, pin_
                                    drop_last=True, collate_fn=yolo_dataset_collate)
            gen_val = DataLoader(val_dataset, batch_size=Batch_size, num_workers=4,pi
                                    drop_last=True, collate_fn=yolo_dataset_collate)
        else:
            gen = Generator(Batch_size, lines[:num_train],
                            (Config["img_h"], Config["img_w"])).generate()
            gen_val = Generator(Batch_size, lines[num_train:],
                            (Config["img_h"], Config["img_w"])).generate()

        epoch_size = num_train//Batch_size
        epoch_size_val = num_val//Batch_size
        #------------------------------------#
        #   解冻后训练
        #------------------------------------#
        for param in model.backbone.parameters():
            param.requires_grad = True

        for epoch in range(Freeze_Epoch,Unfreeze_Epoch):
            fit_ont_epoch(net,yolo_losses,epoch,epoch_size,epoch_size_val,gen,gen_val,
            lr_scheduler.step()
```