睿智的目标检测26——Pytorch搭建yolo3目标检测平台

睿智的目标检测26——Pytorch搭建yolo3目标检测平台

注意事项

yolov3网络结构图中，特征高宽最小的特征层的通道数量不对，正确的输出特征层shape为[batch_size, 13, 13, 512]。代码是正确的。

学习前言

一起来看看yolo3的Pytorch实现吧，顺便训练一下自己的数据。

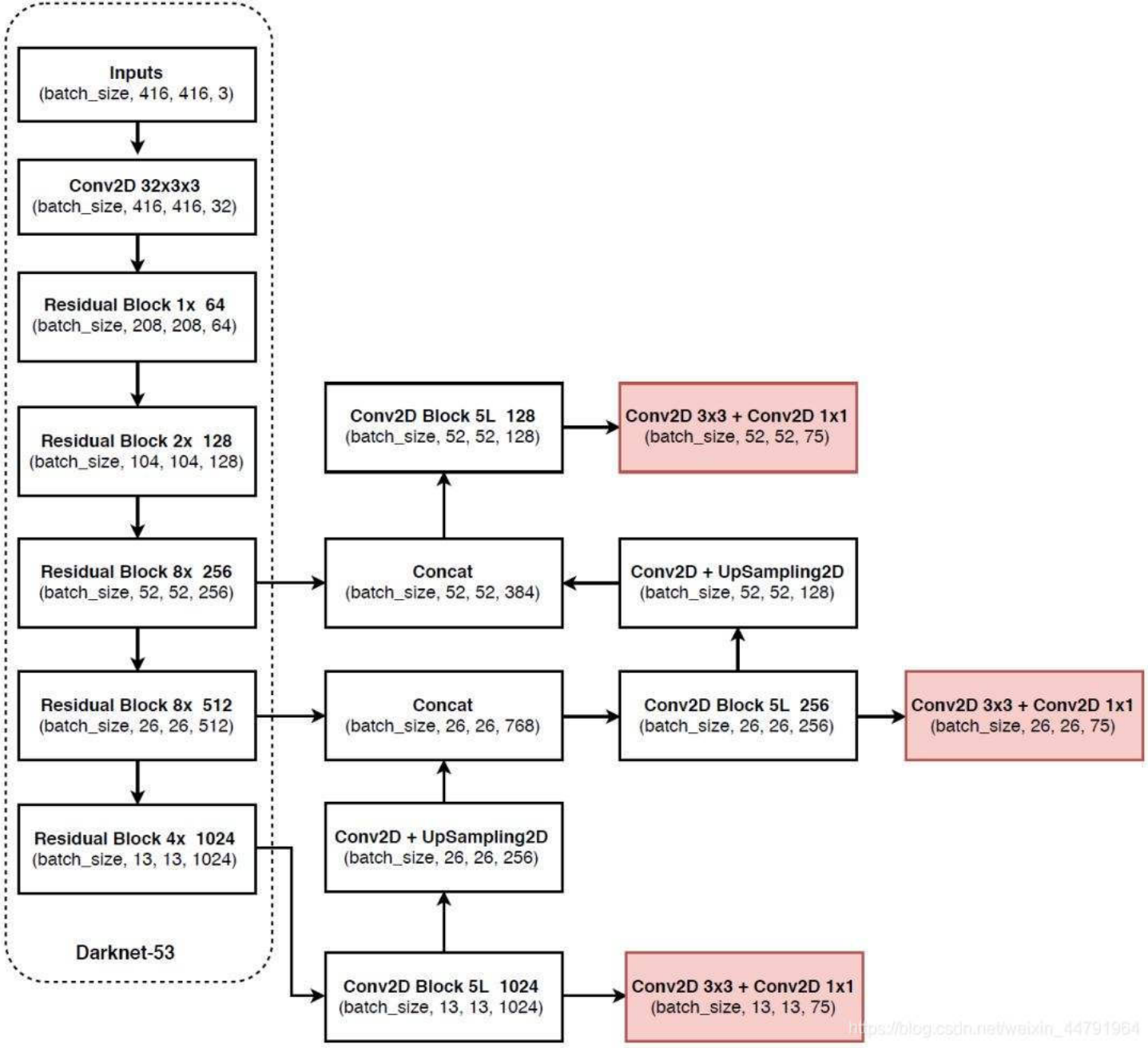
源码下载

https://github.com/bubbliiiing/yolo3-pytorch
喜欢的可以点个star噢。

yolo3实现思路

一、预测部分

1、主题网络darknet53介绍



PS：该图有一些小问题，宽高最小的特征层在经过Conv2D Block 5L的处理后，它的shape按照代码应该为(batch_size,13,13,512)，而非图中的(batch_size,13,13,1024)。

YoloV3所使用的主干特征提取网络为Darknet53，它具有两个重要特点：
1、Darknet53具有一个重要特点是使用了**残差网络Residual**，Darknet53中的残差卷积就是**首先进行一次卷积核大小为3X3、步长为2的卷积，该卷积会压缩输入进来的特征层的宽和高，此时我们可以获得一个特征层，我们将该特征层命名为layer。之后我们再对该特征层进行一次1X1的卷积和一次3X3的卷积，并把这个结果加上layer，此时我们便构成了残差结构。**通过**不断的1X1卷积和3X3卷积以及残差边的叠加，我们便大幅度的加深了网络。**残差网络的特点是**容易优化**，并且能够通过增加相当的**深度来提高准确率。**其内部的**残差块使用了跳跃连接，缓解了在深度神经网络中增加深度带来的梯度消失问题。**

2、Darknet53的每一个卷积部分**使用了特有的DarknetConv2D结构**，每一次卷积的时候进行l2正则化，**完成卷积后进行BatchNormalization标准化与LeakyReLU。普通的ReLU是将所有的负值都设为零，Leaky ReLU则是给所有负值赋予一个非零斜率。以数学的方式我们可以表示为**：

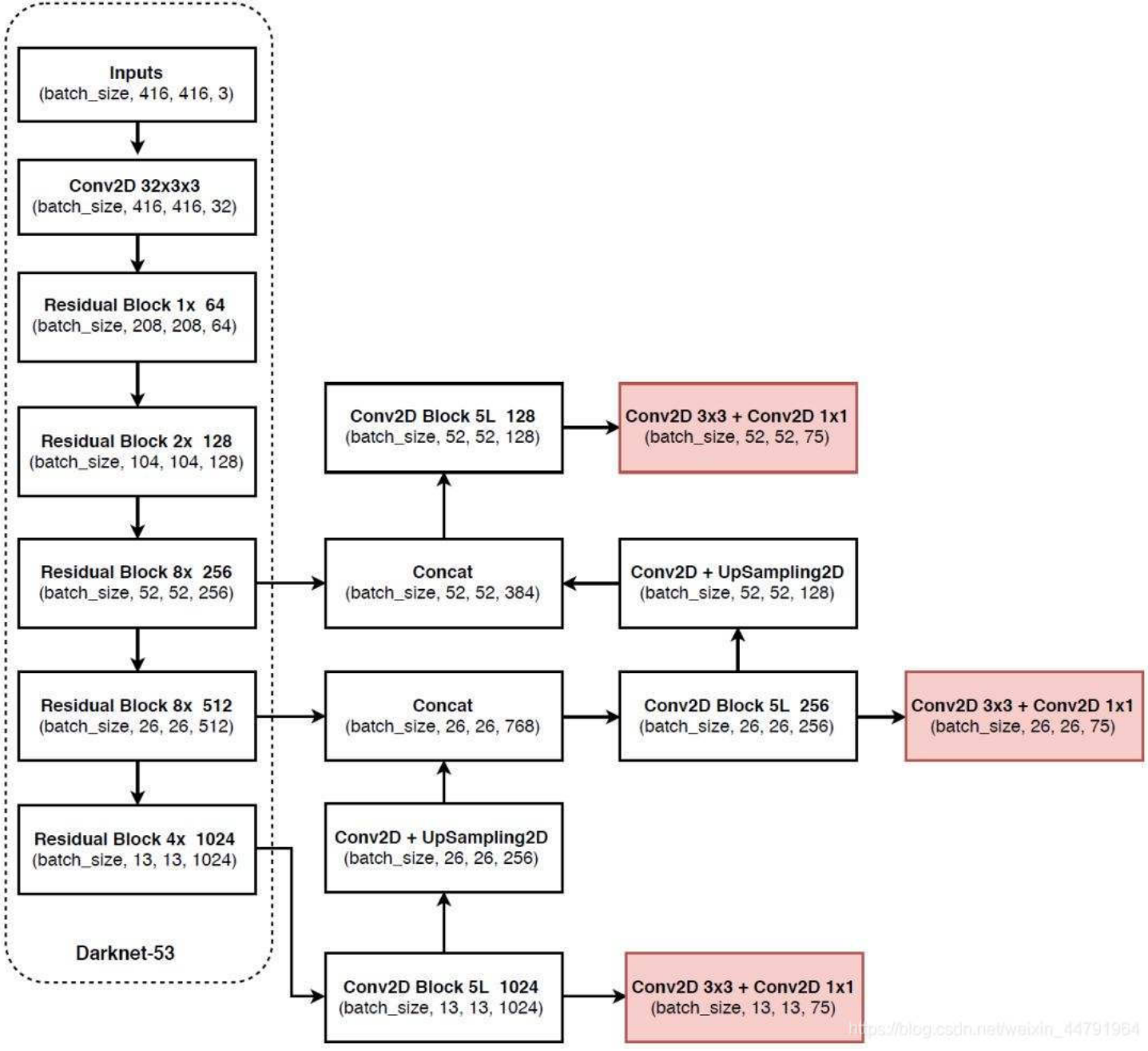
$$y_i = \begin{cases} x_i & \text{if } x_i \geq 0 \\ \frac{x_i}{a_i} & \text{if } x_i < 0, \end{cases}$$

实现代码为：

```python
import math
from collections import OrderedDict

import torch
import torch.nn as nn


#---------------------------------------------------------------------#
#   残差结构
#   利用一个1x1卷积下降通道数，然后利用一个3x3卷积提取特征并且上升通道数
#   最后接上一个残差边
#---------------------------------------------------------------------#
class BasicBlock(nn.Module):
    def __init__(self, inplanes, planes):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(inplanes, planes[0], kernel_size=1,
                               stride=1, padding=0, bias=False)
        self.bn1 = nn.BatchNorm2d(planes[0])
        self.relu1 = nn.LeakyReLU(0.1)

        self.conv2 = nn.Conv2d(planes[0], planes[1], kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes[1])
        self.relu2 = nn.LeakyReLU(0.1)

    def forward(self, x):
        residual = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu1(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu2(out)

        out += residual
        return out


class DarkNet(nn.Module):
    def __init__(self, layers):
        super(DarkNet, self).__init__()
        self.inplanes = 32
        # 416,416,3 -> 416,416,32
```

```
        self.conv1 = nn.Conv2d(3, self.inplanes, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(self.inplanes)
        self.relu1 = nn.LeakyReLU(0.1)

        # 416,416,32 -> 208,208,64
        self.layer1 = self._make_layer([32, 64], layers[0])
        # 208,208,64 -> 104,104,128
        self.layer2 = self._make_layer([64, 128], layers[1])
        # 104,104,128 -> 52,52,256
        self.layer3 = self._make_layer([128, 256], layers[2])
        # 52,52,256 -> 26,26,512
        self.layer4 = self._make_layer([256, 512], layers[3])
        # 26,26,512 -> 13,13,1024
        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_()

    #---------------------------------------------------------------------#
    #   在每一个layer里面，首先利用一个步长为2的3x3卷积进行下采样
    #   然后进行残差结构的堆叠
    #---------------------------------------------------------------------#
    def _make_layer(self, planes, blocks):
        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)))
        layers.append(("ds_bn", nn.BatchNorm2d(planes[1])))
        layers.append(("ds_relu", nn.LeakyReLU(0.1)))
        # 加入残差结构
        self.inplanes = planes[1]
        for i in range(0, blocks):
            layers.append(("residual_{}".format(i), BasicBlock(self.inplanes, planes)))
        return nn.Sequential(OrderedDict(layers))

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu1(x)

        x = self.layer1(x)
        x = self.layer2(x)
        out3 = self.layer3(x)
        out4 = self.layer4(out3)
        out5 = self.layer5(out4)

        return out3, out4, out5

def darknet53(pretrained, **kwargs):
    model = DarkNet([1, 2, 8, 8, 4])
    if pretrained:
        if isinstance(pretrained, str):
            model.load_state_dict(torch.load(pretrained))
        else:
            raise Exception("darknet request a pretrained path. got [{}]".format(pretrained))
    return model
```

2、从特征获取预测结果



从**特征获取预测结果的过程**可以分为两个部分，分别是：

- 构建**FPN特征金字塔进行加强特征提取。**
- 利用**Yolo Head对三个有效特征层进行预测。**

a、构建FPN特征金字塔进行加强特征提取

在特征利用部分，YoloV3提取**多特征层进行目标检测**，一共**提取三个特征层。**
三个特征层位于主干部分Darknet53的不同位置，分别位于**中间层，中下层，底层**，三个特征层的**shape分别为(52,52,256)、(26,26,512)、(13,13,1024)。**

在获得三个有效特征层后，我们利用这三个有效特征层进行FPN层的构建，构建方式为：

1. 13x13x1024的特征层进行5次卷积处理，处理完后**利用YoloHead获得预测结果**，**一部分用于进行上采样UmSampling2d后与26x26x512特征层进行结合**，结合特征层的shape为(26,26,768)。
2. 结合特征层再次进行5次卷积处理，处理完后**利用YoloHead获得预测结果**，**一部分用于进行上采样UmSampling2d后与52x52x256特征层进行结合**，结合特征层的shape为(52,52,384)。
3. 结合特征层再次进行5次卷积处理，处理完后**利用YoloHead获得预测结果。**

特征金字塔可以将**不同shape的特征层进行特征融合**，有利于**提取出更好的特征。**

b、利用Yolo Head获得预测结果

利用FPN特征金字塔，**我们可以获得三个加强特征，这三个加强特征的shape分别为(13,13,512)、(26,26,256)、(52,52,128)，然后我们利用这三个shape的特征层传入Yolo Head获得预测结果。**

Yolo Head本质上是一次3x3卷积加上一次1x1卷积，3x3卷积的作用是特征整合，1x1卷积的作用是调整通道数。

对三个特征层分别进行处理，假设我们预测是的VOC数据集，我们的输出层的shape分别为(**13,13,75**)，(**26,26,75**)，(**52,52,75**)，**最后一个维度为75是因为该图是基于voc数据集的，它的类为20种，YoloV3针对每一个特征层的每一个特征点存在3个先验框，所以预测结果的通道数为3x25；**
如果使用的是coco训练集，类则为80种，最后的维度应该为255 = 3x85，三个特征层的shape为(**13,13,255**)，(**26,26,255**)，(**52,52,255**)

其实际情况就是，输入N张416x416的图片，在经过多层的运算后，会输出三个shape分别为(**N,13,13,255**)，(**N,26,26,255**)，(**N,52,52,255**)的数据，对应每个图分为13x13、26x26、52x52的网格上3个先验框的位置。

实现代码如下：

```python
from collections import OrderedDict

import torch
import torch.nn as nn

from nets.darknet import darknet53


def conv2d(filter_in, filter_out, kernel_size):
    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, padding=pad, bias=False)),
        ("bn", nn.BatchNorm2d(filter_out)),
        ("relu", nn.LeakyReLU(0.1)),
    ]))

#------------------------------------------------------------------------#
#   make_last_layers里面一共有七个卷积，前五个用于提取特征。
#   后两个用于获得yolo网络的预测结果
#------------------------------------------------------------------------#
def make_last_layers(filters_list, in_filters, out_filter):
    m = nn.ModuleList([
        conv2d(in_filters, filters_list[0], 1),
        conv2d(filters_list[0], filters_list[1], 3),
        conv2d(filters_list[1], filters_list[0], 1),
        conv2d(filters_list[0], filters_list[1], 3),
        conv2d(filters_list[1], filters_list[0], 1),
```

```
        conv2d(filters_list[0], filters_list[1], 3),
        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, anchor, num_classes):
        super(YoloBody, self).__init__()
        #---------------------------------------------------#
        #   生成darknet53的主干模型
        #   获得三个有效特征层，他们的shape分别是：
        #   52,52,256
        #   26,26,512
        #   13,13,1024
        #---------------------------------------------------#
        self.backbone = darknet53(None)

        # out_filters : [64, 128, 256, 512, 1024]
        out_filters = self.backbone.layers_out_filters

        #------------------------------------------------------------------------#
        #   计算yolo_head的输出通道数，对于voc数据集而言
        #   final_out_filter0 = final_out_filter1 = final_out_filter2 = 75
        #------------------------------------------------------------------------#
        final_out_filter0 = len(anchor[0]) * (5 + num_classes)
        self.last_layer0 = make_last_layers([512, 1024], out_filters[-1], final_out_filter0)

        final_out_filter1 = len(anchor[1]) * (5 + num_classes)
        self.last_layer1_conv = conv2d(512, 256, 1)
        self.last_layer1_upsample = nn.Upsample(scale_factor=2, mode='nearest')
        self.last_layer1 = make_last_layers([256, 512], out_filters[-2] + 256, final_out_filter1)

        final_out_filter2 = len(anchor[2]) * (5 + num_classes)
        self.last_layer2_conv = conv2d(256, 128, 1)
        self.last_layer2_upsample = nn.Upsample(scale_factor=2, mode='nearest')
        self.last_layer2 = make_last_layers([128, 256], out_filters[-3] + 128, final_out_filter2)


    def forward(self, x):
        def _branch(last_layer, layer_in):
            for i, e in enumerate(last_layer):
                layer_in = e(layer_in)
                if i == 4:
                    out_branch = layer_in
            return layer_in, out_branch
        #---------------------------------------------------#
        #   获得三个有效特征层，他们的shape分别是：
        #   52,52,256；26,26,512；13,13,1024
        #---------------------------------------------------#
        x2, x1, x0 = self.backbone(x)

        #---------------------------------------------------#
        #   第一个特征层
        #   out0 = (batch_size,255,13,13)
        #---------------------------------------------------#
        # 13,13,1024 -> 13,13,512 -> 13,13,1024 -> 13,13,512 -> 13,13,1024 -> 13,13,512
        out0, out0_branch = _branch(self.last_layer0, x0)

        # 13,13,512 -> 13,13,256 -> 26,26,256
        x1_in = self.last_layer1_conv(out0_branch)
        x1_in = self.last_layer1_upsample(x1_in)

        # 26,26,256 + 26,26,512 -> 26,26,768
        x1_in = torch.cat([x1_in, x1], 1)
        #---------------------------------------------------#
        #   第二个特征层
        #   out1 = (batch_size,255,26,26)
        #---------------------------------------------------#
        # 26,26,768 -> 26,26,256 -> 26,26,512 -> 26,26,256 -> 26,26,512 -> 26,26,256
        out1, out1_branch = _branch(self.last_layer1, x1_in)

        # 26,26,256 -> 26,26,128 -> 52,52,128
        x2_in = self.last_layer2_conv(out1_branch)
        x2_in = self.last_layer2_upsample(x2_in)

        # 52,52,128 + 52,52,256 -> 52,52,384
        x2_in = torch.cat([x2_in, x2], 1)
        #---------------------------------------------------#
        #   第一个特征层
        #   out3 = (batch_size,255,52,52)
        #---------------------------------------------------#
        # 52,52,384 -> 52,52,128 -> 52,52,256 -> 52,52,128 -> 52,52,256 -> 52,52,128
        out2, _ = _branch(self.last_layer2, x2_in)
        return out0, out1, out2
```

3、预测结果的解码

由第二步我们可以获得三个特征层的预测结果，shape分别为：

- (**N,13,13,255**)
- (**N,26,26,255**)
- (**N,52,52,255**)

在这里我们简单了解一下每个有效特征层到底做了什么：
每一个有效特征层将整个图片分成与其**长宽对应的网格，如(N,13,13,255)的特征层就是将整个图像分成13x13个网格；然后从每个网格中心建立多个先验框，这些框是网络预先设定好的框，网络的预测结果会判断这些框内是否包含物体，以及这个物体的种类。**

由于每一个网格点都具有三个先验框，所以上述的预测结果可以reshape为：

- (**N,13,13,3,85**)
- (**N,26,26,3,85**)
- (**N,52,52,3,85**)

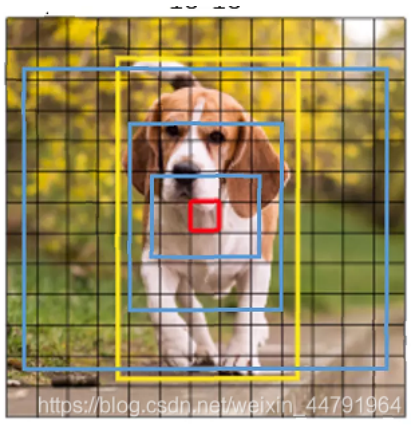
其中的85可以拆分为4+1+80，其中的4代表先验框的调整参数，1代表先验框内是否包含物体，80代表的是这个先验框的种类，由于coco分了80类，所以这里是80。如果YoloV3只检测两类物体，那么这个85就变为了4+1+2 = 7。

即85包含了4+1+80，**分别代表x_offset、y_offset、h和w、置信度、分类结果。**

但是这个预测结果并不对应着最终的预测框在图片上的位置，还需要解码才可以完成。

YoloV3的解码过程分为两步：

- 先**将每个网格点加上它对应的x_offset和y_offset**，加完后的结果就是**预测框的中心。**
- 然后**再利用 先验框和h、w结合 计算出预测框的宽高。**这样就能得到整个预测框的位置了。



得到最终的预测结果后还要进行**得分排序与非极大抑制筛选。**

这一部分基本上是所有目标检测通用的部分。其对于每一个类进行判别：
1、取出每一类得分大于self.obj_threshold的框和得分。
2、利用框的位置和得分进行非极大抑制。

实现代码如下

```
class DecodeBox(nn.Module):
    def __init__(self, anchors, num_classes, img_size):
        super(DecodeBox, self).__init__()
        #-----------------------------------------------------------#
        #   13x13的特征层对应的anchor是[116,90],[156,198],[373,326]
        #   26x26的特征层对应的anchor是[30,61],[62,45],[59,119]
        #   52x52的特征层对应的anchor是[10,13],[16,30],[33,23]
        #-----------------------------------------------------------#
        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):
        #-----------------------------------------------#
        #   输入的input一共有三个，他们的shape分别是
        #   batch_size, 255, 13, 13
        #   batch_size, 255, 26, 26
        #   batch_size, 255, 52, 52
        #-----------------------------------------------#
        batch_size = input.size(0)
        input_height = input.size(2)
        input_width = input.size(3)

        #-----------------------------------------------#
        #   输入为416x416时
        #   stride_h = stride_w = 32、16、8
        #-----------------------------------------------#
        stride_h = self.img_size[1] / input_height
        stride_w = self.img_size[0] / input_width
        #-------------------------------------------------#
        #   此时获得的scaled_anchors大小是相对于特征层的
        #-------------------------------------------------#
        scaled_anchors = [(anchor_width / stride_w, anchor_height / stride_h) for anchor_width, anchor_height in self.anchors]

        #-----------------------------------------------#
        #   输入的input一共有三个，他们的shape分别是
        #   batch_size, 3, 13, 13, 85
        #   batch_size, 3, 26, 26, 85
        #   batch_size, 3, 52, 52, 85
        #-----------------------------------------------#
        prediction = input.view(batch_size, self.num_anchors,
                                self.bbox_attrs, input_height, input_width).permute(0, 1, 3, 4, 2).contiguous()

        # 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]
        h = prediction[..., 3]
        # 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:])

        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_height, 1).repeat(
            batch_size * self.num_anchors, 1, 1).view(x.shape).type(FloatTensor)
        grid_y = torch.linspace(0, input_height - 1, input_height).repeat(input_width, 1).t().repeat(
            batch_size * self.num_anchors, 1, 1).view(y.shape).type(FloatTensor)

        #----------------------------------------------------------#
        #   按照网格格式生成先验框的宽高
        #   batch_size,3,13,13
        #----------------------------------------------------------#
```

```
        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_width).view(w.shape)
        anchor_h = anchor_h.repeat(batch_size, 1).repeat(1, 1, input_height * input_width).view(h.shape)

        #-------------------------------------------------------#
        #   利用预测结果对先验框进行调整
        #   首先调整先验框的中心，从先验框中心向右下角偏移
        #   再调整先验框的宽高。
        #-------------------------------------------------------#
        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

        #-------------------------------------------------------#
        #   将输出结果调整成相对于输入图像大小
        #-------------------------------------------------------#
        _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, -1, self.num_classes)), -1)
        return output.data
```

4、在原图上进行绘制

通过第三步，我们可以获得预测框在原图上的位置，而且这些预测框都是经过筛选的。这些筛选后的框可以直接绘制在图片上，就可以获得结果了。

二、训练部分

1、计算loss所需参数

在计算loss的时候，实际上是pred和target之间的对比：
pred就是网络的预测结果。
target就是网络的真实框情况。

2、pred是什么

对于yolo3的模型来说，网络**最后输出的内容就是三个特征层每个网格点对应的预测框及其种类**，即三个特征层分别对应着图片被分为不同size的网格后，**每个网格点上三个先验框对应的位置、置信度及其种类。**

输出层的shape分别为(**13,13,75**)，(**26,26,75**)，(**52,52,75**)，**最后一个维度为75是因为是基于voc数据集的，它的类为20种，yolo3只有针对每一个特征层存在3个先验框，所以最后维度为3x25；**
如果使用的是coco训练集，类则为80种，最后的维度应该为255 = 3x85，三个特征层的shape为(**13,13,255**)，(**26,26,255**)，(**52,52,255**)

现在的y_pre还是没有解码的，解码了之后才是真实图像上的情况。

3、target是什么。

target就是一个真实图像中，真实框的情况。
第一个维度是batch_size，第二个维度是每一张图片里面真实框的数量，第三个维度内部是真实框的信息，包括位置以及种类。

4、loss的计算过程

拿到pred和target后，不可以简单的减一下作为对比，需要进行如下步骤。

1. 判断真实框在图片中的位置，判断其属于哪一个网格点去检测。
2. 判断真实框和哪个先验框重合程度最高。
3. 计算该网格点应该有怎么样的预测结果才能获得真实框
4. 对所有真实框进行如上处理。
5. 获得网络应该有的预测结果，将其与实际的预测结果对比。

```
import os

import math
import numpy as np
import scipy.signal
import torch
import torch.nn as nn
from matplotlib import pyplot as plt

def jaccard(_box_a, _box_b):
    # 计算真实框的左上角和右下角
    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
    # 计算先验框的左上角和右下角
    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):
    def __init__(self, anchors, num_classes, img_size, cuda, normalize):
        super(YOLOLoss, self).__init__()
        #-----------------------------------------------------------#
        #   13x13的特征层对应的anchor是[116,90],[156,198],[373,326]
        #   26x26的特征层对应的anchor是[30,61],[62,45],[59,119]
        #   52x52的特征层对应的anchor是[10,13],[16,30],[33,23]
        #-----------------------------------------------------------#
        self.anchors = anchors
        self.num_anchors = len(anchors)
        self.num_classes = num_classes
        self.bbox_attrs = 5 + num_classes
        #-----------------------------------#
        #   获得特征层的宽高
        #   13、26、52
        #-----------------------------------#
        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
        self.lambda_xy = 1.0
        self.lambda_wh = 1.0
        self.lambda_conf = 1.0
        self.lambda_cls = 1.0
        self.cuda = cuda
        self.normalize = normalize

    def forward(self, input, targets=None):
        #-----------------------------------------------#
        #   input的shape为  bs, 3*(5+num_classes), 13, 13
        #                   bs, 3*(5+num_classes), 26, 26
        #                   bs, 3*(5+num_classes), 52, 52
        #-----------------------------------------------#

        #-----------------------#
        #   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
        #   如果特征层为26x26的话，一个特征点就对应原来的图片上的16个像素点
        #   如果特征层为52x52的话，一个特征点就对应原来的图片上的8个像素点
        #   stride_h = stride_w = 32、16、8
        #-----------------------------------------------------------------------#
        stride_h = self.img_size[1] / in_h
        stride_w = self.img_size[0] / in_w

        #-------------------------------------------------#
        #   此时获得的scaled_anchors大小是相对于特征层的
        #-------------------------------------------------#
        scaled_anchors = [(a_w / stride_w, a_h / stride_h) for a_w, a_h in self.anchors]

        #-------------------------------------------------#
        #   输入的input一共有三个，他们的shape分别是
        #   batch_size, 3, 13, 13, 5 + num_classes
        #   batch_size, 3, 26, 26, 5 + num_classes
        #   batch_size, 3, 52, 52, 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).contiguous()

        # 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]
        h = prediction[..., 3]
        # 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:])

        #---------------------------------------------------------------#
        #   找到哪些先验框内部包含物体
        #   利用真实框和先验框计算交并比
        #   mask        batch_size, 3, in_h, in_w   无目标的特征点
        #   noobj_mask  batch_size, 3, in_h, in_w   有目标的特征点
        #   tx          batch_size, 3, in_h, in_w   中心x偏移情况
        #   ty          batch_size, 3, in_h, in_w   中心y偏移情况
        #   tw          batch_size, 3, in_h, in_w   宽高调整参数的真实值
        #   th          batch_size, 3, in_h, in_w   宽高调整参数的真实值
        #   tconf       batch_size, 3, in_h, in_w   置信度真实值
        #   tcls        batch_size, 3, in_h, in_w, num_classes  种类真实值
        #---------------------------------------------------------------#
        mask, noobj_mask, tx, ty, tw, th, tconf, tcls, box_loss_scale_x, box_loss_scale_y =\
                                                                    self.get_target(targets, scaled_anchors,
```

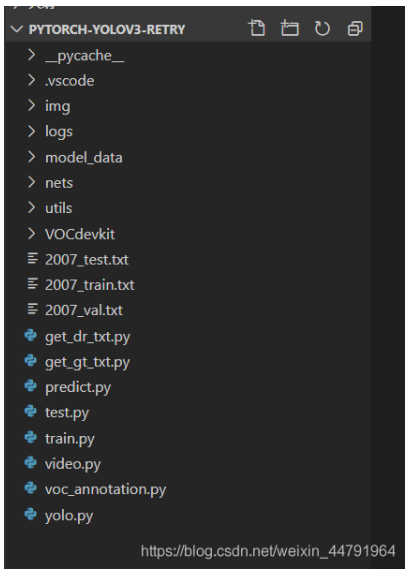
```
                                                                    in_w, in_h,
                                                                    self.ignore_threshold)

        #---------------------------------------------------------------#
        #   将预测结果进行解码，判断预测结果和真实值的重合程度
        #   如果重合程度过大则忽略，因为这些特征点属于预测比较准确的特征点
        #   作为负样本不合适
        #---------------------------------------------------------------#
        noobj_mask = self.get_ignore(prediction, targets, scaled_anchors, in_w, in_h, noobj_mask)

        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

        # 计算中心偏移情况的loss，使用BCELoss效果好一些
        loss_x = torch.sum(BCELoss(x, tx) * box_loss_scale * mask)
        loss_y = torch.sum(BCELoss(y, ty) * box_loss_scale * mask)
        # 计算宽高调整值的loss
        loss_w = torch.sum(MSELoss(w, tw) * 0.5 * box_loss_scale * mask)
        loss_h = torch.sum(MSELoss(h, th) * 0.5 * box_loss_scale * mask)
        # 计算置信度的loss
        loss_conf = torch.sum(BCELoss(conf, mask) * mask) + \
                    torch.sum(BCELoss(conf, mask) * noobj_mask)

        loss_cls = torch.sum(BCELoss(pred_cls[mask == 1], tcls[mask == 1]))

        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

        # 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))
        if self.normalize:
            num_pos = torch.sum(mask)
            num_pos = torch.max(num_pos, torch.ones_like(num_pos))
        else:
            num_pos = bs/3
        return loss, num_pos

    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)
        #-----------------------------------------------------#
        #   获得当前特征层先验框所属的编号，方便后面对先验框筛选
        #-----------------------------------------------------#
        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=False)
        noobj_mask = torch.ones(bs, int(self.num_anchors/3), in_h, in_w, requires_grad=False)

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

        box_loss_scale_x = torch.zeros(bs, int(self.num_anchors/3), in_h, in_w, requires_grad=False)
        box_loss_scale_y = torch.zeros(bs, int(self.num_anchors/3), in_h, in_w, requires_grad=False)
        for b in range(bs):
            if len(target[b])==0:
                continue
            #-----------------------------------------------------#
            #   计算出正样本在特征层上的中心点
            #-----------------------------------------------------#
            gxs = target[b][:, 0:1] * in_w
            gys = target[b][:, 1:2] * in_h

            #-----------------------------------------------------#
            #   计算出正样本相对于特征层的宽高
            #-----------------------------------------------------#
            gws = target[b][:, 2:3] * in_w
            ghs = target[b][:, 3:4] * in_h

            #-----------------------------------------------------#
            #   计算出正样本属于特征层的哪个特征点
            #-----------------------------------------------------#
            gis = torch.floor(gxs)
            gjs = torch.floor(gys)

            #-----------------------------------------------------#
            #   将真实框转换一个形式
            #   num_true_box, 4
            #-----------------------------------------------------#
            gt_box = torch.FloatTensor(torch.cat([torch.zeros_like(gws), torch.zeros_like(ghs), gws, ghs], 1))

            #-----------------------------------------------------#
            #   将先验框转换一个形式
            #   9, 4
            #-----------------------------------------------------#
            anchor_shapes = torch.FloatTensor(torch.cat((torch.zeros((self.num_anchors, 2)), torch.FloatTensor(anchors)), 1))
            #-----------------------------------------------------#
            #   计算交并比
            #   num_true_box, 9
            #-----------------------------------------------------#
            anch_ious = jaccard(gt_box, anchor_shapes)

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

```
            #   计算重合度最大的先验框是哪个
            #   num_true_box,
            #-------------------------------------------------------#
            best_ns = torch.argmax(anch_ious,dim=-1)
            for i, best_n in enumerate(best_ns):
                if best_n not in anchor_index:
                    continue
                #----------------------------------------------------------#
                #   取出各类坐标：
                #   gi和gj代表的是真实框对应的特征点的x轴y轴坐标
                #   gx和gy代表真实框的x轴和y轴坐标
                #   gw和gh代表真实框的宽和高
                #----------------------------------------------------------#
                gi = gis[i].long()
                gj = gjs[i].long()
                gx = gxs[i]
                gy = gys[i]
                gw = gws[i]
                gh = ghs[i]

                if (gj < in_h) and (gi < in_w):
                    best_n = best_n - subtract_index

                    #--------------------------------------#
                    #   noobj_mask代表无目标的特征点
                    #--------------------------------------#
                    noobj_mask[b, best_n, gj, gi] = 0
                    #--------------------------------------#
                    #   mask代表有目标的特征点
                    #--------------------------------------#
                    mask[b, best_n, gj, gi] = 1
                    #--------------------------------------#
                    #   tx、ty代表中心调整参数的真实值
                    #--------------------------------------#
                    tx[b, best_n, gj, gi] = gx - gi.float()
                    ty[b, best_n, gj, gi] = gy - gj.float()
                    #--------------------------------------#
                    #   tw、th代表宽高调整参数的真实值
                    #--------------------------------------#
                    tw[b, best_n, gj, gi] = math.log(gw / anchors[best_n+subtract_index][0])
                    th[b, best_n, gj, gi] = math.log(gh / anchors[best_n+subtract_index][1])
                    #--------------------------------------#
                    #   用于获得xywh的比例
                    #   大目标loss权重小，小目标loss权重大
                    #--------------------------------------#
                    box_loss_scale_x[b, best_n, gj, gi] = target[b][i, 2]
                    box_loss_scale_y[b, best_n, gj, gi] = target[b][i, 3]
                    #--------------------------------------#
                    #   tconf代表物体置信度
                    #--------------------------------------#
                    tconf[b, best_n, gj, gi] = 1
                    #--------------------------------------#
                    #   tcls代表种类置信度
                    #--------------------------------------#
                    tcls[b, best_n, gj, gi, int(target[b][i, 4])] = 1
                else:
                    print('Step {0} out of bound'.format(b))
                    print('gj: {0}, height: {1} | gi: {2}, width: {3}'.format(gj, in_h, gi, in_w))
                    continue

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

    def get_ignore(self,prediction,target,scaled_anchors,in_w, in_h,noobj_mask):
        #-------------------------------------------------------#
        #   计算一共有多少张图片
        #-------------------------------------------------------#
        bs = len(target)
        #-------------------------------------------------------#
        #   获得当前特征层先验框所属的编号，方便后面对先验框筛选
        #-------------------------------------------------------#
        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]

        # 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_h, 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_w, 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]
            #-------------------------------------------------------#
```

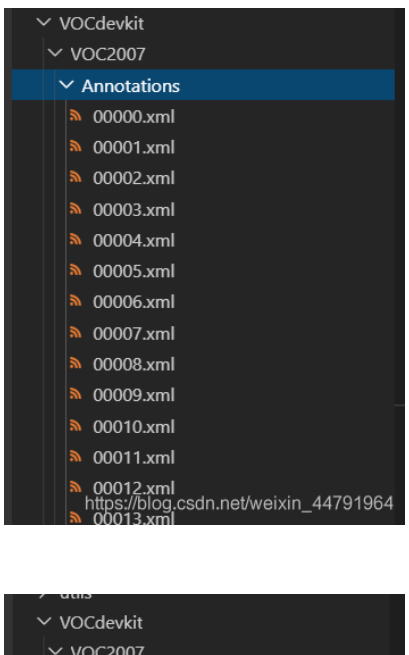
```
            #   将预测结果转换一个形式
            #   pred_boxes_for_ignore      num_anchors, 4
            #-------------------------------------------------------#
            pred_boxes_for_ignore = pred_boxes_for_ignore.view(-1, 4)
            #-------------------------------------------------------#
            #   计算真实框，并把真实框转换成相对于特征层的大小
            #   gt_box      num_true_box, 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(torch.cat([gx, gy, gw, gh],-1)).type(FloatTensor)

                #-------------------------------------------------------#
                #   计算交并比
                #   anch_ious       num_true_box, num_anchors
                #-------------------------------------------------------#
                anch_ious = jaccard(gt_box, pred_boxes_for_ignore)
                #-------------------------------------------------------#
                #   每个先验框对应真实框的最大重合度
                #   anch_ious_max   num_anchors
                #-------------------------------------------------------#
                anch_ious_max, _ = torch.max(anch_ious,dim=0)
                anch_ious_max = anch_ious_max.view(pred_boxes[i].size()[:3])
                noobj_mask[i][anch_ious_max>self.ignore_threshold] = 0
        return noobj_mask
```

训练自己的yolo3模型

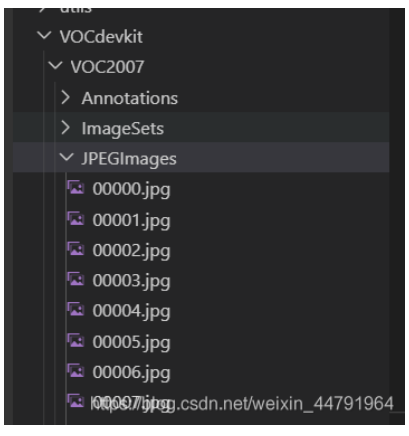
yolo3整体的文件夹构架如下：



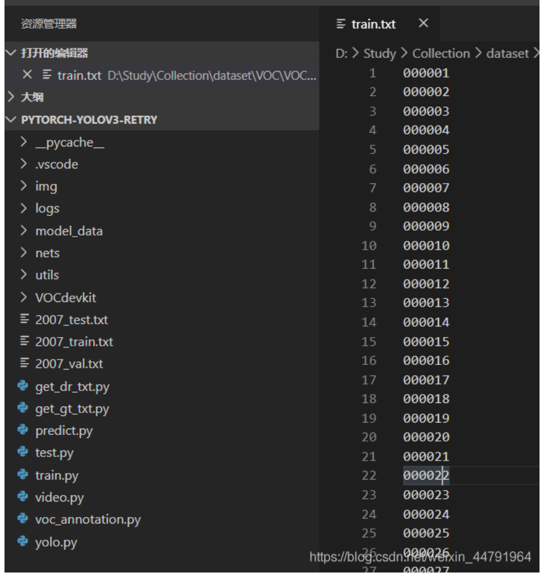
本文使用VOC格式进行训练。
训练前将标签文件放在VOCdevkit文件夹下的VOC2007文件夹下的Annotation中。



训练前将图片文件放在VOCdevkit文件夹下的VOC2007文件夹下的JPEGImages中。

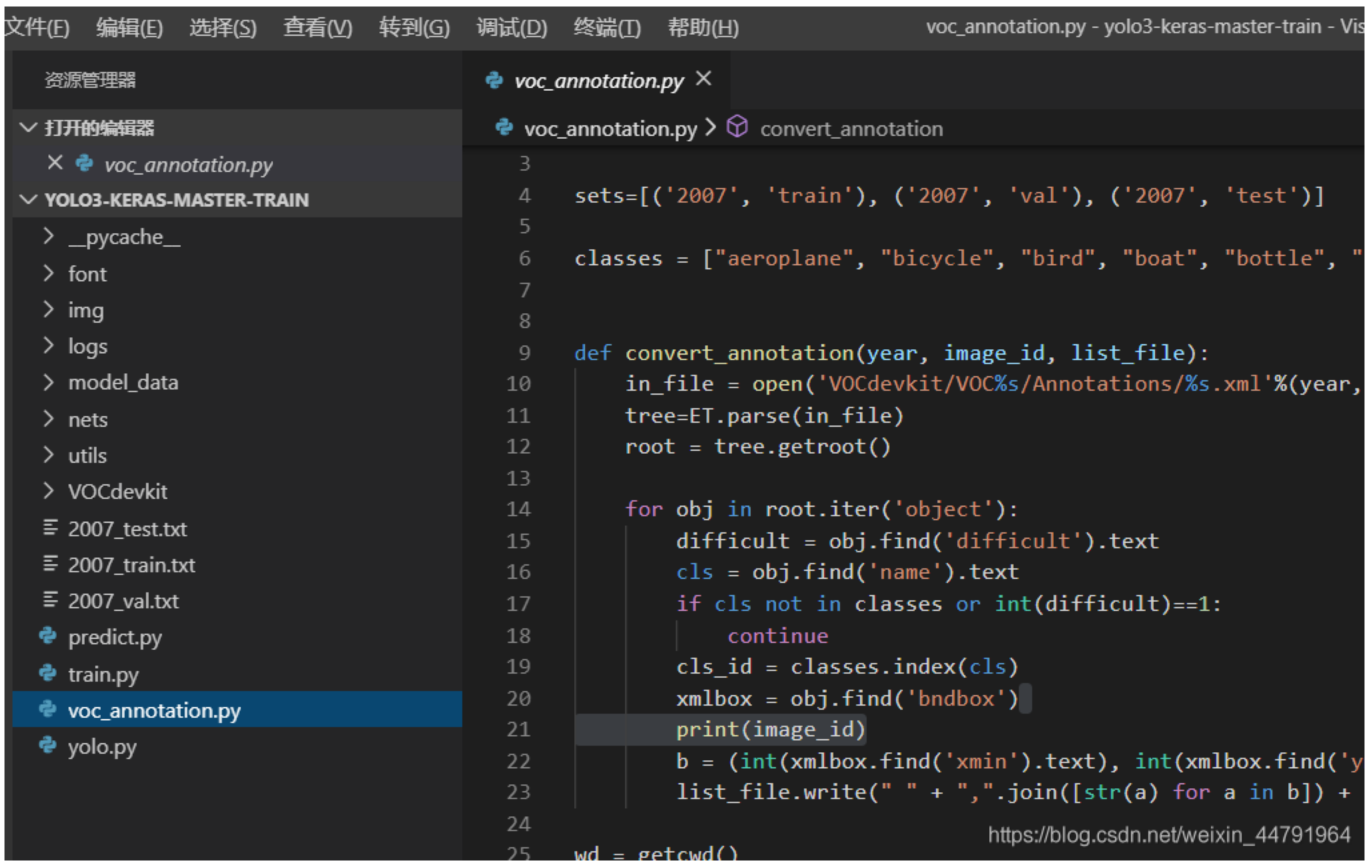


在训练前利用voc2yolo3.py文件生成对应的txt。

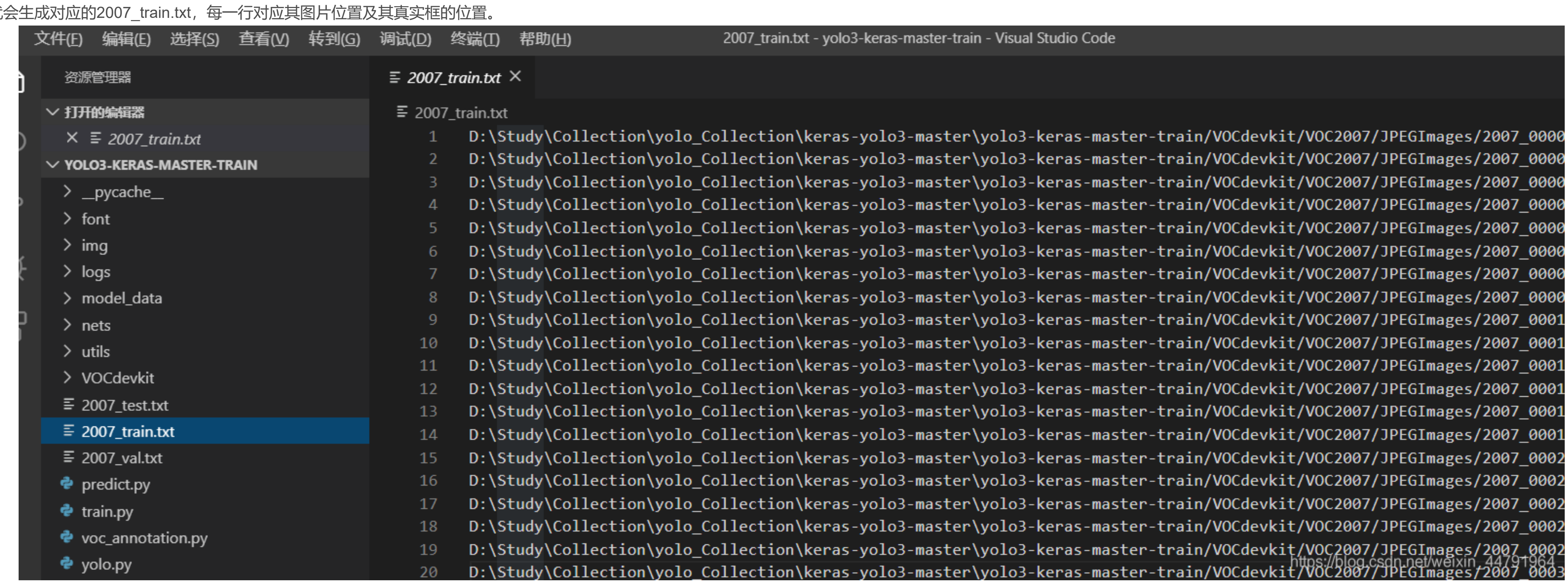


再运行根目录下的voc_annotation.py，运行前需要将classes改成你自己的classes。

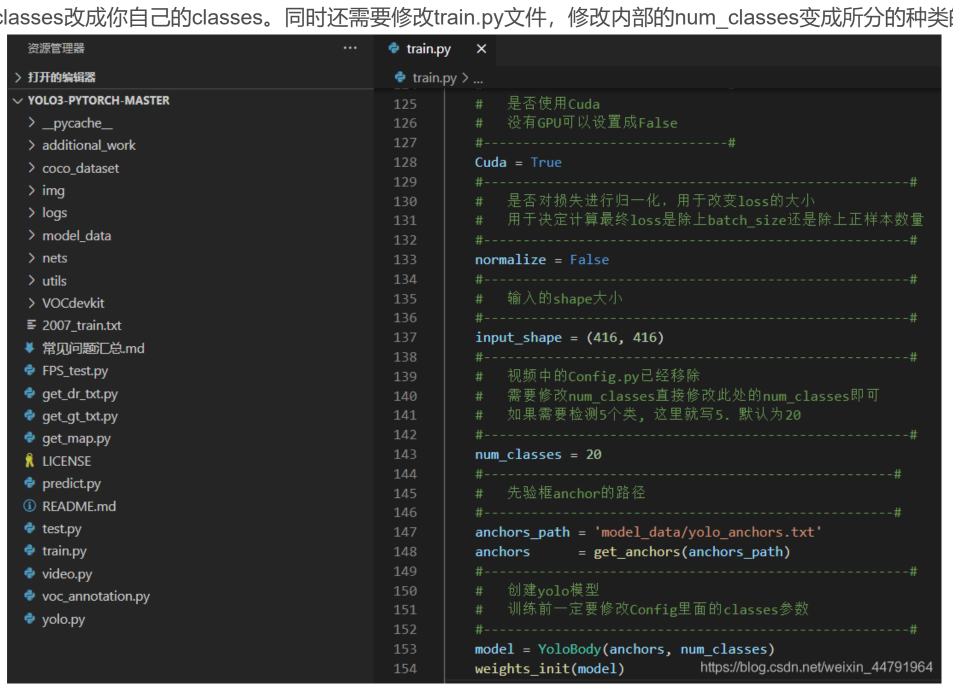
```
classes = ["aeroplane", "bicycle", "bird", "boat", "bottle", "bus", "car", "cat", "chair", "cow", "diningtable", "dog", "horse", "motorbike", "person", "pottedplant", "sheep", "sofa", "train", "tvmonitor"]
```



就会生成对应的2007_train.txt，每一行对应其图片位置及其真实框的位置。



在训练前需要修改model_data里面的voc_classes.txt文件，需要将classes改成你自己的classes。同时还需要修改train.py文件，修改内部的num_classes变成所分的种类的数量。



运行train.py即可开始训练。

```
Epoch:1/25
iter:5/2419 || Total Loss: 8496.1523 || 1.2869s/step
```

相关推荐