
从零开始学习 **YOLOv3** 系列合集

GiantPandaCV 公众号

GiantPandaCV-pprp

2020-03-30

零、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 95 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。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。

欢迎大家关注我们的公众号 GiantPandaCV:



一、YOLO cfg 文件解析

与其他框架不同，Darknet 构建网络架构不是通过代码直接堆叠，而是通过解析 cfg 文件进行生成的。cfg 文件格式是有一定规则，虽然比较简单，但是有些地方需要对 yolov3 有一定程度的熟悉，才能正确设置。

下边以 **yolov3.cfg** 为例进行讲解。

1. Net 层

```
[net]
#Testing
```

```
#batch=1
#subdivisions=1
# 在测试的时候, 设置 batch=1,subdivisions=1
#Training
batch=16
subdivisions=4
# 这里的 batch 与普遍意义上的 batch 不一致的。
# 训练的过程中将一次性加载 16 张图片进内存, 然后分 4 次完成前向传播, 每次 4 张。
# 经过 16 张图片的前向传播以后, 进行一次反向传播。
width=416
height=416
channels=3
# 设置图片进入网络的宽、高和通道个数。
# 由于 YOLOv3 的下采样一般是 32 倍, 所以宽高必须能被 32 整除。
# 多尺度训练选择为 32 的倍数最小 320*320, 最大 608*608。
# 长和宽越大, 对小目标越好, 但是占用显存也会高, 需要权衡。
momentum=0.9
# 动量参数影响着梯度下降到最优值得速度。
decay=0.0005
# 权重衰减正则项, 防止过拟合。
angle=0
# 数据增强, 设置旋转角度。
saturation = 1.5
# 饱和度
exposure = 1.5
# 曝光量
hue=.1
# 色调

learning_rate=0.001
# 学习率: 刚开始训练时可以将学习率设置的高一点, 而一定轮数之后, 将其减小。
# 在训练过程中, 一般根据训练轮数设置动态变化的学习率。
burn_in=1000
max_batches = 500200
# 最大 batch
policy=steps
# 学习率调整的策略, 有以下 policy:
#constant, steps, exp, poly, step, sig, RANDOM, constant 等方式
# 调整学习率的 policy,
# 有如下 policy: constant, steps, exp, poly, step, sig, RANDOM。
#steps# 比较好理解, 按照 steps 来改变学习率。

steps=400000,450000
```

```
scales=.1,.1
# 在达到 40000、45000 的时候将学习率乘以对应的 scale
```

2. 卷积层

```
[convolutional]
batch_normalize=1
# 是否做 BN 操作
filters=32
# 输出特征图的数量
size=3
# 卷积核的尺寸
stride=1
# 做卷积运算的步长
pad=1
# 如果 pad 为 0, padding 由 padding 参数指定。
# 如果 pad 为 1, padding 大小为 size/2, padding 应该是对输入图像左边缘拓展的像素数量
activation=leaky
# 激活函数的类型: logistic, loggy, relu,
#elu, relie, plse, hardtan, lhtan,
#linear, ramp, leaky, tanh, stair
# alexeyAB 版添加了 mish, swish, nrom_chan 等新的激活函数
```

feature map 计算公式:

$$OutFeature = \frac{InFeature + 2 \times padding - size}{stride} + 1$$

3. 下采样

可以通过调整卷积层参数进行下采样:

```
[convolutional]
batch_normalize=1
filters=128
size=3
stride=2
pad=1
activation=leaky
```

可以通过带入以上公式, 可以得到 OutFeature 是 InFeature 的一半。

也可以使用 maxpooling 进行下采样:

```
[maxpool]
size=2
stride=2
```

4. 上采样

```
[upsample]
stride=2
```

上采样是通过线性插值实现的。

5. Shortcut 和 Route 层

```
[shortcut]
from=-3
activation=linear
#shortcut 操作是类似 ResNet 的跨层连接, 参数 from 是 -3,
# 意思是 shortcut 的输出是当前层与先前的倒数第三层相加而得到。
# 通俗来讲就是 add 操作
```

```
[route]
layers = -1, 36
# 当属性有两个值, 就是将上一层和第 36 层进行 concatenate
# 即沿深度的维度连接, 这也要求 feature map 大小是一致的。
```

```
[route]
layers = -4
# 当属性只有一个值时, 它会输出由该值索引的网络层的特征图。
# 本例子中就是提取从当前倒数第四个层输出
```

6. YOLO 层

```
[convolutional]
size=1
stride=1
pad=1
filters=18
# 每一个 [region/yolo] 层前的最后一个卷积层中的
# filters=num(yolo 层个数)*(classes+5), 5 的意义是 5 个坐标,
# 代表论文中的 tx, ty, tw, th, po
# 这里类别个数为 1, (1+5)*3=18
activation=linear
```

```
[yolo]
mask = 6,7,8
# 训练框 mask 的值是 0,1,2,
# 这意味着使用第一, 第二和第三个 anchor
anchors = 10,13, 16,30, 33,23, 30,61, 62,45,\
          59,119, 116,90, 156,198, 373,326
# 总共有三个检测层, 共计 9 个 anchor
# 这里的 anchor 是由 kmeans 聚类算法得到的。
classes=1
# 类别个数
num=9
# 每个 grid 预测的 BoundingBox num/yolo 层个数
jitter=.3
# 利用数据抖动产生更多数据,
# 属于 TTA (Test Time Augmentation)
ignore_thresh = .5
# ignore_thresh 指的是参与计算的 IOU 阈值大小。
# 当预测的检测框与 ground true 的 IOU 大于 ignore_thresh 的时候,
# 不会参与 loss 的计算, 否则, 检测框将会参与损失计算。
# 目的是控制参与 loss 计算的检测框的规模, 当 ignore_thresh 过于大,
# 接近于 1 的时候, 那么参与检测框回归 loss 的个数就会比较少, 同时也容易造成过拟合;
# 而如果 ignore_thresh 设置的过于小, 那么参与计算的会数量规模就会很大。
# 同时也容易在进行检测框回归的时候造成欠拟合。
# ignore_thresh 一般选取 0.5-0.7 之间的一个值
# 小尺度 (13*13) 用的是 0.7,
# 大尺度 (26*26) 用的是 0.5。
```

7. 模块总结

Darknet-53 结构如下图所示:

	Type	Filters	Size	Output
	Convolutional	32	3×3	256×256
	Convolutional	64	$3 \times 3 / 2$	128×128
1×	Convolutional	32	1×1	
	Convolutional	64	3×3	
	Residual			128×128
	Convolutional	128	$3 \times 3 / 2$	64×64
2×	Convolutional	64	1×1	
	Convolutional	128	3×3	
	Residual			64×64
	Convolutional	256	$3 \times 3 / 2$	32×32
8×	Convolutional	128	1×1	
	Convolutional	256	3×3	
	Residual			32×32
	Convolutional	512	$3 \times 3 / 2$	16×16
8×	Convolutional	256	1×1	
	Convolutional	512	3×3	
	Residual			16×16
	Convolutional	1024	$3 \times 3 / 2$	8×8
4×	Convolutional	512	1×1	
	Convolutional	1024	3×3	
	Residual			8×8
	Avgpool		Global	
	Connected		1000	
	Softmax			

https://blog.csdn.net/DD_PP_JJ

它是由重复的类似于 ResNet 的模块组成的，其下采样是通过卷积来完成的。通过对 cfg 文件的观察，提出了以下总结：

不改变 **feature** 大小的模块：

1. 残差模块：

```
[convolutional]
batch_normalize=1
filters=128
size=1
stride=1
pad=1
activation=leaky
```

```
[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky
```

```
[shortcut]
from=-3
activation=linear
```

2. 1x1 卷积：可以降低计算量

```
[convolutional]
batch_normalize=1
filters=512
size=1
stride=1
pad=1
activation=leaky
```

3. 普通 3x3 卷积：可以对 filter 个数进行调整

```
[convolutional]
batch_normalize=1
filters=1024
size=3
stride=1
pad=1
activation=leaky
```

改变 **feature map** 大小

1. feature map 减半：

```
[maxpool]
```



```
size=2  
stride=2
```

或者

```
[convolutional]  
batch_normalize=1  
filters=128  
size=3  
stride=2  
pad=1  
activation=leaky
```

2. feature map 加倍:

```
[maxpool]  
size=2  
stride=1
```

特征融合操作

1. 使用 Route 层获取指定的层（13×13）。
2. 添加卷积层进行学习但不改变 feature map 大小。
3. 进行上采样（26×26）。
4. 从 backbone 中找到对应 feature map 大小的层进行 Route 或者 Shortcut（26×26）。
5. 融合完成。

后记：以上就是笔者之前使用 darknet 过程中收集和总结的一些经验，掌握以上内容并读懂 yolov3 论文后，就可以着手运行代码了。目前使用与 darknet 一致的 cfg 文件解析的有一些，比如原版 Darknet，AlexeyAB 版本的 Darknet，还有一个 pytorch 版本的 yolov3。AlexeyAB 版本的添加了很多新特性，比如 [conv_lstm], [scale_channels] SE/ASFF/BiFPN, [local_avgpool], [sam], [Gaussian_yolo], [reorg3d] (fixed [reorg]), fixed [batchnorm] 等等。而 pytorch 版本的 yolov3 可以很方便的添加我们需要的功能。之后将会对这个版本进行改进，添加空洞卷积、SE、CBAM、SK 等模块。

二、代码配置和数据集处理

前言：本文是讲的是如何配置 pytorch 版本的 yolov3、数据集处理、常用的命令等内容。该库的数据集格式既不是 VOC2007 格式也不是 MS COCO 的格式，而是一种新的格式，跟着文章一步一步来，很简单。另外我们公众号针对 VOC2007 格式数据集转化为本库所需要格式特意开发了一个简单的数据处理库。

1. 环境搭建

1. 将 github 库 download 下来。

```
git clone https://github.com/ultralytics/yolov3.git
```

2. 建议在 linux 环境下使用 anaconda 进行搭建

```
conda create -n yolov3 python=3.7
```

3. 安装需要的软件

```
pip install -r requirements.txt
```

环境要求：

- python >= 3.7
- pytorch >= 1.1
- numpy
- tqdm
- opencv-python

其中只需要注意 pytorch 的安装：

到 <https://pytorch.org/> 中根据操作系统，python 版本，cuda 版本等选择命令即可。

2. 数据集构建

2.1 xml 文件生成需要 Labelimg 软件

在 Windows 下使用 LabelImg 软件进行标注：

- 使用快捷键：

Ctrl + u 加载目录中的所有图像，鼠标点击 Open dir 同功能

Ctrl + r 更改默认注释目标目录(xml文件保存的地址)

Ctrl + s 保存

Ctrl + d 复制当前标签和矩形框

space 将当前图像标记为已验证

w 创建一个矩形框

d	下一张图片
a	上一张图片
del	删除选定的矩形框
Ctrl++	放大
Ctrl--	缩小
↑→↓←	键盘箭头移动选定的矩形框

2.2 VOC2007 数据集格式

```
-data
  - VOCdevkit2007
    - VOC2007
      - Annotations (标签XML文件，用对应的图片处理工具人工生成的)
      - ImageSets (生成的方法是用sh或者MATLAB语言生成)
        - Main
          - test.txt
          - train.txt
          - trainval.txt
          - val.txt
      - JPEGImages(原始文件)
      - labels (xml文件对应的txt文件)
```

通过以上软件主要构造好 JPEGImages 和 Annotations 文件夹中内容,Main 文件夹中的 txt 文件可以通过以下 python 脚本生成:

```
import os
import random

trainval_percent = 0.9
train_percent = 1
xmlfilepath = 'Annotations'
txtsavepath = 'ImageSets/Main'
total_xml = os.listdir(xmlfilepath)

num=len(total_xml)
list=range(num)
tv=int(num*trainval_percent)
tr=int(tv*train_percent)
```

```
trainval= random.sample(list,tv)
train=random.sample(trainval,tr)

ftrainval = open('ImageSets/Main/trainval.txt', 'w')
ftest = open('ImageSets/Main/test.txt', 'w')
ftrain = open('ImageSets/Main/train.txt', 'w')
fval = open('ImageSets/Main/val.txt', 'w')

for i in list:
    name=total_xml[i][:-4]+'\\n'
    if i in trainval:
        ftrainval.write(name)
        if i in train:
            ftrain.write(name)
        else:
            fval.write(name)
    else:
        ftest.write(name)

ftrainval.close()
ftrain.close()
fval.close()
ftest.close()
```

接下来生成 labels 文件夹中的 txt 文件，voc_label.py 文件具体内容如下：

```
# -*- coding: utf-8 -*-
"""
Created on Tue Oct  2 11:42:13 2018
将本文件放到 VOC2007 目录下，然后就可以直接运行
需要修改的地方：
1. sets 中替换为自己的数据集
2. classes 中替换为自己的类别
3. 将本文件放到 VOC2007 目录下
4. 直接开始运行
"""

import xml.etree.ElementTree as ET
import pickle
import os
```

```

from os import listdir, getcwd
from os.path import join
sets=[('2007', 'train'), ('2007', 'val'), ('2007', 'test')] # 替换为自己的数据集
↪ 集
classes = ["person"] # 修改为自己的类别

# 进行归一化
def convert(size, box):
    dw = 1./(size[0])
    dh = 1./(size[1])
    x = (box[0] + box[1])/2.0 - 1
    y = (box[2] + box[3])/2.0 - 1
    w = box[1] - box[0]
    h = box[3] - box[2]
    x = x*dw
    w = w*dw
    y = y*dh
    h = h*dh
    return (x,y,w,h)

def convert_annotation(year, image_id):
    in_file = open('VOC%s/Annotations/%s.xml'%(year, image_id)) # 将数据集放
    ↪ 于当前目录下
    out_file = open('VOC%s/labels/%s.txt'%(year, image_id), 'w')
    tree=ET.parse(in_file)
    root = tree.getroot()
    size = root.find('size')
    w = int(size.find('width').text)
    h = int(size.find('height').text)
    for obj in root.iter('object'):
        difficult = obj.find('difficult').text
        cls = obj.find('name').text
        if cls not in classes or int(difficult)==1:
            continue
        cls_id = classes.index(cls)
        xmlbox = obj.find('bndbox')
        b = (float(xmlbox.find('xmin').text),
    ↪ float(xmlbox.find('xmax').text), float(xmlbox.find('ymin').text),
    ↪ float(xmlbox.find('ymax').text))
        bb = convert((w,h), b)
        out_file.write(str(cls_id) + " " + " ".join([str(a) for a in bb]) +
    ↪ '\n')
wd = getcwd()

```

```
for year, image_set in sets:
    if not os.path.exists('VOC%s/labels/'%(year)):
        os.makedirs('VOC%s/labels/'%(year))
    image_ids = open('VOC%s/ImageSets/Main/%s.txt'%(year,
↪ image_set)).read().strip().split()
    list_file = open('%s_%s.txt'%(year, image_set), 'w')
    for image_id in image_ids:
        list_file.write('VOC%s/JPEGImages/%s.jpg\n'%(year, image_id))
        convert_annotation(year, image_id)
    list_file.close()
```

到底为止，VOC 格式数据集构造完毕，但是还需要继续构造符合 darknet 格式的数据集 (coco)。

需要说明的是：如果打算使用 coco 评价标准，需要构造 coco 中 json 格式，如果要求不高，只需要 VOC 格式即可，使用作者写的 mAP 计算程序即可。

2.3 创建 *.names file,

其中保存的是你的所有的类别，每行一个类别，如 data/coco.names:

person

2.4 更新 data/coco.data, 其中保存的是很多配置信息

```
classes = 1 # 改成你的数据集的类别个数
train = ./data/2007_train.txt # 通过voc_label.py文件生成的txt文件
valid = ./data/2007_test.txt # 通过voc_label.py文件生成的txt文件
names = data/coco.names # 记录类别
backup = backup/ # 在本库中没有用到
eval = coco # 选择map计算方式
```

2.5 更新 cfg 文件，修改类别相关信息

打开 cfg 文件夹下的 yolov3.cfg 文件，大体而言，cfg 文件记录的是整个网络的结构，是核心部分，具体内容讲解请参考之前的文章：【从零开始学习 YOLOv3】1. YOLOv3 的 cfg 文件解析与总结

只需要更改每个 [yolo] 层前边卷积层的 filter 个数即可：

每一个 [region/yolo] 层前的最后一个卷积层中的 filters= 预测框的个数 (mask 对应的个数, 比如 mask=0,1,2, 代表使用了 anchors 中的前三对, 这里预测框个数就应该是 $3 \times (\text{classes} + 5)$, 5 的意义是 5 个坐标 (论文中的 tx,ty,tw,th,po), 3 的意义就是用了 3 个 anchor。

举个例子: 假如我有三个类, $n = 3$, 那么 $\text{filter} = 3 \times (n + 5) = 24$

```
[convolutional]
```

```
size=1
```

```
stride=1
```

```
pad=1
```

```
filters=255 # 改为 24
```

```
activation=linear
```

```
[yolo]
```

```
mask = 6,7,8
```

```
anchors = 10,13, 16,30, 33,23, 30,61, 62,45, 59,119, 116,90, 156,198, 373,326
```

```
classes=80 # 改为 3
```

```
num=9
```

```
jitter=.3
```

```
ignore_thresh = .7
```

```
truth_thresh = 1
```

```
random=1
```

2.6 数据集格式说明

- yolov3

- data

- 2007_train.txt

- 2007_test.txt

- coco.names

- coco.data

- annotations(json files)

- images(将2007_train.txt中的图片放到train2014文件夹中, test同理)

- train2014

- 0001.jpg

- 0002.jpg

- val2014
 - 0003.jpg
 - 0004.jpg
- labels (voc_labels.py生成的内容需要重新组织一下)
 - train2014
 - 0001.txt
 - 0002.txt
 - val2014
 - 0003.txt
 - 0004.txt
- samples(存放待测试图片)

2007_train.txt 内容示例:

```
/home/dpj/yolov3-master/data/images/val2014/Cow_1192.jpg
/home/dpj/yolov3-master/data/images/val2014/Cow_1196.jpg
.....
```

注意 images 和 labels 文件架构一致性, 因为 txt 是通过简单的替换得到的:

```
images -> labels
.jpg -> .txt
```

具体内容可以在 datasets.py 文件中找到详细的替换。

3. 训练模型

预训练模型:

- Darknet*.weightsformat: <https://pjreddie.com/media/files/yolov3.weights>
- PyTorch*.ptformat: <https://drive.google.com/drive/folders/1uxgUBemJVw9wZsdpboYb>

开始训练:

```
python train.py --data data/coco.data --cfg cfg/yolov3.cfg
```

如果日志正常输出那证明可以运行了

221	105.conv_105.bias					true	24	
Model Summary: 222 layers, 6.15345e+07 parameters, 6.15345e+07 gradients								
Epoch	Batch	xy	wh	conf	cls	total	nTargets	time
0/99	0/288	0.201	1.35	135	0.824	138	22	12
0/99	1/288	0.233	1.31	135	0.818	138	20	2.92
0/99	2/288	0.242	1.34	135	0.838	138	38	1.97
0/99	3/288	0.242	1.31	135	0.858	138	26	1.87
0/99	4/288	0.25	1.38	135	0.858	138	16	2
0/99	5/288	0.246	1.35	135	0.855	138	22	2.5
0/99	6/288	0.248	1.33	135	0.852	138	22	2.42

如果中断了，可以恢复训练

```
python train.py --data data/coco.data --cfg cfg/yolov3.cfg --resume
```

4. 测试模型

将待测试图片放到 data/samples 中，然后运行

```
python detect.py --cfg cfg/yolov3.cfg --weights weights/best.pt
```

目前该文件中也可以放入视频进行视频目标检测。



- Image: --source file.jpg
- Video: --source file.mp4
- Directory: --source dir/

- Webcam: `--source 0`
- RTSP stream: `--source rtsp://170.93.143.139/rtplive/470011e600ef003a004ee3369623`
- HTTP stream: `--source http://wmccpinetop.axiscam.net/mjpg/video.mjpg`

5. 评估模型

```
python test.py --weights weights/best.pt
```

如果使用 cocoAPI 使用以下命令：

```
$ python3 test.py --img-size 608 --iou-thr 0.6 --weights ultralytics68.pt -  
-cfg yolov3-spp.cfg
```

```
Namespace(batch_size=32, cfg='yolov3-spp.cfg', conf_thres=0.001, data='data/coco2014_val',  
Using CUDA device0 _CudaDeviceProperties(name='Tesla V100-SXM2-16GB', total_memory=16130MB)
```

```

      Class  Images  Targets      P      R  mAP@0.5      F1: 100% 157/157 [03:00.00]
      all    5e+03  3.51e+04  0.0353  0.891  0.606  0.0673
Average Precision (AP) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.409
Average Precision (AP) @[ IoU=0.50      | area= all | maxDets=100 ] = 0.615
Average Precision (AP) @[ IoU=0.75      | area= all | maxDets=100 ] = 0.437
Average Precision (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.242
Average Precision (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.448
Average Precision (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.519
Average Recall    (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 1 ] = 0.337
Average Recall    (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 10 ] = 0.557
Average Recall    (AR) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.612
Average Recall    (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.438
Average Recall    (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.658
Average Recall    (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.746

```

mAP 计算

- mAP@0.5 run at `--iou-thr 0.5`, mAP@0.5...0.95 run at `--iou-thr 0.7`

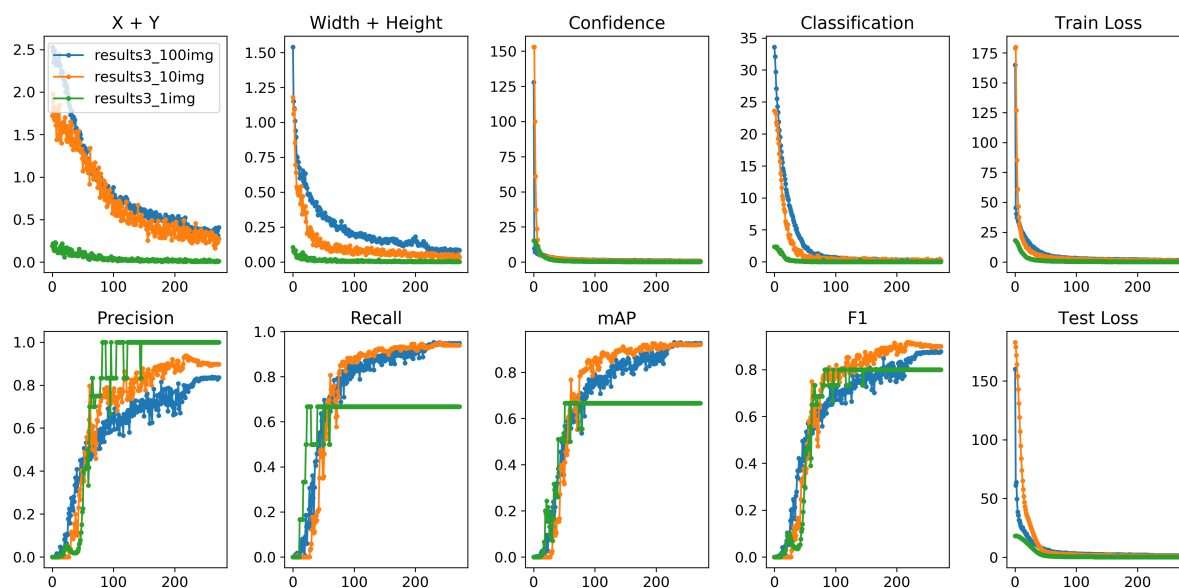
6. 可视化

可以使用 `python -c from utils import utils;utils.plot_results()`

创建 drawLog.py

```
def plot_results():
    # Plot YOLO training results file 'results.txt'
    import glob
    import numpy as np
    import matplotlib.pyplot as plt
    #import os; os.system('rm -rf results.txt && wget
    ↪ https://storage.googleapis.com/ultralytics/results_v1_0.txt')

    plt.figure(figsize=(16, 8))
    s = ['X', 'Y', 'Width', 'Height', 'Objectness', 'Classification', 'Total
    ↪ Loss', 'Precision', 'Recall', 'mAP']
    files = sorted(glob.glob('results.txt'))
    for f in files:
        results = np.loadtxt(f, usecols=[2, 3, 4, 5, 6, 7, 8, 17, 18, 16]).T
    ↪ # column 16 is mAP
        n = results.shape[1]
        for i in range(10):
            plt.subplot(2, 5, i + 1)
            plt.plot(range(1, n), results[i, 1:], marker='.', label=f)
            plt.title(s[i])
            if i == 0:
                plt.legend()
        plt.savefig('./plot.png')
if __name__ == "__main__":
    plot_results()
```



7. 数据集配套代码

如果你看到这里了，恭喜你，你可以避开以上略显复杂的数据处理。我们提供了一套代码，集成了以上脚本，只需要你有 jpg 图片和对应的 xml 文件，就可以直接生成符合要求的数据集，然后按照要求修改一些代码即可。

代码地址：https://github.com/pprp/voc2007_for_yolo_torch

请按照 `readme` 中进行处理就可以得到数据集。

后记：这套代码一直由一个外国的团队进行维护，也添加了很多新的 `trick`。目前已获得了 3.3k 个 `star`，1k `fork`。不仅如此，其团队会经常回复 `issue`，目前也有接近 1k 的 `issue`。只要处理过一遍数据，就会了解到这个库的亮点，非常容易配置，不需要进行编译等操作，易用性极强。再加上提供的配套数据处理代码，在短短 10 多分钟就可以配置好。(๑ω๑)

这是这个系列第二篇内容，之后我们将对 `yolov3` 进行代码级别的学习，也会学习一下这个库提供的新的特性，比如说超参数进化，权重采样机制、`loss` 计算、`Giou` 处理等。希望各位多多关注。

参考内容：

官方代码：<https://github.com/ultralytics/yolov3>

官方讲解：<https://github.com/ultralytics/yolov3/wiki/Train-Custom-Data>

数据集配置库：https://github.com/pprp/voc2007_for_yolo_torch

三、YOLOv3 的数据组织与处理

前言：本文主要讲 YOLOv3 中数据加载部分，主要解析的代码在 `utils/datasets.py` 文件中。通过对数据组织、加载、处理部分代码进行解读，能帮助我们更快地理解 YOLOv3 所要求的数据输出要求，也将有利于对之后训练部分代码进行理解。

1. 标注格式

在上一篇【从零开始学习 YOLOv3】2. YOLOv3 中的代码配置和数据集构建 中，使用到了 `voc_label.py`，其作用是将 xml 文件转成 txt 文件格式，具体文件如下：

```
# class id, x, y, w, h
0 0.8604166666666666 0.5403899721448469 0.058333333333333334
  ↪ 0.055710306406685235
```

其中的 `x,y` 的意义是归一化以后的框的中心坐标，`w,h` 是归一化后的框的宽和高。

具体的归一化方式为：

```
def convert(size, box):
    '''
    size 是图片的长和宽
    box 是 xmin,xmax,ymin,ymax 坐标值
    '''
    dw = 1. / (size[0])
    dh = 1. / (size[1])
    # 得到长和宽的缩放比
    x = (box[0] + box[1]) / 2.0 - 1
    y = (box[2] + box[3]) / 2.0 - 1
    w = box[1] - box[0]
    h = box[3] - box[2]
    # 分别计算中心点坐标，框的宽和高
    x = x * dw
    w = w * dw
    y = y * dh
    h = h * dh
    # 按照图片长和宽进行归一化
    return (x,y,w,h)
```

可以看出，归一化都是相对于图片的宽和高进行归一化的。

2. 调用

下边是 `train.py` 文件中的有关数据的调用：

```
# Dataset
dataset = LoadImagesAndLabels(train_path, img_size, batch_size,
                               augment=True,
                               hyp=hyp, # augmentation hyperparameters
                               rect=opt.rect, # rectangular training
                               cache_labels=True,
                               cache_images=opt.cache_images)

batch_size = min(batch_size, len(dataset))

# 使用多少个线程加载数据集
nw = min([os.cpu_count(), batch_size if batch_size > 1 else 0, 1])

dataloader = DataLoader(dataset,
                        batch_size=batch_size,
                        num_workers=nw,
                        shuffle=not opt.rect,
                        # Shuffle=True
                        #unless rectangular training is used
                        pin_memory=True,
                        collate_fn=dataset.collate_fn)
```

在 `pytorch` 中，数据集加载主要是重构 `datasets` 类，然后再使用 `dataloader` 中加载 `dataset`，就构建好了数据部分。

下面是一个简单的使用模板：

```
import os
from torch.utils.data import Dataset
from torch.utils.data import DataLoader

# 根据自己的数据集格式进行重构
class MyDataset(Dataset):
    def __init__(self):
        # 下载数据、初始化数据，都可以在这里完成
        xy = np.loadtxt('label.txt', delimiter=',', dtype=np.float32)
        # 使用 numpy 读取数据
        self.x_data = torch.from_numpy(xy[:, 0:-1])
        self.y_data = torch.from_numpy(xy[:, [-1]])
        self.len = xy.shape[0]
```

```

def __getitem__(self, index):
    # dataloader 中使用该方法, 通过 index 进行访问
    return self.x_data[index], self.y_data[index]

def __len__(self):
    # 查询数据集中数量, 可以通过 len(mydataset) 得到
    return self.len

# 实例化这个类, 然后我们就得到了 Dataset 类型的数据, 记下来就将这个类传给 DataLoader, 就可
# 以了。
myDataset = MyDataset()

# 构建 dataloader
train_loader = DataLoader(dataset=myDataset,
                           batch_size=32,
                           shuffle=True)

for epoch in range(2):
    for i, data in enumerate(train_loader2):
        # 将数据从 train_loader 中读出来, 一次读取的样本数是 32 个
        inputs, labels = data
        # 将这些数据转换成 Variable 类型
        inputs, labels = Variable(inputs), Variable(labels)
        # 模型训练...

```

通过以上模板就能大致了解 pytorch 中的数据加载机制, 下面开始介绍 YOLOv3 中的数据加载。

3. YOLOv3 中的数据加载

下面解析的是 LoadImagesAndLabels 类中的几个主要的函数:

3.1 init 函数

init 函数中包含了大部分需要处理的数据

```

class LoadImagesAndLabels(Dataset): # for training/testing
    def __init__(self,
                  path,
                  img_size=416,
                  batch_size=16,
                  augment=False,

```

```

        hyp=None,
        rect=False,
        image_weights=False,
        cache_labels=False,
        cache_images=False):
    path = str(Path(path)) # os-agnostic
    assert os.path.isfile(path), 'File not found %. See %s' % (path,
                                                                help_url)

    with open(path, 'r') as f:
        self.img_files = [
            x.replace('/', os.sep)
            for x in f.read().splitlines() # os-agnostic
            if os.path.splitext(x)[-1].lower() in img_formats
        ]
    # img_files 是一个 list, 保存的是图片的路径

    n = len(self.img_files)
    assert n > 0, 'No images found in %. See %s' % (path, help_url)
    bi = np.floor(np.arange(n) / batch_size).astype(np.int) # batch
    ↪ index
    # 如果 n=10, batch=2, bi=[0,0,1,1,2,2,3,3,4,4]
    nb = bi[-1] + 1 # 最多有多少个 batch

    self.n = n
    self.batch = bi # 图片的 batch 索引, 代表第几个 batch 的图片
    self.img_size = img_size
    self.augment = augment
    self.hyp = hyp
    self.image_weights = image_weights # 是否选择根据权重进行采样
    self.rect = False if image_weights else rect
    # 如果选择根据权重进行采样, 将无法使用矩形训练:
    # 具体内容见下文

    # 标签文件是通过 images 替换为 labels, .jpg 替换为.txt 得到的。
    self.label_files = [
        x.replace('images',
                  'labels').replace(os.path.splitext(x)[-1], '.txt')
        for x in self.img_files
    ]

    # 矩形训练具体内容见下文解析
    if self.rect:
        # 获取图片的长和宽 (wh)

```



```
sp = path.replace('.txt', '.shapes')
# 字符串替换
# shapefile path
try:
    with open(sp, 'r') as f: # 读取 shape 文件
        s = [x.split() for x in f.read().splitlines()]
        assert len(s) == n, 'Shapefile out of sync'
except:
    s = [
        exif_size(Image.open(f))
        for f in tqdm(self.img_files, desc='Reading image shapes')
    ]
    np.savetxt(sp, s, fmt='%g') # overwrites existing (if any)

# 根据长宽比进行排序
s = np.array(s, dtype=np.float64)
ar = s[:, 1] / s[:, 0] # aspect ratio
i = ar.argsort()

# 根据顺序重排顺序
self.img_files = [self.img_files[i] for i in i]
self.label_files = [self.label_files[i] for i in i]
self.shapes = s[i] # wh
ar = ar[i]

# 设置训练的图片形状
shapes = [[1, 1]] * nb
for i in range(nb):
    ari = ar[bi == i]
    mini, maxi = ari.min(), ari.max()
    if maxi < 1:
        shapes[i] = [maxi, 1]
    elif mini > 1:
        shapes[i] = [1, 1 / mini]

self.batch_shapes = np.ceil(
    np.array(shapes) * img_size / 32.).astype(np.int) * 32

# 预载标签
# weighted CE 训练时需要这个步骤
# 否则无法按照权重进行采样
self.imgs = [None] * n
self.labels = [None] * n
```

```

    if cache_labels or image_weights: # cache labels for faster training
        self.labels = [np.zeros((0, 5))] * n
        extract_bounding_boxes = False
        create_datasubset = False
        pbar = tqdm(self.label_files, desc='Caching labels')
        nm, nf, ne, ns, nd = 0, 0, 0, 0, 0 # number missing, found,
        ↪ empty, datasubset, duplicate
        for i, file in enumerate(pbar):
            try:
                # 读取每个文件内容
                with open(file, 'r') as f:
                    l = np.array(
                        [x.split() for x in f.read().splitlines()],
                        dtype=np.float32)
            except:
                nm += 1 # print('missing labels for image %s' %
        ↪ self.img_files[i]) # file missing
                continue

            if l.shape[0]:
                # 判断文件内容是否符合要求
                # 所有的值需要>0, <1, 一共 5 列
                assert l.shape[1] == 5, '> 5 label columns: %s' % file
                assert (l >= 0).all(), 'negative labels: %s' % file
                assert (l[:, 1:] <= 1).all(
                ↪ %s' % file
                ), 'non-normalized or out of bounds coordinate labels:

            if np.unique(
                l, axis=0).shape[0] < l.shape[0]: # duplicate rows
                nd += 1 # print('WARNING: duplicate rows in %s' %
        ↪ self.label_files[i]) # duplicate rows

            self.labels[i] = l
            nf += 1 # file found

            # 创建一个小型的数据集进行试验
            if create_datasubset and ns < 1E4:
                if ns == 0:
                    create_folder(path='./datasubset')
                    os.makedirs('./datasubset/images')
                    exclude_classes = 43
                    if exclude_classes not in l[:, 0]:
                        ns += 1

```

```

        # shutil.copy(src=self.img_files[i],
        # ↪ dst='./datasubset/images/') # copy image
        with open('./datasubset/images.txt', 'a') as f:
            f.write(self.img_files[i] + '\n')

# 为两阶段分类器提取目标检测的检测框
# 默认开关是关掉的，不是很理解
if extract_bounding_boxes:
    p = Path(self.img_files[i])
    img = cv2.imread(str(p))
    h, w = img.shape[:2]
    for j, x in enumerate(l):
        f = '%s%sclassifier%s%g_%g_%s' % (p.parent.parent,
                                           os.sep, os.sep,
                                           x[0], j, p.name)

        if not os.path.exists(Path(f).parent):
            os.makedirs(Path(f).parent)
            # make new output folder

        b = x[1:] * np.array([w, h, w, h]) # box
        b[2:] = b[2:].max() # rectangle to square
        b[2:] = b[2:] * 1.3 + 30 # pad

        b =
        ↪ xywh2xyxy(b.reshape(-1,4)).ravel().astype(np.int)

        b[[0,2]] = np.clip(b[[0, 2]], 0,w) # clip boxes
        ↪ outside of image

        b[[1, 3]] = np.clip(b[[1, 3]], 0, h)
        assert cv2.imwrite(f, img[b[1]:b[3], b[0]:b[2]]),
            ↪ 'Failure extracting classifier boxes'

    else:
        ne += 1

    pbar.desc = 'Caching labels (%g found, %g missing, %g empty,
    ↪ %g duplicate, for %g images)'
    % (nf, nm, ne, nd, n) # 统计发现，丢失，空，重复标签的数量。
    assert nf > 0, 'No labels found. See %s' % help_url

# 将图片加载到内存中，可以加速训练
# 警告：如果在数据比较多的情况下可能会超出 RAM
if cache_images: # if training
    gb = 0 # 计算缓存到内存中的图片占用的空间 GB 为单位

```

```

pbar = tqdm(range(len(self.img_files)), desc='Caching images')
self.img_hw0, self.img_hw = [None] * n, [None] * n
for i in pbar: # max 10k images
    self.imgs[i], self.img_hw0[i], self.img_hw[i] = load_image(
        self, i) # img, hw_original, hw_resized
    gb += self.imgs[i].nbytes
    pbar.desc = 'Caching images (%.1fGB)' % (gb / 1E9)

# 删除损坏的文件
# 根据需要进行手动开关
detect_corrupted_images = False
if detect_corrupted_images:
    from skimage import io # conda install -c conda-forge
    ↪ scikit-image
    for file in tqdm(self.img_files,
                      desc='Detecting corrupted images'):
        try:
            _ = io.imread(file)
        except:
            print('Corrupted image detected: %s' % file)

```

Rectangular inference（矩形推理）

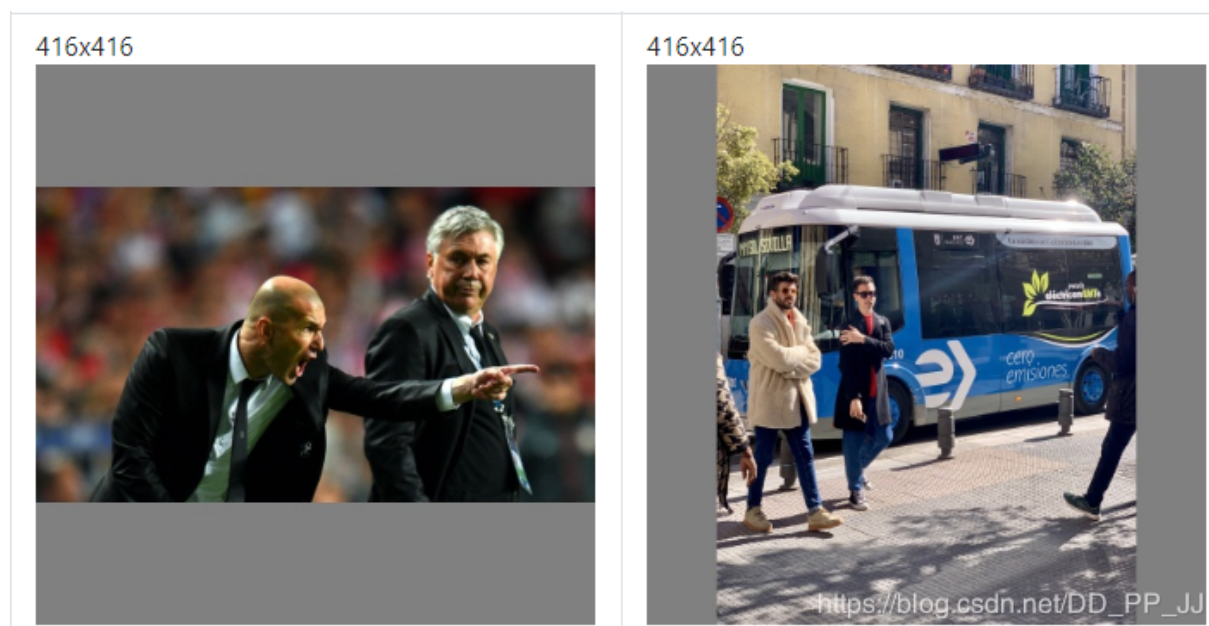
1. 矩形推理是在 detect.py，也就是测试过程中的实现，可以减少推理时间。YOLOv3 中是下采样 32 倍，长宽也必须是 32 的倍数，所以在进入模型前，数据需要处理到 416×416 大小，这个过程称为仿射变换，如果用 opencv 实现可以用以下代码：

```

# 来自 https://zhuanlan.zhihu.com/p/93822508
def cv2_letterbox_image(image, expected_size):
    ih, iw = image.shape[0:2]
    ew, eh = expected_size
    scale = min(eh / ih, ew / iw)
    nh = int(ih * scale)
    nw = int(iw * scale)
    image = cv2.resize(image, (nw, nh), interpolation=cv2.INTER_CUBIC)
    top = (eh - nh) // 2
    bottom = eh - nh - top
    left = (ew - nw) // 2
    right = ew - nw - left
    new_img = cv2.copyMakeBorder(image, top, bottom, left, right,
    ↪ cv2.BORDER_CONSTANT)
    return new_img

```

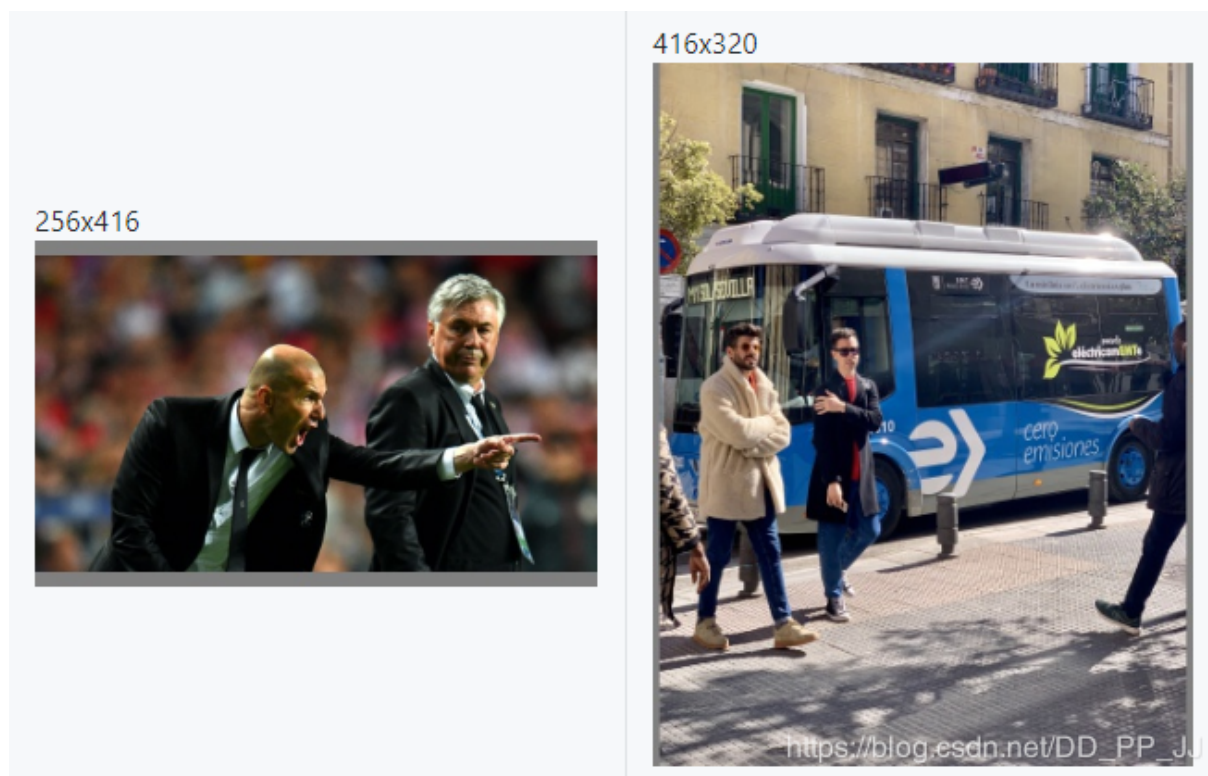
比如下图是一个 h>w，一个是 w>h 的图片经过仿射变换后 resize 到 416×416 的示例：



以上就是正方形推理，但是可以看出以上通过补充得到的结果会存在很多冗余信息，而 **Rectangular Training** 思路就是想要去掉这些冗余的部分。

具体过程为：求得较长边缩放到 416 的比例，然后对图片 $w:h$ 按这个比例缩放，使得较长边达到 416，再对较短边进行尽量少的填充使得较短边满足 32 的倍数。

示例如下：



Rectangular Training（矩形训练）

很自然的，训练的过程也可以用到这个想法，减少冗余。不过训练的时候情况比较复杂，由于在训练过程中是一个 batch 的图片，而每个 batch 图片是有可能长宽比不同的，这就是与测试最大的区别。具体是实现是取这个 batch 中最大的场合宽，然后将整个 batch 中填充到 max width 和 max height，这样操作对小一些的图片来说也是比较浪费。这里的 yolov3 的实现主要就是优化了一下如何将比例相近的图片放在一个 batch，这样显然填充的就更少一些了。作者在 issue 中提到，在 coco 数据集中使用这个策略进行训练，能够快 1/3。

而如果选择开启矩形训练，必须要关闭 dataloader 中的 shuffle 参数，防止对数据的顺序进行调整。同时如果选择 image_weights, 根据图片进行采样，也无法与矩阵训练同时使用。

3.2 getitem 函数

```
def __getitem__(self, index):
    # 新的下角标
    if self.image_weights:
        index = self.indices[index]

    img_path = self.img_files[index]
    label_path = self.label_files[index]
```

```

hyp = self.hyp
mosaic = True and self.augment
# 如果开启镶嵌增强、数据增强
# 加载四张图片，作为一个镶嵌，具体看下文解析。
if mosaic:
    # 加载镶嵌内容
    img, labels = load_mosaic(self, index)
    shapes = None

else:
    # 加载图片
    img, (h0, w0), (h, w) = load_image(self, index)

    # 仿射变换
    shape = self.batch_shapes[self.batch[
        index]] if self.rect else self.img_size
    img, ratio, pad = letterbox(img,
                                shape,
                                auto=False,
                                scaleup=self.augment)

    shapes = (h0, w0), (
        (h / h0, w / w0), pad)

    # 加载标注文件
    labels = []
    if os.path.isfile(label_path):
        x = self.labels[index]
        if x is None: # 如果标签没有加载，读取 label_path 内容
            with open(label_path, 'r') as f:
                x = np.array(
                    [x.split() for x in f.read().splitlines()],
                    dtype=np.float32)

        if x.size > 0:
            # 将归一化后的 xywh 转化为左上角、右下角的表达形式
            labels = x.copy()
            labels[:, 1] = ratio[0] * w * (
                x[:, 1] - x[:, 3] / 2) + pad[0] # pad width
            labels[:, 2] = ratio[1] * h * (
                x[:, 2] - x[:, 4] / 2) + pad[1] # pad height
            labels[:, 3] = ratio[0] * w * (x[:, 1] +
                x[:, 3] / 2) + pad[0]

```

```
labels[:, 4] = ratio[1] * h * (x[:, 2] +
                               x[:, 4] / 2) + pad[1]

if self.augment:
    # 图片空间的数据增强
    if not mosaic:
        # 如果没有使用镶嵌的方法，那么对图片进行随机放射
        img, labels = random_affine(img,
                                     labels,
                                     degrees=hyp['degrees'],
                                     translate=hyp['translate'],
                                     scale=hyp['scale'],
                                     shear=hyp['shear'])

    # 增强 hsv 空间
    augment_hsv(img,
                hgain=hyp['hsv_h'],
                sgain=hyp['hsv_s'],
                vgain=hyp['hsv_v'])

nL = len(labels) # 标注文件个数

if nL:
    # 将 xyxy 格式转化为 xywh 格式
    labels[:, 1:5] = xyxy2xywh(labels[:, 1:5])

    # 归一化到 0-1 之间
    labels[:, [2, 4]] /= img.shape[0] # height
    labels[:, [1, 3]] /= img.shape[1] # width

if self.augment:
    # 随机左右翻转
    lr_flip = True
    if lr_flip and random.random() < 0.5:
        img = np.fliplr(img)
        if nL:
            labels[:, 1] = 1 - labels[:, 1]

    # 随机上下翻转
    ud_flip = False
    if ud_flip and random.random() < 0.5:
        img = np.flipud(img)
        if nL:
```



```

labels[:, 2] = 1 - labels[:, 2]

labels_out = torch.zeros((nL, 6))
if nL:
    labels_out[:, 1:] = torch.from_numpy(labels)

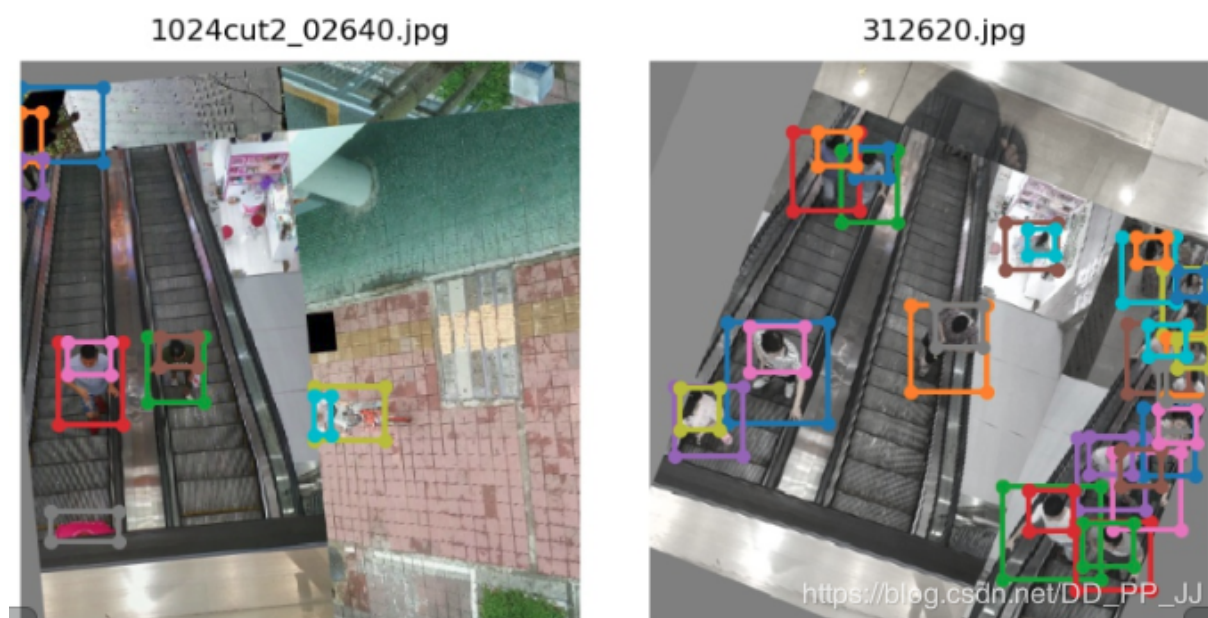
# 图像维度转换
img = img[:, :, ::-1].transpose(2, 0, 1) # BGR to RGB, to 3x416x416
img = np.ascontiguousarray(img)

return torch.from_numpy(img), labels_out, img_path, shapes

```

下图是开启了镶嵌和旋转以后的增强效果（mosaic不知道翻译的对不对，如果有问题，欢迎指正。）

这里理解镶嵌就是将四张图片，以不同的比例，合成为一张图片。



3.3 collate_fn 函数

```

@staticmethod
def collate_fn(batch):
    img, label, path, shapes = zip(*batch) # transposed
    for i, l in enumerate(label):
        l[:, 0] = i # add target image index for build_targets()
    return torch.stack(img, 0), torch.cat(label, 0), path, shapes

```

还有最后一点内容，是关于 pytorch 的数据读取机制，本人曾经单纯的认为 dataloader 仅仅是通过

调用 `__getitem__(self, index)`，然后就可以直接返回结果。但是之前做过的项目打破了这样的认知，在 `pytorch` 的 `dataloader` 中是会对通过 `getitem` 方法得到的结果（`batch`）进行包装，而这个包装可能与我们想要的有所不同。默认的方法可以看以下代码：

```
def default_collate(batch):
    """Puts each data field into a tensor with outer dimension batch size"""

    elem_type = type(batch[0])
    if isinstance(batch[0], torch.Tensor):
        out = None
        if _use_shared_memory:
            # If we're in a background process, concatenate directly into a
            # shared memory tensor to avoid an extra copy
            numel = sum([x.numel() for x in batch])
            storage = batch[0].storage()._new_shared(numel)
            out = batch[0].new(storage)
        return torch.stack(batch, 0, out=out)
    elif elem_type.__module__ == 'numpy' and elem_type.__name__ != 'str_' \
        and elem_type.__name__ != 'string_':
        elem = batch[0]
        if elem_type.__name__ == 'ndarray':
            # array of string classes and object
            if np_str_obj_array_pattern.search(elem.dtype.str) is not None:
                raise TypeError(error_msg_fmt.format(elem.dtype))

            return default_collate([torch.from_numpy(b) for b in batch])
        if elem.shape == (): # scalars
            py_type = float if elem.dtype.name.startswith('float') else int
            return numpy_type_map[elem.dtype.name](list(map(py_type, batch)))
    elif isinstance(batch[0], float):
        return torch.tensor(batch, dtype=torch.float64)
    elif isinstance(batch[0], int_classes):
        return torch.tensor(batch)
    elif isinstance(batch[0], string_classes):
        return batch
    elif isinstance(batch[0], container_abcs.Mapping):
        return {key: default_collate([d[key] for d in batch]) for key in
            ↪ batch[0]}
    elif isinstance(batch[0], tuple) and hasattr(batch[0], '_fields'): #
        ↪ namedtuple
        return type(batch[0])(*(default_collate(samples) for samples in
            ↪ zip(*batch)))
    elif isinstance(batch[0], container_abcs.Sequence):
```

```
transposed = zip(*batch)
return [default_collate(samples) for samples in transposed]

raise TypeError((error_msg_fmt.format(type(batch[0]))))
```

会根据你的数据类型进行相应的处理，但是这往往不是我们需要的，所以需要修改 `collate_fn`，具体内容请看代码，比较简单，就不多赘述。

后记：今天的代码读的比较费力，仅仅通过数据加载这部分就能感受到作者所添加的 `trick`，还有思维的严禁，对数据的限制，处理，都已经提前想好了。不仅如此，作者还添加了巨多的数据增强方法，不仅有传统的仿射变换、上下翻转、左右翻转还有比较新颖的比如镶嵌。以上就是为各位大致理了一遍思路，具体的实现还需要再进行细细的琢磨，不过就使用而言，以上信息就已经足够。由于时间仓促，可能还有一些内容调查的不够严谨，比如说镶嵌这个翻译是否正确，欢迎有这方面了解的大佬与我沟通，期待您的指教。

参考文献

矩形训练相关: <https://blog.csdn.net/songwsx/article/details/102639770>

仿射变换: <https://zhuanlan.zhihu.com/p/93822508>

Rectangle Training: <https://github.com/ultralytics/yolov3/issues/232>

数据自由读取: <https://zhuanlan.zhihu.com/p/30385675>

四、YOLOv3 中的参数搜索

前言：YOLOv3 代码中也提供了参数搜索，可以为对应的数据集进化一套合适的超参数。本文建档分析一下有关这部分的操作方法以及其参数的具体进化方法。

1. 超参数

YOLOv3 中的超参数在 `train.py` 中提供，其中包含了一些数据增强参数设置，具体内容如下：

```
hyp = {'giou': 3.54, # giou loss gain
       'cls': 37.4, # cls loss gain
       'cls_pw': 1.0, # cls BCELoss positive_weight
       'obj': 49.5, # obj loss gain (*=img_size/320 if img_size != 320)
       'obj_pw': 1.0, # obj BCELoss positive_weight
       'iou_t': 0.225, # iou training threshold
       'lr0': 0.00579, # initial learning rate (SGD=1E-3, Adam=9E-5)
       'lrf': -4., # final LambdaLR learning rate = lr0 * (10 ** lrf)
       'momentum': 0.937, # SGD momentum
```

```
'weight_decay': 0.000484, # optimizer weight decay
'fl_gamma': 0.5, # focal loss gamma
'hsv_h': 0.0138, # image HSV-Hue augmentation (fraction)
'hsv_s': 0.678, # image HSV-Saturation augmentation (fraction)
'hsv_v': 0.36, # image HSV-Value augmentation (fraction)
'degrees': 1.98, # image rotation (+/- deg)
'translate': 0.05, # image translation (+/- fraction)
'scale': 0.05, # image scale (+/- gain)
'shear': 0.641} # image shear (+/- deg)
```

2. 使用方法

在训练的时候，`train.py` 提供了一个可选参数`--evolve`，这个参数决定了是否进行超参数搜索与进化（默认是不开启超参数搜索的）。

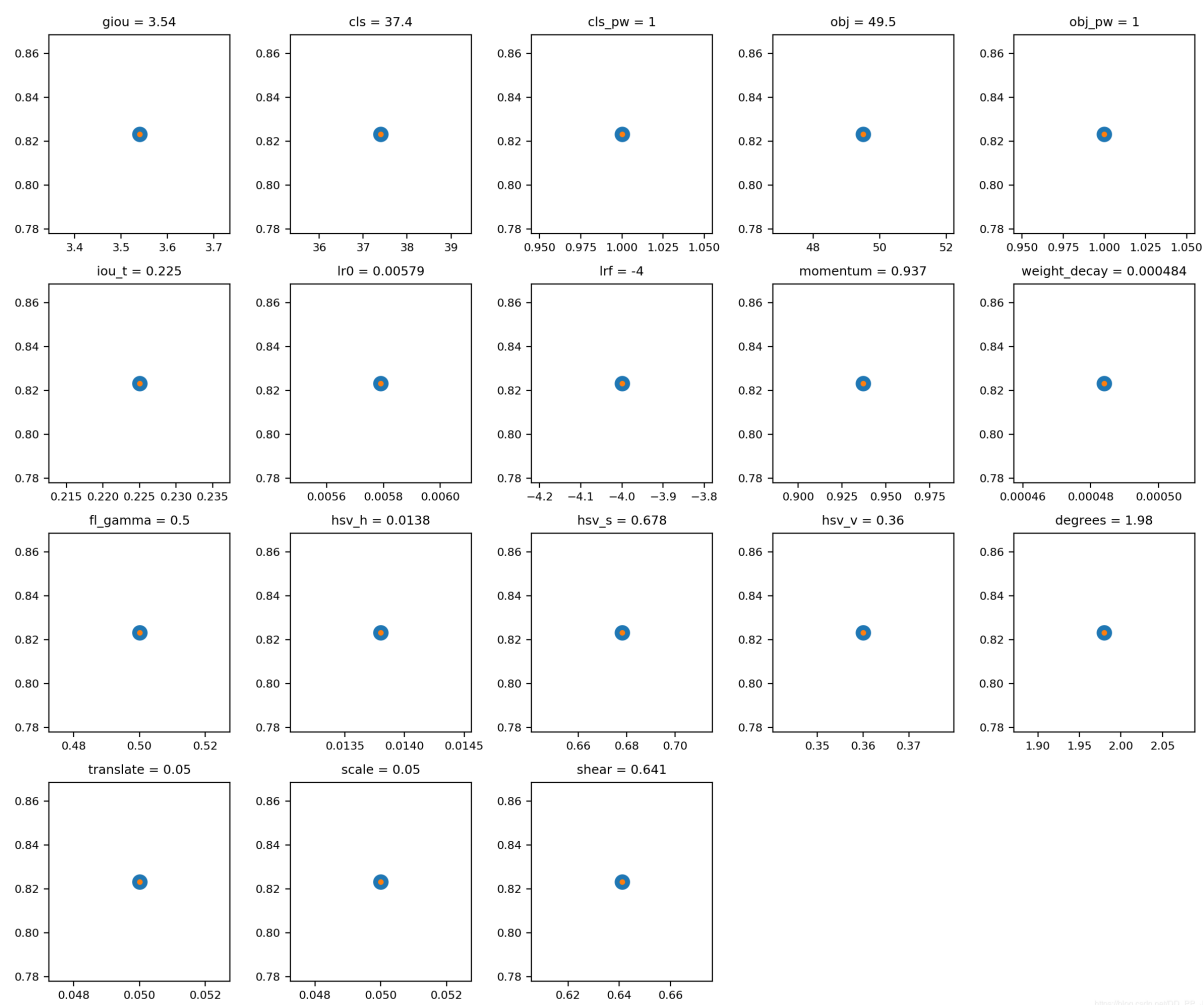
具体使用方法也很简单：

```
python train.py --data data/voc.data
                 --cfg cfg/yolov3-tiny.cfg
                 --img-size 416
                 --epochs 273
                 --evolve
```

实际使用的时候，需要进行修改，`train.py` 中的约 444 行：

```
for _ in range(1): # generations to evolve
```

将其中的 1 修改为你想设置的迭代数，比如 200 代，如果不设置，结果将会如下图所示，实际上就是只有一代。



3. 原理

整个过程比较简单，对于进化过程中的新一代，都选了了适应性最高的前一代（在前几代中）进行突变。以上所有的参数将有约 20% 的 1-sigma 的正态分布几率同时突变。

```
s = 0.2 # sigma
```

整个进化过程需要搞清楚两个点：

1. 如何评判其中一代的好坏？
2. 下一代如何根据上一代进行进化？

第一个问题：判断好坏的标准。

```
def fitness(x):
    w = [0.0, 0.0, 0.8, 0.2]
```

```
# weights for [P, R, mAP, F1]@0.5
return (x[:, :4] * w).sum(1)
```

YOLOv3 进化部分是通过以上的适应度函数判断的，适应度越高，代表这一代的性能越好。而在适应度中，是通过 Precision, Recall, mAP, F1 这四个指标作为适应度的评价标准。

其中的 **w** 是设置的加权，如果更关心 mAP 的值，可以提高 mAP 的权重；如果更关心 F1，则设置更高的权重在对应的 F1 上。这里分配 mAP 权重为 0.8、F1 权重为 0.2。

第二个问题：如何进行进化？

进化过程中有两个重要的参数：

第一个参数为 **parent**，可选值为 **single** 或者 **weighted**，这个参数的作用是：决定如何选择上一代。如果选择 **single**，代表只选择上一代中最好的那个。

```
if parent == 'single' or len(x) == 1:
    x = x[fitness(x).argmax()]
```

如果选择 **weighted**，代表选择得分的前 10 个加权平均的结果作为下一代，具体操作如下：

```
elif parent == 'weighted': # weighted combination
    n = min(10, len(x)) # number to merge
    x = x[np.argsort(-fitness(x))][:n] # top n mutations
    w = fitness(x) - fitness(x).min() # weights
    x = (x * w.reshape(n, 1)).sum(0) / w.sum() # new parent
```

第二个参数为 **method**，可选值为 1, 2, 3，分别代表使用三种模式来进化：

```
# Mutate
method = 2
s = 0.2 # 20% sigma
np.random.seed(int(time.time()))
g = np.array([1, 1, 1, 1, 1, 1, 1, 0, .1, \
              1, 0, 1, 1, 1, 1, 1, 1, 1]) # gains
# 这里的 g 类似加权
ng = len(g)
if method == 1:
    v = (np.random.randn(ng) *
          np.random.random() * g * s + 1) ** 2.0
elif method == 2:
    v = (np.random.randn(ng) *
          np.random.random() * g * s + 1) ** 2.0
elif method == 3:
    v = np.ones(ng)
    while all(v == 1):
```

```
# 为了防止重复,直到有变化才停下来
r = (np.random.random(ng) < 0.1) * np.random.randn(ng)
# 10% 的突变几率
v = (g * s * r + 1) ** 2.0

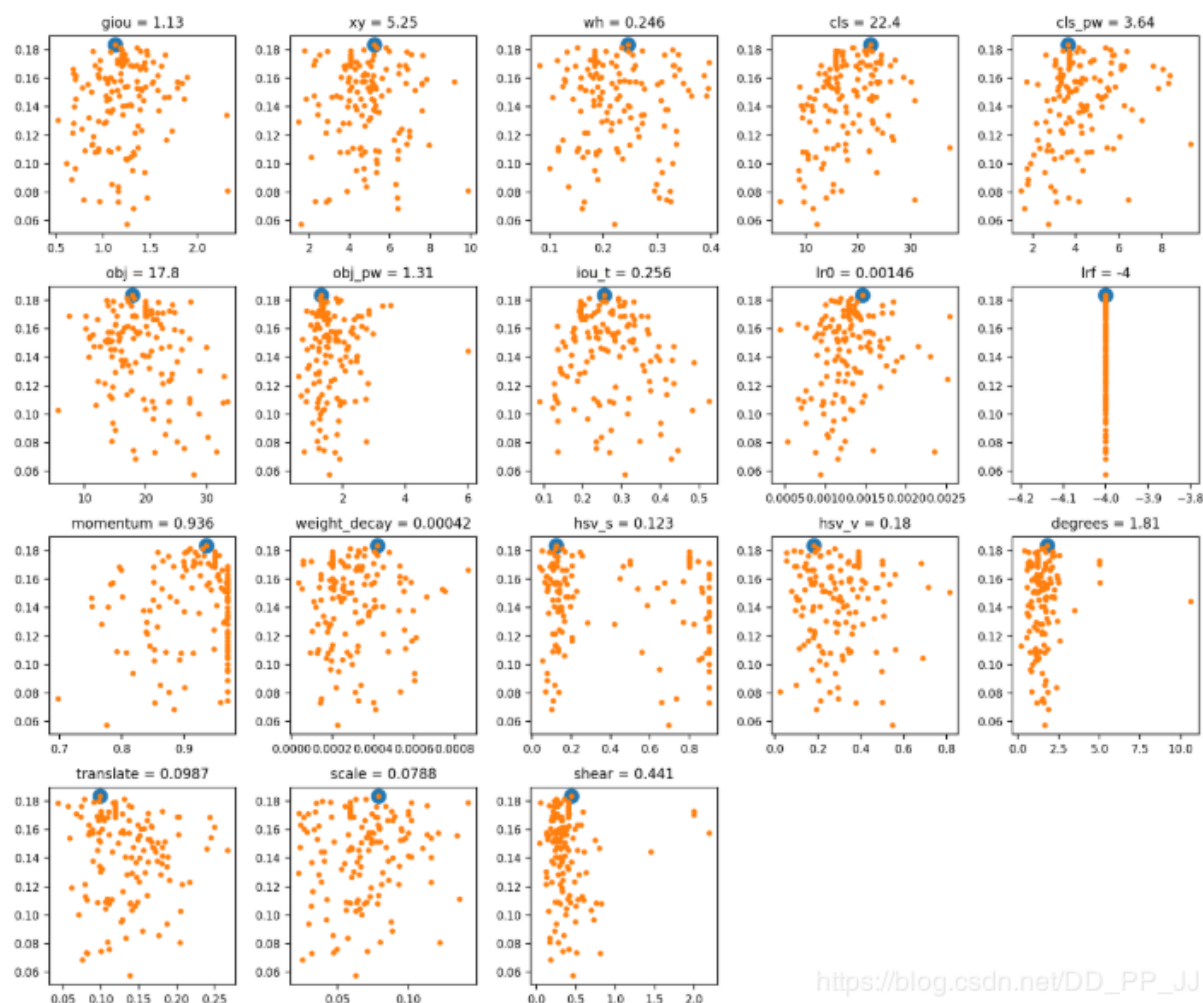
for i, k in enumerate(hyp.keys()):
    hyp[k] = x[i + 7] * v[i]
    # 进行突变
```

另外,为了防止突变过程,导致参数出现明显不合理的范围,需要用一個范围进行框定,将超出范围的内容剪切掉。具体方法如下:

```
# Clip to limits
keys = ['lr0', 'iou_t', 'momentum',
        'weight_decay', 'hsv_s',
        'hsv_v', 'translate',
        'scale', 'fl_gamma']
limits = [(1e-5, 1e-2), (0.00, 0.70),
          (0.60, 0.98), (0, 0.001),
          (0, .9), (0, .9), (0, .9),
          (0, .9), (0, 3)]

for k, v in zip(keys, limits):
    hyp[k] = np.clip(hyp[k], v[0], v[1])
```

最终训练的超参数搜索的结果可视化:



参考资料:

官方 issue: <https://github.com/ultralytics/yolov3/issues/392>

官方代码: <https://github.com/ultralytics/yolov3>

五、网络模型的构建

前言: 之前几篇讲了 `cfg` 文件的理解、数据集的构建、数据加载机制和超参数进化机制, 本文将讲解 YOLOv3 如何从 `cfg` 文件构造模型。本文涉及到一个比较有用的部分就是 `bias` 的设置, 可以提升 `mAP`、`F1`、`P`、`R` 等指标, 还能让训练过程更加平滑。

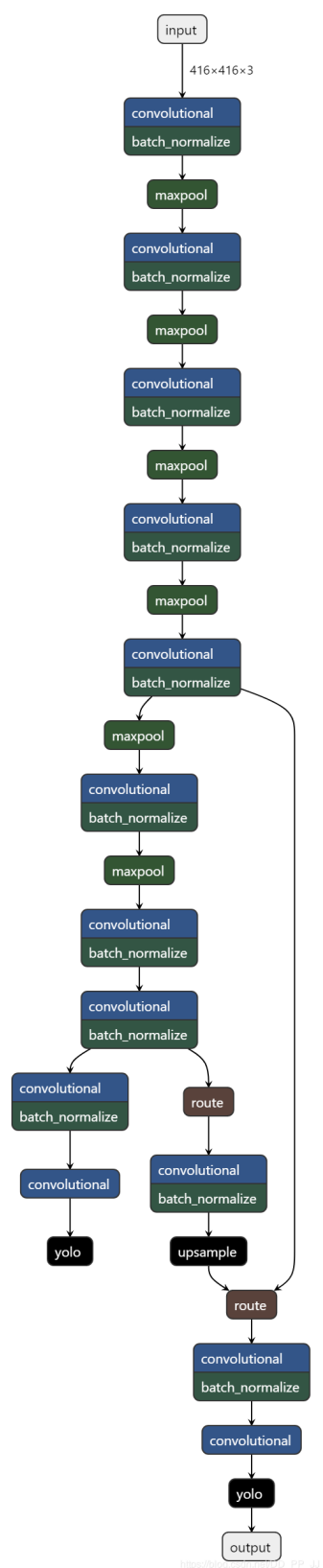
1. cfg 文件

在 YOLOv3 中，修改网络结构很容易，只需要修改 cfg 文件即可。目前，cfg 文件支持 convolutional, maxpool, unsample, route, shortcut, yolo 这几个层。

而且作者也提供了多个 cfg 文件来进行网络构建，比如：yolov3.cfg、yolov3-tiny.cfg、yolov3-spp.cfg、csresnext50-panet-spp.cfg 文件（提供的 yolov3-spp-pan-scale.cfg 文件，在代码级别还没有提供支持）。

如果想要添加自定义的模块也很方便，比如说注意力机制模块、空洞卷积等，都可以简单地得到添加或者修改。

为了更加方便的理解 cfg 文件网络是如何构建的，在这里推荐一个 Github 上的网络结构可视化软件：Netron，下图是可视化 yolov3-tiny 的结果：



2. 网络模型构建

从 `train.py` 文件入手，其中涉及的网络构建的代码为：

```
# Initialize model
model = Darknet(cfg, arc=opt.arc).to(device)
```

然后沿着 `Darknet` 实现进行讲解：

```
class Darknet(nn.Module):
    # YOLOv3 object detection model
    def __init__(self, cfg, img_size=(416, 416), arc='default'):
        super(Darknet, self).__init__()
        self.module_defs = parse_model_cfg(cfg)
        self.module_list, self.routes = create_modules(self.module_defs,
            ↪ img_size, arc)
        self.yolo_layers = get_yolo_layers(self)

    # Darknet Header
    self.version = np.array([0, 2, 5], dtype=np.int32)
    # (int32) version info: major, minor, revision
    self.seen = np.array([0], dtype=np.int64)
    # (int64) number of images seen during training
```

以上文件中，比较关键的就是成员函数变量 `module_defs`、`module_list`、`routes`、`yolo_layers` 四个成员函数，先对这几个参数的意义进行解释：

2.1 module_defs

调用了 `parse_model_cfg` 函数，得到了 `module_defs` 对象。实际上该函数是通过解析 `cfg` 文件，得到一个 `list`，`list` 中包含多个字典，每个字典保存的内容就是一个模块内容，比如说：

```
[convolutional]
batch_normalize=1
filters=128
size=3
stride=2
pad=1
activation=leaky
```

函数代码如下：

```
def parse_model_cfg(path):
    # path 参数为: cfg/yolov3-tiny.cfg
```

```

if not path.endswith('.cfg'):
    path += '.cfg'
if not os.path.exists(path) and os.path.exists('cfg' + os.sep + path):
    path = 'cfg' + os.sep + path

with open(path, 'r') as f:
    lines = f.read().split('\n')

# 去除以 # 开头的, 属于注释部分的内容
lines = [x for x in lines if x and not x.startswith('#')]
lines = [x.rstrip().lstrip() for x in lines]
mdefs = [] # 模块的定义
for line in lines:
    if line.startswith('['): # 标志着一个模块的开始
        '''
        比如:
        [shortcut]
        from=-3
        activation=linear
        '''
        mdefs.append({})
        mdefs[-1]['type'] = line[1:-1].rstrip()
        if mdefs[-1]['type'] == 'convolutional':
            mdefs[-1]['batch_normalize'] = 0
            # pre-populate with zeros (may be overwritten later)
        else:
            # 将键和键值放入字典
            key, val = line.split("=")
            key = key.rstrip()

            if 'anchors' in key:
                mdefs[-1][key] = np.array([float(x) for x in
                ↪ val.split(',')]).reshape((-1, 2)) # np anchors
            else:
                mdefs[-1][key] = val.strip()

# 支持的参数类型
supported = ['type', 'batch_normalize', 'filters', 'size', \
             'stride', 'pad', 'activation', 'layers', 'groups', \
             'from', 'mask', 'anchors', 'classes', 'num', 'jitter', \
             'ignore_thresh', 'truth_thresh', 'random', \
             'stride_x', 'stride_y']

```

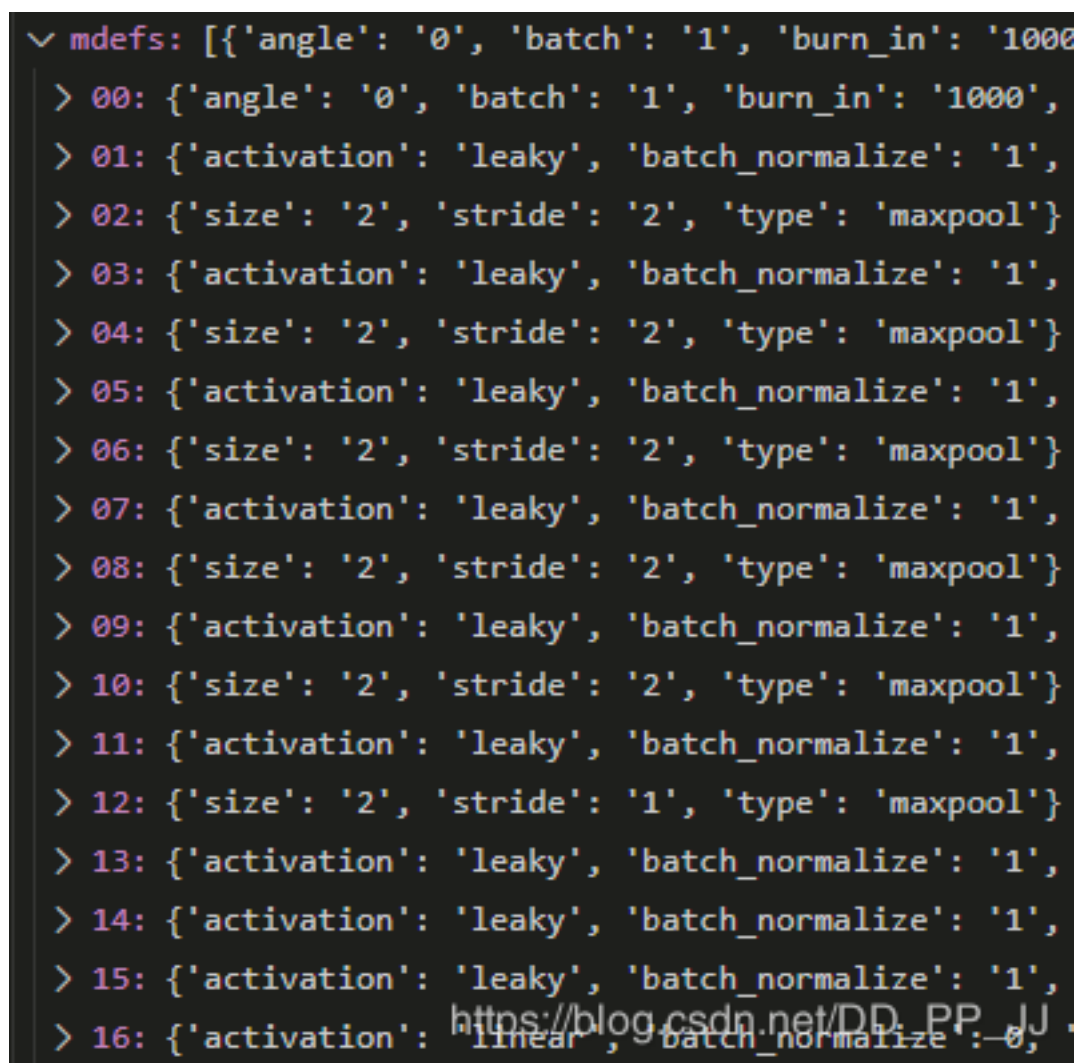
```

# 判断所有参数中是否有不符合要求的 key
f = []
for x in mdefs[1:]:
    [f.append(k) for k in x if k not in f]
u = [x for x in f if x not in supported] # unsupported fields
assert not any(u), "Unsupported fields %s in %s. See
    ↪ https://github.com/ultralytics/yolov3/issues/631" % (u, path)

return mdefs

```

返回的内容通过 debug 模式进行查看：



```

v mdefs: [{'angle': '0', 'batch': '1', 'burn_in': '1000',
> 00: {'angle': '0', 'batch': '1', 'burn_in': '1000',
> 01: {'activation': 'leaky', 'batch_normalize': '1',
> 02: {'size': '2', 'stride': '2', 'type': 'maxpool'}
> 03: {'activation': 'leaky', 'batch_normalize': '1',
> 04: {'size': '2', 'stride': '2', 'type': 'maxpool'}
> 05: {'activation': 'leaky', 'batch_normalize': '1',
> 06: {'size': '2', 'stride': '2', 'type': 'maxpool'}
> 07: {'activation': 'leaky', 'batch_normalize': '1',
> 08: {'size': '2', 'stride': '2', 'type': 'maxpool'}
> 09: {'activation': 'leaky', 'batch_normalize': '1',
> 10: {'size': '2', 'stride': '2', 'type': 'maxpool'}
> 11: {'activation': 'leaky', 'batch_normalize': '1',
> 12: {'size': '2', 'stride': '1', 'type': 'maxpool'}
> 13: {'activation': 'leaky', 'batch_normalize': '1',
> 14: {'activation': 'leaky', 'batch_normalize': '1',
> 15: {'activation': 'leaky', 'batch_normalize': '1',
> 16: {'activation': 'linear', 'batch_normalize': '0',

```

其中需要关注的就是 anchor 的组织：

```

✓ 17: {'anchors': array([[          10, ...    319]])}
  ✓ 'anchors': array([[          10,          14],\n
    ✓ [0:6] : [array([          10, ...    14]), arr
    > 0: array([          10,          14])
    > 1: array([          23,          27])
    > 2: array([          37,          58])
    > 3: array([          81,          82])
    > 4: array([         135,         169])
    > 5: array([         344,         319])

```

可以看出，anchor 是按照每两个一对进行组织的，与我们的理解一致。

2.2 module_list&routs

这个部分是本文的核心，也是理解模型构建的关键。

在 pytorch 中，构建模型常见的有通过 Sequential 或者 ModuleList 进行构建。

通过 **Sequential** 构建

```

model=nn.Sequential()
model.add_module('conv',nn.Conv2d(3,3,3))
model.add_module('batchnorm',nn.BatchNorm2d(3))
model.add_module('activation_layer',nn.ReLU())

```

或者

```

model=nn.Sequential(
    nn.Conv2d(3,3,3),
    nn.BatchNorm2d(3),
    nn.ReLU()
)

```

或者

```

from collections import OrderedDict
model=nn.Sequential(OrderedDict([
    ('conv',nn.Conv2d(3,3,3)),
    ('batchnorm',nn.BatchNorm2d(3)),
    ('activation_layer',nn.ReLU())
]))

```

通过 `sequential` 构建的模块内部实现了 **forward** 函数，可以直接传入参数，进行调用。

通过 **ModuleList** 构建

```
model=nn.ModuleList([nn.Linear(3,4),
                      nn.ReLU(),
                      nn.Linear(4,2)])
```

`ModuleList` 类似 `list`，内部没有实现 **forward** 函数，使用的时候需要构建 `forward` 函数，构建自己模型常用 `ModuleList` 函数建立子模型，建立 `forward` 函数实现前向传播。

在 YOLOv3 中，灵活地结合了两种使用方式，通过解析以上得到的 `module_defs`，进行构建一个 `ModuleList`，然后再通过构建 `forward` 函数进行前向传播即可。

具体代码如下：

```
def create_modules(module_defs, img_size, arc):
    # 通过 module_defs 进行构建模型
    hyperparams = module_defs.pop(0)
    output_filters = [int(hyperparams['channels'])]
    module_list = nn.ModuleList()
    routs = [] # 存储了所有的层，在 route、shortcut 会使用到。
    yolo_index = -1

    for i, mdef in enumerate(module_defs):
        modules = nn.Sequential()
        '''
        通过 type 字样不同的类型，来进行模型构建
        '''
        if mdef['type'] == 'convolutional':
            bn = int(mdef['batch_normalize'])
            filters = int(mdef['filters'])
            size = int(mdef['size'])
            stride = int(mdef['stride']) if 'stride' in mdef else (int(
                mdef['stride_y']), int(mdef['stride_x']))
            pad = (size - 1) // 2 if int(mdef['pad']) else 0
            modules.add_module(
                'Conv2d',
                nn.Conv2d(
                    in_channels=output_filters[-1],
                    out_channels=filters,
                    kernel_size=size,
                    stride=stride,
                    padding=pad,
                    groups=int(mdef['groups']) if 'groups' in mdef else 1,
```

```

        bias=not bn))
    if bn:
        modules.add_module('BatchNorm2d',
                            nn.BatchNorm2d(filters, momentum=0.1))
    if mdef['activation'] == 'leaky': # TODO: activation study
        ↪ https://github.com/ultralytics/yolov3/issues/441
        modules.add_module('activation', nn.LeakyReLU(0.1,
                                                       inplace=True))

    elif mdef['activation'] == 'swish':
        modules.add_module('activation', Swish())
    # 在此处可以添加新的激活函数

elif mdef['type'] == 'maxpool':
    # 最大池化操作
    size = int(mdef['size'])
    stride = int(mdef['stride'])
    maxpool = nn.MaxPool2d(kernel_size=size,
                           stride=stride,
                           padding=int((size - 1) // 2))
    if size == 2 and stride == 1: # yolov3-tiny
        modules.add_module('ZeroPad2d', nn.ZeroPad2d((0, 1, 0, 1)))
        modules.add_module('MaxPool2d', maxpool)
    else:
        modules = maxpool

elif mdef['type'] == 'upsample':
    # 通过近邻插值完成上采样
    modules = nn.Upsample(scale_factor=int(mdef['stride']),
                           mode='nearest')

elif mdef['type'] == 'route':
    # nn.Sequential() placeholder for 'route' layer
    layers = [int(x) for x in mdef['layers'].split(',')]
    filters = sum(
        [output_filters[i + 1 if i > 0 else i] for i in layers])
    # extend 表示添加一系列对象
    routs.extend([l if l > 0 else l + i for l in layers])

elif mdef['type'] == 'shortcut':
    # nn.Sequential() placeholder for 'shortcut' layer
    filters = output_filters[int(mdef['from'])]
    layer = int(mdef['from'])
    routs.extend([i + layer if layer < 0 else layer])

```



```

elif mdef['type'] == 'yolo':
    yolo_index += 1
    mask = [int(x) for x in mdef['mask'].split(',')] # anchor mask
    modules = YOLOLayer(
        anchors=mdef['anchors'][mask], # anchor list
        nc=int(mdef['classes']), # number of classes
        img_size=img_size, # (416, 416)
        yolo_index=yolo_index, # 0, 1 or 2
        arc=arc) # yolo architecture

# 这是在 focal loss 文章中提到的为卷积层添加 bias
# 主要用于解决样本不平衡问题
# (论文地址 https://arxiv.org/pdf/1708.02002.pdf section 3.3)
# 具体讲解见下方
try:
    if arc == 'defaultpw' or arc == 'Fdefaultpw':
        # default with positive weights
        b = [-5.0, -5.0] # obj, cls
    elif arc == 'default':
        # default no pw (40 cls, 80 obj)
        b = [-5.0, -5.0]
    elif arc == 'uBCE':
        # unified BCE (80 classes)
        b = [0, -9.0]
    elif arc == 'uCE':
        # unified CE (1 background + 80 classes)
        b = [10, -0.1]
    elif arc == 'Fdefault':
        # Focal default no pw (28 cls, 21 obj, no pw)
        b = [-2.1, -1.8]
    elif arc == 'uFBCE' or arc == 'uFBCEpw':
        # unified FocalBCE (5120 obj, 80 classes)
        b = [0, -6.5]
    elif arc == 'uFCE':
        # unified FocalCE (64 cls, 1 background + 80 classes)
        b = [7.7, -1.1]

    bias = module_list[-1][0].bias.view(len(mask), -1)
    # 255 to 3x85
    bias[:, 4] += b[0] - bias[:, 4].mean() # obj
    bias[:, 5:] += b[1] - bias[:, 5:].mean() # cls

```

```
# 将新的偏移量赋值回模型中
module_list[-1][0].bias = torch.nn.Parameter(bias.view(-1))

except:
    print('WARNING: smart bias initialization failure.')

else:
    print('Warning: Unrecognized Layer Type: ' + mdef['type'])

# 将 module 内容保存在 module_list 中。
module_list.append(modules)
# 保存所有的 filter 个数
output_filters.append(filters)

return module_list, routs
```

bias 部分讲解

其中在 YOLO Layer 部分涉及到一个初始化的 trick，来自 Focal Loss 中关于模型初始化的讨论，具体内容请阅读论文，<https://arxiv.org/pdf/1708.02002.pdf> 的第 3.3 节。

3.3. Class Imbalance and Model Initialization

Binary classification models are by default initialized to have equal probability of outputting either $y = -1$ or 1 . Under such an initialization, in the presence of class imbalance, the loss due to the frequent class can dominate total loss and cause instability in early training. To counter this, we introduce the concept of a ‘prior’ for the value of p estimated by the model for the rare class (foreground) *at the start of training*. We denote the prior by π and set it so that the model’s estimated p for examples of the rare class is low, *e.g.* 0.01. We note that this is a change in model initialization (see §4.1) and *not* of the loss function. We found this to improve training stability for both the cross entropy and focal loss in the case of heavy class imbalance.

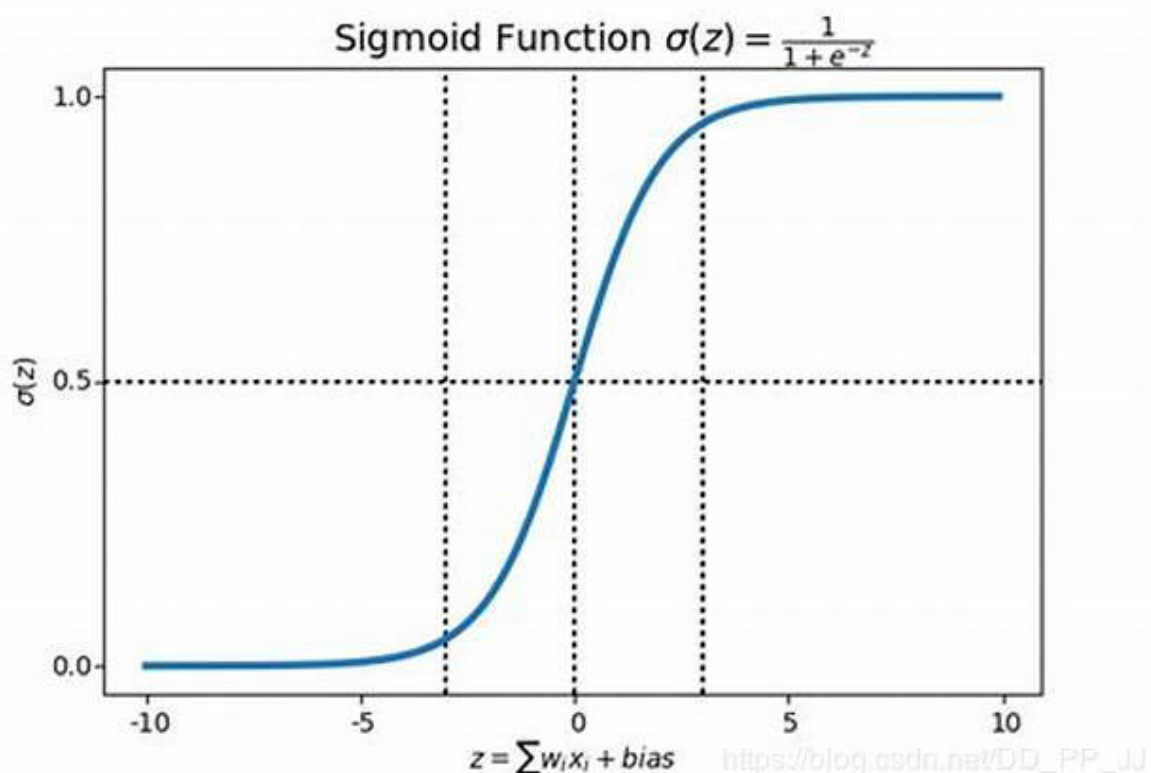
这里涉及到一个非常 insight 的点，笔者与 BBuf 讨论了很长时间，才理解这样做的原因。

我们在第一篇中介绍了，YOLO 层前一个卷积的 filter 个数计算公式如下：

$$filter = (class + 5) \times 3$$

5 代表 x,y,w,h, score, score 代表该格子中是否存在目标，3 代表这个格子中会分配 3 个 anchor 进行匹配。在 YOLOLayer 中的 forward 函数中，有以下代码，需要通过 sigmoid 激活函数：

```
if 'default' in self.arc: # seperate obj and cls
    torch.sigmoid_(io[..., 4])
elif 'BCE' in self.arc: # unified BCE (80 classes)
    torch.sigmoid_(io[..., 5:])
    io[..., 4] = 1
elif 'CE' in self.arc: # unified CE (1 background + 80 classes)
    io[..., 4:] = F.softmax(io[..., 4:], dim=4)
    io[..., 4] = 1
```



可以观察到，Sigmoid 梯度是有限的，大致在 $[-10, 10]$ 之间。

在 `pytorch` 中的卷积层默认的初始化是以 0 为中心点的正态分布，这样进行的初始化会导致很多 `gird` 中大约一半得到了激活，在计算 `loss` 的时候就会计算上所有的激活的点对应的坐标信息，这样计算 `loss` 就会变得很大。

根据这个现象，作者选择在 `YOLOLayer` 的前一个卷积层添加 `bias`，来避免这种情况，实际操作就是在原有的 `bias` 上减去 5，这样通过卷积得到的数值就不会被激活，可以防止在初始阶段的第一个 `batch` 中就进行过拟合。通过以上操作，能够让所有的神经元在前几个 `batch` 中输出空的检测。

经过作者的实验，通过使用 `bias` 的 `trick`，可以提升 `mAP`、`F1`、`P`、`R` 等指标，还能让训练过程更加平滑。

2.3 yolo_layers

代码如下：

```
def get_yolo_layers(model):
    return [i for i, x in enumerate(model.module_defs) if x['type'] ==
            ↪ 'yolo']
    # [82, 94, 106] for yolov3
```

`yolo layer` 的获取是通过解析 `module_defs` 这个存储 `cfg` 文件中的信息的变量得到的。以 `yolov3.cfg` 为例，最终返回的是 `yolo` 层在整个 `module` 的序号。比如：第 83,94,106 个层是 `YOLO` 层。

3. forward 函数

在 `YOLO` 中，如果能理解前向传播的过程，那整个网络的构建也就很清楚了。

```
def forward(self, x, var=None):
    img_size = x.shape[-2:]
    layer_outputs = []
    output = []

    for i, (mdef,
            module) in enumerate(zip(self.module_defs, self.module_list)):
        mtype = mdef['type']
        if mtype in ['convolutional', 'upsample', 'maxpool']:
            # 卷积层，上采样，池化层只需要经过即可
            x = module(x)
        elif mtype == 'route':
            # route 操作就是将几个层的内容拼接起来，具体可以看 cfg 文件解析
            layers = [int(x) for x in mdef['layers'].split(',')]
            if len(layers) == 1:
```

```

        x = layer_outputs[layers[0]]
    else:
        try:
            x = torch.cat([layer_outputs[i] for i in layers], 1)
        except:
            # apply stride 2 for darknet reorg layer
            layer_outputs[layers[1]] = F.interpolate(
                layer_outputs[layers[1]], scale_factor=[0.5, 0.5])
            x = torch.cat([layer_outputs[i] for i in layers], 1)

    elif mtype == 'shortcut':
        x = x + layer_outputs[int(mdef['from'])]
    elif mtype == 'yolo':
        output.append(module(x, img_size))
    # 记录 route 对应的层
    layer_outputs.append(x if i in self.routes else [])

if self.training:
    # 如果训练，直接输出 YOLO 要求的 Tensor
    # 3*(class+5)
    return output

elif ONNX_EXPORT: # 这个是对应的 onnx 导出的内容
    x = [torch.cat(x, 0) for x in zip(*output)]
    return x[0], torch.cat(x[1:3], 1) # scores, boxes: 3780x80,
    ↪ 3780x4
else:
    # 对应测试阶段
    io, p = list(zip(*output)) # inference output, training output
    return torch.cat(io, 1), p

```

forward 的过程也比较简单，通过得到的 module_defs 和 module_list 变量，通过 for 循环将整个 module_list 中的内容进行一遍串联，需要得到的最终结果是 YOLO 层的输出。（ps：下一篇文章再进行 YOLOLayer 的代码解析）

参考资料

sequential 用法https://blog.csdn.net/happyday_d/article/details/85629119

<https://arxiv.org/pdf/1708.02002.pdf>

六、模型构建中的 YOLOLayer

前言：上次讲了 YOLOv3 中的模型构建，从头到尾理了一遍从 cfg 读取到模型整个构建的过程。其中模型构建中最重要的 YOLOLayer 还没有梳理，本文将从代码的角度理解 YOLOLayer 的构建与实现。

1. Grid 创建

YOLOv3 是一个单阶段的目标检测器，将目标划分为不同的 grid，每个 grid 分配 3 个 anchor 作为先验框来进行匹配。首先读一下代码中关于 grid 创建的部分。

首先了解一下 pytorch 中的 API: `torch.meshgrid`

举一个简单的例子就比较清楚了：

```
Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)] ::
  ↳ Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> a = torch.arange(3)
>>> b = torch.arange(5)
>>> x,y = torch.meshgrid(a,b)
>>> a
tensor([0, 1, 2])
>>> b
tensor([0, 1, 2, 3, 4])
>>> x
tensor([[0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1],
        [2, 2, 2, 2, 2]])
>>> y
tensor([[0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4]])
>>>
```

单纯看输入输出，可能不是很明白，列举一个例子：

```
>>> for i in range(3):
...     for j in range(4):
...         print("(" , x[i,j], "," ,y[i,j],")")
...
( tensor(0) , tensor(0) )
```

```
( tensor(0) , tensor(1) )
( tensor(0) , tensor(2) )
( tensor(0) , tensor(3) )
( tensor(1) , tensor(0) )
( tensor(1) , tensor(1) )
( tensor(1) , tensor(2) )
( tensor(1) , tensor(3) )
( tensor(2) , tensor(0) )
( tensor(2) , tensor(1) )
( tensor(2) , tensor(2) )
( tensor(2) , tensor(3) )
```

```
>>> torch.stack((x,y),2)
tensor([[[[0, 0],
          [0, 1],
          [0, 2],
          [0, 3],
          [0, 4]],

         [[1, 0],
          [1, 1],
          [1, 2],
          [1, 3],
          [1, 4]],

         [[2, 0],
          [2, 1],
          [2, 2],
          [2, 3],
          [2, 4]]]])
>>>
```

现在就比较清楚了，划分了 3×4 的网格，通过遍历得到的 x 和 y 就能遍历全部格子。

下面是 yolov3 中提供的代码 (需要注意的是这是针对某一层 YOLOLayer，而不是所有的 YOLOLayer):

```
def create_grids(self,
                 img_size=416,
                 ng=(13, 13),
                 device='cpu',
                 type=torch.float32):
    nx, ny = ng # 网格尺寸
    self.img_size = max(img_size)
    # 下采样倍数为 32
```

```
self.stride = self.img_size / max(ng)

# 划分网格，构建相对左上角的偏移量
yv, xv = torch.meshgrid([torch.arange(ny), torch.arange(nx)])
# 通过以上例子很容易理解
self.grid_xy = torch.stack((xv, yv), 2).to(device).type(type).view(
    (1, 1, ny, nx, 2))

# 处理 anchor，将其除以以下采样倍数
self.anchor_vec = self.anchors.to(device) / self.stride
self.anchor_wh = self.anchor_vec.view(1, self.na, 1, 1,
                                         2).to(device).type(type)

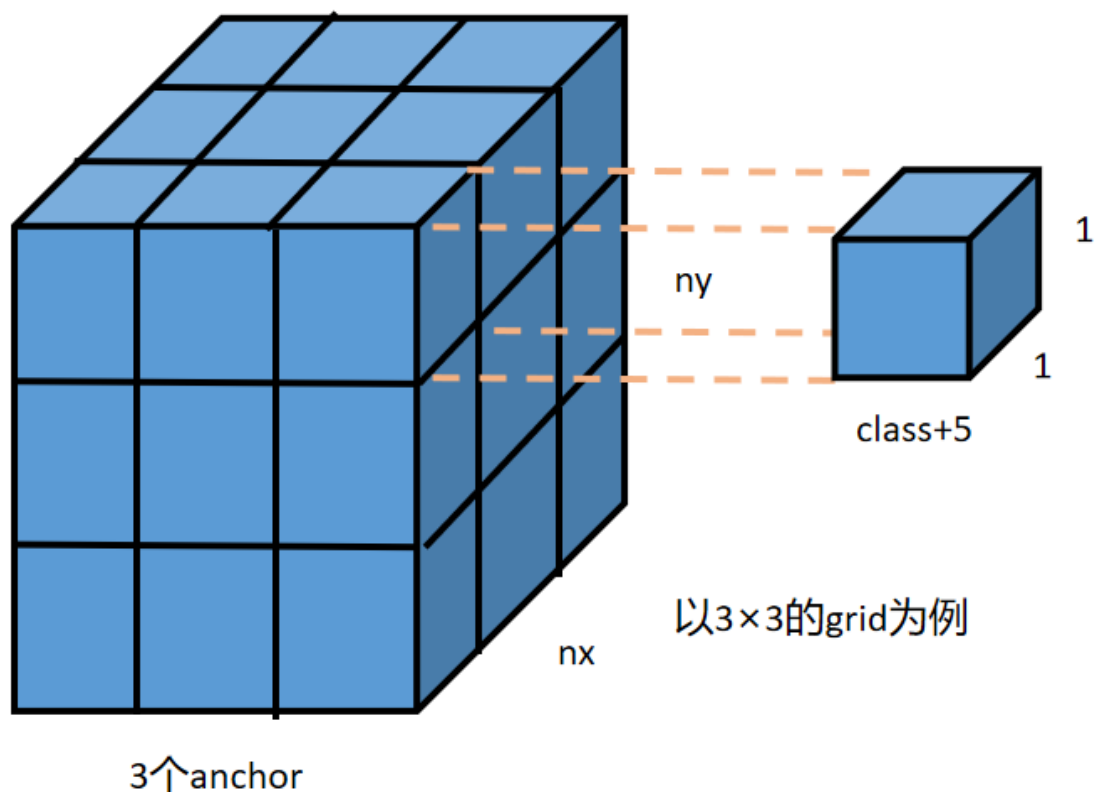
self.ng = torch.Tensor(ng).to(device)
self.nx = nx
self.ny = ny
```

2. YOLOLayer

在之前的文章中讲过，YOLO 层前一层卷积层的 filter 个数具有特殊的要求，计算方法为：

$$filter_num = anchor_num \times (5 + classes_num)$$

如下图所示：



P-> feature map

https://blog.csdn.net/DD_PP_JJ

训练过程:

YOLOLayer 的作用就是对上一个卷积层得到的张量进行处理，具体可以看 training 过程涉及的代码 (暂时不关心 ONNX 部分的代码):

```
class YOLOLayer(nn.Module):
    def __init__(self, anchors, nc, img_size, yolo_index, arc):
        super(YOLOLayer, self).__init__()

        self.anchors = torch.Tensor(anchors)
        self.na = len(anchors) # 该 YOLOLayer 分配给每个 grid 的 anchor 的个数
        self.nc = nc # 类别个数
        self.no = nc + 5 # 每个格子对应输出的维度 class + 5 中 5 代表 x,y,w,h,conf
        self.nx = 0 # 初始化 x 方向上的格子数量
        self.ny = 0 # 初始化 y 方向上的格子数量
        self.arc = arc

        if ONNX_EXPORT: # grids must be computed in __init__
            stride = [32, 16, 8][yolo_index] # stride of this layer
```

```

        nx = int(img_size[1] / stride) # number x grid points
        ny = int(img_size[0] / stride) # number y grid points
        create_grids(self, img_size, (nx, ny))

    def forward(self, p, img_size, var=None):
        """
        onnx 代表开放式神经网络交换
        pytorch 中的模型都可以导出或转换为标准 ONNX 格式
        在模型采用 ONNX 格式后,即可在各种平台和设备上运行
        在这里 ONNX 代表规范化的推理过程
        """
        if ONNX_EXPORT:
            bs = 1 # batch size
        else:
            bs, _, ny, nx = p.shape # bs, 255, 13, 13
            if (self.nx, self.ny) != (nx, ny):
                create_grids(self, img_size, (nx, ny), p.device, p.dtype)

            # p.view(bs, 255, 13, 13) --> (bs, 3, 13, 13, 85)
            # (bs, anchors, grid, grid, classes + xywh)
            p = p.view(bs, self.na, self.no, self.ny,
                       self.nx).permute(0, 1, 3, 4, 2).contiguous()

        if self.training:
            return p

```

在理解以上代码的时候,需要理解每一个通道所代表的意义,原先的P是由上一层卷积得到的 feature map,形状为(以 80 个类别、输入 416、下采样 32 倍为例):【batch size, anchor×(80+5), 13, 13】,在训练的过程中,将 feature map 通过张量操作转化的形状为:【batch size, anchor, 13, 13, 85】。

测试过程:

```

# p 的形状目前为:【bs, anchor_num, gridx,gridy,xywhc+class】
else: # 测试推理过程
    # s = 1.5 # scale_xy (pxy = pxy * s - (s - 1) / 2)
    io = p.clone() # 测试过程输出就是 io
    io[..., :2] = torch.sigmoid(io[..., :2]) + self.grid_xy # xy
    # grid_xy 是左上角再加上偏移量 io[..., :2] 代表 xy 偏移
    io[..., 2:4] = torch.exp(
        io[..., 2:4]) * self.anchor_wh # wh yolo method
    # io[..., 2:4] = ((torch.sigmoid(io[..., 2:4]) * 2) ** 3) *
    ↪ self.anchor_wh
    # wh power method
    io[..., :4] *= self.stride

```

```

if 'default' in self.arc: # seperate obj and cls
    torch.sigmoid_(io[..., 4])
elif 'BCE' in self.arc: # unified BCE (80 classes)
    torch.sigmoid_(io[..., 5:])
    io[..., 4] = 1
elif 'CE' in self.arc: # unified CE (1 background + 80 classes)
    io[..., 4:] = F.softmax(io[..., 4:], dim=4)
    io[..., 4] = 1

if self.nc == 1:
    io[..., 5] = 1
    # single-class model https://github.com/ultralytics/yolov3/issues/235

# reshape from [1, 3, 13, 13, 85] to [1, 507, 85]
return io.view(bs, -1, self.no), p

```

理解以上内容是需要对应以下公式：

$$b_x = \sigma(t_x) + c_x$$

$$b_y = \sigma(t_y) + c_y$$

$$b_w = p_w e^{t_x}$$

$$b_h = p_h e^{t_h}$$

xy 部分：

$$b_x = \sigma(t_x) + c_x$$

$$b_y = \sigma(t_y) + c_y$$

c_x, c_y 代表的是格子的左上角坐标； t_x, t_y 代表的是网络预测的结果； σ 代表 sigmoid 激活函数。对应代码理解：

```

io[..., :2] = torch.sigmoid(io[..., :2]) + self.grid_xy # xy
# grid_xy 是左上角再加上偏移量 io[..., :2] 代表 xy 偏移

```

wh 部分：

$$b_w = p_w e^{t_x}$$

$$b_h = p_h e^{t_h}$$

p_w, p_h 代表的是 anchor 先验框在 feature map 上对应的大小。 t_w, t_h 代表的是网络学习得到的缩放系数。对应代码理解：

```
# wh yolo method
io[..., 2:4] = torch.exp(io[..., 2:4]) * self.anchor_wh
```

class 部分：

在类别部分，提供了几种方法，根据 `arc` 参数来进行不同模式的选择。以 CE（crossEntropy）为例：

```
#io: (bs, anchors, grid, grid, xywh+classes)
io[..., 4:] = F.softmax(io[..., 4:], dim=4)# 使用 softmax
io[..., 4] = 1
```

3. 参考资料

pytorch 的官方 API

输出解码：<https://zhuanlan.zhihu.com/p/76802514>

七、在 YOLOv3 模型中添加 Attention 机制

前言：【从零开始学习 YOLOv3】系列越写越多，本来安排的内容比较少，但是在阅读代码的过程中慢慢发掘了一些新的亮点，所以不断加入到这个系列中。之前都在读 YOLOv3 中的代码，已经学习了 `cfg` 文件、模型构建等内容。本文在之前的基础上，对模型的代码进行修改，将之前 Attention 系列中的 SE 模块和 CBAM 模块集成到 YOLOv3 中。

1. 规定格式

正如 `[convolutional],[maxpool],[net],[route]` 等层在 `cfg` 中的定义一样，我们再添加全新的模块的时候，要规定一下 `cfg` 的格式。做出以下规定：

在 SE 模块（具体讲解见：【cv 中的 Attention 机制】最简单最易实现的 SE 模块）中，有一个参数为 `reduction`，这个参数默认是 16，所以在这个模块中的详细参数我们按照以下内容进行设置：

```
[se]
reduction=16
```

在 CBAM 模块（具体讲解见：【CV 中的 Attention 机制】ECCV 2018 Convolutional Block Attention Module）中，空间注意力机制和通道注意力机制中共存在两个参数：ratio 和 kernel_size，所以这样规定 CBAM 在 cfg 文件中的格式：

```
[cbam]
ratio=16
kernel_size=7
```

2. 修改解析部分

由于我们添加的这些参数都是自定义的，所以需要修改解析 cfg 文件的函数，之前讲过，需要修改 parse_config.py 中的部分内容：

```
def parse_model_cfg(path):
    # path 参数为: cfg/yolov3-tiny.cfg
    if not path.endswith('.cfg'):
        path += '.cfg'
    if not os.path.exists(path) and \
        os.path.exists('cfg' + os.sep + path):
        path = 'cfg' + os.sep + path

    with open(path, 'r') as f:
        lines = f.read().split('\n')

    # 去除以 # 开头的，属于注释部分的内容
    lines = [x for x in lines if x and not x.startswith('#')]
    lines = [x.rstrip().lstrip() for x in lines]
    mdefs = [] # 模块的定义
    for line in lines:
        if line.startswith('['): # 标志着一个模块的开始
            '''
            eg:
            [shortcut]
            from=-3
            activation=linear
            '''
            mdefs.append({})
            mdefs[-1]['type'] = line[1:-1].rstrip()
            if mdefs[-1]['type'] == 'convolutional':
                mdefs[-1]['batch_normalize'] = 0
```

```

    else:
        key, val = line.split("=")
        key = key.rstrip()

        if 'anchors' in key:
            mdefs[-1][key] = np.array([float(x) for x in
↪ val.split(',')]).reshape((-1, 2))
        else:
            mdefs[-1][key] = val.strip()

# Check all fields are supported
supported = ['type', 'batch_normalize', 'filters', 'size', \
            'stride', 'pad', 'activation', 'layers', \
            'groups', 'from', 'mask', 'anchors', \
            'classes', 'num', 'jitter', 'ignore_thresh', \
            'truth_thresh', 'random', \
            'stride_x', 'stride_y']

f = [] # fields
for x in mdefs[1:]:
    [f.append(k) for k in x if k not in f]
u = [x for x in f if x not in supported] # unsupported fields
assert not any(u), "Unsupported fields %s in %s. See
↪ https://github.com/ultralytics/yolov3/issues/631" % (u, path)

return mdefs

```

以上内容中，需要改的是 `supported` 中的字段，将我们的内容添加进去：

```

supported = ['type', 'batch_normalize', 'filters', 'size', \
            'stride', 'pad', 'activation', 'layers', \
            'groups', 'from', 'mask', 'anchors', \
            'classes', 'num', 'jitter', 'ignore_thresh', \
            'truth_thresh', 'random', \
            'stride_x', 'stride_y', \
            'ratio', 'reduction', 'kernel_size']

```

3. 实现 SE 和 CBAM

具体原理还请见【cv 中的 Attention 机制】最简单最易实现的 SE 模块和【CV 中的 Attention 机制】ECCV 2018 Convolutional Block Attention Module 这两篇文章，下边直接使用以上两篇文章中的代码：

SE

```

class SELayer(nn.Module):
    def __init__(self, channel, reduction=16):
        super(SELayer, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Linear(channel, channel // reduction, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(channel // reduction, channel, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        b, c, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        y = self.fc(y).view(b, c, 1, 1)
        return x * y.expand_as(x)

```

CBAM

```

class SpatialAttention(nn.Module):
    def __init__(self, kernel_size=7):
        super(SpatialAttention, self).__init__()
        assert kernel_size in (3, 7), "kernel size must be 3 or 7"
        padding = 3 if kernel_size == 7 else 1

        self.conv = nn.Conv2d(2, 1, kernel_size, padding=padding, bias=False)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        avgout = torch.mean(x, dim=1, keepdim=True)
        maxout, _ = torch.max(x, dim=1, keepdim=True)
        x = torch.cat([avgout, maxout], dim=1)
        x = self.conv(x)
        return self.sigmoid(x)

class ChannelAttention(nn.Module):
    def __init__(self, in_planes, ratio=16):
        super(ChannelAttention, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.max_pool = nn.AdaptiveMaxPool2d(1)

        self.sharedMLP = nn.Sequential(

```

```

        nn.Conv2d(in_planes, in_planes // ratio, 1, bias=False),
        ↪ nn.ReLU(),
        nn.Conv2d(in_planes // rotio, in_planes, 1, bias=False))
    self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        avgout = self.sharedMLP(self.avg_pool(x))
        maxout = self.sharedMLP(self.max_pool(x))
        return self.sigmoid(avgout + maxout)

```

以上就是两个模块的代码，添加到 `models.py` 文件中。

4. 设计 `cfg` 文件

这里以 `yolov3-tiny.cfg` 为 baseline，然后添加注意力机制模块。

CBAM 与 SE 类似，所以以 SE 为例，添加到 `backbone` 之后的部分，进行信息重构 (refinement)。

```

[net]
# Testing
batch=1
subdivisions=1
# Training
# batch=64
# subdivisions=2
width=416
height=416
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

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

[convolutional]

```



```
batch_normalize=1
filters=16
size=3
stride=1
pad=1
activation=leaky
```

```
[maxpool]
size=2
stride=2
```

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

```
[maxpool]
size=2
stride=2
```

```
[convolutional]
batch_normalize=1
filters=64
size=3
stride=1
pad=1
activation=leaky
```

```
[maxpool]
size=2
stride=2
```

```
[convolutional]
batch_normalize=1
filters=128
size=3
stride=1
pad=1
activation=leaky
```

```
[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=1

[convolutional]
batch_normalize=1
filters=1024
size=3
stride=1
pad=1
activation=leaky

[se]
reduction=16

# 在 backbone 结束的地方添加 se 模块
#####backbone#####

[convolutional]
```

```
batch_normalize=1
filters=256
size=1
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=18
activation=linear

[yolo]
mask = 3,4,5
anchors = 10,14, 23,27, 37,58, 81,82, 135,169, 344,319
classes=1
num=6
jitter=.3
ignore_thresh = .7
truth_thresh = 1
random=1

[route]
layers = -4

[convolutional]
batch_normalize=1
filters=128
size=1
stride=1
pad=1
```

```
activation=leaky

[upsample]
stride=2

[route]
layers = -1, 8

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=18
activation=linear

[yolo]
mask = 0,1,2
anchors = 10,14, 23,27, 37,58, 81,82, 135,169, 344,319
classes=1
num=6
jitter=.3
ignore_thresh = .7
truth_thresh = 1
random=1
```

5. 模型构建

以上都是准备工作，以 SE 为例，我们修改 `model.py` 文件中的模型加载部分，并修改 `forward` 函数部分的代码，让其正常发挥作用：

在 `model.py` 中的 `create_modules` 函数中进行添加：

```
elif mdef['type'] == 'se':
    modules.add_module(
```

```

        'se_module',
        SELayer(output_filters[-1],
        ↪ reduction=int(mdef['reduction'])))

```

然后修改 Darknet 中的 forward 部分的函数：

```

def forward(self, x, var=None):
    img_size = x.shape[-2:]
    layer_outputs = []
    output = []

    for i, (mdef,
            module) in enumerate(zip(self.module_defs, self.module_list)):
        mtype = mdef['type']
        if mtype in ['convolutional', 'upsample', 'maxpool']:
            x = module(x)
        elif mtype == 'route':
            layers = [int(x) for x in mdef['layers'].split(',')]
            if len(layers) == 1:
                x = layer_outputs[layers[0]]
            else:
                try:
                    x = torch.cat([layer_outputs[i] for i in layers], 1)
                except: # apply stride 2 for darknet reorg layer
                    layer_outputs[layers[1]] = F.interpolate(
                        layer_outputs[layers[1]], scale_factor=[0.5, 0.5])
                    x = torch.cat([layer_outputs[i] for i in layers], 1)

        elif mtype == 'shortcut':
            x = x + layer_outputs[int(mdef['from'])]
        elif mtype == 'yolo':
            output.append(module(x, img_size))
            layer_outputs.append(x if i in self.routes else [])

```

在 forward 中加入 SE 模块，其实很简单。SE 模块与卷积层，上采样，最大池化层地位是一样的，不需要更多操作，只需要将以上部分代码进行修改：

```

for i, (mdef,
        module) in enumerate(zip(self.module_defs, self.module_list)):
    mtype = mdef['type']
    if mtype in ['convolutional', 'upsample', 'maxpool', 'se']:
        x = module(x)

```

CBAM 的整体过程类似，可以自己尝试一下，顺便熟悉一下 YOLOv3 的整体流程。

后记：本文的内容很简单，只是添加了注意力模块，很容易实现。不过具体注意力机制的位置、放多少个模块等都需要做实验来验证。注意力机制并不是万金油，需要多调参，多尝试才能得到满意的结果。

八、YOLOv3 中 Loss 部分计算

YOLOv1 是一个 anchor-free 的，从 YOLOv2 开始引入了 Anchor，在 VOC2007 数据集上将 mAP 提升了 10 个百分点。YOLOv3 也继续使用了 Anchor，本文主要讲 ultralytics 版 YOLOv3 的 Loss 部分的计算，实际上这部分 loss 和原版差距非常大，并且可以通过 arc 指定 loss 的构建方式，如果想看原版的 loss 可以在下方 release 的 v6 中下载源码。

Github 地址: <https://github.com/ultralytics/yolov3>

Github release: <https://github.com/ultralytics/yolov3/releases>

1. Anchor

Faster R-CNN 中 Anchor 的大小和比例是由人手工设计的，可能并不贴合数据集，有可能会给模型性能带来负面影响。YOLOv2 和 YOLOv3 则是通过聚类算法得到最适合的 k 个框。聚类距离是通过 IoU 来定义，IoU 越大，边框距离越近。

$$d(box, centroid) = 1 - IoU(box, centroid)$$

Anchor 越多，平均 IoU 会越大，效果越好，但是会带来计算量上的负担，下图是 YOLOv2 论文中的聚类数量和平均 IoU 的关系图，在 YOLOv2 中选择了 5 个 anchor 作为精度和速度的平衡。

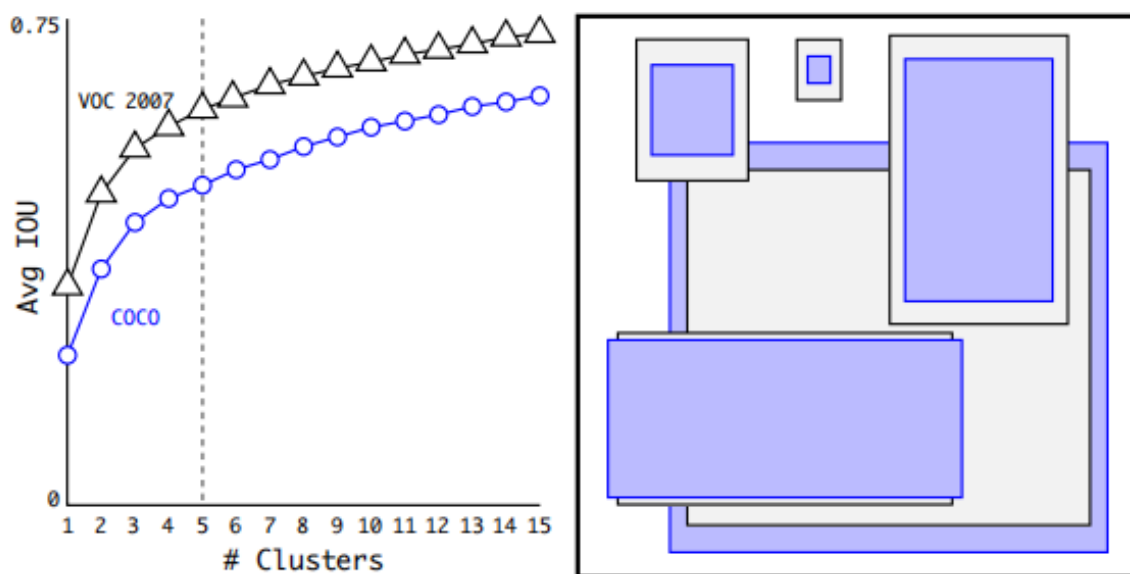


Figure 2: Clustering box dimensions on VOC and COCO. We run k-means clustering on the dimensions of bounding boxes to get good priors for our model. The left image shows the average IOU we get with various choices for k . We find that $k = 5$ gives a good tradeoff for recall vs. complexity of the model. The right image shows the relative centroids for VOC and COCO. Both sets of priors favor thinner, taller boxes while COCO has greater variation in size than VOC.

Figure 1: YOLOv2 中聚类 Anchor 数量和 IoU 的关系图

2. 偏移公式

在 Faster RCNN 中，中心坐标的偏移公式是：

$$\begin{cases} x = (t_x \times w_a) + x_a \\ y = (t_y \times h_a) + y_a \end{cases}$$

其中 x_a 、 y_a 代表中心坐标， w_a 和 h_a 代表宽和高， t_x 和 t_y 是模型预测的 Anchor 相对于 Ground Truth 的偏移量，通过计算得到的 x, y 就是最终预测框的中心坐标。

而在 YOLOv2 和 YOLOv3 中，对偏移量进行了限制，如果不限制偏移量，那么边框的中心可以在图像

任何位置，可能导致训练的不稳定。

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

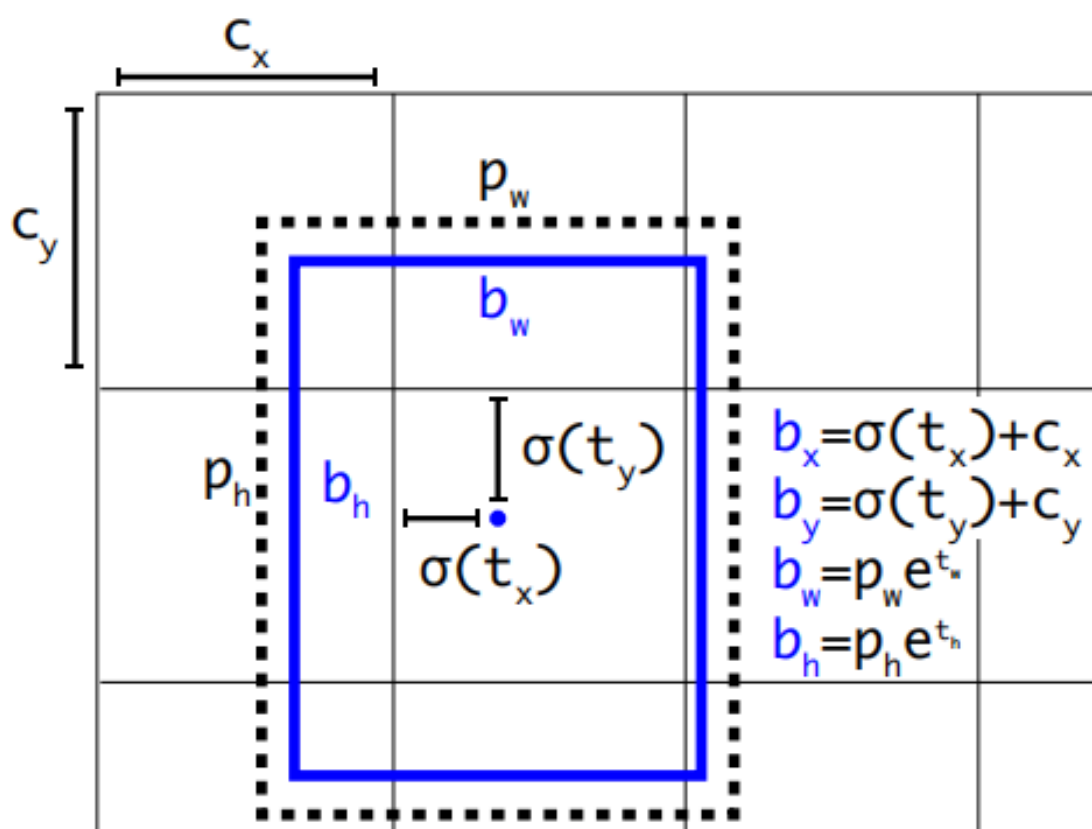


Figure 3: Bounding boxes with dimension priors and location prediction. We predict the width and height of the box as offsets from cluster centroids. We predict the center coordinates of the box relative to the location of filter application using a sigmoid function.

https://blog.csdn.net/DD_PP_JJ

Figure 2: 公式对应的意义

对照上图进行理解：

- c_x 和 c_y 分别代表中心点所处区域的左上角坐标。
- p_w 和 p_h 分别代表 Anchor 的宽和高。
- $\sigma(t_x)$ 和 $\sigma(t_y)$ 分别代表预测框中心点和左上角的距离， σ 代表 sigmoid 函数，将偏移量限制在当前 grid 中，有利于模型收敛。
- t_w 和 t_h 代表预测的宽高偏移量，Anchor 的宽和高乘上指数化后的宽高，对 Anchor 的长宽进行调整。
- $\sigma(t_o)$ 是置信度预测值，是当前框有目标的概率乘以 bounding box 和 ground truth 的 IoU 的结果

3. Loss

YOLOv3 中有一个参数是 `ignore_thresh`，在 ultralytics 版版的 YOLOv3 中对应的是 `train.py` 文件中的 `iou_t` 参数（默认为 0.225）。

正负样本是按照以下规则决定的：

- 如果一个预测框与所有的 Ground Truth 的最大 $\text{IoU} < \text{ignore_thresh}$ 时，那这个预测框就是负样本。
- 如果 Ground Truth 的中心点落在一个区域中，该区域就负责检测该物体。将与该物体有最大 IoU 的预测框作为正样本（注意这里没有用到 `ignore thresh`，即使该最大 $\text{IoU} < \text{ignore thresh}$ 也不会影响该预测框为正样本）

在 YOLOv3 中，Loss 分为三个部分：

- 一个是 xywh 部分带来的误差，也就是 bbox 带来的 loss
- 一个是置信度带来的误差，也就是 obj 带来的 loss
- 最后一个类别带来的误差，也就是 class 带来的 loss

在代码中分别对应 `lbox`, `lobj`, `lcls`，yolov3 中使用的 loss 公式如下：

$$\begin{aligned}
lbox &= \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{i,j}^{obj} (2 - w_i \times h_i) [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + (w_i - \hat{w}_i)^2 + (h_i - \hat{h}_i)^2] \\
lcls &= \lambda_{class} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{i,j}^{obj} \sum_{c \in classes} p_i(c) \log(\hat{p}_i(c)) \\
lobj &= \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{i,j}^{noobj} (c_i - \hat{c}_i)^2 + \lambda_{obj} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{i,j}^{obj} (c_i - \hat{c}_i)^2 \\
loss &= lbox + lobj + lcls
\end{aligned}$$

其中：

S: 代表 grid size, S^2 代表 13x13, 26x26, 52x52

B: box

$1_{i,j}^{obj}$: 如果在 i,j 处的 box 有目标, 其值为 1, 否则为 0

$1_{i,j}^{noobj}$: 如果在 i,j 处的 box 没有目标, 其值为 1, 否则为 0

BCE (binary cross entropy) 具体计算公式如下：

$$BCE(\hat{c}_i, c_i) = -\hat{c}_i \times \log(c_i) - (1 - \hat{c}_i) \times \log(1 - c_i)$$

以上是论文中 yolov3 对应的 darknet。而 pytorch 版本的 yolov3 改动比较大, 有较大的改动空间, 可以通过参数进行调整。

分成三个部分进行具体分析：

1. lbox 部分

在 ultralytics 版版的 YOLOv3 中, 使用的是 GIoU, 具体讲解见 GIoU 讲解链接。

简单来说是这样的公式, IoU 公式如下：

$$IoU = \frac{|A \cap B|}{|A \cup B|}$$

而 GIoU 公式如下：

$$GIoU = IoU - \frac{|A_c - U|}{|A_c|}$$

其中 A_c 代表两个框最小闭包区域面积, 也就是同时包含了预测框和真实框的最小框的面积。

yolov3 中提供了 IoU、GIoU、DIoU 和 CIoU 等计算方式, 以 GIoU 为例：

if GIoU: # Generalized IoU <https://arxiv.org/pdf/1902.09630.pdf>

```

c_area = cw * ch + 1e-16 # convex area
return iou - (c_area - union) / c_area # GIoU

```

可以看到代码和 GIoU 公式是一致的，再来看一下 lbox 计算部分：

```

giou = bbox_iou(pbox.t(), tbox[i],
                 x1y1x2y2=False, GIoU=True)
lbox += (1.0 - giou).sum() if red == 'sum' else (1.0 - giou).mean()

```

可以看到 box 的 loss 是 1-giou 的值。

2. lobj 部分

lobj 代表置信度，即该 bounding box 中是否含有物体的概率。在 yolov3 代码中 obj loss 可以通过 arc 来指定，有两种模式：

如果采用 default 模式，使用 BCEWithLogitsLoss，将 obj loss 和 cls loss 分开计算：

```

BCEobj = nn.BCEWithLogitsLoss(pos_weight=ft([h['obj_pw']]), reduction=red)
if 'default' in arc: # separate obj and cls
    lobj += BCEobj(pi[..., 4], tobj) # obj loss
    # pi[...,4] 对应的是该框中含有目标的置信度，和 giou 计算 BCE
    # 相当于将 obj loss 和 cls loss 分开计算

```

如果采用 BCE 模式，使用的也是 BCEWithLogitsLoss，计算对象是所有的 cls loss：

```

BCE = nn.BCEWithLogitsLoss(reduction=red)
elif 'BCE' in arc: # unified BCE (80 classes)
    t = torch.zeros_like(pi[..., 5:]) # targets
    if nb:
        t[b, a, gj, gi, tcls[i]] = 1.0 # 对应正样本 class 置信度设置为 1
        lobj += BCE(pi[..., 5:], t) # pi[...,5:] 对应的是所有的 class

```

3. lcls 部分

如果是单类的情况，cls loss=0

如果是多类的情况，也分两个模式：

如果采用 default 模式，使用的是 BCEWithLogitsLoss 计算 class loss。

```

BCEcls = nn.BCEWithLogitsLoss(pos_weight=ft([h['cls_pw']]), reduction=red)
# cls loss 只计算多类之间的 loss，单类不计算
if 'default' in arc and model.nc > 1:
    t = torch.zeros_like(ps[:, 5:]) # targets
    t[range(nb), tcls[i]] = 1.0 # 设置对应 class 为 1
    lcls += BCEcls(ps[:, 5:], t) # 使用 BCE 计算分类 loss

```

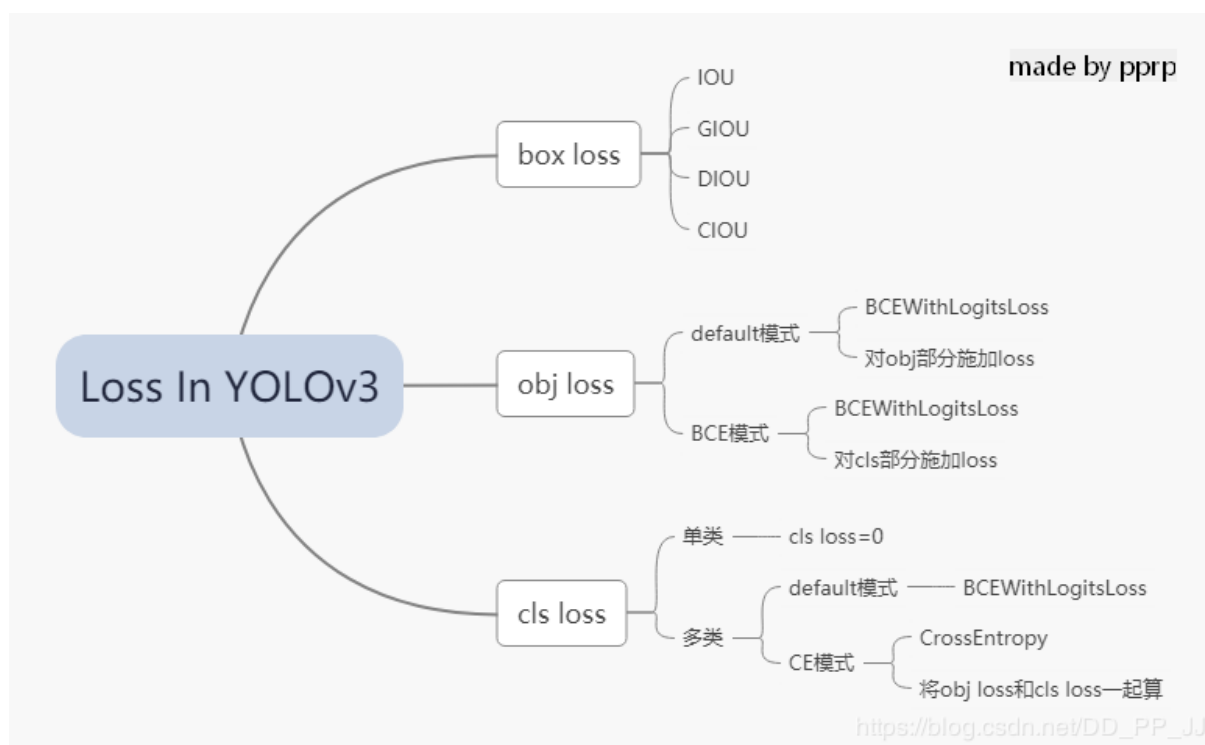
如果采用 CE 模式，使用的是 CrossEntropy 同时计算 obj loss 和 cls loss。

```

CE = nn.CrossEntropyLoss(reduction=red)
elif 'CE' in arc: # unified CE (1 background + 80 classes)
    t = torch.zeros_like(pi[..., 0], dtype=torch.long) # targets
    if nb:
        t[b, a, gj, gi] = tcls[i] + 1 # 由于 cls 是从零开始计数的, 所以 +1
        lcls += CE(pi[..., 4:].view(-1, model.nc + 1), t.view(-1))
    # 这里将 obj loss 和 cls loss 一起计算, 使用 CrossEntropy Loss

```

以上三部分总结下来就是下图:



4. 代码

ultralytics 版的 yolov3 的 loss 已经和论文中提出的部分大相径庭了, 代码中很多地方地方是来自作者的经验。另外, 这里读的代码是 2020 年 2 月份左右作者发布的版本, 关注这个库的人会知道, 作者更新速度非常快, 在笔者写这篇文章的时候, loss 也出现了大幅改动, 添加了 label smoothing 等新的机制, 去掉了通过 arc 来调整 loss 的机制, 简化了 loss 部分。

这部分的代码添加了大量注释, 很多是笔者通过 debug 得到的结果, 理解的时候需要讲一下 debug 的配置:

- 单类数据集 class=1
- batch size=2

- 模型是 yolov3.cfg

计算 loss 这部分代码可以大概上分为两部分，一部分是正负样本选取，一部分是 loss 计算。

1. 正负样本选取部分

这部分主要工作是在每个 yolo 层将预设的 anchor 和 ground truth 进行匹配，得到正样本，回顾一下上文中在 YOLOv3 中正负样本选取规则：

- 如果一个预测框与所有的 Ground Truth 的最大 IoU < ignore_thresh 时，那这个预测框就是负样本。
- 如果 Ground Truth 的中心点落在一个区域中，该区域就负责检测该物体。将与该物体有最大 IoU 的预测框作为正样本（注意这里没有用到 ignore thresh, 即使该最大 IoU < ignore thresh 也不会影响该预测框为正样本）

```
def build_targets(model, targets):
    # targets = [image, class, x, y, w, h]
    # 这里的 image 是一个数字，代表是当前 batch 的第几个图片
    # x,y,w,h 都进行了归一化，除以了宽或者高

    nt = len(targets)

    tcls, tbox, indices, av = [], [], [], []

    multi_gpu = type(model) in (nn.parallel.DataParallel,
                                 nn.parallel.DistributedDataParallel)

    reject, use_all_anchors = True, True
    for i in model.yolo_layers:
        # yolov3.cfg 中有三个 yolo 层，这部分用于获取对应 yolo 层的 grid 尺寸和
        #   anchor 大小
        # ng 代表 num of grid (13,13) anchor_vec [[x,y],[x,y]]
        # 注意这里的 anchor_vec: 假如现在是 yolo 第一个层 (downsample rate=32)
        # 这一层对应 anchor 为: [116, 90], [156, 198], [373, 326]
        # anchor_vec 实际值为以上除以 32 的结果:
        #   [3.6, 2.8], [4.875, 6.18], [11.6, 10.1]
        # 原图 416x416 对应的 anchor 为 [116, 90]
        # 下采样 32 倍后 13x13 对应的 anchor 为 [3.6, 2.8]
        if multi_gpu:
            ng = model.module.module_list[i].ng
            anchor_vec = model.module.module_list[i].anchor_vec
        else:
            ng = model.module_list[i].ng,
            anchor_vec = model.module_list[i].anchor_vec
```

```

# iou of targets-anchors
# targets 中保存的是 ground truth
t, a = targets, []

gwh = t[:, 4:6] * ng[0]

if nt: # 如果存在目标
    # anchor_vec: shape = [3, 2] 代表 3 个 anchor
    # gwh: shape = [2, 2] 代表 2 个 ground truth
    # iou: shape = [3, 2] 代表 3 个 anchor 与对应的两个 ground truth 的 iou
    iou = wh_iou(anchor_vec, gwh) # 计算先验框和 GT 的 iou

    if use_all_anchors:
        na = len(anchor_vec) # number of anchors
        a = torch.arange(na).view(
            (-1, 1)).repeat([1, nt]).view(-1) # 构造 3x2 -> view 到 6
        # a = [0,0,1,1,2,2]
        t = targets.repeat([na, 1])
        # targets: [image, cls, x, y, w, h]
        # 复制 3 个: shape[2,6] to shape[6,6]
        gwh = gwh.repeat([na, 1])
        # gwh shape:[6,2]
    else: # use best anchor only
        iou, a = iou.max(0) # best iou and anchor
        # 取 iou 最大值是 darknet 的默认做法, 返回的 a 是下角标

# reject anchors below iou_thres (OPTIONAL, increases P, lowers R)
if reject:
    # 在这里将所有阈值小于 ignore thresh 的去掉
    j = iou.view(-1) > model.hyp['iou_t']
    # iou threshold hyperparameter
    t, a, gwh = t[j], a[j], gwh[j]

# Indices
b, c = t[:, :2].long().t() # target image, class
# 取的是 targets[image, class, x,y,w,h] 中 [image, class]

gxy = t[:, 2:4] * ng[0] # grid x, y

gi, gj = gxy.long().t() # grid x, y indices
# 注意这里通过 long 将其转化为整形, 代表格子的左上角

```

```

indices.append((b, a, gj, gi))
# indice 结构体保存内容为:
'''
b: 一个 batch 中的角标
a: 代表所选中的正样本的 anchor 的下角标
gj, gi: 代表所选中的 grid 的左上角坐标
'''

# Box
gxy -= gxy.floor() # xy
# 现在 gxy 保存的是偏移量, 是需要 YOLO 进行拟合的对象
tbox.append(torch.cat((gxy, gwh), 1)) # xywh (grids)
# 保存对应偏移量和宽高 (对应 13x13 大小的)
av.append(anchor_vec[a]) # anchor vec
# av 是 anchor vec 的缩写, 保存的是匹配上的 anchor 的列表

# Class
tcls.append(c)
# tcls 用于保存匹配上的类别列表
if c.shape[0]: # if any targets
    assert c.max() < model.nc, 'Model accepts %g classes labeled from
        ↪ 0-%g, however you labelled a class %g. ' \
        'See
        ↪ https://github.com/ultralytics/yolov3/wiki/Tr
        ↪ Custom-Data' %
        ↪ (
            model.nc, model.nc - 1, c.max())

return tcls, tbox, indices, av

```

梳理一下在每个 YOLO 层的匹配流程:

- 将 ground truth 和 anchor 进行匹配, 得到 iou
- 然后有两个方法匹配:
 - 使用 yolov3 原版的匹配机制, 仅仅选择 iou 最大的作为正样本
 - 使用 ultralytics 版 yolov3 的默认匹配机制, use_all_anchors=True 的时候, 选择所有的匹配对
- 对以上匹配的部分在进行筛选, 对应原版 yolo 中 ignore_thresh 部分, 将以上匹配到的部分中 iou<ignore_thresh 的部分筛选掉。
- 最后将匹配得到的内容返回到 compute_loss 函数中。

2. loss 计算部分

这部分就是 yolov3 中核心 loss 计算, 这部分对照上文的讲解进行理解。

```
def compute_loss(p, targets, model):
    # p: (bs, anchors, grid, grid, classes + xywh)
    # predictions, targets, model
    ft = torch.cuda.FloatTensor if p[0].is_cuda else torch.Tensor
    lcls, lbox, lobj = ft([0]), ft([0]), ft([0])
    tcls, tbox, indices, anchor_vec = build_targets(model, targets)
    '''
    以 yolov3 为例, 有三个 yolo 层
    tcls: 一个 list 保存三个 tensor, 每个 tensor 中有 6(2 个 gtx3 个 anchor) 个代表
    ↪ 类别的数字
    tbox: 一个 list 保存三个 tensor, 每个 tensor 形状 [6,4], 6(2 个 gtx3 个
    ↪ anchor) 个 bbox
    indices: 一个 list 保存三个 tuple, 每个 tuple 中保存 4 个 tensor:
            分别代表          b: 一个 batch 中的角标
                            a: 代表所选中的正样本的 anchor 的下角标
                            gj, gi: 代表所选中的 grid 的左上角坐标
    anchor_vec: 一个 list 保存三个 tensor, 每个 tensor 形状 [6,2],
                6(2 个 gtx3 个 anchor) 个 anchor, 注意大小是相对于 13x13feature
    ↪ map 的 anchor 大小
    '''

    h = model.hyp # hyperparameters
    arc = model.arc # # (default, uCE, uBCE) detection architectures
    # 具体使用的损失函数是通过 arc 参数决定的
    red = 'sum' # Loss reduction (sum or mean)

    # Define criteria
    BCEcls = nn.BCEWithLogitsLoss(pos_weight=ft([h['cls_pw']]]),
    ↪ reduction=red)
    BCEobj = nn.BCEWithLogitsLoss(pos_weight=ft([h['obj_pw']]]),
    ↪ reduction=red)
    #BCEWithLogitsLoss = sigmoid + BCELoss
    BCE = nn.BCEWithLogitsLoss(reduction=red)
    CE = nn.CrossEntropyLoss(reduction=red) # weight=model.class_weights

    # class label smoothing https://arxiv.org/pdf/1902.04103.pdf eqn 3
    # cp, cn = smooth_BCE(eps=0.0)
    # 这是最新的版本中提供了 label smoothing 的功能, 只能用在多类问题

    if 'F' in arc: # add focal loss
        g = h['fl_gamma']
        BCEcls, BCEobj, BCE, CE = FocalLoss(BCEcls, g), FocalLoss(
            BCEobj, g), FocalLoss(BCE, g), FocalLoss(CE, g)
```



```

    # focal loss 可以用在 cls loss 或者 obj loss

# Compute losses
np, ng = 0, 0 # number grid points, targets
# np 这个命名真的迷, 建议改一下和 numpy 缩写重复
for i, pi in enumerate(p): # layer index, layer predictions
    # 在 yolov3 中, p 有三个 yolo layer 的输出 pi
    # 形状为:(bs, anchors, grid, grid, classes + xywh)
    b, a, gj, gi = indices[i] # image, anchor, gridy, gridx
    tobj = torch.zeros_like(pi[..., 0])
    # tobj = target obj, 形状为 (bs, anchors, grid, grid)
    np += tobj.numel() # 返回 tobj 中元素个数

# Compute losses
nb = len(b)
if nb:
    ng += nb # number of targets 用于最后算平均 loss
    # (bs, anchors, grid, grid, classes + xywh)
    ps = pi[b, a, gj, gi] # 即找到了对应目标的 classes+xywh, 形状为
    ↪ [6(2x3),6]

    # GIoU
    pxy = torch.sigmoid(
        ps[:, 0:2] # 将 x,y 进行 sigmoid
    ) # pxy = pxy * s - (s - 1) / 2, s = 1.5 (scale_xy)
    pwh = torch.exp(ps[:, 2:4]).clamp(max=1E3) * anchor_vec[i]
    # 防止溢出进行 clamp 操作, 乘以 13x13feature map 对应的 anchor
    # 这部分和上文中偏移公式是一致的
    pbox = torch.cat((pxy, pwh), 1) # predicted box
    # pbox: predicted bbox shape:[6, 4]
    giou = bbox_iou(pbox.t(), tobj[i], x1y1x2y2=False,
                    GIoU=True) # giou computation
    # 计算 giou loss, 形状为 6
    lbox += (1.0 - giou).sum() if red == 'sum' else (1.0 -
    ↪ giou).mean()
    # bbox loss 直接由 giou 决定
    tobj[b, a, gj, gi] = giou.detach().type(tobj.dtype)
    # target obj 用 giou 取代 1, 代表该点对应置信度

# cls loss 只计算多类之间的 loss, 单类不计算
if 'default' in arc and model.nc > 1:
    t = torch.zeros_like(ps[:, 5:]) # targets
    t[range(nb), tcls[i]] = 1.0 # 设置对应 class 为 1

```

```

        lcls += BCEcls(ps[:, 5:], t) # 使用 BCE 计算分类 loss

    if 'default' in arc: # separate obj and cls
        lobj += BCEobj(pi[..., 4], tobj) # obj loss
        # pi[...,4] 对应的是该框中含有目标的置信度, 和 giou 计算 BCE
        # 相当于将 obj loss 和 cls loss 分开计算

    elif 'BCE' in arc: # unified BCE (80 classes)
        t = torch.zeros_like(pi[..., 5:]) # targets
        if nb:
            t[b, a, gj, gi, tcls[i]] = 1.0 # 对应正样本 class 置信度设置为 1
            lobj += BCE(pi[..., 5:], t)
            # pi[...,5:] 对应的是所有的 class

    elif 'CE' in arc: # unified CE (1 background + 80 classes)
        t = torch.zeros_like(pi[..., 0], dtype=torch.long) # targets
        if nb:
            t[b, a, gj, gi] = tcls[i] + 1 # 由于 cls 是从零开始计数的, 所以 +1
            lcls += CE(pi[..., 4:].view(-1, model.nc + 1), t.view(-1))
            # 这里将 obj loss 和 cls loss 一起计算, 使用 CrossEntropy Loss
# 使用对应的权重来平衡, 这个参数是作者通过参数搜索 (random search) 的方法搜索得到的
lbox *= h['giou']
lobj *= h['obj']
lcls *= h['cls']

if red == 'sum':
    bs = tobj.shape[0] # batch size
    lobj *= 3 / (6300 * bs) * 2
    # 6300 = (10 ** 2 + 20 ** 2 + 40 ** 2) * 3
    # 输入为 320x320 的图片, 则存在 6300 个 anchor
    # 3 代表 3 个 yolo 层, 2 是一个超参数, 通过实验获取
    # 如果不想计算的话, 可以修改 red='mean'
    if ng:
        lcls *= 3 / ng / model.nc
        lbox *= 3 / ng
    loss = lbox + lobj + lcls
    return loss, torch.cat((lbox, lobj, lcls, loss)).detach()

```

需要注意的是, 三个部分的 loss 的平衡权重不是按照 yolov3 原文的设置来做的, 是通过超参数进化来搜索得到的, 具体请看: 【从零开始学习 YOLOv3】4. YOLOv3 中的参数进化

5. 补充

补充一下 BCEWithLogitsLoss 的用法，在这之前先看一下 BCELoss:

`torch.nn.BCELoss` 的功能是二分类任务是交叉熵计算函数，可以认为是 `CrossEntropy` 的特例。其分类限定为二分类，`y` 的值必须为 `{0,1}`，`input` 应该是概率分布的形式。在使用 `BCELoss` 前一般会先加一个 `sigmoid` 激活层，常用在自编码器中。

计算公式:

$$l_n = -w_n[y_n \log(x_n) + (1 - y_n) \log(1 - x_n)]$$

w_n 是每个类别的 loss 权重，用于类别不均衡问题。

`torch.nn.BCEWithLogitsLoss` 的相当于 `Sigmoid+BCELoss`，即 `input` 会经过 `Sigmoid` 激活函数，将 `input` 变为概率分布的形式。

计算公式:

$$l_n = -w_n[y_n \log \sigma(x_n) + (1 - y_n) \log(1 - \sigma(x_n))]$$