# 代码流程

## 1.入口脚本：toolos/train\_net.py

## 2.入口函数：train\_model()：lib/utils/train.py

### create\_model() ：lib/modeling/model\_builder.py 创建模型

1、output\_dir = **get\_output\_dir**() # 模型输出路径

2、if cfg.TRAIN.AUTO\_RESUME: # True， 自动断点恢复训练

[1] if os.path.exists(final\_path): # 检查是否存在最终已训练好的

模型 model\_final.pkl

[2] files = os.listdir(output\_dir) # 找到最近的检查点（最高迭代

次数），并从下一次开始训练

[3] weights\_file = os.path.join(output\_dir, resume\_weights\_file)

# 根据找到的检查点替换初始化权重文件

3、model = model\_builder.**create**(cfg.MODEL.TYPE, train=True) #

cfg.MODEL.TYPE = generalized\_rcnn

4、if cfg.MEMONGER: **optimize\_memory**(model)    # GPU显存优化

5、workspace.**RunNetOnce**(model.param\_init\_net)  # 随机初始化模型权重

#### build\_generic\_retinanet\_model()

lib/modeling/model\_builder.py

build\_generic\_retinanet\_model(model,get\_func(cfg.**MODEL.CONV\_BODY**))

# CONV\_BODY: **FPN.add\_fpn\_ResNet101\_conv5\_body FPN.py**

##### \_single\_gpu\_build\_func(model)

###### add\_conv\_body\_func(model)：加载res101+fpn网络、model\_builder.py

blobs, dim, spatial\_scales = **add\_conv\_body\_func**(model)

-> **add\_fpn\_ResNet101\_conv5\_body**(model):**lib/modeling/FPN.py**

-> **add\_fpn\_onto\_conv\_body()**:**lib/modeling/FPN.py**

add\_ResNet101\_conv5\_body: lib/modeling/ResNet.py、加载res101网络结构

**add\_ResNet\_convX\_body**(model, (3, 4, 23, 3),freeze\_at=2) (备注：res2:3,res3:4,res4:23,res5:3, freeze\_at=2: res3之前梯度不传播 )

* **add\_stage()**
* **add\_residual\_block() #添加那个residual\_block**
* **tr =bottleneck\_transformation()、sc = add\_shortcut()、tr+sc**

**代码如下：**

p = model.**Conv(**'data', 'conv1', 3, 64, 7, pad=3, stride=2, no\_bias=1)#(7\*7,64,s=2) **图片缩一倍**

p = model.**AffineChannel**(p, 'res\_conv1\_bn', dim=64, inplace=True)

p = model.**Relu**(p, p)

p = model.**MaxPool**(p, 'pool1', kernel=3, pad=1, stride=2)

dim\_in = 64

dim\_bottleneck = cfg.RESNETS.NUM\_GROUPS \* cfg.RESNETS.WIDTH\_PER\_GROUP #1 ==> ResNet; > 1 ==> ResNeXt

(n1, n2, n3) = block\_counts[:3] #(3,4,23)

s, dim\_in = **add\_stage**(model, '**res2**', p, n1, dim\_in, 256, dim\_bottleneck, 1)

s, dim\_in = **add\_stage**(model, '**res3**', s, n2, dim\_in, 512, dim\_bottleneck \* 2, 1)

s, dim\_in = **add\_stage**( model, '**res4**', s, n3, dim\_in, 1024, dim\_bottleneck \* 4, 1)

**或**

s, dim\_in = **add\_stage**(model, '**res5**', s, n4, dim\_in, 2048, dim\_bottleneck \* 8,

cfg.RESNETS.RES5\_DILATION)

**（'res5\_2\_sum', 'res4\_22\_sum', 'res3\_3\_sum', 'res2\_2\_sum'）**

**备注：**

**add\_stage**(

model,

prefix, #层名的前缀，比如‘res2’

blob\_in, #层名,比gpu\_0/pool1、gpu\_0/res2\_2\_sum、gpu\_0/res3\_3\_sum、gpu\_0/res4\_22\_sum

n, #bottlenet的个数

dim\_in, #bottlenet的输入通道数，比如 64

dim\_out, #bottlenet的输出通道数，比如 64，dim\_in=dim\_out

dim\_inner,# =dim\_bottleneck\*2^n (n=1,2,4,8)

dilation,

stride\_init=2

)

for i in range(n):

blob\_in = **add\_residual\_block**(

model,

'{}\_{}'.format(prefix, i), # prefix = res<stage>\_<sub\_stage>, e.g., res2\_3

blob\_in, #

dim\_in,

dim\_out,

dim\_inner,

dilation,

stride\_init,

# Not using inplace for the last block;

# it may be fetched externally or used by FPN

inplace\_sum=i < n - 1

)

dim\_in = dim\_out

**bottleneck\_transformation**(

model,

blob\_in,

dim\_in,

dim\_out,

stride,

prefix,

dim\_inner,

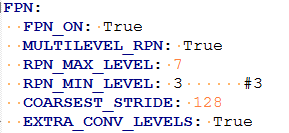
dilation=1,

group=1

):

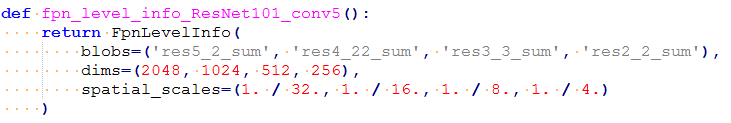
add\_fpn: lib/modeling/FPN.py :加载FPN层

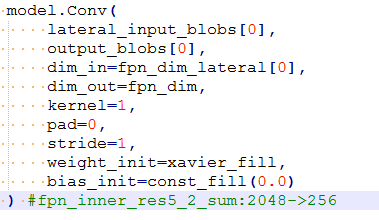
**参数：**

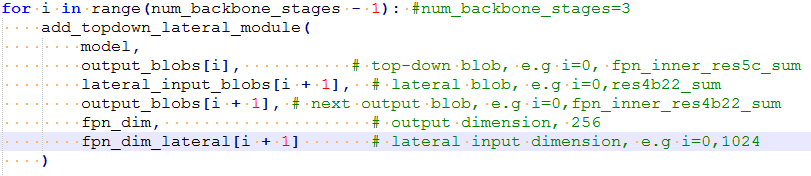


blobs\_fpn, dim\_fpn, spatial\_scales\_fpn = **add\_fpn**(model, fpn\_level\_info\_func())

**fpn\_level\_info\_func = fpn\_level\_info\_ResNet101\_conv5**







add\_topdown\_lateral\_module()函数的输出：dim=256

**i=0, fpn\_inner\_res4\_22\_sum (备注：上采样求和后也是这个名字)**

**i=1, fpn\_inner\_res3\_3\_sum**

**备注：代码中是先将'res5\_2\_sum', 'res4\_22\_sum', 'res3\_3\_sum'的通道数改成256，再上**

**采样，再求和。**

**fpn\_inner\_res5\_2\_sum: w/32,h/32, 256**

**fpn\_inner\_res4\_22\_sum: w/16,h/16, 256**

**fpn\_inner\_res3\_3\_sum: w/8,h/8, 256**

**再经过一个3\*3、s=1,p=1的不改变图片大小的卷积：**

fpn\_ res5\_2\_sum: w/32,h/32, 256

fpn\_ res4\_22\_sum: w/16,h/16, 256

fpn\_ res3\_3\_sum: w/8,h/8, 256

**备注：用3\*3的卷积核对每个融合结果进行卷积**，**目的是消除上采样的混叠效应。**

**blobs\_fpn = [**fpn\_ res5\_2\_sum、fpn\_ res4\_22\_sum、fpn\_ res3\_3\_sum**]**

**spatial\_scales=**[1/32、1/16、1/8]

**参数： EXTRA\_CONV\_LEVELS=True、RPN\_MAX\_LEVEL=7 会再增强fpn\_6、fpn\_7层**

**fpn\_6:** w/64,h/64,256**（是**fpn\_ res5\_2\_sum经过3\*3,s=2,p=1,256的卷积得到的**）**

**fpn\_7:** w/128,h/128,256**（是**fpn\_ 6经过3\*3,s=2,p=1,256的卷积得到的**）**

**最后：**

**blobs\_fpn = [**fpn\_7、fpn\_6、fpn\_ res5\_2\_sum、fpn\_ res4\_22\_sum、fpn\_ res3\_3\_sum**]**

**spatial\_scales =** [1/128、1/64、1/32、1/16、1/8]

**dim\_fpn = 256**

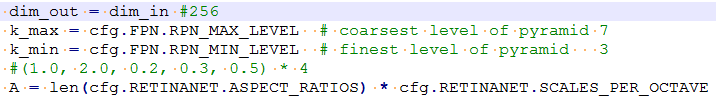
###### add\_fpn\_retinanet\_outputs：(bbox、cls)modeling/retinanet\_heads.py

retinanet\_heads.**add\_fpn\_retinanet\_outputs**(model, blobs, dim, spatial\_scales)

**blobs = [**fpn\_7、fpn\_6、fpn\_ res5\_2\_sum、fpn\_ res4\_22\_sum、fpn\_ res3\_3\_sum**]**

**spatial\_scales =** [1/128、1/64、1/32、1/16、1/8]

**dim = 256**



cls\_pred\_dim = model.num\_classes – 1 #**13**

bbox\_regr\_dim = **4**

**A**= (1.0, 2.0, 0.2, 0.3, 0.5) \* 4 = 20

**NUM\_CONVS=4**

for nconv in range(cfg.RETINANET.NUM\_CONVS):

suffix = 'n{}\_fpn{}'.format(nconv, lvl)

retnet\_cls\_conv\_n(卷积序号)\_fpn(3-7): **下面以**nconv=0**为例说明**

**retnet\_cls\_conv\_n0\_fpn3**：w/8,h/8,256 (是fpn\_ res3\_3\_sum经过3\*3,s=1,p=1,256获得的)

以下w、b参数共享：

retnet\_cls\_conv\_n0\_fpn4：w/16,h/16,256 (是fpn\_ res4\_22\_sum经过3\*3,s=1,p=1,256获得的)

retnet\_cls\_conv\_n0\_fpn5：w/32,h/32,256 (是fpn\_ res5\_2\_sum经过3\*3,s=1,p=1,256获得的)

retnet\_cls\_conv\_n0\_fpn6：w/64,h/64,256 (是fpn\_ 6经过3\*3,s=1,p=1,256获得的)

retnet\_cls\_conv\_n0\_fpn7：w/128,h/128,256 (是fpn\_7经过3\*3,s=1,p=1,256获得的)

**retnet\_cls\_pred\_fpn3**：w/8,h/8,260(retnet\_cls\_conv\_n0\_fpn3经过3\*3,s=1,p=1,**13\*20**获得的)

以下w、b参数共享：

retnet\_cls\_pred\_fpn4：w/8,h/8,260(retnet\_cls\_conv\_n0\_fpn4经过3\*3,s=1,p=1,**13\*20**获得的)

retnet\_cls\_pred\_fpn5：w/32,h/32,260(retnet\_cls\_conv\_n0\_fpn5经过3\*3,s=1,p=1,**13\*20**获得的)

retnet\_cls\_pred\_fpn6：w/64,h/64,260(retnet\_cls\_conv\_n0\_fpn6经过3\*3,s=1,p=1,**13\*20**获得的)

retnet\_cls\_pred\_fpn7：w/128,h/128,260(retnet\_cls\_conv\_n0\_fpn7经过3\*3,s=1,p=1,**13\*20**)

retnet\_bbox\_conv\_n(卷积序号)\_fpn(3-7): **下面以**nconv=**0为例说明**

**retnet\_ bbox\_conv\_n0\_fpn3**：w/8,h/8,256 (是fpn\_ res3\_3\_sum经过3\*3,s=1,p=1,256获得的)

以下w、b参数共享：

retnet\_bbox \_conv\_n0\_fpn4：w/16,h/16,256 (是fpn\_ res4\_22\_sum经过3\*3,s=1,p=1,256获得)

retnet\_bbox \_conv\_n0\_fpn5：w/32,h/32,256 (是fpn\_ res5\_2\_sum经过3\*3,s=1,p=1,256获得)

retnet\_bbox \_conv\_n0\_fpn6：w/64,h/64,256 (是fpn\_ 6经过3\*3,s=1,p=1,256获得的)

retnet\_bbox \_conv\_n0\_fpn7：w/128,h/128,256 (是fpn\_7经过3\*3,s=1,p=1,256获得的)

bbox\_feat\_list = [**retnet\_ bbox\_conv\_n0\_fpn3、**retnet\_bbox \_conv\_n0\_fpn4、retnet\_bbox

\_conv\_n0\_fpn5、retnet\_bbox \_conv\_n0\_fpn6、retnet\_bbox \_conv\_n0\_fpn7]

**retnet\_bbox\_pred\_fpn3**：w/8,h/8,80(retnet\_bbox\_conv\_n0\_fpn3经过3\*3,s=1,p=1,**4\*20**获得)

以下w、b参数共享：

retnet\_bbox\_pred\_fpn4：w/8,h/8,80(retnet\_bbox\_conv\_n0\_fpn4经过3\*3,s=1,p=1,**4\*20**获得的)

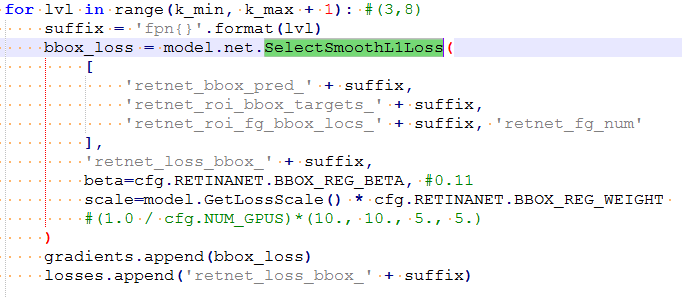
retnet\_bbox\_pred\_fpn5：w/32,h/32,80(retnet\_bbox\_conv\_n0\_fpn5经过3\*3,s=1,p=1,**4\*20**获得)

retnet\_bbox\_pred\_fpn6：w/64,h/64,80(retnet\_bbox\_conv\_n0\_fpn6经过3\*3,s=1,p=1,**4\*20**获得)

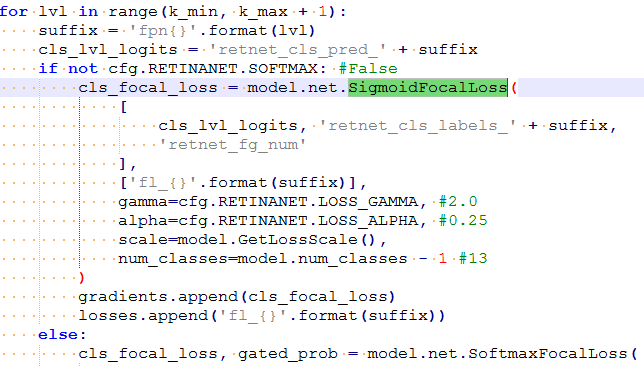
retnet\_bbox\_pred\_fpn7:w/128,h/128,80(retnet\_bbox\_conv\_n0\_fpn7经过3\*3,s=1,p=1,**4\*20**)

###### add\_fpn\_retinanet\_losses：训练时modeling/retinanet\_heads.py

loss\_gradients = retinanet\_heads.**add\_fpn\_retinanet\_losses**(model)



例如： retnet\_loss\_bbox\_fpn3



例如：f1\_fpn3

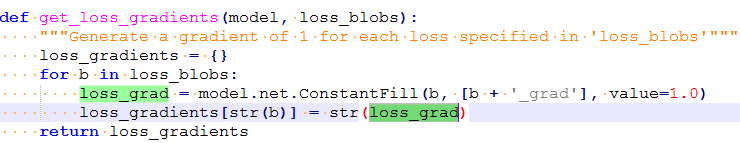
C:\Users\zhangjing1\AppData\Roaming\JunDaoIM\tempImages\image_lh.png

get\_loss\_gradients: util/blob.py

blob\_utils.**get\_loss\_gradients**(model, gradients)

gradients.append(bbox\_loss) #3-7层

gradients.append(cls\_focal\_loss) #3-7层



**gradients =**

[BlobReference("gpu\_0/retnet\_loss\_bbox\_fpn3"), BlobReference("gpu\_0/retnet\_loss\_bbox\_fpn4"), BlobReference("gpu\_0/retnet\_loss\_bbox\_fpn5"), BlobReference("gpu\_0/retnet\_loss\_bbox\_fpn6"), BlobReference("gpu\_0/retnet\_loss\_bbox\_fpn7"),

BlobReference("gpu\_0/fl\_fpn3"), BlobReference("gpu\_0/fl\_fpn4"),

BlobReference("gpu\_0/fl\_fpn5"), BlobReference("gpu\_0/fl\_fpn6"),

BlobReference("gpu\_0/fl\_fpn7")]

**losses =**

[u'retnet\_loss\_bbox\_fpn3',u'retnet\_loss\_bbox\_fpn4',u'retnet\_loss\_bbox\_fpn5', u'retnet\_loss\_bbox\_fpn6', u'retnet\_loss\_bbox\_fpn7',

u'fl\_fpn3',u'fl\_fpn4', u'fl\_fpn5', u'fl\_fpn6', u'fl\_fpn7']

**loss\_gradients** =

{'gpu\_0/fl\_fpn3': 'gpu\_0/**fl\_fpn3\_**grad**'**,

'gpu\_0/fl\_fpn6': 'gpu\_0/fl\_fpn6\_grad',

'gpu\_0/fl\_fpn7': 'gpu\_0/fl\_fpn7\_grad',

'gpu\_0/fl\_fpn4': 'gpu\_0/fl\_fpn4\_grad',

'gpu\_0/fl\_fpn5': 'gpu\_0/fl\_fpn5\_grad',

'gpu\_0/retnet\_loss\_bbox\_fpn3': 'gpu\_0/retnet\_loss\_bbox\_fpn3\_**grad**',

'gpu\_0/retnet\_loss\_bbox\_fpn4': 'gpu\_0/retnet\_loss\_bbox\_fpn4\_grad',

'gpu\_0/retnet\_loss\_bbox\_fpn5': 'gpu\_0/retnet\_loss\_bbox\_fpn5\_grad',

'gpu\_0/retnet\_loss\_bbox\_fpn6': 'gpu\_0/retnet\_loss\_bbox\_fpn6\_grad',

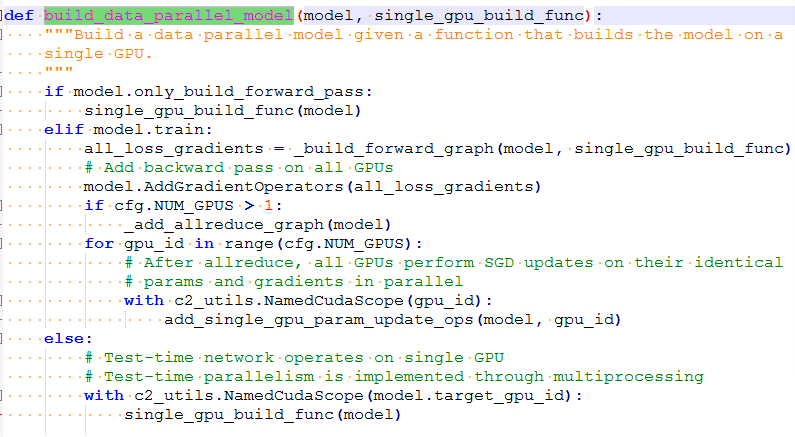
'gpu\_0/retnet\_loss\_bbox\_fpn7': 'gpu\_0/retnet\_loss\_bbox\_fpn7\_grad'}

**losses** = ['fl\_fpn6', 'fl\_fpn7', 'fl\_fpn5', 'fl\_fpn4', 'fl\_fpn3',

'retnet\_loss\_bbox\_fpn3','retnet\_loss\_bbox\_fpn4','retnet\_loss\_bbox\_fpn5', 'retnet\_loss\_bbox\_fpn6', 'retnet\_loss\_bbox\_fpn7']

##### build\_data\_parallel\_model

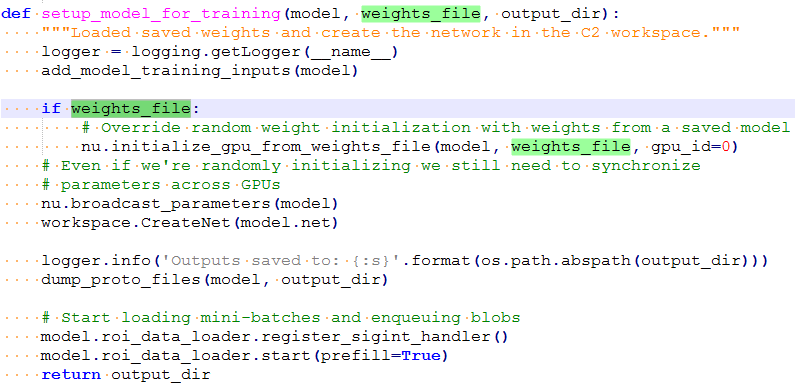
optim.**build\_data\_parallel\_model**(model, \_single\_gpu\_build\_func)



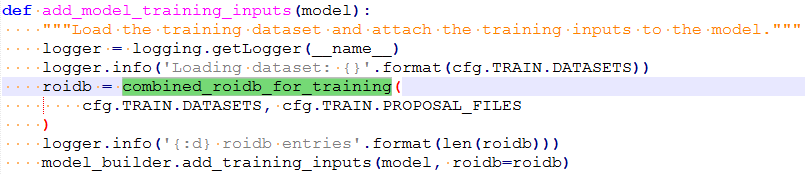
###### \_build\_forward\_graph:

all\_loss\_gradients = **\_build\_forward\_graph**(model, single\_gpu\_build\_func)

### setup\_model\_for\_training()    # 设置训练模型



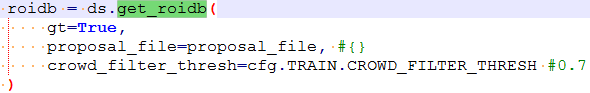
#### add\_model\_training\_inputs()



##### combined\_roidb\_for\_training()：datasets/roidb.py、json\_dataset.py

roidb = **combined\_roidb\_for\_training**(cfg.TRAIN.DATASETS, cfg.TRAIN.PROPOSAL\_FILES)

* **datasets/roidb.py:get\_roidb**(dataset\_name, proposal\_file)
* **datasets/json\_dataset.py: get\_roidb**()



###### Json数据->roidb格式：

self.anns = **anns** #dict={bbox索引：anns}

self.imgToAnns = **imgToAnns #** dict={图片索引：anns}

self.catToImgs = **catToImgs #** dict={类别索引：图片名}

self.imgs = **imgs #** dict={图片名：images (图片宽高、名字等信息)}

self.cats = **cats #** dict={类别索引：categories（类别索引、名称）}

如**{1: {u'supercategory': u'none', u'id': 1, u'name': u'suv'}}**

**roidb** = copy.deepcopy(self.COCO.**loadImgs**(image\_ids))

#[{u'file\_name':u'20180927\_1C1B0D228AF1\_00074.jpg',u'width':1920,u'id':u'20180927\_1C1B0D228AF1\_00074', u'height': 1080},

{u'file\_name':u'20180927\_1C1B0D228AF1\_00208.jpg',u'width':1920,u'id': u'20180927\_1C1B0D228AF1\_00208', u'height': 1080}]

for entry in roidb:

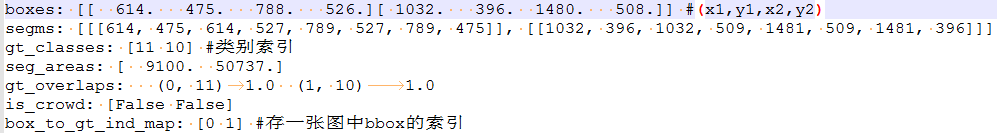
self.**\_prep\_roidb\_entry**(entry) #roidb新加字典键值初始化

for entry in roidb:

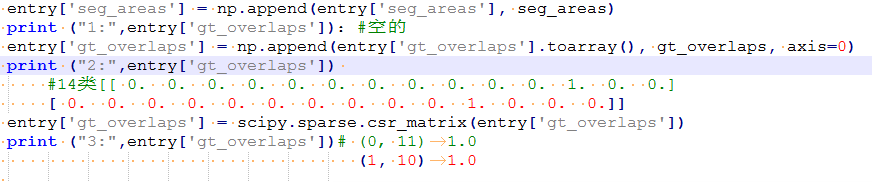
self.**\_add\_gt\_annotations**(entry) #将一张图的bbox信息重新组合，组合成下图所示：

**备注: (x1, y1, w, h) to (x1, y1, x2, y2) 坐标保护**

for obj in objs: #obj里面存放的是一个bbox的信息

****

entry['**gt\_overlaps**']:

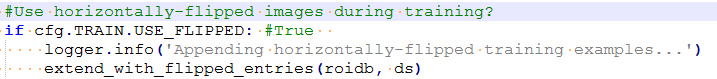


**\_add\_class\_assignments**(roidb):

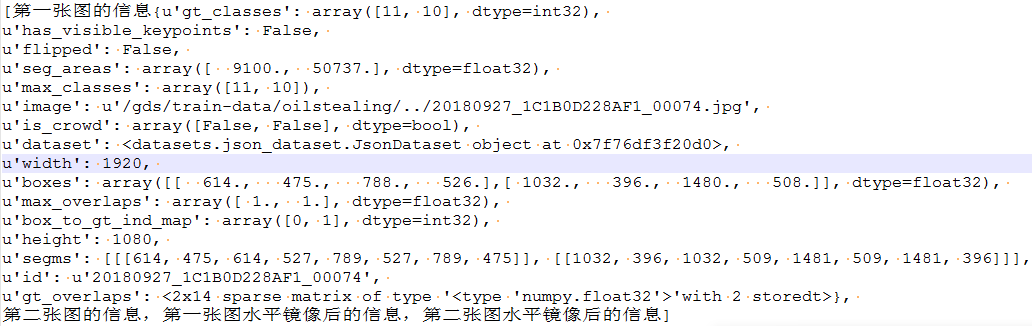
entry['max\_classes'] =[ [11 10]]

entry['max\_overlaps'] = [[ 1. 1.]]

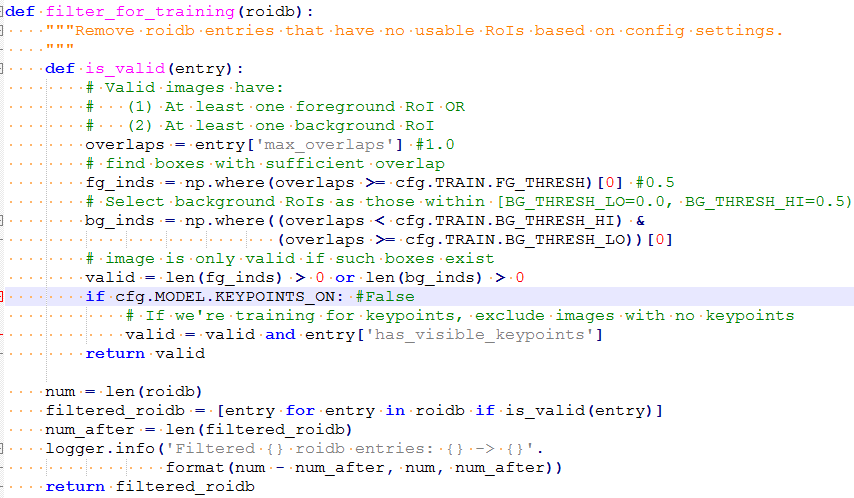
**训练数据实现水平镜像**： 代码如下



**roidb:**

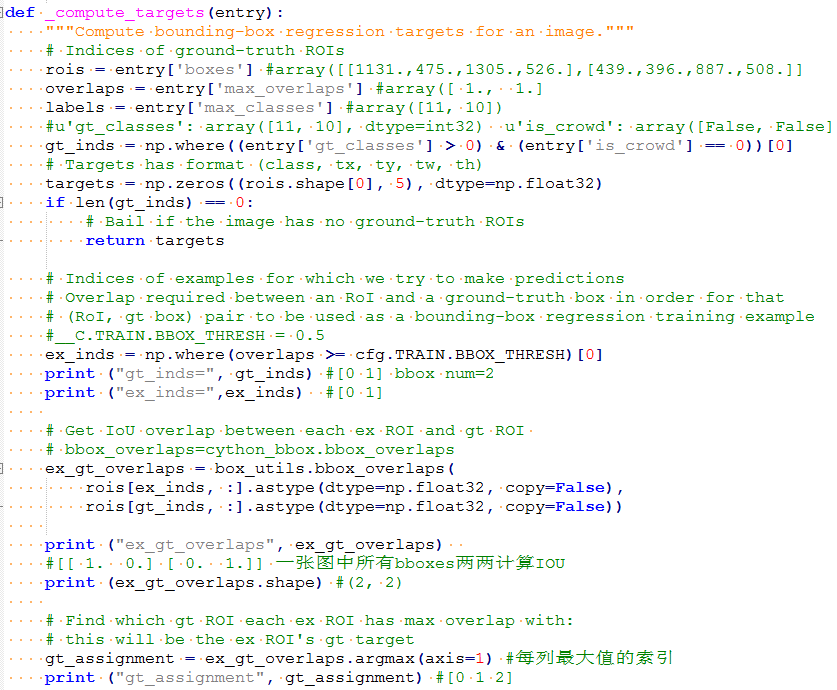
****

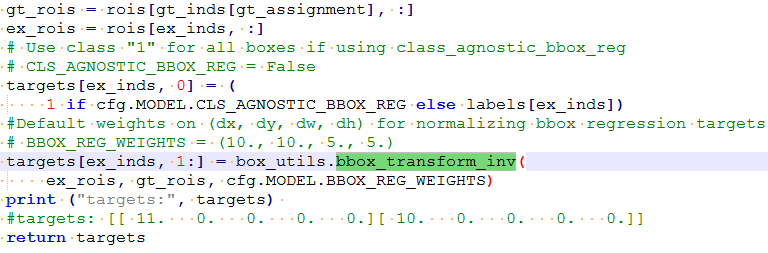
roidb = **filter\_for\_training**(roidb) #移除没有用的ROI，满足下面条件，就不用过滤：overlaps >= cfg.TRAIN.FG\_THRESH or cfg.TRAIN.BG\_THRESH\_LO<= overlaps < cfg.TRAIN.BG\_THRESH\_HI)



###### add\_bbox\_regression\_targets(roidb):生成entry['bbox\_targets'] =[[classid,tx,ty,tw,th],]

**\_compute\_targets**(entry)

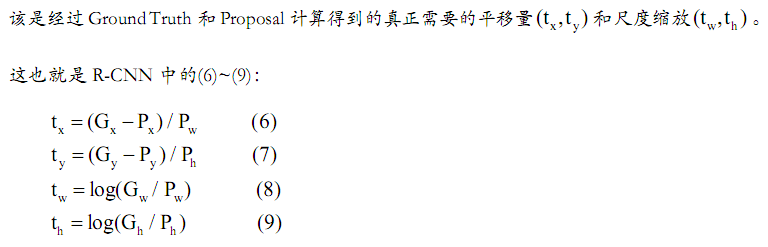




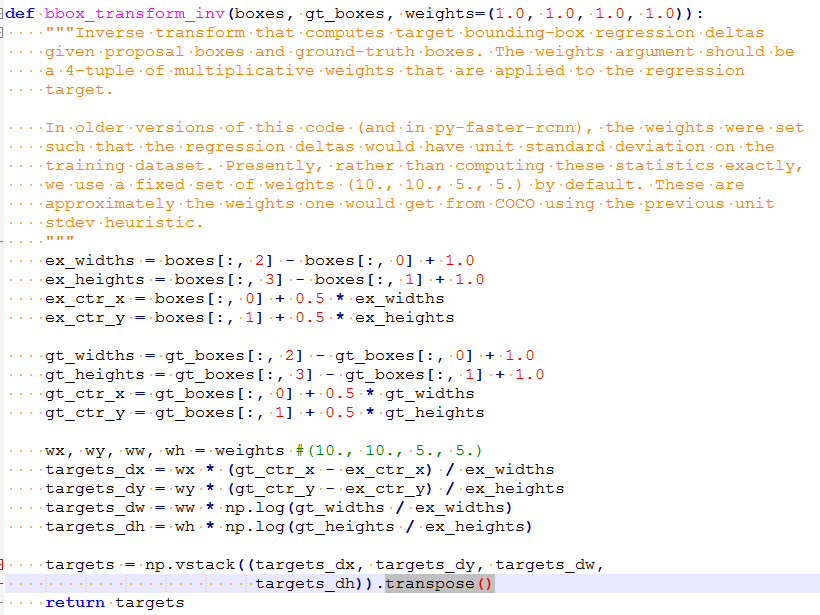
**entry['bbox\_targets'] = targets= [[classid,tx,ty,tw,th] [classid,tx,ty,tw,th]]**

#BBOX\_REG\_WEIGHTS = (10., 10., 5., 5.)

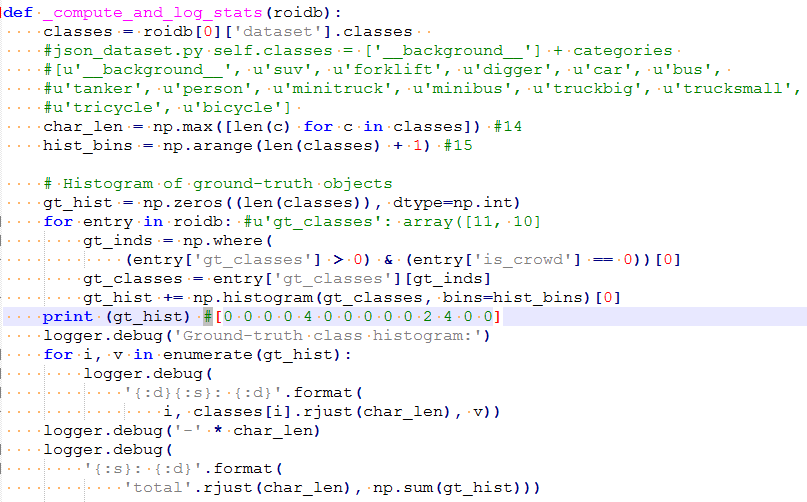
box\_utils**.bbox\_transform\_inv**(ex\_rois,gt\_rois,cfg.MODEL.BBOX\_REG\_WEIGHTS)



**备注**：是中心点相减



**\_compute\_and\_log\_stats(roidb) #统计每个类别的直方图**

****

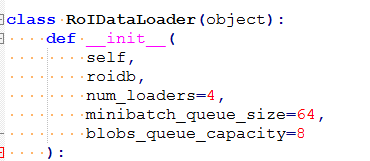
##### add\_training\_inputs:

model\_builder.**add\_training\_inputs**(model, roidb=roidb)

###### RoIDataLoader：lib/roi\_data/loader.py 多线程加载roidb

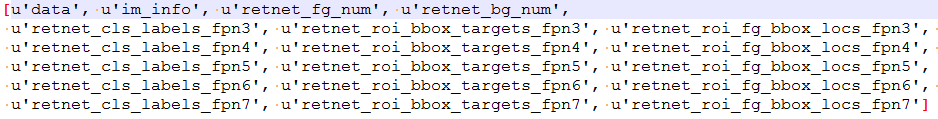
#\_\_C.DATA\_LOADER.NUM\_THREADS = 4

model.roi\_data\_loader = **RoIDataLoader**(roidb, num\_loaders=cfg.DATA\_LOADER.NUM\_THREADS)



get\_minibatch\_blob\_names()#roi\_data/minibatch.py

self.\_output\_names = get\_minibatch\_blob\_names()

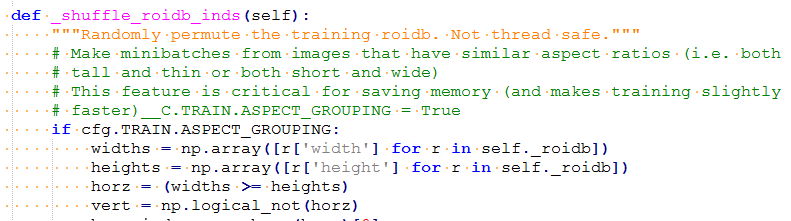


**retnet\_cls\_labels\_ fpn3**

**retnet\_roi\_bbox\_targets\_ fpn3**

**retnet\_roi\_fg\_bbox\_locs\_fpn3**

self.\_shuffle\_roidb\_inds()：将相似的anchor放到一个minbatches中，这样可以加速学习

****

self.create\_threads()

**self.\_workers = [**

**threading.Thread(target=self.minibatch\_loader\_thread)**

**for \_ in range(self.\_num\_loaders)**

**]**

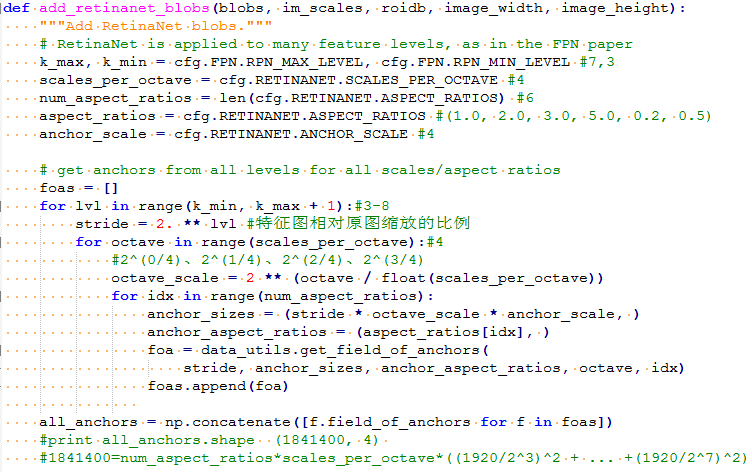
minibatch\_loader\_thread()

* **get\_next\_minibatch()**
* **get\_minibatch(roidb) #roi\_data/minibatch.py**
* **get\_minibatch\_blob\_names() + \_get\_image\_blob(roidb)+** **add\_retinanet\_blobs()**

valid = roi\_data.retinanet.**add\_retinanet\_blob**s(blobs, im\_scales, roidb, im\_width, im\_height)

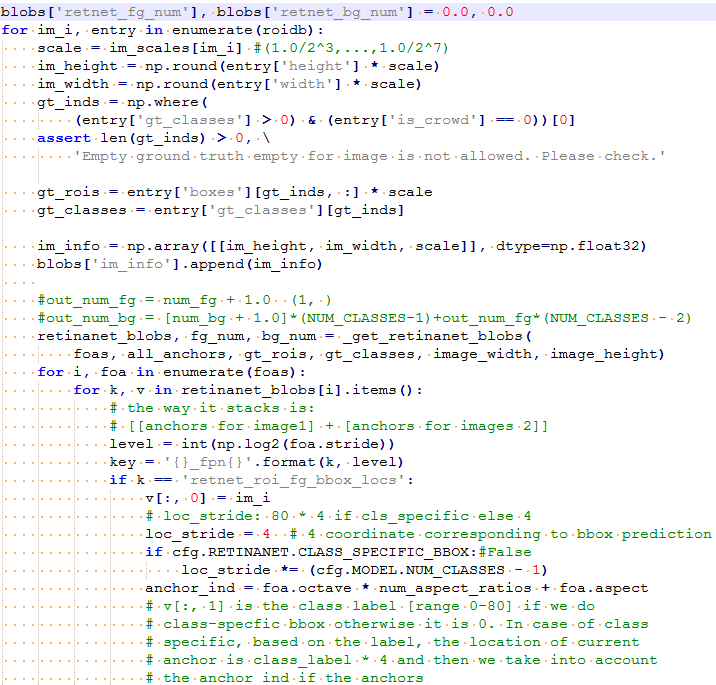
add\_retinanet\_blobs: #roi\_data/retinanet.py 获取计算loss的值

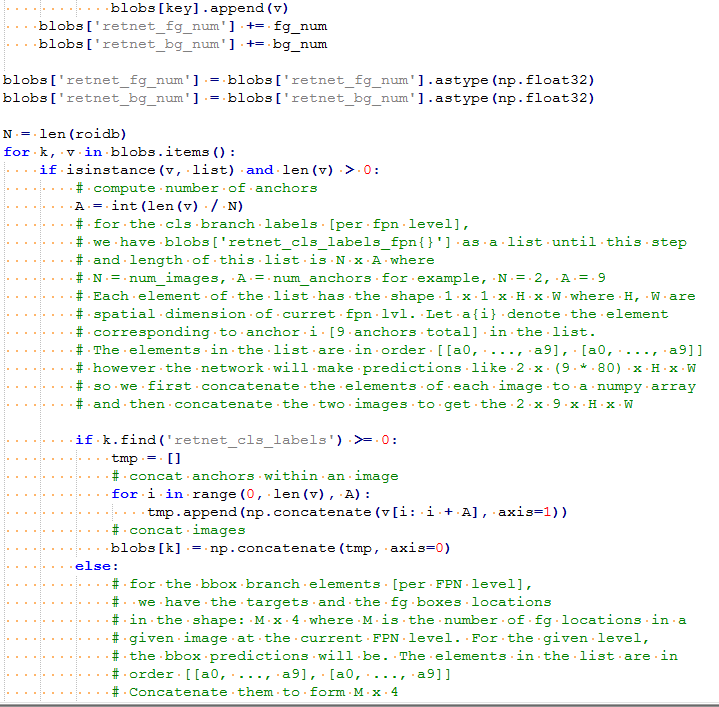
参考文献： <https://blog.csdn.net/Mr_health/article/details/84887015>



anchor\_sizes = (stride \* octave\_scale \* anchor\_scale, ) #anchor的边长

可以看出新增的那两个参数主要用于构成每一层FPN的anchor\_sizes ，也就是在原本的stride的基础上，乘以anchor\_scale，之后再乘以octave\_scale，由于octave\_scale有三个值，也就是相比于原来RPN+RPN网络一个FPN层只产生一种大小的anchor\_sizes，retinanet网络产生三种anchor\_sizes，直观上来说增加了anchor的数量，大大增加了anchor与gt的覆盖的可能。



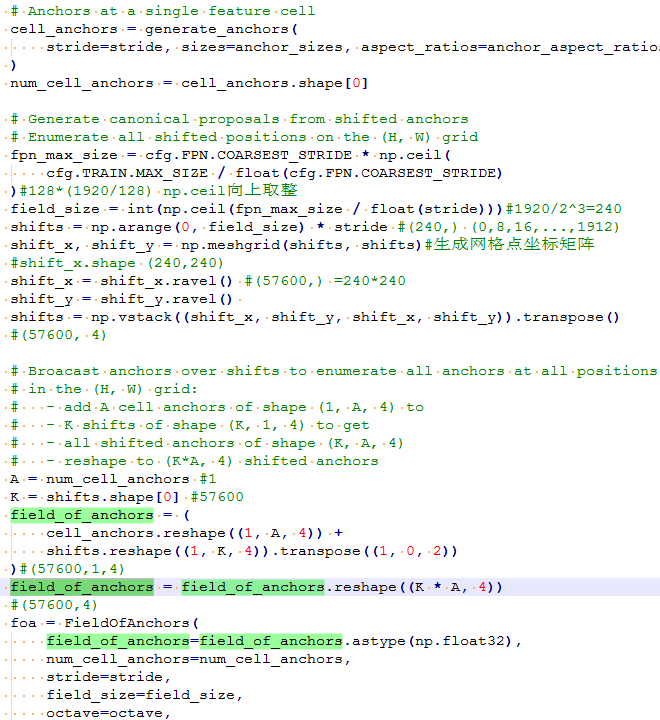


#roi\_data/data\_util.py

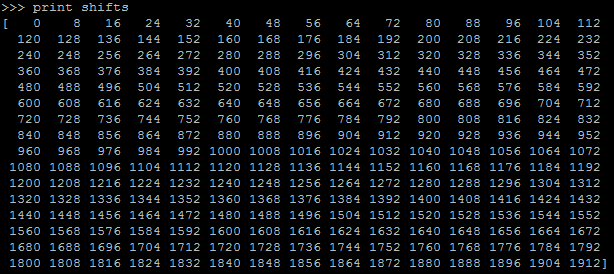
foa = data\_utils**.get\_field\_of\_anchors**(stride, anchor\_sizes, anchor\_aspect\_ratios, octave, idx)

* **generate\_anchors**() #modeling/generate\_anchors.py

**get\_field\_of\_anchors：**



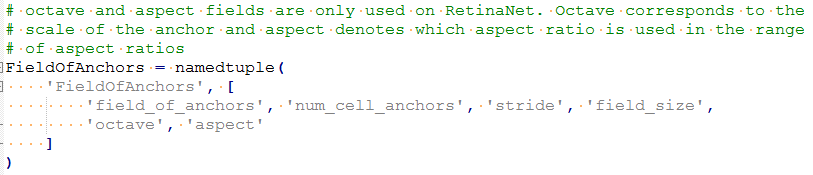
shifts = np.arange(0, 240) \* 8 #



foa = data\_utils.get\_field\_of\_anchors(stride, anchor\_sizes, anchor\_aspect\_ratios, octave, idx)

foas.append(foa)

foa存放的内容为下图：



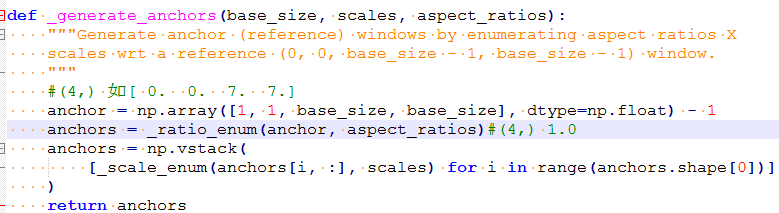
# Anchors at a single feature cell

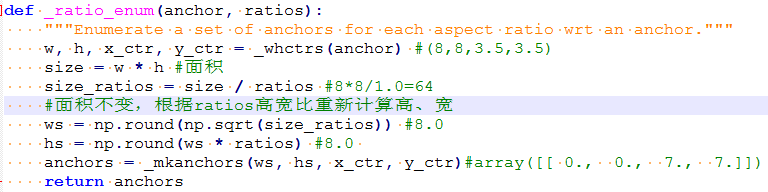
cell\_anchors=**generate\_anchors**(stride=stride,sizes=anchor\_sizes,aspect\_ratios=anchor\_aspect\_ratios)

例如: res3、resstride=2^3,sizes=(2^3) \*(2^0) \*4=32 ,aspect\_ratios=1.0)

**\_generate\_anchors**(8,32,1.0)

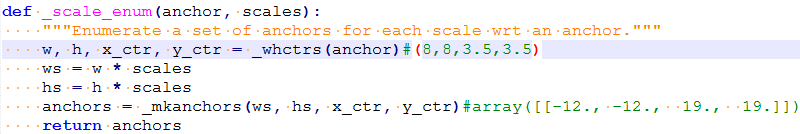
参考文献： <https://blog.csdn.net/sinat_33486980/article/details/81099093>





**备注**： hs/ws=ratios (高宽比)

\_ratio\_enum() 根据anchor的高宽比计算anchor的bbox



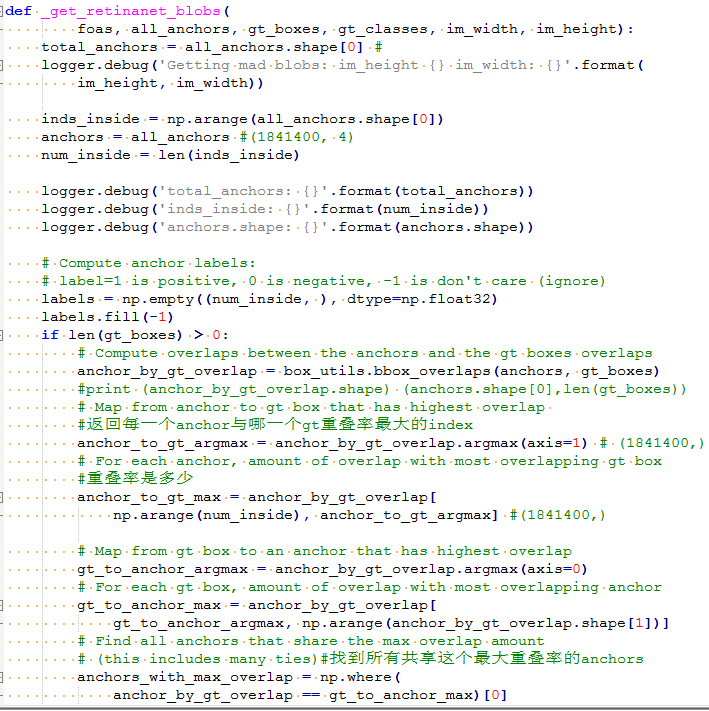
\_scale\_enum() 根据anchor的面积缩放计算anchor的bbox

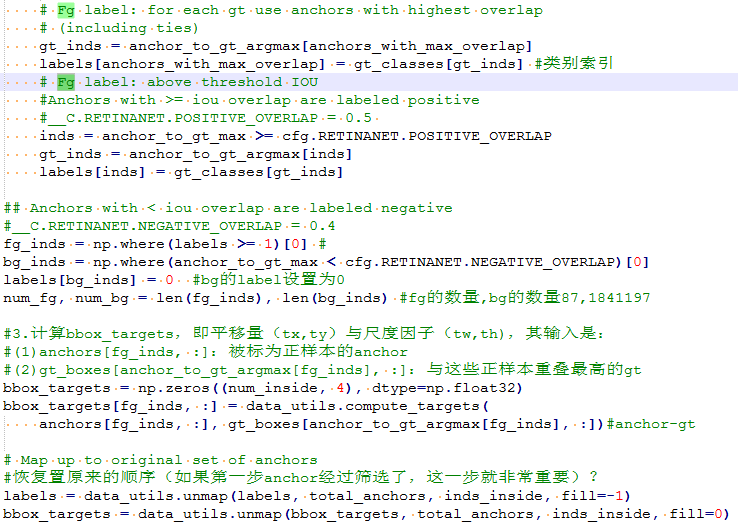
* **\_get\_retinanet\_blobs() :**

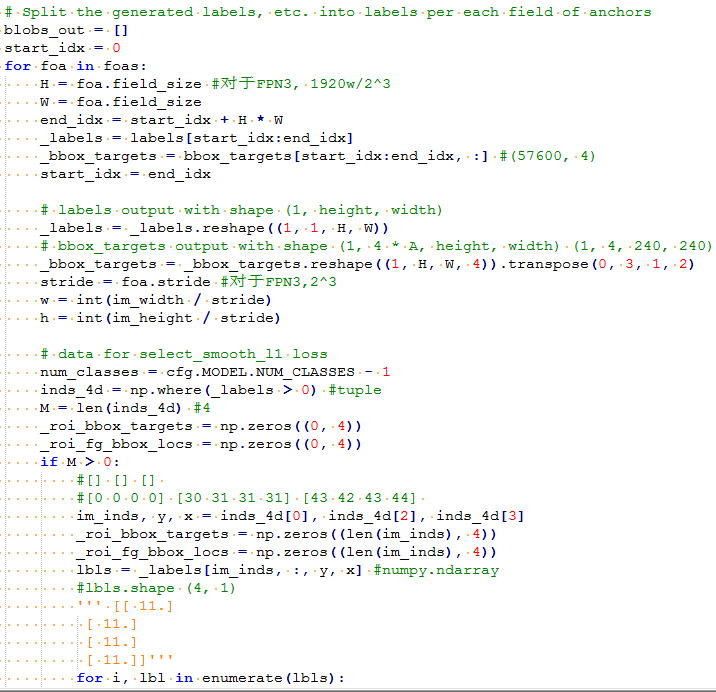
retinanet\_blobs, fg\_num, bg\_num = **\_get\_retinanet\_blobs**(

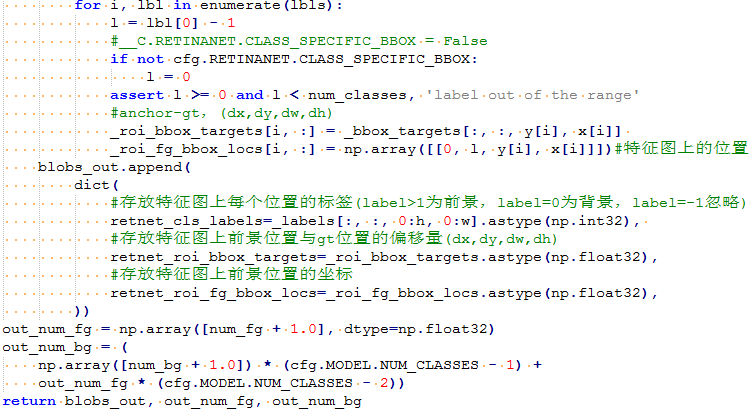
foas, all\_anchors, gt\_rois, gt\_classes, image\_width, image\_height)

参考文献：<https://blog.csdn.net/Mr_health/article/details/84887015>





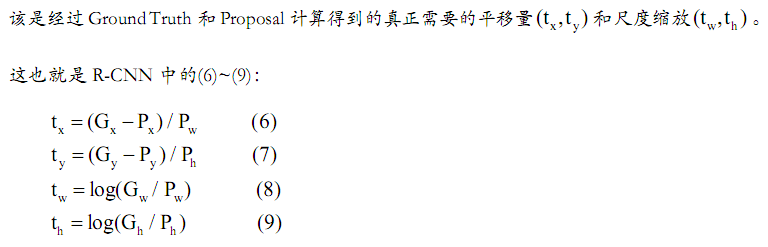




bbox\_targets[fg\_inds, :] = data\_utils.**compute\_targets**(

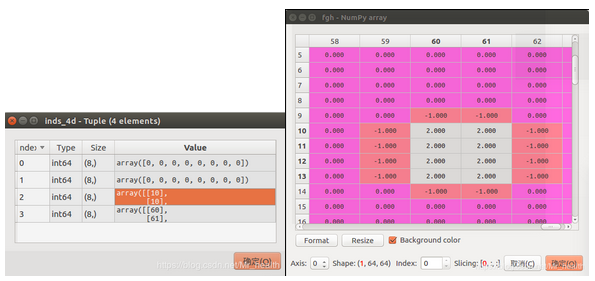
anchors[fg\_inds, :], gt\_boxes[anchor\_to\_gt\_argmax[fg\_inds], :])

* box\_utils.**bbox\_transform\_inv**(ex\_rois, gt\_rois, weights).astype(np.float32, copy=False

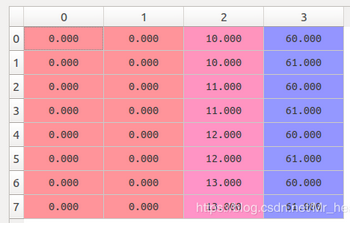


备注：是中心点相减

那如果inds\_4d是非空呢，如下左图，由于inds\_4d是四维所以不能直接可视化，我去掉第一维度可视化，即（1，1，W，H）->（1，W，H），此时就可以看到下面有些值为2代表fg。对比下面的label，以及inds\_4d中第三维度和第四维度的值就可以看出来，inds\_4d中第三维度表示的是label的行序号，第四维度表示的是label的列序号。

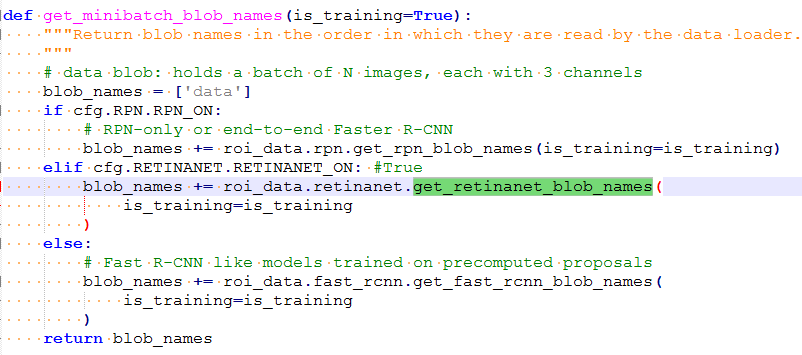


之后就是通过\_roi\_bbox\_targets[i, :] = \_bbox\_targets[:, :, y[i], x[i]]，\_roi\_fg\_bbox\_locs[i, :] = np.array([[0, l, y[i], x[i]]]) 取出fg对应的值。此时\_roi\_fg\_bbox\_locs并不象bg那样全为0了，如下，在第三列和第第四列存储了fg对应的行列号（按照个行顺序来，一行一行的存储）



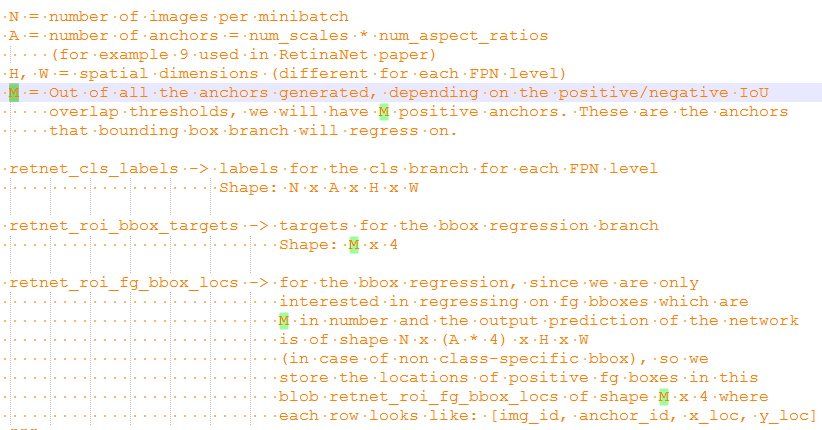
###### get\_minibatch\_blob\_names：roi\_data/minibatch.py

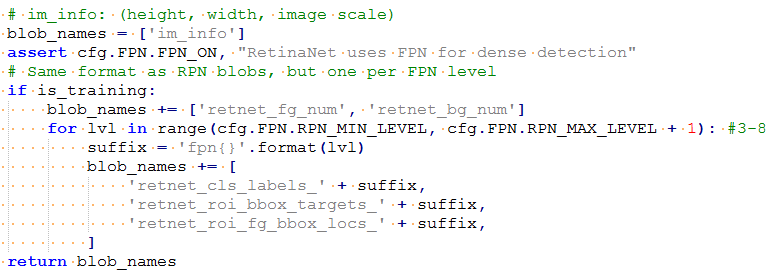
blob\_names = roi\_data.minibatch.**get\_minibatch\_blob\_names**(is\_training=True)



get\_retinanet\_blob\_names: roi\_data/retinanet.py

def **get\_retinanet\_blob\_names**(is\_training=True):





retnet\_cls\_labels\_fpn3： N x A x H x W（batchsize,anchors num, h,w）

retnet\_roi\_bbox\_targets\_fpn3: M x 4(M: 生成的anchors中IOU M positive anchors，

4：dx,dy,dw,dh)

[img\_id, anchor\_id, x\_loc, y\_loc]

retnet\_roi\_fg\_bbox\_locs\_fpn3:

#### initialize\_gpu\_from\_weights\_file(): utils/net.py 加载预训练模型

nu.**initialize\_gpu\_from\_weights\_file**(model, weights\_file, gpu\_id=0)

unscoped\_param\_names = OrderedDict() #创建一个有序字典

for blob in model.params: #只用字典的键值

unscoped\_param\_names[c2\_utils.UnscopeName(str(blob))] = True

with c2\_utils.NamedCudaScope(gpu\_id):

for unscoped\_param\_name in unscoped\_param\_names.keys():

….

print("11111111:", model.params)

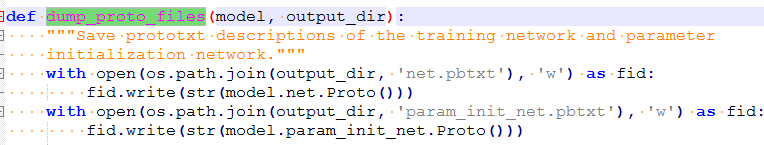
**权重共享：**

("gpu\_0/retnet\_cls\_conv\_n0\_fpn3\_w"),("gpu\_0/retnet\_cls\_conv\_n0\_fpn3\_b"), ("gpu\_0/retnet\_cls\_conv\_n1\_fpn3\_w"),("gpu\_0/retnet\_cls\_conv\_n1\_fpn3\_b"), ("gpu\_0/retnet\_cls\_conv\_n2\_fpn3\_w"),("gpu\_0/retnet\_cls\_conv\_n2\_fpn3\_b"), ("gpu\_0/retnet\_cls\_conv\_n3\_fpn3\_w"),("gpu\_0/retnet\_cls\_conv\_n3\_fpn3\_b"), **("gpu\_0/retnet\_cls\_pred\_fpn3\_w"),("gpu\_0/retnet\_cls\_pred\_fpn3\_b"),** ("gpu\_0/retnet\_bbox\_conv\_n0\_fpn3\_w"),("gpu\_0/retnet\_bbox\_conv\_n0\_fpn3\_b"), ("gpu\_0/retnet\_bbox\_conv\_n1\_fpn3\_w"),("gpu\_0/retnet\_bbox\_conv\_n1\_fpn3\_b"), ("gpu\_0/retnet\_bbox\_conv\_n2\_fpn3\_w"),("gpu\_0/retnet\_bbox\_conv\_n2\_fpn3\_b"), ("gpu\_0/retnet\_bbox\_conv\_n3\_fpn3\_w"),("gpu\_0/retnet\_bbox\_conv\_n3\_fpn3\_b"), **("gpu\_0/retnet\_bbox\_pred\_fpn3\_w"), ("gpu\_0/retnet\_bbox\_pred\_fpn3\_b")]**

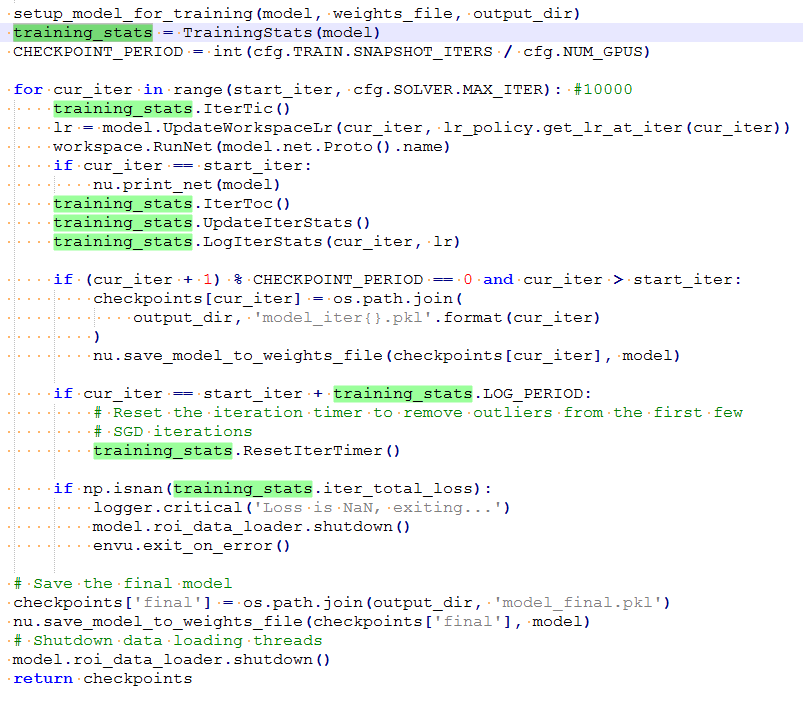
#### nu.broadcast\_parameters(model)

#### workspace.CreateNet(model.net)

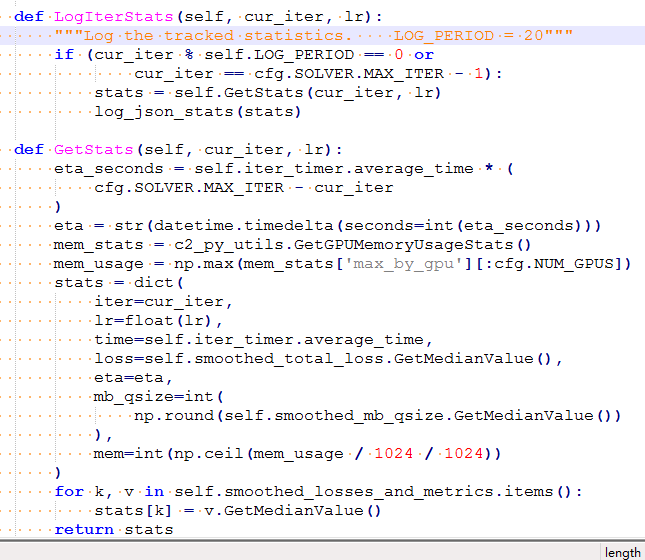
#### dump\_proto\_files：初始化网络



### TrainingStats(model) ：utils/ training\_stats.py追踪训练统计量



##### training\_stats.LogIterStats(cur\_iter, lr)：屏幕上打印的迭代次数



### UpdateWorkspaceLr：modeling/detector.py更新学习率

# cur\_iter 迭代次数 for cur\_iter in range(start\_iter, cfg.SOLVER.MAX\_ITER):

lr = model.**UpdateWorkspaceLr(**cur\_iter, lr\_policy.**get\_lr\_at\_iter**(cur\_iter))

cur\_lr = workspace.**FetchBlob**('gpu\_0/lr')[0] #取lr的值

workspace**.FeedBlob**('gpu\_{}/lr'.format(i), np.array([new\_lr], dtype=np.float32)) #重新给lr赋值

### nu.print\_net(model): util/train.py ->util/net.py

**备注**： 'MAX\_SIZE': 1920,'SCALES': (1080,), 'COARSEST\_STRIDE': 128

输入1080\*1920，补黑边(128的整数倍)，变成1152\*1920

data (1, 3, 1152, 1920)

* conv1 (1, 64, 576, 960) (7\*7,64,s=2,p=3) 1/2
* pool1(1, 64, 288, 480) 1/4
* res2\_0\_branch2a (1, 64, 288, 480)
* res2\_2\_sum(1, 256, 288, 480)
* res3\_0\_branch2a(1, 128, 144, 240) 1/8
* res3\_3\_sum(1, 512, 144, 240)
* res4\_0\_branch2a(1, 256, 72, 120) 1/16
* res4\_22\_sum(1, 1024, 72, 120)
* res5\_0\_branch2a(1, 512, 36, 60) 1/32
* res5\_2\_sum(1, 2048, 36, 60)

res5\_2\_sum : (1, 2048, 36, 60)

=> **fpn\_inner\_res5\_2\_sum** (1, 256, 36, 60)

=> fpn\_inner\_res4\_22\_sum\_topdown (1, 256, 72, 120)

res4\_22\_sum (1, 1024, 72, 120) => fpn\_inner\_res4\_22\_sum\_lateral(1, 256, 72, 120)

fpn\_inner\_res4\_22\_sum\_lateral(1, 256, 72, 120)+ fpn\_inner\_res4\_22\_sum\_topdown (1, 256, 72, 120)=> **fpn\_inner\_res4\_22\_sum (1, 256, 72, 120)**

res3\_3\_sum (1, 512, 144, 240) => fpn\_inner\_res3\_3\_sum\_lateral (1, 256, 144, 240) fpn\_inner\_res4\_22\_sum (1, 256, 72, 120) => fpn\_inner\_res3\_3\_sum\_topdown: (1, 256, 144, 240)

fpn\_inner\_res3\_3\_sum\_lateral (1, 256, 144, 240)+ fpn\_inner\_res3\_3\_sum\_topdown(1, 256, 144, 240)=>  **fpn\_inner\_res3\_3\_sum** (1, 256, 144, 240)

fpn\_inner\_res5\_2\_sum (1, 256, 36, 60) => **fpn\_res5\_2\_sum** (1, 256, 36, 60) --(op: Conv)

fpn\_inner\_res4\_22\_sum (1, 256, 72, 120) => **fpn\_res4\_22\_sum** (1, 256, 72, 120) --(op: Conv)

fpn\_inner\_res3\_3\_sum (1, 256, 144, 240) => **fpn\_res3\_3\_sum (1, 256, 144, 240)** --(op: Conv)

res5\_2\_sum (1, 2048, 36, 60) => fpn\_6 (1, 256, 18, 30)+relu

fpn\_6\_relu (1, 256, 18, 30) => fpn\_7 (1, 256, 9, 15)

**fpn\_res3\_3\_sum (1, 256, 144, 240)** => retnet\_cls\_conv\_n0\_fpn3 (1, 256, 144, 240)

retnet\_cls\_conv\_n0\_fpn3 (1, 256, 144, 240) => retnet\_cls\_conv\_n0\_fpn3 (1, 256, 144, 240)

retnet\_cls\_conv\_n0\_fpn3 (1, 256, 144, 240) => retnet\_cls\_conv\_n1\_fpn3 (1, 256, 144, 240)

retnet\_cls\_conv\_n1\_fpn3 (1, 256, 144, 240) => retnet\_cls\_conv\_n1\_fpn3 (1, 256, 144, 240)

retnet\_cls\_conv\_n1\_fpn3 (1, 256, 144, 240) => retnet\_cls\_conv\_n2\_fpn3 (1, 256, 144, 240)

retnet\_cls\_conv\_n2\_fpn3 (1, 256, 144, 240) => retnet\_cls\_conv\_n2\_fpn3 (1, 256, 144, 240)

retnet\_cls\_conv\_n2\_fpn3 (1, 256, 144, 240) => retnet\_cls\_conv\_n3\_fpn3 (1, 256, 144, 240)

retnet\_cls\_conv\_n3\_fpn3 (1, 256, 144, 240) => retnet\_cls\_conv\_n3\_fpn3 (1, 256, 144, 240)

retnet\_cls\_conv\_n3\_fpn3 (1, 256, 144, 240) => **retnet\_cls\_pred\_fpn3** (1, **312**, 144, 240)

**(1.0, 2.0, 3.0, 5.0, 0.2, 0.5)\* 4 （SCALES\_PER\_OCTAVE）\*13类（不含背景）=312**

**fpn\_res4\_22\_sum** (1, 256, 72, 120)… => **retnet\_cls\_pred\_fpn4** (1, 312, 72, 120)

**fpn\_res5\_2\_sum** (1, 256, 36, 60) …=> **retnet\_cls\_pred\_fpn5** (1, 312, 36, 60)

**fpn\_6 (1, 256, 18, 30)**… => **retnet\_cls\_pred\_fpn6** (1, 312, 18, 30)

**fpn\_7 (1, 256, 9, 15)** … => **retnet\_cls\_pred\_fpn7** (1, 312, 9, 15)

**fpn\_res3\_3\_sum (1, 256, 144, 240)**  => retnet\_bbox\_conv\_n0\_fpn3 (1, 256, 144, 240)

retnet\_bbox\_conv\_n0\_fpn3 (1, 256, 144, 240) => retnet\_bbox\_conv\_n0\_fpn3 (1, 256, 144, 240)

retnet\_bbox\_conv\_n0\_fpn3 (1, 256, 144, 240) => retnet\_bbox\_conv\_n1\_fpn3 (1, 256, 144, 240)

retnet\_bbox\_conv\_n1\_fpn3 (1, 256, 144, 240) => retnet\_bbox\_conv\_n1\_fpn3 (1, 256, 144, 240)

retnet\_bbox\_conv\_n1\_fpn3 (1, 256, 144, 240) => retnet\_bbox\_conv\_n2\_fpn3 (1, 256, 144, 240)

retnet\_bbox\_conv\_n2\_fpn3 (1, 256, 144, 240) => retnet\_bbox\_conv\_n2\_fpn3 (1, 256, 144, 240)

retnet\_bbox\_conv\_n2\_fpn3 (1, 256, 144, 240) => retnet\_bbox\_conv\_n3\_fpn3 (1, 256, 144, 240)

retnet\_bbox\_conv\_n3\_fpn3 (1, 256, 144, 240) => **retnet\_bbox\_conv\_n3\_fpn3** (1, 256, 144, 240)

retnet\_bbox\_conv\_n3\_fpn3 (1, 256, 144, 240) => **retnet\_bbox\_pred\_fpn3** (1, 96, 144, 240) retnet\_bbox\_conv\_n3\_fpn4 (1, 256, 72, 120) => **retnet\_bbox\_pred\_fpn4** (1, 96, 72, 120)

retnet\_bbox\_conv\_n3\_fpn5 (1, 256, 36, 60) => **retnet\_bbox\_pred\_fpn5** (1, 96, 36, 60)

retnet\_bbox\_conv\_n3\_fpn6 (1, 256, 18, 30) => **retnet\_bbox\_pred\_fpn6** (1, 96, 18, 30)

retnet\_bbox\_conv\_n3\_fpn7 (1, 256, 9, 15) => retnet\_bbox\_pred\_fpn7 (1, 96, 9, 15)

**(1.0, 2.0, 3.0, 5.0, 0.2, 0.5)\* 4 （SCALES\_PER\_OCTAVE）\*4（x,yw,h）=96**

Loss:

r**etnet\_bbox\_pred\_fpn3** (1, 96, 144, 240) + **retnet\_roi\_bbox\_targets\_fpn3**: (0, 4) +

**retnet\_roi\_fg\_bbox\_locs\_fpn3**(0,4) + **retnet\_fg\_num**(1,)

=>**retnet\_loss\_bbox\_fpn3()**--(op: **SelectSmoothL1Loss**)

(retnet\_loss\_bbox\_fpn4、retnet\_loss\_bbox\_fpn5、retnet\_loss\_bbox\_fpn6、retnet\_loss\_bbox\_fpn7)

r**etnet\_cls\_pred\_fpn3** (1, 312, 144, 240) + **retnet\_cls\_labels\_fpn3** (1, 24, 144, 240) +

**retnet\_fg\_num** (1,) => **fl\_fpn3:** () --(op: **SigmoidFocalLoss**

fl\_fpn4 、fl\_fpn5 、fl\_fpn6 、fl\_fpn7)

**备注：**

r**etnet\_bbox\_pred\_fpn3** (1, 96, 144, 240)：预测的bbox结果

**retnet\_roi\_bbox\_targets\_fpn3**: (0, 4)：

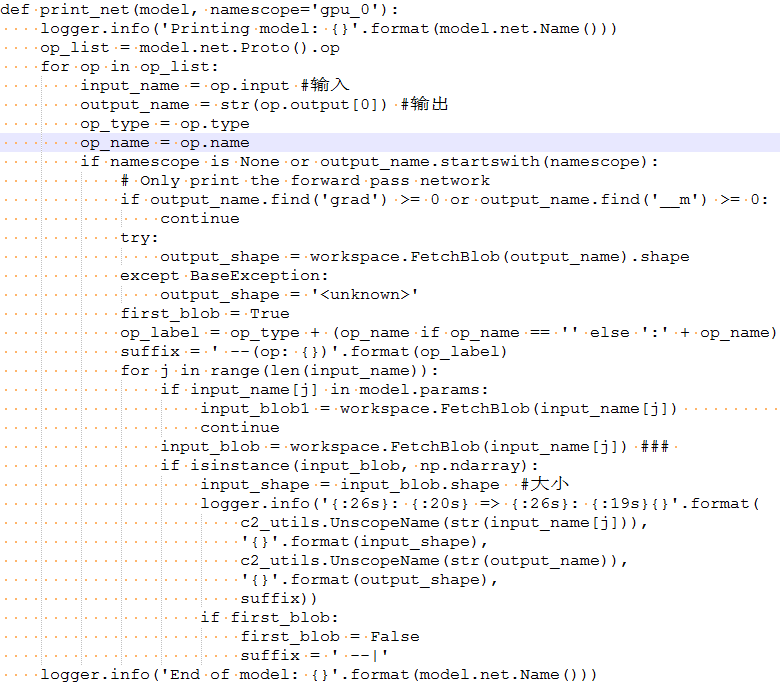
**retnet\_roi\_fg\_bbox\_locs\_fpn3**(0,4)：

**retnet\_fg\_num**(1,)：

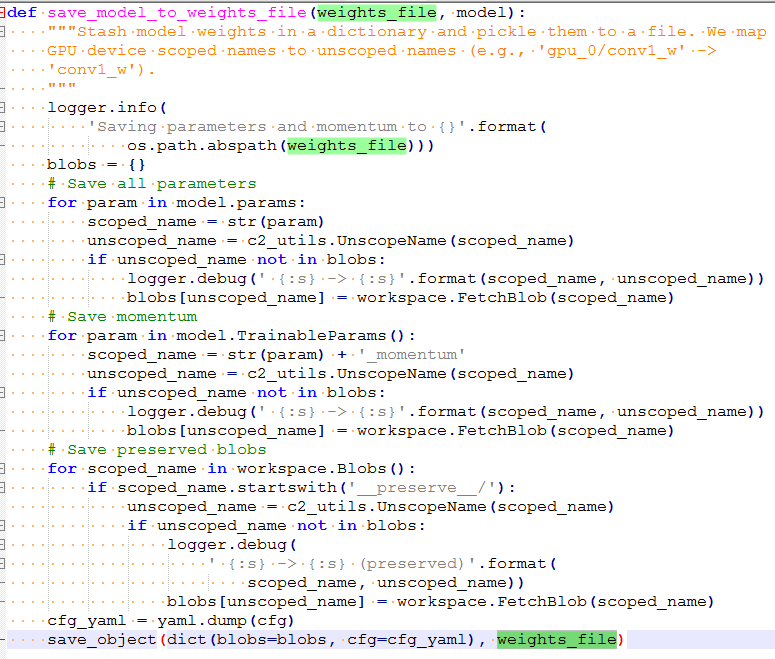
r**etnet\_cls\_pred\_fpn3** (1, 312, 144, 240)：预测的cls结果

**retnet\_cls\_labels\_fpn3** (1, 24, 144, 240)：

#### 代码部分：



### save\_model\_to\_weights\_file: utils/net.py 保存模型



### model.roi\_data\_loader.shutdown() # 关闭数据加载线程

## 备注

# 加载训练数据集lib/datasets/roidb.py

[1] **roidb** = combined\_roidb\_for\_training() #roidb数据结构

1) roidbs = [get\_roidb(\*args) for args in zip(dataset\_names,

proposal\_files)]

① ds = JsonDataset(dataset\_name) # 加载JsonDataset

② roidb = ds.get\_roidb() # 读取 roidb

③ if cfg.TRAIN.USE\_FLIPPED:extend\_with\_flipped\_entries(roidb, ds)

# 水平翻转训练

2) roidb = **filter\_for\_training**(roidb) # 过滤掉没有可用 ROIs 的 roidb 元素

3) **add\_bbox\_regression\_targets**(roidb) #计算 bbox 回归 target（dx,dy,dw,dh）

4) **\_compute\_and\_log\_stats**(roidb) # 统计数据分布

[2] model\_builder.**add\_training\_inputs**(model, roidb=roidb)

#创建训练网络输入 input ops

1) model.roi\_data\_loader = **RoIDataLoader**() # 多线程数据加载,config.py

2) blob\_names=roi\_data\_minibatch.**get\_minibatch\_blob\_names**(is\_training=True)

#获取 blob\_names

cfg.RPN.RPN\_ON=true、cfg.RETINANET.RETINANET\_ON=true

blob\_names????????? model\_builder.py

看一下blob\_name是咋命名的

3) workspace.**CreateBlob**(core.ScopedName(blob\_name)) # workspace 创建 blob

4) model.net.**DequeueBlobs**()

5) 将 input op 移到网络 net 开始处

#### nu.initialize\_gpu\_from\_weights\_file()

# 利用预训练模型权重初始化 GPU-0 网络权重，覆盖随机化的权重参数

#### nu.broadcast\_parameters()

# 将 GPU-0 权重参数复制到其他 GPU 权重初始化参数

#### workspace.CreateNet(model.net) # workspace 创建网络 net

#### dump\_proto\_files() # 保存初始化的模型结构和模型参数

#### model.roi\_data\_loader.register\_sigint\_handler()

—> model.roi\_data\_loader.start(prefill=True) #开始加载 mini-batches 和队列 blobs 线程

### 训练过程

1、lr = model.UpdateWorkspaceLr(cur\_iter, lr\_policy.get\_lr\_at\_iter(cur\_iter)) # **更新 lr 参数**

2、workspace.**RunNet**(model.net.Proto().name) # 进行一次训练

3、if cur\_iter == start\_iter: nu.print\_net(model) # 在第一次迭代时打印网络结构

4、if (cur\_iter + 1) % CHECKPOINT\_PERIOD == 0 and cur\_iter > start\_iter: # 每

CHECKPOINT\_PERIOD（20000） 次

nu.**save\_model\_to\_weights\_file**(checkpoints[cur\_iter], model) # 保存检查点训练模型

5、if cur\_iter == start\_iter + training\_stats.LOG\_PERIOD: training\_stats.ResetIterTimer()

# 重置迭代计时器

6、if np.isnan(training\_stats.iter\_total\_loss): # 出错

logger.critical('Loss is NaN, exiting...') —> model.roi\_data\_loader.shutdown() —>

envu.exit\_on\_error()

7、回到 1，进行下一次迭代训练

8、nu.**save\_model\_to\_weights\_file**(checkpoints['final'], model) # 训练结束保存最后模型

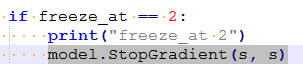
9、model.roi\_data\_loader.shutdown() # 关闭数据加载线程

<https://blog.csdn.net/Mr_health/article/details/84857321>

# 知识点备注

## resnet101函数

### model.StopGradient(s, s):停止梯度传播



### AffineChannel：detector.py

p = model.AffineChannel(p, 'res\_conv1\_bn', dim=64, inplace=True)

### ConvAffine

conv -> BN

### bottleneck\_transformation

bottleneck\_transformation() 指的是下面这个模块：

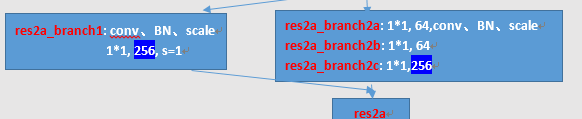


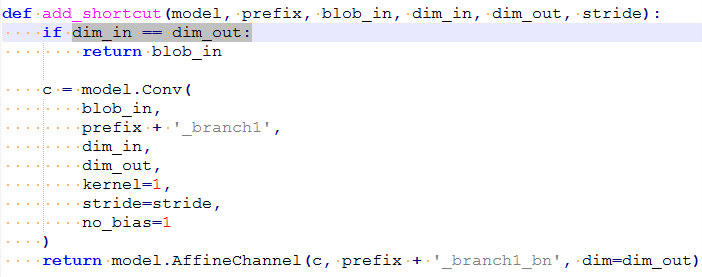
### add\_shortcut



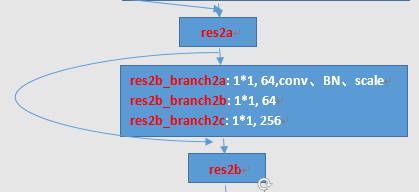
### add\_residual\_block

add\_residual\_block()的输出就是如下图的 res2a



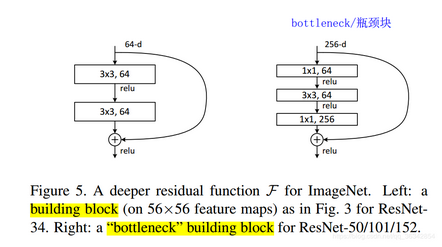


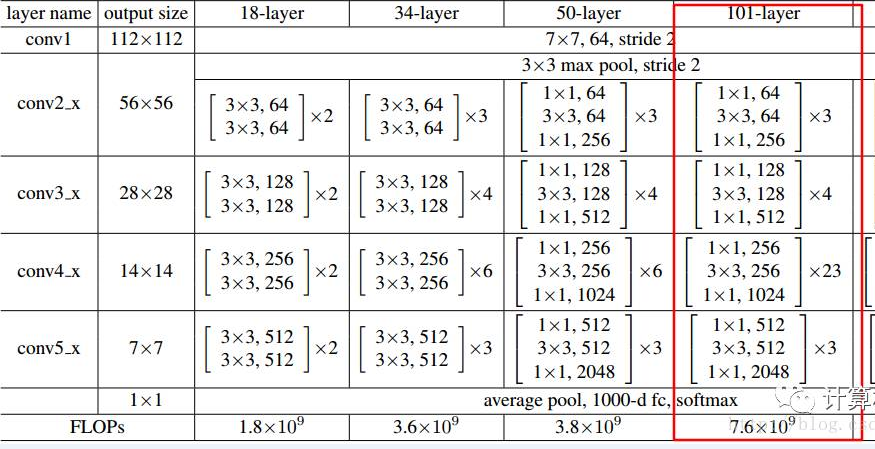
当dim\_in==dim\_out时，执行下面模块

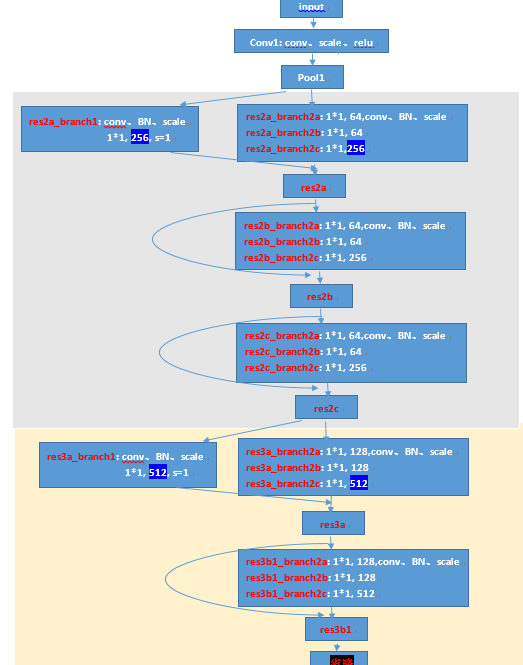


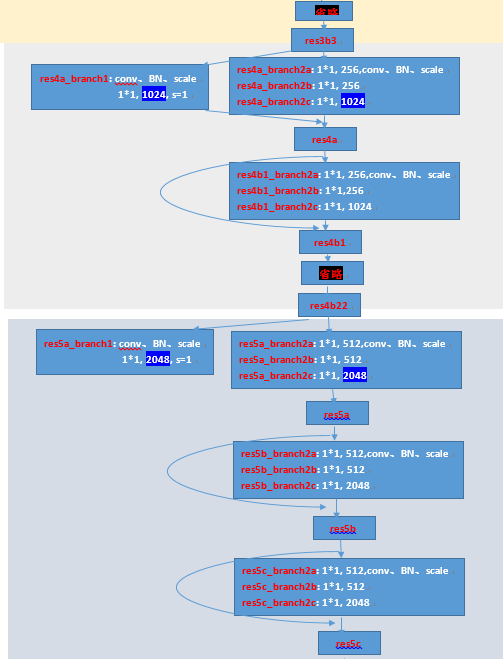
## resNet:

首先是ResNet的两种block：building block和“bottleneck” building block：





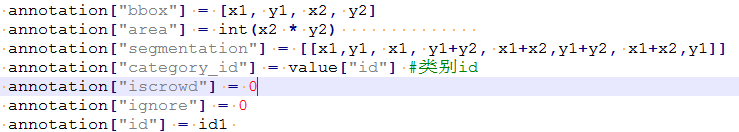




C:\Users\ZHANGJ~2\AppData\Local\Temp\1562660263(1).png

## Json格式





**备注**： **annotation["bbox"] = [x1, y1, w, h]**

## roidb数据结构：

**参考文献**：<https://www.jianshu.com/p/29ab7457b61e>

**combined\_roidb\_for\_training**(cfg.TRAIN.DATASETS, cfg.TRAIN.PROPOSAL\_FILES)

roidb的类型是list, 其中的每个元素的数据类型都是dict, roidb列表的长度为数据集的数量(即图片的数量),

for entry in roidb:

entry['id'] : int 代表了当前image的img\_id

entry['file\_name']: str 表示当前图片的文件名(带有.jpg后缀)

entry['boxes']: float32,numpy数组(num\_objs, 4),num\_objs为当前图片中的

目标物体个数, 4代表bbox的坐标

entry**['gt\_classes']:** int32, numpy数组(num\_objs),指明当前图片中每一个obj的

真实类别

entry['gt\_overlaps']: float32, scipy.sparse.csr\_matrix数据(num\_objs, 81),代表每一个

obj与81个不同类别的overlap

entry['box\_to\_gt\_ind\_map']: int32, numpy数组(num\_objs),该列表存储着box的顺序下

标值, 同样是一维数组, 直接拼接,将每一个roi映射到一个index上, index是在entry['gt\_classes']>0的rois列表的下标

Detectron 的数据载入类 RoIDdataLoader 也是将roidb数据结构作为成员变量使用的.

### get\_roidb():lib/datasets/json\_dataset.py、pycocotools

**将json数据转成roidb数据结构**

train()->add\_model\_training\_inputs()->roidb=combined\_roidb\_for\_training():detectron/datasets/roidb -> **get\_roidb()**

category\_to\_id\_map = dict(zip(className, className\_ids)) #建立类别的name 与 id之间的对应关系,

需要对数据进行过滤、保护处理。

entry['bbox\_targets']=(n，5) #(class, **tx, ty, tw, th**)

**注意：**

annotation["bbox"] = [x1, y1, x2, y2]

annotation["area"] = int(x2 \* y2)

annotation**["segmentation"]** = [[x1,y1, x1, y1+y2, x1+x2,y1+y2, x1+x2,y1]]

annotation["category\_id"] = value["id"] #类别id

annotation["ignore"] = 0

annotation["id"] = id1 #图片索引

annotation["iscrowd"] = 0

annotation["image\_id"] = img\_id\_temp #图片名字

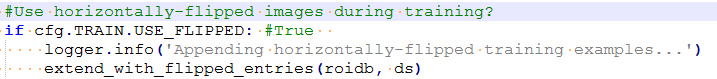
x1, y1, x2, y2 = box\_utils.**clip\_xyxy\_to\_image**(x1, y1, x2, y2, height, width) #保护bbox不越界

## 保存模型：迭代次数跟gpu个数有关

CHECKPOINT\_PERIOD = int(cfg.TRAIN.SNAPSHOT\_ITERS / cfg.NUM\_GPUS)

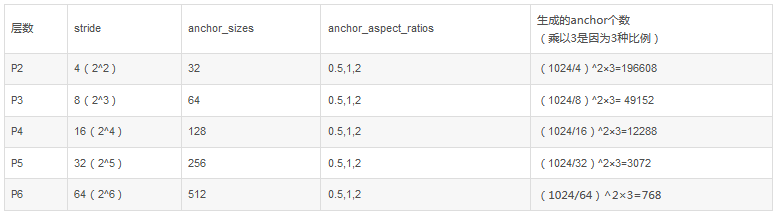
## 训练数据水平镜像： lib/datasets/roidb.py

\_\_C.TRAIN.USE\_FLIPPED = True



## **RPN+FPN网络是如何产生anchor**

这里以P2-P6的FPN网络为例（训练图片的大小为1024×1024，其余参数皆为默认）：



stride：表示以stride为步长，每隔一个stride生成一个anchor

anchor\_sizes：在同一个FPN层生成的anchor面积为anchor\_sizes×anchor\_sizes，其中对于

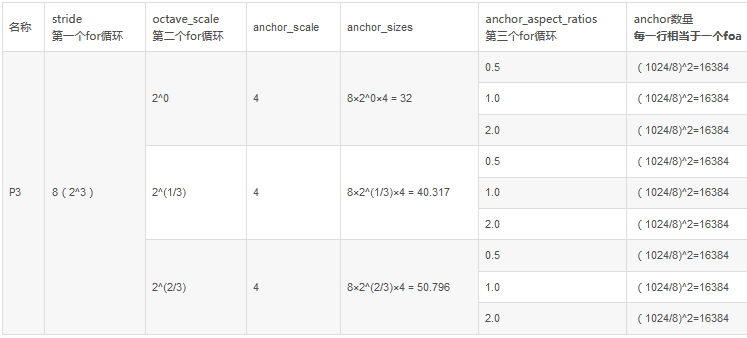
P2层，其anchor\_sizes的大小是由参数C.FPN.RPN\_ANCHOR\_START\_SIZE决定的，后一层的anchor\_sizes是前一层的2倍。

anchor\_aspect\_ratios：在同以FPN层，在满足面积相同的情况下，生成三种长宽比例的anchor。

## retinaNet生成anchor

foa = data\_utils.**get\_field\_of\_anchors(** **stride**, **anchor\_sizes**, **anchor\_aspect\_ratios**, octave, idx)

下面这个表格以P3层为例，（训练图片的大小为1024×1024，其余参数皆为默认），可以看出对于P3层生成anchor总数为16384×9。



之后通过代码foas.append(foa)，将每一次生成的foa都添加到foas中。一层FPN要添加9个foa，走完上面三个for循环，foas就会有5（p3-p7）×9=45个foa。最后通过all\_anchors = np.concatenate([f.field\_of\_anchors for f in foas])，将所有层的anchor放在一起，如下：



（1）两种类型的网络默认anchor\_aspect\_ratios相同，为1.0 2.0 0.5

（2）RPN+FPN网络，每一层FPN只有一种anchor\_sizes，由于3种长宽比，生成1×3=3种

类型的anchor。

（3）retianet网络，每一层FPN产生三种anchor\_sizes，再加上3种长宽比，生成3×3=9种

类型的anchor。

（4）无论是FPN生成anchor还是Retianet生成anchor都调用了get\_field\_of\_anchors这个

函数.

（5）all\_anchor是将所有FPN层的anchor放在一起.

## 构成输入blob

### **针对每一张roidb生成blob，主要调用\_get\_retinanet\_blob函数**

针对每一张送入的样本图片，获取其gt信息，主要包括：

* gt\_rois：gt的box
* gt\_classes ：gt的label
* im\_height ：图片的高度（放缩后）
* im\_width ：图片的宽度（放缩后）

 计算重叠率，并根据重叠率计算labels: # label=1 正样本, 0 负样本, -1 忽略

1. 将与gt重叠率最大的anchor的label设置为gt的label;
2. 将与gt重叠率大于cfg.RETINANET.POSITIVE\_OVERLAP的anchor的label设置为gt的label;
3. 将与gt重叠率小于cfg.RETINANET. NEGATIVE\_OVERLAP的anchor的

label设置为bg的label;

1. 其余的anchor的label为-1.

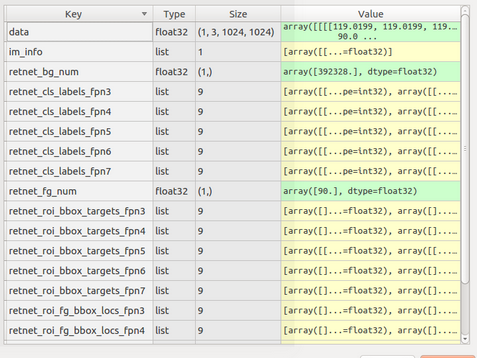
 计算bbox\_targets，即平移量（tx,ty）与尺度因子（tw,th)，其输入是：

#(1)anchors[fg\_inds, :]：被标为正样本的anchor

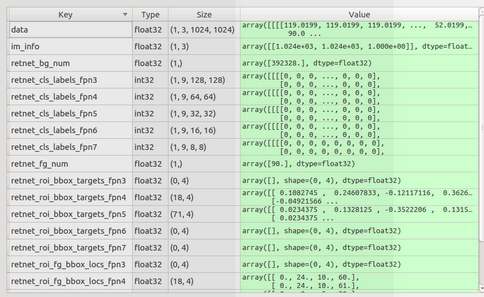
#(2)gt\_boxes[anchor\_to\_gt\_argmax[fg\_inds], :]：与这些正样本重叠最高的gt

**备注**：平移量是中心点的距离

代码中第一个for循环是按照foas中的顺序开始，第二个for循环retinanet\_blobs[i].items()保持了与foa的一一对应。所以无论第二个for循环怎么改，只要第一个for循环不进行下去，level = int(np.log2(foa.stride))是不变的。因此这一段代码就是将retinanet\_blob中同一FPN层的不同类型的anchor对应的retnet\_cls\_labels，retnet\_roi\_bbox\_targets和retnet\_roi\_fg\_bbox\_locs合并在一起。所以最后blob如下，例如相应的retnet\_cls\_labels\_fpn3变成了含有9个list的字典。



**将list合并为矩阵的形式**



## 预测

test\_net.py

* run\_inference()
* result\_getter() -> child\_func() 就是test\_net\_on\_dataset ()
* test\_net()
* im\_detect\_all() core/test.py ->test\_retinanet.**im\_detect\_bbox**(model, im, timers)

core/test\_retinanet.py 主要的预测函数

all\_boxes, all\_segms, all\_keyps = **test\_net**(weights\_file, dataset\_name, proposal\_file,

output\_dir, gpu\_id=gpu\_id)

**模型评测：**

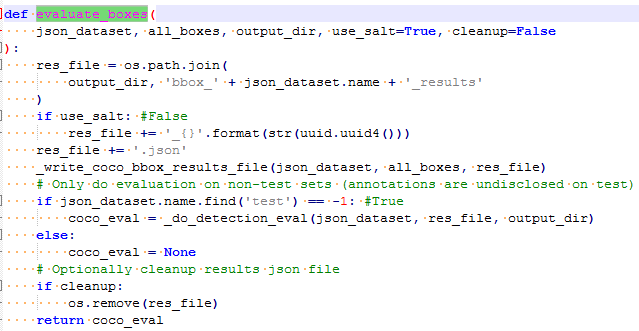
* evaluate\_all() datasets/task\_evaluation.py

results = task\_evaluation.**evaluate\_all**(dataset, all\_boxes, all\_segms, all\_keyps, output\_dir)

* evaluate\_boxes()
* **evaluate\_boxes()**

coco\_eval = json\_dataset\_evaluator.**evaluate\_boxes**(dataset, all\_boxes, output\_dir, use\_salt=not\_comp, cleanup=not\_comp)

* \_coco\_eval\_to\_box\_results(coco\_eval)
* **\_do\_detection\_eval**()



**备注：**

res\_file = os.path.join(output\_dir, 'bbox\_' + json\_dataset.name + '\_results')+ '.json'

#./test/coco\_2007\_val\_jiuquan/retinanet/bbox\_coco\_2007\_val\_jiuquan\_results.json

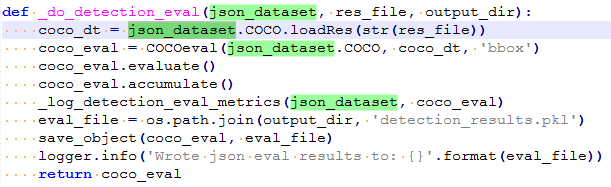
**\_write\_coco\_bbox\_results\_file**(json\_dataset, all\_boxes, res\_file)

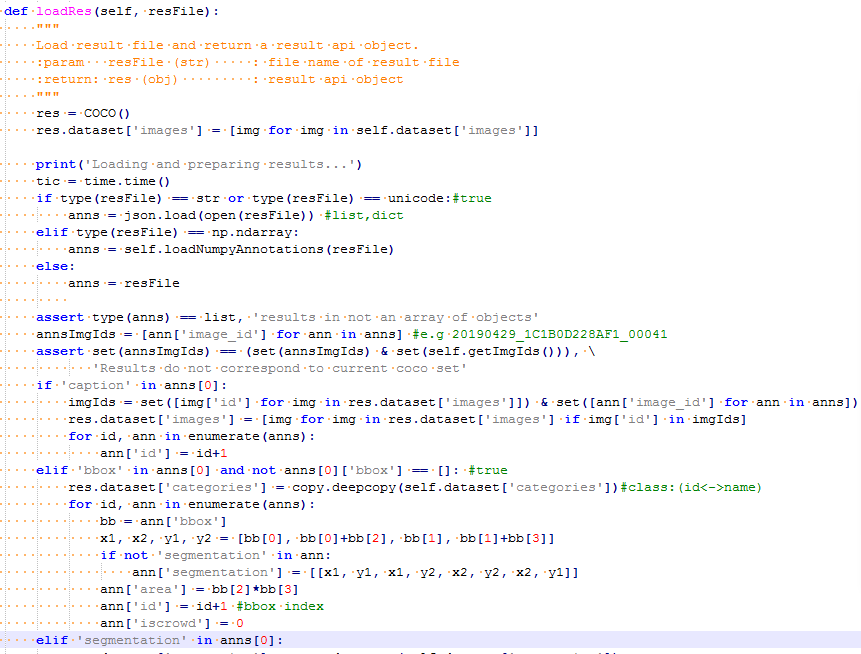
#将预测的bbox信息写到res\_file中，如下图所示：

C:\Users\ZHANGJ~2\AppData\Local\Temp\1563864187(1).png

加载gts、dts； computeIoU(dt,gt);

### \_do\_detection\_eval()





**备注：**

**self.dataset['images']**:

[{u**'file\_name**':u'20190429\_1C1B0D228AF1\_00041.jpg',u'height':1080,

u'id': u'20190429\_1C1B0D228AF1\_00041', u'width': 1920},

{u'**file\_name**':u'20190621\_B06EBF342E00\_01324.jpg',u'height':1080,

u'id': u'20190621\_B06EBF342E00\_01324', u'width': 1920}]

**res.dataset['categories']**:

[{u'supercategory': u'none', u'id': 1, u'name': u'suv'}, {u'supercategory': u'none', u'id': 2, u'name': u'forklift'}, {u'supercategory': u'none', u'id': 3, u'name': u'digger'}, {u'supercategory': u'none', u'id': 4, u'name': u'car'}, {u'supercategory': u'none', u'id': 5, u'name': u'bus'}, {u'supercategory': u'none', u'id': 6, u'name': u'tanker'}, {u'supercategory': u'none', u'id': 7, u'name': u'person'}, {u'supercategory': u'none', u'id': 8, u'name': u'minitruck'}, {u'supercategory': u'none', u'id': 9, u'name': u'minibus'}, {u'supercategory': u'none', u'id': 10, u'name': u'truckbig'}, {u'supercategory': u'none', u'id': 11, u'name': u'trucksmall'}, {u'supercategory': u'none', u'id': 12, u'name': u'tricycle'}, {u'supercategory': u'none', u'id': 13, u'name': u'bicycle'}])

x1, x2, y1, y2 = [bb[0], bb[0]+bb[2], bb[1], bb[1]+bb[3]] # x,y,w,h,float

**ann['area']** = bb[2]\*bb[3]

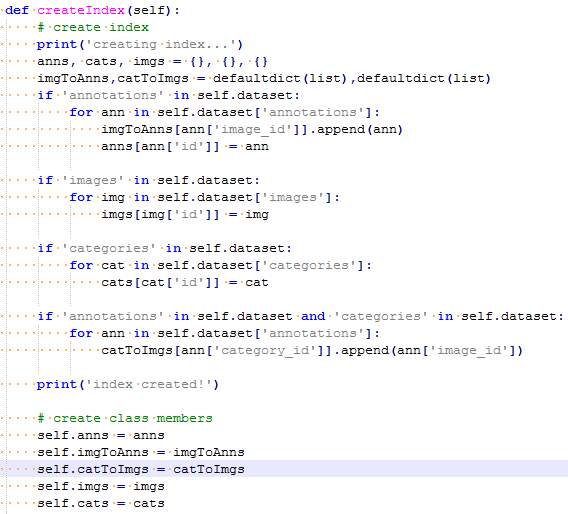
**ann['id']** = id+1 #bbox index，从1开始

ann['iscrowd'] = 0

res.**dataset['annotations']** = anns

res.createIndex()

return res



**ann:**

{'segmentation': [[1705.996216, 534.834595, 1705.996216, 1080.0, 1920.0, 1080.0, 1920.0, 534.834595]], 'area': 116667.45957589251, 'iscrowd': 0, u'image\_id': u'20190429\_1C1B0D228AF1\_00041', u'score': 0.006158, u'bbox': [1705.996216, 534.834595, 214.003784, 545.165405], u'category\_id': 1, **'id': 1}**

{'segmentation': [[1421.506348, 706.502258, 1421.506348, 954.121277, 1920.0, 954.121277, 1920.0, 706.502258]], 'area': 123436.5090859674, 'iscrowd': 0, u'image\_id': u'20190429\_1C1B0D228AF1\_00041', u'score': 0.004639, u'bbox': [1421.506348, 706.502258, 498.493652, 247.619019], u'category\_id': 1, **'id': 2**},..}

**self.anns** = anns: #按照bbox索引进行存放ann

{**1:** {'segmentation': [[1705.996216, 534.834595, 1705.996216, 1080.0, 1920.0, 1080.0, 1920.0, 534.834595]], 'area': 116667.45957589251, 'iscrowd': 0, u'image\_id': u'20190429\_1C1B0D228AF1\_00041', u'score': 0.006158, u'bbox': [1705.996216, 534.834595, 214.003784, 545.165405], u'category\_id': 1, 'id': 1},

**2:** {'segmentation': [[1421.506348, 706.502258, 1421.506348, 954.121277, 1920.0, 954.121277, 1920.0, 706.502258]], 'area': 123436.5090859674, 'iscrowd': 0, u'image\_id': u'20190429\_1C1B0D228AF1\_00041', u'score': 0.004639, u'bbox': [1421.506348, 706.502258, 498.493652, 247.619019], u'category\_id': 1, 'id': 2},..}

**imgToAnns： 图片名称：anns（bbox等信息）**

**catToImgs: 类别索引：图片名称**

{**1**: [u'20190429\_1C1B0D228AF1\_00041', u'20190429\_1C1B0D228AF1\_00041',

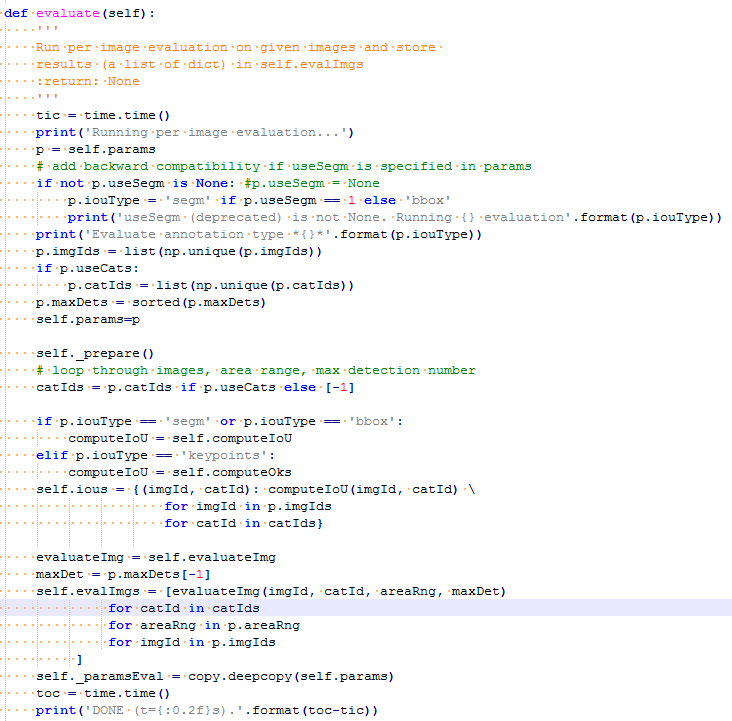
u'20190621\_B06EBF342E00\_01324', u'20190621\_B06EBF342E00\_01324'],

**2**:[u'20190429\_1C1B0D228AF1\_00041',u'20190429\_1C1B0D228AF1\_00041', u'20190429\_1C1B0D228AF1\_00041', ], …}

**imgs：图片名称：图片信息**

**cats：类别索引：类别信息**

### evaluate()：统计每张图每个类别的IoU、area的分段情况

****

**p.imgIds:** ['20190429\_1C1B0D228AF1\_00041', '20190621\_B06EBF342E00\_01324'] **图片名称**

**p.catIds:** [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13] **类别索引**

**p.maxDets: [1, 10, 100] #每张图的检测到最多bbox，最多100次**

**\_prepare():**

**gts ： 存放xml的bbox信息**

[{u'segmentation': [[1440, 699, 1440, 935, 1750, 935, 1750, 699]], u'area': 73160,

u'iscrowd': 0, u'ignore': 0, u'image\_id': u'20190429\_1C1B0D228AF1\_00041', u'bbox':

[1440, 699, 310, 236], u'category\_id': 3, u'id': 1},

{u'segmentation': [[792, 734, 792,

814, 833, 814, 833, 734]], u'area': 3280, u'iscrowd': 0, u'ignore': 0, u'image\_id':

u'20190621\_B06EBF342E00\_01324', u'bbox': [792, 734, 41, 80], u'category\_id': 7,

u'id': 2}])

**dts: 存放检测结果的bbox信息**

[{'segmentation': [[1705.996216, 534.834534, 1705.996216, 1080.0, 1920.0,1080.0,

1920.0, 534.834534]], 'area': 116667.47263012335, 'iscrowd': 0, u'image\_id': u'20190429\_1C1B0D228AF1\_00041', u'score': 0.006158, u'bbox': [1705.996216, 534.834534, 214.003784, 545.165466], u'category\_id': 1, 'id': 1},

{'segmentation': [[1421.506348, 706.502258, 1421.506348, 954.121277, 1920.0, 954.121277, 1920.0, 706.502258]], 'area': 123436.5090859674, 'iscrowd': 0, u'image\_id': u'20190429\_1C1B0D228AF1\_00041', u'score': 0.004639, u'bbox': [1421.506348, 706.502258, 498.493652, 247.619019], u'category\_id': 1, 'id': 2},..]

**self.\_gts：**('image\_id', 'category\_id')

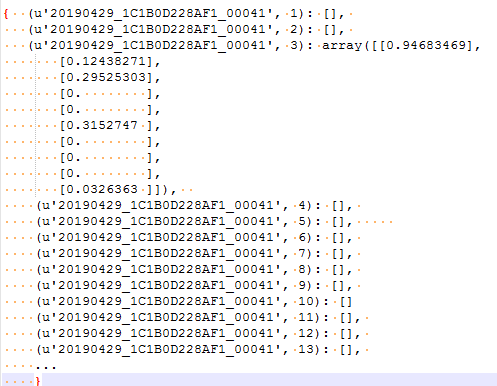
defaultdict(<type 'list'>,

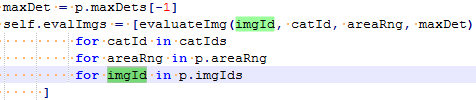
{(u'20190621\_B06EBF342E00\_01324', 7): [{u'segmentation': [[792, 734, 792, 814, 833, 814, 833, 734]], u'area': 3280, u'iscrowd': 0, u'ignore': 0, u'image\_id': u'20190621\_B06EBF342E00\_01324', u'bbox': [792, 734, 41, 80], u'category\_id': 7, u'id': 2}],

(u'20190429\_1C1B0D228AF1\_00041', 3): [{u'segmentation': [[1440, 699, 1440, 935, 1750, 935, 1750, 699]], u'area': 73160, u'iscrowd': 0, u'ignore': 0, u'image\_id': u'20190429\_1C1B0D228AF1\_00041', u'bbox': [1440, 699, 310, 236], u'category\_id': 3, u'id': 1}]})

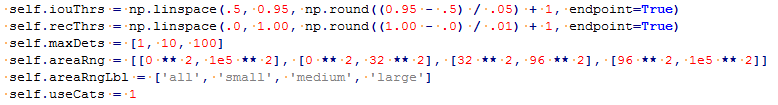
**C:\Users\ZHANGJ~2\AppData\Local\Temp\1563945331(1).png**

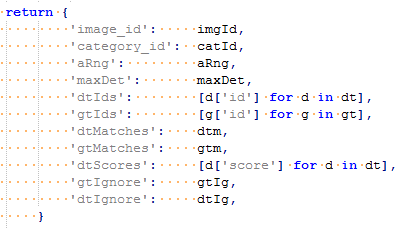
**self.ious的结果如下图所示： （图片名称，类别id）图片中只有一个目标(id=3)**

****

**** len(p.catIds),len(p.areaRng),len(p.imgIds) = （13,4,2）

#### evaluateImg()：统计每张图每个类别的IoU、area的分段情况

****

****

'image\_id': imgId, #单张的，图片名称

'category\_id': catId, #类别索引，1-13中的一个

'aRng': aRng, #面积区域，4个区域中的一个

'maxDet': maxDet, #100

'dtIds': [d['id'] for d in dt],

'gtIds': [g['id'] for g in gt],

'dtMatches': dtm, （iou分段的长度, dt的长度）这张图、这个类别、满足这个面积

'gtMatches': gtm, （iou分段的长度, gt的长度）这张图、这个类别、满足这个面积

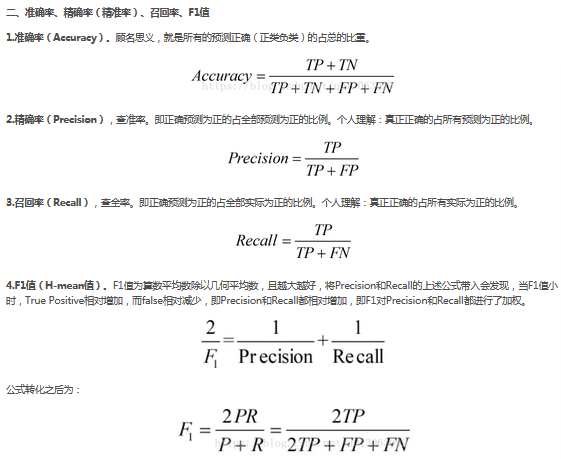
'dtScores': [d['score'] for d in dt],

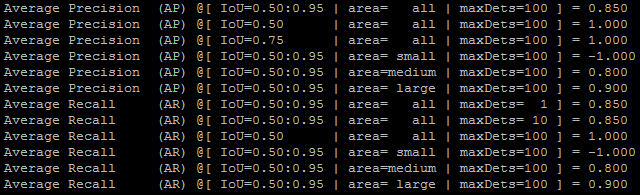
'gtIgnore': gtIg,

'dtIgnore': dtIg,

### accumulate()：precision、recall、scores

****





Recall=检测出率，下面2张的recall：

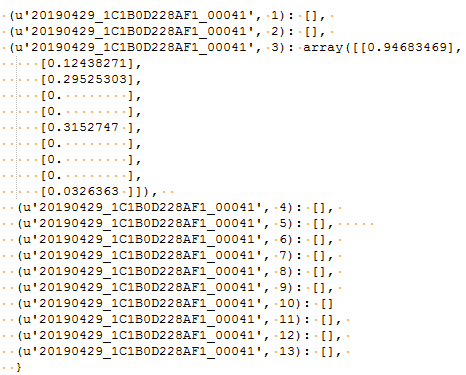
Average Recall (AR) @[ IoU=0.50 | area= all | maxDets=100 ] = 1.000

AP：每个类别的平均值

Average Precision (AP) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.850

'20190429\_1C1B0D228AF1\_00041'、类别：IoU值(gt,dt) **digger-3, (1441,700,1750,935)**

**diggerAP=90 (面积在[0, 1e5 \*\* 2], iou在[0.5,0.95,0.05]的平均值) 与score无关**

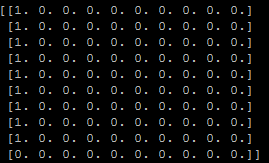


if catId==3:

print (dtm)

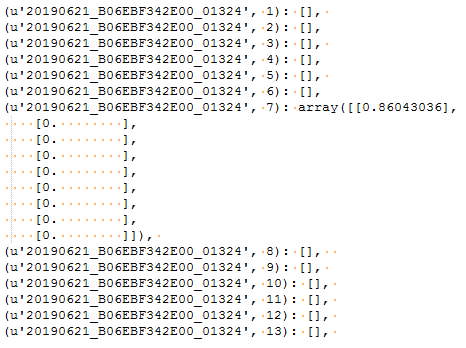
dtm = np.zeros((T,D)) #（iou分段数，dt个数）#(10,10)

iou分成10段：[0.5 , 0.55, 0.6 , 0.65, 0.7 , 0.75, 0.8 , 0.85, 0.9 , 0.95]



'20190621\_B06EBF342E00\_01324'、类别：IoU值(gt,dt) **person-7, (793,735,833,814)**

**personAP=80 (面积在[0, 1e5 \*\* 2], iou在[0.5,0.95,0.05]的平均值)**



if catId==7:

print (dtm)

dtm = np.zeros((T,D)) #（iou分段数，dt个数） #(10,9)

iou分成10段：[0.5 , 0.55, 0.6 , 0.65, 0.7 , 0.75, 0.8 , 0.85, 0.9 , 0.95]

