RetinaNet技术文档

# 概述：

## 资料地址：

### 论文地址

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

《Focal Loss for Dense Object Detection》

Tsung-Yi Lin、Priya Goyal、Ross Girshick、Kaiming He、Piotr Dollar

Facebook AI Research (FAIR) 2018年2月

### 代码地址

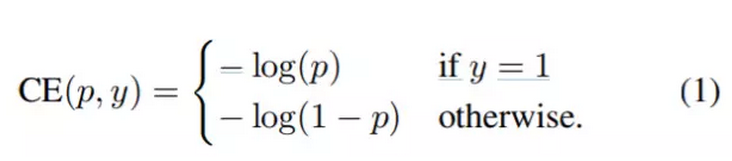
<https://github.com/facebookresearch/Detectron>

### 镜像地址

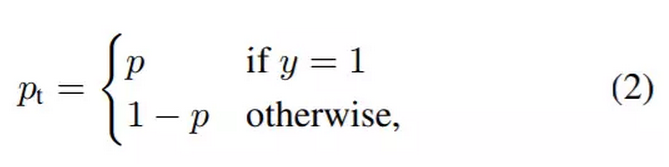
192.168.15.100：/home/tar/detectron\_gpu.tar

## 技术理论

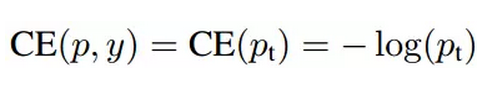
首先，我们介绍下二进制分类(binary classification)的交叉熵(CE)损失函数：



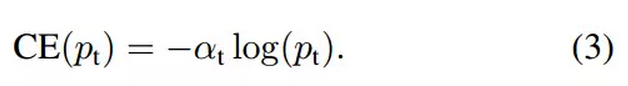
公式1中，y∈{±1} 指定了 ground-truth class，p∈[0,1] 是模型对于标签为 y = 1 的类的估计概率。为了方便起见，我们定义 pt 为：



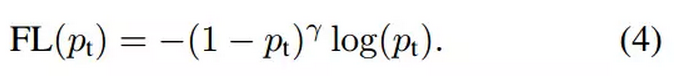
公式1可以转写称：



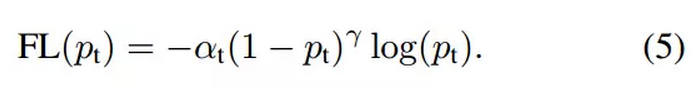
解决 class imbalance 的常见方法是分别为 class 1 和 class -1 引入加权因子 α∈[0; 1]、1-α。 α-balanced 的CE损耗可写为：



更正式地，我们建议为交叉熵损失增加一个调节因子（1 - pt）γ，其中 γ≥0。于是 Focal Loss 可定义为：

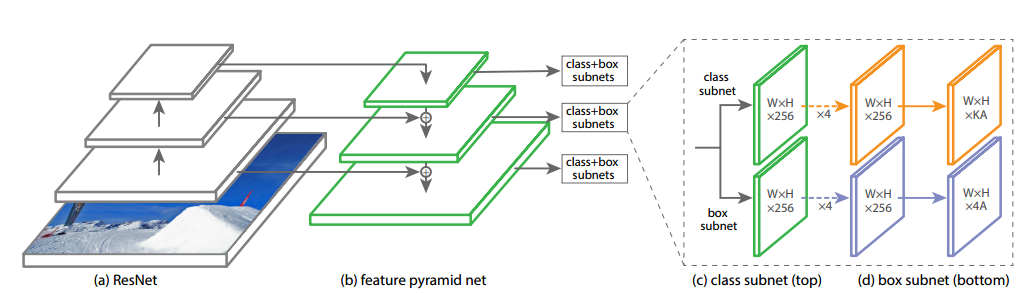


以下是我们在实践中使用的 Focal Loss：

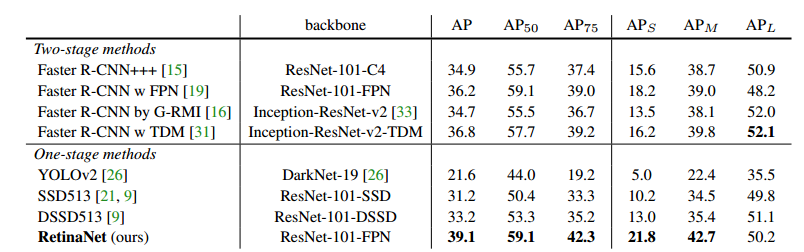


RetinaNet 是由一个骨干网络和两个特定任务子网组成的单一网络。

骨干网络负责在整个输入图像上计算卷积特征图，并且是一个现成的卷积网络。 两个特定任务子网： 第一个子网在骨干网络的输出上执行卷积对象分类；第二个子网执行卷积边界框回归。如下图所示。



下表是目标检测单模型结果（边界框AP）VS COCO test-dev 最先进的方法：



## 补充理论

**参考文献**：<https://zhuanlan.zhihu.com/p/28873248>

<https://blog.csdn.net/qq_34564947/article/details/77200104>

### 创新点1：focal loss，**控制正负样本的权重、容易分类和难分类样本的权重**

**引入问题：**为什么one-stage的检测器准确率不高的问题？

**原因：**样本的类别不均衡导致

**Class Imbalance:**之前的one-stage方法训练时都面临类别不平衡问题。要评估10^4-10^5候选定位，但是只有很少包含对象。类别不平衡导致两个问题：

1. 训练不高效，大多数位置都是easy negtive，贡献很少的有用学习信号（useful learning signal）。

（2）easy negative 会overwhelm主导训练，导致退化的模型（degenerate models）。

**解决办法**：提出focal loss，控制正负样本的权重、容易分类和难分类样本的权重。

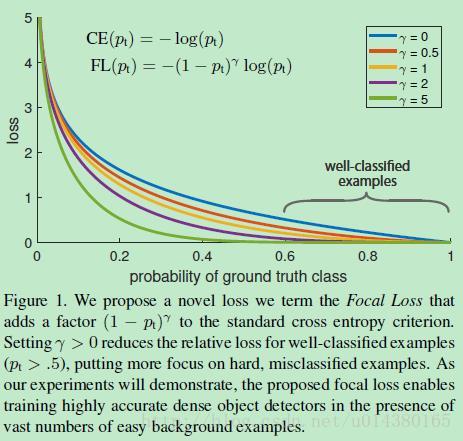
C:\Users\zhangjing1\AppData\Roaming\JunDaoIM\tempImages\image_kA.png

我们知道object detection的算法主要可以分为两大类：**two-stage detector和one-stage detector**。前者是指类似Faster RCNN，RFCN这样需要region proposal的检测算法，这类算法可以达到很高的准确率，但是速度较慢。虽然可以通过减少proposal的数量或降低输入图像的分辨率等方式达到提速，但是速度并没有质的提升。后者是指类似YOLO，SSD这样不需要region proposal，直接回归的检测算法，这类算法速度很快，但是准确率不如前者。**作者提出focal loss的出发点也是希望one-stage detector可以达到two-stage detector的准确率，同时不影响原有的速度。**

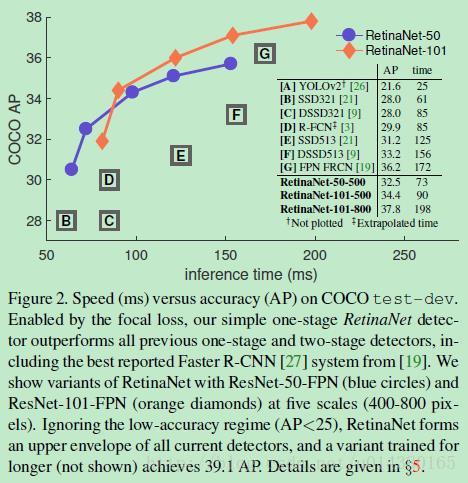
既然有了出发点，**那么就要找one-stage detector的准确率不如two-stage detector的原因，作者认为原因是：样本的类别不均衡导致的**。我们知道在object detection领域，一张图像可能生成成千上万的candidate locations，但是其中只有很少一部分是包含object的，这就带来了类别不均衡。**那么类别不均衡会带来什么后果呢？**引用原文讲的两个后果：**(1) training is inefficient as most locations are easy negatives that contribute no useful learning signal; (2) en masse, the easy negatives can overwhelm training and lead to degenerate models.** 什么意思呢？**负样本数量太大，占总的loss的大部分，而且多是容易分类的，因此使得模型的优化方向并不是我们所希望的那样**。其实先前也有一些算法来处理类别不均衡的问题，比如OHEM（online hard example mining），OHEM的主要思想可以用原文的一句话概括：In OHEM each example is scored by its loss, non-maximum suppression (nms) is then applied, and a minibatch is constructed with the highest-loss examples。**OHEM算法虽然增加了错分类样本的权重，但是OHEM算法忽略了容易分类的样本。**

（对于two-stage的检测器而言，通常分为两个步骤，第一个步骤即产生合适的候选区域，而这些候选区域经过筛选，一般控制一个比例（比如正负样本1：3），另外还通过**hard negatiive mining(OHEM)**，控制难分样本占据的比例，以解决样本类别不均衡的问题。）

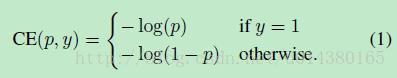
**因此针对类别不均衡问题，作者提出一种新的损失函数：focal loss，这个损失函数是在标准交叉熵损失基础上修改得到的。这个函数可以通过减少易分类样本的权重，使得模型在训练时更专注于难分类的样本。**为了证明focal loss的有效性，作者设计了一个dense detector：**RetinaNet**，并且在训练时采用focal loss训练。**实验证明RetinaNet不仅可以达到one-stage detector的速度，也能有two-stage detector的准确率。**



**focal loss的含义可以看如下Figure1**，横坐标是pt，纵坐标是loss。CE（pt）表示标准的交叉熵公式，FL（pt）表示focal loss中用到的改进的交叉熵，可以看出和原来的交叉熵对比多了一个调制系数（modulating factor）。为什么要加上这个调制系数呢？**目的是通过减少易分类样本的权重，从而使得模型在训练时更专注于难分类的样本。**首先pt的范围是0到1，所以不管γ是多少，这个调制系数都是大于等于0的。易分类的样本再多，你的权重很小，那么对于total loss的共享也就不会太大。那么怎么控制样本权重呢？举个例子，假设一个二分类，样本x1属于类别1的pt=0.9，样本x2属于类别1的pt=0.6，显然前者更可能是类别1，假设γ=1，那么对于pt=0.9，调制系数则为0.1；对于pt=0.6，调制系数则为0.4，这个调制系数就是这个样本对loss的贡献程度，也就是权重，所以难分的样本（pt=0.6）的权重更大。**Figure1中γ=0的蓝色曲线就是标准的交叉熵损失。**

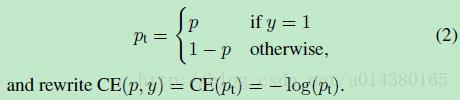


看完实验结果和提出算法的出发点，接下来就要介绍focal loss了。在介绍focal loss之前，先来看看交叉熵损失，这里以二分类为例，p表示概率，公式如下：



因为是二分类，所以y的值是正1或负1，p的范围为0到1。当真实label是1，也就是y=1时，假如某个样本x预测为1这个类的概率p=0.6，那么损失就是-log(0.6)，注意这个损失是大于等于0的。如果p=0.9，那么损失就是-log(0.9)，所以p=0.6的损失要大于p=0.9的损失，这很容易理解。

为了方便，用pt代替p，如下公式2:。这里的pt就是前面Figure1中的横坐标。



接下来介绍一个最基本的对交叉熵的改进，也将作为本文实验的baseline，如下公式3。什么意思呢？增加了一个系数at，跟pt的定义类似，当label=1的时候，at=a；当label=-1的时候，at=1-a，a的范围也是0到1。**因此可以通过设定a的值（一般而言假如1这个类的样本数比-1这个类的样本数多很多，那么a会取0到0.5来增加-1这个类的样本的权重）来控制正负样本对总的loss的共享权重。**

这里写图片描述

**显然前面的公式3虽然可以控制正负样本的权重，但是没法控制容易分类和难分类样本的权重**，于是就有了focal loss：

这里写图片描述

这里的γ称作focusing parameter，γ>=0。

这里写图片描述

称为调制系数（modulating factor）   
**这里介绍下focal loss的两个重要性质：1、当一个样本被分错的时候，pt是很小的（请结合公式2，比如当y=1时，p要小于0.5才是错分类，此时pt就比较小，反之亦然），因此调制系数就趋于1，也就是说相比原来的loss是没有什么大的改变的。当pt趋于1的时候（此时分类正确而且是易分类样本），调制系数趋于0，也就是对于总的loss的贡献很小。2、当γ=0的时候，focal loss就是传统的交叉熵损失，当γ增加的时候，调制系数也会增加。**   
**focal loss的两个性质算是核心，其实就是用一个合适的函数去度量难分类和易分类样本对总的损失的贡献。**

**作者在实验中采用的是公式5的focal loss（结合了公式3和公式4，这样既能调整正负样本的权重，又能控制难易分类样本的权重）：**

这里写图片描述

在实验中a的选择范围也很广，**一般而言当γ增加的时候，a需要减小一点**（实验中γ=2，a=0.25的效果最好）

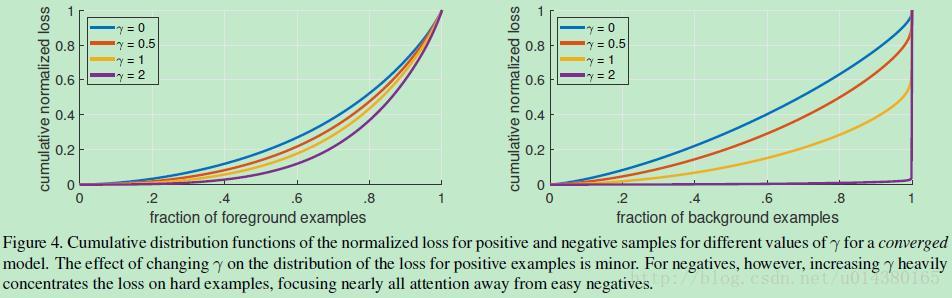


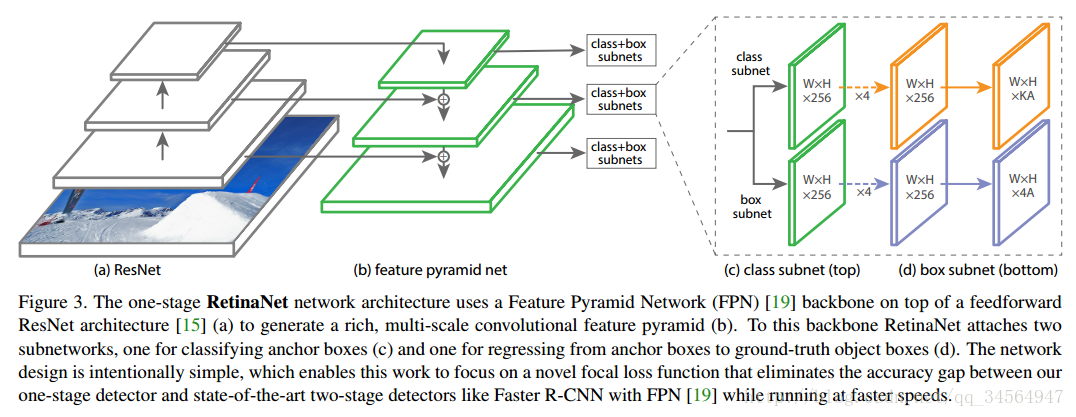
Figure4是对比forground和background样本在不同γ情况下的累积误差。纵坐标是归一化后的损失，横坐标是总的foreground或background样本数的百分比。可以看出γ的变化对正（forground）样本的累积误差的影响并不大，但是对于负（background）样本的累积误差的影响还是很大的（**γ=2时，将近99%的background样本的损失都非常小**）。

### 创新点2：Model Initialization

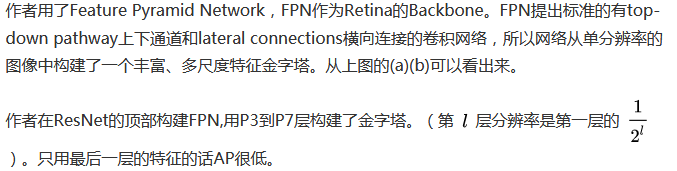
Binary分类模型是默认初始化为对于y=-1和y=1有相同的概率的。在这样的初始化之下，由于类不平衡，出现频率高的类会主导总的损失，在训练早期导致不稳定。为了对抗这个，作者提出“优先”的概念，在训练初期对于模型对于低频率的类（背景）估计的p给予“优先”。作者把这个**“优先”（prior）记做 pi**,设定它，以至于模型对于低频率类别（rare class）的样本的估计p很低，比如说0.001。这是模型初始化的改变，而不是损失函数的改变。作者发现这点能改进训练的稳定性（对于在类极不平衡的情况下的交叉熵和focal loss都有效）。

### 实验框架

本文作者实验时设计了一个叫**RetinaNet的one-satge的网络结构**，以证明通过Focal Loss，one-stage的网络结构也能够达到two-stage的准确率，实际上采用的是基于**resnet的FPN**（特征金字塔网络，可自行查阅论文[论文链接](https://arxiv.org/pdf/1612.03144.pdf)），网络框架如下：



### Feature Pyramid Network Backbone: P3-P7(1.0 、2.0 、0.5)



#### ResNet+FPN的结构图：

输入图像的宽高是按照同一个比例缩放，所以图像中的目标不存在变形问题。

**P3：**分类回归

Res3大层的最后一层

**上采样**

**P4:**分类回归

Res4层的最后一层

**横向连接**

**P5:**分类回归

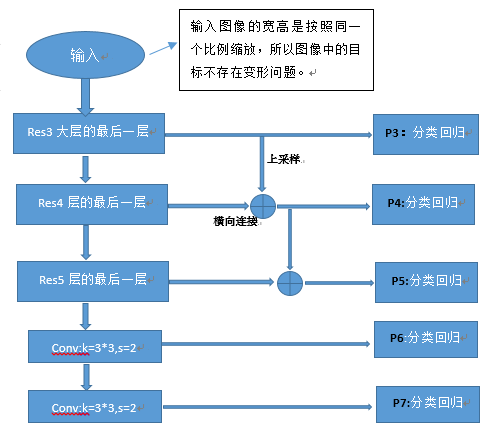
Res5层的最后一层

**P6**:分类回归

Conv:k=3\*3,s=2

**P7:**分类回归

Conv:k=3\*3,s=2



**配置文件**：configs/12\_2017\_baselines /retinanet\_R-101-FPN\_1x.yaml

**输入图片：**假如输入图片大小为1920\*1080h,配置文件中（TRAIN: SCALES: (800,) MAX\_SIZE:

1333，FPN: COARSEST\_STRIDE: 128），网络真正的输入**1408\*768h**

1. 从1920、1080中取最小的1080，用1080/800=1.35，1920/1.35=**1422.2>1333**，
2. 1920/1333=1.44, 1080/1.44=749.8
3. 1333/128=10.4(取11)，11\*128 =1408
4. 749.8/128=5.857（取6），6\*128=768，（除128再乘128是为了对图像补黑边）

假如输入图片大小为1920\*1080h,配置文件中（TRAIN: SCALES: (830,) MAX\_SIZE:

1. FPN: COARSEST\_STRIDE: 128），络真正的输入**1536\*896h**

1) 从1920、1080中取最小的1080，用1080/830=1.30，1920/1.30=**1476.9<1480**，

2) 830/128=6.48(取7)， 7\*128=896

3)1476.9/128=11.5(取12)，12\*128=1536

**基础网络：ResNet101 + FPN**

**P3-p5:** 是FPN的结构

**P3:** 特征图大小=原图/2^3，宽高比{1:1、1:2、2:1}

**P4:** 特征图大小=原图/2^4，宽高比{1:1、1:2、2:1}

**P5:** 特征图大小=原图/2^5，宽高比{1:1、1:2、2:1}

**P6:** 特征图大小=原图/2^6，宽高比{1:1、1:2、2:1}

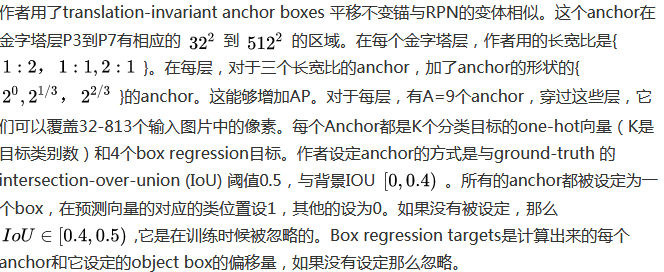
**P7:** 特征图大小=原图/2^7，宽高比{1:1、1:2、2:1}

由于计算量的问题，没有使用高分辨率金字塔P2;

P6、P7是通过卷积进行分类回归的，而不是采用下采样；

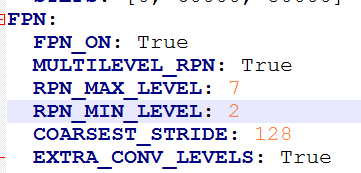
P7改善大目标检测。

#### Anchors: 每层的anchor个数是一样的



#### 参数说明

##### 修改FPN的层数



如上图，RPN\_MAX\_LEVEL是最高层，RPN\_MIN\_LEVEL是最低层；

当FPN\_ON为TRUE, MULTILEVEL\_RPN 为true时（retinaNet-FPN这样配置），使用RPN\_MAX\_LEVEL和RPN\_MIN\_LEVEL。

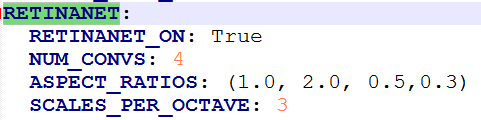
当FPN\_ON为TRUE, MULTILEVEL\_RPN 为true，cfg.FPN.MULTILEVEL\_ROIS也为true时，（ mask rcnn-FPN这样配置），使用RPN\_MAX\_LEVEL和ROI\_MAX\_LEVEL的最大值，RPN\_MIN\_LEVEL和ROI\_MIN\_LEVEL的最小值。

另外，这里的COARSEST\_STRIDE表示FPN的最高层的缩放比例。如果RPN\_MAX\_LEVEL或者ROI\_MAX\_LEVEL修改了，该值也应该修改。

LOSS\_GAMMA: 2.0

LOSS\_ALPHA: 0.25

##### 修改anchors比例和scale



RetinaNet中ASPECT\_RATIOS表示宽高比例，SCALES\_PER\_OCTAVE表示每层anchors分为几个尺度。

如果不是retinaNet，如果是maskrcnn等，分成两种：**使用FPN和不使用FPN。**

如果使用了FPN，每一层anchors的ratios比例都是由FPN.RPN\_ASPECT\_RATIOS统一配置的。每一层anchors的scale只有一个，由FPN.RPN\_ANCHOR\_START\_SIZE决定。该值默认为32，随着FPN层的升高，该值会以2倍的比例升高。

如果不使用FPN，根据RPN.STRIDE，RPN.SIZES和RPN.ASPECT\_RATIOS一次生成所有的anchors。这里RPN.STRIDE默认为16，因为ResNet都是从C4层连接RPN的，C4层对应原网络图的缩放比例是16。如下面是e2e\_mask\_rcnn\_R-50-C4\_1x.yaml的配置：



这里RPN.SIZES最小是32，对应了C4特征图中的两个像素。

##### Focal Loss参数：

LOSS\_GAMMA: 2.0

LOSS\_ALPHA: 0.25（实验中γ=2，a=0.25的效果最好）

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

#### 实验说明

##### 默认anchor：3个ratio、3个scale、P2-P7

**配置文件：分析默认的**configs/12\_2017\_baselines /retinanet\_R-101-FPN\_1x.yaml

**FPN**: MULTILEVEL\_RPN=True，RPN\_MAX\_LEVEL=7，RPN\_MIN\_LEVEL=3,

COARSEST\_STRIDE=128；

**RETINANET**: SCALES\_PER\_OCTAVE=3,ANCHOR\_SCALE=4，ASPECT\_RATIOS={1.0, 2.0, 0.5}默认；

图片大小：1408\*768 （配置为800，1333）

上面的配置中，每一层对应的 1:1的anchors的宽高对应特征图上的像素个数分别为：

4\*，4\*， 4\*，即4，6，7。 (anchor面积)

因此，在P3,P4,P5,P6,P7层分类回归，这样每一层对应的anchors尺寸如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| FPN层 | Stride | 最小anchors的宽高对应网络输入图的尺寸的像素个数 | 最大anchors的宽高对应网络输入图的尺寸的像素个数 | 最小anchors的宽高对应原图的尺寸的像素个数 | 最大anchors的宽高对应原图的尺寸的像素个数 |
| P3 | 8 | 32 （4\*8） | 56 （7\*8） | 46 （32\*1.44） | 80 （56\*1.44） |
| P4 | 16 | 64 （4\*16） | 112 （7\*16） | 92 （64\*1.44） | 160（112\*1.44） |
| P5 | 32 | 128 （4\*32） | 224 （7\*32） | 184 （128\*1.44） | 320（224\*1.44） |
| P6 | 64 | 256 （4\*64） | 448 （7\*64） | 368（256\*1.44） | 640（448\*1.44） |
| P7 | 128 | 512 （4\*128） | 896 （7\*128） | 736（512\*1.44） | 1280（896\*1.44） |

注意上表中，当最小anchors的宽高对应网络输入图的尺寸的像素个数转换到对应原图的尺寸的像素个数时，乘以比例1.44，这里的1.44是缩放比例（1920/1333=1080/750=1.44）。（原图的尺寸是1920\*1080，配置为1333\*800）

##### P2至P6，6个anchors比例和4个scale

**FPN:**RPN\_MAX\_LEVEL修改为6，RPN\_MIN\_LEVEL修改为 2, FPN.COARSEST\_STRIDE修改为64；因为FPN增加了P2之后，很耗内存。原因是P2层的feature map较大， P2对应的anchors比较多。所以GTX1080只能支持batchsize为1。

这次刚好有两块GPU， 所以其他值都尽量调大了。

**RETINANET:** SCALES\_PER\_OCTAVE修改为 4， ANCHOR\_SCALE为 4，ASPECT\_RATIOS修改为 (1.0, 2.0, 3.0，0.5，0.4，0.3)；

训练时的SCALES修改为900，MAX\_SIZE修改为1600。 因此，最后的网络输入图的尺寸为960\*1600。输入图的尺寸和原图的尺寸的缩放比例为1.2（1920/1600=1080/900=1.2）。

上面的配置中，每一层对应的 1:1的anchors的宽高对应特征图上的像素个数分别为： 4\*，4\*， 4\*，即4，5，6，7。

这样最终只在P2,P3,P4,P5,P6层分类回归，对应的anchors尺寸如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| FPN  层 | Stride | 最小anchors的宽高对应网络输入图的尺寸的像素个数 | 最大anchors的宽高对应网络输入图的尺寸的像素个数 | 最小anchors的宽高对应原图的尺寸的像素个数 | 最大anchors的宽高对应原图的尺寸的像素个数 |
| P2 | 4 | 16 （4\*4） | 28 （7\*4） | 19 （16\*1.2） | 33 （28\*1.2） |
| P3 | 8 | 32 （4\*8） | 56 （7\*8） | 38 （34\*1.2） | 66 （56\*1.2） |
| P4 | 16 | 64 （4\*16） | 112 （7\*16） | 76 （64\*1.2） | 132 （112\*1.2） |
| P5 | 32 | 128 （4\*32） | 224 （7\*32） | 152 （128\*1.2） | 264 （224\*1.2） |
| P6 | 64 | 256 （4\*64） | 448 （7\*64） | 304 （256\*1.2） | 528 （448\*1.2） |

因此，这次配置中最小的anchors尺度为19，默认配置中最小的anchors尺度为46。而且增加了一些anchors比例，理论上这次比之前的检测效果要好很多。

##### 实验对比

**类别**：举手、点赞、反对、睡觉

**数据集说明：** 训练图片中没有网络图片（只还有举手、点赞这2类），其它的训练图片都是自

己拍摄的、场景比较单一，还有教室图片。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 实验配置 | Loss | 训练集  (2652) | 验证集  (441) | 测试集  (273网络图) | 测试集  (594教室图) |
| 默认配置：（800\*1333）  P3->P7 ，Scale：3  Ratios:（1.0，2.0，0.5） | 0.51🡪0.05 | 0.9164 | 0.9211 | 0.1256\*2 | 0.8077 |
| 调整anchors：（890\*1582）  P2->P7，Scale：3  Ratios:（1.0, 2.0, 0.5） | 0.61🡪0.08 | 0.9295 | 0.9316 | 0.1161\*2 | 0.8611 |
| 调整anchors：（900\*1600）  P2->P7，Scale：4  Ratios:（1.0, 2.0, 3.0，0.5，0.4，0.3）  FPN.COARSEST\_STRIDE为64； | 0.61🡪0.07 | 0.8155 | 0.8072 | 0.2833\*2 | 0.7089 |
| 调整anchors：（900\*1600）  P2->P6，Scale：4  Ratios:（1.0, 2.0, 3.0，0.5，0.4，0.3） | 0.68🡪0.02 | 0.8003 | 0.7935 | 0.3721\*2 | 0.6792 |

SSD的实验中，如果训练图片中没有网络图片，对网络图片的检测率=**0.24707\*2，**没有retainaNet泛化能力好。

#### retinanet\_X-101-64x4d-FPN

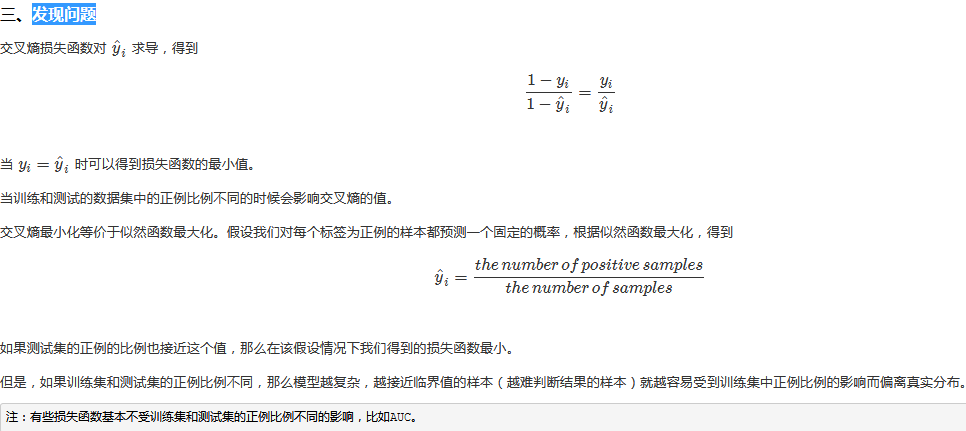
|  |  |  |
| --- | --- | --- |
| 网络 |  |  |
|  | 输入：data |  |
| conv1 |  | 7\*7,p=3,s=2,num\_output=64, bias\_term=false |
| "BatchNorm"+"Scale"+ "ReLU" |  | "Scale"中bias\_term: true |
| pool1 |  | 3\*3,s=2,Max |
|  | 输出：pool1 |  |
|  | **输入：pool1** |  |
| res2a\_branch1 |  | 1\*1,p=0,s=1,num\_output=256,bias\_term: false |
| "BatchNorm"+"Scale" |  | "Scale"中bias\_term: true |
|  | 输出：res2a\_branch1 |  |
|  | **输入：pool1** |  |
| res2a\_branch2a |  | 1\*1,p=0,s=1,num\_output=64,bias\_term: false |
| "BatchNorm"+"Scale"+ "ReLU" |  | "Scale"中bias\_term: true |
|  | 输出：res2a\_branch2a |  |
|  | 输入：res2a\_branch2a |  |
| res2a\_branch2b |  | 3\*3,p=1,s=1,num\_output=64, bias\_term: false |
| "BatchNorm"+"Scale"+ "ReLU" |  | "Scale"中bias\_term: true |
|  | 输出：res2a\_branch2b |  |
|  | 输入：res2a\_branch2b |  |
| res2a\_branch2c |  | 1\*1,p=0,s=1,num\_output=256, bias\_term: false |
| "BatchNorm"+"Scale" |  | "Scale"中bias\_term: true |
|  | 输出：res2a\_branch2c |  |
|  | 输入：res2a\_branch2c  res2a\_branch1 |  |
| "Eltwise"+“ReLU” |  |  |
|  | 输出: res2a |  |
|  | 输入：res2a |  |
| res2b\_branch2a |  | 1\*1,p=0,s=1,64, bias\_term: false |
| "BatchNorm"+"Scale"+ "ReLU" |  |  |
|  | 输出：res2b\_branch2a |  |
|  | 输入：res2b\_branch2a |  |
| res2b\_branch2b |  | 3\*3,p=1,s=1,64, bias\_term: false |
| "BatchNorm"+"Scale"+ "ReLU" |  |  |
|  | 输出：res2b\_branch2b |  |
|  | 输入：res2b\_branch2b |  |
| res2b\_branch2c |  | 1\*1,p=0,s=1,256, bias\_term: false |
| "BatchNorm"+"Scale" |  |  |
|  | 输出：res2b\_branch2c |  |
|  | 输入：res2b\_branch2c  res2a |  |
| "Eltwise"+“ReLU” | 输出：res2b |  |
|  | 输入：res2b |  |
| res2c\_branch2a |  | 1\*1,p=0,s=1,64, bias\_term: false |
| "BatchNorm"+"Scale"+ "ReLU" |  |  |
|  | 输出：res2c\_branch2a |  |
|  | 输入：res2c\_branch2a |  |
| res2c\_branch2b |  | 3\*3,p=1,s=1,64, bias\_term: false |
| "BatchNorm"+"Scale"+ "ReLU" |  |  |
|  | 输出：res2c\_branch2b |  |
|  | 输入：res2c\_branch2b |  |
| res2c\_branch2c |  | 1\*1,p=0,s=1,256, bias\_term: false |
| "BatchNorm"+"Scale" |  |  |
|  | 输出：res2c\_branch2c |  |
|  | 输入：res2c\_branch2c  res2b |  |
| "Eltwise"+“ReLU” | 输出：res2c |  |
|  | 输入：res2c |  |
| res3a\_branch1 |  | 1\*1,p=0,s=2,512, |
| "BatchNorm"+"Scale" |  |  |
|  | 输出：res3a\_branch1 |  |
|  | 输入：res2c |  |
| res3a\_branch2a |  | 1\*1,p=0,s=2,128, |
| "BatchNorm"+"Scale"+ "ReLU" |  |  |
|  | 输出：res3a\_branch2a |  |
|  | 输入：res3a\_branch2a |  |
| res3a\_branch2b |  | 3\*3,p=1,s=1,128, |
| "BatchNorm"+"Scale"+ "ReLU" |  |  |
|  | 输出：res3a\_branch2b |  |
|  | 输入：res3a\_branch2b |  |
| res3a\_branch2c |  | 1\*1,p=0,s=1,512, |
| "BatchNorm"+"Scale" |  |  |
|  | 输出：res3a\_branch2c |  |
|  | 输入：res3a\_branch2c  res3a\_branch1 |  |
| "Eltwise"+“ReLU” | 输出：res3a |  |
|  | 输入：res3a |  |
| res3b1\_branch2a |  | 1\*1,p=0,s=1,128, |
| "BatchNorm"+"Scale"+ "ReLU" |  |  |
|  | 输出：res3b1\_branch2a |  |
|  | 输入：res3b1\_branch2a |  |
| res3b1\_branch2b |  | 3\*3,p=1,s=1,128 |
| "BatchNorm"+"Scale"+ "ReLU" |  |  |
|  | 输出：res3b1\_branch2b |  |
|  | 输入：res3b1\_branch2b |  |
| res3b1\_branch2c |  | 1\*1,p=0,s=1,512, |
| "BatchNorm"+"Scale" |  |  |
|  | 输出：res3b1\_branch2c |  |
|  | 输入：res3b1\_branch2c  res3a |  |
| "Eltwise"+“ReLU” | 输出：res3b1 |  |
|  |  |  |

#### 知识备注：

**交叉熵**（cross entropy）是**用来评估当前训练得到的概率分布与真实分布的差异情况**，减少交叉熵损失就是在提高模型的预测准确率。其离散函数形式H(p,q)=−∑p(x)logq(x)

C:\Users\zhangjing1\AppData\Roaming\JunDaoIM\tempImages\image_Q2.png

y为真实的



**参考文献**： <https://www.cnblogs.com/viredery/p/cross_entropy.html>

<https://blog.csdn.net/u014380165/article/details/77019084>

# 代码调试

## 配置文件

yml定义了很多配置。如模型名称retinanet，类别数81（coco数据集的类别），基础网络FPN.add\_fpn\_ResNet101\_conv5\_body。。。。。

另外，关注一下RETINANET中的参数：

LOSS\_GAMMA和LOSS\_AIPHA是计算损失函数的参数；LOSS\_GAMMA越大，损失函数中对分类好的样本的比重就越低，困难样本的比重就越高。

ASPECT\_RATIOS是anchors的ratio比例；SCALES\_PER\_OCTAVE是anchors的scale的个数。最终anchors的个数就是这二者之积。

NUM\_CONVS是FPN每一层需要先经过多少个卷积层后再连接最后的分类预测和回归位置预测的卷积层。论文中为4。

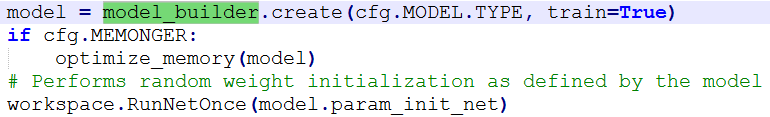
RETINANET\_ON为true，这个参数决定tools/test\_net.py中调用的child\_func和parent\_func。

在Detectron\lib\core\config.py中的RETINANET\_ON默认为false，如下图：



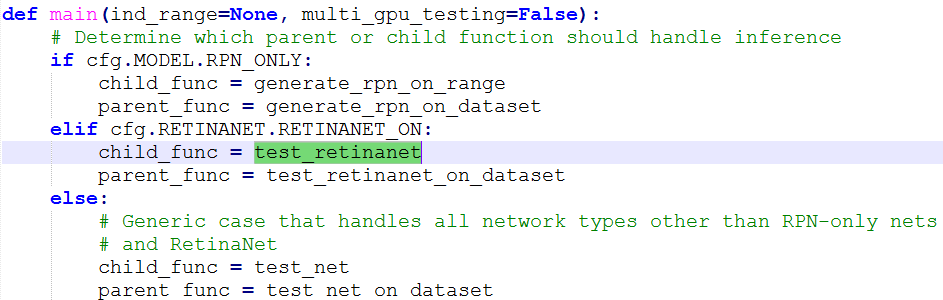
## 测试和训练的模型创建入口

tools\train\_net.py最终使用create函数创建模型：



即model\_builder.create函数创建我们的模型。

tools\test\_net.py根据RETINANET\_ON为true，从而调用test\_retinanet函数



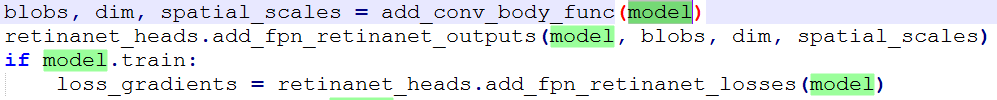
test\_retinanet函数最终也是调用model\_builder.create函数创建我们的模型。

## model\_builder.create创建模型

在lib\modeling\model\_builder.py中的create函数会根据前面yml中MODEL的TYPE调用相应的函数。

当yml中MODEL的TYPE设置为retinanet时，就会调用model\_builder.py中的retinanet函数，该函数会进一步调用build\_generic\_retinanet\_model函数，并传入yml中MODEL的CONV\_BODY参数，即FPN.add\_fpn\_ResNet101\_conv5\_body。

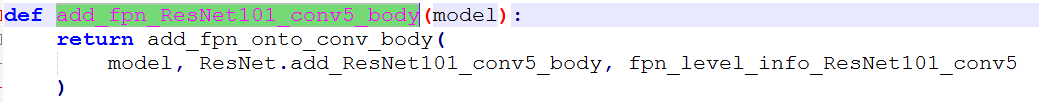
下面是build\_generic\_retinanet\_model函数的内容，add\_conv\_body\_func实际就是前面传入的FPN.add\_fpn\_ResNet101\_conv5\_body。



（1） add\_conv\_body\_fun函数

**add\_conv\_body\_func，即FPN.add\_fpn\_ResNet101\_conv5\_body**

lib\modeling\FPN.py中add\_fpn\_ResNet101\_conv5\_body函数如下：

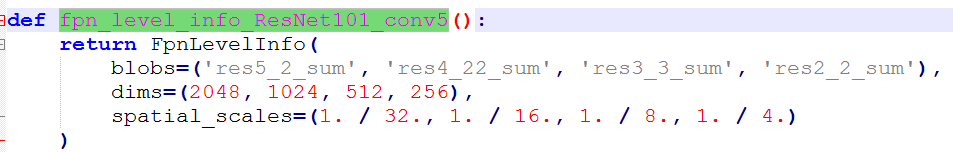


add\_fpn\_onto\_conv\_body函数的流程如下：

先创建ResNet.add\_ResNet101\_conv5\_body的网络结构；

再使用add\_fpn函数创建fpn\_level\_info\_ResNet101\_conv5的网络结构；在 lib\modeling\FPN.py中使用add\_fpn函数创建FPN结构。

fpn\_level\_info\_ResNet101\_conv5中设置了在哪些层上创建特征金字塔结构，如下图：



res5\_2\_sum, res4\_22\_sum, res3\_3\_sum, res2\_2\_sum分别是4大层中每个大层的最后一个residual单元的输出。

（2）add\_fpn\_retinanet\_outputs函数

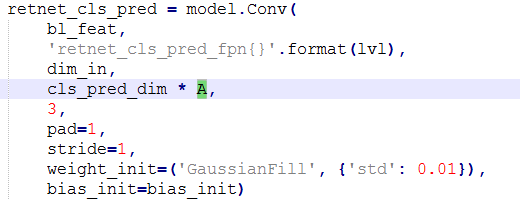
在lib\modeling\retinanet\_heads.py中定义了add\_fpn\_retinanet\_outputs函数。

add\_fpn\_retinanet\_outputs函数主要是在fpn\_ResNet101\_conv5\_body的基础上增加类别预测分支，和位置预测分支。

fpn\_ResNet101\_conv5\_body中4层特征图。其中每一层都一个类别预测分支，和位置预测分支。

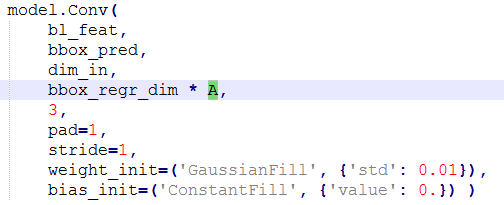
每个类别预测的卷积层中输出卷积核的个数为anchors的个数\*预测的类别数，即为代码中的A\*cls\_pred\_dim。

**最后预测分类的卷积层如下：（**输出卷积核的个数为A\*cls\_pred\_dim）



每个位置预测的卷积层中输出卷积核的个数为anchors的个数\*预测的位置坐标，即为代码中的A\* bbox\_regr\_dim，前面已经指出这里bbox\_regr\_dim是忽略类别的回归，即为4。

**最后预测回归类别的卷积层如下：（**输出卷积核的个数为A\* bbox\_regr\_dim）



（3）add\_fpn\_retinanet\_losses 函数

lib\modeling\retinanet\_heads.py中定义了add\_fpn\_retinanet\_losses函数。

如果是训练，还会调用add\_fpn\_retinanet\_losses来计算损失函数。

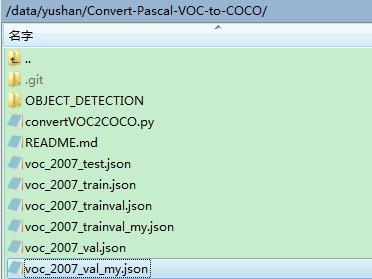
这里的loss分为两部分，一部分是位置预测的loss，一部分是类别预测的loss。其中类别预测的loss计算中使用了focal loss。其中类别预测的loss会使用配置文件中的cfg.RETINANET.LOSS\_GAM和 cfg.RETINANET.LOSS\_ALPHA参数。

# 训练过程

### 将自己数据中的xml转换成Json

使用convertVOC2COCO.py脚本（**自己写的**）将xml文件转化为json文件。

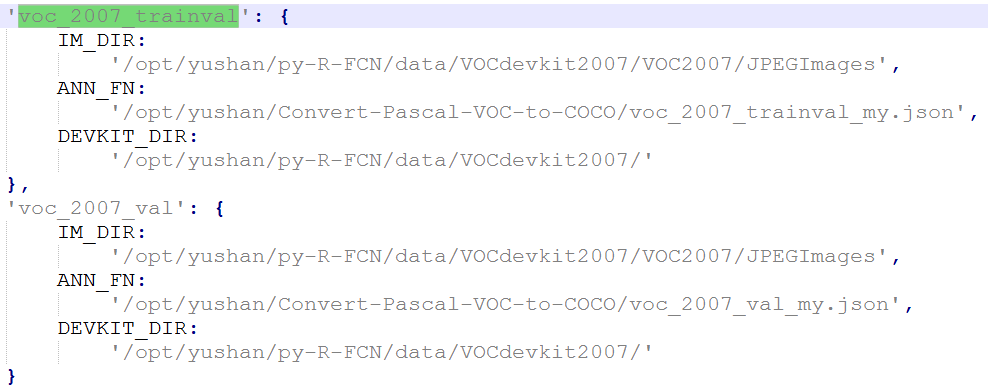
最后生成voc\_2007\_trainval\_my.json和voc\_2007\_test\_my.json，如下图：



### 指定自己的数据的位置

这里使用Detectron\configs\12\_2017\_baselines\retinanet\_R-50-FPN\_1x.yaml。

该yaml 文件中的TRAIN的**DATASETS**配置为voc\_2007\_trainval和voc\_2007\_val。因此只需要修改lib\datasets\dataset\_catalog.py中DATASETS的voc\_2007\_trainval和voc\_2007\_val的值，如下图：



### 修改yaml中的相关配置

上面已经说过这里仍使用Detectron\configs\12\_2017\_baselines\retinanet\_R-50-FPN\_1x.yaml。

1. yaml文件中的TRAIN的DATASETS配置为voc\_2007\_trainval
2. yaml文件中的类别数（这里有4类目标，加上背景一共5类）
3. yaml文件中的GPU的个数

原始yaml文件中的NUM\_GPUS为8。

由于我的环境中只有2个GPU，因此设置NUM\_GPUS为2。当然也可以设置NUM\_GPUS为1，这样就会只会使用2个GPU中的一个，一般默认使用GPU0。

### 训练脚本

训练脚本：

python tools/train\_net.py

--cfg configs/12\_2017\_baselines/retinanet\_R-50-FPN\_1x.yaml

OUTPUT\_DIR output

这里OUTPUT\_DIR是保存训练好的模型的文件夹。如果没有这个文件夹，会自动在当前目录下创建。最终会根据yaml文件中的TRAIN的DATASETS等其他配置自动生成的模型的完整保存路径。

# 使用训练好的模型测试

**测试AP的脚本如下**：

python tools/test\_net.py \

--cfg configs/12\_2017\_baselines/ retinanet\_R-50-FPN\_1x.yaml \

TEST.WEIGHTS output/train/voc\_2007\_trainval:voc\_2007\_val/retinanet/model\_iter89999.pkl \

NUM\_GPUS 1

--cfg指定的yaml文件中配置了网络的名称和结构

TEST.WEIGHTS是模型的权重文件

NUM\_GPUS是使用GPU的个数