**V1:** CVPR2016 目标检测论文[YOLO: Unified, Real-Time Object Detection](http://link.zhihu.com/?target=https%3A//arxiv.org/abs/1506.02640)

**Yolo: you only look once**

**V2:** [YOLO9000: Better, Faster, Stronger](https://arxiv.org/abs/1612.08242)

**V3:** 2018年，YOLOv3: An Incremental Improvement

<https://pjreddie.com/media/files/papers/YOLOv3.pdf>

**Yolo**采用**Darknet**: Open Source Neural Networks in C C语言版本的开源神经网络。

**github**: <https://github.com/pjreddie/darknet>

**主页：**<https://pjreddie.com/darknet/yolo/>

所以这发展的历程应该是这样的：

**yolo—SSD—yolov2—FPN、Focal loss（RetainaNet）、DSSD......—yolov3**

**备注：**实时为1s处理24张图片、一般imageNet数据集训练好的模型需要将所有图像处理100倍。bacthsize越大，学习步长也越大。

# 环境搭建

## 步骤

git clone https://github.com/pjreddie/darknet

cd darknet

make

wget <https://pjreddie.com/media/files/yolov3.weights> （下载yolov3权重）

wget <https://pjreddie.com/media/files/darknet53.conv.74> (下载预训练模型)

### Error

1. 修改makefile文件，make –j16， （make: Nothing to be done for `all'. 这句提示是说明你已经编译好了，而且没有对代码进行任何改动。先make clean，再make）

2.内存不足：

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

## 训练

**参考文献：** <https://blog.csdn.net/vvyuervv/article/details/77160262>

### 编译GPU版本的环境

如果已经安装有cuda和cudnn，先修改Makefile文件的第一行和第二行，0改为1，

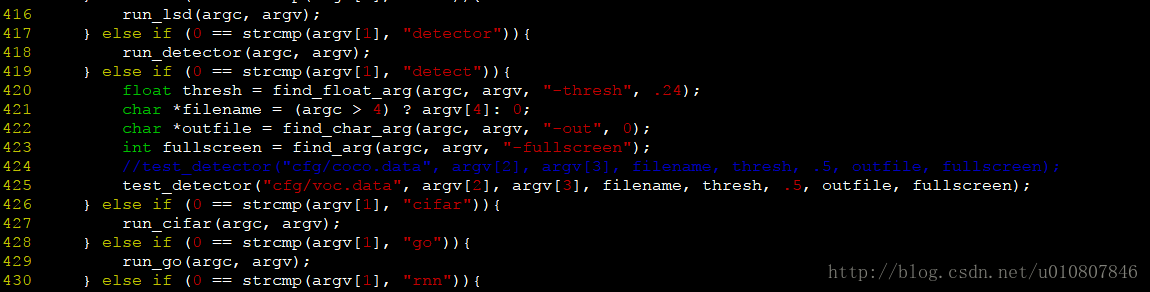
执行make，得到的darknet执行文件支持cuda和cudnn。

### 修改：

#### 1、修改example/darknet.c文件

修改example/darknet.c文件的第424行，将“cfg/coco.data”改为“cfg/voc.data”，如下，

这一步如果不修改，最后执行darknet命令去检测的时候，得到的BoundingBox显示的类名会跟实际检测到的物体“风马牛不相及”。



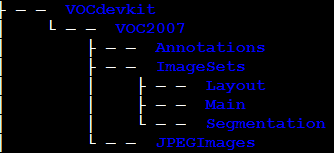
### 准备数据集：VOC数据集格式

#### 步骤1：安装VOC数据集格式存放图片、xml，并写脚本生成只有图片名称的txt

因为要用到YOLO自带的**voc\_label.py**文件，其位于darknet/scripts目录下。功能一：把标注工具得到的xml文件转为label.txt；功能二：得到train.txt。

voc\_label.py会沿着VOC数据集文件层级结构的文件名称去寻找，所以一定要拼写正确。

**如果用自己的命名方式，需要修改voc\_label.py文件里的路径和文件名称**。



Annotations ---- 这个文件夹是放所有xml描述文件的。

JPEGImages ---- 这个文件夹是放所有jpg图片文件的。

ImageSets -> Main ---- 这个文件里稍后会生成一个名为train.txt的文档，train.txt文档内容是所有训练集图片的**名字**，**没有后缀名。**

**train.txt内容如下：**



#### 步骤2：使用scripts/voc\_label.py生成图片名对于的txt、图片全路径的txt（data/2007\_train.txt）

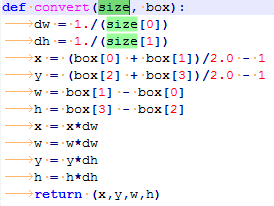
执行python scripts/voc\_label.py，会根据train.txt中的图片文件名，生成对应的txt，保存在与Annotations 同级目录的labels目录中。比如图片、xml的文件名0001，生成labels/0001.txt，0001.txt的内容如下：

C:\Users\zhangjing1\AppData\Roaming\JunDaoIM\tempImages\image_c5.png

classes = ["autotruck", "crane", "digger", "mixerTruck", "forklift", "colorPlate", "pit", "bricksPile", "mound", "worker", "car"]

8:是类别的索引，从0开始；后面4个是该类对应的bbox转换后的(x,y,w,h)。

**转换函数为：**



size=(image\_w, image\_h), box=(xmin, xmax, ymin, ymax)

### 修改配置文件

#### cfg/ yolov3-voc.cfg

**备注：**yolov3-voc.cfg与yolov3.cfg的区别就是类别个数不同（20类、80类），对应的filters个数也不同（filters=75=（3\*（20+5））、filters=255=（3\*（80+5）））

##### 修改1：batch

**# Training**

**batch=64** #每batch个样本更新一次参数。

**subdivisions=8**  #如果内存不够大，将batch分割为subdivisions个子batch，每个

子batch的大小为batch/subdivisions。在darknet代码中，会将

batch/subdivisions命名为batch。

##### 修改2：类别个数

[yolo]

**classes=11** #类别个数，不含背景

**备注：**对于yolov3有个层输出，所以需要改3处。

##### 修改3：filters个数

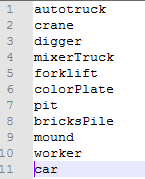
[convolutional]

**filters=48 #**(classes+ coords+ 1)\* (NUM)=（11+4+1）\*3

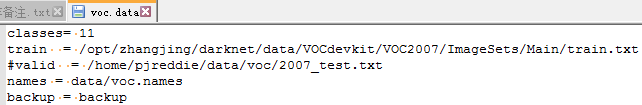
**备注：**只改输出的filters个数，也就是说最后一层[convolutional]的filters个数，对于yolov3有个层输出，所以需要改3处。

#### data/voc.names：类别名称

11类的类别名称如下：



#### cfg/voc.data



**备注：**修改classes=11、valid注释掉即可、train.txt中图片路径要写成绝对路径(比如：

/opt/zhangjing/darknet/data/VOCdevkit/VOC2007/JPEGImages/0232.jpg)

#### 保存模型

**路径：**模型存放的位置为cfg/voc.data中的参数backup，路径为相对路径；

**间隔多久保存一次模型:** 设置需要在源码examples/detector.c中修改，if(i%10000==0 || (i <

1000 && i%100 == 0))，意思是1000次迭代（cfg/yolov3-voc.cfg文件中batch为一次迭代）之前每100保存一次模型，1000次迭代以上每10000次保存一次模型。

**模型名称**：例如yolov3-voc\_200.weights，yolov3-voc是cfg/yolov3-voc.cfg的名称、200是迭

代次数。

### 训练：

**./darknet detector train cfg/voc.data cfg/yolov3-voc.cfg -gpus 0,1,2,3**

#**用官网的预训练模型 darknet53.conv.74（大小约为76 MB）做初始化，**

<https://pjreddie.com/media/files/darknet53.conv.74>

**./darknet detector train cfg/voc.data cfg/yolov3-voc.cfg darknet53.conv.74 -gpus 0**

**>backup/ train.log**

**#如果训练中断，执行下述命令，继续训练**

**./darknet detector train cfg/voc.data cfg/yolov3-voc.cfg backup/final.weights -gpus 0,1,2,3**

**./darknet detector train cfg/voc.data cfg/yolov3-voc.cfg backup/**yolov3-voc\_10000.weights**-gpus 0**

**备注：**模型保存在backup，命名为final.weights；

输入图片大小会被随机缩放到不同尺度进行训练。

#### 训练输出

**参考文献**：<https://timebutt.github.io/static/understanding-yolov2-training-output/>

<https://www.zhihu.com/question/58374346/answer/164178426>

<https://blog.csdn.net/renhanchi/article/details/71077830> （很多参数）

<http://www.mamicode.com/info-detail-1974310.html> （部分代码解释）

##### 每subdivision的输出：

Region 106 Avg IOU: 0.814567, Class: 0.937782, Obj: 0.528317, No Obj: 0.000056, .5R: 1.000000, .75R: 1.000000, count: 1

**备注：**每行为当前subdivision的处理结果，batch=64、subdivision=16，所以会打印16条类似上面结果。

**Region Avg IOU：** 这个是预测出的bbox和实际标注的bbox的交集 除以 他们的并集。显

然，这个数值越大，说明预测的结果越好。

**Class: =** avg\_cat/class\_count, 平均类别辨识率；

**Avg Recall：** 这个表示平均召回率， 意思是  检测出物体的个数 除以 标注的所有物体个数。

**.5R、.75R:** IOU为0.5、0.75的召回率

**Obj**： True Pos的平均置信度；（正类预测为正样本，TP）

**No Obj**：False Pos 的平均置信度；（负类预测为正样本，TP）（**准确率=P=TP/（TP+FP）**）

**count：** 标注的所有物体的个数。 如果 count = 6， recall = 0.66667， 就是表示一共有6

个物体（可能包含不同类别，这个不管类别），然后我预测出来了4个，所以Recall 就是 4 除以 6 = 0.66667。

**代码：src/region\_layer.c**

printf("Region Avg IOU: %f, Class: %f, Obj: %f, No Obj: %f, Avg Recall: %f, count: %d\n", avg\_iou/count, avg\_cat/class\_count, avg\_obj/count, avg\_anyobj/(l.w\*l.h\*l.n\*l.batch), recall/count, count);

}

训练log中各参数的意义 ：

**Region Avg IOU**：平均的IOU，代表预测的bounding box和ground truth的交集与并集之比，期望该值趋近于1。

**Class:**是标注物体的概率，期望该值趋近于1.   
**Obj：**期望该值趋近于1.   
**No Obj**:期望该值越来越小但不为零.   
**Avg Recall：**期望该值趋近1   
**avg**：平均损失，期望该值趋近于0

##### 每一小节的输出：

**1212: 1.665630, 1.398193 avg, 0.001000 rate, 8.464698 seconds, 77568 images**

iteration次数、train loss、avg train loss、学习率、一个batch的处理时间、已经一共处理了多少张图片。

重点关注 train loss 和avg train loss，这两个值应该是随着iteration增加而逐渐降低的。如果loss增大到几百那就是训练发散了，如果loss在一段时间不变，就需要降低learning rate或者改变batch来加强学习效果。当然也可能是训练已经充分。这个需要自己判断。

#### 调优实验

##### 1.改变图像大小

改训练图片的输入大小，从416\*416改成608\*608。（本身训练就是间隔一段时间将将输入图像resize到一个尺度）

##### 2.改学习率

在yolo训练中，网络训练160epoches，初始学习率为0.001，在60和90epochs时将学习率除以10。

#### 实验

##### 实验1

**存放位置（100）：**/home/zhangjing/darknet/yolov3-voc.cfg

**容器：**test\_zj (nvidia-docker run -idt -v /data:/opt --name test\_zj caffe\_ssd)

**镜像：**caffe\_ssd

**训练图片大小**：416\*416 (多尺度训练，会每个一定间隔调整输入图片大小)

训练集3406，850验证集(4256,8:2)，349测试集

**类别**：11 [autotruck、crane、digger、mixerTruck、forklift、colorPlate、pit、

bricksPile、mound、worker、car]

**Bachsize**： 4=**batch=64**/**subdivisions=16** #每batch个样本更新一次参数。

在darknet代码中，会将batch/subdivisions命名为batch。

**GPU内存**：6.7G

**模型名称：**backup/yolov3-voc\_900.weights

**实验描述：**原版YOLOv3

**训练：./darknet detector train cfg/voc.data cfg/yolov3-voc.cfg darknet53.conv.74 -gpus 0**

**>backup/ train.log**

**#如果训练中断，执行下述命令，继续训练**

**./darknet detector train cfg/voc.data cfg/yolov3-voc.cfg backup/**yolov3-voc\_10000.weights**-gpus 0**

**生成测试结果txt: ./darknet detector valid cfg/voc\_test.data cfg/yolov3-voc-test.cfg backup/yolov3-**

**voc\_14000.weights -out 608\_14000\_**

**计算mAP:** python scripts/clac\_mAP.py

**时间**： (gpu1、416\*416) 25ms

(gpu1、608\*608) 45ms

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **迭代次数** | **训练集**  **（3406个）**  **416\*416** | **验证集**  **（850个）**  **416\*416** | **测试集**  **（349个）**  **416\*416 25ms** | **测试集**  **（349个）**  **608\*608 45ms** |
| 14000  416\*416 | autotruck : 0.967  crane : 0.888  digger : 0.940  mixerTruck : 0.866  forklift : 0.947  colorPlate : 0.881  pit : 0.618  bricksPile : 0.988  mound : 0.962  worker : 0.934  car : 0.981  **eval= 0.907** | autotruck : 0.828  crane : 0.778  digger : 0.748  mixerTruck : 0.667  forklift : 0.736  colorPlate : 0.614  pit : 0.491  bricksPile : 0.914  mound : 0.707  worker : 0.723  car : 0.937  **eval= 0.740** | autotruck : 0.722  crane : 0.832  digger : 0.796  mixerTruck : 0.734  forklift : 0.815  colorPlate : 0.476  pit : 0.429  bricksPile : 0.867  mound : 0.453  worker : 0.442  car : 0.809  **eval= 0.670** | autotruck : 0.628  crane : 0.704  digger : 0.774  mixerTruck : 0.593  forklift : 0.817  colorPlate : 0.424  pit : 0.329  bricksPile : 0.857  mound : 0.321  worker : 0.507  car : 0.799  **eval= 0.614** |
| 15800 |  |  |  | autotruck : 0.747  crane : 0.750  digger : 0.822  mixerTruck : 0.754  forklift : 0.856  colorPlate : 0.453  pit : 0.309  bricksPile : 0.849  mound : 0.474  worker : 0.454  car : 0.811  **eval= 0.662** |

## 测试

**#**处理一张图片，会保存结果到predictions.png，有检测框及类别信息

**./darknet yolo test cfg/yolov3-voc-test.cfg backup/yolov3-voc\_10000.weights data/0232.jpg (**没有预测结果)

**./darknet detect cfg/yolov3-voc-test.cfg backup/yolov3-voc\_10000.weights data/0232.jpg （**有预测结果，colorPlate: 99%，colorPlate: 93%）

**#**（需要安装opencv，make时OPENCV=1）

**./darknet detector demo cfg/voc.data cfg/ yolov3-voc-test.cfg backup/yolov3-voc\_500.weights**

#Enter Image Path:提示输入图片路径

**./darknet detector test cfg/voc\_test.data cfg/yolov3-voc-test.cfg backup/yolov3-voc\_900.weights  -gpus 0**

### 在测试图片上画框并保存

#### 方法1：

**代码**：examples/yolo.c: test\_yolo(char \*cfgfile, char \*weightfile, char \*filename, float thresh)

save\_image(im, "predictions"); time=clock()统计时间。

**备注**：**./darknet yolo test cfg/yolov3-voc-test.cfg backup/yolov3-voc\_500.weights data/0232.jpg**

原始代码只处理一张图片并保存结果。

#### 方法2：

**./darknet detector valid cfg/voc\_test.data cfg/yolov3-voc-test.cfg backup/yolov3-voc\_10000.weights -out 1000\_ -thresh .5** 生成每个类别检测结果的txt；再自己写代码将检测结果画处理。

**备注：**-thresh .5参数好像不起作用，因为生成的txt中还有置信度小于该参数的。

**./darknet yolo valid cfg/voc\_test.data cfg/yolov3-voc-test.cfg backup/yolov3-voc\_10000.weights -out 1000\_**

**Error:** Segmentation fault (core dumped)

**解决办法：**取掉上面的**cfg/voc\_test.data**参数，就不会报错了，但是结果是生成voc数据集中类别对应的txt.

## 评估模型：

### 召回率

**./darknet detector recall cfg/voc\_test.data cfg/yolov3-voc-test.cfg backup/yolov3-voc\_10000.weights -thresh .5**

**Error:** Couldn't open file: data/coco\_val\_5k.list

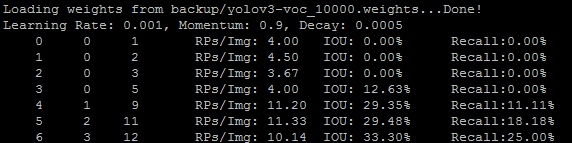
**解决办法：**修改examples/detector.c 中validate\_detector\_recall(char \*cfgfile, char \*weightfile)

中的**get\_paths路径**；

**备注**：detector recall调用examples/detector.c 中validate\_detector\_recall函数，输出结果为

每张图对应的IOU、recall等信息。

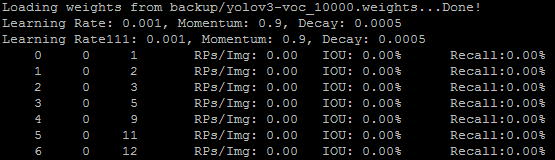
**cfg/voc\_test.data 参数还必须有，不然会报Segmentation fault (core dumped)；但是处理的图片路径还是代码中get\_paths路径起作用。**



**./darknet yolo recall cfg/voc\_test.data cfg/yolov3-voc-test.cfg backup/yolov3-voc\_10000.weights** -thresh .5

**Error:** Segmentation fault (core dumped)

**解决办法：**取掉上面的**cfg/voc\_test.data**参数，就不会报错了，但是结果不对，如下图所示：

****

观察avg（loss）这个值的变化，如果这个值基本不再变小，那么训练就可以停止了。

根据输出IOU和Recall值，找到最好的weight文件。一般IOU越大越好，Recall也是越大越好。

### 生成检测结果文件:每个类对应一个txt

**./darknet detector valid cfg/voc\_test.data cfg/yolov3-voc-test.cfg backup/yolov3-voc\_10000.weights -out 1000\_ -thresh .5**

**代码：**examples/yolo.c/validate\_yolo(char \*cfg, char \*weights)

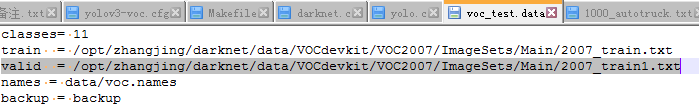
**保存的位置及名称：**results/1000\_autotruck.txt

base+1000\_类名, char \*base = "results/ ";

**352**

**Total Detection Time: 9.713208 Seconds**

**处理的数据：**valid对应的txt

****

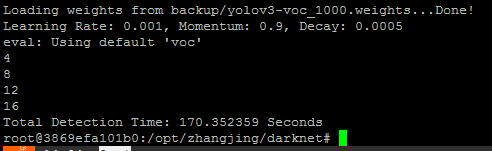
**txt的内容如下：**results/1000\_autotruck.txt

**C:\Users\zhangjing1\AppData\Roaming\JunDaoIM\tempImages\image_La.png**

**备注：**图片名称、该类别的置信度、xmin、ymin、xmax、ymax，参数**-thresh .5不起作用。**

所以需要自己写代码将上图中的检测结果画到对应的图上。

**输出结果：**

****

**备注：**统计时间**time(0)**

### mAP

**参考文献**：

<https://blog.csdn.net/lucky__ing/article/details/78310510?utm_source=debugrun&utm_medium=referral>

目标检测中衡量识别精度的指标是mAP（mean average precision）。多个类别物体检测中，每一个类别都可以根据recall和precision绘制一条曲线，AP就是该曲线下的面积，mAP是多个类别AP的平均值。

**voc\_eval.py:**  <https://github.com/lucky-ing/voc_eval>

**执行命令：**

python 123.py /opt/zhangjing/darknet/results/14000\_bricksPile.txt /opt/zhangjing/darknet/data/VOCdevkit/2007\_benchmark\_349.txt bricksPile

**备注：**10000\_bricksPile.txt是通过命令**detector valid生成的；**

2007\_benchmark\_349.txt是生成10000\_bricksPile.txt的原始图片绝对路径；

bricksPile是类名。

必须将所有图片、xml放到一个文件夹A，并且2007\_benchmark\_349.txt中图片路径是A文件夹的绝对路径，该脚本只处理一个类别。

把car.txt 用faster rcnn 中voc\_eval计算mAP

/home/sam/src/caffeup2date\_pyfasterrcnn/lib/datasets/compute\_mAP.py

from voc\_eval import voc\_eval

print voc\_eval('/home/sam/src/darknet/results/{}.txt', '/home/sam/datasets/car2/VOC2007/Annotations/{}.xml', '/home/sam/datasets/car2/VOC2007/ImageSets/Main/test.txt', 'car', '.')

第三个结果就是

如果只想计算大于0.3的输出结果的mAP,把 voc\_eval.py文件中如下代码更改

sorted\_ind = np.argsort(-confidence)  
sorted\_ind1 = np.where(confidence[sorted\_ind] >= .3)[0]#np.argsort(-confidence<=-.3)  
sorted\_ind = sorted\_ind[sorted\_ind1]

**参考文献**：<https://blog.csdn.net/renhanchi/article/details/71077830>

<https://blog.csdn.net/northeastsqure/article/details/54848665>

### 备注

#### 对比1：

Examples/detector.c 中：

**validate\_detector\_recall**(char \*cfgfile, char \*weightfile)

**validate\_detector**(char \*datacfg, char \*cfgfile, char \*weightfile, char \*outfile)

根据上面2个函数参数对比可知，为何**detector valid**命令的路径使用的是cfg/voc\_test.data中valid的；而**detector recall**命令的路径需要写代码中修改。

# 参数说明：cfg/yolov1.cfg

**参考文献：**<https://blog.csdn.net/Jinlong_Xu/article/details/76375334>

## [net]

# Testing

batch=1

subdivisions=1

# Training

# **batch=64** #每batch个样本更新一次参数。

# **subdivisions=8**  #如果内存不够大，将batch分割为subdivisions个子batch，每个

子batch的大小为batch/subdivisions。在darknet代码中，会将

batch/subdivisions命名为batch。

height=448 # input图像的高

width=448

channels=3

momentum=0.9 #动量

decay=0.0005 #权重衰减正则项，防止过拟合

saturation=1.5 #通过调整饱和度来生成更多训练样本

exposure = 1.5 #通过调整曝光量来生成更多训练样本

hue=.1 #通过调整色调来生成更多训练样本

**learning\_rate=0.0005** #初始学习率

policy=steps #调整学习率的policy，有如下policy：CONSTANT, STEP, EXP,

POLY, STEPS, SIG, RANDOM

steps=200,400,600,20000,30000 #根据batch\_num调整学习率

scales=2.5,2,2,.1,.1 #学习率变化的比例，累计相乘

**max\_batches = 40000** #训练达到max\_batches后停止学习

## [convolutional]

**batch\_normalize=1** 是否做BN

filters=32 输出多少个特征图

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

**activation=leaky** 激活函数：logistic，loggy，relu，elu，

relie，plse，hardtan，lhtan，linear，ramp，

leaky，tanh，stair

## [maxpool]

size=2 池化层尺寸

stride=2 池化步进

## [detection]:v1

classes=20 类别数

coords=4 每个box的4个坐标tx,ty,tw,th

**num=2** （v1:2，v2:5,v3:9） 每个grid cell预测几个box

**side=7** 分成7\*7个网格

softmax=0 是否使用softmax做激活函数

jitter=.2 通过抖动增加噪声来抑制过拟合

rescore=1 暂理解为一个开关，非0时通过重打分来调整

sqrt=1

#优化loss的参数，**v3没有这些参数**

object\_scale=1 暂理解为计算损失时预测框中有物体时的权重

noobject\_scale=0.5 暂理解为计算损失时预测框中无物体时的权重

class\_scale=1 暂理解为计算类别损失时的权重

coord\_scale=5 暂理解为计算损失时坐标偏差的权重

## [region]:v2

**anchors** = 0.57273, 0.677385, 1.87446, 2.06253, 3.33843, 5.47434, 7.88282,

3.52778, 9.77052, 9.16828

bias\_match=1

classes=80

coords=4

num=5

**softmax=1**

jitter=.3

rescore=1

object\_scale=5

noobject\_scale=1

class\_scale=1

coord\_scale=1

absolute=1

thresh = .6

random=1 是否随机确定最后一个预测框

## [yolo]:v3

**mask = 0,1,2**

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

373,326 **（v3）**预选框，可以手工挑选，也可以通过k means 从训练样本中学出

预测框的初始宽高，第一个是w，第二个是h，总数量是num\*2，YOLOv2作者说anchors是使用K-MEANS获得，其实就是计算出哪种类型的框比较多，可以增加收敛速度，如果不设置anchors，默认是0.5;

**classes=80**

**num=9 anchor个数**

**jitter=.3**

**ignore\_thresh = .5**

**truth\_thresh = 1**

**random=1 #**random为1时会启用Multi-Scale Training，随机使用不同尺寸的图片进行训练,如

果为0，每次训练大小与输入大小一致;

# YOLO版本对比

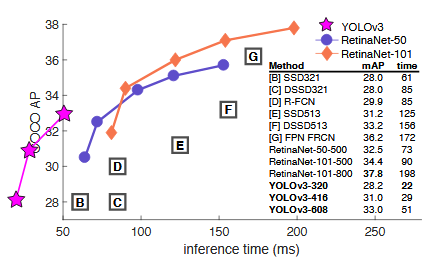
**测试：**

./darknet detect cfg/yolov3.cfg cfg/yolov3.weights data/dog.jpg

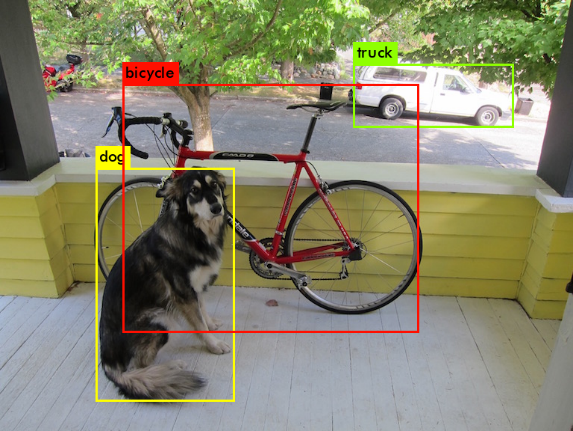
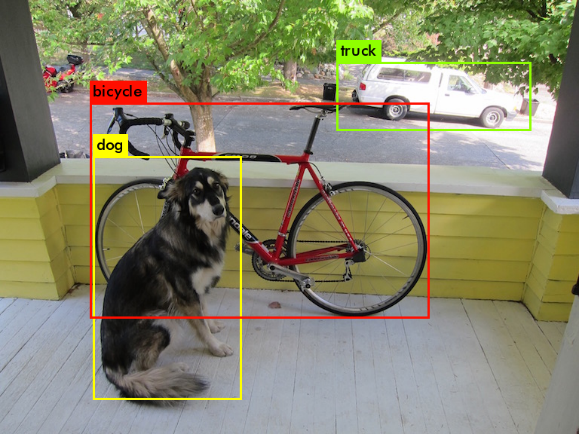
Yolov1测试: ./darknet yolo test cfg/yolov1.cfg cfg/yolov1.weights data/dog.jpg

**结果：**dog.jpg (768\*576h)

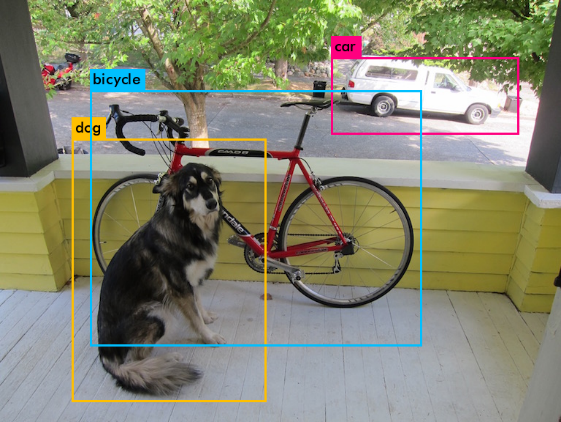
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **版本** | **准确率**  **cpu** | **网络、**VOC 2007 test set | **输入图片** | **输出** | **Loss** |
| **V1** | dog: 26%  car: 74%  bicycle: 39%  6.9s | **yolo:**24个卷积层+2个全连接层  45 fps mAP=63.4%  **fast yolo:**9个卷积+2个全连接层  155fps mAP= 52.7% | 448\*448[**缺陷**](#_缺陷：) | S\*S\*（B\*5+C）=1470维  S=7,B=2,C=20(类别)  5(x,y,w,h,confidence) | 网络输出的S\*S\*(B\*5 + C)维向量与真实的该向量的**均方和误差** |
| **V2** | dog: 81%  truck: 74%  bicycle: 83%  5.43 s | 67FPS下mAP达到76.8%，  40FPS下mAP达到78.6%  **dark\_19:**19个卷积层+5个最大池化层  （卷积3\*3、1\*1组合） | 416\*416 | S\*S\*B\*（5+C）维  S\*S为特征图大小=输入/32  B=5(anchor),C=20(类别)  5(x,y,w,h,confidence)  可检测9000个物种， | **多尺度训练** |
| **V3** | dog: 99%  truck: 92%  bicycle: 99%  10.81 s | **dark\_53:**52个卷积层+1个全连接层+残差模块（3\*3,1\*1）  **yolov3**:AP50=57.9,51ms,TitanX  **retainaNet**:AP50=57.5,198ms,TitanX | 416\*416 | S1\*S1\*B\*（5+C）维  S2\*S2\*B\*（5+C）维  S3\*S3\*B\*（5+C）维  S1\*S1为特征图大小=输入/32  S2\*S2为特征图大小=输入/16  S3\*S3为特征图大小=输入/8  B=3(anchor),C=80(类别)  5(x,y,w,h,confidence) | 分类损失用**binary cross-entropy loss.**  **多尺度训练、多层预测** |



**V3**

**V3结果 V2结果**



**V1结果**

yolo类算法，从开始到现在已经有了3代，我们称之为v1、v2、v3，一路走来，让人能感觉到的是算法的性能在不断的改进，以至于现在成为了开源通用目标检测算法的领头羊（ps：虽然本人一直都很欣赏SSD，但是不得不说V3版本已经达到目前的颠覆）。一直以来，有一个问题困扰许久，**那就是如何检测两个距离很近的同类的物体，当然又或者是距离很近的不同类的物体？**绝大部分算法都会对传入的data做resize到一个更小的resolution，它们对于这种情况都会给出一个目标框，因为在它们的特征提取或者回归过程看来，这就是一个物体（可想本来就很近，一放缩之间的近距离越发明显了），而事实上这是两个相同（或不同）类型的物体靠的很近，这个难题是目标检测和跟踪领域的一个挑战。就好像对小目标的检测，一直以来也被看做是算法的一种评估。但是啊，**v3版本却做到了，它对这种距离很近的物体或者小物体有很好的鲁棒性**。

yolo的v1和v2都不如SSD算法，原谅这么直白，原因是**v1版本的448和v2版本的416都不如SSD的300**，当然以上结论都是实验测的，v3版本的416应该比SSD512好，可见其性能。

对官方yolo做了实验，实验中，采用同一个视频、同一张显卡，在阈值为0.3的前提下，对比了v3和v2的测试效果之后，有了下面两个疑问：

**1.为什么v3和v2版本的测试性能提高很大，但速度却没有降低？**

**2.为什么v3性能上能有这么大的改进？或者说为什么v3在没有提高输入数据分辨率的前**

**提下，对小目标检测变得这么好？**

 要回答上述两个问题，必须要看看作者发布的v3论文了，将v3和v2不一样的地方总结一下：

**loss不同**：作者v3替换了v2的softmax loss 变成logistic loss，而且每个ground truth只匹

配一个先验框。

**anchor bbox prior不同**：v2作者用了5个anchor，一个折衷的选择，所以v3用了9个anchor，

提高了IOU。

**detection的策略不同：**v2只有一个detection，v3一下变成了3个，分别是一个下采样的，

feature map为**13\*13**，还有2个上采样的eltwise sum，feature map为**26\*26，52\*52**，也就是说v3的416版本已经用到了52的feature map，而v2把多尺度考虑到训练的data采样上，最后也只是用到了13的feature map，这应该是对小目标影响最大的地方。

**backbone不同**：这和上一点是有关系的，v2的darknet-19变成了v3的darknet-53，为啥呢？

就是需要上采样啊，卷积层的数量自然就多了，另外作者还是用了**一连串的3\*3、1\*1卷积，3\*3的卷积增加channel，而1\*1的卷积在于压缩3\*3卷积后的特征表示，**这波操作很具有实用性，一增一减，效果棒棒。

# YOLOv3

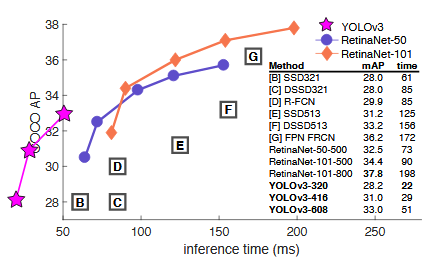
<https://pjreddie.com/media/files/papers/YOLOv3.pdf>

**参考文献：**<https://www.cnblogs.com/makefile/p/YOLOv3.html>

## 速度对比

YOLOv3在Pascal Titan X上处理608x608图像速度达到20FPS，在 COCO test-dev 上 [mAP@0.5](mailto:mAP@0.5) 达到 57.9%，与RetinaNet（FocalLoss论文所提出的单阶段网络）的结果相近，并且速度快4倍.

YOLO v3的模型比之前的模型复杂了不少，可以通过改变模型结构的大小来权衡速度与精度。



## 改进之处：

* 多尺度预测 （类FPN）
* 更好的基础分类网络（类ResNet）和分类器

## **分类器-类别预测**：

YOLOv3不使用Softmax对每个框进行分类，主要考虑因素有两个：

1．Softmax使得每个框分配一个类别（score最大的一个），而对于Open Images这种数据集，**目标可能有重叠的类别标签**，因此Softmax不适用于多标签分类。

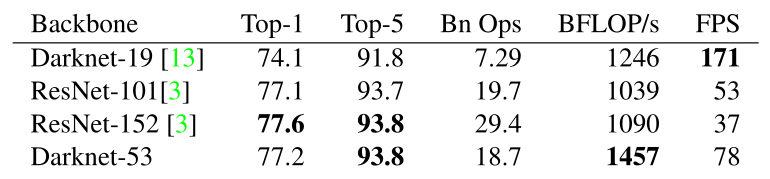
2.Softmax可被独立的多个logistic分类器替代，且准确率不会下降。分类损失采用**binary cross-entropy loss.**

## **多尺度预测**

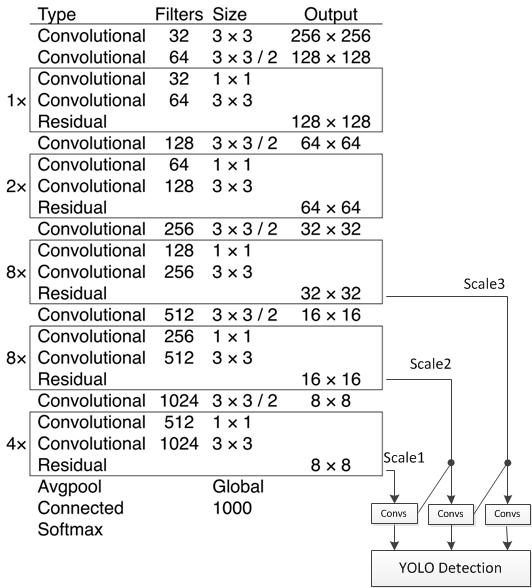
每种尺度预测**3个box**, anchor的设计方式仍然使用聚类,得到9个聚类中心,将其按照大小均分给3个尺度.

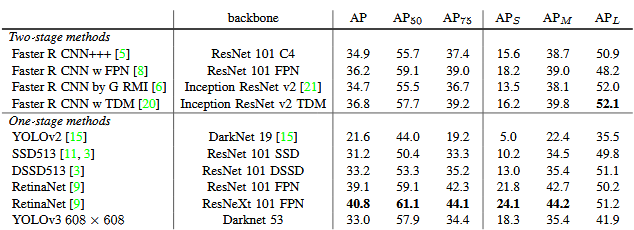
* 尺度1: 在基础网络之后添加一些卷积层再输出box信息.
* 尺度2: 从尺度1中的倒数第二层的卷积层上采样(x2)再与最后一个16x16大小的特征图相加,再次通过多个卷积后输出box信息.相比尺度1变大两倍.
* 尺度3: 与尺度2类似,使用了32x32大小的特征图.

## **基础网络 Darknet-53**

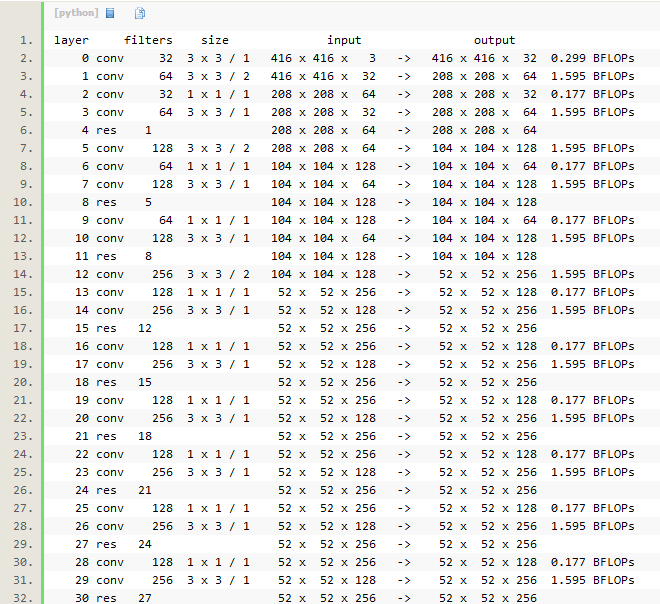
仿ResNet, 与ResNet-101或ResNet-152准确率接近,但速度更快.对比如下:  
****

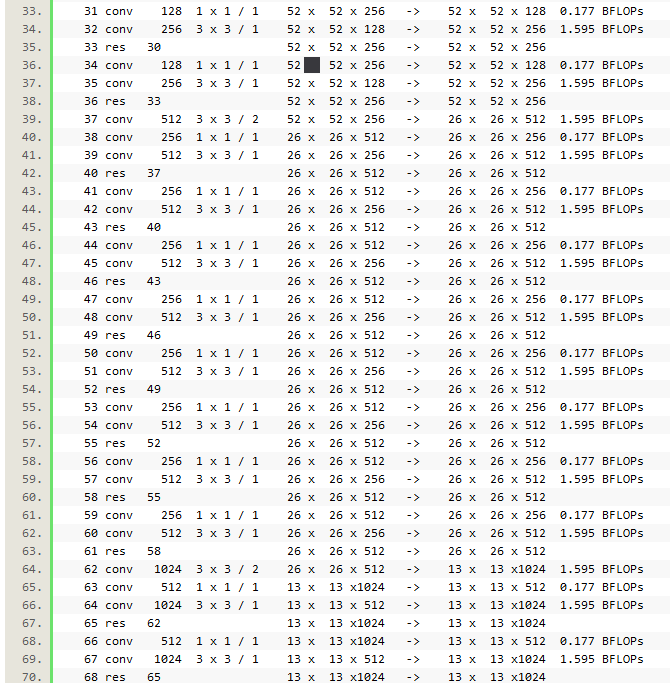
**网络结构如下：**

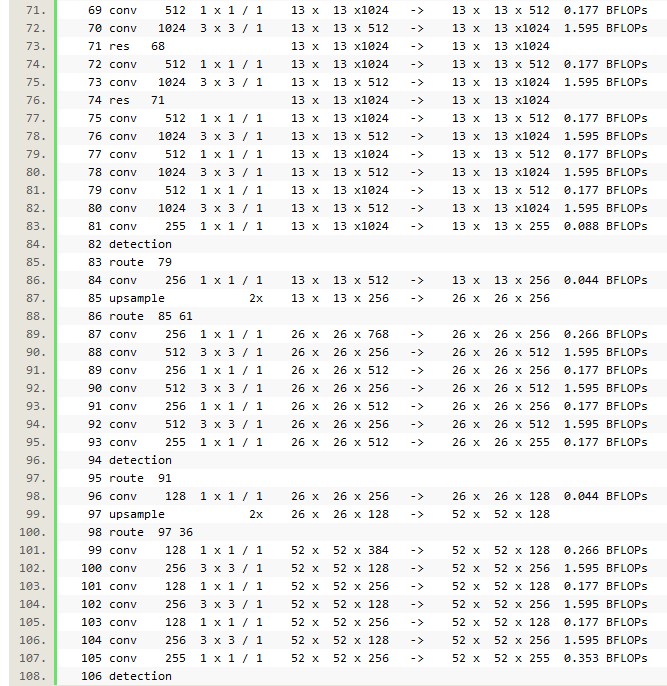
****



YOLOv3在mAP@0.5及小目标AP*S*上具有不错的结果，但随着IOU的增大，性能下降，说明YOLOv3不能很好地与ground truth切合。







## Darknet 框架

Darknet 由 C 语言和 CUDA 实现, 对GPU显存利用效率较高(CPU速度差一些, 通过与SSD的Caffe程序对比发现存在CPU较慢,GPU较快的情况). Darknet 对第三方库的依赖较少,且仅使用了少量GNU linux平台C接口,因此很容易移植到其它平台,如Windows或嵌入式设备.参考[Windows 版 Darknet (YOLOv2) 移植](http://www.cnblogs.com/makefile/p/darknet-win-port.html), [代码在此](https://github.com/makefile/darknet).

**region层**:参数anchors指定kmeans计算出来的anchor box的长宽的绝对值(与网络输入大小相关),num参数为anchor box的数量,  
另外还有bias\_match,classes,coords等参数.在parser.c代码中的parse\_region函数中解析这些参数,并保存在region\_layer.num参数保存在l.n变量中;anchors保存在l.biases数组中.region\_layer的前向传播中使用for(n = 0; n < l.n; ++n)这样的语句,因此,如果在配置文件中anchors的数量大于num时,仅使用前num个,小于时内存越界.

region层的输入和输出大小与前一层(1x1 conv)的输出大小和网络的输入大小相关.

**Detection层**: 坐标及类别结果输出层.

# YOLOv2

[YOLO9000: Better, Faster, Stronger](https://arxiv.org/abs/1612.08242)

## Convolutional With Anchor Boxes：卷积层替换全连接层

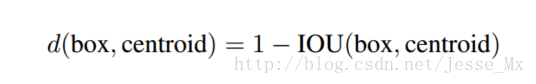
之前的YOLO利用**全连接层**的数据完成边框的预测，**导致丢失较多的空间信息**，定位不准。作者在这一版本中借鉴了Faster R-CNN中的anchor思想，回顾一下，anchor是RNP网络中的一个关键步骤，说的是在卷积特征图上进行滑窗操作，每一个中心可以预测9种不同大小的建议框。看到YOLOv2的这一借鉴，我只能说SSD的作者是有先见之明的。

为了引入anchor boxes来预测bounding boxes，作者在网络中果断**去掉了全连接层**。剩下的具体怎么操作呢？首先，作者**去掉了后面的一个池化层**以确保输出的卷积特征图有更高的分辨率。然后，通过缩减网络，让图片输入分辨率为416 \* 416，这一步的目的是为了让后面产生的卷积特征图宽高都为奇数，这样就可以产生一个center cell。作者观察到，大物体通常占据了图像的中间位置， 就可以只用中心的一个cell来预测这些物体的位置，否则就要用中间的4个cell来进行预测，这个技巧可稍稍提升效率。最后，YOLOv2使用了卷积层降采样（factor为32），使得输入卷积网络的416 \* 416图片最终得到13 \* 13的卷积特征图（416/32=13）。

## Dimension Clusters（维度聚类）：统计anchor宽高比

作者在使用anchor的时候遇到了两个问题，第一个是anchor boxes的宽高维度往往是精选的先验框（hand-picked priors），虽说在训练过程中网络也会学习调整boxes的宽高维度，最终得到准确的bounding boxes。但是，如果一开始就选择了更好的、更有代表性的先验boxes维度，那么网络就更容易学到准确的预测位置。

和以前的精选boxes维度不同，作者**使用了K-means聚类方法类训练bounding boxes，可以自动找到更好的boxes宽高维度。**传统的K-means聚类方法使用的是欧氏距离函数，也就意味着较大的boxes会比较小的boxes产生更多的error，聚类结果可能会偏离。为此，**作者采用的评判标准是IOU得分**（也就是boxes之间的交集除以并集），这样的话，error就和box的尺度无关了，最终的距离函数为：

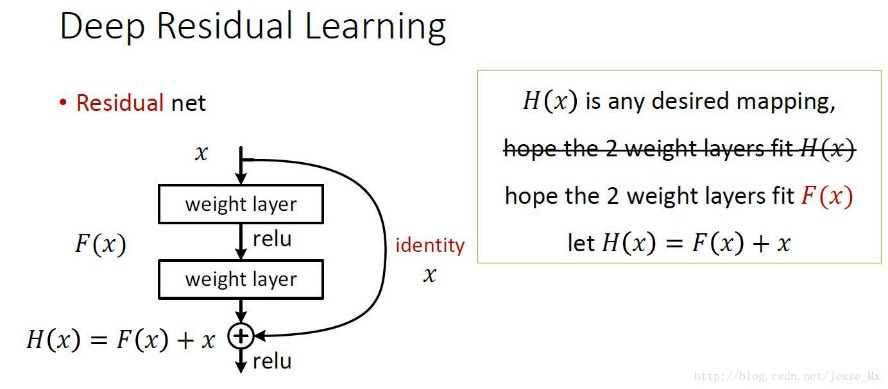


可以看到，平衡复杂度和IOU之后，最终得到k值为5，意味着作者选择了5种大小的box维度来进行定位预测，这与手动精选的box维度不同。结果中扁长的框较少，而瘦高的框更多（这符合行人的特征），这种结论如不通过聚类实验恐怕是发现不了的。

当然，作者也做了实验来对比两种策略的优劣，如下图，使用聚类方法，仅仅5种boxes的召回率就和Faster R-CNN的9种相当。说明K-means方法的引入使得生成的boxes更具有代表性，为后面的检测任务提供了便利。

## Fine-Grained Features（细粒度特征）：上下特征图融合

上述网络上的修改使YOLO最终在13 \* 13的特征图上进行预测，虽然这足以胜任大尺度物体的检测，但是用上细粒度特征的话，这**可能对小尺度的物体检测有帮助**。Faser R-CNN和SSD都在不同层次的特征图上产生区域建议（SSD直接就可看得出来这一点），获得了多尺度的适应性。这里使用了一种不同的方法，**简单添加了一个转移层（ passthrough layer），这一层要把浅层特征图（分辨率为26 \* 26，是底层分辨率4倍）连接到深层特征图**。



这个转移层也就是把高低两种分辨率的特征图做了一次连结，连接方式是叠加特征到不同的通道而不是空间位置，类似于Resnet中的identity mappings。这个方法把26 \* 26 \* 512的特征图连接到了13 \* 13 \* 2048的特征图，这个特征图与原来的特征相连接。YOLO的检测器使用的就是经过扩张的特征图，它可以拥有更好的细粒度特征，使得模型的性能获得了1%的提升。

**补充：**关于passthrough layer，具体来说就是特征重排（不涉及到参数学习），前面26 \* 26 \* 512的特征图使用按行和按列隔行采样的方法，就可以得到4个新的特征图，维度都是13 \* 13 \* 512，然后做concat操作，得到13 \* 13 \* 2048的特征图，将其拼接到后面的层，相当于做了一次特征融合，有利于检测小目标。

## Multi-Scale Training：不同输入大小训练

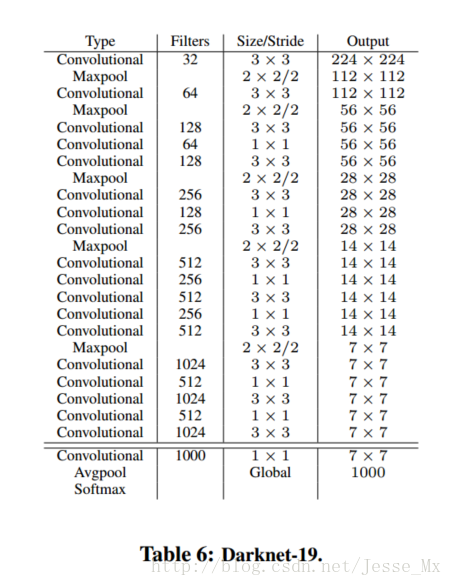
原来的YOLO网络使用固定的448 \* 448的图片作为输入，现在加入anchor boxes后，输入变成了416 \* 416。目前的网络**只用到了卷积层和池化层，那么就可以进行动态调整（意思是可检测任意大小图片）**。作者希望YOLOv2具有不同尺寸图片的鲁棒性，因此在训练的时候也考虑了这一点。

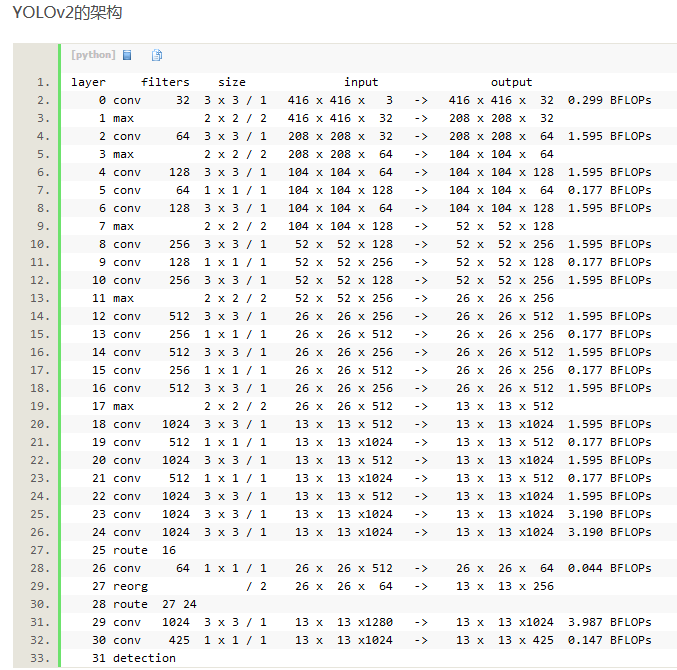
不同于固定输入网络的图片尺寸的方法，作者在几次迭代后就会微调网络。**每经过10次训练（10 epoch），就会随机选择新的图片尺寸**。YOLO网络使用的降采样参数为32，那么就使用32的倍数进行尺度池化**{320,352，…，608}**。最终最小的尺寸为320 \* 320，最大的尺寸为608 \* 608。接着按照输入尺寸调整网络进行训练。

## Darknet-19：网络结构

YOLOv2使用了一个新的分类网络作为特征提取部分，参考了前人的先进经验，比如类似于VGG，作者使用了较多的3 \* 3卷积核，在每一次池化操作后把通道数翻倍。借鉴了network in network的思想，网络使用了全局平均池化（global average pooling），**把1 \* 1的卷积核置于3 \* 3的卷积核之间，用来压缩特征。**也用了batch normalization（前面介绍过）稳定模型训练。

最终得出的基础模型就是Darknet-19，如下图，其包含19个卷积层、5个最大值池化层（maxpooling layers ），下图展示网络具体结构。Darknet-19运算次数为55.8亿次，imagenet图片分类top-1准确率72.9%，top-5准确率91.2%。

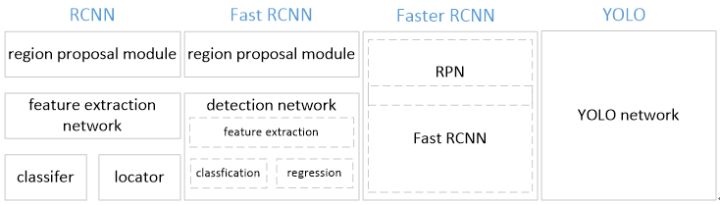
****



# YOLOv1: CVPR2016

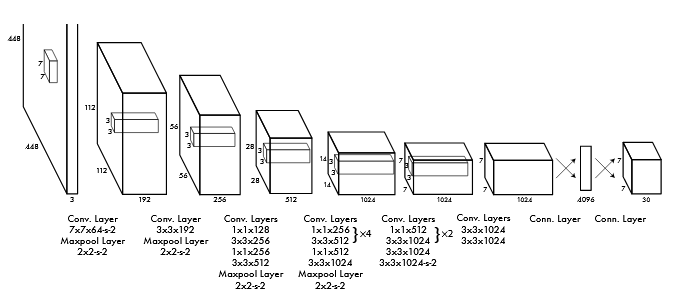
CVPR2016 目标检测论文[YOLO: Unified, Real-Time Object Detection](https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1506.02640)

从R-CNN到Fast R-CNN一直采用的思路是proposal+分类 （proposal 提供位置信息， 分类提供类别信息）精度已经很高，但是速度还不行。 YOLO提供了另一种更为直接的思路： 直接在输出层回归bounding box的位置和bounding box所属的类别(整张图作为网络的输入，把 Object Detection 的问题转化成一个 Regression 问题)。



## 网络结构:

YOLO检测网络包括24个卷积层和2个全连接层，如下图所示。



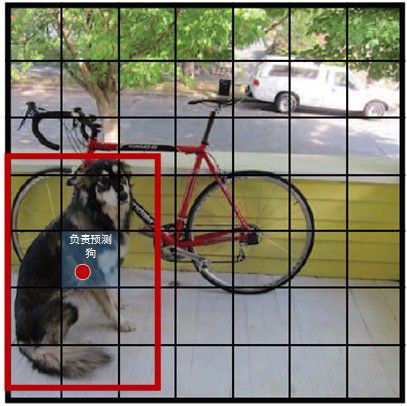
其中，卷积层用来提取图像特征，全连接层用来预测图像位置和类别概率值。

YOLO网络借鉴了GoogLeNet分类网络结构。不同的是，YOLO未使用inception module，而是使用1x1卷积层（此处1x1卷积层的存在是为了跨通道信息整合）+3x3卷积层简单替代。

YOLO论文中，作者还给出一个更轻快的检测网络**fast YOLO**，它只有9个卷积层和2个全连接层。使用titan x GPU，fast YOLO可以达到155fps的检测速度，但是mAP值也从YOLO的63.4%降到了52.7%，但却仍然远高于以往的实时物体检测方法（DPM）的mAP值。

## 全连接输出层的定义（最后输出）

YOLO将输入图像分成SxS个格子，每个格子负责检测‘落入’该格子的物体。若某个物体的中心位置的坐标落入到某个格子，那么这个格子就负责检测出这个物体。如下图所示，图中物体狗的中心点（红色原点）落入第5行、第2列的格子内，所以这个格子负责预测图像中的物体狗。



每个格子输出B个bounding box（包含物体的矩形区域）信息，以及C个物体属于某种类别的概率信息。

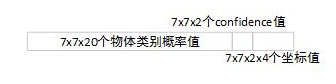
Bounding box信息包含5个数据值，分别是x,y,w,h,和confidence。

x,y是bounding box中心位置相对于当前格子位置的偏移值，并且被归一化到[0,1]；

w,h是bounding box的宽度和高度。注意：实际训练过程中，w和h的值使用图像的宽度和高度进行归一化到[0,1]区间内。

confidence反映当前bounding box是否包含物体以及物体位置的准确性，计算方式如下：

**confidence = P(object)\* IOU**, 其中，若bounding box包含物体，则P(object) = 1；否则P(object) = 0。 IOU(intersection over union)为预测boundingbox与物体真实区域的交集面积（以像素为单位，用真实区域的像素面积归一化到[0,1]区间）。**注意**：class信息是针对每个网格的，confidence信息是针对每个bounding box的）



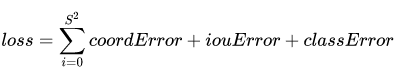
### 备注：

由于输出层为全连接层，因此在检测（预测）时，YOLO模型只支持与训练图像相同的输入分辨率。**虽然每个格子可以预测B个bounding box，但是最终只选择IOU最高的bounding box作为物体检测输出，**即每个格子最多只预测出一个物体。当物体占画面比例较小，如图像中包含畜群或鸟群时，每个格子包含多个物体，但却只能检测出其中一个。这是YOLO方法的一个缺陷。

可以预测出7 \* 7 \* B(B=2) = 98个目标窗口，然后根据阈值去除可能性比较低的目标窗口，再由NMS去除冗余窗口即可。

## Loss函数定义

YOLO使用**均方和误差**作为loss函数来优化模型参数，即网络输出的S\*S\*(B\*5 + C)维向量与真实图像的对应S\*S\*(B\*5 + C)维向量的均方和误差。如下式所示。其中，coordError、iouError和classError分别代表预测数据与标定数据之间的**坐标误差、IOU误差和分类误差。**



YOLO对上式loss的计算进行了如下修正：

**修改1：坐标误差、IOU误差、分类误差对网络loss的贡献值是不同的**，因此YOLO在计算

loss时，使用**λcoord= 5**来修正coordError；

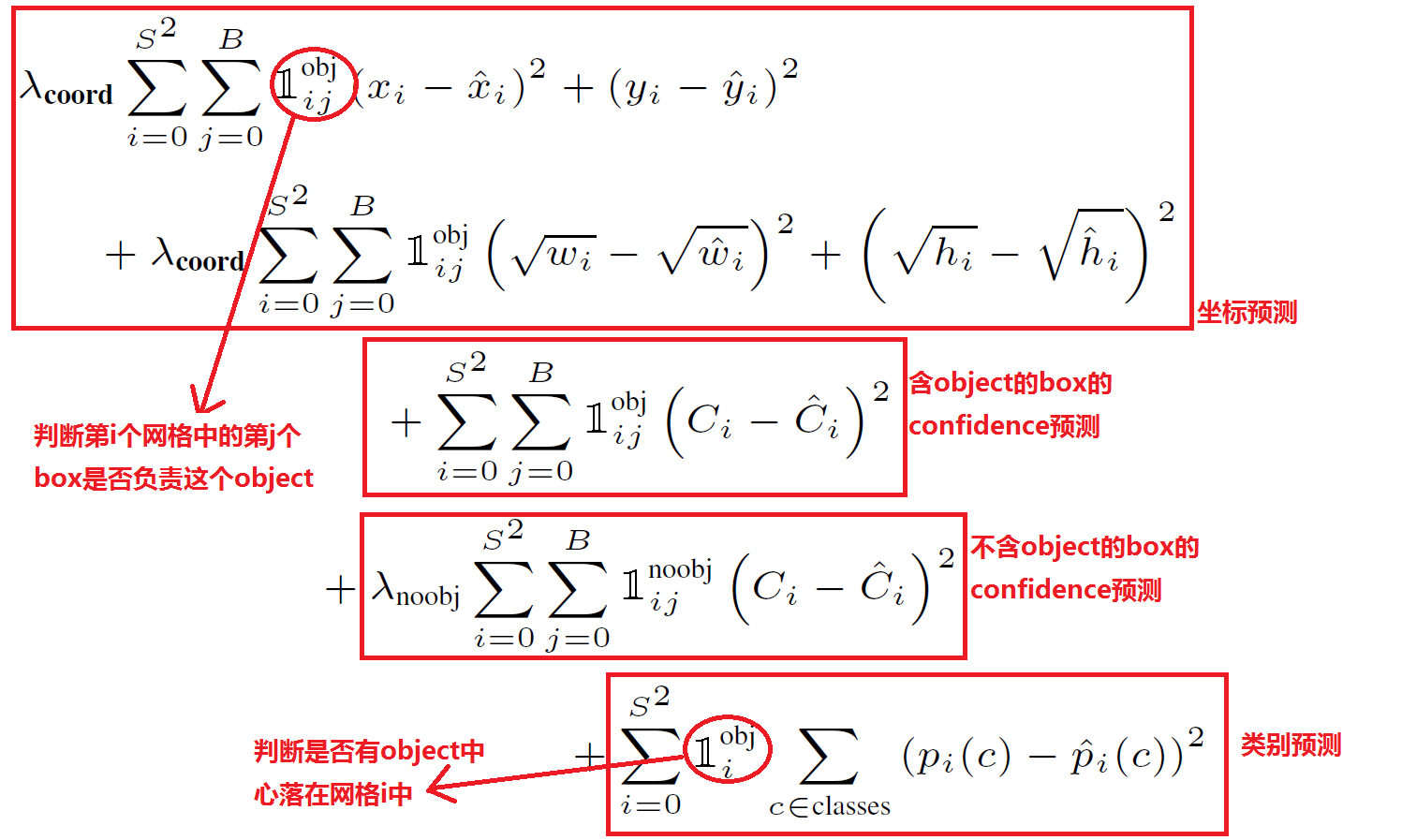
**修改2：**在计算IOU误差时，**包含物体的格子与不包含物体的格子，二者的IOU误差对网络**

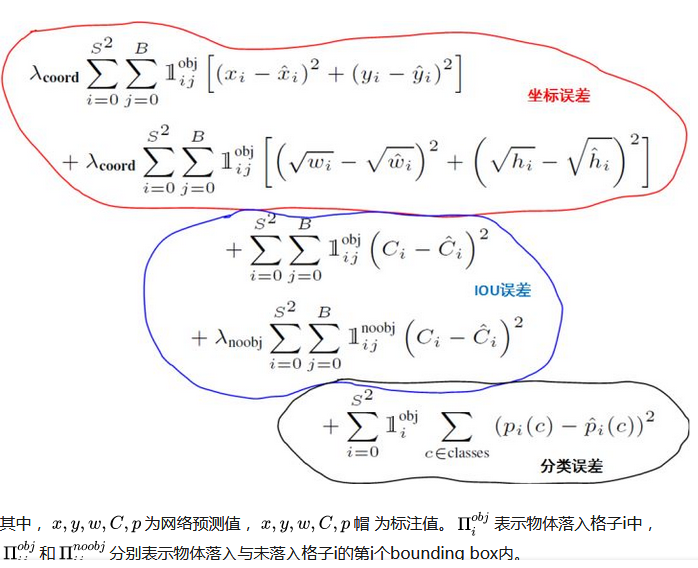
**loss的贡献值是不同的**。若采用相同的权值，那么不包含物体的格子的confidence值近似为0，变相放大了包含物体的格子的confidence误差在计算网络参数梯度时的影响。为解决这个问题，YOLO 使用**λnoobj=0.5**来修正iouError。（**备注**：此处的“包含”是指存在一个物体，它的中心坐标落入到格子内。）

**修改3：**对于相等的误差值，**大物体误差对检测的影响应小于小物体误差对检测的影响**。这

是因为，相同的位置偏差占大物体的比例远小于同等偏差占小物体的比例。YOLO将物体大小的信息项（w和h）进行求平方根来改进这个问题。（**注**：这个方法并不能完全解决这个问题）。

综上，YOLO在训练过程中Loss计算如下式所示：





### 备注：存在的缺陷

YOLO方法模型训练依赖于物体识别标注数据，因此，对于非常规的物体形状或比例，YOLO的检测效果并不理想。

YOLO采用了多个下采样层，网络学到的物体特征并不精细，因此也会影响检测效果。

YOLO loss函数中，大物体IOU误差和小物体IOU误差对网络训练中loss贡献值接近（虽然采用求平方根方式，但没有根本解决问题）。因此，对于小物体，小的IOU误差也会对网络优化过程造成很大的影响，从而降低了物体检测的定位准确性。

## 训练：

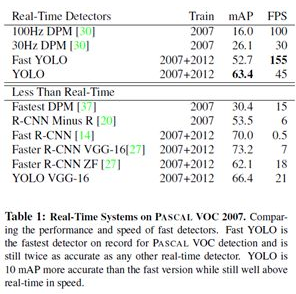
YOLO模型训练分为两步：

1）预训练。使用ImageNet  
1000类数据训练YOLO网络的前20个卷积层+1个average池化层+1个全连接层。训练图像分辨率resize到224x224。

2）用步骤1）得到的前20个卷积层网络参数来初始化YOLO模型前20个卷积层的网络参数，然后用VOC 20类标注数据进行YOLO模型训练。为提高图像精度，在训练检测模型时，将输入图像分辨率resize到448x448。

### 效果

下表给出了YOLO与其他物体检测方法，在检测速度和准确性方面的比较结果（使用VOC 2007数据集）。



YOLO具有如下优点：

**快。**YOLO将物体检测作为回归问题进行求解，整个检测网络pipeline简单。在titan x GPU

上，在保证检测准确率的前提下（63.4% mAP，VOC 2007 test set），可以达到45fps的检测速度。

**背景误检率低。**YOLO在训练和推理过程中能‘看到’整张图像的整体信息，而基于region

proposal的物体检测方法（如rcnn/fast rcnn），在检测过程中，只‘看到’候选框内的局部图像信息。因此，若当图像背景（非物体）中的部分数据被包含在候选框中送入检测网络进行检测时，容易被误检测成物体。测试证明，YOLO对于背景图像的误检率低于fast rcnn误检率的一半。

**通用性强。**YOLO对于艺术类作品中的物体检测同样适用。它对非自然图像物体的检测率远

远高于DPM和RCNN系列检测方法。

但相比RCNN系列物体检测方法，YOLO具有以下缺点：

识别物体位置精准性差。

召回率低。

## 缺陷：

* YOLO对相互靠的很近的物体（挨在一起且中点都落在同一个格子上的情况），还有很小的群体 检测效果不好，这是因为**一个网格中只预测了两个框，并且只属于一类。**
* 测试图像中，当同一类物体出现的不常见的长宽比和其他情况时泛化能力偏弱。
* 由于损失函数的问题，定位误差是影响检测效果的主要原因，尤其是大小物体的处理上，还有待加强。

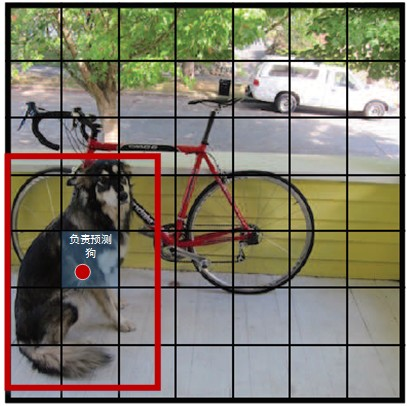
# 以下内容可忽略

### 预训练分类网络:

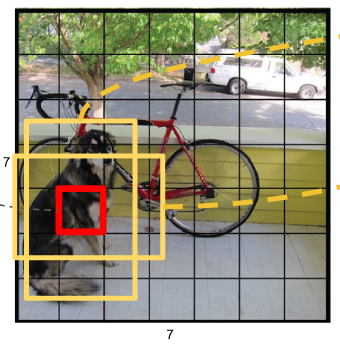
在 ImageNet 1000-class competition dataset上预训练一个分类网络，这个网络是Figure3中的前20个卷机网络+average-pooling layer+ fully connected layer （此时网络输入是224\*224）。训练检测网络：

转换模型去执行检测任务，《Object detection networks on convolutional feature maps》提到说在预训练网络中增加卷积和全链接层可以改善性能。在他们例子基础上添加4个卷积层和2个全链接层，随机初始化权重。检测要求细粒度的视觉信息，所以把网络输入也又224\*224变成448\*448。见Figure3。

* 一幅图片分成7x7个网格(grid cell)，某个物体的中心落在这个网格中此网格就负责预测这个物体。



最后一层输出为 （7\*7）\*30的维度。每个 1\*1\*30的维度对应原图7\*7个cell中的一个，1\*1\*30中含有类别预测和bbox坐标预测。总得来讲就是让网格负责类别信息，bounding box主要负责坐标信息(部分负责类别信息：confidence也算类别信息)。

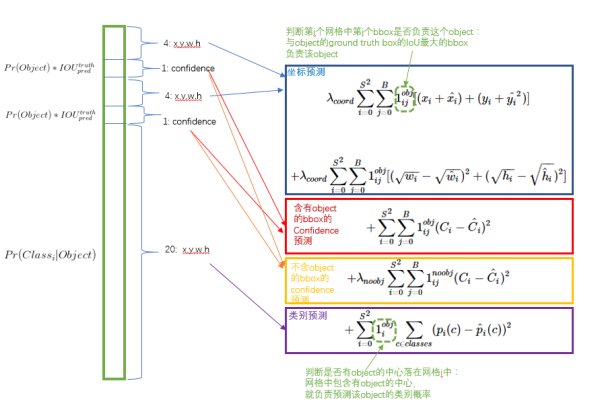


具体如下：

**每个网格（1\*1\*30维度对应原图中的cell）要预测2个bounding box** （图中黄色实线框）的坐标（x_{center},y_{center},w,h） ，其中：中心坐标的x_{center},y_{center} 相对于对应的网格归一化到0-1之间，w,h用图像的width和height归一化到0-1之间。 每个bounding box除了要回归自身的位置之外，还要附带预测一个confidence值。 这个confidence代表了所预测的box中含有object的置信度和这个box预测的有多准两重信息：confidence = Pr(Object) \ast IOU^{truth}_{pred}。其中如果有ground true box(人工标记的物体)落在一个grid cell里，第一项取1，否则取0。 第二项是预测的bounding box和实际的ground truth box之间的IOU值。即：每个bounding box要预测 x_{center},y_{center},w,h,confidence,共5个值 ，2个bounding box共10个值，对应 1\*1\*30维度特征中的前10个。

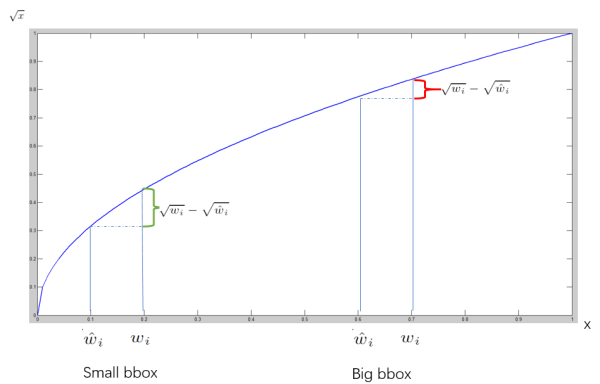
**每个网格还要预测类别信息**，论文中有20类。7x7的网格，每个网格要预测2个 bounding box 和 20个类别概率，输出就是 7x7x(5x2 + 20) 。 (通用公式： SxS个网格，每个网格要预测B个bounding box还要预测C个categories，输出就是S x S x (5\*B+C)的一个tensor。 注意：class信息是针对每个网格的，confidence信息是针对每个bounding box的）

## 损失函数设计：



损失函数的设计目标就是让坐标（x,y,w,h），confidence，classification 这个三个方面达到很好的平衡.简单的全部采用了sum-squared error loss来做这件事会有以下不足： a) 8维的localization error和20维的classification error同等重要显然是不合理的； b) 如果一个网格中没有object（一幅图中这种网格很多），那么就会将这些网格中的box的confidence push到0，相比于较少的有object的网格，这种做法是overpowering的，这会导致网络不稳定甚至发散。 解决方案如下：

* 更重视8维的坐标预测，给这些损失前面赋予更大的loss weight, 记为 \lambda_{coord},在pascal VOC训练中取5。（上图蓝色框）
* 对没有object的bbox的confidence loss，赋予小的loss weight，记为 \lambda_{noobj}，在pascal VOC训练中取0.5。（上图橙色框）
* 有object的bbox的confidence loss (上图红色框) 和类别的loss （上图紫色框）的loss weight正常取1。
* 对不同大小的bbox预测中，相比于大bbox预测偏一点，小box预测偏一点更不能忍受。而sum-square error loss中对同样的偏移loss是一样。 为了缓和这个问题，作者用了一个比较取巧的办法，就是将box的width和height取平方根代替原本的height和width。 如下图：small bbox的横轴值较小，发生偏移时，反应到y轴上的loss（下图绿色）比big box(下图红色)要大。



一个网格预测多个bounding box，在训练时我们希望每个object（ground true box）只有一个bounding box专门负责（一个object 一个bbox）。具体做法是与ground true box（object）的IOU最大的bounding box 负责该ground true box(object)的预测。这种做法称作bounding box predictor的specialization(专职化)。每个预测器会对特定（sizes,aspect ratio or classed of object）的ground true box预测的越来越好。（个人理解：IOU最大者偏移会更少一些，可以更快速的学习到正确位置）

## 测试

**Test的时候**，**每个网格预测的class信息( Pr(Class_i | Object))**和**bounding box预测的confidence信息( Pr(Object) \ast IOU^{truth}_{pred})** 相乘，就得到每个bounding box的class-specific confidence score。

https://pic2.zhimg.com/v2-9bc2b9b6e509c304323be0bbeb72709d_b.png

等式左边第一项就是每个网格预测的类别信息，第二三项就是每个bounding box预测的confidence。这个乘积即encode了预测的box属于某一类的概率，也有该box准确度的信息。

对每一个网格的每一个bbox执行同样操作： 7x7x2 = 98 bbox （每个bbox既有对应的class信息又有坐标信息）

得到每个bbox的class-specific confidence score以后，设置阈值，滤掉得分低的boxes，对保留的boxes进行NMS处理，就得到最终的检测结果。

## v1:

448\*448 , 7\*7\*(5\*B + C) B（B=2）个bounding box信息c=20类

**tiny:** 8个卷积层 + 1个全连接层,

**small** : 24个卷积层 + 3个全连接成

**yolo**: 24个卷积层 + 1个全连接层

## YOLO9000

新的YOLO版本论文全名叫“YOLO9000: Better, Faster, Stronger”，主要有两个大方面的改进：

第一，作者使用了一系列的方法对原来的YOLO多目标检测框架进行了改进，在保持原有速度的优势之下，精度上得以提升。VOC 2007数据集[测试](http://lib.csdn.net/base/softwaretest)，67FPS下mAP达到76.8%，40FPS下mAP达到78.6%，基本上可以与Faster R-CNN和SSD一战。这一部分是本文主要关心的地方。

第二，作者提出了**一种目标分类与检测的联合训练方法**，通过这种方法，YOLO9000可以同时在COCO和ImageNet数据集中进行训练，训练后的模型可以实现多达9000种物体的实时检测。

## Yolo v2:

1. YOLOv1\v2网络在每一个卷积层后添加**batch normalization**，通过这一方法，mAP获得了2%的提升。batch normalization 也有助于规范化模型，可以在舍弃dropout优化后依然**不会过拟合**。
2. **训练图像大小在不断调整**；yolo v2 (608\*608),调整机制是啥？？？

使用32的倍数进行尺度池化{320,352，…，608}，降采样

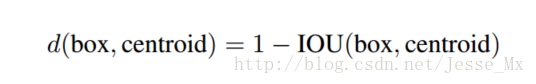
if(l.random && count++%10 == 0)

dim = (rand() % 10 + 10) \* 32 //dim是图像尺度

1. 之前的YOLO利用全连接层的数据完成边框的预测，导致丢失较多的空间信息，定位不准。作者在这一版本中**借鉴了Faster R-CNN中的anchor思想**，回顾一下，anchor是RNP网络中的一个关键步骤，说的是在卷积特征图上进行**滑窗操作**，每一个中心可以预测9种不同大小的**建议框**。看到YOLOv2的这一借鉴，我只能说SSD的作者是有先见之明的。
2. 23 卷积层，取消yolo v1的全连接层。
3. **Dimension Clusters（维度聚类）**

作者在使用anchor的时候遇到了两个问题，

1）第一个是**anchor boxes的宽高维度往往是精选的先验框**（hand-picked priors），虽说在训练过程中网络也会**学习调整boxes的宽高维度**，最终得到准确的bounding boxes。但是，如果一开始就选择了更好的、更有代表性的先验boxes维度，那么网络就更容易学到准确的预测位置。

和以前的精选boxes维度不同，作者使用了**K-means聚类方法类训练bounding boxes**，可以自动找到更好的boxes宽高维度。传统的K-means聚类方法使用的是欧氏距离函数，也就意味着较大的boxes会比较小的boxes产生更多的error，聚类结果可能会偏离。为此，作者采用的评判标准是IOU得分（也就是boxes之间的交集除以并集），这样的话，error就和box的**尺度无关**了，最终的距离函数为：   


2）作者在使用anchor boxes时发现的第二个问题就是：**模型不稳定（**会产生模型不稳定的问题，该问题产生于边界框位置的预测。简单的解释，如果训练的图片中的物体一张是在左面，下一张又在右面，就会产生这样的波动，显然的这个过程是不受控制的**）**

定位预测值被归一化后，参数就更容易得到学习，模型就更稳定。作者又把ground truth限制在了0到1之间，利用logistic回归函数来进行这一限制。 修改了bounding box。

1. **Fine-Grained Features（细粒度特征**）:

有别于faster rcnn和SSD采用多尺度的特征图进行预测，yolov2提出了一个全新的思路，作者引入了**passthrough layer**，这个层的作用就是将上一层特征图的相邻像素都切除一部分组成了另外一个通道。例如，将**26\*26\*512的特征图变为13\*13\*2048的特征图**（这里具体的实现过程需要看作者的源码，但是，为了解释这个变化过程，可以做这样的解释，就是将一个26\*26的图的像素放到4个13\*13的图中，水平每2个像素取1个，垂直也是每2个像素取一个，一共就可以得到2\*2=4个，512\*4=2048），使得特征图的数目提高了4倍，同时，相比于26\*26的特征图，13\*13的特征图更有利用小目标物的检测。

## 参数

yolo-tiny.cfg

#### [connected]

output= 1470 # S\*S\*(B\*5 + class\_num)

activation=linear

#### [detection]

classes=20 # 改成实际的class\_num

coords=4 #框框的4个坐标

rescore=1 # 得分

side=7 # 分的越多，检测的可能越准

num=2 #bounding box的个数

softmax=0

sqrt=1

jitter=.2

object\_scale=1

noobject\_scale=.5

class\_scale=1

coord\_scale=5

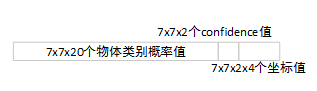
## 备注：

1. Yolo.cfg与yolo.weights版本必须一致，不然检测效果不行或者检测不到物体；
2. 如果没有opencv，检测结果就只会报存在predictions.png,不会弹出来。
3. Yolo.2.0.cfg是中间版本
4. Yolov1测试: ./darknet **yolo test** cfg/yolo.cfg cfg/yolo.weights data/dog.jpg -thresh 0.5
5. Yolov2测试：./darknet **detect** cfg/yolo.cfg cfg/yolo.weights data/dog.jpg -thresh 0.5
6. 最新版本，输入图片大小从448\*448变成608\*608
7. Yolov1: bounding box=2， yolov2：bouding box=5，但是都是水平的矩形，一对一回归
8. Yolo在打标签时只给(u,v,w,h), 没有给confidence, 是通过IOU计算来的。

confidence反映当前bounding box是否包含物体以及物体位置的准确性，计算方式如下：confidence = P(object) \* IOU, 其中，若bounding box包含物体，则P(object) = 1；否则P(object) = 0. IOU(intersection over union)为预测bounding box与物体真实区域的交集面积（以像素为单位，用真实区域的像素面积归一化到[0,1]区间）。

1. 修改makefile文件，make –j16， （make: Nothing to be done for `all'. 这句提示是说明你已经编译好了，而且没有对代码进行任何改动。先make clean，再make）
2. Yolo v1每个网格**只预测一个物体**，**最后接的是全连接层（所以输入图像大小必须一致）**，大小是7\*7\*(B\*5+C)，B是bounding box的个数=2，C表示类别的概率，大小是类别数=20，5=四个点坐标+confidence.

图片大小448\*448，输出向量为7\*7\*(20 + 2\*5)=1470维；



1. Yolo v2每个网格**可预测多个物体**，最后是卷积层，大小是S\*S\*B（C+4+1），B=5个boundingbox, C=类别个数，4是四个点坐标，1指confidence；

**备注**：**yolo v2的图像大小在不断调整**。

1. 将yolo v2的网络结构换成别的需要修改的地方：
2. cfg/yolo.cfg 的网络结构、batch大小（不然CUDA Error: out of memory）、类别大小（不含背景）、最后卷积层的滤波器个数需要计算准确（**滤波器个数=B\*（C+4+1）**）；
3. cfg/voc.data 修改样本路径，类别名字等；
4. 如果使用GPU跑训练，修改makefile，重新make –j16；

**备注**：

1. **跑训练代码**：./darknet **detector train** ./cfg/voc-fcrn.data ./cfg/yolo-fcrn.cfg

**测试**：./darknet **detector test** cfg/voc.data cfg/yolo-voc.cfg backup/yolo-voc\_6000.weights data/dog.jpg -thresh 0.5

2）制作数据集：数据(images+labels)：/data/train.txt /data/test.txt

图片：/notebooks/yushan/darknet-master/textdata/images/kathmandu\_147\_28.jpg

标签：/notebooks/yushan/darknet-master/textdata/labels/kathmandu\_147\_28.txt

/notebooks/yushan/darknet-master/textdata/labels 只需给出.txt文件，代码会在.txt图片路径的上一层找labels.

3）/src/darknet.c (入口)

/src/detector.c （run\_detector():属性(train、valid、valid2、recall、demo))

/src/network.c ()

//每100次或者1000次保存一次权重

4）**屏幕上显示：**

**detector.c loss**

1: 510.682007, 510.682007 avg, 0.000000 rate, 17.967775 seconds, 1 images

printf("%d: **%f**, %f avg, %f rate, %lf seconds, %d images\n", get\_current\_batch(net), loss, **avg\_loss**, get\_current\_rate(net), sec(clock()-time), i\*imgs);

get\_current\_batch(net) = \*net.seen/imgs (imgs=net.batch\*net.subdivisions, imgs是一次加载到内存的图像数量，/net.seen就是已经经过网络训练（看）的图片数量)

region\_layer.c

printf("Region Avg IOU: %f, Class: %f, Obj: %f, No Obj: %f, Avg Recall: %f, count: %d\n", avg\_iou/count, avg\_cat/class\_count, avg\_obj/count, avg\_anyobj/(l.w\*l.h\*l.n\*l.batch), recall/count, count);

printf 在屏幕上打印

subdivisions越大，越等降低提示信息频率

1. 训练过程中屏幕打印的loss是一个batch的，而不是一个epoch后的loss；

还不写log，可以像这样写log：

例如./darknet detector train ./cfg/voc-fcrn.data ./cfg/yolo.cfg **| tee train.log**

**使用验证集评估模型**：./darknet detector valid cfg/voc-fcrn.data cfg/yolo-fcrn.cfg backup/yolo-fcrn\_70000.weights

**查看召回率**：./darknet detector recall cfg/voc-fcrn.data cfg/yolo-fcrn.cfg backup/yolo-fcrn\_70000.weights