深入浅出Yolo系列之Yolov5核心基础知识完整讲解

江大白
公众号「江大白」，领取《人工智能算法岗江湖武林秘籍》

已关注

大白在之前写过《深入浅出Yolo系列之Yolov3&Yolov4核心基础知识完整讲解》

对**Yolov4**的相关**基础知识**做了比较系统的梳理，但**Yolov4**后不久，又出现了**Yolov5**，虽然作者没有放上和**Yolov4**的直接测试对比，但在COCO数据集的测试效果还是很可观的。

很多人考虑到**Yolov5**的**创新性不足**，对算法是否能够**进化**，称得上**Yolov5**而议论纷纷。

但既然称之为**Yolov5**，也有很多非常不错的地方值得我们学习。不过因为**Yolov5**的网络结构和**Yolov3**、**Yolov4**相比，不好可视化，导致很多同学看**Yolov5**看的云里雾里。

因此本文，大白主要对Yolov5四种网络结构的各个细节做一个深入浅出的分析总结，和大家一些探讨学习。

更新提醒（2021.05.21）：Yolov3&Yolov4的相关视频，已经更新上传，可点击查看。

求职跳槽福利：为了便于大家求职、跳槽的准备，大白将**45家大厂**的**3500篇面经**，按照知识框架，整理成700多页的**《人工智能算法岗江湖武林秘籍》**，限时开放下载，点击查看下载。

本文目录

1 Yolov5四种网络模型

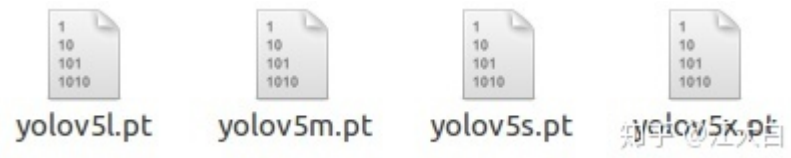
Yolov5官方代码中，给出的目标检测网络中一共有4个版本，分别是**Yolov5s、Yolov5m、Yolov5l、Yolov5x**四个模型。

学习一个新的算法，最好在脑海中对**算法网络的整体架构**有一个清晰的理解。

但比较尴尬的是，**Yolov5代码**中给出的网络文件是**yaml格式**，和原本Yolov3、Yolov4中的**cfg**不同。

因此无法用netron工具直接可视化的查看网络结构，造成有的同学不知道如何去学习这样的网络。

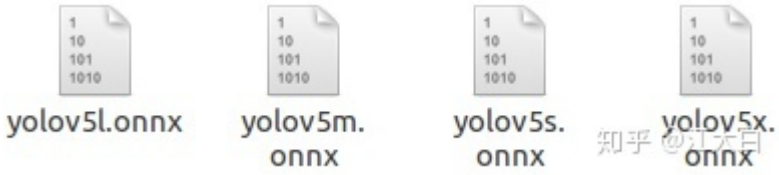
比如下载了**Yolov5的四个pt格式**的权重模型：



大白在《深入浅出Yolo系列之Yolov3&Yolov4核心基础完整讲解》中讲到，可以使用netron工具打开网络模型。

但因为netron对pt格式的文件兼容性并不好，直接使用netron工具打开，会发现，根本无法显示全部网络。

因此可以采用**pt->onnx->netron**的折中方式，先使用Yolov5代码中**models/export.py**脚本将pt文件转换为onnx格式，再用netron工具打开，这样就可以看全网络的整体架构了。



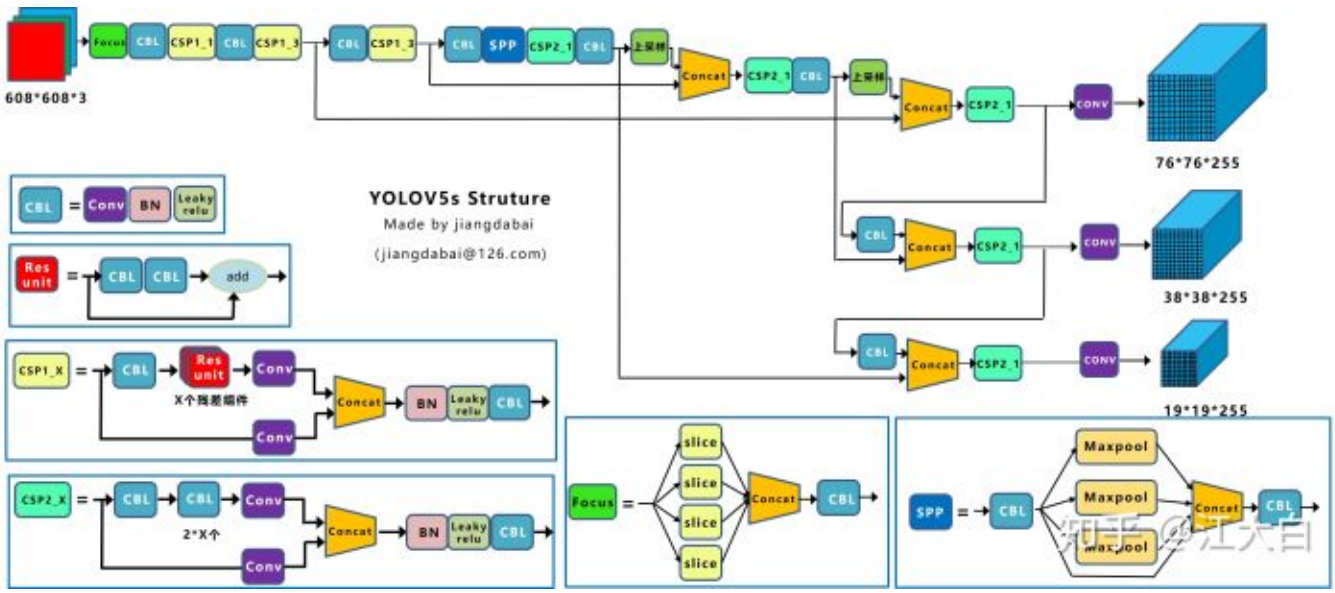
如果有同学对netron工具还不是很熟悉，这里还是放上安装netron工具的详解，如果需要安装，可以移步大白的另一篇文章：《网络可视化工具netron详细安装流程》

如需下载**Yolov5**整体的4个网络pt文件及onnx文件，也可点击链接查看下载，便于直观的学习。

1.1 Yolov5网络结构图

安装好netron工具，就可以可视化的打开Yolov5的网络结构。

这里大白也和之前讲解Yolov3&Yolov4同样的方式，绘制了Yolov5s整体的网络结构图。配合netron的可视化网络结构查看，脑海中的架构会更加清晰。



本文也会以Yolov5s的网络结构为主线，讲解与其他三个模型（**Yolov5m、Yolov5l、Yolov5x**）的不同点，让大家对于Yolov5有一个深入浅出的了解。

1.2 网络结构可视化

将四种模型pt文件的转换成对应的onnx文件后，即可使用netron工具查看。
但是，有些同学可能不方便，使用脚本转换查看。
因此，大白也上传了每个网络结构图的图片，也可以直接点击查看。
虽然没有netron工具更直观，但是也可以学习了解。

1.2.1 Yolov5s网络结构

Yolov5s网络是Yolov5系列中**深度最小**，特征图的**宽度最小**的网络。后面的3种都是在此基础上不断加深，不断加宽。

上图绘制出的网络结构图也是**Yolov5s**的结构，大家也可直接点击查看，Yolov5s的网络结构可视化的图片。

1.2.2 Yolov5m网络结构

此处也放上netron打开的**Yolov5m**网络结构可视图，点击即可查看，后面第二版块会详细说明不同模型的不同点。

1.2.3 Yolov5l网络结构

此处也放上netronx打开的**Yolov5l**网络结构可视图，点击即可查看。

1.2.4 Yolov5x网络结构

此处也放上netronx打开的**Yolov5x**网络结构可视图，点击即可查看。

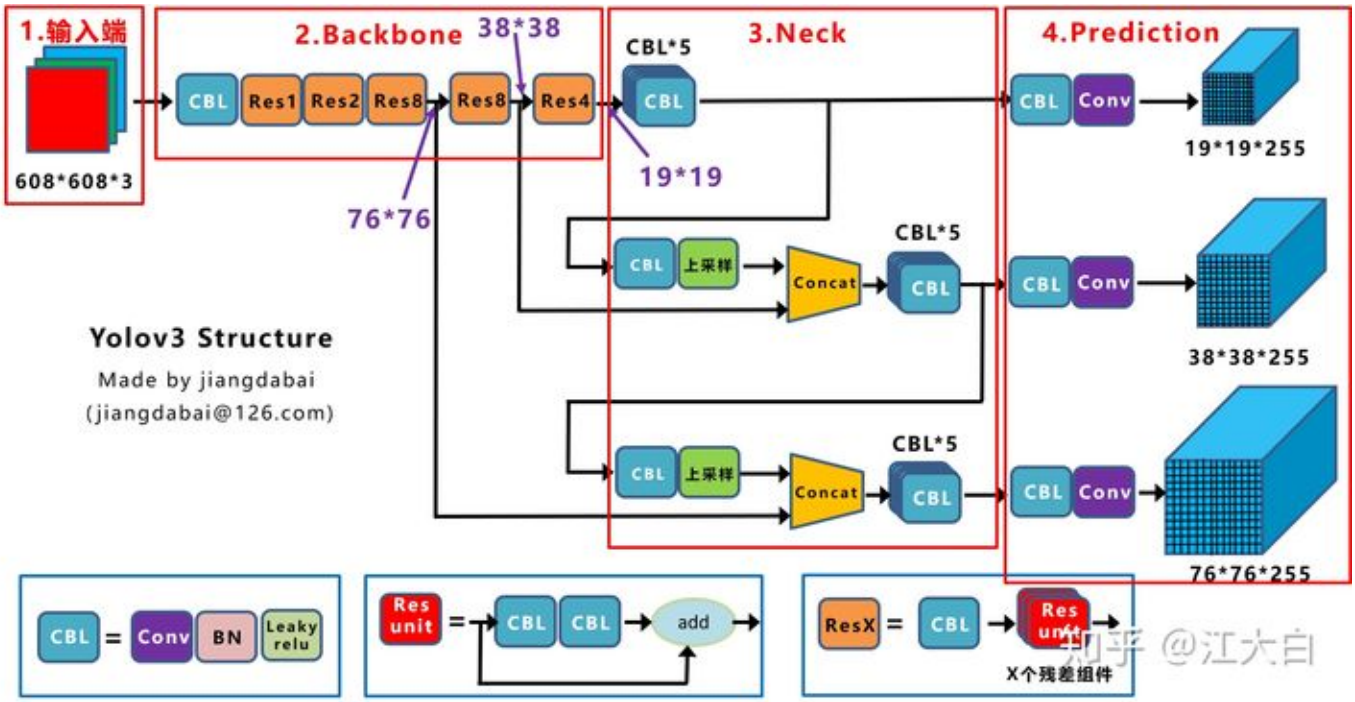
2 核心基础内容

2.1 Yolov3&Yolov4网络结构图

2.1.1 Yolov3网络结构图

Yolov3的网络结构是比较经典的**one-stage结构**，分为**输入端、Backbone、Neck和Prediction**四个部分。

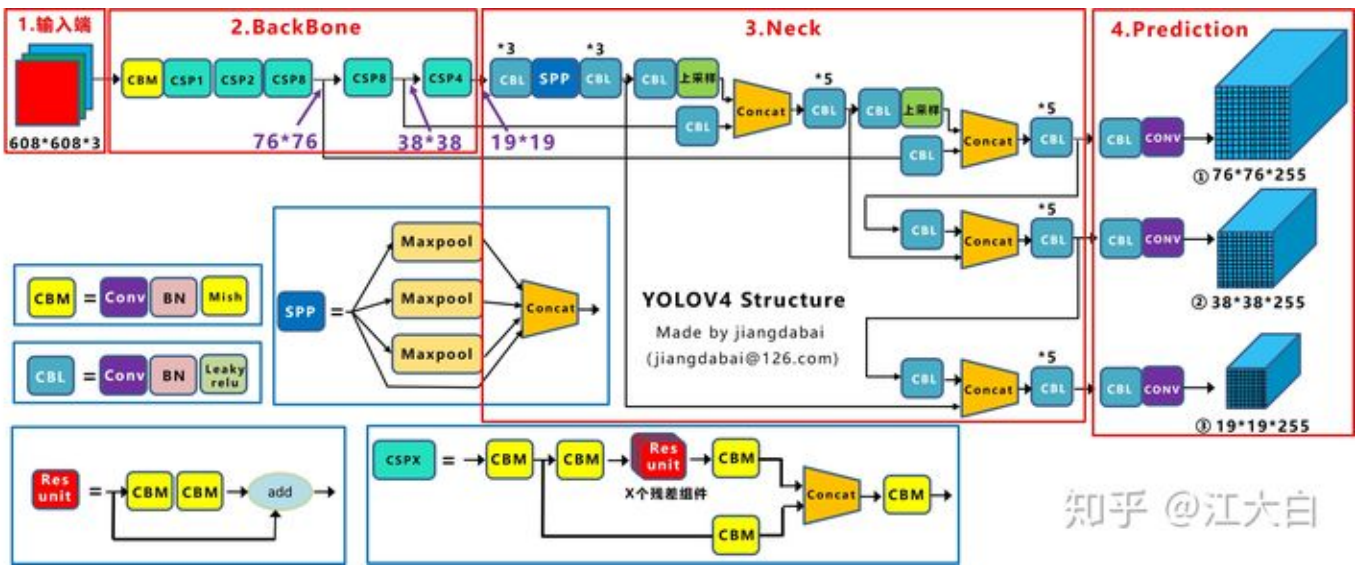
大白在之前的《深入浅出Yolo系列之Yolov3&Yolov4核心基础知识完整讲解》中讲了很多，这里不多说，还是放上绘制的**Yolov3的网络结构图**。



2.1.2 Yolov4网络结构图

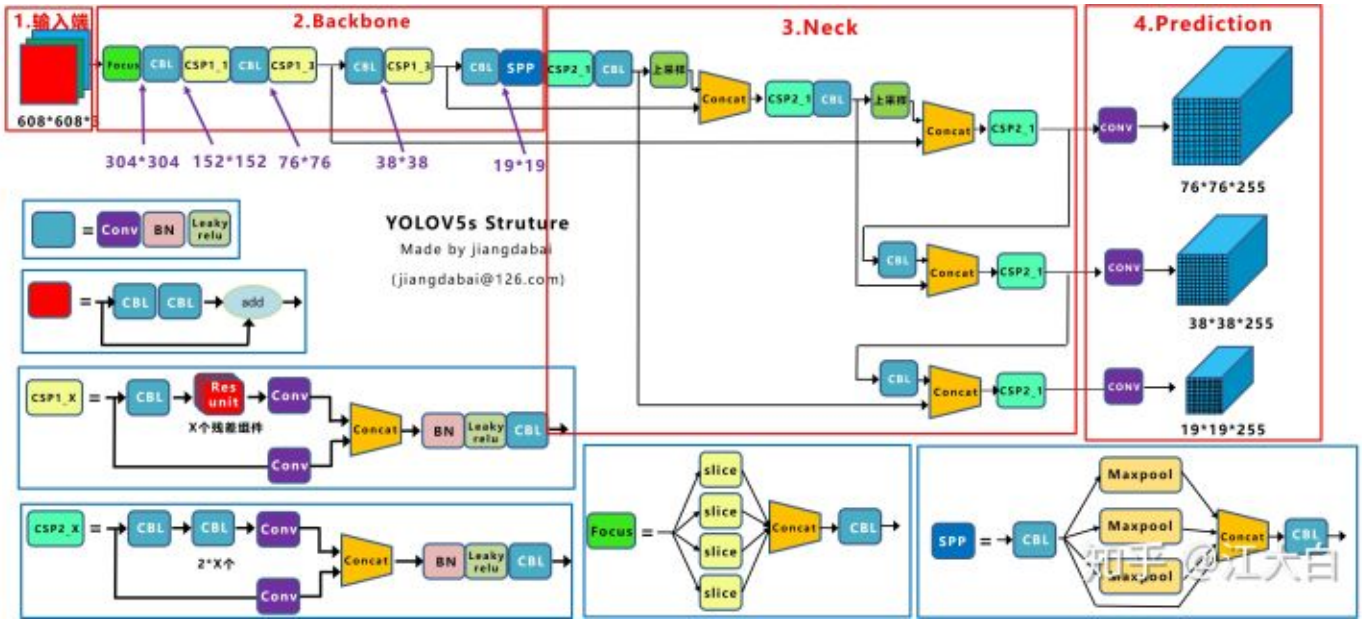
Yolov4在Yolov3的基础上进行了很多的创新。
比如**输入端**采用mosaic数据增强，
Backbone上采用了CSPDarknet53、Mish激活函数、Dropblock等方式，
Neck中采用了SPP、FPN+PAN的结构，
输出端则采用CIOU_Loss、DIOU_nms操作。

因此Yolov4对Yolov3的各个部分都进行了很多的整合创新，关于Yolov4详细的讲解还是可以参照大白之前写的《深入浅出Yolo系列之Yolov3&Yolov4核心基础知识完整讲解》，写的比较详细。



2.2 Yolov5核心基础内容

Yolov5的结构和Yolov4**很相似**，但也有一些不同，大白还是按照从整体到细节的方式，对每个板块进行讲解。

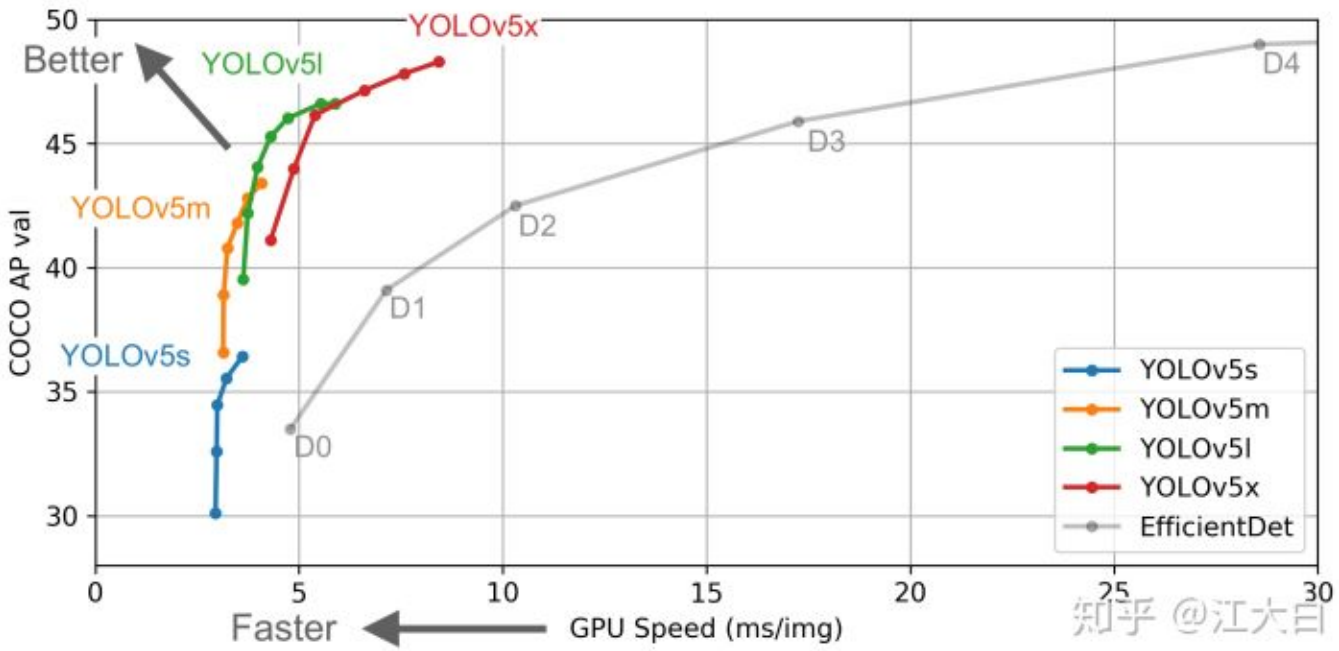


上图即**Yolov5**的网络结构图，可以看出，还是分为**输入端、Backbone、Neck、Prediction**四个部分。

大家可能对**Yolov3**比较熟悉，因此大白列举它和Yolov3的一些主要的不同点，并和Yolov4进行比较。

（1）输入端：Mosaic数据增强、自适应锚框计算、自适应图片缩放
（2）Backbone：Focus结构，CSP结构
（3）Neck：FPN+PAN结构
（4）Prediction：GIOU_Loss

下面丢上Yolov5作者的算法性能测试图：



Yolov5作者也是在COCO数据集上进行的测试，大白在之前的文章讲过，COCO数据集的小目标占比，因此最终的四种网络结构，性能上来说各有千秋。

Yolov5s网络最小，速度最少，AP精度也最低。但如果检测的以大目标为主，追求速度，倒也是个不错的选择。

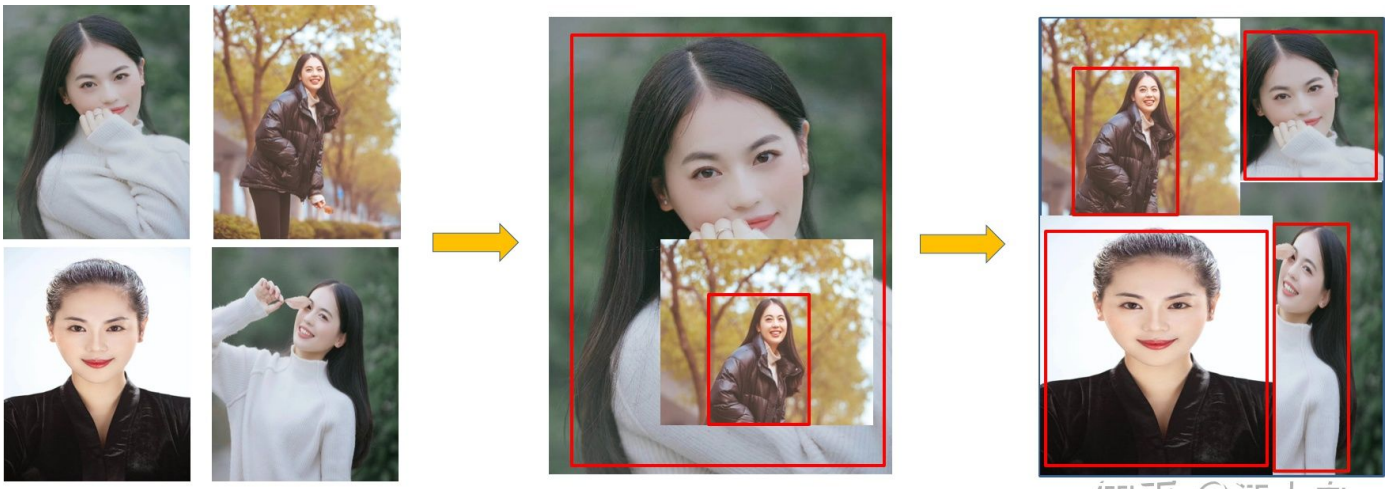
其他的三种网络，在此基础上，不断加深加宽网络，AP精度也不断提升，但速度的消耗也在不断增加。

2.2.1 输入端

（1）Mosaic数据增强

Yolov5的输入端采用了和Yolov4一样的Mosaic数据增强的方式。

Mosaic数据增强提出的作者也是来自Yolov5团队的成员，不过，**随机缩放**、**随机裁剪**、**随机排布**的方式进行拼接，对于小目标的检测效果还是很不错的。



Mosaic数据增强的内容在之前《深入浅出Yolo系列之Yolov3&Yolov4核心基础知识完整讲解》文章中写的很详细，详情可以查看之前的内容。

（2） 自适应锚框计算

在Yolo算法中，针对不同的数据集，都会有**初始设定长宽的锚框。**

在网络训练中，网络在初始锚框的基础上输出预测框，进而和**真实框groundtruth**进行比对，计算两者差距，再反向更新，**迭代网络参数。**

因此初始锚框也是比较重要的一部分，比如Yolov5在Coco数据集上初始设定的锚框：

```
anchors:
  - [116,90, 156,198, 373,326]  # P5/32
  - [30,61, 62,45, 59,119]  # P4/16
  - [10,13, 16,30, 33,23]  # P3/8
```

在Yolov3、Yolov4中，训练不同的数据集时，计算初始锚框的值是通过单独的程序运行的。

但Yolov5中将此功能嵌入到代码中，每次训练时，自适应的计算不同训练集中的最佳锚框值。

当然，如果觉得计算的锚框效果不是很好，也可以在代码中将自动计算锚框功能**关闭**。

```
parser.add_argument('--noautoanchor', action='store_true', help='disable autoanchor check')
```

控制的代码即**train.py**中上面一行代码，设置成**False**，每次训练时，不会自动计算。

（3）自适应图片缩放

在常用的目标检测算法中，不同的图片长宽都不相同，因此常用的方式是将原始图片统一缩放到一个标准尺寸，再送入检测网络中。

比如Yolo算法中常用**416*416，608*608**等尺寸，比如对下面**800*600**的图像进行缩放。

但**Yolov5代码**中对此进行了改进，也是**Yolov5推理速度**能够很快的一个不错的trick。

作者认为，在项目实际使用时，很多图片的长宽比不同，因此缩放填充后，两端的黑边大小都不同，而如果填充的比较多，则存在信息冗余，影响推理速度。

因此在Yolov5的代码中datasets.py的letterbox函数中进行了修改，对原始图像**自适应的添加最少的黑边**。

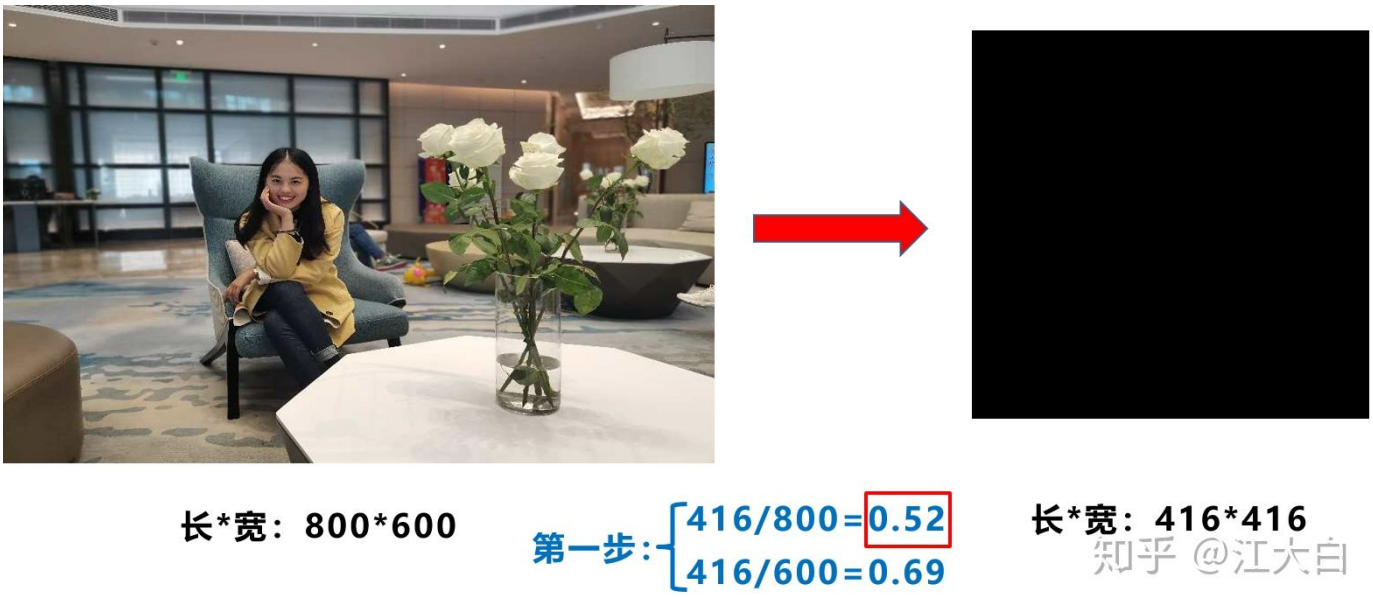
图像高度上两端的黑边变少了，在推理时，计算量也会减少，即目标检测速度会得到提升。

这种方式在之前github上Yolov3中也进行了讨论：
https://github.com/ultralytics/yolov3/issues/232

在讨论中，通过这种简单的改进，推理速度得到了37%的提升，可以说效果很明显。

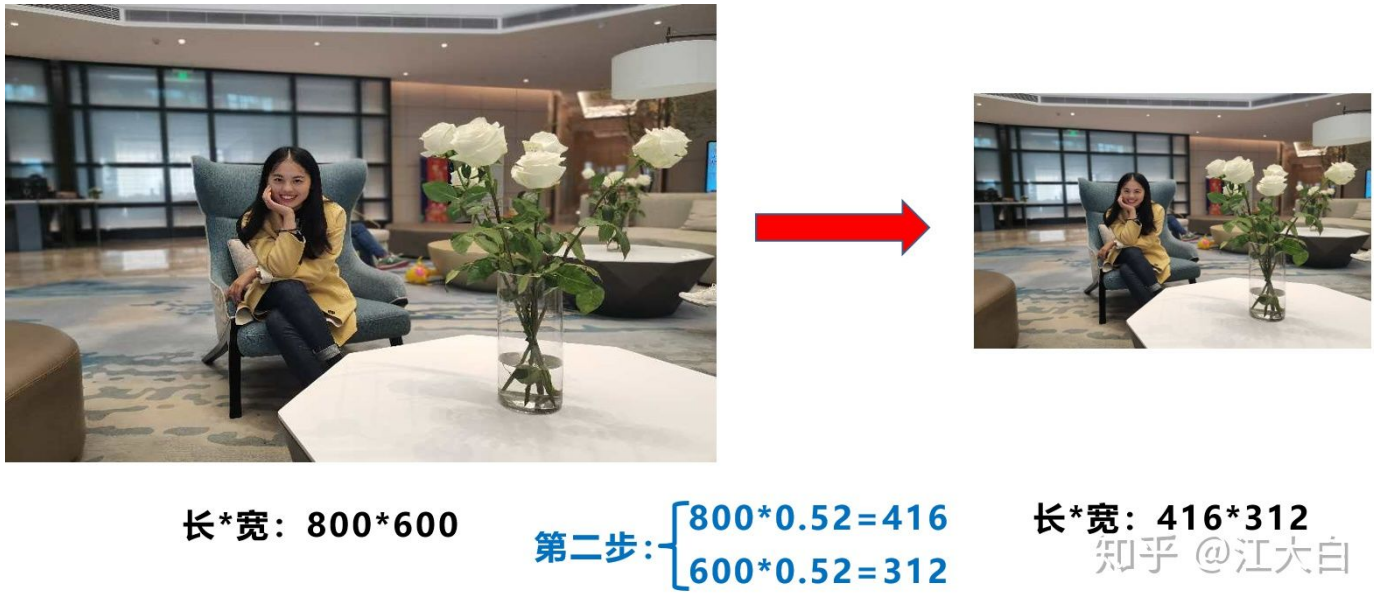
但是有的同学可能会有**大大的问号？？**如何进行计算的呢？大白按照Yolov5中的思路详细的讲解一下，在**datasets.py的letterbox函数中**也有详细的代码。

第一步：计算缩放比例



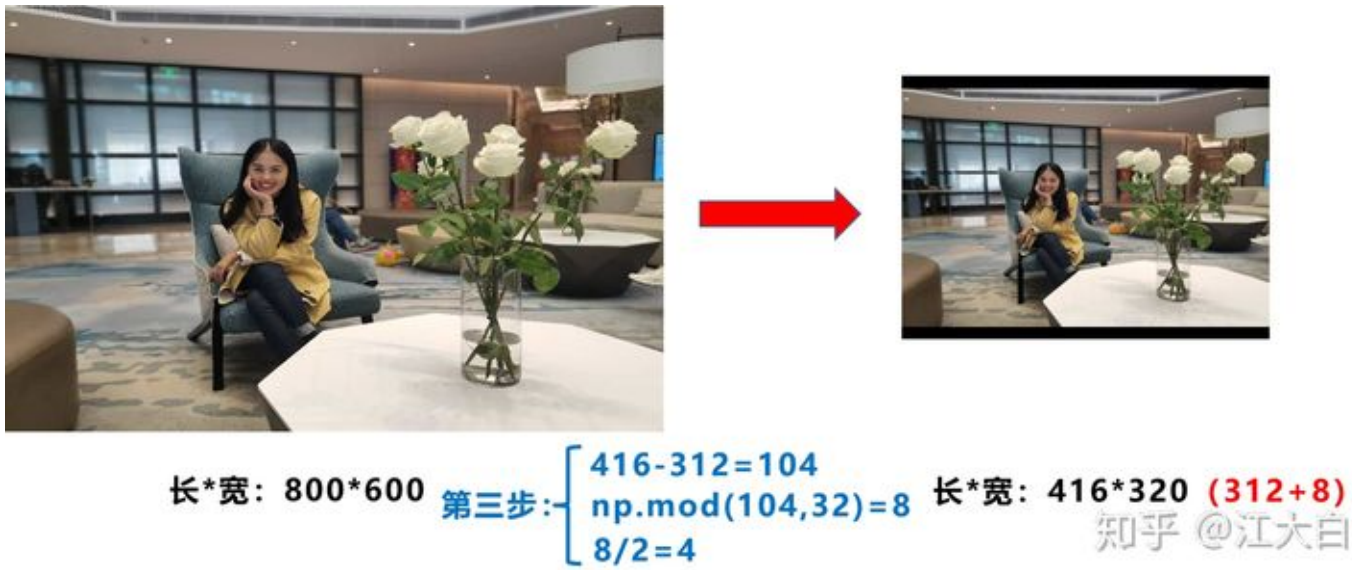
原始缩放尺寸是416*416，都除以原始图像的尺寸后，可以得到0.52，和0.69两个缩放系数，选择小的缩放系数。

第二步：计算缩放后的尺寸



原始图片的长宽都乘以最小的缩放系数0.52，宽变成了416，而高变成了312。

第三步：计算黑边填充数值



将416-312=104，得到原本需要填充的高度。再采用numpy中np.mod取余数的方式，得到8个像素，再除以2，即得到图片高度两端需要填充的数值。

此外，需要注意的是：

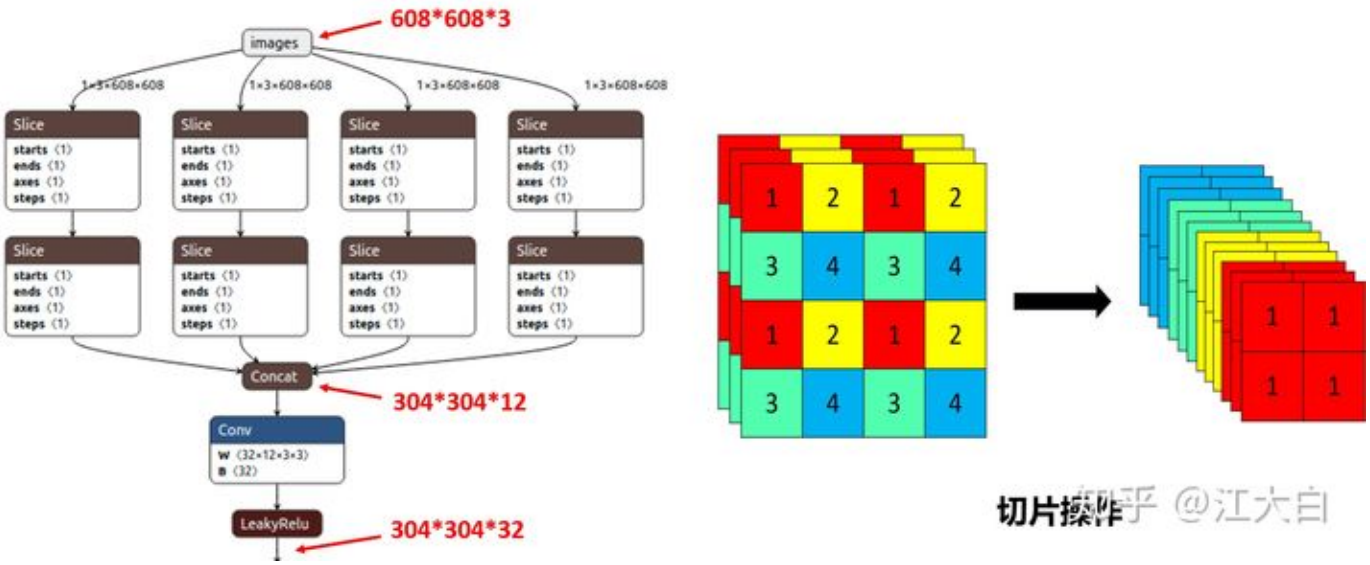
a.这里大白填充的是黑色，即 **(0，0，0)** ，而Yolov5中填充的是灰色，即 **(114,114,114)** ，都是一样的效果。

b.训练时没有采用缩减黑边的方式，还是采用传统填充的方式，即缩放到416*416大小。只是在测试，使用模型推理时，才采用缩减黑边的方式，提高目标检测，推理的速度。

c.为什么np.mod函数的后面用**32**？因为Yolov5的网络经过5次下采样，而2的5次方，等于**32**。所以至少要去掉32的倍数，再进行取余。

2.2.2 Backbone

（1）Focus结构



Focus结构，在Yolov3&Yolov4中并没有这个结构，其中比较关键是切片操作。

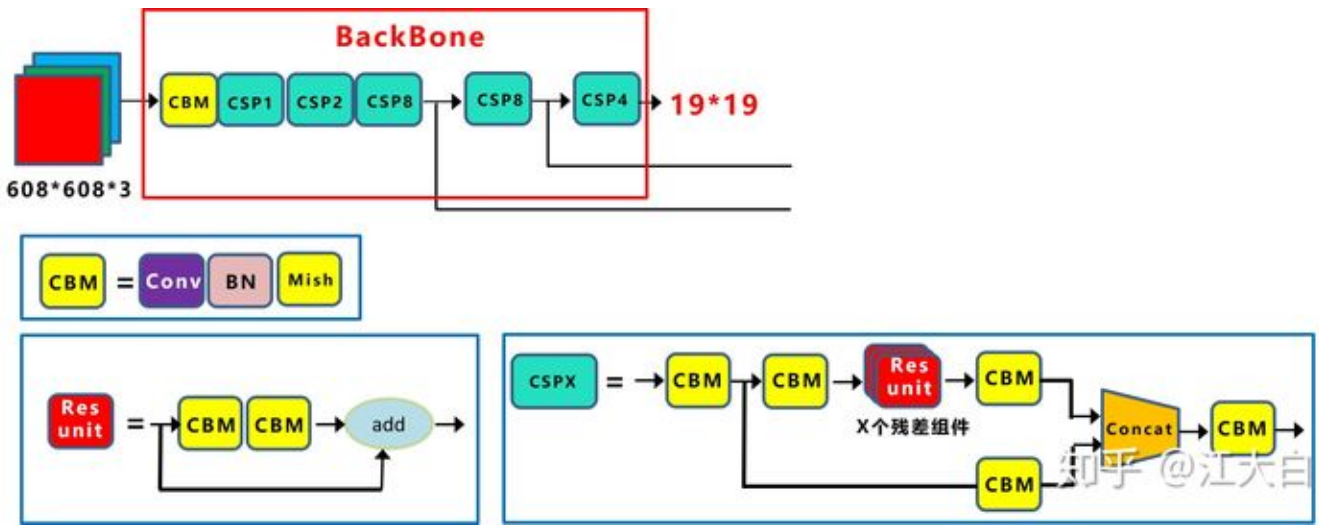
比如右图的切片示意图，4*4*3的图像切片后变成2*2*12的特征图。

以Yolov5s的结构为例，原始608*608*3的图像输入Focus结构，采用切片操作，先变成304*304*12的特征图，再经过一次32个卷积核的卷积操作，最终变成304*304*32的特征图。

需要注意的是：Yolov5s的Focus结构最后使用了32个卷积核，而其他三种结构，使用的数量有所增加，先注意下，后面会讲解到四种结构的不同点。

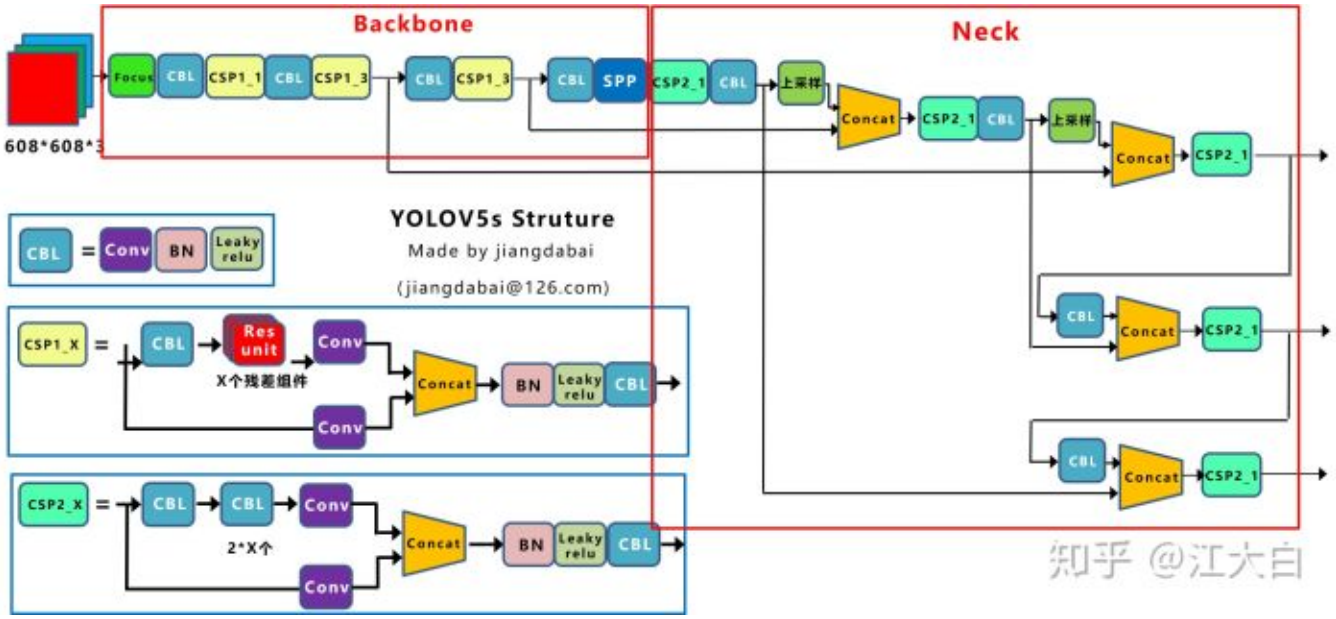
（2）CSP结构

Yolov4网络结构中，借鉴了CSPNet的设计思路，在主干网络中设计了CSP结构。



Yolov5与Yolov4不同点在于，Yolov4中只有主干网络使用了CSP结构。

而Yolov5中设计了两种CSP结构，以**Yolov5s网络**为例，**CSP1_X结构**应用于**Backbone主干网络**，另一种**CSP2_X**结构则应用于**Neck**中。

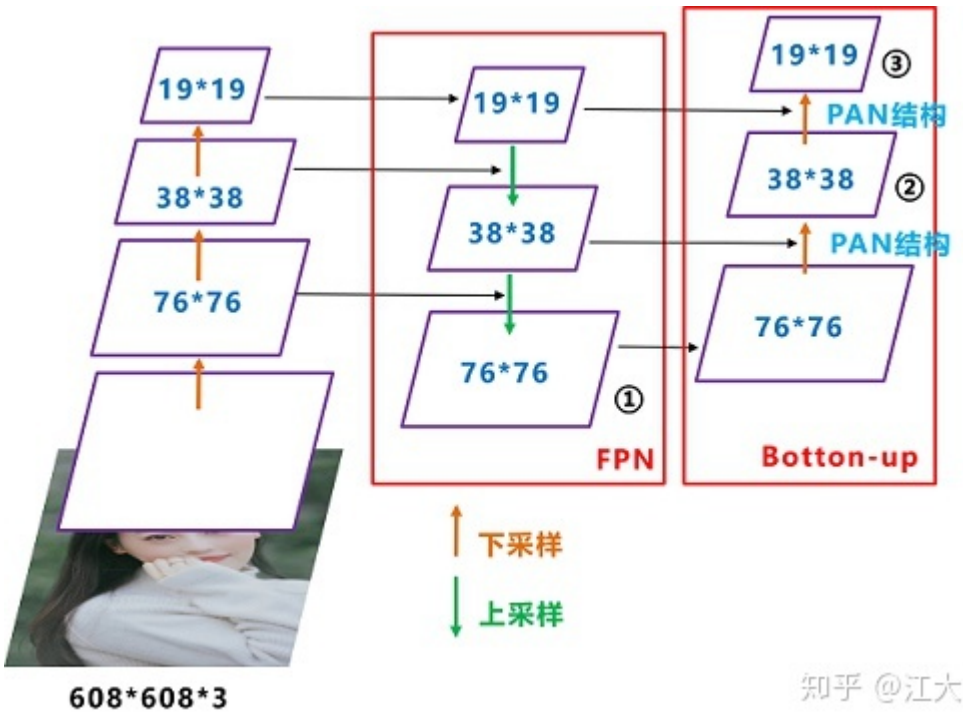


这里关于CSPNet的内容，也可以查看大白之前的《深入浅出Yolo系列之Yolov3&Yolov4核心基础完整讲解》。

2.2.3 Neck

Yolov5现在的Neck和Yolov4中一样，都采用FPN+PAN的结构，但在Yolov5刚出来时，只使用了FPN结构，后面才增加了PAN结构，此外网络中其他部分也进行了调整。

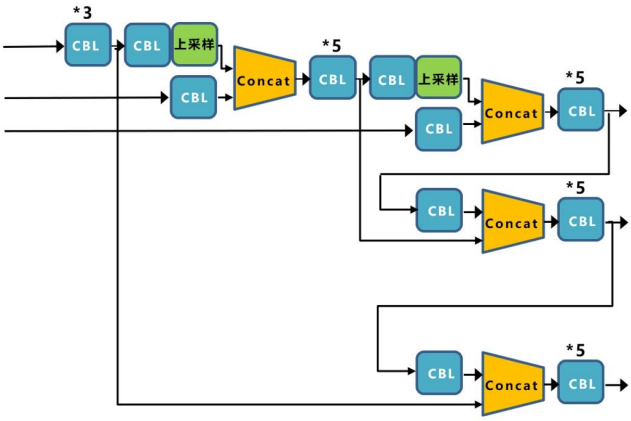
因此，大白在Yolov5刚提出时，画的很多结构图，又都重新进行了调整。



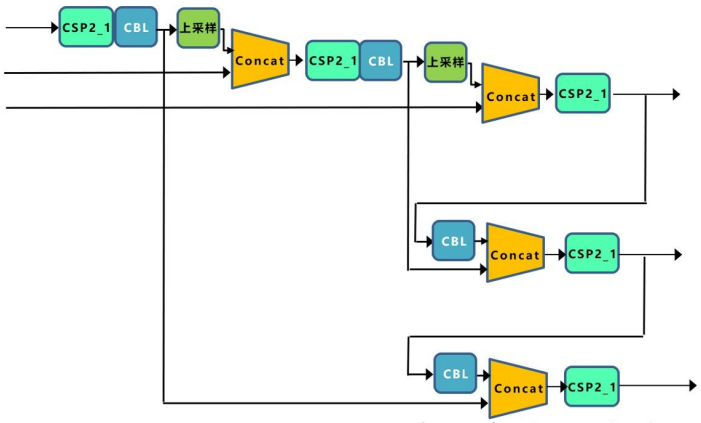
这里关于FPN+PAN的结构，大白在《深入浅出Yolo系列之Yolov3&Yolov4核心基础知识完整讲解》中，讲的很多，大家应该都有理解。

但如上面CSPNet结构中讲到，Yolov5和Yolov4的不同点在于，

Yolov4的Neck结构中，采用的都是普通的卷积操作。而Yolov5的Neck结构中，采用借鉴CSPnet设计的CSP2结构，加强网络特征融合的能力。



YOLOv4_Neck



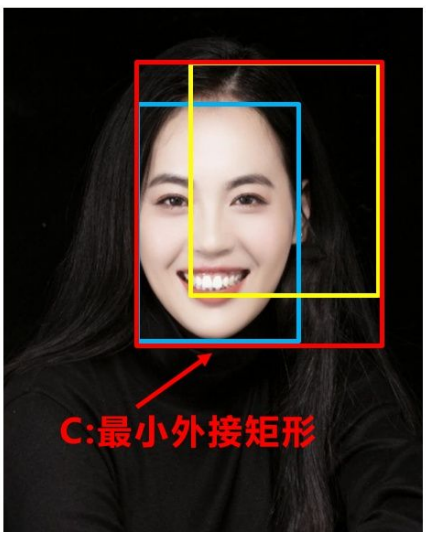
YOLOv5s_Neck

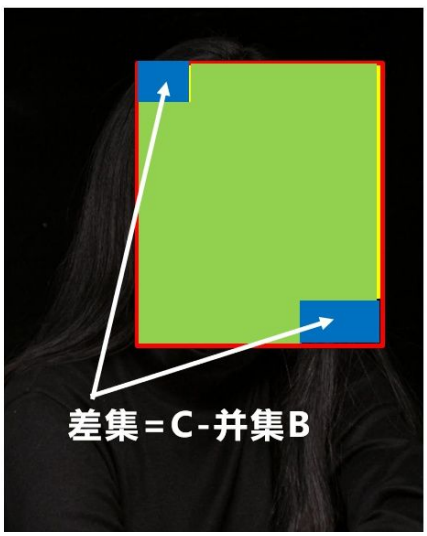
2.2.4 输出端

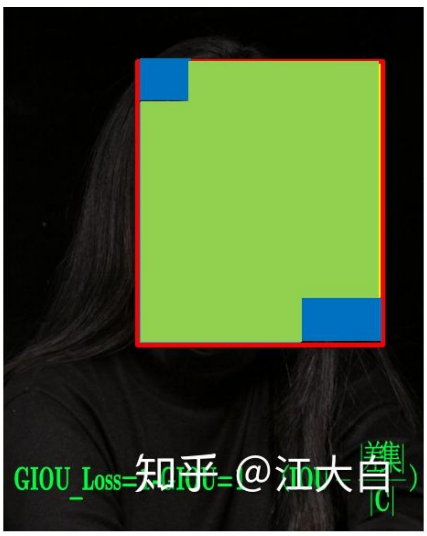
（1）Bounding box损失函数

在《深入浅出Yolo系列之Yolov3&Yolov4核心基础知识完整讲解》中，大白详细的讲解了IOU_Loss，以及进化版的GIOU_Loss，DIOU_Loss，以及CIOU_Loss。

Yolov5中采用其中的GIOU_Loss做Bounding box的损失函数。







而Yolov4中采用CIOU_Loss作为目标Bounding box的损失。

$$CIOU\_Loss=1-CIOU=1-\left(IOU-\frac{Distance\_2^2}{Distance\_C^2}-\frac{\nu^2}{(1-IOU)+\nu}\right)$$

$$\nu=\frac{4}{\pi^2}\left(\arctan\frac{w^{gt}}{h^{gt}}-\arctan\frac{w^{p}}{h^{p}}\right)^2$$

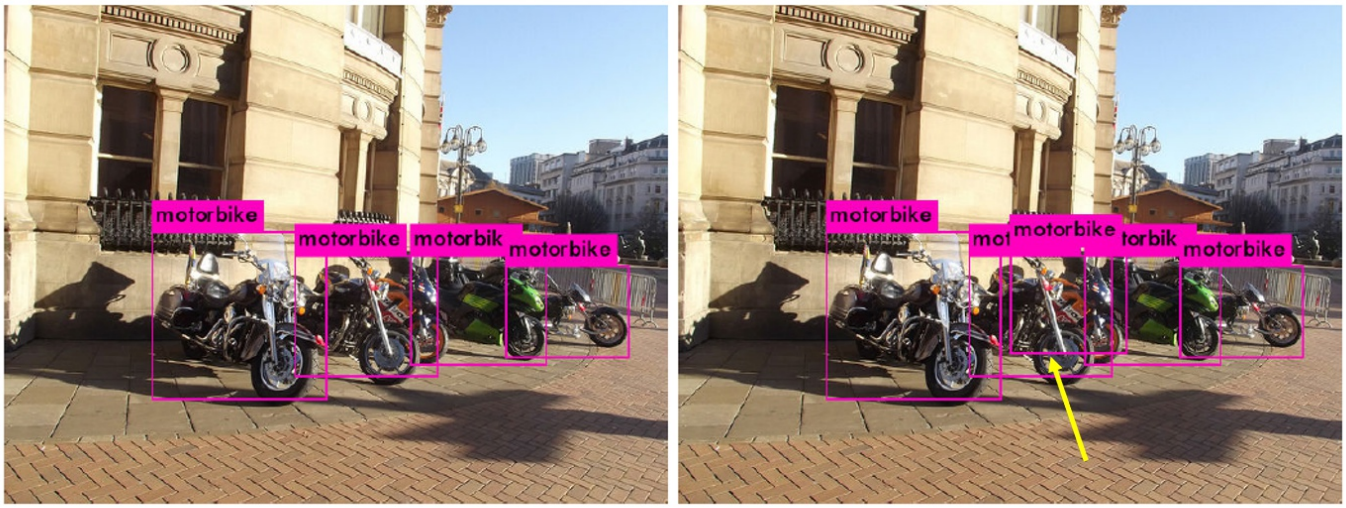
（2）nms非极大值抑制

在目标检测的后处理过程中，针对很多目标框的筛选，通常需要nms操作。

因为CIOU_Loss中包含影响因子v，涉及groudtruth的信息，而测试推理时，是没有groundtruth的。

所以Yolov4在DIOU_Loss的基础上采用DIOU_nms的方式，而Yolov5中采用加权nms的方式。

可以看出，采用DIOU_nms，下方中间箭头的黄色部分，原本被遮挡的摩托车也可以检出。



CIOU_Loss+nms　　CIOU_Loss+DIOU_nms

大白在项目中，也采用了DIOU_nms的方式，在同样的参数情况下，将nms中IOU修改成DIOU_nms。对于一些**遮挡重叠的目标**，确实会有一些改进。

比如下面**黄色箭头部分**，原本两个人重叠的部分，在参数和普通的IOU_nms一致的情况下，修改成DIOU_nms，可以将两个目标检出。

虽然大多数状态下效果差不多，但在不增加计算成本的情况下，有**稍微的改进**也是好的。

IOU_nms

DIOU_nms

2.3 Yolov5四种网络结构的不同点

Yolov5代码中的四种网络，和之前的Yolov3，Yolov4中的**cfg文件**不同，都是以**yaml**的形式来呈现。

而且四个文件的内容基本上都是一样的，只有最上方的**depth_multiple**和**width_multiple**两个参数不同，很多同学看的**一脸懵逼**，不知道只通过两个参数是如何控制四种结构的？

2.3.1 四种结构的参数

大白先取出Yolov5代码中，每个网络结构的两个参数：

（1）Yolov5s.yaml

```
depth_multiple: 0.33  # model depth multiple
width_multiple: 0.50  # layer channel multiple
```

（2）Yolov5m.yaml

```
depth_multiple: 0.67  # model depth multiple
width_multiple: 0.75  # layer channel multiple
```

（3）Yolov5l.yaml

```
depth_multiple: 1.0  # model depth multiple
width_multiple: 1.0  # layer channel multiple
```

（4）Yolov5x.yaml

```
depth_multiple: 1.33  # model depth multiple
width_multiple: 1.25  # layer channel multiple
```

四种结构就是通过上面的两个参数，来进行控制网络的**深度**和**宽度**。其中**depth_multiple**控制网络的**深度**，**width_multiple**控制网络的**宽度**。

2.3.2 Yolov5网络结构

四种结构的yaml文件中，下方的网络架构代码都是一样的。

为了便于讲解，大白将其中的Backbone部分提取出来，讲解如何控制网络的宽度和深度，yaml文件中的Head部分也是同样的原理。

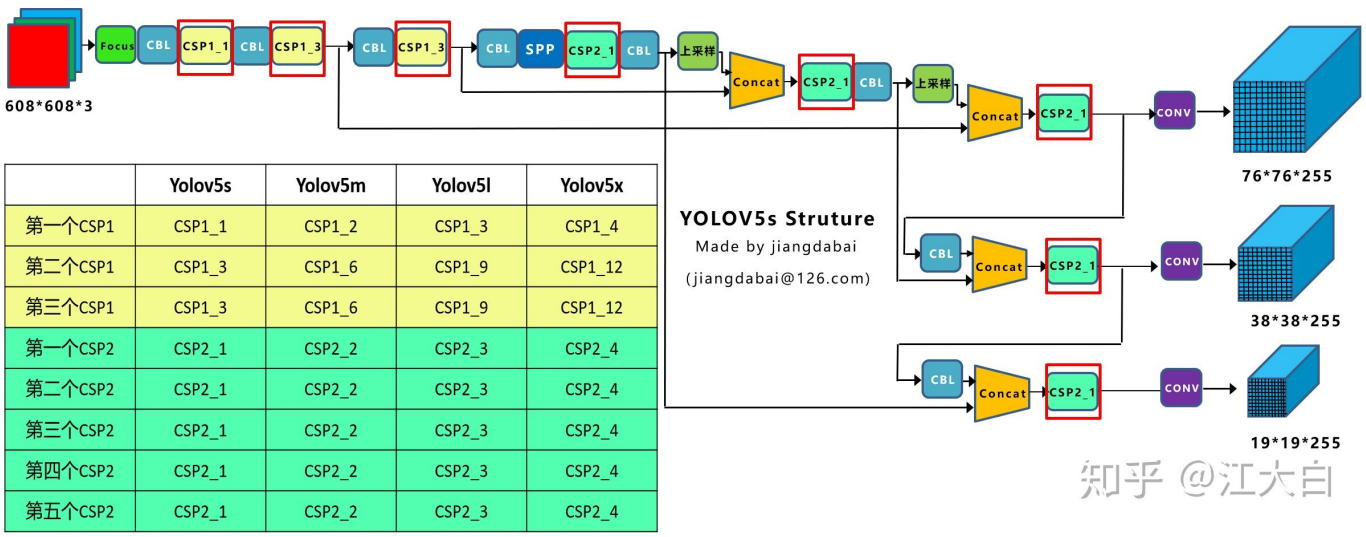
```yaml
# YOLOv5 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Focus, [64, 3]],  # 0-P1/2
   [-1, 1, Conv, [128, 3, 2]],  # 1-P2/4
   [-1, 3, BottleneckCSP, [128]],
   [-1, 1, Conv, [256, 3, 2]],  # 3-P3/8
   [-1, 9, BottleneckCSP, [256]],
   [-1, 1, Conv, [512, 3, 2]],  # 5-P4/16
   [-1, 9, BottleneckCSP, [512]],
   [-1, 1, Conv, [1024, 3, 2]], # 7-P5/32
   [-1, 1, SPP, [1024, [5, 9, 13]]],
  ]
```


在对网络结构进行解析时，yolo.py中下方的这一行代码将四种结构的**depth_multiple**，**width_multiple**提取出，赋值给**gd**，**gw**。后面主要对这**gd**，**gw**这两个参数进行讲解。

```
anchors, nc, gd, gw = d['anchors'], d['nc'], d['depth_multiple'], d['width_multiple']
```

下面再细致的剖析下，看是如何控制每种结构，深度和宽度的。

2.3.3 Yolov5四种网络的深度



	Yolov5s	Yolov5m	Yolov5l	Yolov5x
第一个CSP1	CSP1_1	CSP1_2	CSP1_3	CSP1_4
第二个CSP1	CSP1_3	CSP1_6	CSP1_9	CSP1_12
第三个CSP1	CSP1_3	CSP1_6	CSP1_9	CSP1_12
第一个CSP2	CSP2_1	CSP2_2	CSP2_3	CSP2_4
第二个CSP2	CSP2_1	CSP2_2	CSP2_3	CSP2_4
第三个CSP2	CSP2_1	CSP2_2	CSP2_3	CSP2_4
第四个CSP2	CSP2_1	CSP2_2	CSP2_3	CSP2_4
第五个CSP2	CSP2_1	CSP2_2	CSP2_3	CSP2_4

（1）不同网络的深度

在上图中，大白画了两种CSP结构，CSP1和CSP2，其中CSP1结构主要应用于Backbone中，CSP2结构主要应用于Neck中。

需要注意的是，四种网络结构中每个CSP结构的深度都是不同的。

a.以yolov5s为例，第一个CSP1中，使用了1个残差组件，因此是**CSP1_1**。而在Yolov5m中，则增加了网络的深度，在第一个CSP1中，使用了2个残差组件，因此是**CSP1_2**。

而Yolov5l中，同样的位置，则使用了**3个残差组件**，Yolov5x中，使用了**4个残差组件**。

其余的第二个CSP1和第三个CSP1也是同样的原理。

b.在第二种CSP2结构中也是同样的方式，以第一个CSP2结构为例，Yolov5s组件中使用了2×X=2×1=2个卷积，因为X=1，所以使用了1组卷积，因此是**CSP2_1**。

而Yolov5m中使用了2**组**，Yolov5l中使用了3**组**，Yolov5x中使用了4**组**。

其他的四个CSP2结构，也是同理。

Yolov5中，网络的不断加深，也在不断**增加网络特征提取**和**特征融合**的能力。

（2）控制深度的代码

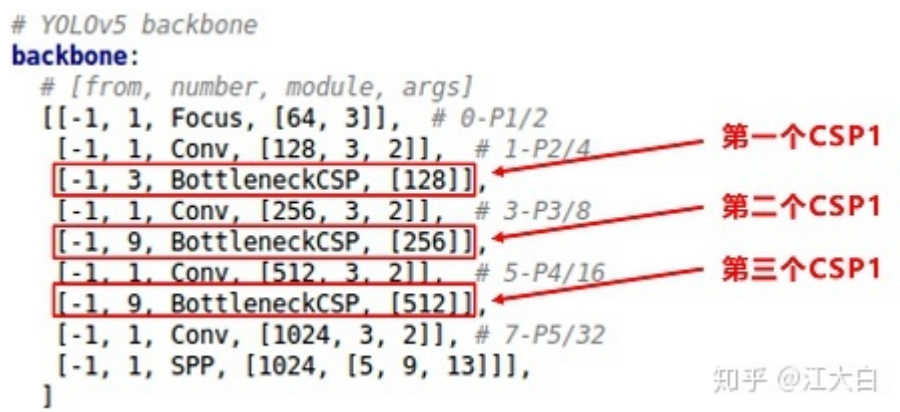
控制四种网络结构的核心代码是**yolo.py**中下面的代码，存在两个变量，**n和gd**。

我们再将**n和gd**带入计算，看每种网络的变化结果。

```
n = max(round(n * gd), 1) if n > 1 else n  # depth gain
```

（3）验证控制深度的有效性

我们选择**最小的yolov5s.yaml**和中间的**yolov5l.yaml**两个网络结构，将**gd(height_multiple)**系数带入，看是否正确。




```
# YOLOv5 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Focus, [64, 3]],  # 0-P1/2
   [-1, 1, Conv, [128, 3, 2]],  # 1-P2/4
   [-1, 3, BottleneckCSP, [128]],
   [-1, 1, Conv, [256, 3, 2]],  # 3-P3/8
   [-1, 9, BottleneckCSP, [256]],
   [-1, 1, Conv, [512, 3, 2]],  # 5-P4/16
   [-1, 9, BottleneckCSP, [512]],
   [-1, 1, Conv, [1024, 3, 2]], # 7-P5/32
   [-1, 1, SPP, [1024, [5, 9, 13]]],
  ]
```

a. yolov5s.yaml

其中**height_multiple=0.33**，即**gd=0.33**，而n则由上面红色框中的信息获得。

以上面网络框图中的第一个CSP1为例，即上面的第一个红色框。n等于第二个数值3。

而**gd=0.33**，带入（2）中的计算代码，结果n=1。因此第一个CSP1结构内只有1个残差组件，即CSP1_1。

第二个CSP1结构中，n等于第二个数值9，而**gd=0.33**，带入（2）中计算，结果**n=3**，因此第二个CSP1结构中有3个残差组件，即CSP1_3。

第三个CSP1结构也是同理，这里不多说。

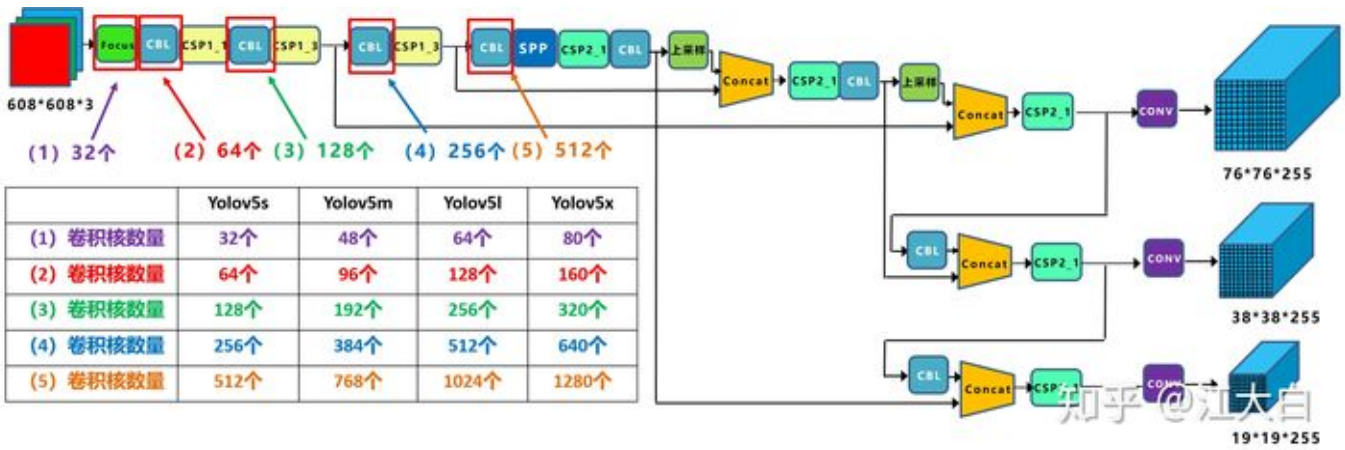
b. yolov5l.xml

其中**height_multiple=1**，即**gd=1**

和上面的计算方式相同，第一个CSP1结构中，n=3，带入代码中，结果n=3，因此为CSP1_3。

下面第二个CSP1和第三个CSP1结构都是同样的原理。

2.3.4 Yolov5四种网络的宽度



	Yolov5s	Yolov5m	Yolov5l	Yolov5x
(1) 卷积核数量	32个	48个	64个	80个
(2) 卷积核数量	64个	96个	128个	160个
(3) 卷积核数量	128个	192个	256个	320个
(4) 卷积核数量	256个	384个	512个	640个
(5) 卷积核数量	512个	768个	1024个	1280个

（1）不同网络的宽度:

如上图表格中所示，四种yolov5结构在不同阶段的卷积核的数量都是不一样的，因此也直接影响卷积后特征图的第三维度，即**厚度**，大白这里表示为网络的**宽度**。

a.以Yolov5s结构为例，第一个Focus结构中，最后卷积操作时，卷积核的数量是32个，因此经过**Focus结构**，特征图的大小变成**304*304*32**。

而yolov5m的**Focus结构**中的卷积操作使用了48个卷积核，因此**Focus结构**后的特征图变成3**04*304*48**。yolov5l，yolov5x也是同样的原理。

b. 第二个卷积操作时，yolov5s使用了64个卷积核，因此得到的特征图是**152*152*64**。而yolov5m使用96个特征图，因此得到的特征图是**152*152*96**。yolov5l，yolov5x也是同理。

c. 后面三个卷积下采样操作也是同样的原理，这样大白不过多讲解。

四种不同结构的卷积核的数量不同，这也直接影响网络中，比如**CSP1，CSP2等结构**，以及各个普通卷积，卷积操作时的卷积核数量也同步在调整，影响整体网络的计算量。

大家最好可以将结构图和前面第一部分四个网络的特征图链接，对应查看，思路会更加清晰。

当然卷积核的数量越多，特征图的厚度，即**宽度越宽**，网络提取特征的**学习能力也越强。**

（2）控制宽度的代码

在yolov5的代码中，控制宽度的核心代码是**yolo.py**文件里面的这一行：

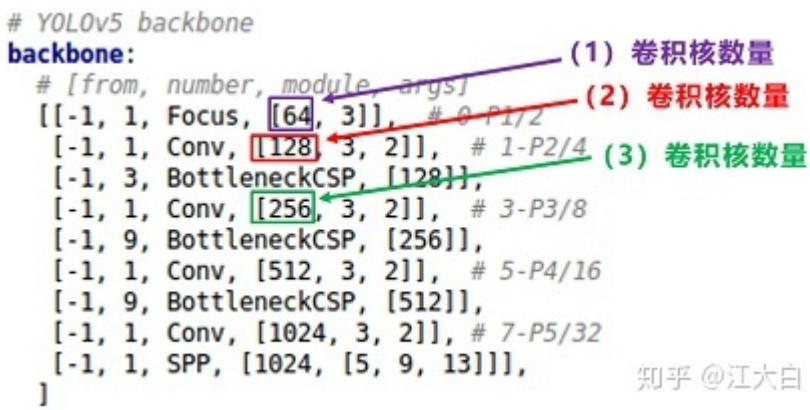
```
c2 = make_divisible(c2 * gw, 8) if c2 != no else c2
```

它所调用的子函数**make_divisible**的功能是：

```
def make_divisible(x, divisor):
    # Returns x evenly divisble by divisor
    return math.ceil(x / divisor) * divisor
```

（3）验证控制宽度的有效性

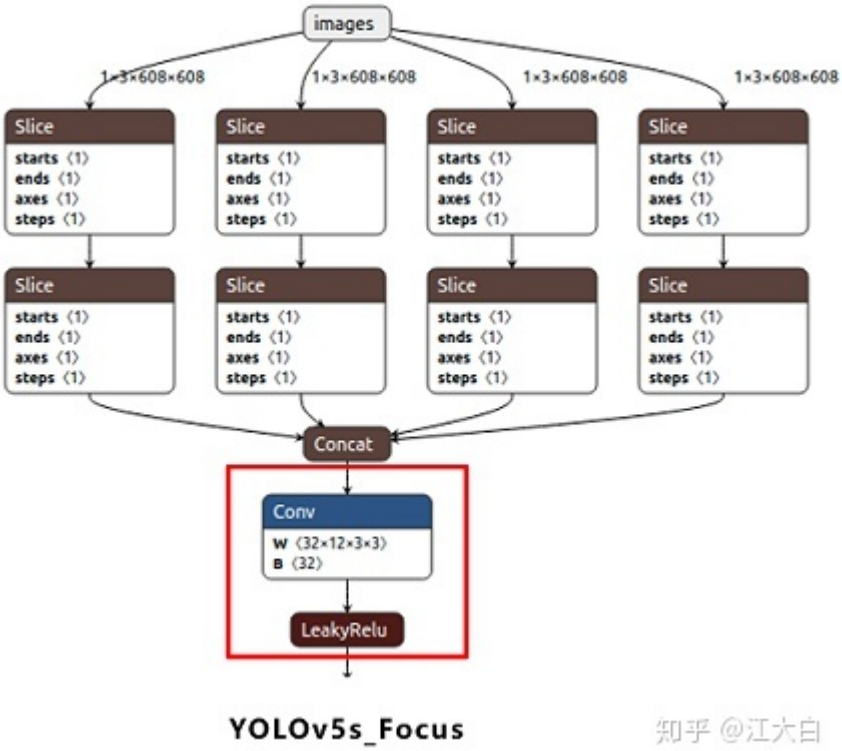
我们还是选择**最小的yolov5s**和**中间的yolov5l**两个网络结构，将**width_multiple**系数带入，看是否正确。




```
# YOLOv5 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Focus, [64, 3]],  # 0-P1/2
   [-1, 1, Conv, [128, 3, 2]],  # 1-P2/4
   [-1, 3, BottleneckCSP, [128]],
   [-1, 1, Conv, [256, 3, 2]],  # 3-P3/8
   [-1, 9, BottleneckCSP, [256]],
   [-1, 1, Conv, [512, 3, 2]],  # 5-P4/16
   [-1, 9, BottleneckCSP, [512]],
   [-1, 1, Conv, [1024, 3, 2]], # 7-P5/32
   [-1, 1, SPP, [1024, [5, 9, 13]]],
  ]
```

a. yolov5s.yaml

其中**width_multiple=0.5**，即**gw=0.5。**



以第一个卷积下采样为例，即Focus结构中下面的卷积操作。

按照上面Backbone的信息，我们知道Focus中，标准的c2=64，而**gw=0.5**，代入（2）中的计算公式，最后的结果=32。即Yolov5s的Focus结构中，卷积下采样操作的卷积核数量为**32个。**

再计算后面的第二个卷积下采样操作，标准c2的值=128，**gw=0.5**，代入（2）中公式，最后的结果=64，也是正确的。

b. yolov5l.yaml

其中**width_multiple=1**，即**gw=1**，而标准的**c2=64**，代入上面（2）的计算公式中，可以得到Yolov5l的Focus结构中，卷积下采样操作的卷积核的数量为64个，而第二个卷积下采样的卷积核数量是128个。

另外的三个卷积下采样操作，以及**yolov5m，yolov5x结构**也是同样的计算方式，大白这里不过多解释。

3 Yolov5相关论文及代码

3.1 代码

Yolov5的作者并没有发表论文，因此只能从代码角度进行分析。

Yolov5代码：github.com/ultralytics/...

大家可以根据网页的说明，下载训练，及测试，流程还是比较简单的。

3.2 相关论文

另外一篇论文，**PP-Yolo**，在Yolov3的原理上，采用了很多的**tricks调参方式**，也挺有意思。

感兴趣的话可以参照另一个博主的文章：点击查看

4 小目标分割检测

目标检测发展很快，但对于**小目标的检测**还是有一定的瓶颈，特别是**大分辨率图像小目标检测**。比如**7920*2160**，甚至**16000*16000**的图像。

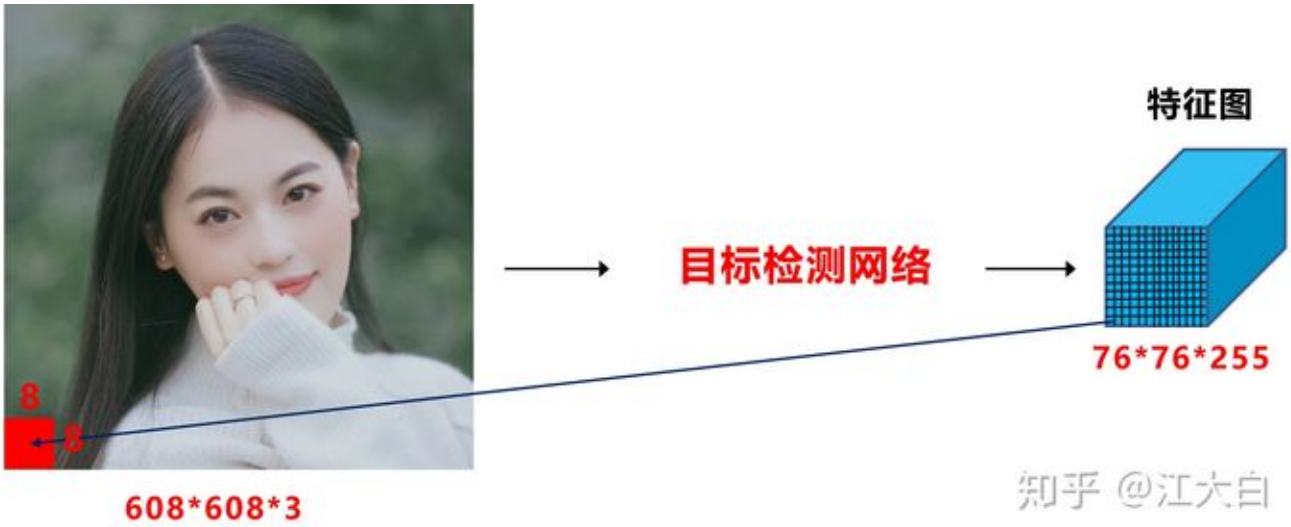
图像的分辨率很大，但又有很多小的目标需要检测。但是如果直接输入检测网络，比如yolov3，检出效果并不好。

主要原因是：

（1）小目标尺寸

以网络的输入608*608为例，yolov3、yolov4，yolov5中下采样都使用了5次，因此最后的特征图大小是**19*19，38*38，76*76。**

三个特征图中，最大的76*76负责检测小目标，而对应到608*608上，每格特征图的感受野是**608/76=8*8**大小。



再将608*608对应到7680*2160上，以最长边7680为例，7680/608*8=101。

即如果原始图像中目标的宽或高小于101像素，网络很难学习到目标的特征信息。

（PS：这里忽略多尺度训练的因素及增加网络检测分支的情况）

（2）高分辨率

而在很多遥感图像中，长宽比的分辨率比7680*2160更大，比如上面的16000*16000，如果采用直接输入原图的方式，很多小目标都无法检测出。

（3）显卡爆炸

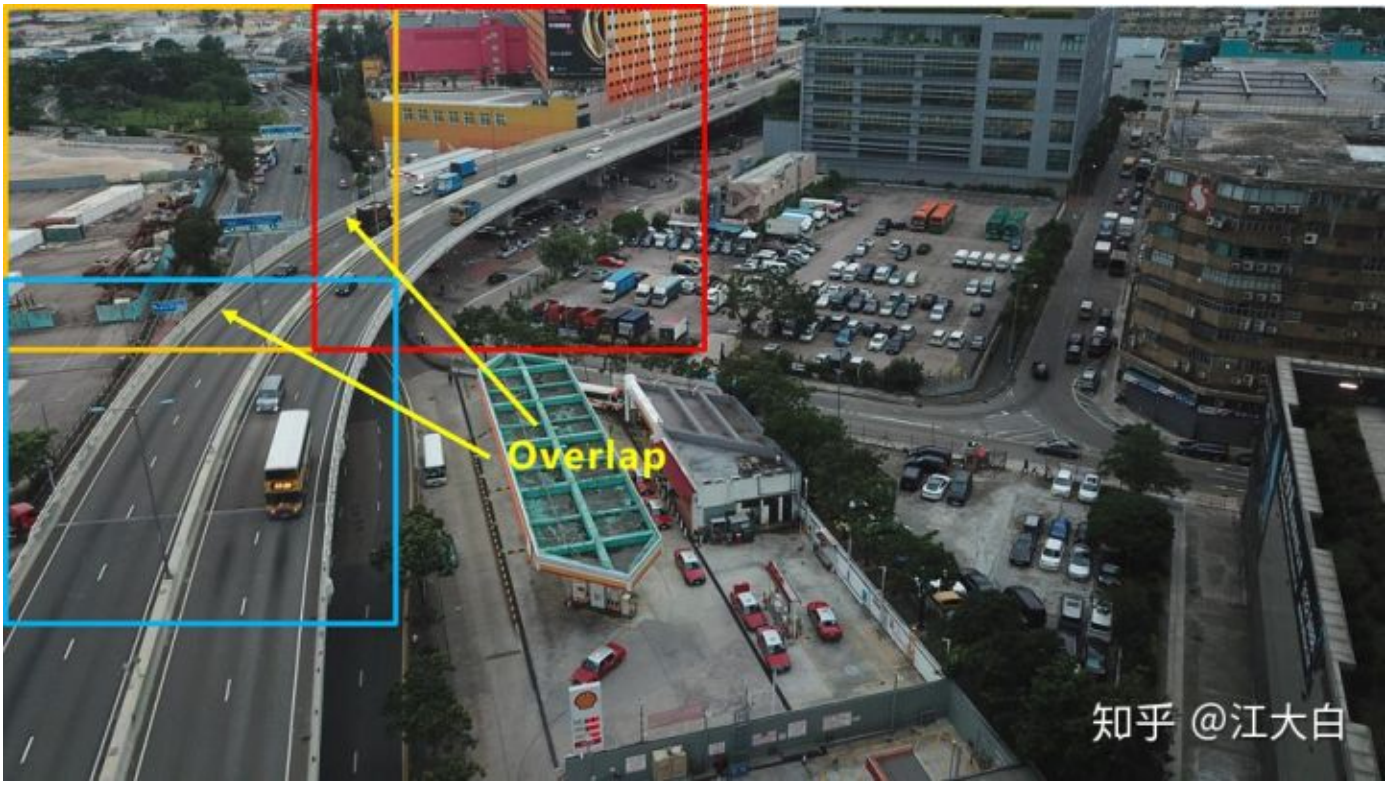
很多图像分辨率很大，如果简单的进行下采样，下采样的倍数太大，容易丢失数据信息。

但是倍数太小，网络前向传播需要在内存中保存大量的特征图，极大耗尽GPU资源,很容易发生**显存爆炸**，无法正常的训练及推理。

因此可以借鉴2018年YOLT算法的方式，改变一下思维，对大分辨率图片先进行分割，变成一张张小图，再进行检测。

需要注意的是：

为了避免两张小图之间，一些目标正好被分割截断，所以两个小图之间设置**overlap重叠区域**，比如分割的小图是**960*960**像素大小，则overlap可以设置为**960*20%=192**像素。



每个小图检测完成后，再将所有的框放到大图上，对大图整体做一次**nms操作**，将重叠区域的很多重复框去除。

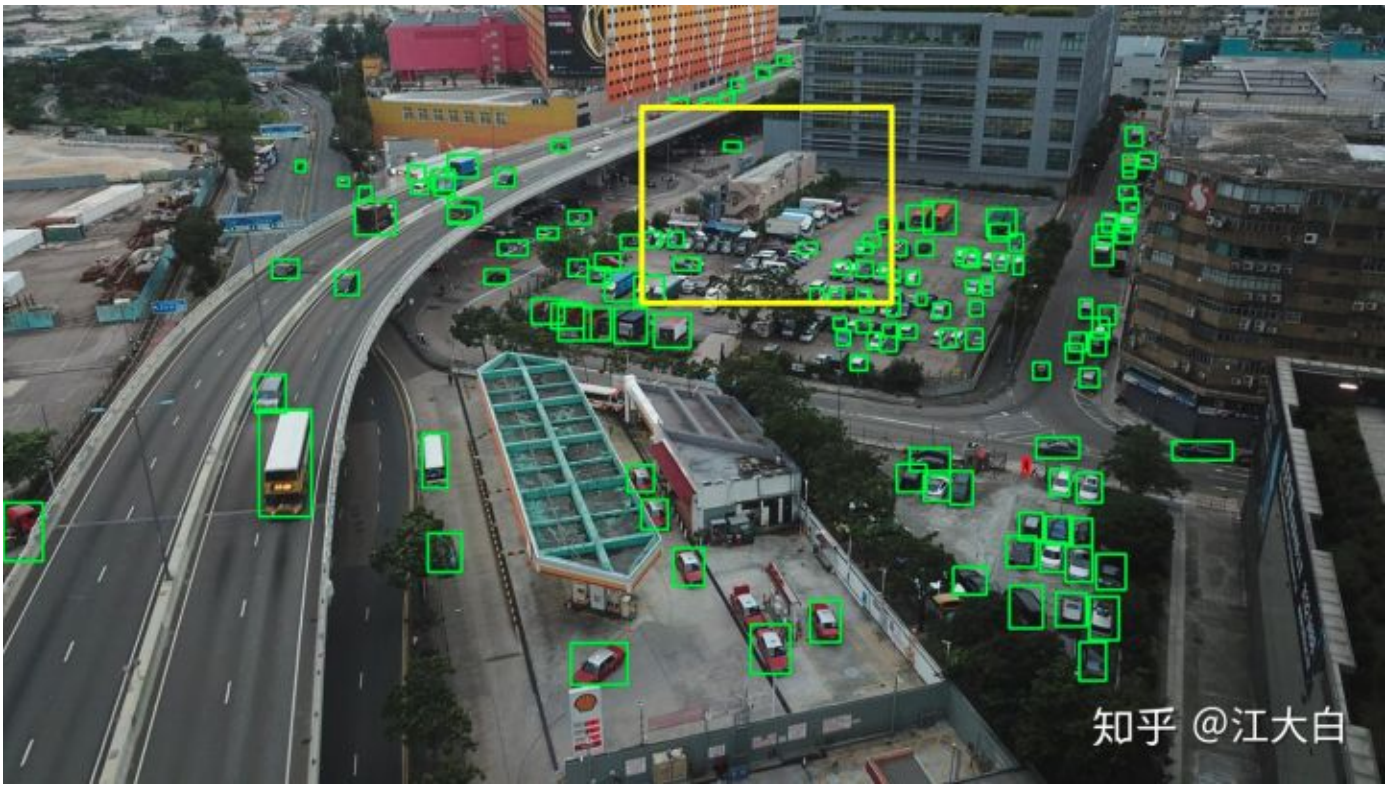
这样操作，可以将很多小目标检出，比如**16000*16000像素**的遥感图像。

注意：这里关于小图检测后，放到大图上的方法，发现评论中，很多的同学可能想的过于复杂了，采用的方式，其实按照在大图上裁剪的位置，直接回归到大图即可。：

此外，国内还有一个10亿像素图像目标检测的比赛，也是用的这样的方式，大白将其中一个讲解不错的视频，也放到这个，大家可以**点击查看**。

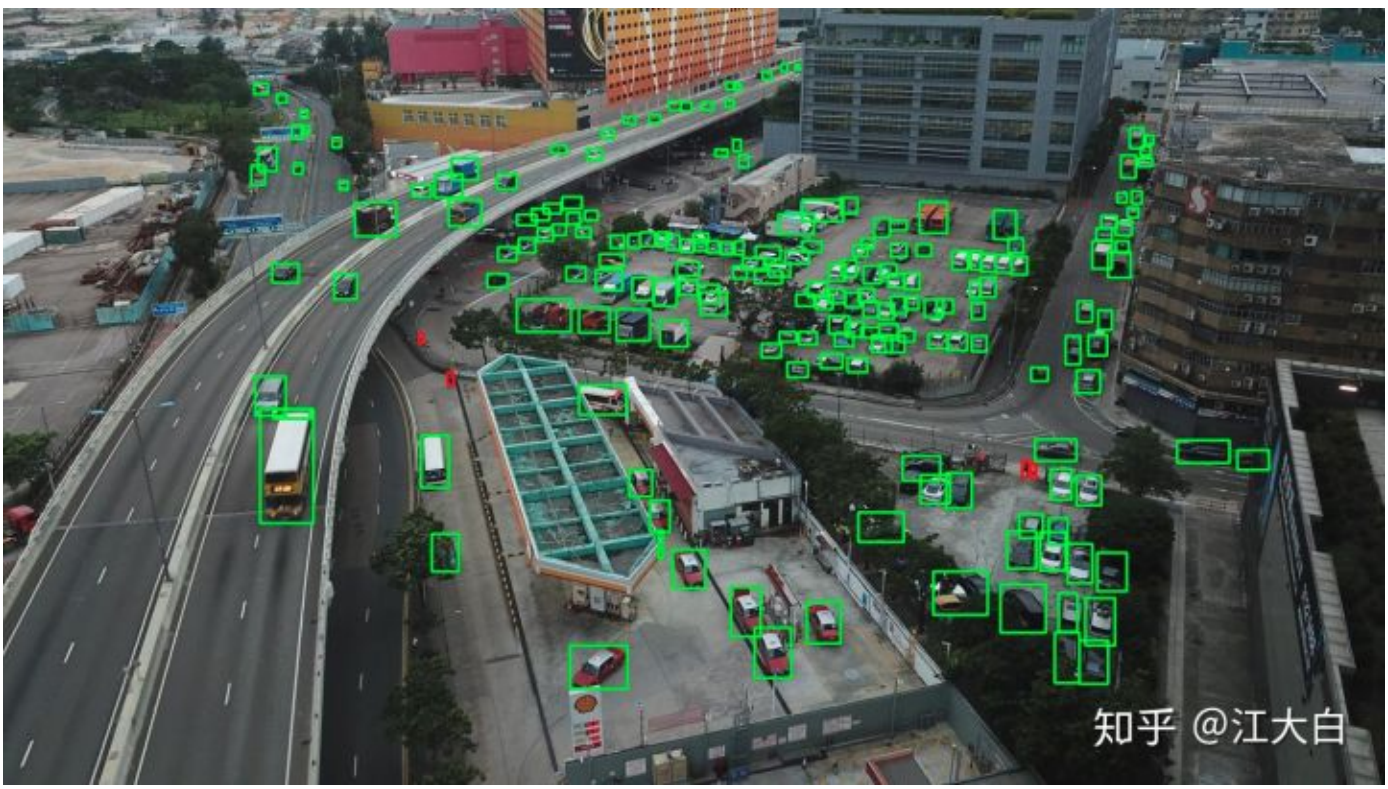
无人机视角下，也有很多小的目标。大白也进行了测试，效果还是不错的。

比如下图是将**原始大图->416*416**大小，直接使用目标检测网络输出的效果：



可以看到**中间黄色框区域**，很多汽车检测漏掉。

再使用分割的方式，将大图**先分割成小图**，**再对每个小图检测**，可以看出中间区域很多的汽车都被检测出来：



不过这样的方式有优点也有缺点：

优点：

（1）准确性

分割后的小图，再输入目标检测网络中，对于**最小目标像素的下限**会大大降低。

比如分割成608*608大小，送入输入图像大小608*608的网络中，按照上面的计算方式，原始图片上，长宽**大于8个像素**的小目标都可以学习到特征。

（2）检测方式

在大分辨率图像，比如遥感图像，或者无人机图像，如果无需考虑实时性的检测，且对**小目标检测**也有需求的项目，可以尝试此种方式。

缺点：

（1）增加计算量

比如原本7680*2160的图像，如果使用直接大图检测的方式，一次即可检测完。

但采用分割的方式，切分成N张608*608大小的图像，再进行N次检测，会大大增加检测时间。

借鉴Yolov5的四种网络方式，我们可以采用**尽量轻**的网络，比如Yolov5s网络结构或者更轻的网络。

当然Yolov4和Yolov5的网络各有优势，我们也可以借鉴Yolov5的设计方式，对Yolov4进行**轻量化改造**，或者进行**剪枝**。

5 后语

综合而言，在实际测试中，Yolov4的准确性有不错的优势，但Yolov5的多种网络结构使用起来更加灵活，我们可以根据不同的项目需求，**取长补短**，发挥不同检测网络的优势。

希望在人工智能的道路上，**和大家共同进步**。

编辑于 11 小时前