

YOLO v5

YOLOv5

by



Download on the
App Store



Coming Soon on
Google Play

一、 yolo v5解读

- (一) 、几种数据增强方法
- (二) 、backbone网络细节
- (三) 、边框预测细节
- (四) 、正样本采样细节
- (五) 、损失计算细节

二、 yolo v5训练

- (一) 、打标自己的数据
- (二) 、训练自己的yolov5
- (三) 、保存模型，导出onnx， yolov5-lite

三、 yolo v5复现

- (一) 、backbone网络复现
- (二) 、损失计算复现
- (三) 、训练自己的数据
- (四) 、扩展yolov5支持关键点检测
- (五) 、扩展yolov5支持实例分割

一、yolo v5解读

(一) 几种数据增强方法

rectangular: 同个batch里做rectangle宽高等比变换，加快训练

hsv_h: 0.015 # image HSV-Hue augmentation (fraction), 色调

hsv_s: 0.7 # image HSV-Saturation augmentation (fraction), 饱和度

hsv_v: 0.4 # image HSV-Value augmentation (fraction), 曝光度

degrees: 0.0 # image rotation (+/- deg), 旋转

translate: 0.1 # image translation (+/- fraction), 平移

scale: 0.5 # image scale (+/- gain), 缩放

shear: 0.0 # image shear (+/- deg), 错切/非垂直投影

perspective: 0.0 # image perspective (+/- fraction), range 0-0.001, 透视变换

flipud: 0.0 # image flip up-down (probability), 上下翻转

fliplr: 0.5 # image flip left-right (probability), 左右翻转

mosaic: 1.0 # image mosaic (probability), 4图拼接

mixup: 0.0 # image mixup (probability), 图像互相融合

copy_paste: 0.0 # segment copy-paste (probability), 分割填补

最后box坐标转换，segment坐标转换

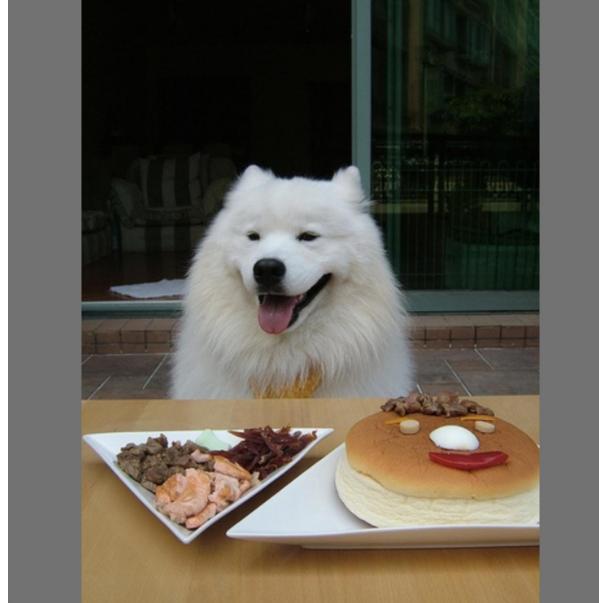
一、yolo v5解读

(一)、几种数据增强方法

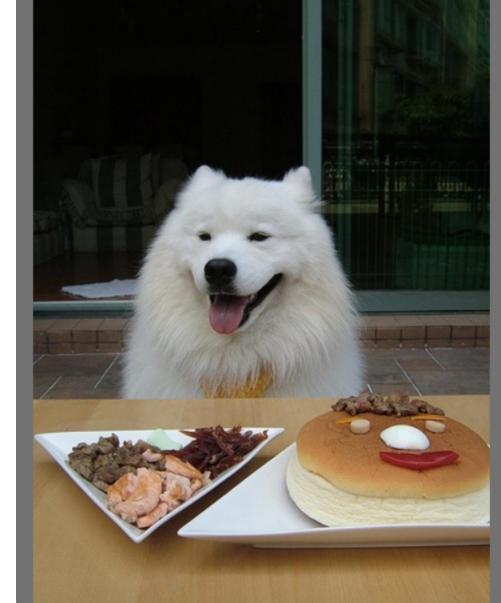
rectangular: 同个batch里做rectangle宽高等比变换，加快训练



原图: (375×500)



通常的resize: (640×640)



rectangular resize:
 (520×640)

注意点:

yolov5里rect为True时，对每个batch都是单独一个input shape，
所以每个batch都会尽量的减少pad的黑边，减少计算量

参考: yolov5/utils/datasets.py/#L462-L484

一、yolo v5解读

(一) 几种数据增强方法

rectangular: 同个batch里做rectangle宽高等比变换，加快训练

```
# Rectangular Training
if self.rect:
    # 所有训练图片的shape(w,h)
    s = self.shapes
    # 计算高宽比
    ar = s[:, 1] / s[:, 0]
    # 高宽比按小到大排序
    irect = ar.argsort()
    self.img_files = [self.img_files[i] for i in irect]
    self.label_files = [self.label_files[i] for i in irect]
    self.labels = [self.labels[i] for i in irect]
    self.shapes = s[irect] # wh
    ar = ar[irect]

    # Set training image shapes
    shapes = [[1, 1]] * nb
    for i in range(nb):
        # 同个batch的图片拿出来
        ari = ar[bi == i]
        mini, maxi = ari.min(), ari.max()
        if maxi < 1:
            # 同个batch里最大的高宽比小于1, 将batch里的所有图片处理成[640*maxi, 640*1], 注意这里是[h,w]格式
            shapes[i] = [maxi, 1]
        elif mini > 1:
            # 同个batch里最小的高宽比大于1, 将batch里的所有图片处理成[640*1, 640*(1/mini)], 注意这里是[h,w]格式
            shapes[i] = [1, 1 / mini]
    self.batch_shapes = np.ceil(np.array(shapes) * img_size / stride + pad).astype(np.int) * stride
```

一、yolo v5解读

(一)、几种数据增强方法

hsv_h: 0.015 # image HSV-Hue augmentation (fraction), 色调

hsv_s: 0.7 # image HSV-Saturation augmentation (fraction), 饱和度

hsv_v: 0.4 # image HSV-Value augmentation (fraction), 曝光度



原图



色调调整



饱和度调整



曝光调整



综合调整

yolov5 opencv, hsv变换:

```
r = np.random.uniform(-1, 1, 3) * [hgain, sgain, vgain] + 1
hue, sat, val = cv2.split(cv2.cvtColor(im, cv2.COLOR_BGR2HSV))
dtype = im.dtype
x = np.arange(0, 256, dtype=r.dtype)
lut_hue = ((x * r[0]) % 180).astype(dtype)
lut_sat = np.clip(x * r[1], 0, 255).astype(dtype)
lut_val = np.clip(x * r[2], 0, 255).astype(dtype)
im_hsv = cv2.merge((cv2.LUT(hue, lut_hue), cv2.LUT(sat, lut_sat), cv2.LUT(val, lut_val)))
cv2.cvtColor(im_hsv, cv2.COLOR_HSV2BGR, dst=im)
```

一、yolo v5解读

(一) 几种数据增强方法

degrees: 0.0 # image rotation (+/- deg), 旋转

scale: 0.5 # image scale (+/- gain), 缩放



$$\times \begin{bmatrix} 0.99640066 & -0.31456676 & 0. \\ 0.31456676 & 0.99640066 & 0. \\ 0. & 0. & 1. \end{bmatrix} =$$



$$\text{dst}\left(\frac{M_{11}x + M_{12}y + M_{13}}{M_{31}x + M_{32}y + M_{33}}, \cdot \frac{M_{21}x + M_{22}y + M_{23}}{M_{31}x + M_{32}y + M_{33}}\right) = \text{src}(x, y)$$

注意点:

1. src为左边图片, dst为右边旋转缩放变换后的图片, xy为横纵坐标, M为旋转缩放矩阵
 2. 旋转参数主要是M[0,1], M[1, 0]起作用, 且M[0,1], M[1, 0]互为相反数
 3. 缩放参数主要是M[0,0], M[1,1]起作用
- 详见: yolov5/utils/argumentations.py/#L143-L149

一、yolo v5解读

(一)、几种数据增强方法

degrees: 0.0 # image rotation (+/- deg), 旋转

scale: 0.5 # image scale (+/- gain), 缩放

```
def rotate_scale(im, degrees, scale):
    """旋转缩放"""
    im_copy = im.copy()
    h, w, _ = im.shape
    # Rotation and Scale matrix
    RS = np.eye(3)
    angle = random.uniform(-degrees, degrees)
    random_scale = random.uniform(1 - scale, 1 + scale)
    RS[:2] = cv2.getRotationMatrix2D(angle=angle, center=(0, 0), scale=random_scale)
    new_im = cv2.warpPerspective(im_copy, RS, dsize=(w, h), borderValue=(114, 114, 114))
    return new_im
```

yolov5中OpenCV实现旋转缩放

一、yolo v5解读

(一)、几种数据增强方法

translate: 0.1 # image translation (+/- fraction), 平移



$$\times \begin{bmatrix} 1. & 0. & 95. \\ 0. & 1. & 127. \\ 0. & 0. & 1. \end{bmatrix} =$$



$$\text{dst} \left(\frac{M_{11}x + M_{12}y + M_{13}}{M_{31}x + M_{32}y + M_{33}}, \cdot \frac{M_{21}x + M_{22}y + M_{23}}{M_{31}x + M_{32}y + M_{33}} \right) = \text{src}(x, y)$$

注意点:

1. src为左边图片, dst为右边旋转缩放变换后的图片, xy为横纵坐标, M为平移矩阵
2. x轴平移参数主要是M[0,2]起作用
3. y轴平移参数主要是M[1,2]起作用

详见: yolov5/utils/argumentations.py/#L156-L159

一、yolo v5解读

(一) 几种数据增强方法

translate: 0.1 # image translation (+/- fraction), 平移

```
def translate(im, t=0.1):
    """平移"""
    im_copy = im.copy()
    h, w, _ = im.shape
    T = np.eye(3)
    T[0, 2] = random.uniform(0.5 - t, 0.5 + t) * w * 0.5
    T[1, 2] = random.uniform(0.5 - t, 0.5 + t) * h * 0.5
    new_im = cv2.warpPerspective(im_copy, T, dsize=(w, h), borderValue=(114, 114, 114))
    return new_im
```

yolov5中OpenCV实现平移

一、yolo v5解读

(一)、几种数据增强方法

shear: 0.0 # image shear (+/- deg), 错切/非垂直投影



$$\begin{bmatrix} 1. & 0.12266402 & 0. \\ -0.76603159 & 1. & 0. \\ 0. & 0. & 1. \end{bmatrix} =$$



$$\text{dst} \left(\frac{M_{11}x + M_{12}y + M_{13}}{M_{31}x + M_{32}y + M_{33}}, \cdot \frac{M_{21}x + M_{22}y + M_{23}}{M_{31}x + M_{32}y + M_{33}} \right) = \text{src}(x, y)$$

注意点:

1. 错切的类似于固定图片一边，对另外平行一边施加一个推力形成的变形
2. **src**为左边图片，**dst**为右边错切变换后的图片，**xy**为横纵坐标，**M**为错切矩阵
3. 错切参数主要是**M[0,1]**, **M[1, 0]**起作用

详见: [yolov5/utils/argumentations.py/#L152-L154](#)

一、yolo v5解读

(一)、几种数据增强方法

shear: 0.0 # image shear (+/- deg), 错切/非垂直投影

```
def shear(im, degree):
    """错切"""
    im_copy = im.copy()
    h, w, _ = im.shape
    S = np.eye(3)
    # 错切和旋转都是通过[0,1],[1,0]两个参数控制，不同的是旋转两个参数互为相反数，错切则不然
    S[0, 1] = math.tan(random.uniform(-degree, degree) * math.pi / 180)
    S[1, 0] = math.tan(random.uniform(-degree, degree) * math.pi / 180)
    print(S)
    new_im = cv2.warpPerspective(im_copy, S, dsize=(w, h), borderValue=(114, 114, 114))
    return new_im
```

yolov5 错切实现

一、yolo v5解读

(一) 几种数据增强方法

perspective: 0.0 # image perspective (+/- fraction), range 0-0.001, 透视变换



$$\begin{bmatrix} 1. & 0. & 0. \\ 0. & 1. & 0. \\ 0.000752396836 & 0.0000695 & 1. \end{bmatrix}$$



$$\text{dst} \left(\frac{M_{11}x + M_{12}y + M_{13}}{M_{31}x + M_{32}y + M_{33}}, \cdot \frac{M_{21}x + M_{22}y + M_{23}}{M_{31}x + M_{32}y + M_{33}} \right) = \text{src}(x, y)$$

注意点:

1. src为左边图片, dst为右边透视变换后的图片, xy为横纵坐标, M为变换矩阵
2. 变换参数主要是M[2,0], M[2,1]起作用

一、yolo v5解读

(一) 几种数据增强方法

perspective: 0.0 # image perspective (+/- fraction), range 0-0.001, 透视变换

```
def perspective(im, p=0.001):
    """透视变换"""
    h, w, c = im.shape
    im_copy = im.copy()
    P = np.eye(3)
    P[2, 0] = random.uniform(-p, p)
    P[2, 1] = random.uniform(-p, p)
    im_copy = cv2.warpPerspective(im_copy, P, dsize=(w, h), borderValue=(114, 114, 114))
    return im_copy
```

yolov5中OpenCV实现透视变换

```
def perspective(im, p=0.001):
    """透视变换"""
    h, w, c = im.shape
    P = np.eye(3)
    P[2, 0] = random.uniform(-p, p)
    P[2, 1] = random.uniform(-p, p)
    new_im = np.zeros_like(im) + 114.
    for row in range(h):
        for col in range(w):
            col_new = (P[0, 0] * col + P[0, 1] * row + P[0, 2]) / (P[2, 0] * col + P[2, 1] * row + P[2, 2])
            row_new = (P[1, 0] * col + P[1, 1] * row + P[1, 2]) / (P[2, 0] * col + P[2, 1] * row + P[2, 2])
            new_im[int(row_new), int(col_new), :] = im[row, col, :]

    new_im = np.array(new_im, dtype=np.uint8)
    return new_im
```

根据变换公式实现透视变换

一、yolo v5解读

(一)、几种数据增强方法

flipud: 0.0 # image flip up-down (probability), 上下翻转

fliplr: 0.5 # image flip left-right (probability), 左右翻转



原图



上下翻转



左右翻转

yolov5 翻转变换：

上下翻转

im_up = np.flipud(im)

box坐标y翻转

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

左右翻转

im_right = np.fliplr(im)

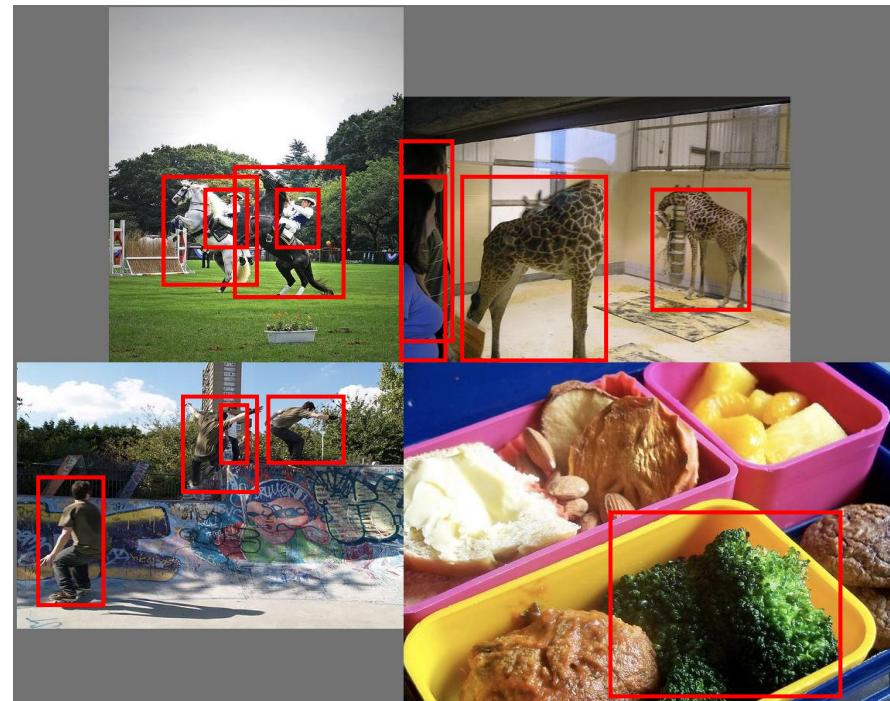
box坐标x翻转

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

一、yolo v5解读

(一)、几种数据增强方法

mosaic: 1.0 # image mosaic (probability), 4图拼接



拼接流程:

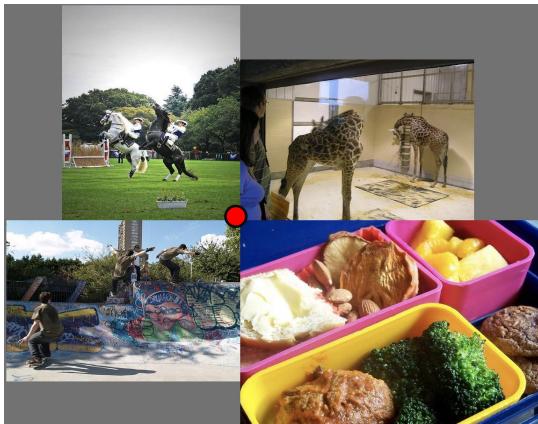
1. 初始化整个背景图, 大小为 $(2*\text{image_size}, 2*\text{image_size}, 3)$
2. 随机取一个中心点
3. 基于中心点分别将4个图放到左上, 右上, 左下, 右下, **此部分可能会由于中心点小于4张图片的宽高, 所以拼接的时候可能会进行裁剪**
4. 重新将打标边框的偏移量计算上
详见: [yolov5/utils/datasets.py/#L681-L711](#)

一、yolo v5解读

(一) 几种数据增强方法

mosaic: 1.0 # image mosaic (probability), 4图拼接

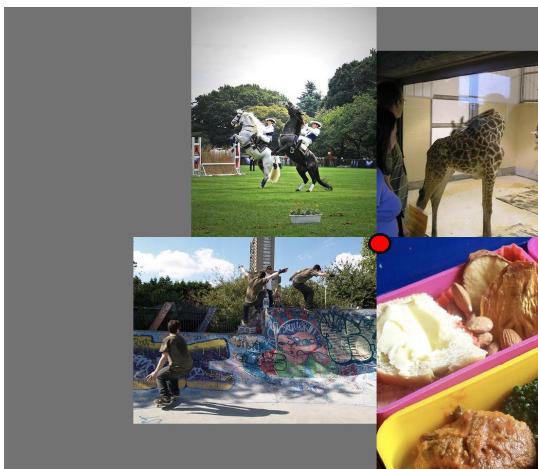
以下示例是由于中心点(下图红点)随机选取不同，可能会出现的裁剪情况



中心点左右上下
刚好大于任何图
片的宽高



中心点下半部分
小于图片高
下边图片进行裁剪



中心点右半部分
小于图片宽
右边图片进行裁剪

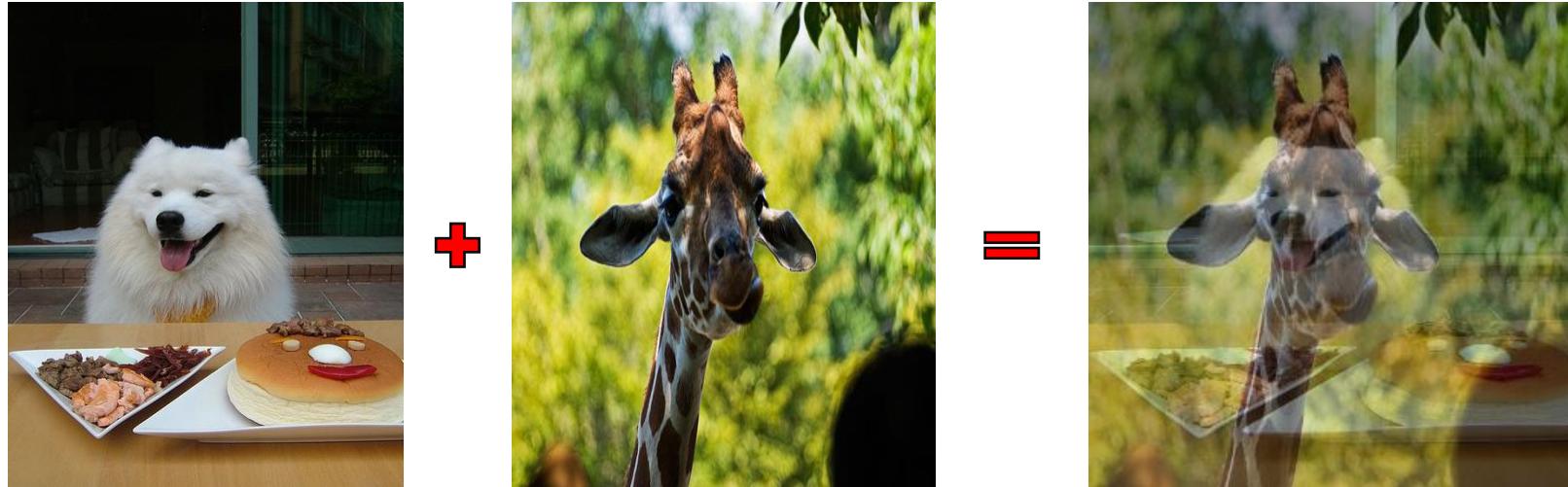


中心点左半部分
小于图片宽
左边图片进行裁剪

一、yolo v5解读

(一) 几种数据增强方法

mixup: 0.0 # image mixup (probability), 图像互相融合



mixup的思想其实就是简单地将两张图叠加到一起，通过不同的透明度区分

下面是yolov5的实现代码:

```
def mixup(im, labels, im2, labels2):
    # Applies MixUp augmentation https://arxiv.org/pdf/1710.09412.pdf
    r = np.random.beta(32.0, 32.0) # mixup ratio, alpha=beta=32.0
    im = (im * r + im2 * (1 - r)).astype(np.uint8)
    labels = np.concatenate((labels, labels2), 0)
    return im, labels
```

一、yolo v5解读

(一) 几种数据增强方法

copy_paste: 0.0 # segment copy-paste (probability), 分割填补



+



=



+



=



分割填补注意点:

1. 分割出图像的目标后，需要计算该目标边框与填补图片中的所有目标边框**IOU<0.3**(实现参数)
2. 详见yolov5/utils/argumentations.py/#L213-L234

一、yolo v5解读

(一) 几种数据增强方法

最后box坐标转换, segment坐标转换

这部分参考: yolov5/utils/argumentations.py/#L175-L208

1. 将所有变换矩阵连乘得到最终的变换矩阵: $M = T @ S @ R @ P @ C$

2. 将坐标点(x,y)处理成(x,y,1), 其中segment.shape=[n, 2], 表示物体轮廓各个坐标点:

`xy = np.ones((len(segment), 3))`

`xy[:, :2] = segment`

如果是box坐标, 这里targets每行为[x1,y1,x2,y2], n为行数, 表示目标边框个数:

`xy = np.ones((n * 4, 3))`

`xy[:, :2] = targets[:, [1, 2, 3, 4, 1, 4, 3, 2]].reshape(n * 4, 2)`

3. 应用旋转矩阵: `xy = xy @ M.T`

4. rescale操作, 这里如果透视变换参数perspective不为0, 就需要做rescale:

`xy = xy[:, :2] / xy[:, 2:3]`

透视变换参数为0, 则无需做rescale:

`xy = xy[:, :2]`

为什么perspective这里需要做rescale?

$$\text{dst} \left(\frac{M_{11}x + M_{12}y + M_{13}}{M_{31}x + M_{32}y + M_{33}}, \frac{M_{21}x + M_{22}y + M_{23}}{M_{31}x + M_{32}y + M_{33}} \right) = \text{src}(x, y)$$

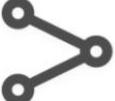
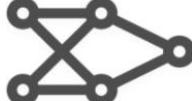
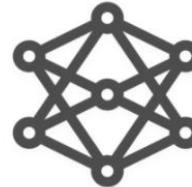
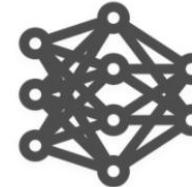
如果是perspective, rescale就跟图像变换一样, 还需要除以上述分母

5. 将坐标clip到[0, width],[0,height]区间内

6. 进一步过滤, 留下那些w,h>2,宽高比<20,变换后面积比之前比>0.1的那些xy

一、yolo v5解读

(二) backbone网络 参数量/体积

			
Small YOLOv5s	Medium YOLOv5m	Large YOLOv5l	XLarge YOLOv5x
14 MB _{FP16} 2.0 ms _{V100} 37.2 mAP _{COCO}	41 MB _{FP16} 2.7 ms _{V100} 44.5 mAP _{COCO}	90 MB _{FP16} 3.8 ms _{V100} 48.2 mAP _{COCO}	168 MB _{FP16} 6.1 ms _{V100} 50.4 mAP _{COCO}

tf2.x复现参数量:

参数量: 7202491
float16: ≈ 13.5M

参数量: 21025275
float16: ≈ 40M

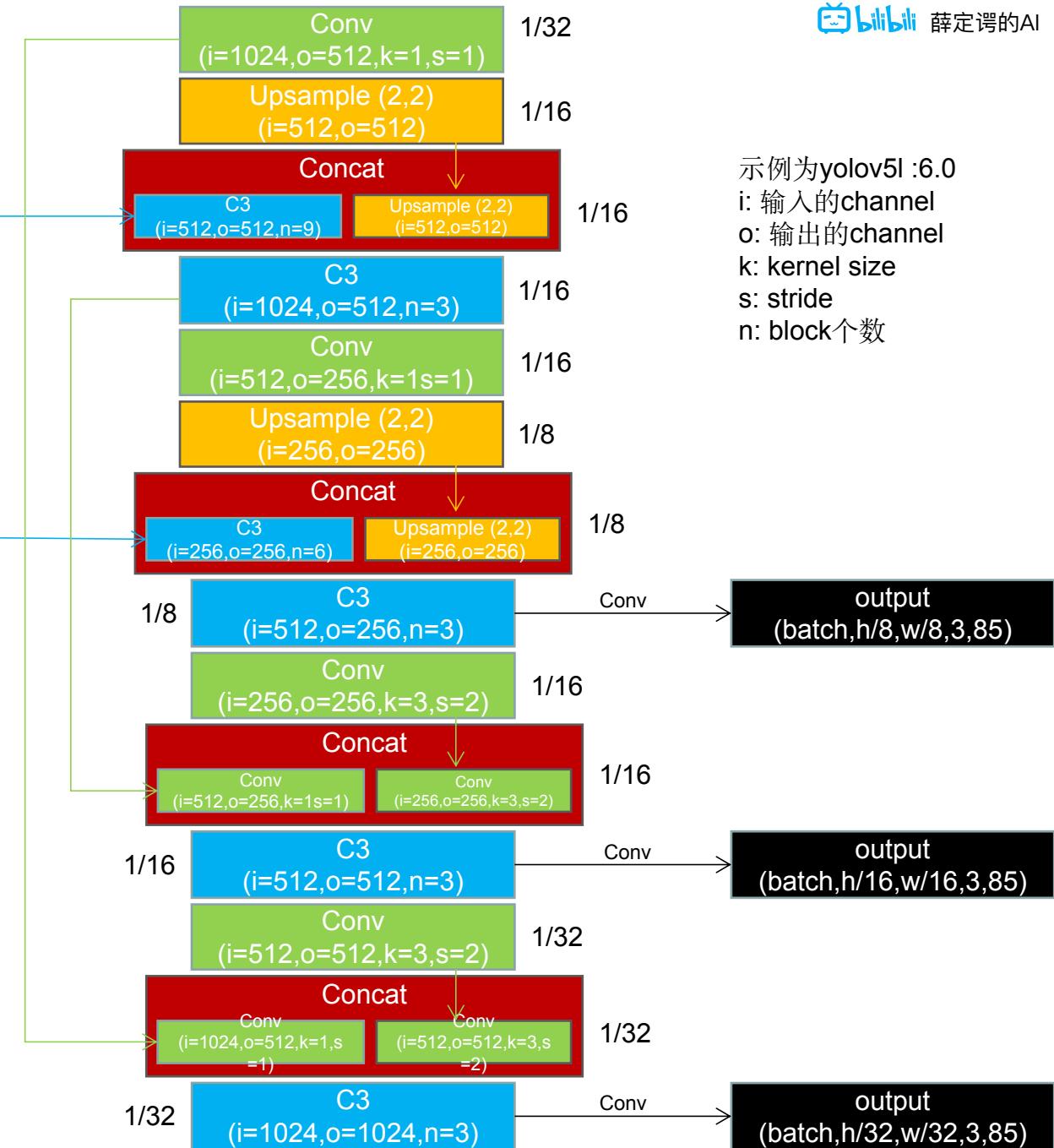
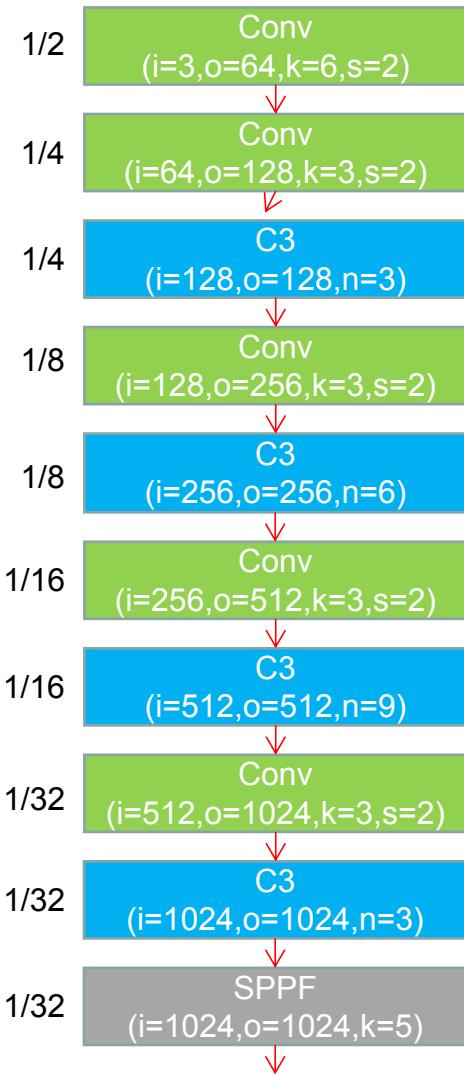
参数量: 46354491
float16: ≈ 88.4M

参数量: 86501755
float16: ≈ 165M

参数换算公式: 参数量 * 2 / 1024 / 1024 (一个float16占2个字节, 1024个字节为1k)

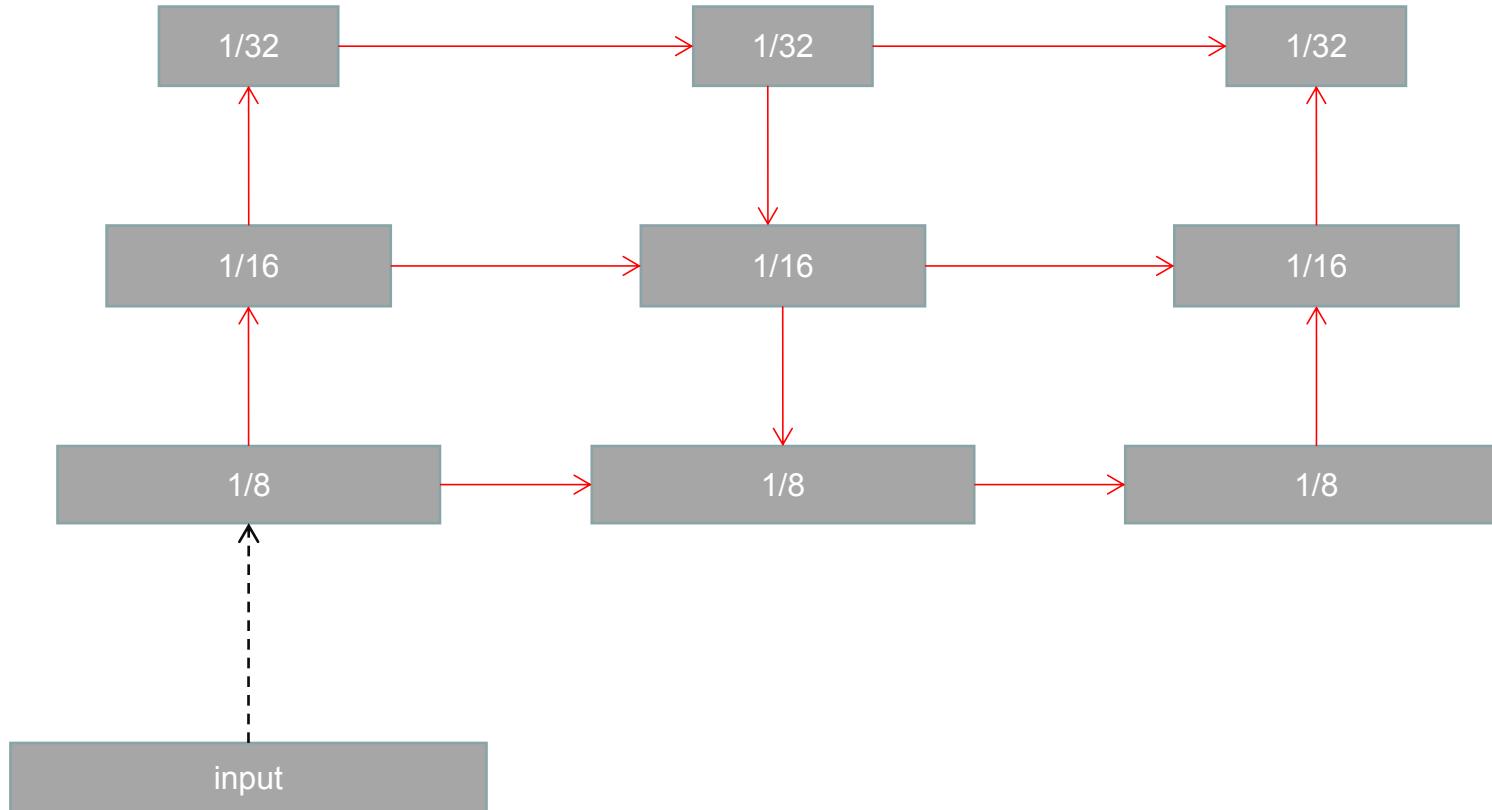
一、yolo v5解读

(二) backbone网络



一、yolo v5解读

(二) backbone网络: FPN+PAN



一、yolo v5解读

(二) backbone网络 Conv层

```
class Conv(nn.Module):
    # Standard convolution
    def __init__(self, c1, c2, k=1, s=1, p=None, g=1, act=True): # ch_in, ch_out, kernel, stride, padding, groups
        super().__init__()
        self.conv = nn.Conv2d(c1, c2, k, s, autopad(k, p), groups=g, bias=False)
        self.bn = nn.BatchNorm2d(c2)
        self.act = nn.SiLU() if act is True else (act if isinstance(act, nn.Module) else nn.Identity())

    def forward(self, x):
        return self.act(self.bn(self.conv(x)))

    def forward_fuse(self, x):
        return self.act(self.conv(x))
```

1. 关于分组卷积，主要是减少参数减低计算量用.

2. 其计算逻辑大概为:

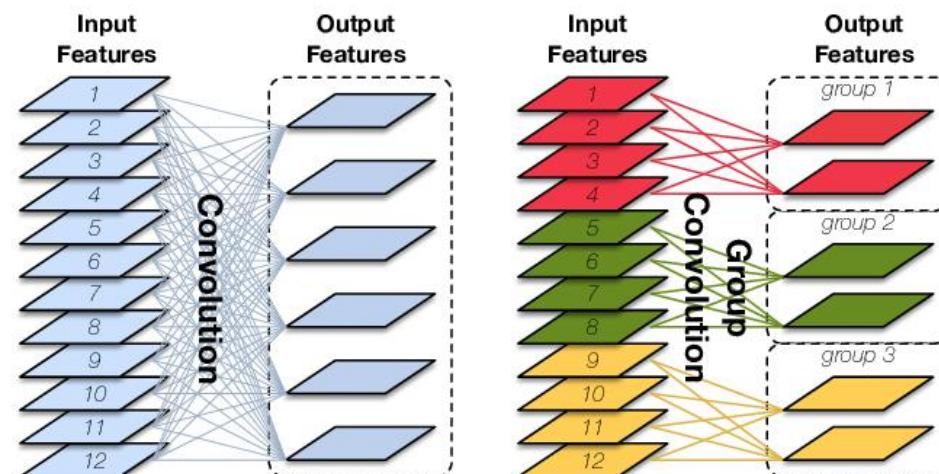
如右图，假设输入channel为12，输出channel为6，
kernel_size=3，分组groups=3，bias=False

- 那么传统的卷积每个卷积核大小为 $3 \times 3 \times 12$ ，总的参数量为 $6 \times 3 \times 3 \times 12$

- 分组后每个group的输入channel= $12/3=4$ ，每个group输出channel= $6/3=2$ ，每个group里面的卷积核大小= $3 \times 3 \times 4$ ，总的参数量为 $6 \times 3 \times 3 \times 4$

可以看出来分组后参数量为传统卷积的 $1/groups$ 。

不过，在最新的yolov5 6.0代码里面不做这个分组。



分组卷积
[图片链接](#)

一、yolo v5解读

(二) backbone网络 Conv层

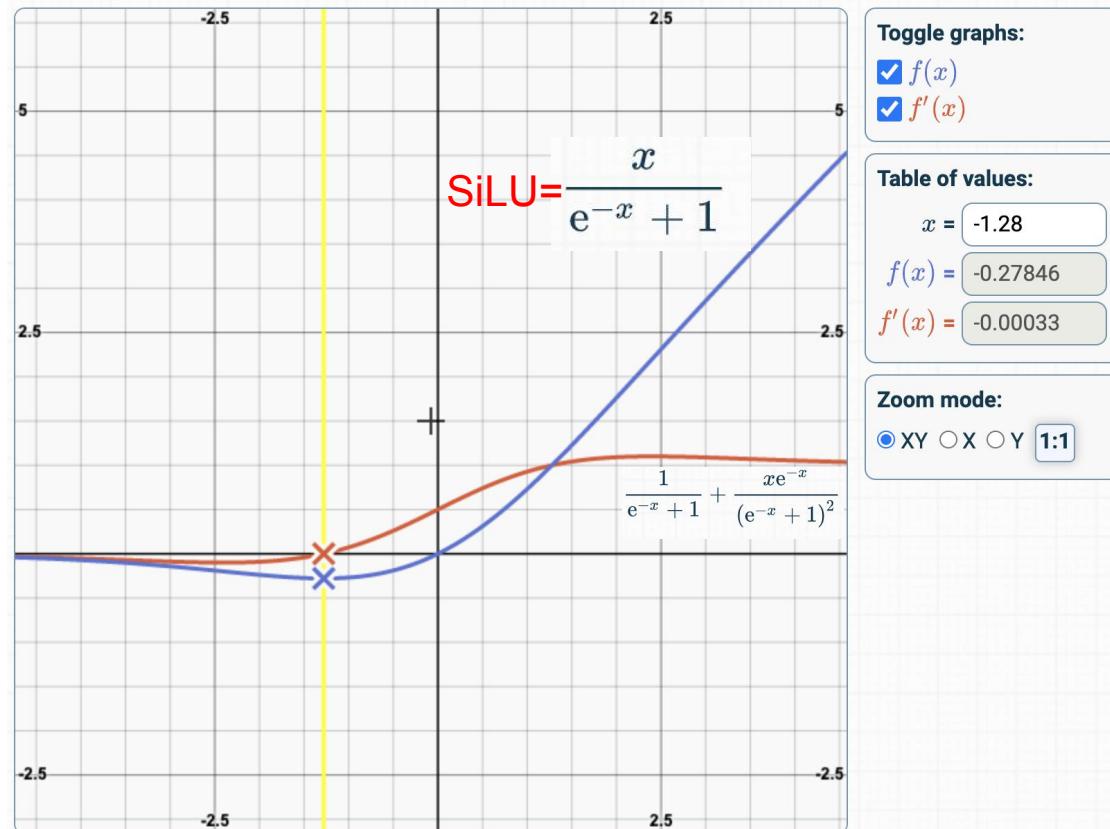
```
class Conv(nn.Module):
    # Standard convolution
    def __init__(self, c1, c2, k=1, s=1, p=None, g=1, act=True): # ch_in, ch_out, kernel, stride, padding, groups
        super().__init__()
        self.conv = nn.Conv2d(c1, c2, k, s, autopad(k, p), groups=g, bias=False)
        self.bn = nn.BatchNorm2d(c2)
        self.act = nn.SiLU() if act is True else (act if isinstance(act, nn.Module) else nn.Identity())

    def forward(self, x):
        return self.act(self.bn(self.conv(x)))

    def forward_fuse(self, x):
        return self.act(self.conv(x))
```

1. SiLU, 其表达式以及表现形式如右图蓝线.
2. SiLU的函数形式跟ReLU十分相似, 区别在于SiLU在<0处有负值, 随着输入的绝对值越大, 其跟ReLU就越相似, 表现出ReLU非线性特点, 以及避免梯度弥散消失, 同时也具有一定的正则效果.
3. SiLU最早是[这篇论文](#)里提出的用于强化学习的, 作者提到SiLU具有自稳定性, 相关实验可以在这个[论文](#)看到, 同时SiLU非单调递增特征, 在x=-1.28存在函数极小值, 作者在其论文也解释到这种存在了一定的隐性正则效果.

最后为什么yolov5采用SiLU, 其实更多还是实验的结果: [这里](#)可以看到作者实验了11种.
如果你有新的act, 也可以像这个[issue](#)跟作者讨论.

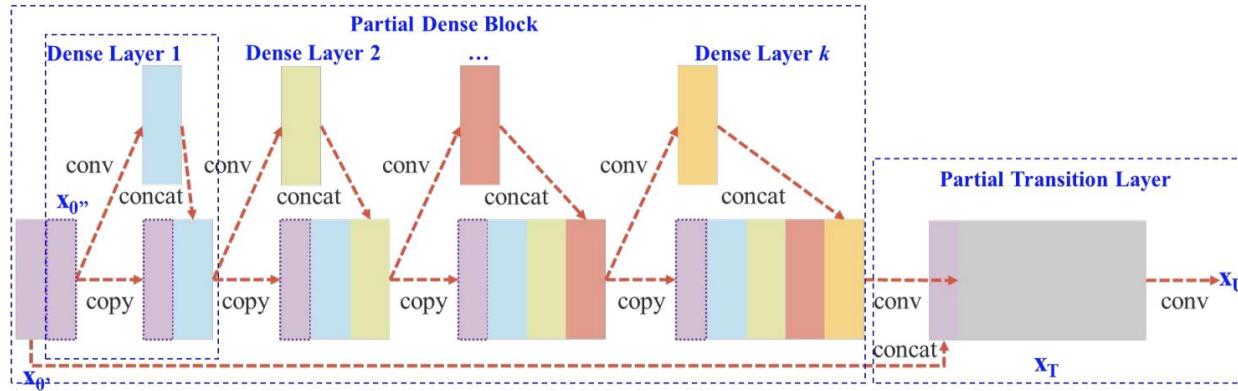


一、yolo v5解读

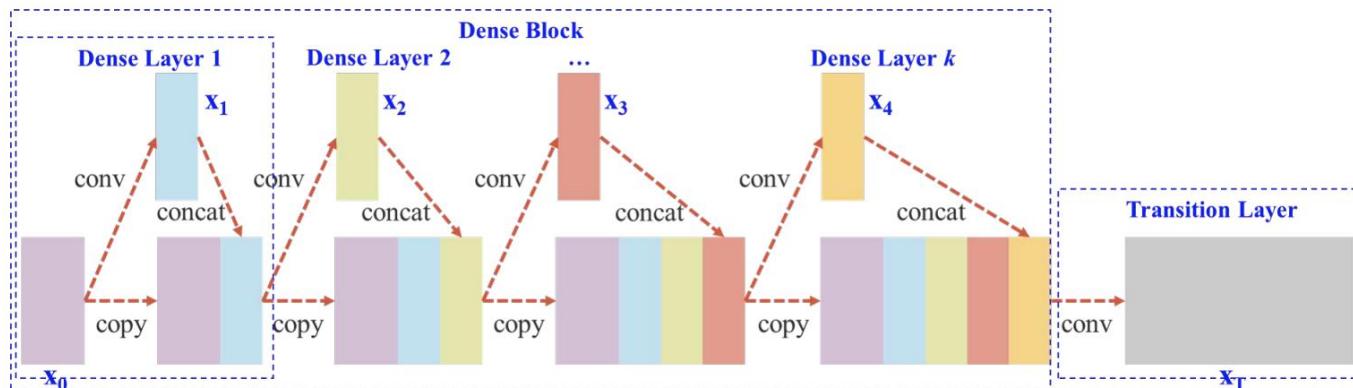
(二)、backbone网络 C3层

BottleneckCSP 与 C3

1. CSP结构是这篇[论文](#)中提出的，如下图所示：



2. CSPNet论文中主要是对比denseNet的特征融合，目的是为了提高模型的速度，同时兼顾精度，denseNet的concat结构如下图：



3. 可以看出CSP结构通过分离通道层，只对其中一部分特征做卷积，再concat，以此减少计算量，同时又能保证精度。

一、yolo v5解读

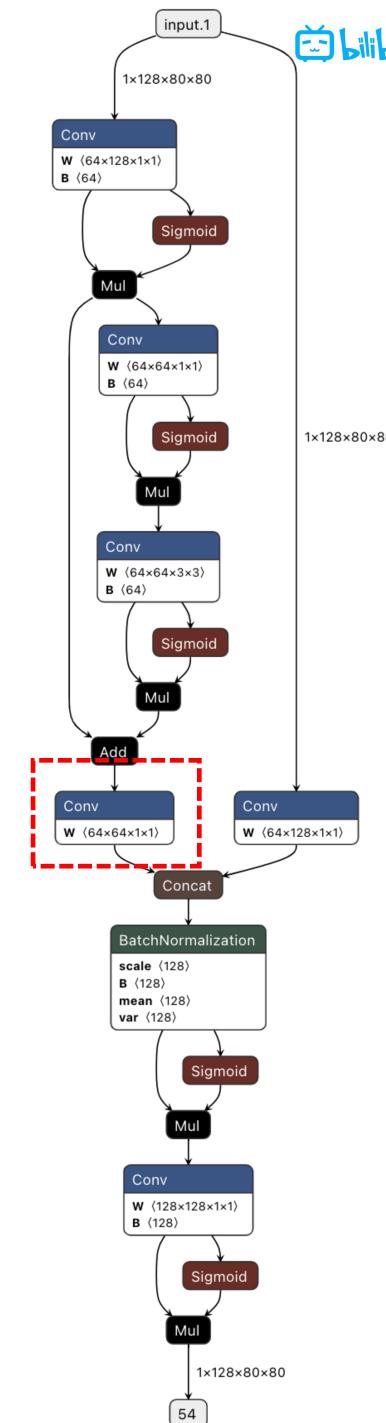
(二)、backbone网络 C3层

BottleneckCSP 与 C3

在最新的yolov5中(release-v6.0), bottleneckCSP作者修改了原论文关于CSP的实现, 不采用通道分离, 而是直接全部参与卷积:

```
class BottleneckCSP(nn.Module):
    # CSP Bottleneck https://github.com/WongKinYiu/CrossStagePartialNetworks
    def __init__(self, c1, c2, n=1, shortcut=True, g=1, e=0.5):
        super().__init__()
        c_ = int(c2 * e) # hidden channels
        self.cv1 = Conv(c1, c_, 1, 1)
        self.cv2 = nn.Conv2d(c1, c_, 1, 1, bias=False)
        self.cv3 = nn.Conv2d(c_, c_, 1, 1, bias=False)
        self.cv4 = Conv(2 * c_, c2, 1, 1)
        self.bn = nn.BatchNorm2d(2 * c_) # applied to cat(cv2, cv3)
        self.act = nn.SiLU()
        self.m = nn.Sequential(*(Bottleneck(c_, c_, shortcut, g, e=1.0) for _ in range(n)))

    def forward(self, x):
        y1 = self.cv3(self.m(self.cv1(x)))
        y2 = self.cv2(x)
        return self.cv4(self.act(self.bn(torch.cat((y1, y2), dim=1))))
```



一、yolo v5解读

(二)、backbone网络 C3层

在最新的yolov5中(release-v6.0), 最终采用的C3的结构, 而C3名称的由来, 就是bottleneckCSP中4个conv卷积层, 去掉了1层, 剩下3层, 所以称为C3, 具体去掉的哪一层卷积, 位于上一P中CSP结构中画红圈部分.

```
class C3(nn.Module):
    # CSP Bottleneck with 3 convolutions
    def __init__(self, c1, c2, n=1, shortcut=True, g=1, e=0.5):
        super().__init__()
        c_ = int(c2 * e) # hidden channels
        self.cv1 = Conv(c1, c_, 1, 1)
        self.cv2 = Conv(c1, c_, 1, 1)
        self.cv3 = Conv(2 * c_, c2, 1) # act=FReLU(c2)
        self.m = nn.Sequential(*(Bottleneck(c_, c_, shortcut, g, e=1.0) for _ in range(n)))

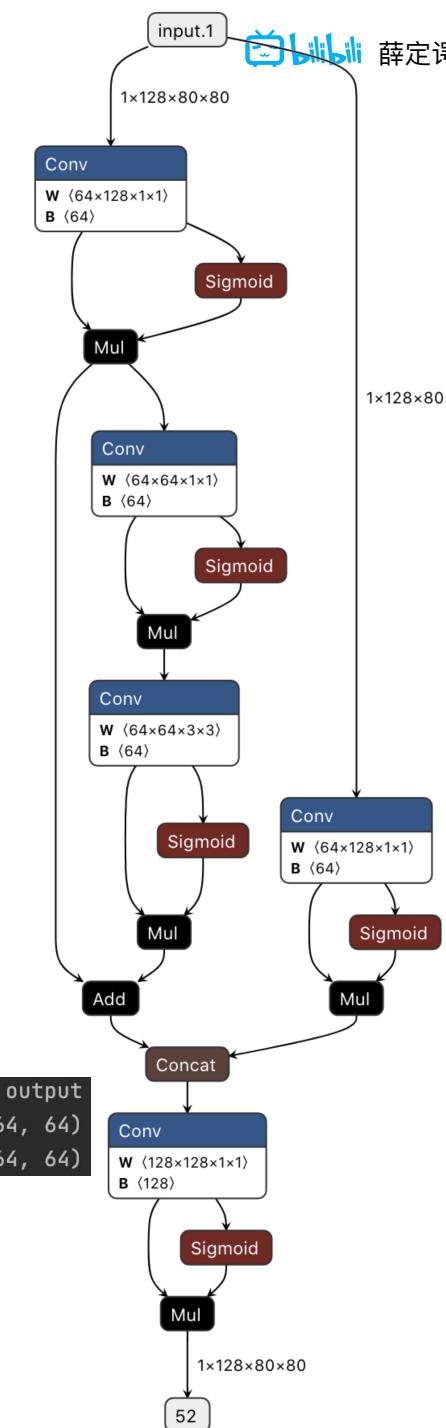
    def forward(self, x):
        return self.cv3(torch.cat((self.m(self.cv1(x)), self.cv2(x)), dim=1))
```

所以最后采用C3这种结构, 同样是实验的结果, bottleneckCSP和C3性能上的对比如下图(上面是bottleneckCSP, 下面是C3), 从前向计算, 反向传播来说, C3都比bottleneckCSP快:

Params	GFLOPs	GPU_mem (GB)	forward (ms)	backward (ms)	input	output
7611392	0	2.166	48.01	87.05	(6, 1024, 64, 64)	(6, 1024, 64, 64)
7348224	0	2.020	30.66	58.69	(6, 1024, 64, 64)	(6, 1024, 64, 64)

实验代码:

```
csp = BottleneckCSP(1024,1024,2)
c3 = C3(1024,1024,2)
result = profile(input=torch.randn(6,1024,64,64), ops=[csp, c3],n=50)
```

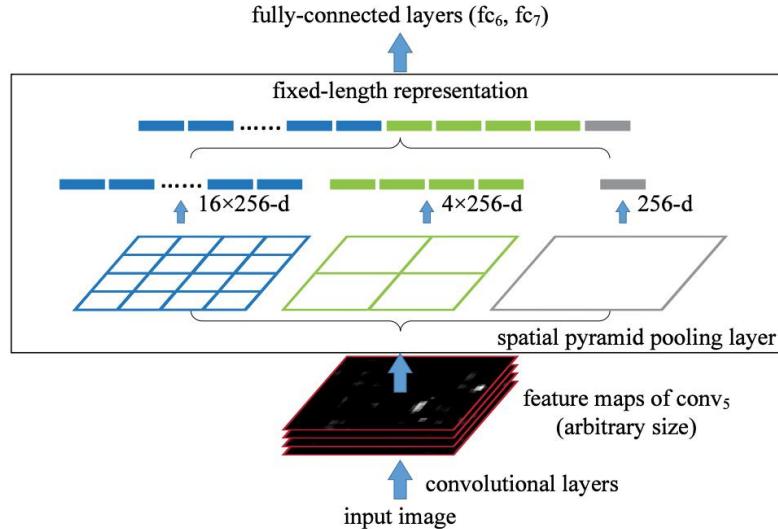


一、yolo v5解读

(二) backbone网络 SPPF层

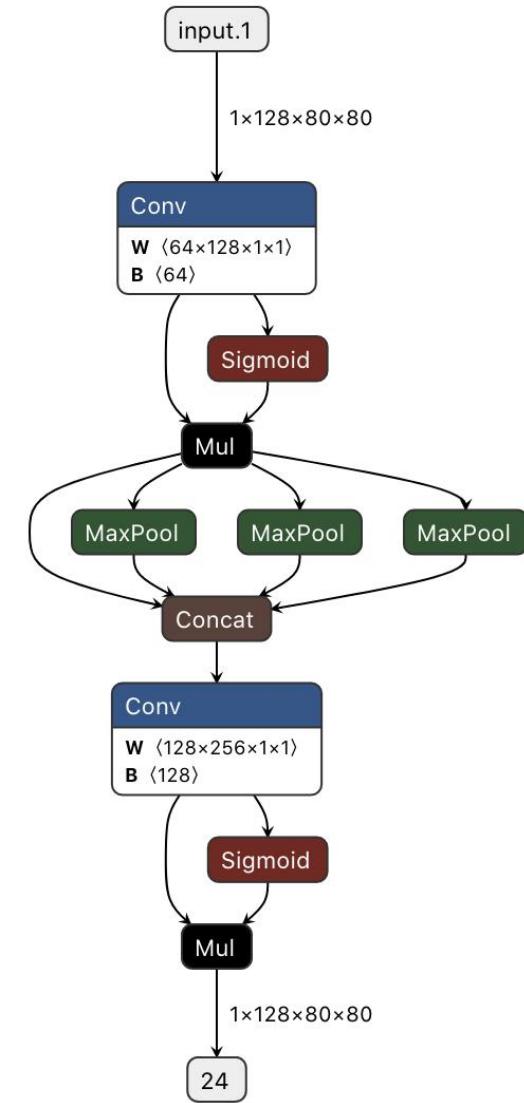
SPP与SPPF

1. SPP是15年何凯明在[论文](#)中提出的一种空间金字塔池化结构，如下图：



本意上是为了解决RCNN目标检测中不同proposal边框最后都能接上FC层，通过对卷积层施加自适应的大小3种池化操作，最后得到结果再做flatten到固定的大小，就可以统一接fc，就可以避免当时的ROIwrap层的裁剪+reisze丢失精度。

2. 右图早期的yolov5 spp实现，yolov5修改了spp中的池化padding,stride，使得所有三种不同尺度的池化通过不同kernel_size的池化后，都能输出统一的大小。



yolov5 spp层

一、yolo v5解读

(二) backbone网络 SPPF层

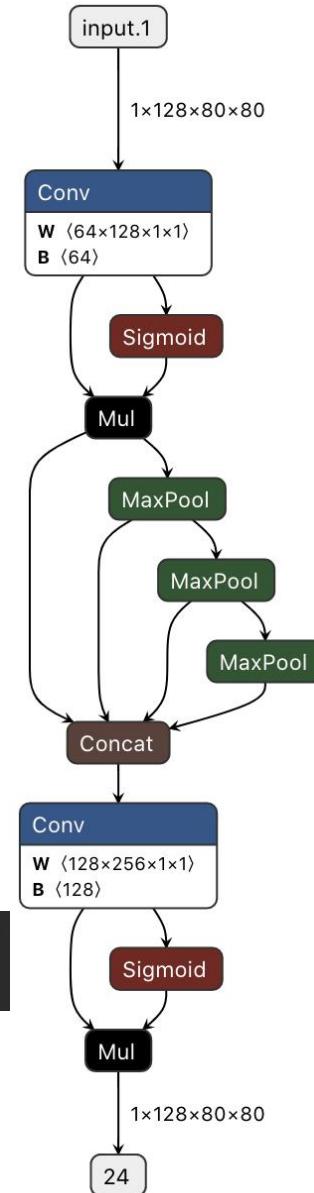
在最新的yolov5中(release-v6.0)

1. 作者将早期的SPP替换为SPPF，可以在[这个issue](#)看到，修改的地方是将3个并行的max_pool替换为串行，并且kernel_size全部都是一样，如右图所示。

2. 这么修改的目的就只有一个，就是实践得到串行的max_pool同样能得到spp的结果，同时能带来更多的性能上的提升，包括前向计算，反向计算，对比结果可以看下面截图，上面是SPP的实验效果，下面是SPPF的实验效果：

```
from models.common import SPPF,SPP
from utils.torch_utils import profile
m1 = SPP(1024, 1024)
m2 = SPPF(1024, 1024)
results = profile(input=torch.randn(16, 1024, 64, 64), ops=[m1, m2], n=100)
```

Params	GFLOPs	GPU_mem (GB)	forward (ms)	backward (ms)	input	output
2624512	0	4.586	80.9	171.2	(16, 1024, 64, 64)	(16, 1024, 64, 64)
2624512	0	4.452	55.23	114.7	(16, 1024, 64, 64)	(16, 1024, 64, 64)



一、yolo v5解读

(三) 边框预测细节

$$b_x = (2\sigma(t_x) - 0.5 + c_x) * stride$$

$$b_y = (2\sigma(t_y) - 0.5 + c_y) * stride$$

$$b_w = (2\sigma(t_w))^2 * anchor_grid$$

$$b_h = (2\sigma(t_h))^2 * anchor_grid$$

几个注意点：

1. 其中tx, ty, tw, th为模型预测输出，
bx,by,bw,bh为最终预测目标边框中心点， 宽高
2. cx, cy为当前预测的Tensor对应的grid坐标， 值域为[0, grid_size)，
stride为[8, 16, 32]，举例加入输入[416, 416, 3]，预测输出的tensor是
[batch, 52, 52, 3, 5+num_class]，则cx是在[0, 52)区间， stride为8
3. anchor_grid同理，值域为[0, image_size]，具体实现是anchor * stride，
其中anchor会被处理到[0, grid_size)区间，再通过 * stride放大到原图输入大小[0, image_size)

一、yolo v5解读

(三) 边框预测细节

$$x_offset = \sigma(t_x)$$

$$y_offset = \sigma(t_y)$$

yolov3 中心点xy偏移量预测
(未加上对应grid坐标)

$$x_offset = 2\sigma(t_x) - 0.5$$

$$y_offset = 2\sigma(t_y) - 0.5$$

yolov5 中心点xy偏移量预测
(未加上对应grid坐标)

公式改动点：

1. yolov5 相比 v3, 偏移量在经过sigmoid函数归一化到[0,1]后, 乘以了2再减0.5, 最后值域空间为[-0.5, 1.5]
2. 偏移量值域空间从[0,1]转成[-0.5, 1.5], 主要是v5正采样逻辑, 每个**ground true grid cell** 上下左右4个位置都参与到采样空间, 往右和往上最多需要减0.5, 往左和往下最多需要加1.5, 具体采样细节看第(4)部分

一、yolo v5解读

(三) 边框预测细节

$$w = e^{t_w}$$

$$h = e^{t_h}$$

$$w = (2\sigma(t_w))^2$$

$$h = (2\sigma(t_h))^2$$

yolov3 边框宽高w,h预测
(未乘上对应anchor宽高)

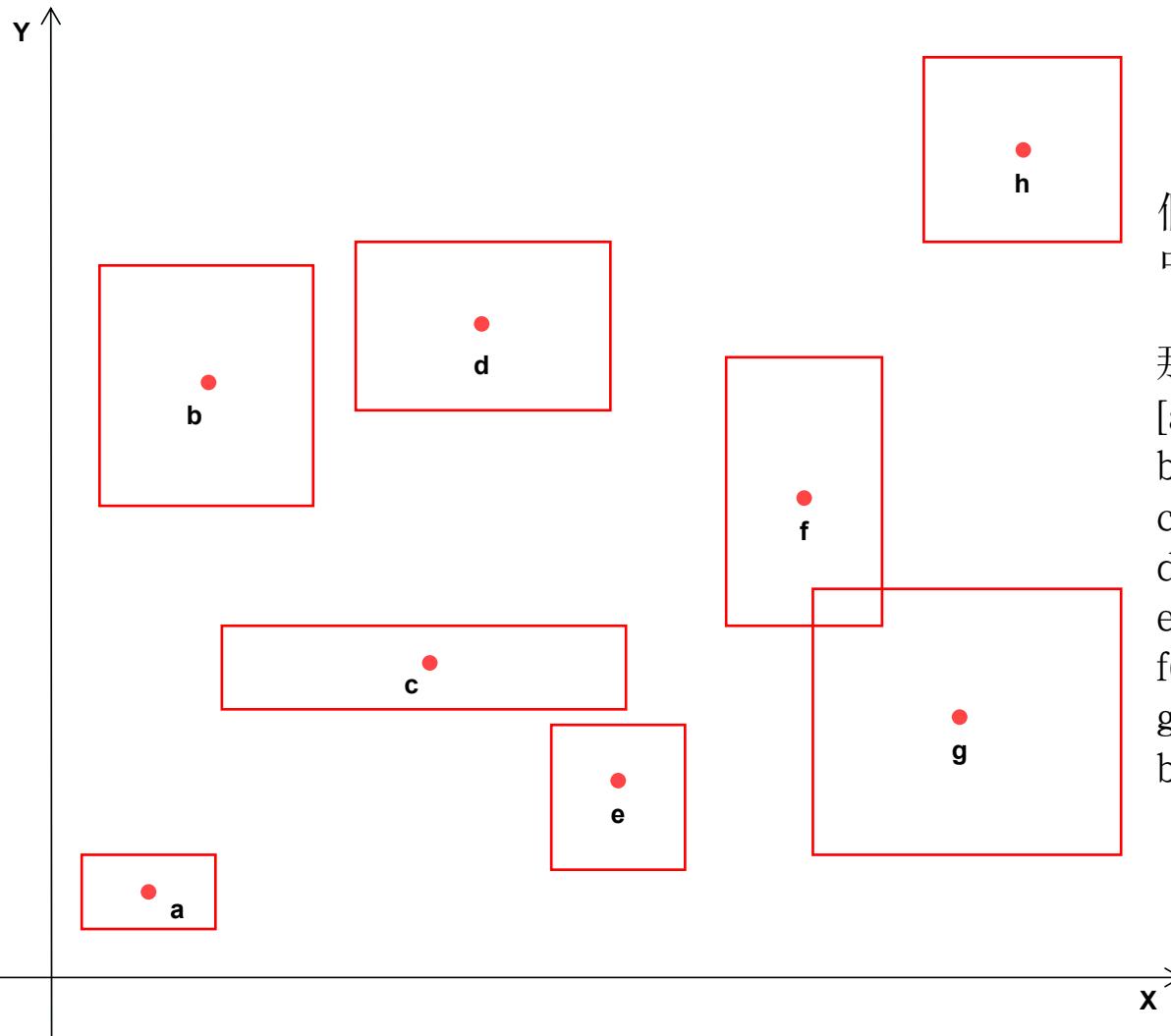
yolov5 边框宽高w,h预测
(未乘上对应anchor宽高)

公式改动点：

1. yolov5 相比 v3, 偏移量在经过sigmoid函数归一化到[0,1]后, 乘以了2再做平方, 最后值域空间为[0, 4], v3用e做指数得到最终w,h
2. 宽高预测值域空间转成[0, 4], 主要是v5边框筛选逻辑, 每个**ground true** 目标边框宽高与当前尺度下的**anchor**宽高计算比值, 只要那些比值在**[0.25, 4]**之间, 具体筛选细节看v5源码
`utils/loss.py/ComputeLoss/build_targets/ r跟j两个变量`

一、yolo v5解读

(四) 正样本采样细节



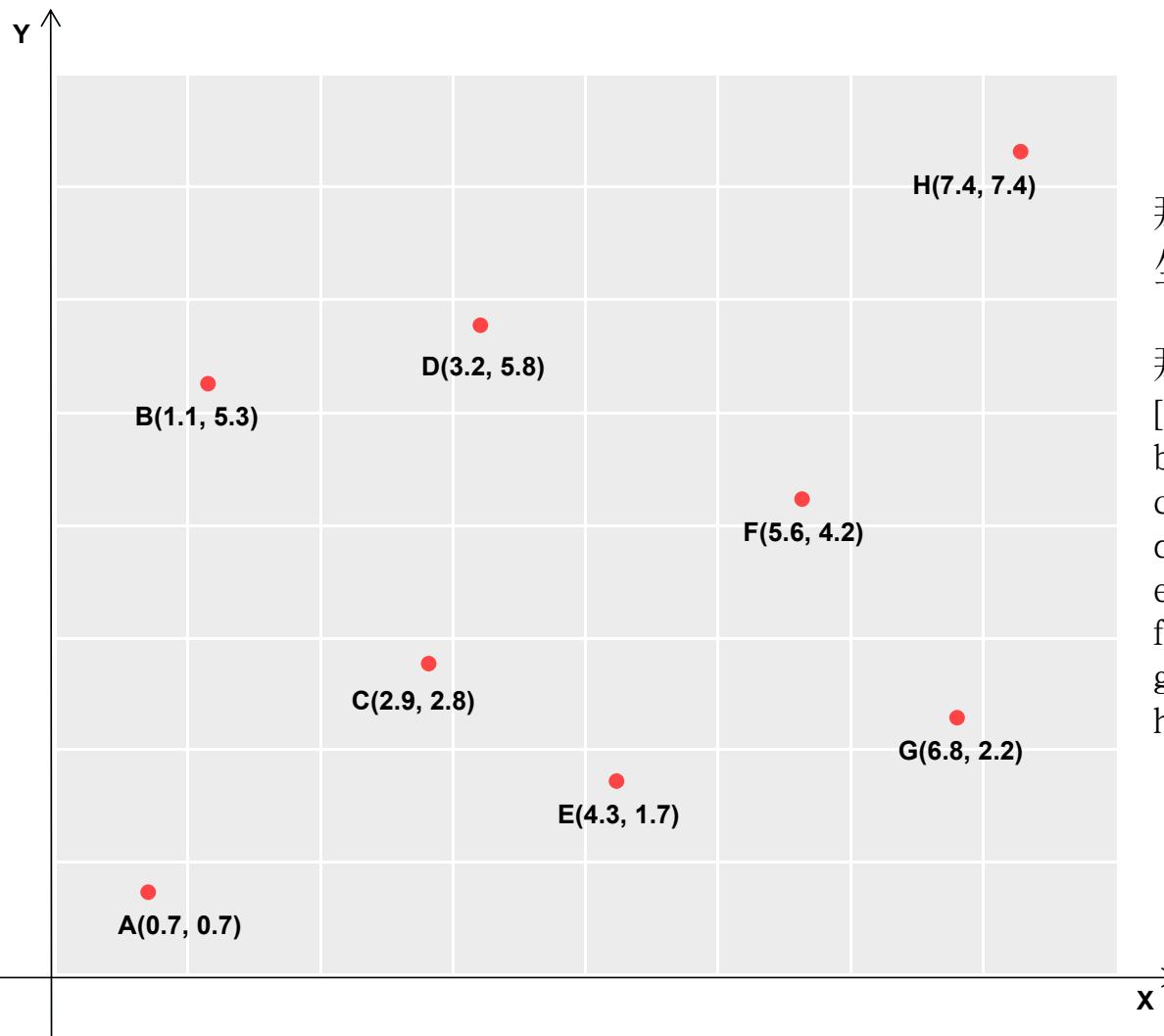
假设a-h总共8个目标边框，其中红点为边框中心点，如左图：

那么输入的target数据为：

[a(image_id, class_id, x, y, w, h),
b(image_id, class_id, x, y, w, h),
c(image_id, class_id, x, y, w, h),
d(image_id, class_id, x, y, w, h),
e(image_id, class_id, x, y, w, h),
f(image_id, class_id, x, y, w, h),
g(image_id, class_id, x, y, w, h),
h(image_id, class_id, x, y, w, h)]

一、yolo v5解读

(四) 正样本采样细节



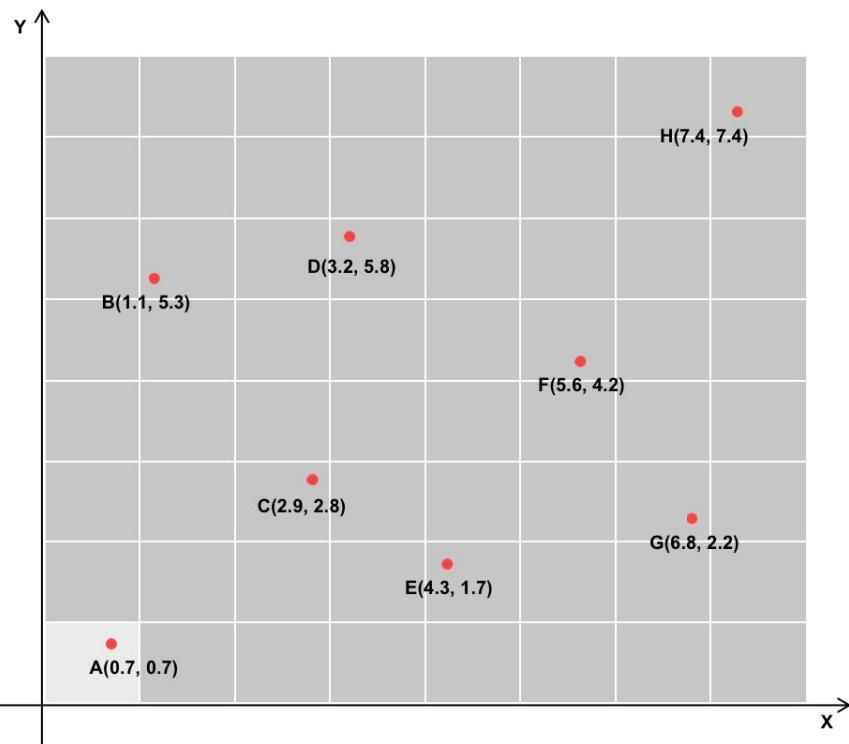
那么a-h 8个边框对应的中心点坐标A-H如图：

那么输入的target数据为：

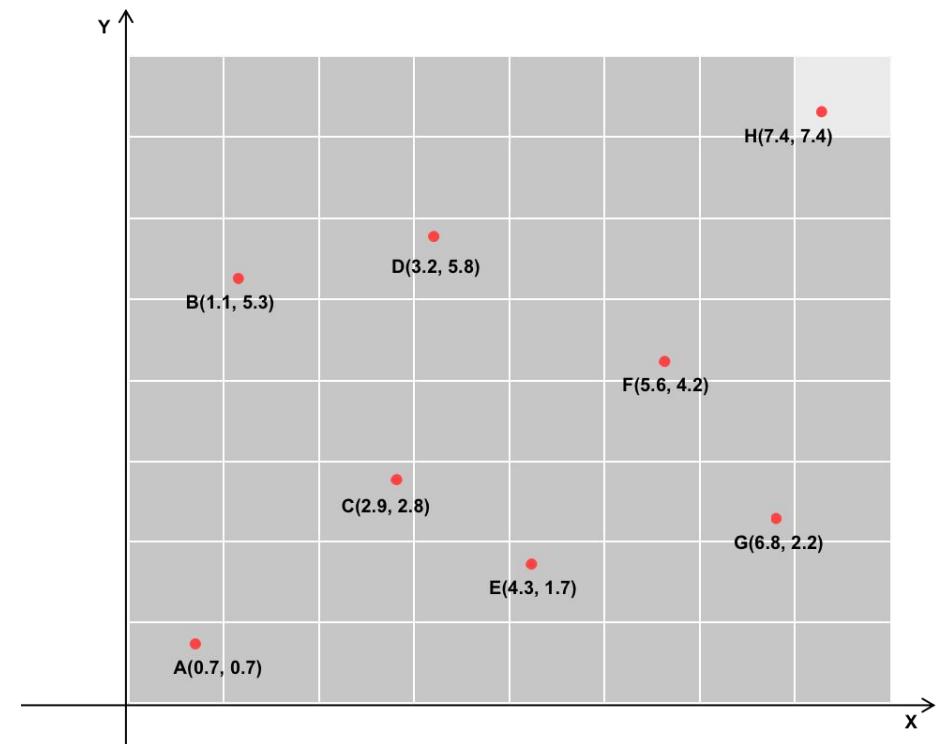
```
[a(image_id, class_id, 0.7, 0.7, w, h),  
 b(image_id, class_id, 1.1, 5.3, w, h),  
 c(image_id, class_id, 2.9, 2.8, w, h),  
 d(image_id, class_id, 3.2, 5.8, w, h),  
 e(image_id, class_id, 4.3, 1.7, w, h),  
 f(image_id, class_id, 5.6, 4.2, w, h),  
 g(image_id, class_id, 6.8, 2.2, w, h),  
 h(image_id, class_id, 7.4, 7.4, w, h)]
```

一、yolo v5解读

(四) 正样本采样细节



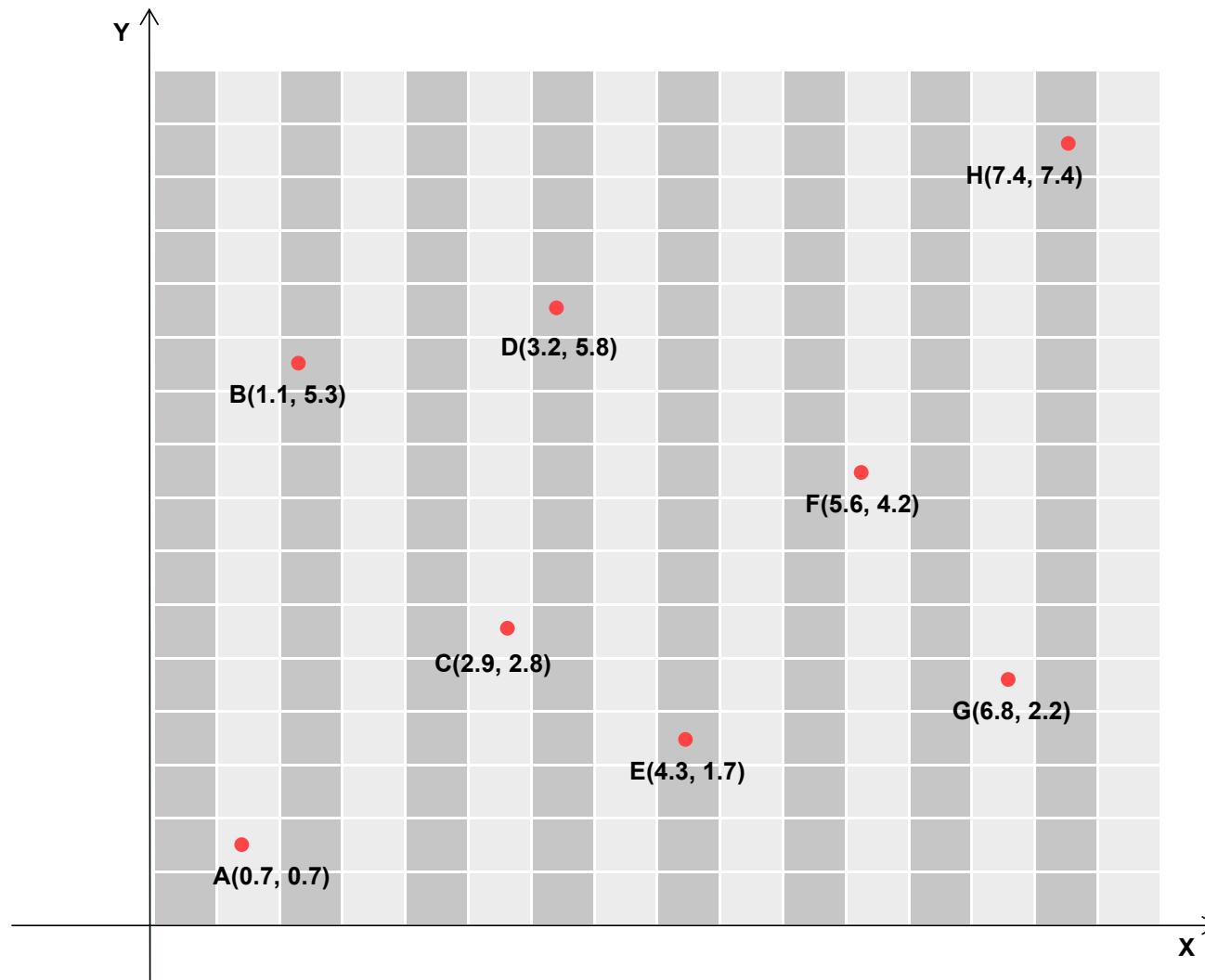
$(g_{xy} > 1)$



$(g_{xi} > 1)$, 其中 $g_{xi} = (w, h) - g_{xy} = (8, 8) - g_{xy}$

一、yolo v5解读

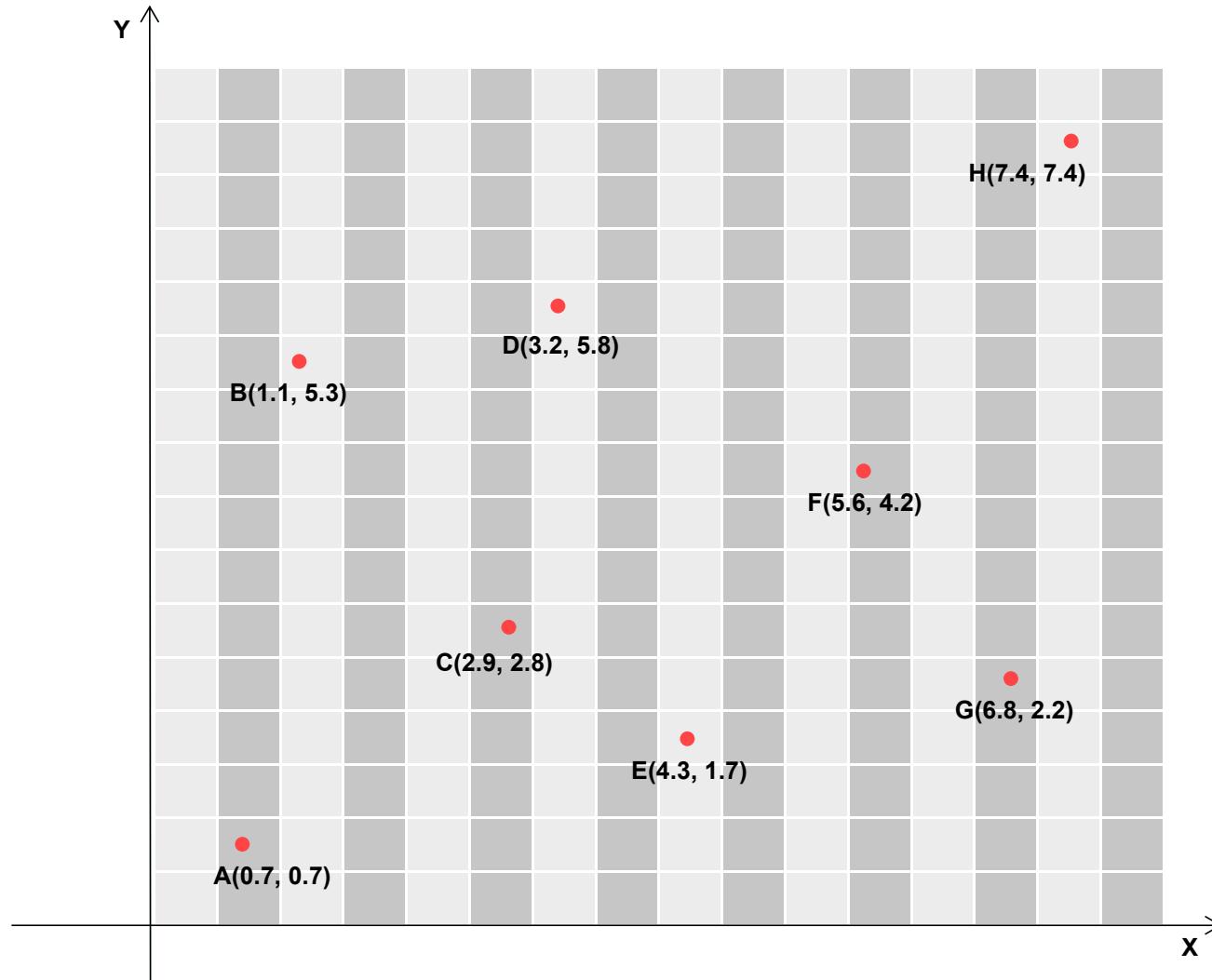
(四) 正样本采样细节



gxy_x % 1 < 0.5

一、yolo v5解读

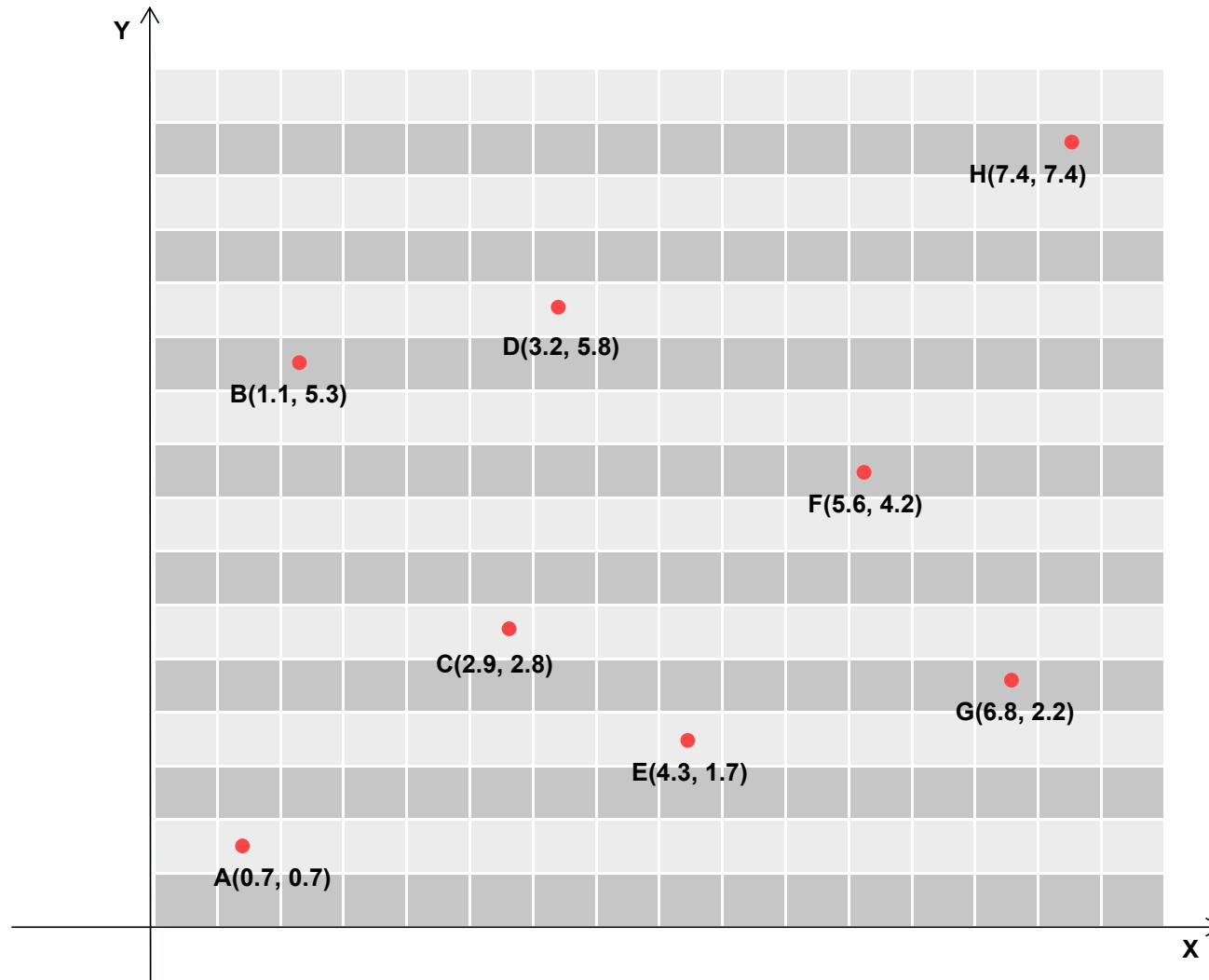
(四) 正样本采样细节



$gxi_x \% 1 < 0.5$

一、yolo v5解读

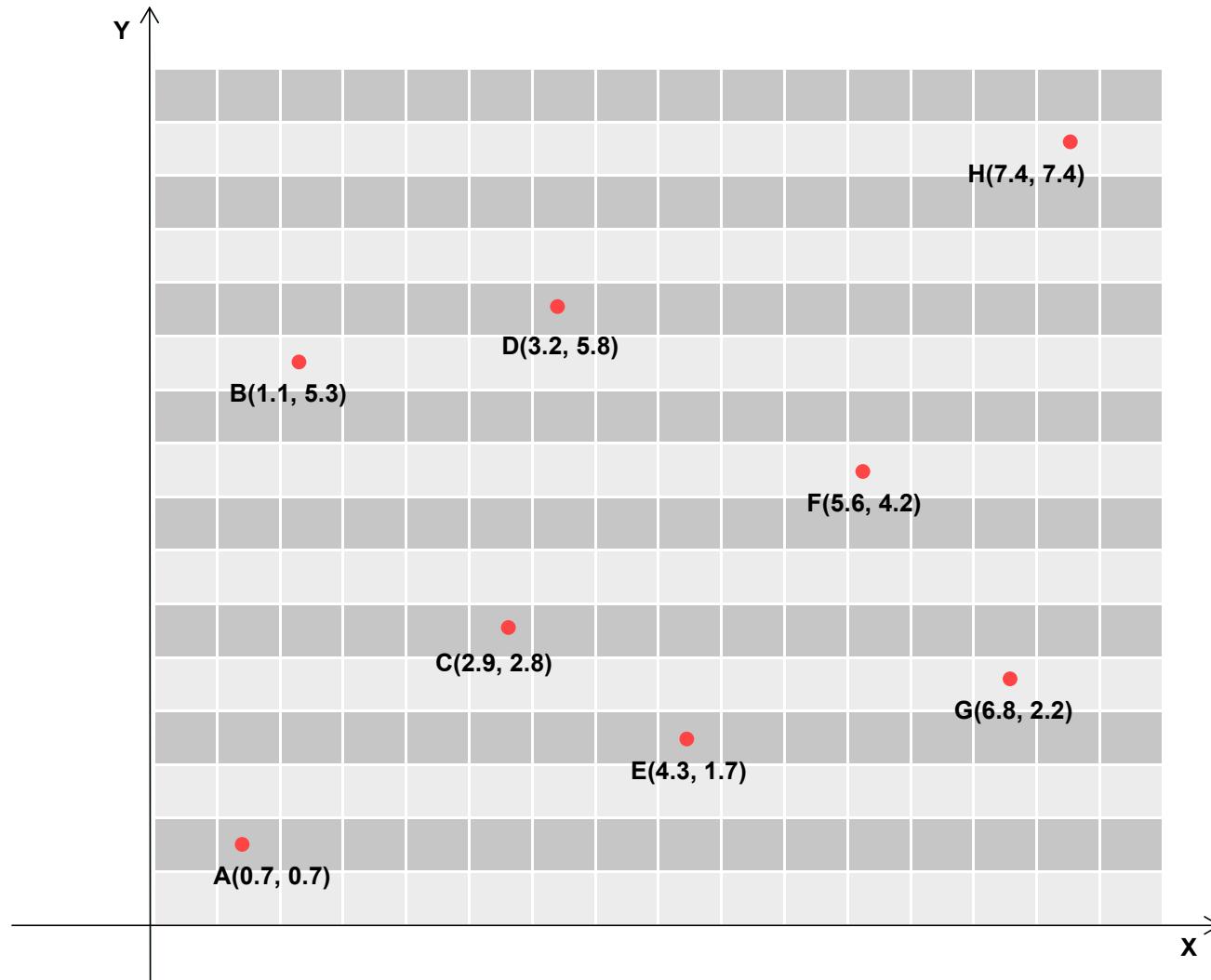
(四) 正样本采样细节



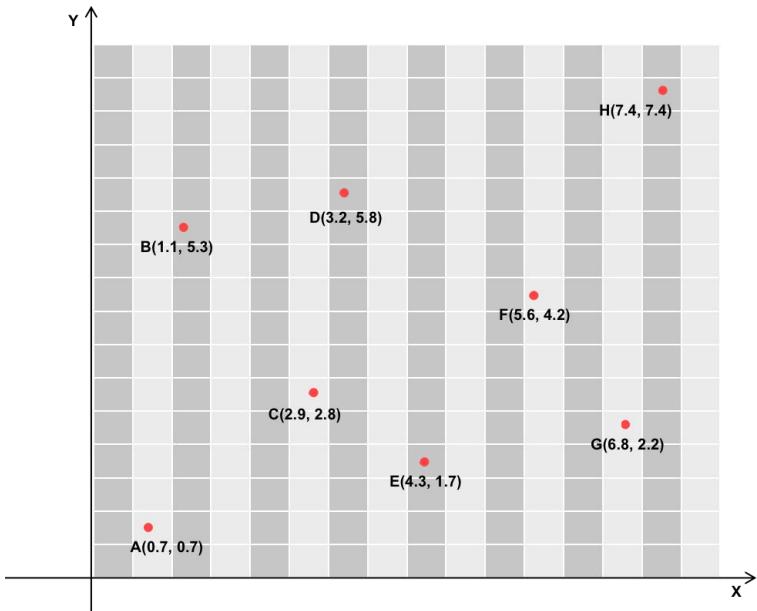
$gxy_y \% 1 < 0.5$

一、yolo v5解读

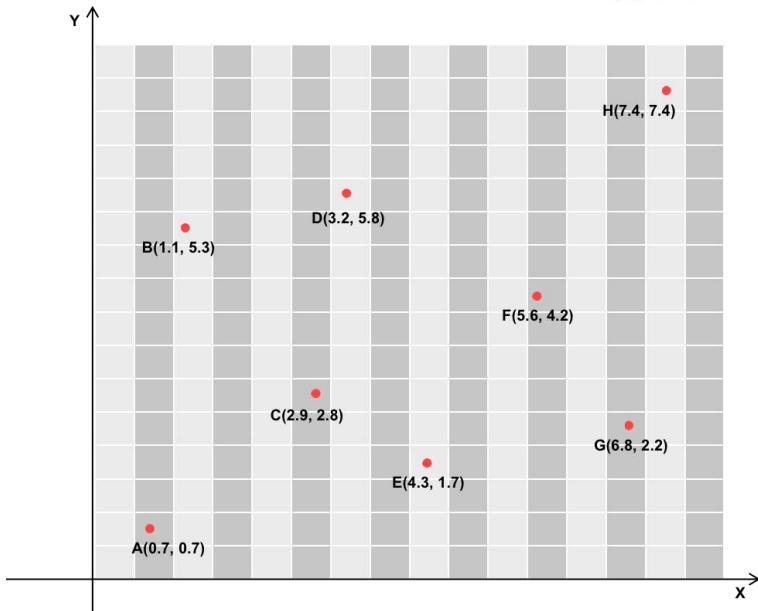
(四) 正样本采样细节



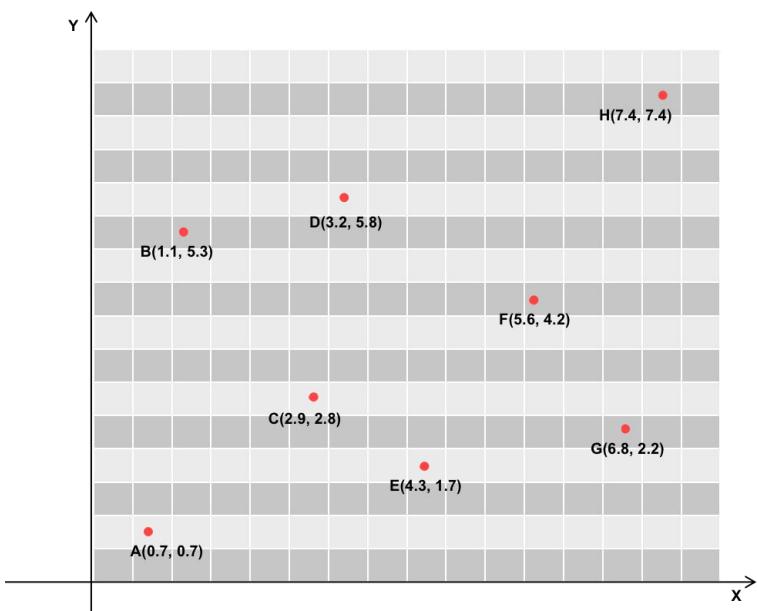
gxi_y % 1 < 0.5



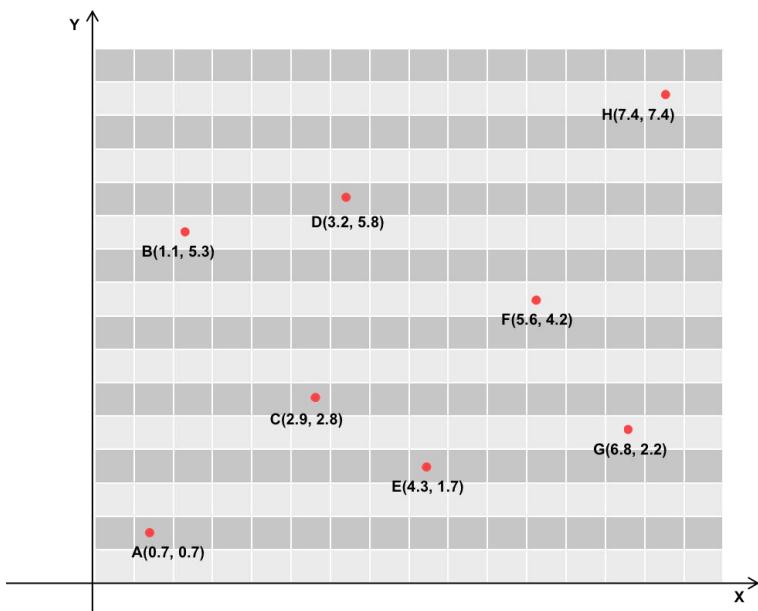
`gxy_x % 1 < 0.5`



`gxi_x % 1 < 0.5`



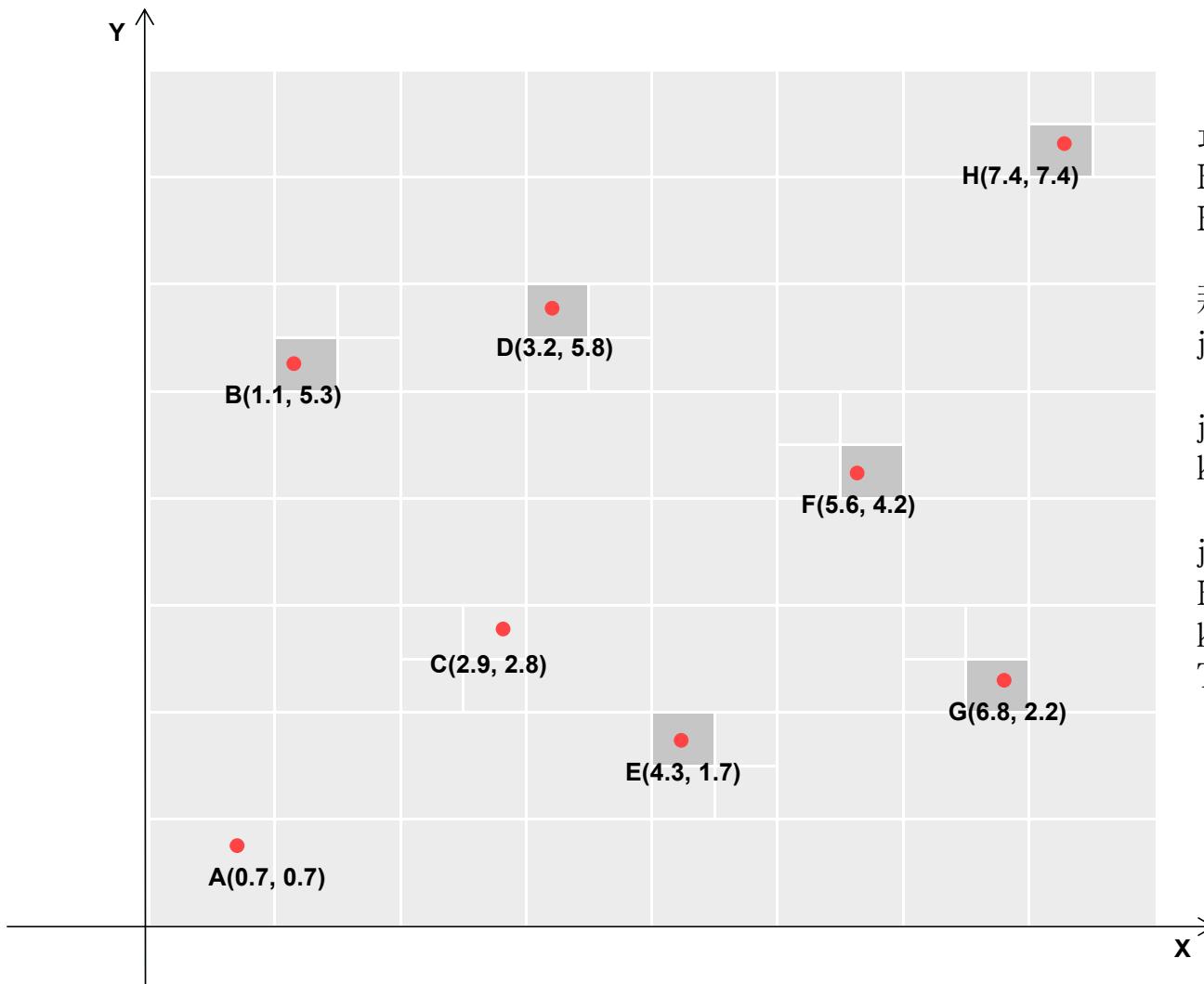
`gxy_y % 1 < 0.5`



`gxi_y % 1 < 0.5`

一、yolo v5解读

(四) 正样本采样细节



最终筛选得到，其中：

B, D, E, H 为 $x \% 1 < 0.5$, 且 $x > 1$

B, F, G, H 为 $y \% 1 < 0.5$, 且 $y > 1$

那么：

$j, k = (g_{xy} \% 1 < 0.5) \& (g_{xy} > 1)$

j 即 $x \% 1 < 0.5$, 且 $x > 1$

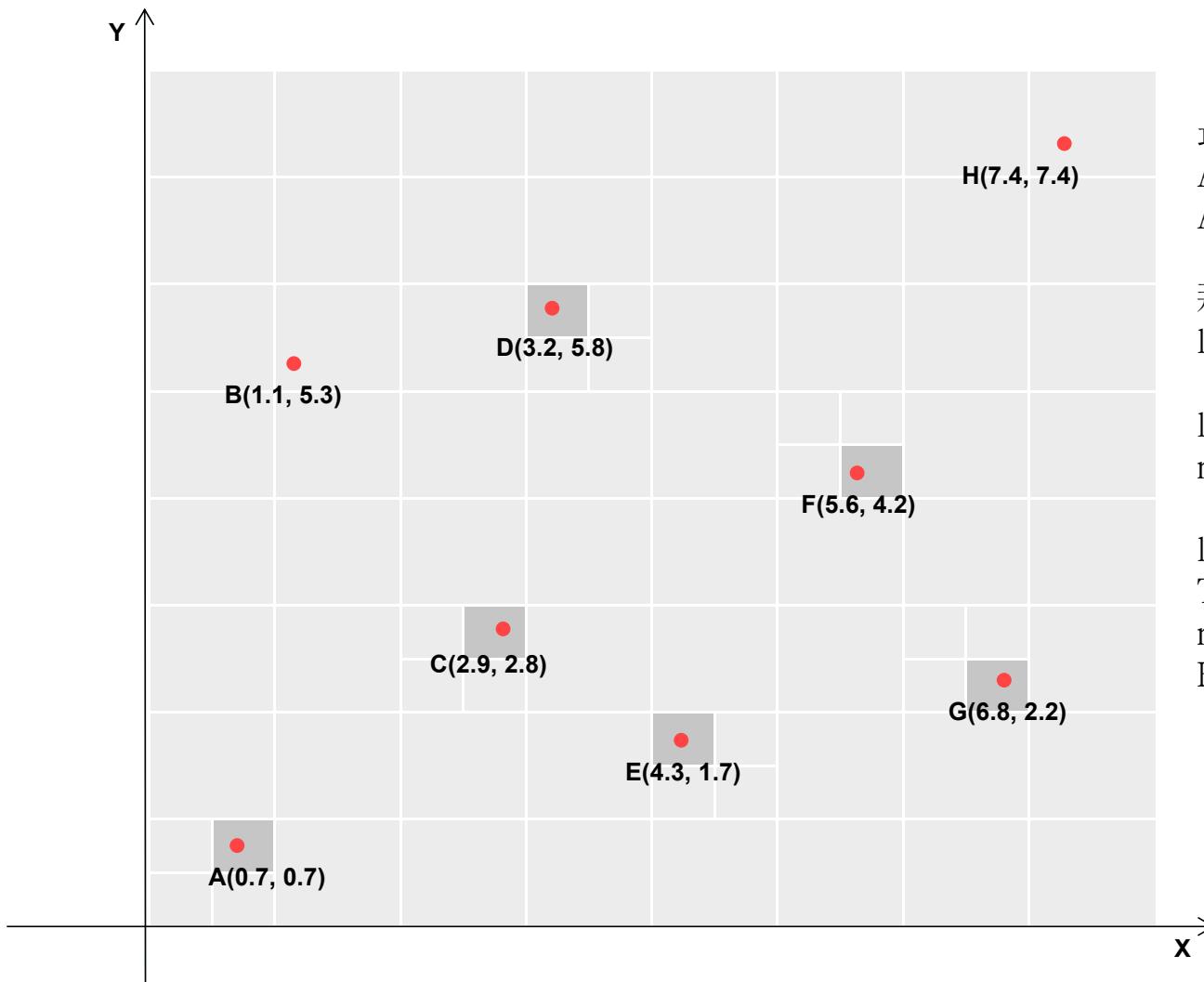
k 即 $y \% 1 < 0.5$, 且 $y > 1$

$j = [\text{False}, \text{True}, \text{False}, \text{True}, \text{True}, \text{False}, \text{False}, \text{True}]$

$k = [\text{False}, \text{True}, \text{False}, \text{False}, \text{False}, \text{True}, \text{True}, \text{True}]$

一、yolo v5解读

(四) 正样本采样细节



$(gxi > 1) \& (gxi \% 1 < 0.5)$

最终筛选得到，其中：

A, C, F, G 为 $x \% 1 < 0.5$, 且 $x > 1$

A, C, D, E 为 $y \% 1 < 0.5$, 且 $y > 1$

那么：

$l, m = (gxi \% 1 < 0.5) \& (gxi > 1)$

l 即 $x \% 1 < 0.5$, 且 $x > 1$

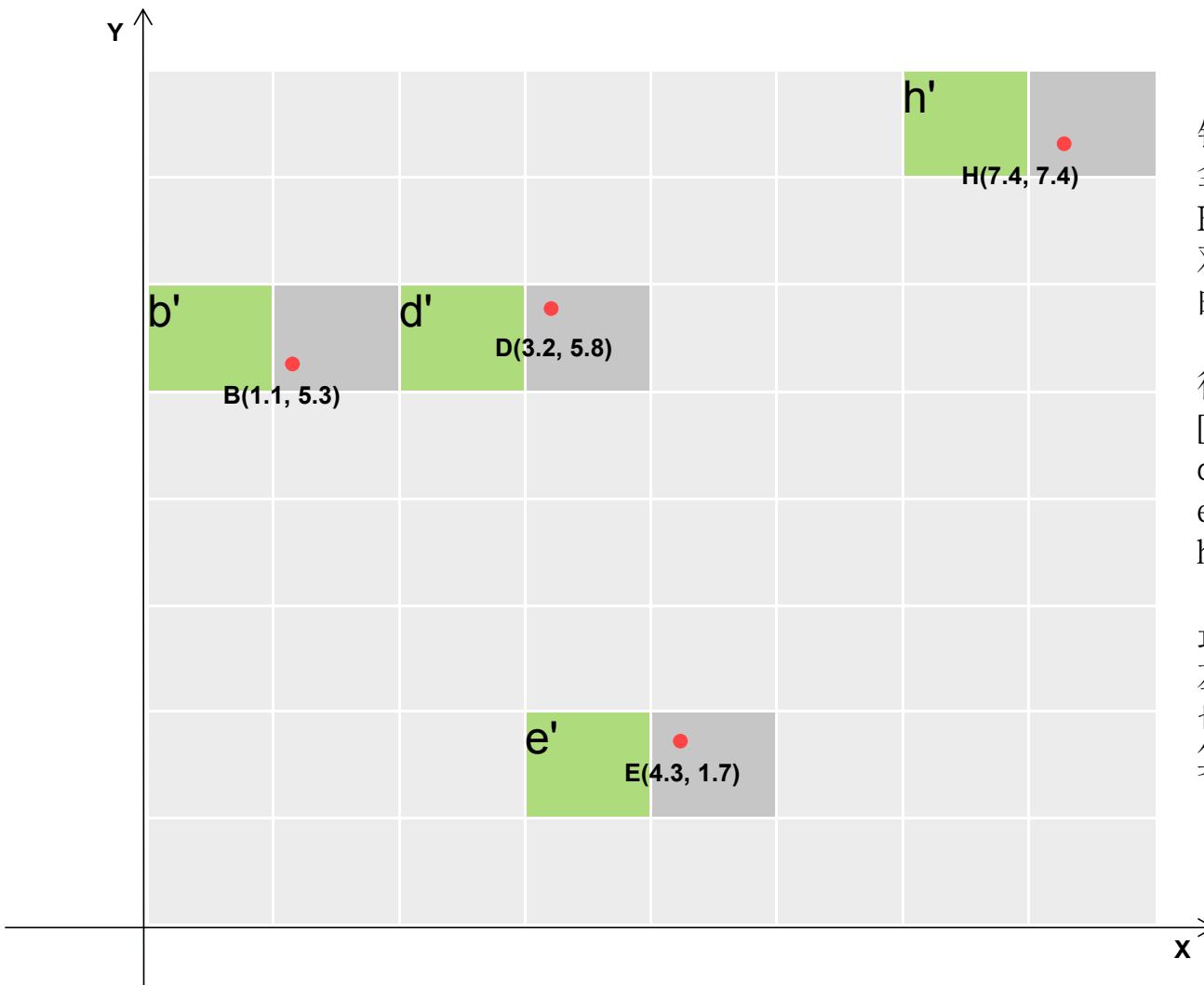
m 即 $y \% 1 < 0.5$, 且 $y > 1$

$l = [\text{True}, \text{False}, \text{True}, \text{False}, \text{False}, \text{True}, \text{True}, \text{False}]$

$m = [\text{True}, \text{False}, \text{True}, \text{True}, \text{True}, \text{False}, \text{False}, \text{False}]$

一、yolo v5解读

(四) 正样本采样细节



针对j做offset处理:

拿 $j = [\text{False}, \text{True}, \text{False}, \text{True}, \text{True}, \text{False}, \text{False}, \text{True}]$

对应的点, B, D, E, H, 减去对应的offset[0.5, 0]

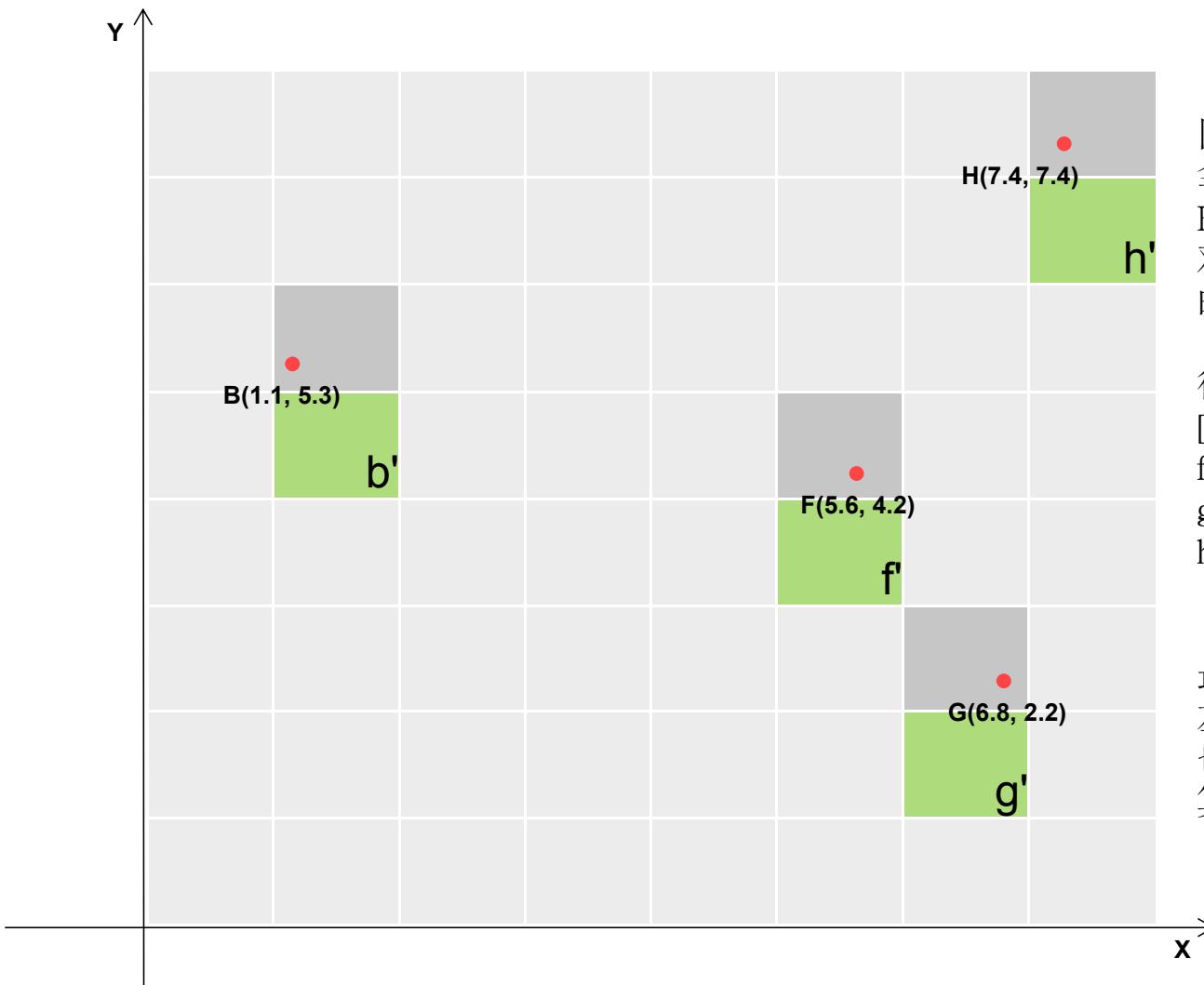
得到:

$[b'(\text{image_id}, \text{class_id}, 0.6, 5.3, w, h),$
 $d'(\text{image_id}, \text{class_id}, 2.7, 5.8, w, h),$
 $e'(\text{image_id}, \text{class_id}, 3.8, 1.7, w, h),$
 $h'(\text{image_id}, \text{class_id}, 6.9, 7.4, w, h)]$

最终 b' d' e' h' 对应回原图位置如左边绿色部分, 也就是该些位置, 也会作为ground truth数据用于计算损失.

一、yolo v5解读

(四) 正样本采样细节



同理k做offset处理:

拿 k= [False, True, False, False, False, True, True, True]

对应的点, B, F, G, H, 减去对应的 offset[0, 0.5]

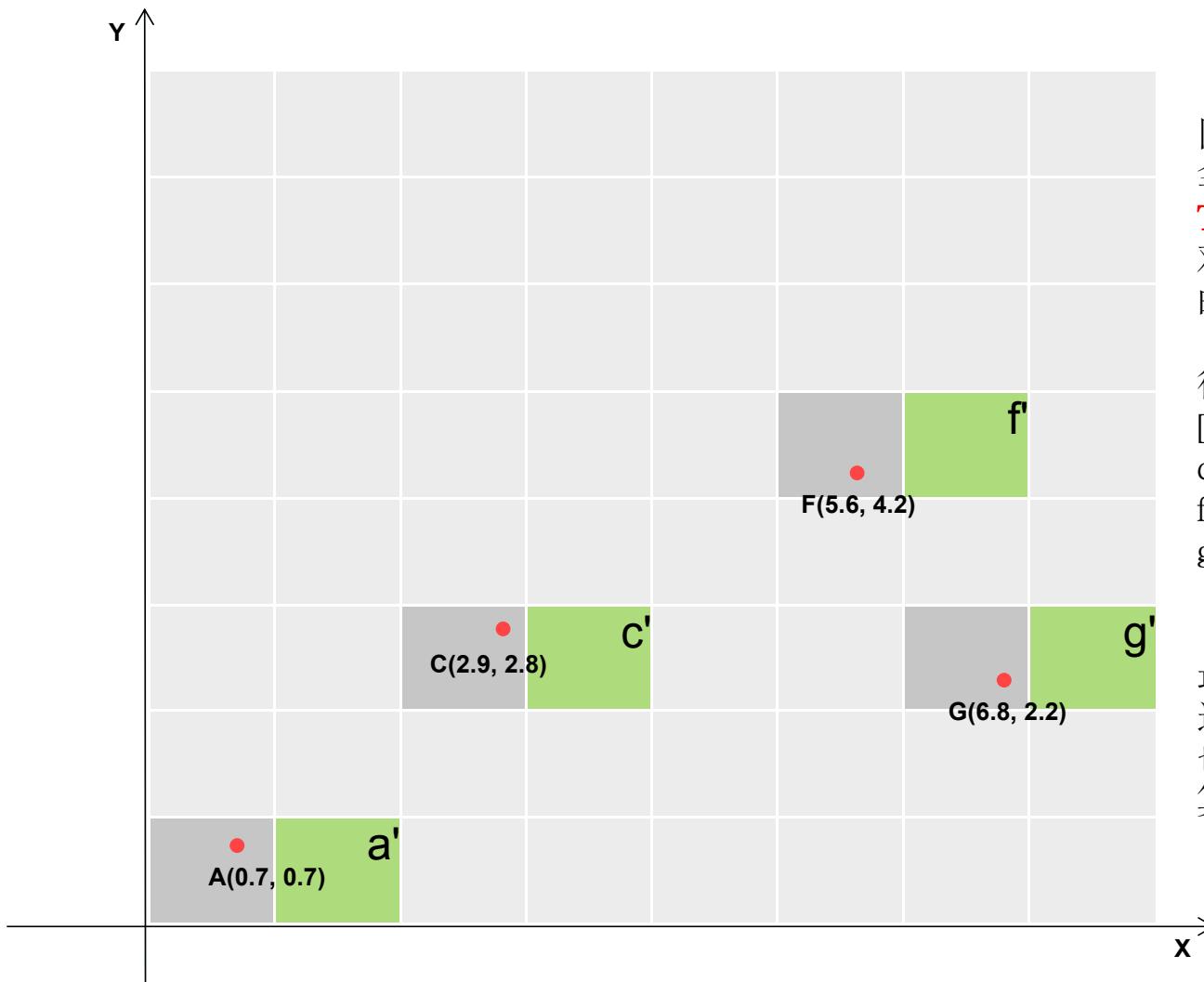
得到:

[b'(image_id, class_id, 1.1, 4.8, w, h),
f'(image_id, class_id, 5.6, 3.7, w, h),
g'(image_id, class_id, 6.8, 1.7, w, h),
h'(image_id, class_id, 7.4, 6.9, w, h)]

最终 b' d' e' h' 对应回原图位置如左边绿色部分, 也就是该些位置, 也会作为ground truth数据用于计算损失.

一、yolo v5解读

(四) 正样本采样细节



同理对做offset处理:

拿 $l = [\text{True}, \text{False}, \text{True}, \text{False}, \text{False}, \text{True}, \text{True}, \text{False}]$

对应的点, A, C, F, G , 加上对应的offset[0.5, 0]

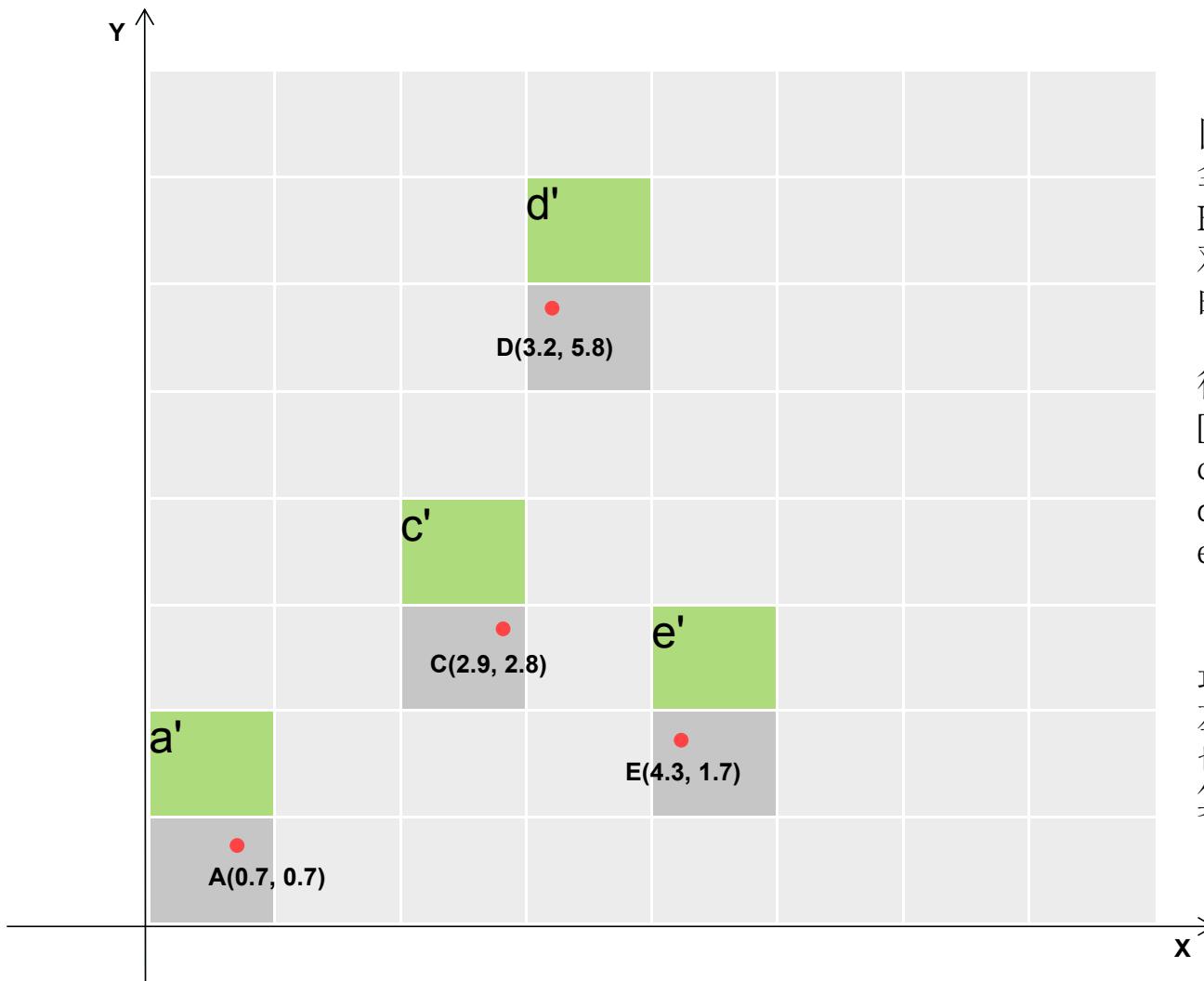
得到:

$[a'(\text{image_id}, \text{class_id}, 1.2, 0.7, w, h),$
 $c'(\text{image_id}, \text{class_id}, 3.4, 2.8, w, h),$
 $f'(\text{image_id}, \text{class_id}, 6.1, 4.2, w, h),$
 $g'(\text{image_id}, \text{class_id}, 7.3, 2.2, w, h)]$

最终 a', c', f', g' 对应回原图位置如左边绿色部分, 也就是该些位置, 也会作为ground truth数据用于计算损失.

一、yolo v5解读

(四) 正样本采样细节



同理m做offset处理:

拿 $m = [\text{True}, \text{False}, \text{True}, \text{True}, \text{True}, \text{False}, \text{False}, \text{False}]$
 对应的点, A, C, D, E , 加上对应的offset[0, 0.5]

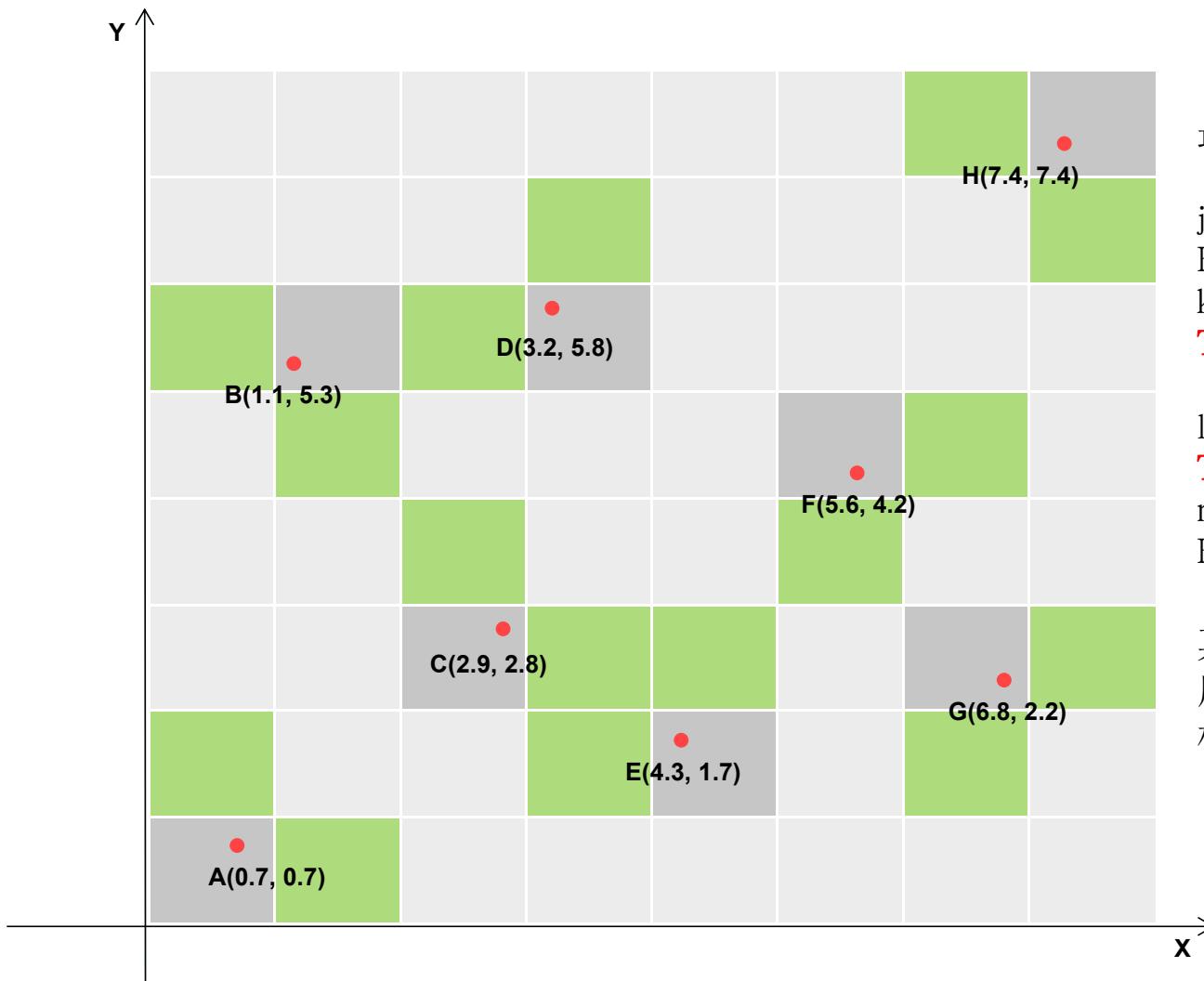
得到:

$[a'(\text{image_id}, \text{class_id}, 0.7, 1.2, w, h),$
 $c'(\text{image_id}, \text{class_id}, 2.9, 3.3, w, h),$
 $d'(\text{image_id}, \text{class_id}, 3.2, 6.3, w, h),$
 $e'(\text{image_id}, \text{class_id}, 4.3, 2.2, w, h)]$

最终 a', c', d', e' 对应回原图位置如左边绿色部分, 也就是该些位置, 也会作为ground truth数据用于计算损失.

一、yolo v5解读

(四) 正样本采样细节



最终合并j, k, l, m:

j = [False, True, False, True, True,
False, False, True]

k = [False, True, False, False, False,
True, True, True]

l = [True, False, True, False, False,
True, True, False]

m = [True, False, True, True, True,
False, False, False]

其中深灰色即原ground truth对应边框数据，绿色为补充正样本边框数据。

一、yolo v5解读

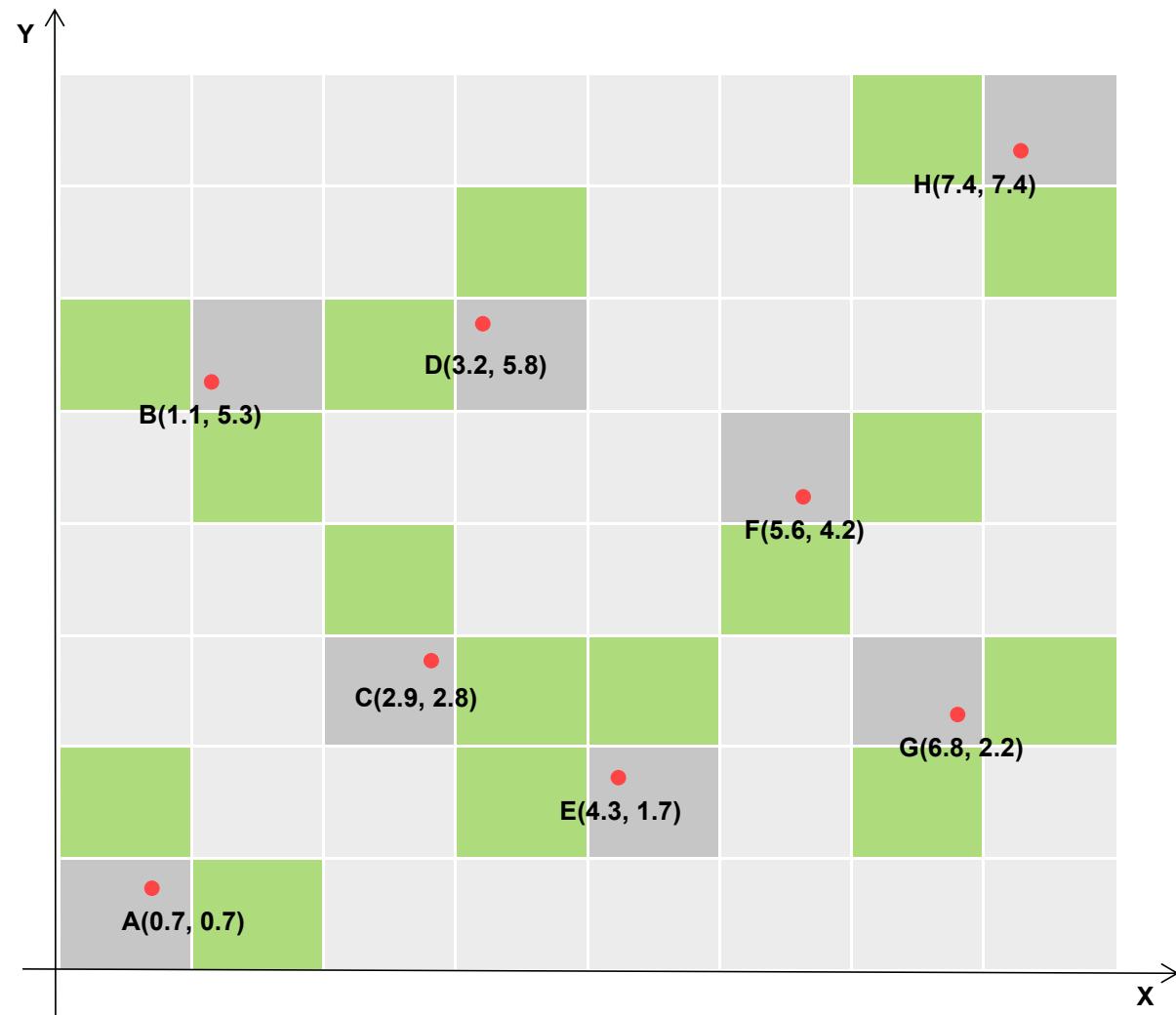
(四) 正样本采样细节

最后得到:



薛定谔的AI

点 => 取整得到grid坐标 => 相减得到该grid的边框xy偏移量



A => grid(0,0) => (0.7,0.7) - (0,0) = (0.7, 0.7)

A^{up} => grid(0,1) => (0.7, 0.7) - (0,1) = (0.7, -0.3)

A^{right} => grid(1,0) => (0.7, 0.7) - (1,0) = (-0.3, 0.7)

B => grid(1,5) => (1.1, 5.3) - (1,5) = (0.1, 0.3)

B^{left} => grid(0, 5) => (1.1, 5.3) - (0, 5) = (1.1, 0.3)

B^{down} => grid(1, 4) => (1.1, 5.3) - (1, 4) = (0.1, 1.3)

C => grid(2,2) => (2.9,2.8) - (2,2) = (0.9,0.8)

C^{up} => grid(2,3) => (2.9,2.8) - (2,3) = (0.9,-0.2)

C^{right} => grid(3,2) => (2.9,2.8) - (3,2) = (-0.1,0.8)

D => grid(3,5) => (3.2, 5.8) - (3,5) = (0.2,0.8)

D^{up} => grid(3,6) => (3.2, 5.8) - (3,6) = (0.2,-0.2)

D^{left} => grid(2,5) => (3.2, 5.8) - (2,5) = (1.2,0.8)

E => grid(4,1) => (4.3, 1.7) - (4,1) = (0.3,0.7)

E^{up} => grid(4,2) => (4.3, 1.7) - (4,2) = (0.3,-0.3)

E^{left} => grid(3,1) => (4.3, 1.7) - (3,1) = (1.3,0.7)

F => grid(5,4) => (5.6, 4.2) - (5,4) = (0.6,0.2)

F^{right} => grid(6,4) => (5.6, 4.2) - (6,4) = (-0.4,0.2)

F^{down} => grid(5,3) => (5.6, 4.2) - (5,3) = (0.6,1.2)

G => grid(6,2) => (6.8, 2.2) - (6,2) = (0.8,0.2)

G^{right} => grid(7,2) => (6.8, 2.2) - (7,2) = (-0.2,0.2)

G^{down} => grid(6,1) => (6.8, 2.2) - (6,1) = (0.8,1.2)

H => grid(7,7) => (7.4, 7.4) - (7,7) = (0.4,0.4)

H^{left} => grid(6,7) => (7.4, 7.4) - (6,7) = (1.4,0.4)

H^{down} => grid(7,6) => (7.4, 7.4) - (7,6) = (0.4,1.4)

一、yolo v5解读

(五) 损失计算细节 box loss

GIoU: [《Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression》](#)

$$\lambda_{coord} \sum_i^S \sum_j^{anchors} 1_{ij}^{obj} \sum_{l \in [x,y,w,h]} (l_{ij}^{true} - l_{ij}^{pred})^2$$

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

$$\mathcal{L}_{IoU} = 1 - IoU$$

$$GIoU = IoU - \frac{|C \setminus (A \cup B)|}{|C|}$$

$$\mathcal{L}_{GIoU} = 1 - GIoU$$

左边是yolov3里用的L2损失，中间是传统的IoU及其边框损失表达式，右边是GIoU及其边框损失表达式

1. GIoU在IoU的基础上考虑多了**非交叉面积比例**，如右图1红色虚线框就是A,B边框的最小包围框，灰色斜线面积占整个红色边框面积就是非交叉面积占比

2. 对比L2损失，IoU和GIoU具有**尺度不变性**，意味着当目标边框等比放大时，损失能依旧保持同样的量级，无需针对大小不同边框分别处理。

3. 对比IoU损失，L2和GIoU具有**偏离趋势度量能力**，如下图2，传统IoU=0时，边框距离的远近已经对最终损失都是一样，但是GIoU随着两个边框距离越远，表现得越接近-1，换算成损失就是越大，同样GIoU会驱使模型预测边框分布于真实边框的上下左右方向，对斜方向预测结果施加更大损失，如下图3.

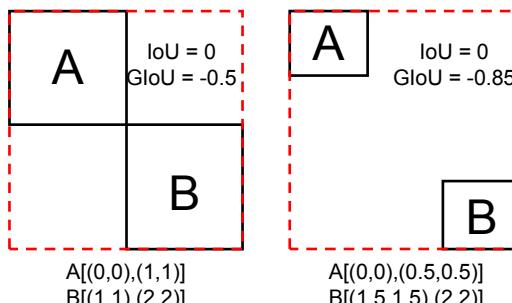


图2

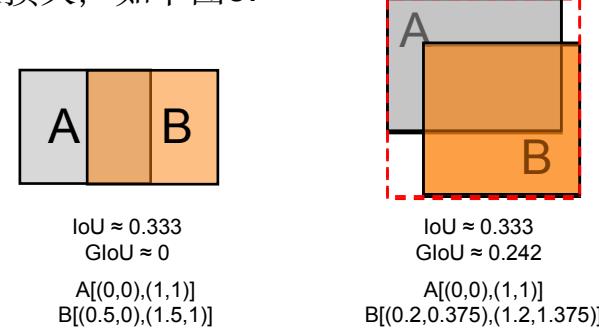


图3

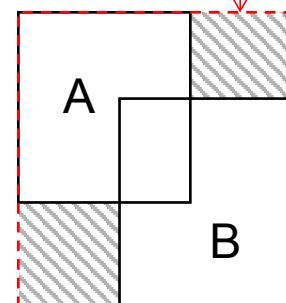


图1

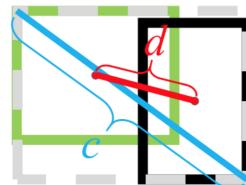
一、yolo v5解读

(五) 损失计算细节 box loss

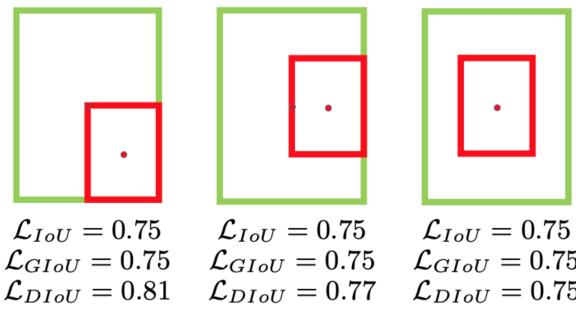
DIoU: [《Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression》](#)

$$\mathcal{L}_{DIoU} = 1 - IoU + \frac{\rho^2(\mathbf{b}, \mathbf{b}^{gt})}{c^2}$$

DIoU损失在1-IoU的基础上，增加了**中心点距离占比惩罚项**，其中惩罚项分子是预测边框中心点与真实边框中心点的距离，分母是预测边框与真实边框的最小包围框对角线长，如下图d和c:



1. 对比GIoU损失，DIoU能更好度量预测边框和真实边框的**中心点距离和方向**，表现如下图所示，绿色真实边框，红色预测边框，当预测边框与真实边框互相包含，或者互相垂直交叉，水平交叉，GIoU会退化成为IoU，从而失去非交叉占比的惩罚项，而DIoU依旧能为模型提供更好的梯度方向：



- 与GIoU损失一样，DIoU也具有**尺度不变性**，意味着当目标边框等比放大时，损失能依旧保持同样的量级，无需针对大小不同边框分别处理。
- 与GIoU损失一样，DIoU损失值域空间为[0,2]，当完美拟合损失0，当距离无限远且不交叉时，损失是2。

一、yolo v5解读

(五) 损失计算细节 box loss

CIoU: [《Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression》](#)

[《Enhancing Geometric Factors in Model Learning and Inference for Object Detection and Instance Segmentation》](#)

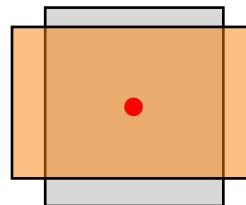
$$\mathcal{L}_{CIoU} = 1 - IoU + \frac{\rho^2(\mathbf{p}, \mathbf{p}^{gt})}{c^2} + \alpha V.$$

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

$$\alpha = \frac{v}{(1 - IoU) + v}$$

CIoU损失在DIoU的基础上，增加了宽高比惩罚项，其中v为真实边框与预测边框的宽高比损失， α 为宽高比损失系数：

1. 对比DIoU损失，当预测边框和真实边框的中心点重合，CIoU具有更好的宽高拟合效果，如下图所示，预测边框与真实边框中心点重合，DIoU损失中的中心点距离惩罚项=0，DIoU损失退化成IoU损失，但是此时CIoU仍有宽高比损失惩罚，能进一步调整宽高比例：



2. CIoU综合了IoU的交叉面积占比损失，DIoU的中心点偏移损失，以及自身宽高比损失3种度量优点。

一、yolo v5解读

(五) 损失计算细节 box loss

CloU: [《Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression》](#)

[《Enhancing Geometric Factors in Model Learning and Inference for Object Detection and Instance Segmentation》](#)

$$\mathcal{L}_{CIoU} = 1 - IoU + \frac{\rho^2(\mathbf{p}, \mathbf{p}^{gt})}{c^2} + \alpha V.$$

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

$$\alpha = \frac{v}{(1 - IoU) + v}$$

原论文CloU损失在实现上做了一点小调整，在求导时 α 作为常数项不参与梯度更新，只针对 v 里的 w 和 h 分别求导，会得到如下：

$$\frac{\partial v}{\partial w} = \frac{8}{\pi^2} \left(\arctan \frac{w^{gt}}{h^{gt}} - \arctan \frac{w}{h} \right) \times \frac{h}{w^2 + h^2},$$

$$\frac{\partial v}{\partial h} = -\frac{8}{\pi^2} \left(\arctan \frac{w^{gt}}{h^{gt}} - \arctan \frac{w}{h} \right) \times \frac{w}{w^2 + h^2}.$$

1. 其中 w^2+h^2 通常会由于 w 或者 h 太小而造成反向传播的时候梯度爆炸，所以原作者[最初版本](#)的实现如下：

with torch.no_grad():

arctan = torch.atan(w2 / h2) - torch.atan(w1 / h1)

v = (4 / (math.pi ** 2)) * torch.pow((torch.atan(w2 / h2) - torch.atan(w1 / h1)), 2)

S = 1 - iou

alpha = v / (S + v)

w_temp = 2 * w1

ar = (8 / (math.pi ** 2)) * arctan * ((w1 - w_temp) * h1)

cious = iou - (u + alpha * ar)

其中 α 和 v 均不参与梯度更新，只有 ar 处直接写成了求导形式，最后对 w, h 求导只会剩下 $h, -w$ ，没有 w^2+h^2

一、yolo v5解读

(五) 损失计算细节 box loss

CloU: [《Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression》](#)

[《Enhancing Geometric Factors in Model Learning and Inference for Object Detection and Instance Segmentation》](#)

$$\begin{aligned}\mathcal{L}_{CIoU} &= 1 - IoU + \frac{\rho^2(\mathbf{p}, \mathbf{p}^{gt})}{c^2} + \alpha V \\ v &= \frac{4}{\pi^2} \left(\arctan \frac{w^{gt}}{h^{gt}} - \arctan \frac{w}{h} \right)^2 \\ \alpha &= \frac{v}{(1 - IoU) + v}\end{aligned}$$

在最新的CloU, 实现上改为如下:

```
v = (4 / (math.pi ** 2)) * torch.pow((torch.atan(w2 / h2) - torch.atan(w1 / h1)), 2)
with torch.no_grad():
    S = 1 - iou
    alpha = v / (S + v)
    ciou = iou - (u + alpha * v)
    ciou = torch.clamp(ciou,min=-1.0,max = 1.0)
```

其中同样的alpha不参与参数的梯度更新, 只是作为一个常数, 但是v的修改已经默认了不对w²+h²问题做额外处理, 早期的版本虽然兼顾了w²+h²对最终梯度问题的影响, 反向传播形式没变, 但是正向表达式中的v变了, yolov5由于对wh有做进一步筛选, 所以避免了w²+h²过小对梯度的影响。

一、yolo v5解读

(五)、损失计算细节 object loss

yolov5的object loss默认用的01二分类交叉熵:

这里直接拿CIoU作为预测的目标

score_iou = iou.detach().clamp(0).type(tobj.dtype)

这里其实self.gr最开始作者的设想是综合yolov2 v3 v4的做法，在目标概率1和CIoU之间做一个加权平均

后面就全部用CIoU了

tobj[b, a, gj, gi] = (1.0 - self.gr) + self.gr * score_iou

self.BCEobj是nn.BCEWithLogitsLoss

obji = self.BCEobj(pi[..., 4], tobj)

由于yolov5最终预测3层，每层的object损失系数分别为:[4.0, 1.0, 0.4]

lobj += obji * self.balance[i]

假如object loss参数fl_gamma>0,会在object loss的基础上加一层focal loss, 用于平衡正负样本损失差异

pred_prob = torch.sigmoid(pred)

p_t = true * pred_prob + (1 - true) * (1 - pred_prob)

这里alpha=0.25,gamma=1.25,对于未正确分类的样本,会给予更高的权重

alpha_factor = true * self.alpha + (1 - true) * (1 - self.alpha)

modulating_factor = (1.0 - p_t) ** self.gamma

loss *= alpha_factor * modulating_factor

一、yolo v5解读

(五) 损失计算细节 class loss

yolov5的class loss默认用的01二分类交叉熵:

```
# 这里的cn=0.05, cp=0.95是作者参考这篇论文做的类别平滑处理,  
# 如果是多分类, cp = 1.0 - label_smoothing, cn = label_smoothing / num_classes  
# 如果是01二分类, cp = (1.0 - label_smoothing), cn = 0.5 * label_smoothing  
# 论文是直接处理成0/1二分类交叉熵, 所以是下者  
t = torch.full_like(ps[:, 5:], self.cn, device=device)  
t[range(n), tcls[i]] = self.cp  
lcls += self.BCEcls(ps[:, 5:], t)
```

→

```
# 假如object loss参数fl_gamma>0,会在object loss的基础上加一层focal loss, 用于平衡正负样本损失差异  
pred_prob = torch.sigmoid(pred)  
p_t = true * pred_prob + (1 - true) * (1 - pred_prob)
```

```
# 这里alpha=0.25,gamma=1.25,对于未正确分类的样本,会给予更高的权重  
alpha_factor = true * self.alpha + (1 - true) * (1 - self.alpha)  
modulating_factor = (1.0 - p_t) ** self.gamma  
loss *= alpha_factor * modulating_factor
```

YOLOv5

by



完 结

欢迎关注b站：薛定谔的AI



Download on the
App Store



Coming Soon on
Google Play