深蓝学院
shenlanxueyuan.com

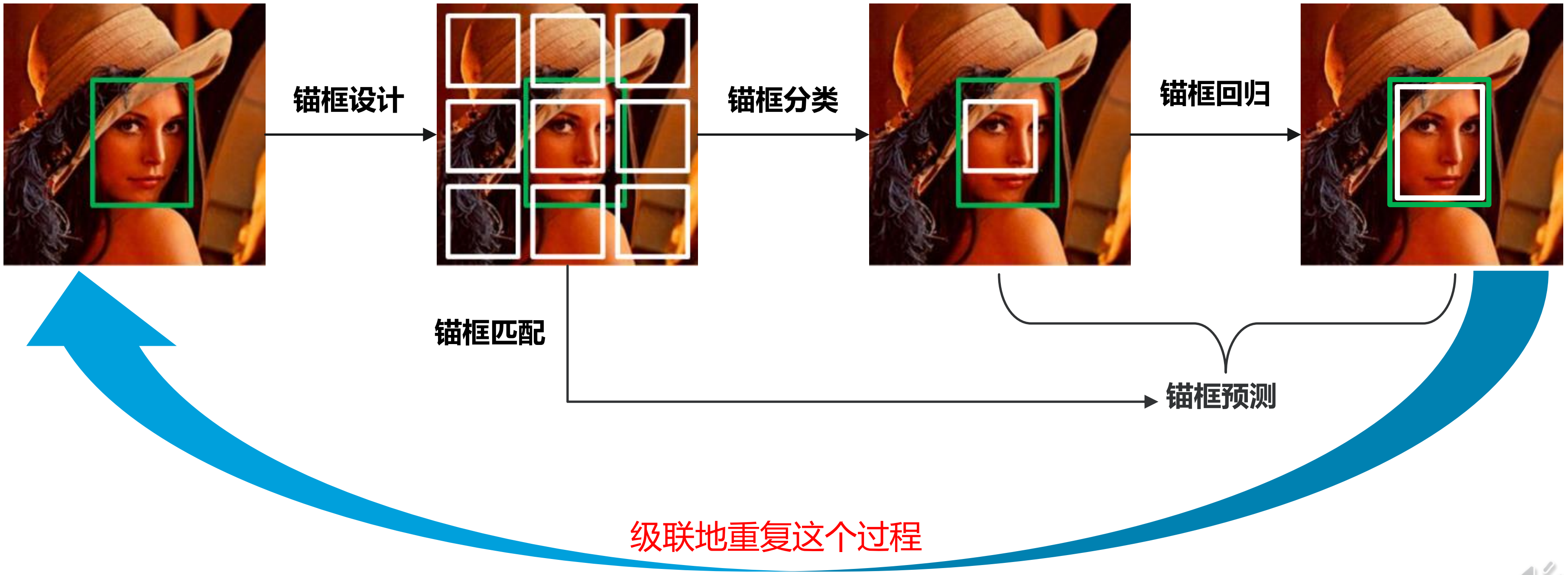
通用物体检测

主讲人 张士峰

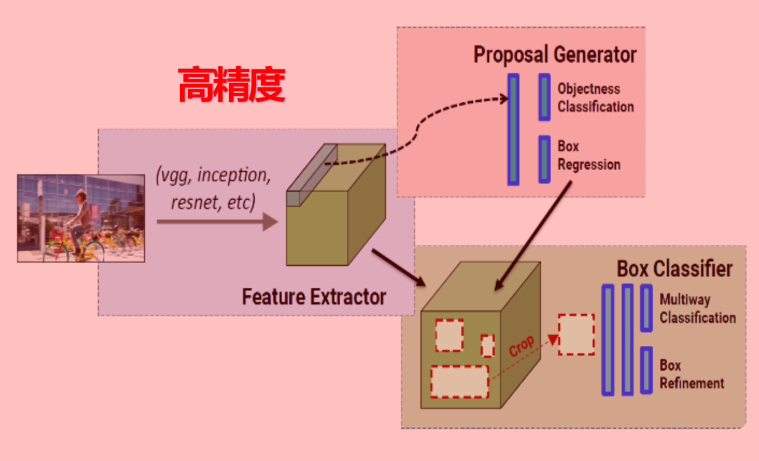
中国科学院自动化研究所
模式识别国家重点实验室

内容回顾：基于锚框的检测算法

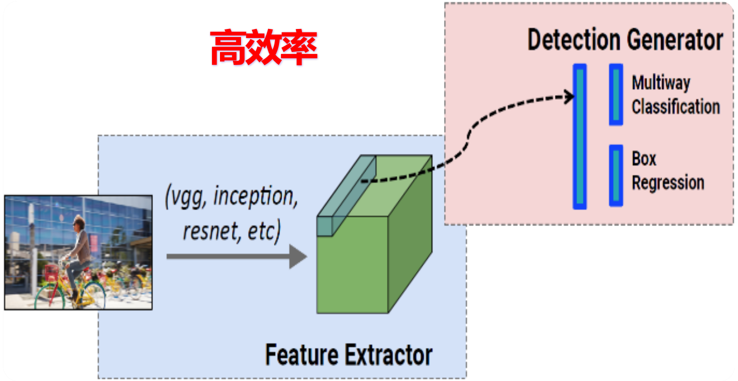
锚框机制是该类物体检测算法的核心



内容回顾：多阶段法

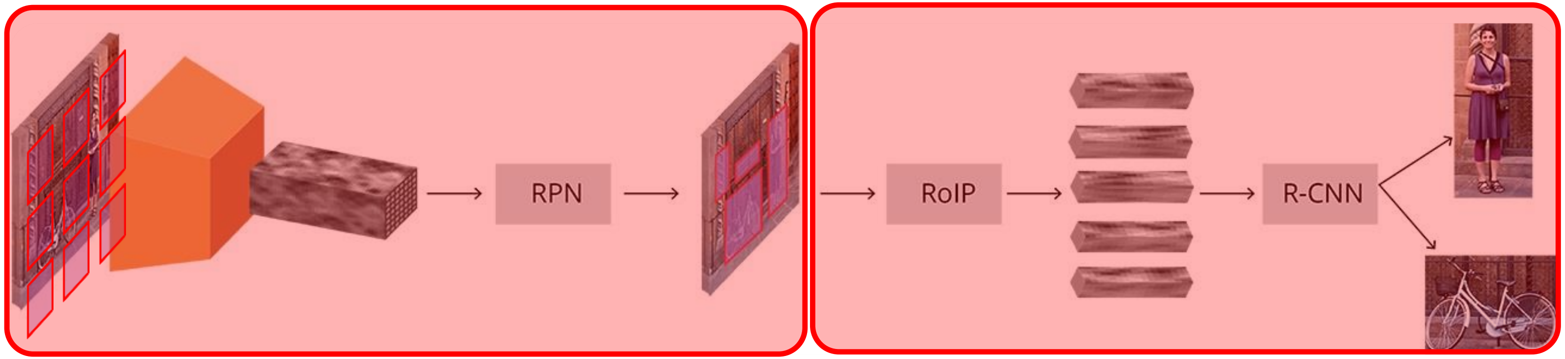


多阶段法



单阶段法

内容回顾：多阶段法Faster R-CNN



Faster R-CNN中RPN步骤：

① 整张图传入VGG16或ResNet提取特征

② 选择下采样倍数为16的特征层作为检测层

③ 根据检测层预设一系列大小和比例的锚框（9个）

④ 对锚框进行二分类和回归得到若干候选区域

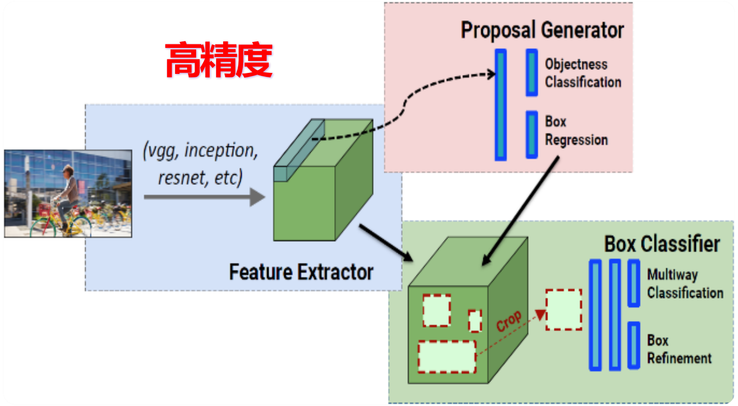
Faster R-CNN中Fast R-CNN步骤：

① 利用RoIPooling在检测层的特征上提取每个候选区域对应的特征

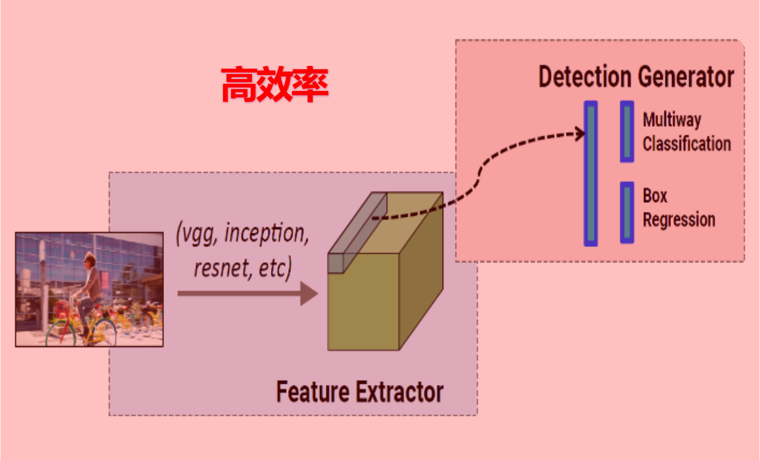
② 输入CNN/FC子网络来增强候选区域的特征

③ 对候选区域进行多分类和回归得到检测结果

基于锚框的物体检测

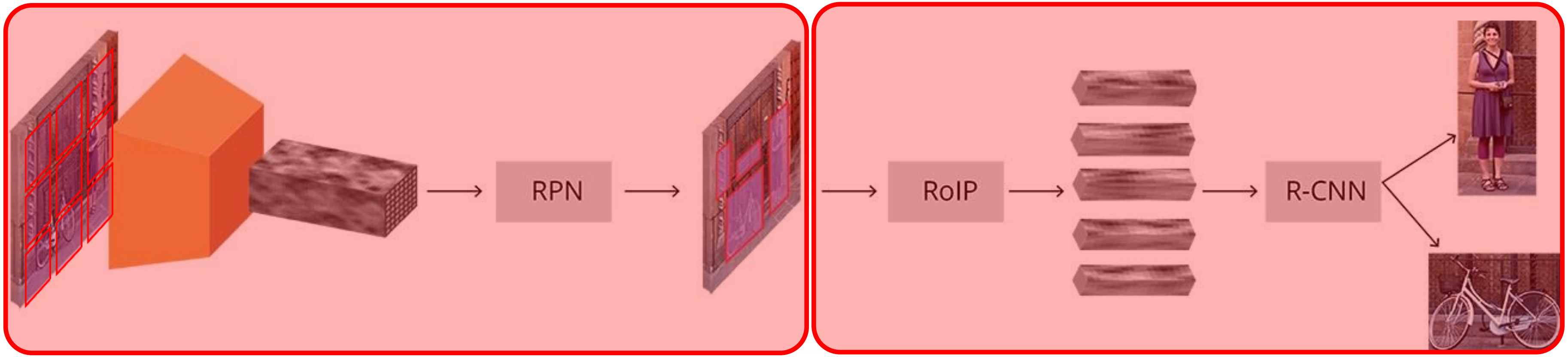


多阶段法



单阶段法

基于锚框的单阶段检测算法



Faster R-CNN中RPN步骤：

① 整张图传入VGG16或ResNet提取特征

② 选择下采样倍数为16的特征层作为检测层

③ 根据检测层预设一系列大小和比例的锚框（9个）

④ 对锚框进行二分类和回归得到若干候选区域

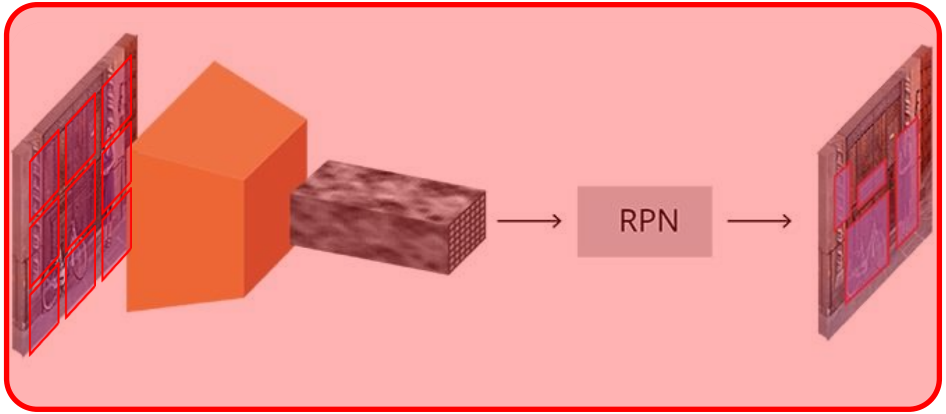
Faster R-CNN中Fast R-CNN步骤：

① 利用RoIPooling在检测层的特征上提取每个候选区域对应的特征

② 输入CNN/FC子网络来增强候选区域的特征

③ 对候选区域进行多分类和回归得到检测结果

基于锚框的单阶段检测算法



Faster R-CNN中RPN步骤：

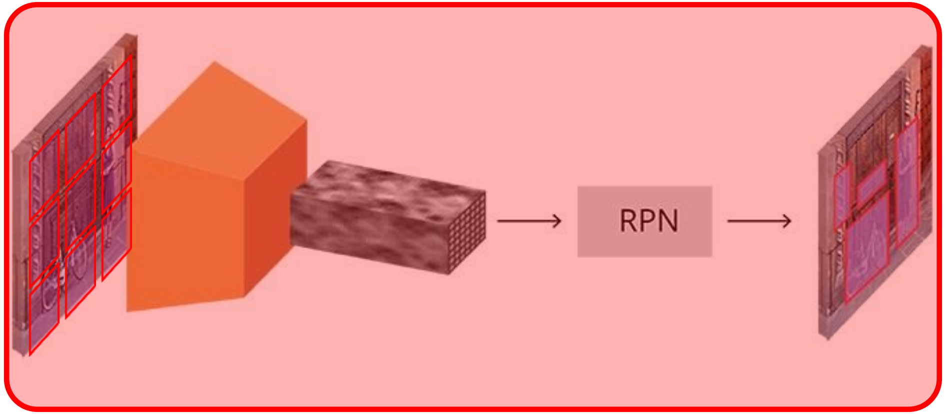
① 整张图传入VGG16或ResNet提取特征

② 选择下采样倍数为16的特征层作为检测层

③ 根据检测层预设一系列大小和比例的锚框（9个）

④ 对锚框进行二分类和回归得到若干候选区域

基于锚框的单阶段检测算法



Faster R-CNN中RPN步骤：

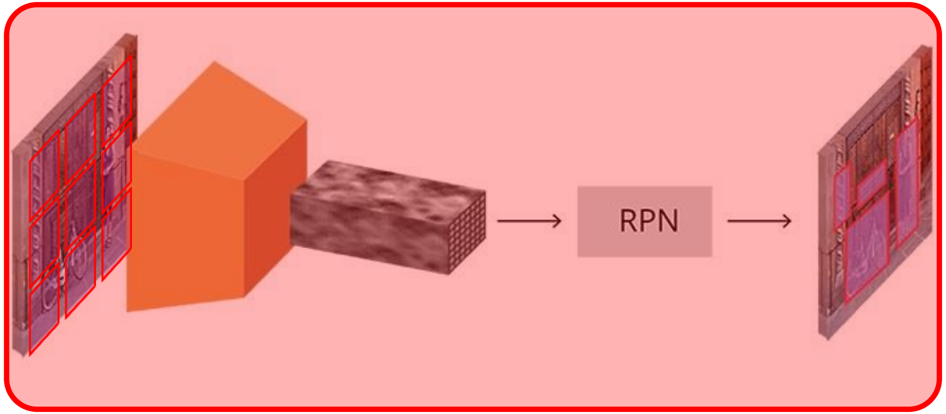
① 整张图传入VGG16或ResNet提取特征

② 选择下采样倍数为16的特征层作为检测层

③ 根据检测层预设一系列大小和比例的锚框（9个）

④ 对锚框进行~~二分类~~ **多分类**和回归得到~~候选区域~~ **检测结果**

基于锚框的单阶段检测算法



Faster R-CNN中RPN步骤：

① 整张图传入VGG16或ResNet提取特征

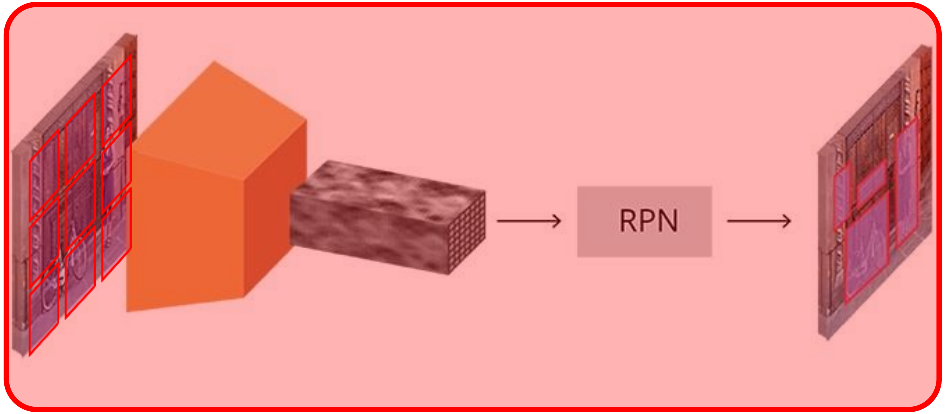
② 选择下采样倍数为16的特征层作为检测层

③ 根据检测层预设一系列大小和比例的锚框（9个）

④ 对锚框进行~~二分类~~ **多分类**和回归得到~~候选区域~~ **检测结果**

基于锚框的单阶段检测算法流程

基于锚框的单阶段检测算法



Faster R-CNN中RPN步骤：

① 整张图传入VGG16或ResNet提取特征

② 选择下采样倍数为16的特征层作为检测层

③ 根据检测层预设一系列大小和比例的锚框（9个）

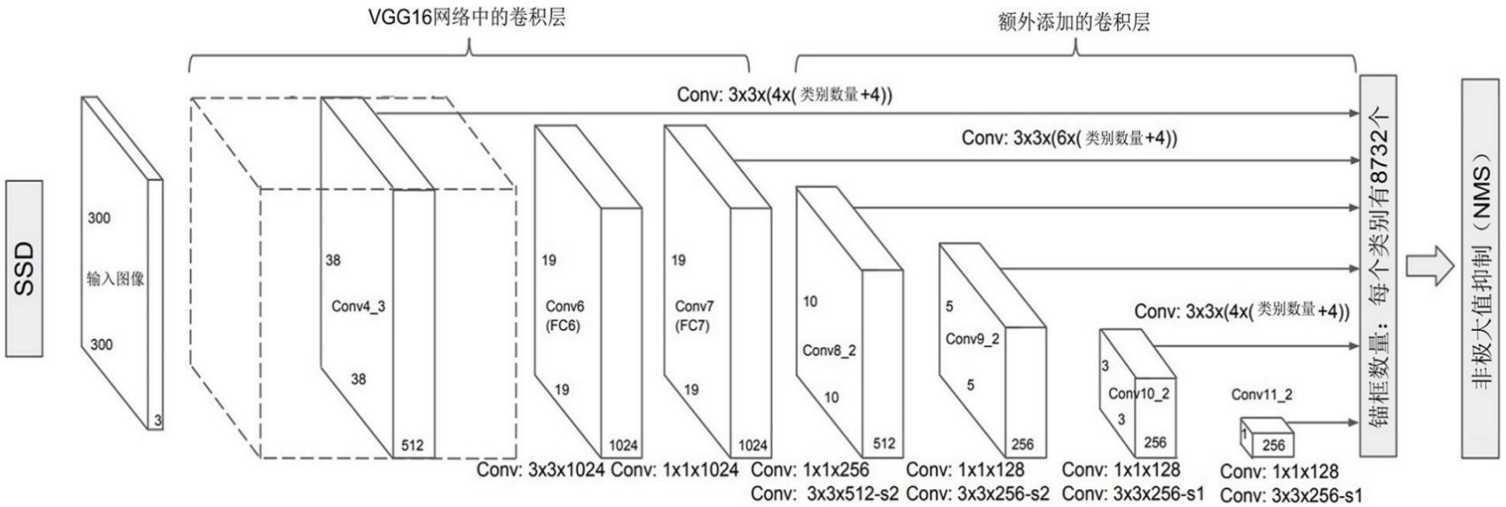
④ 对锚框进行~~二分类~~ **多分类**和回归得到~~候选区域~~ **检测结果**

基于锚框的单阶段检测算法流程

SSD

RetinaNet

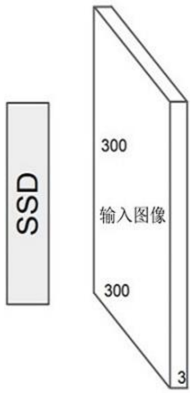
基于锚框的单阶段检测算法：SSD (Single Shot MultiBox Detector)



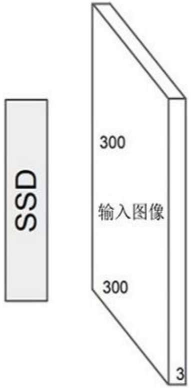
- 输入图像：经过丰富的数据增广，得到300x300或512x512的输入图像
- 基础网络：ImageNet预训练的VGG16网络 + 一些额外的卷积层
- 多检测层：选择6个特征层作为检测层，每层关联不同的锚框
- 难负样本挖掘：利用误差损失值的大小，选择出3倍正样本的负样本参与训练

SSD: Single Shot MultiBox Detector, ECCV 2016

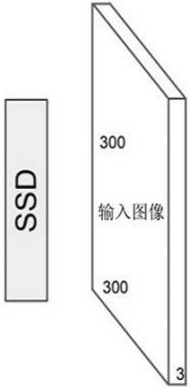
SSD检测算法：输入图像

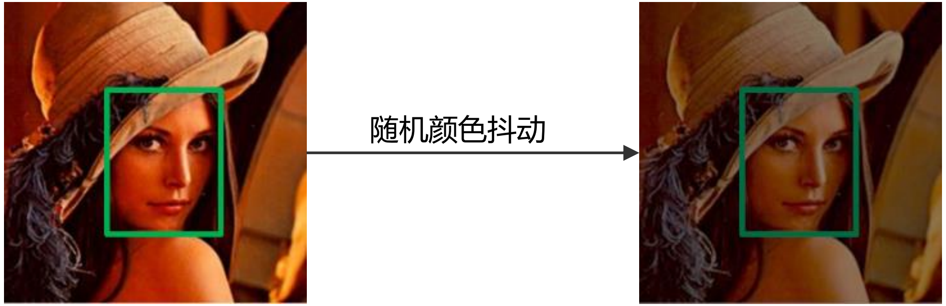


SSD检测算法：输入图像

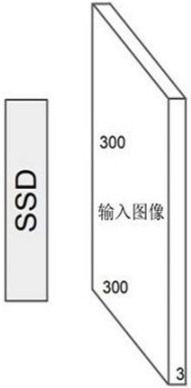


SSD检测算法：输入图像

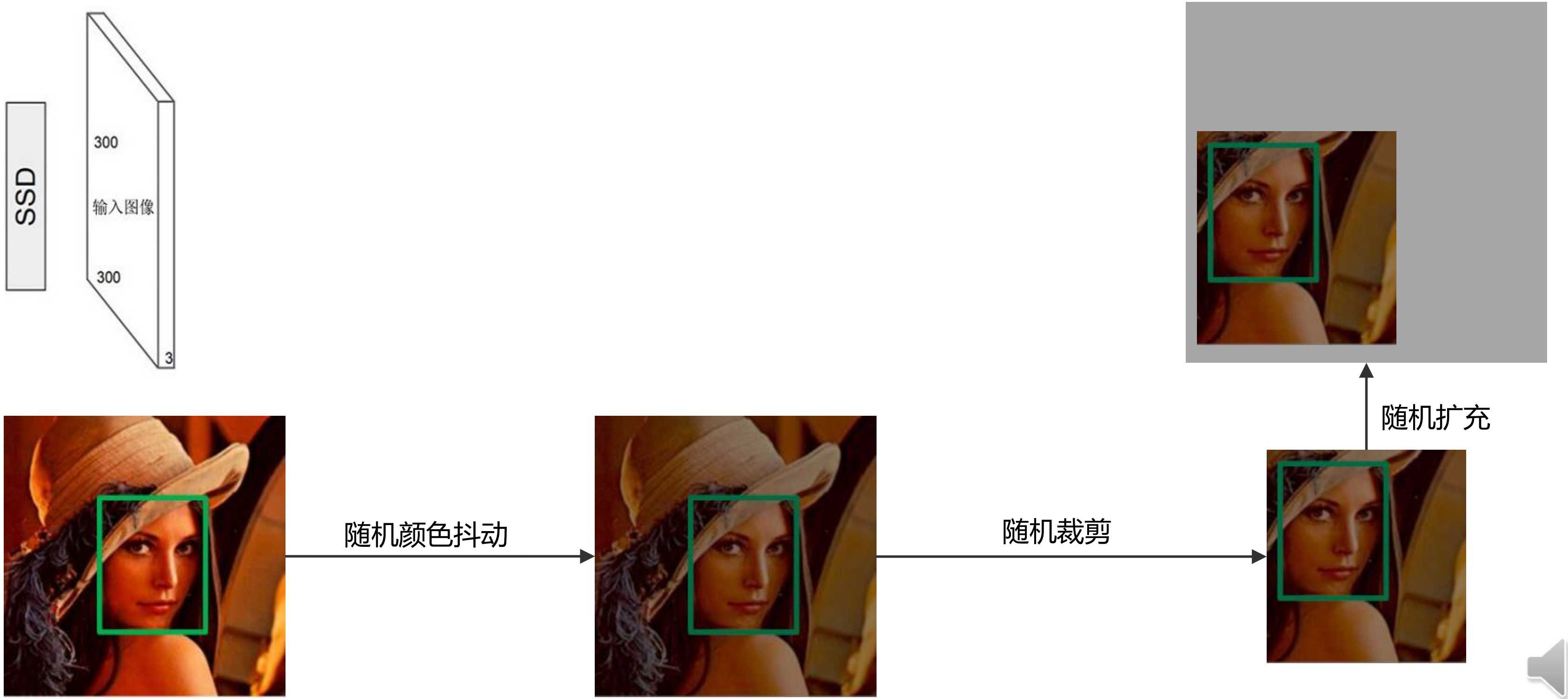




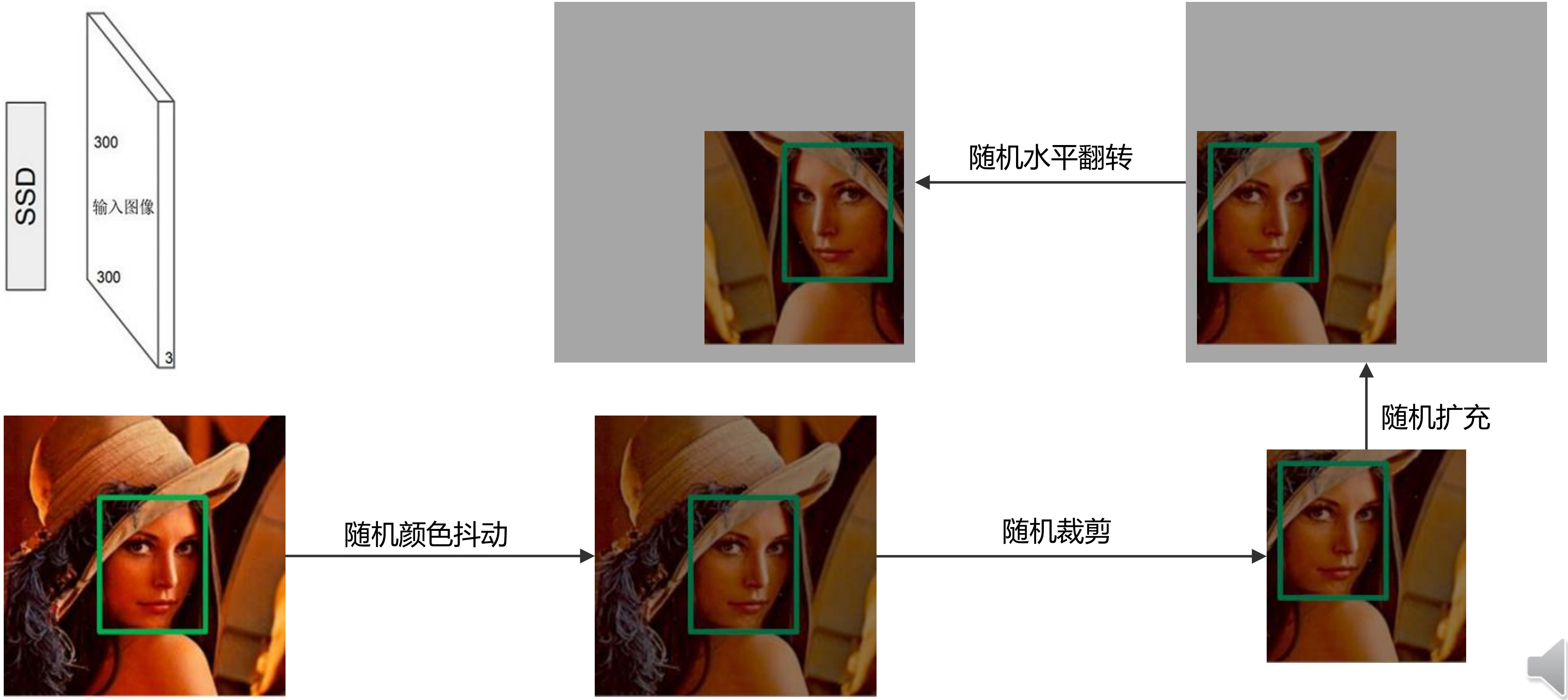
SSD检测算法：输入图像



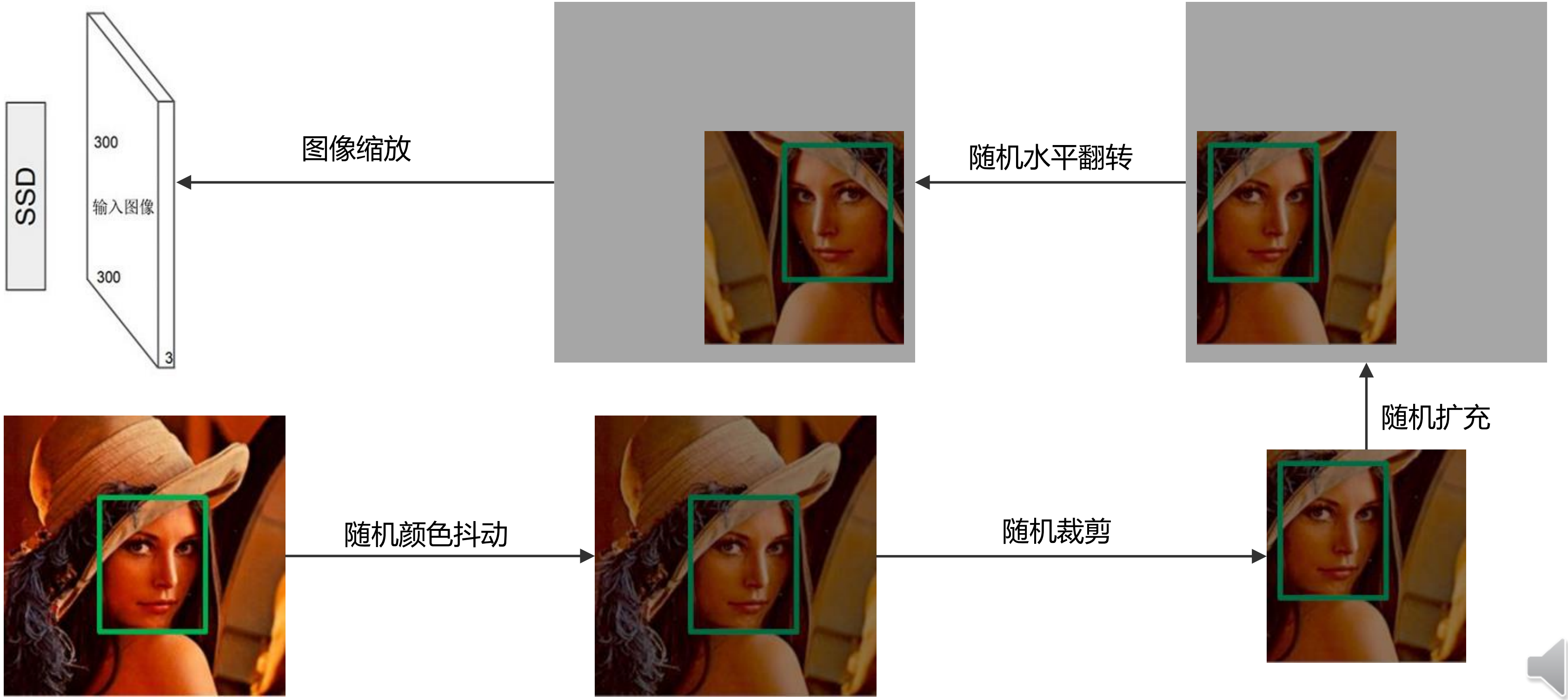
SSD检测算法：输入图像



SSD检测算法：输入图像

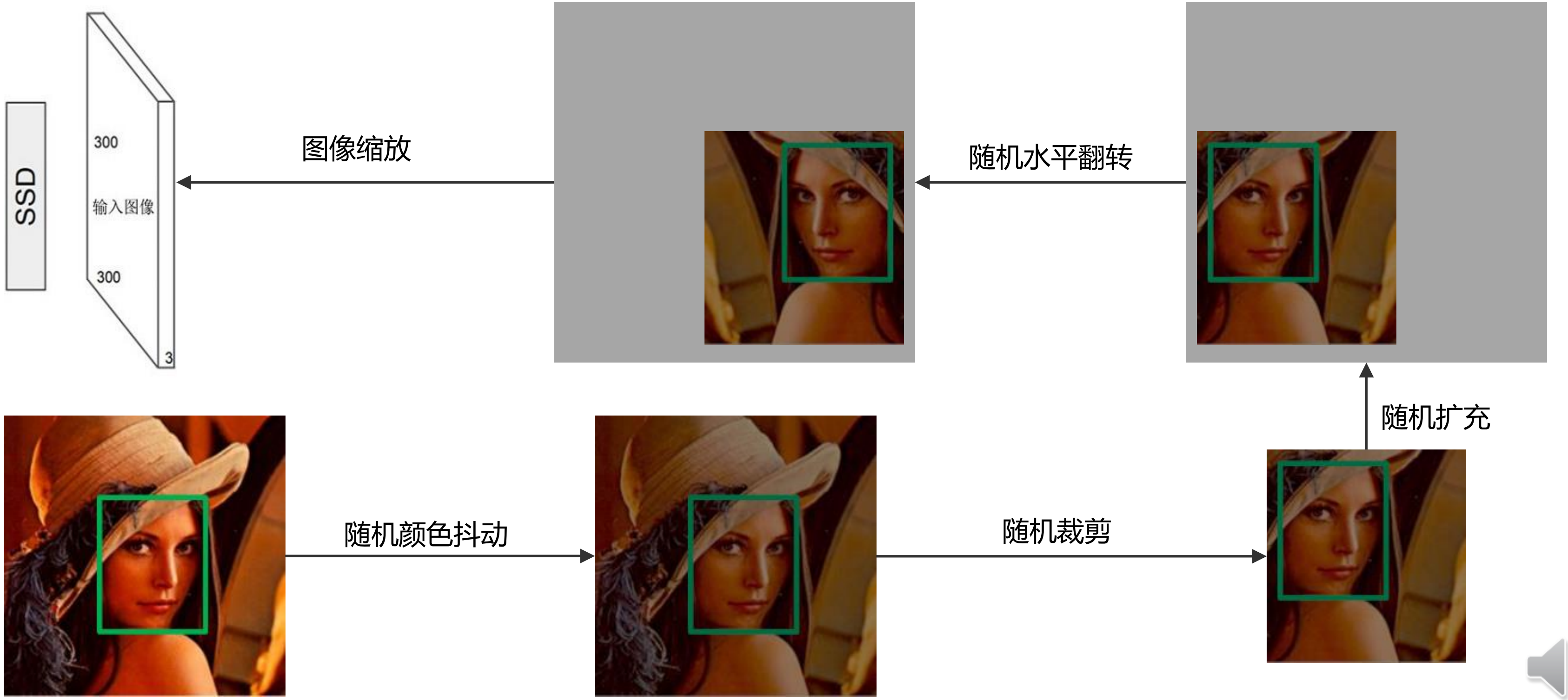


SSD检测算法：输入图像

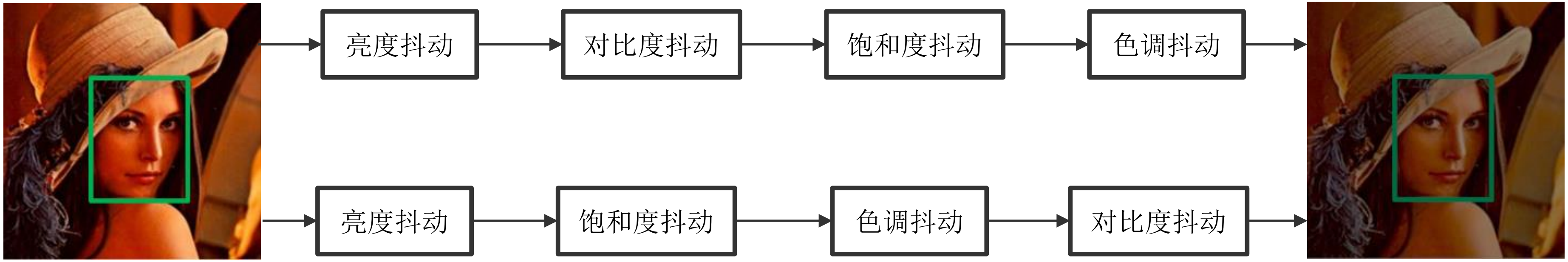


SSD检测算法：输入图像

测试阶段，只对输入图像进行图像缩放操作，其他数据增广操作不执行



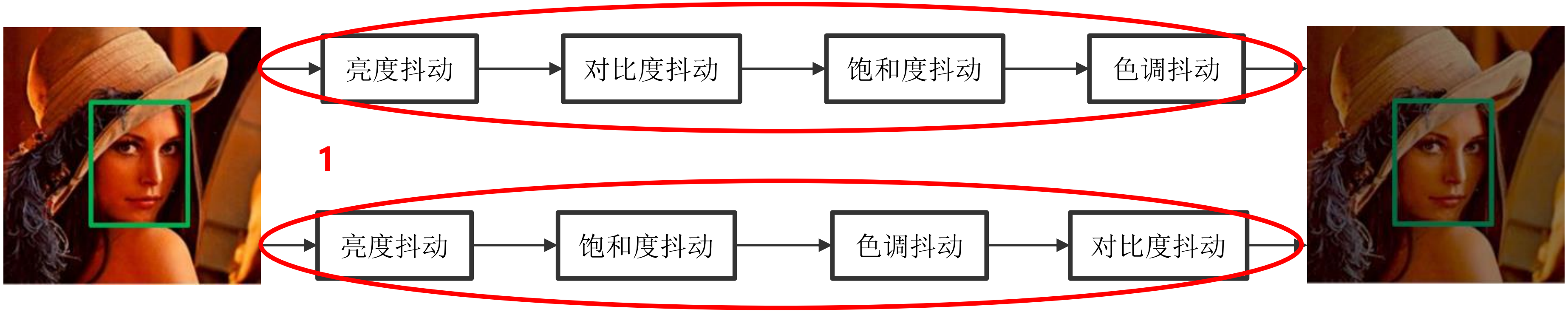
SSD检测算法：输入图像的随机颜色抖动



- **亮度抖动**：RGB图像 + random.uniform(-32, 32)
- **对比度抖动**：RGB图像 * random.uniform(0.5, 1.5)
- **饱和度抖动**：HSV图像的S通道 * random.uniform(0.5, 1.5)
- **色调抖动**：HSV图像的H通道 + random.randint(-18, 18)

注：HSV颜色空间, H->色调, S->饱和度, V->明度

SSD检测算法：输入图像的随机颜色抖动



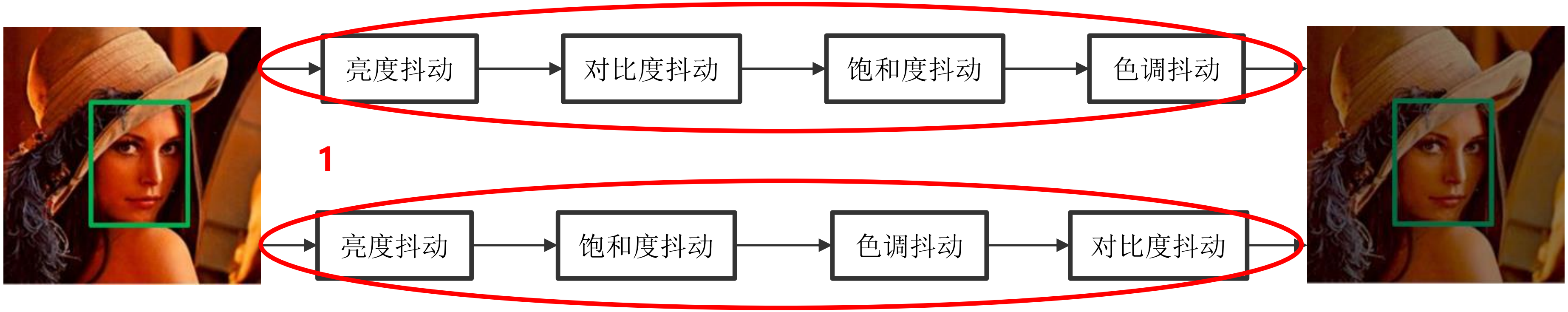
- **亮度抖动**：RGB图像 + random.uniform(-32, 32)
- **对比度抖动**：RGB图像 * random.uniform(0.5, 1.5)
- **饱和度抖动**：HSV图像的S通道 * random.uniform(0.5, 1.5)
- **色调抖动**：HSV图像的H通道 + random.randint(-18, 18)

- 随机的3层含义

1. 从设计的两条固定路线中，以1/2的概率随机选一条

注：HSV颜色空间, H->色调, S->饱和度, V->明度

SSD检测算法：输入图像的随机颜色抖动



2

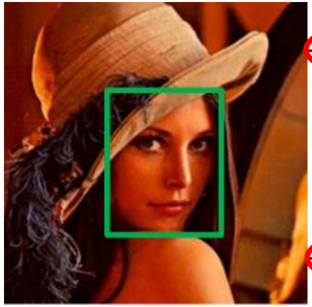
- **亮度抖动**：RGB图像 + random.uniform(-32, 32)
- **对比度抖动**：RGB图像 * random.uniform(0.5, 1.5)
- **饱和度抖动**：HSV图像的S通道 * random.uniform(0.5, 1.5)
- **色调抖动**：HSV图像的H通道 + random.randint(-18, 18)

- 随机的3层含义

1. 从设计的两条固定路线中，以1/2的概率随机选一条
2. 每个抖动操作以1/2的概率执行

注：HSV颜色空间, H->色调, S->饱和度, V->明度

SSD检测算法：输入图像的随机颜色抖动



亮度抖动 → 对比度抖动 → 饱和度抖动 → 色调抖动

1

亮度抖动 → 饱和度抖动 → 色调抖动 → 对比度抖动

2

3

- **亮度抖动**：RGB图像 + random.uniform(-32, 32)
- **对比度抖动**：RGB图像 * random.uniform(0.5, 1.5)
- **饱和度抖动**：HSV图像的S通道 * random.uniform(0.5, 1.5)
- **色调抖动**：HSV图像的H通道 + random.randint(-18, 18)

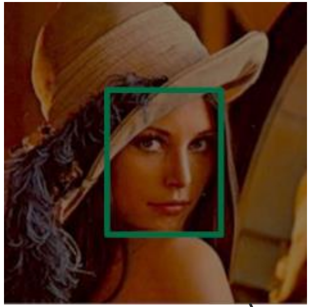
- 随机的3层含义

1. 从设计的两条固定路线中，以1/2的概率随机选一条
2. 每个抖动操作以1/2的概率执行
3. 每个抖动操作中的参数都是随机生成的

注：HSV颜色空间, H->色调, S->饱和度, V->明度

SSD检测算法：输入图像的随机裁剪

SSD检测算法：输入图像的随机裁剪

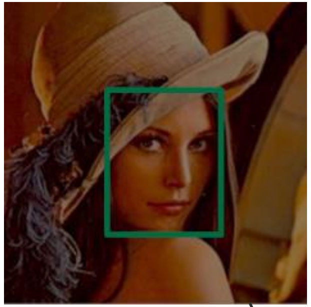


随机在0.3~1.0

之间生成一个数

0.7

SSD检测算法：输入图像的随机裁剪



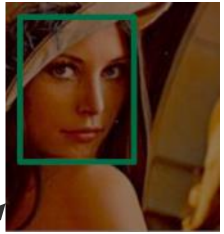
随机在0.3~1.0

之间生成一个数

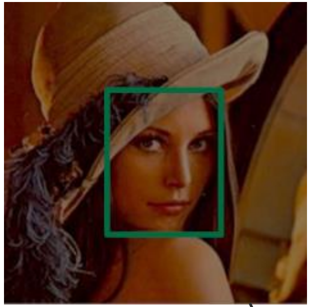
0.7

随机裁剪出0.7倍

大小的图像



SSD检测算法：输入图像的随机裁剪



随机在0.3~1.0

之间生成一个数

0.7

随机裁剪出0.7倍

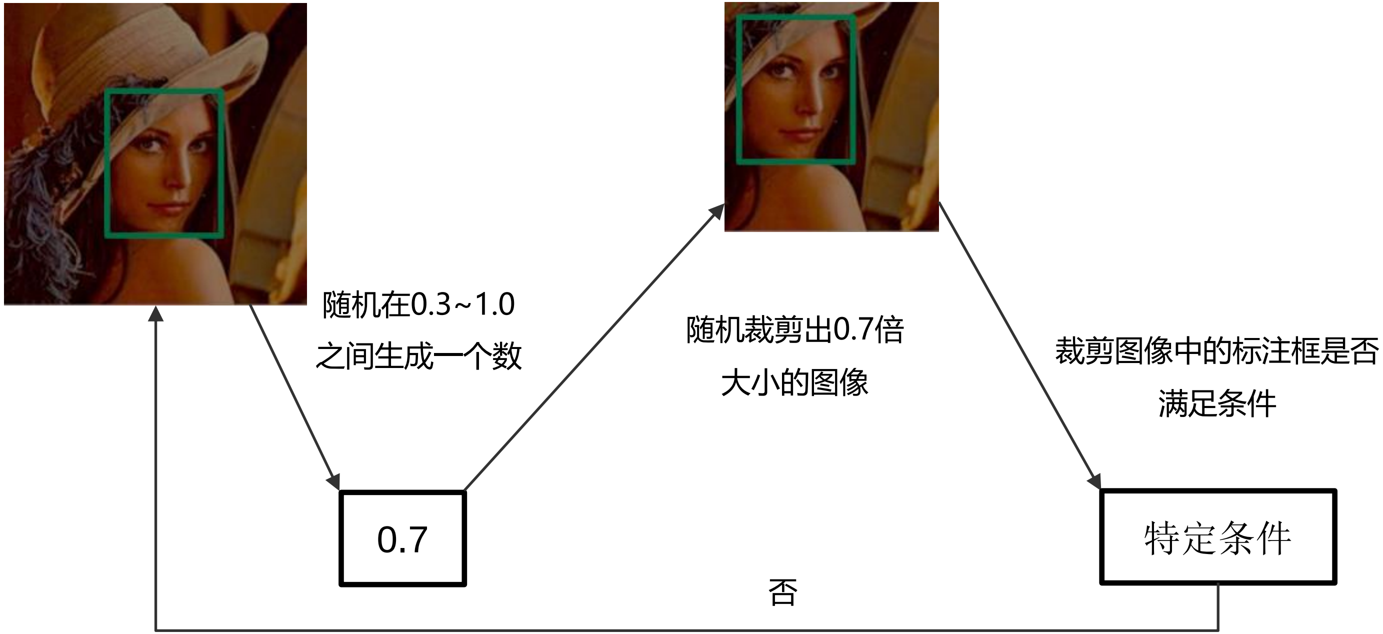
大小的图像

裁剪图像中的标注框是否

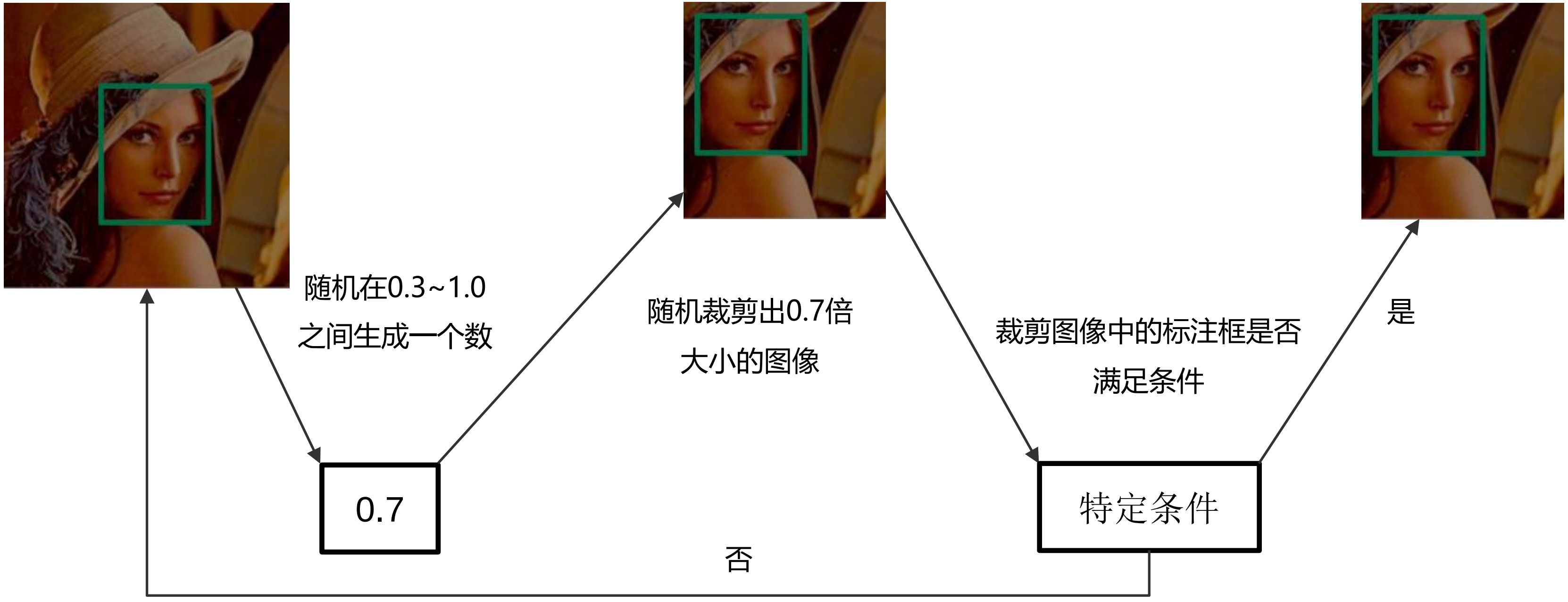
满足条件

特定条件

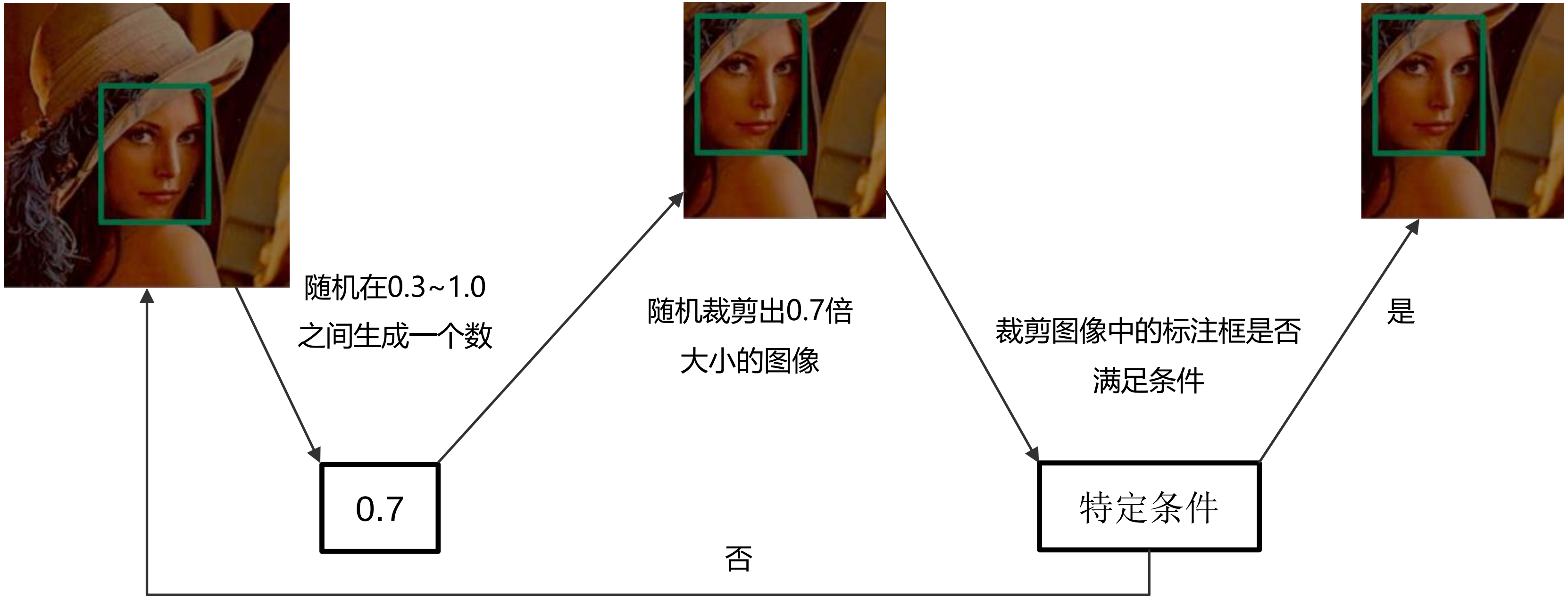
SSD检测算法：输入图像的随机裁剪



SSD检测算法：输入图像的随机裁剪

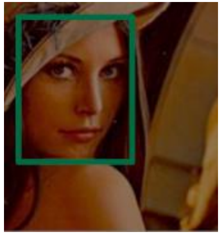


SSD检测算法：输入图像的随机裁剪



- 随机裁剪的数据增广操作以1/2的概率执行

SSD检测算法：输入图像的随机扩充



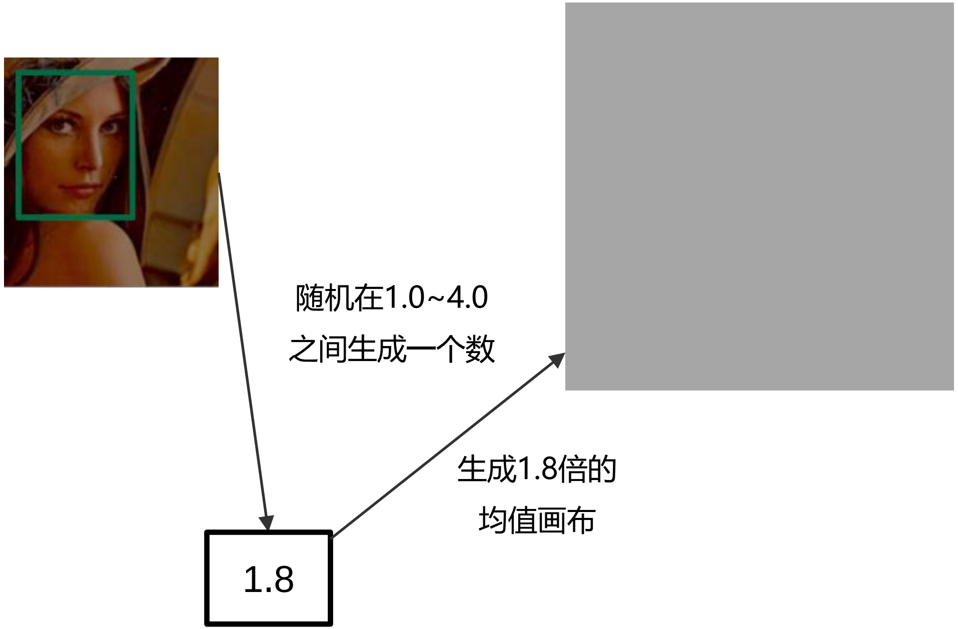
SSD检测算法：输入图像的随机扩充

随机在1.0~4.0

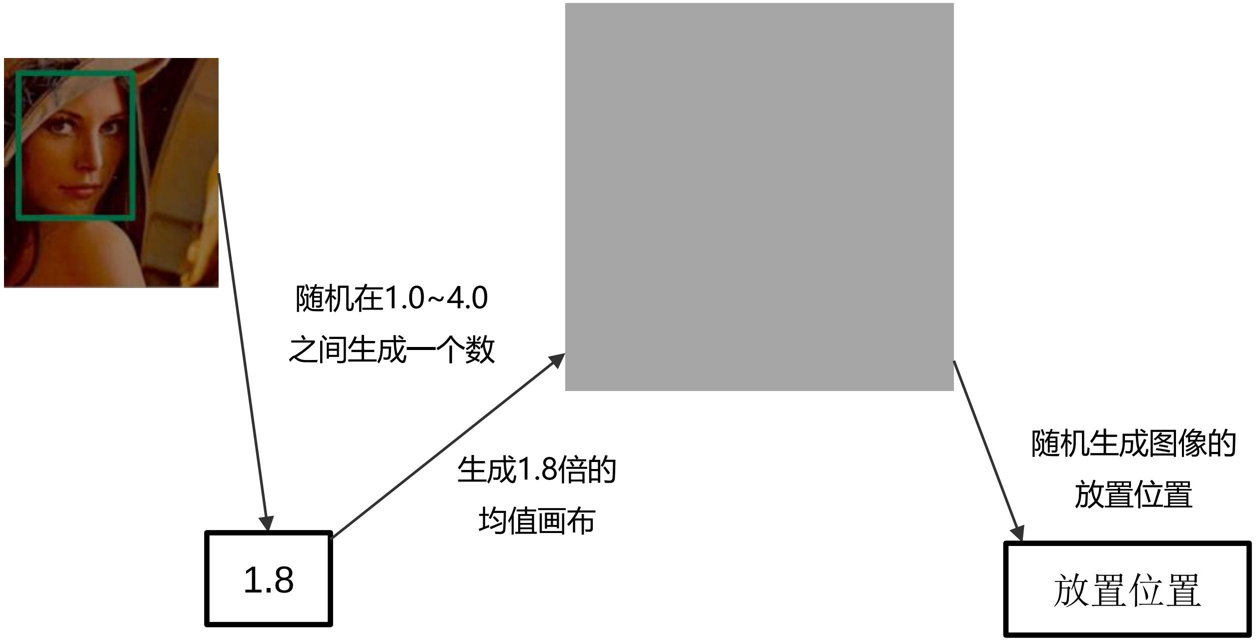
之间生成一个数

$\boxed{1.8}$

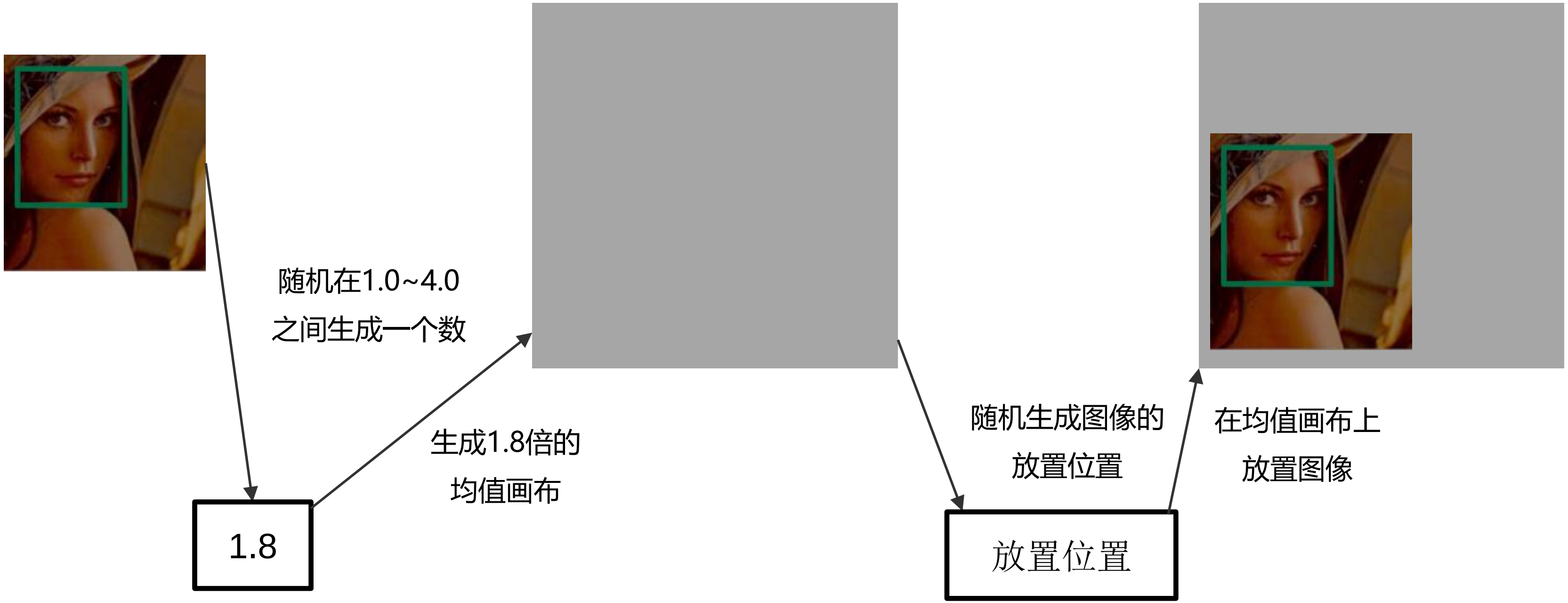
SSD检测算法：输入图像的随机扩充



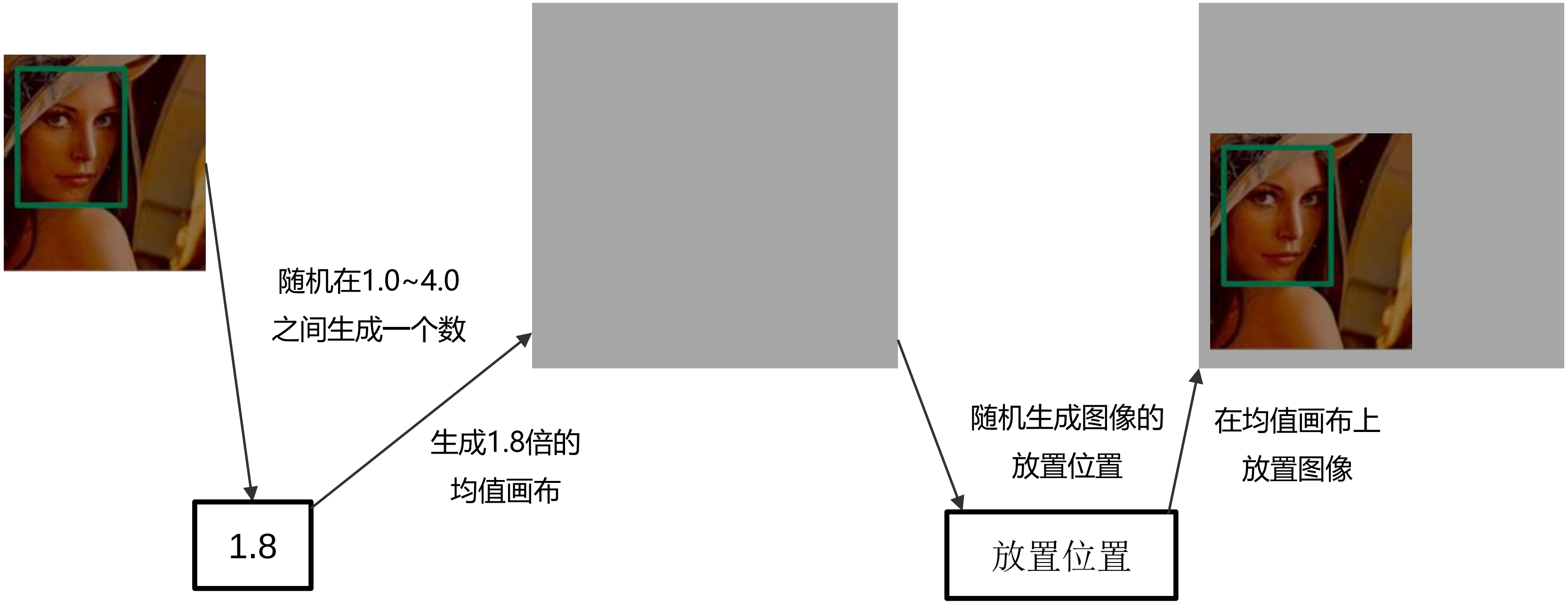
SSD检测算法：输入图像的随机扩充



SSD检测算法：输入图像的随机扩充



SSD检测算法：输入图像的随机扩充

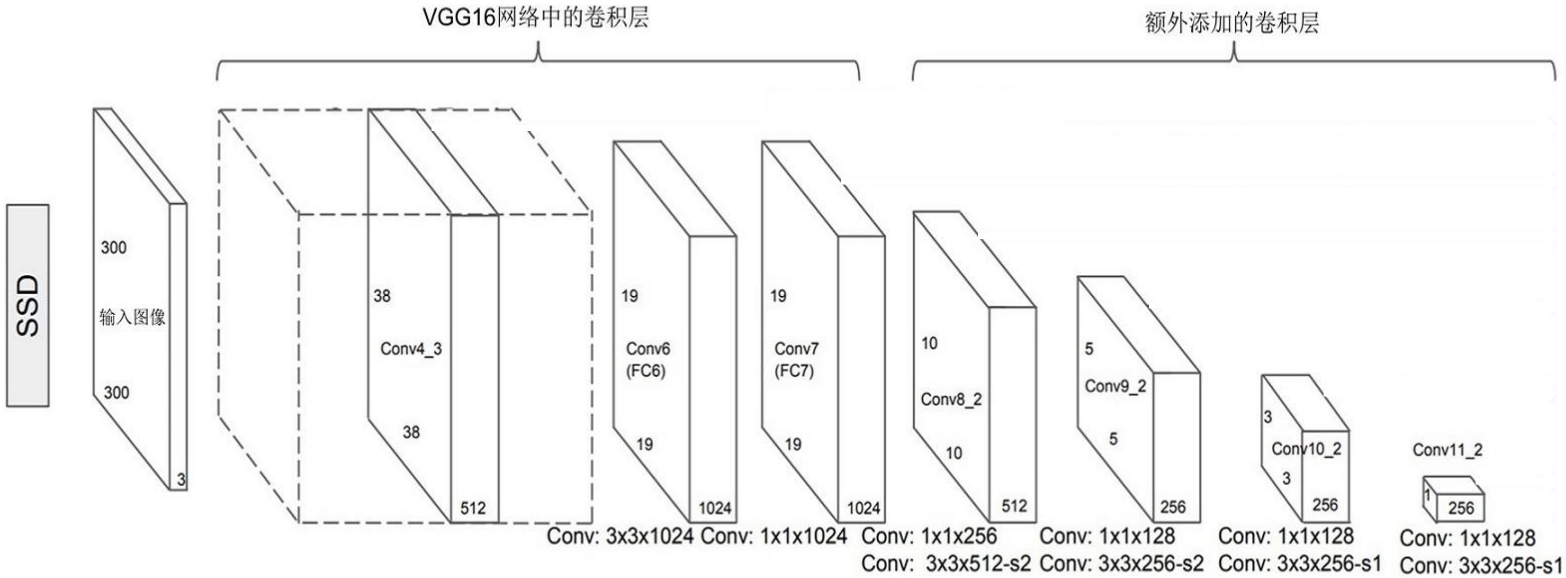


- 随机扩充的数据增广操作以1/2的概率执行

SSD检测算法：输入图像的随机水平翻转和缩放

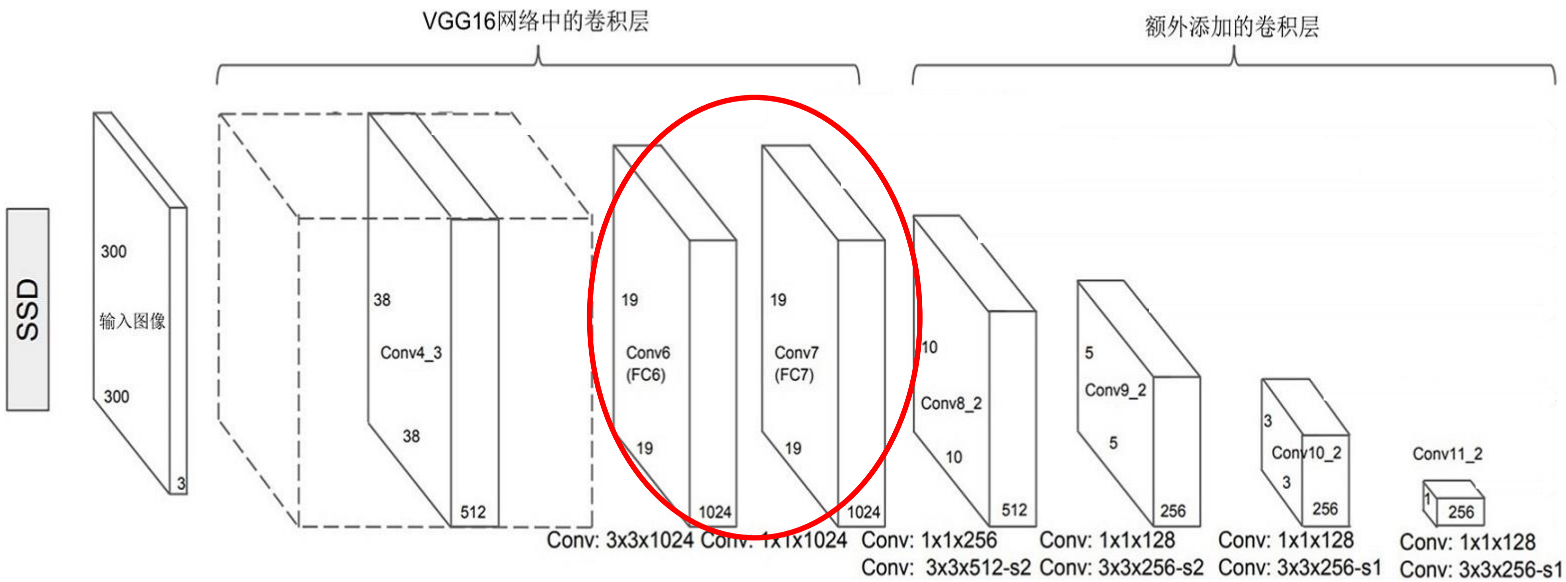
- 随机水平翻转：以1/2的概率对图像进行水平方向的翻转
- 图像缩放：把任意大小的图像缩放到300x300大小（**物体比例会改变**）

SSD检测算法：基础网络



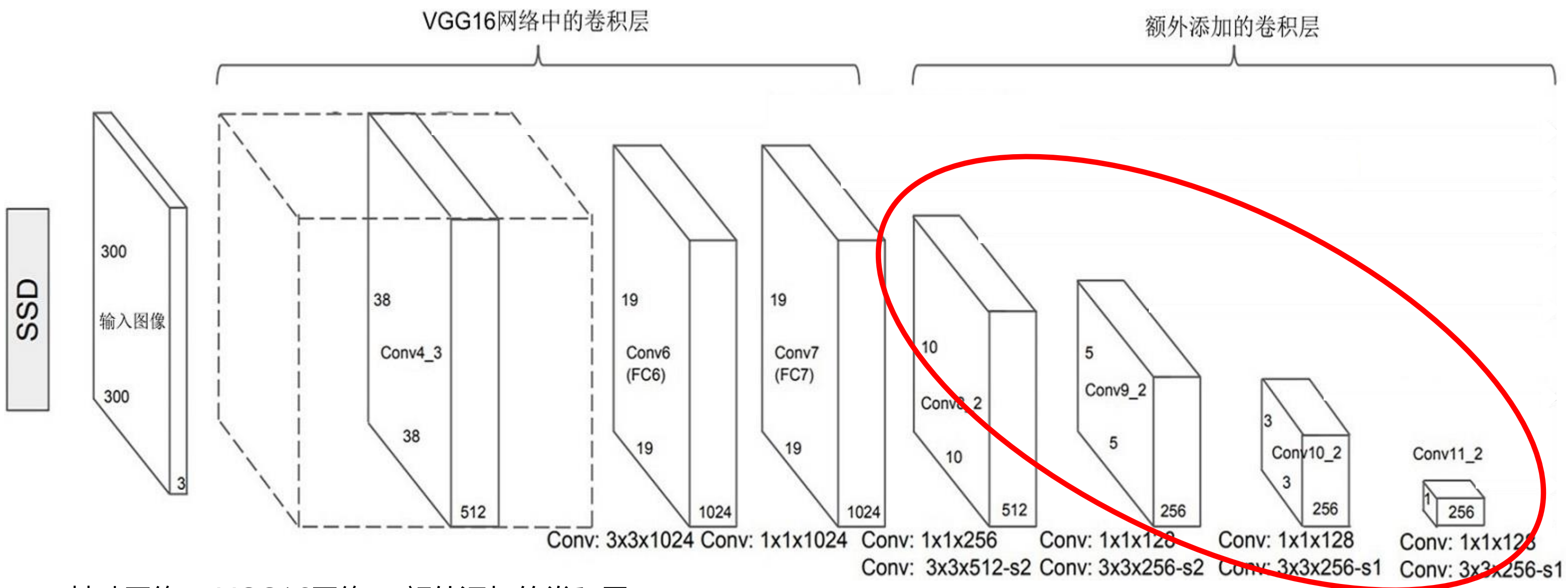
- 基础网络 = VGG16网络 + 额外添加的卷积层

SSD检测算法：基础网络



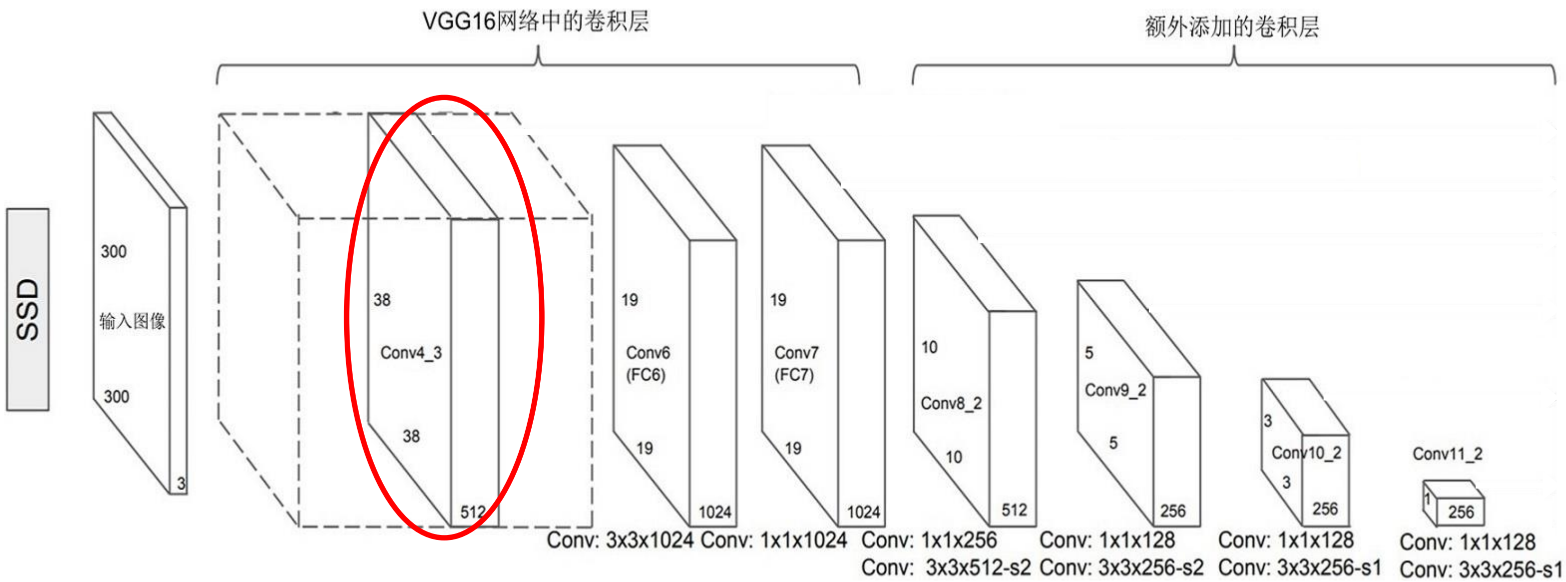
- 基础网络 = VGG16网络 + 额外添加的卷积层
- VGG16中的全连接层FC6和FC7通过权重采样变成卷积层Conv6和Conv7

SSD检测算法：基础网络



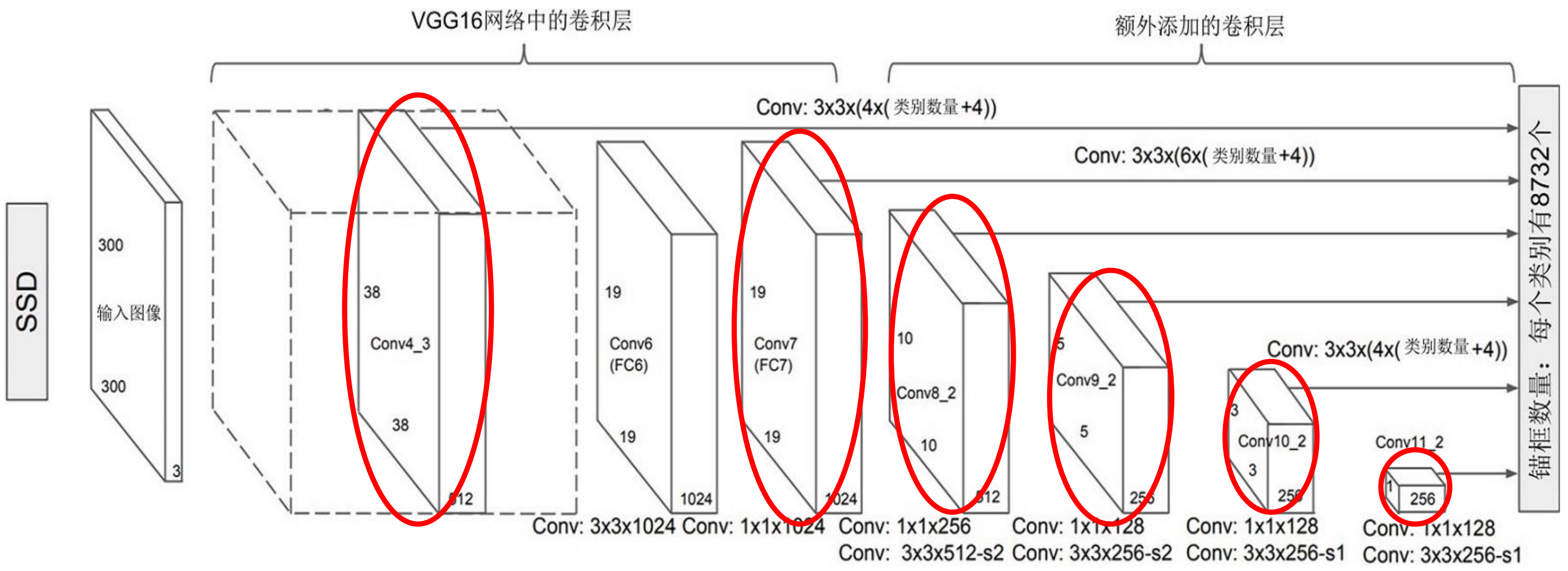
- 基础网络 = VGG16网络 + 额外添加的卷积层
- VGG16中的全连接层FC6和FC7通过权重采样变成卷积层Conv6和Conv7
- 额外添加了8个卷积层，每2个卷积层为1组，有着相同的下采样倍数

SSD检测算法：基础网络



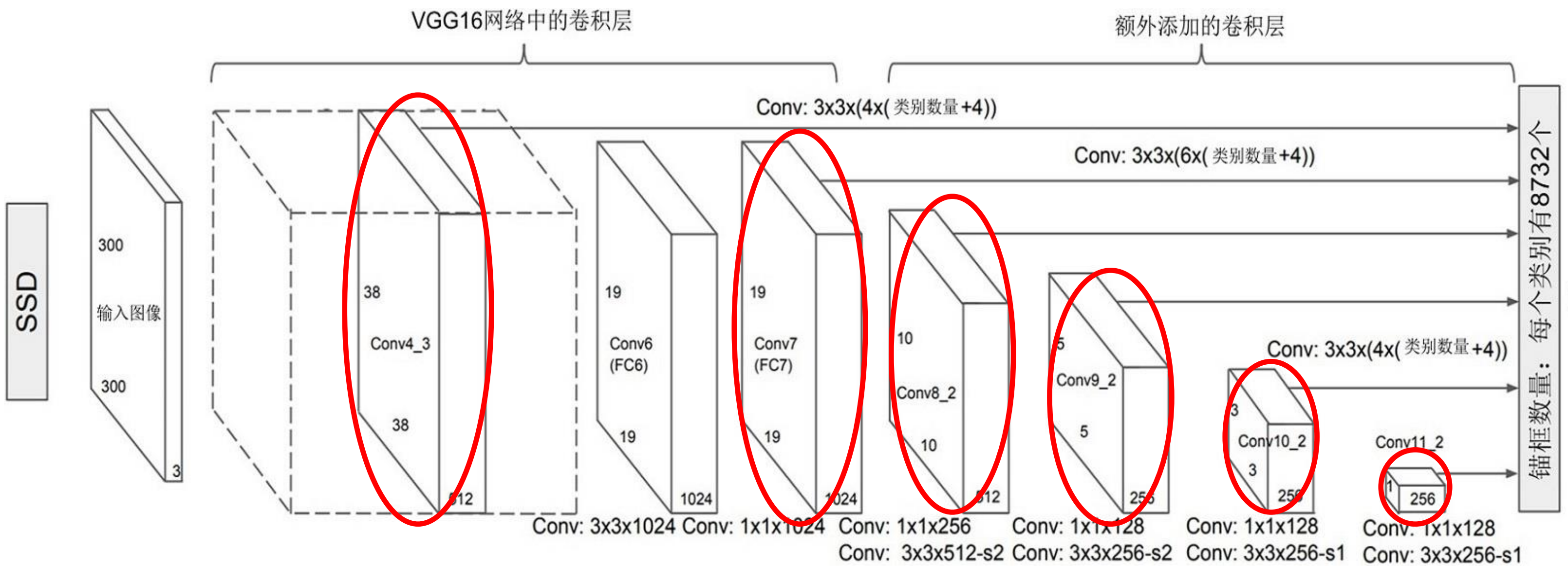
- 基础网络 = VGG16网络 + 额外添加的卷积层
- VGG16中的全连接层FC6和FC7通过权重采样变成卷积层Conv6和Conv7
- 额外添加了8个卷积层，每2个卷积层为1组，有着相同的下采样倍数
- VGG16中的Conv4_3卷积层的幅值太大，使用归一化操作把幅值变成20，并反传学习该参数

SSD检测算法：多检测层



- 6个检测层：Conv4_3、Conv_7、 Conv8_2、Conv9_2、Conv10_2、Conv11_2

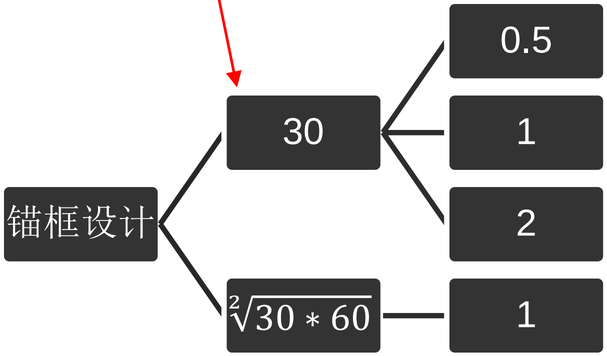
SSD检测算法：多检测层

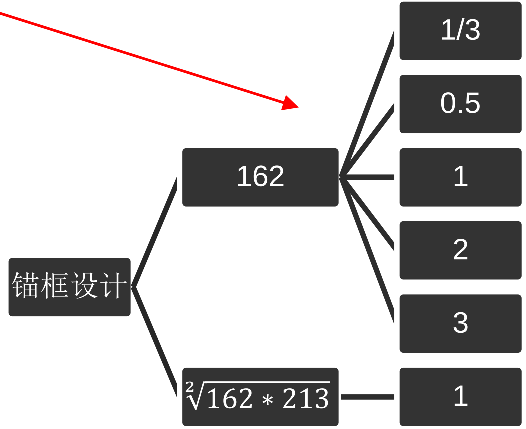


- 6个检测层：Conv4_3、Conv_7、 Conv8_2、Conv9_2、Conv10_2、Conv11_2
- 下采样倍数： 8、 16、 32、 64、 128、 256
- 锚框大小： [30, 60]、[60, 111]、[111, 162]、[162, 213]、[213, 264]、[264, 315]
- 锚框比例： [0.5,1,2]、[$\frac{1}{3}$,0.5,1,2,3]、[$\frac{1}{3}$,0.5,1,2,3]、[$\frac{1}{3}$,0.5,1,2,3]、 [0.5,1,2]、 [0.5,1,2]

SSD检测算法：多检测层的锚框设计

- 6个检测层：Conv4_3、Conv_7、 Conv8_2、Conv9_2、Conv10_2、Conv11_2
- 下采样倍数： 8 16、 32、 64、 128、 256
- 锚框大小： [30, 60]、[60, 111]、[111, 162]、[162, 213]、[213, 264]、[264, 315]
- 锚框比例： [0.5,1,2]、[$\frac{1}{3}$,0.5,1,2,3]、[$\frac{1}{3}$,0.5,1,2,3]、[$\frac{1}{3}$,0.5,1,2,3]、 [0.5,1,2]、 [0.5,1,2]

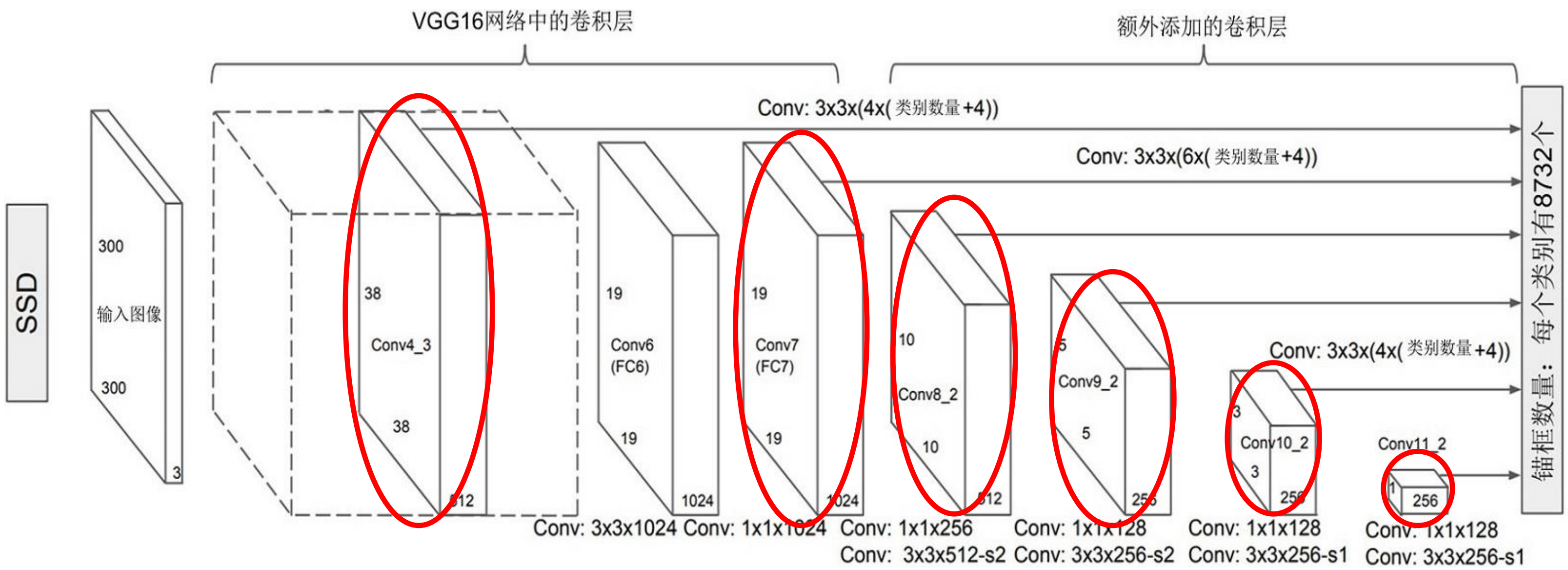




* **SSD中生成锚框的规则：每个检测层的锚框大小都有两个尺度，即[S_{min}, S_{max}]**

S_{min}生成与所有锚框比例结合生成锚框，S_{max}只跟1:1的锚框比例结合生成锚框

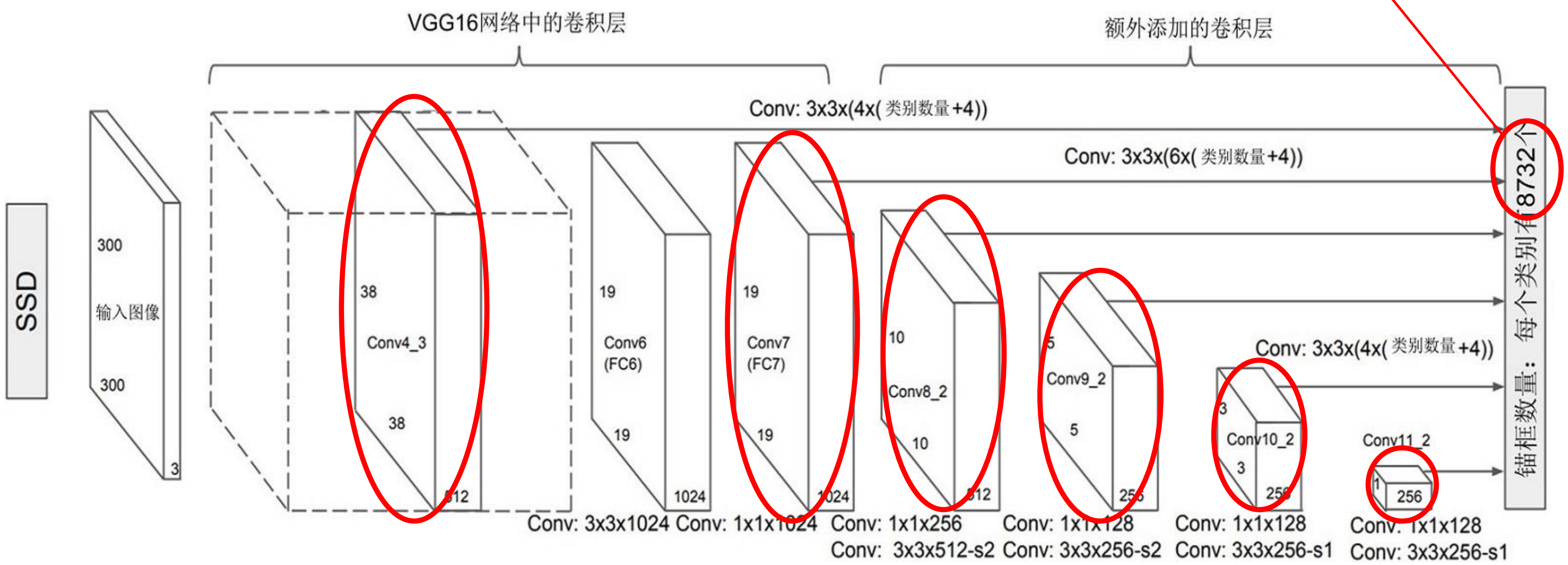
SSD检测算法：多检测层



- 6个检测层：Conv4_3、Conv_7、 Conv8_2、Conv9_2、Conv10_2、Conv11_2
- 下采样倍数： 8、 16、 32、 64、 128、 256
- 锚框大小： [30, 60]、[60, 111]、[111, 162]、[162, 213]、[213, 264]、[264, 315]
- 锚框比例： [0.5,1,2]、$[\frac{1}{3},0.5,1,2,3]$、$[\frac{1}{3},0.5,1,2,3]$、$[\frac{1}{3},0.5,1,2,3]$、 [0.5,1,2]、 [0.5,1,2]
- 锚框个数： 4、 6、 6、 6、 4、 4

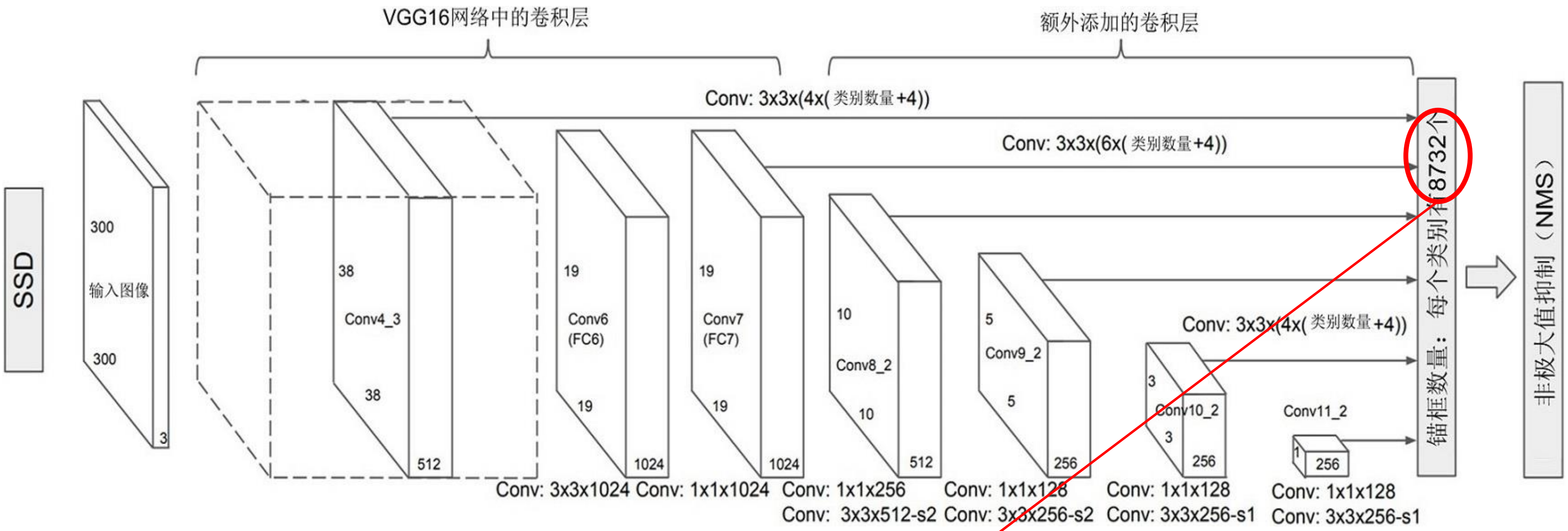
SSD检测算法：多检测层

38x38x4+19x19x6+10x10x6+5x5x6+3x3x4+1x1x4 = 8732



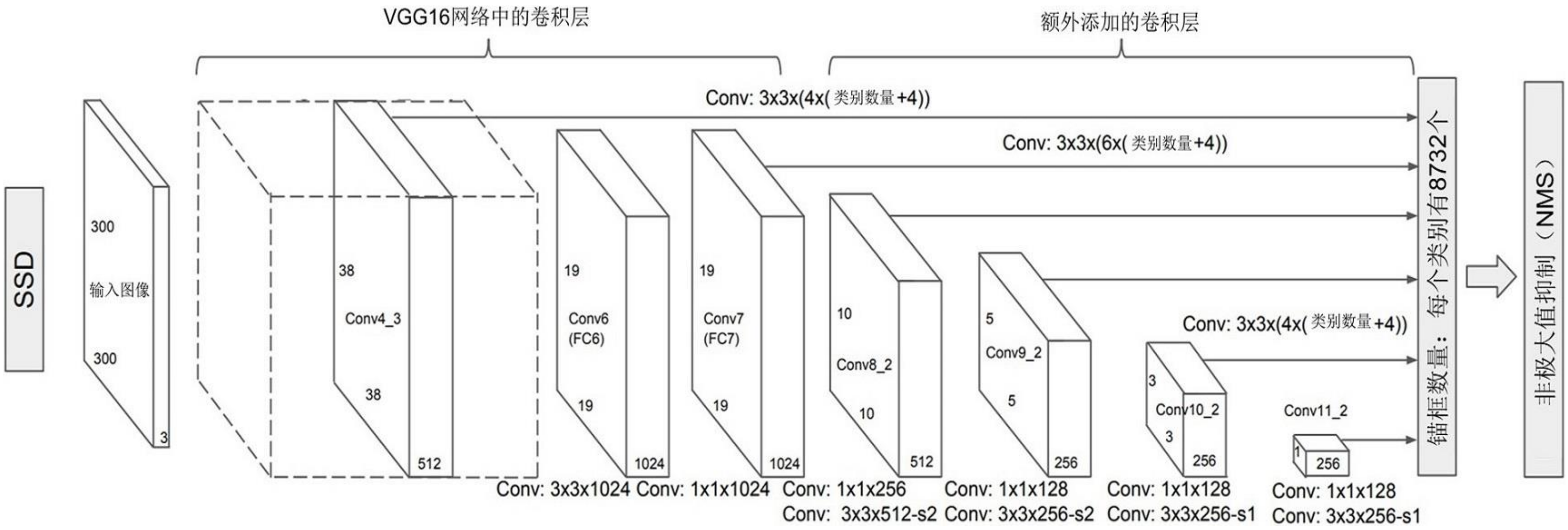
- 6个检测层：Conv4_3、Conv_7、 Conv8_2、Conv9_2、Conv10_2、Conv11_2
- 下采样倍数： 8、 16、 32、 64、 128、 256
- 锚框大小： [30, 60]、[60, 111]、[111, 162]、[162, 213]、[213, 264]、[264, 315]
- 锚框比例： [0.5,1,2]、$[\frac{1}{3}$,0.5,1,2,3]、$[\frac{1}{3}$,0.5,1,2,3]、$[\frac{1}{3}$,0.5,1,2,3]、 [0.5,1,2]、 [0.5,1,2]
- 锚框个数： 4、 6、 6、 6、 4、 4

SSD检测算法：难负样本挖掘

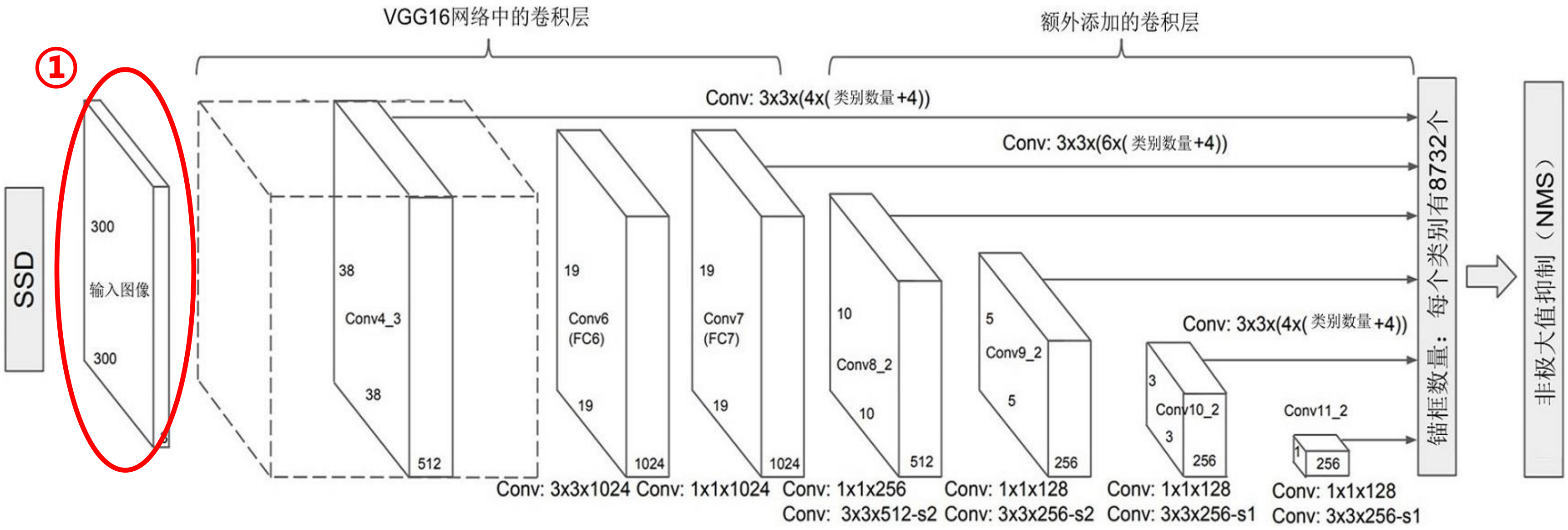


- 锚框匹配：①正样本：最佳匹配或IoU≥0.5；②负样本：IoU < 0.5
- 匹配结果：把锚框划分为正负样本之后，一般只有几十个锚框是正样本，其他都是负样本
- 比例失衡：正样本/负样本 = 约50个正样本/约8700个负样本 ≈ 1/174
- 导致问题：极度不平衡的正负样本比例，让网络训练朝着错误的方向优化
- RPN的解决方案：正样本保持不变，随机选择一小部分负样本出来，其他负样本忽略
- SSD的解决方案：正样本保持不变，根据分类误差损失值对负样本进行降序排序，选出正样本数量的3倍，其他忽略

SSD检测算法总结：主要贡献

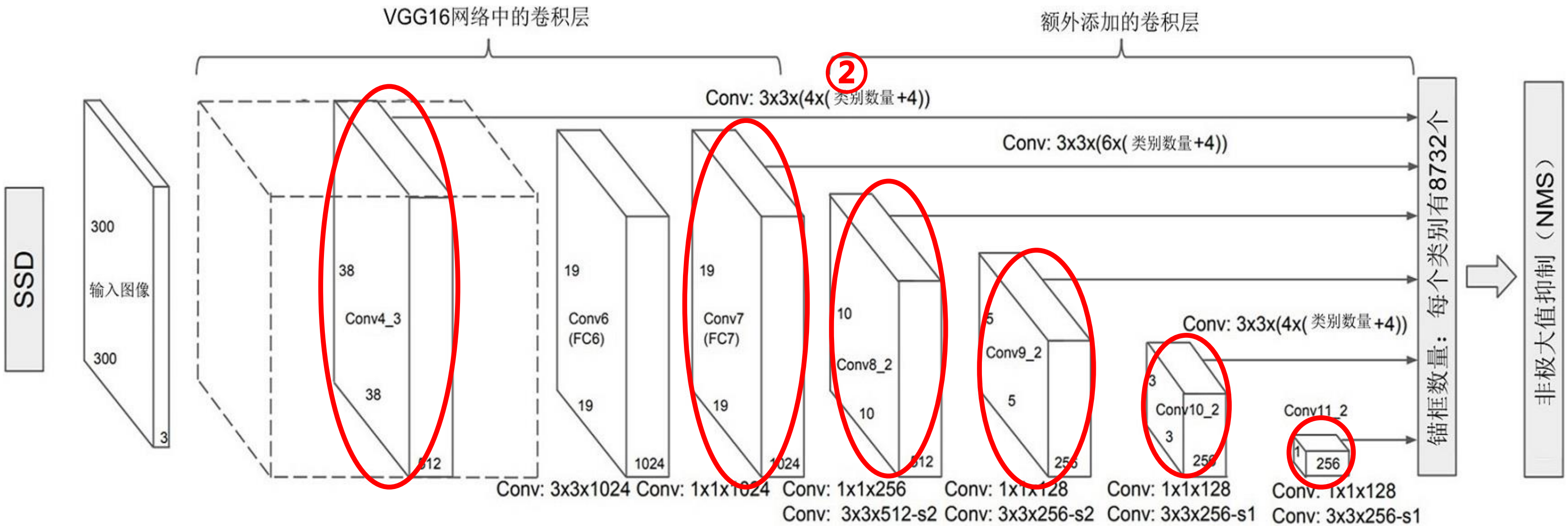


SSD检测算法总结：主要贡献



① 数据增广：RPN只是用了水平翻转和静态多尺度训练，而SSD设计了随机性更强、更丰富的数据增广，例如颜色抖动、随机裁剪+扩充（动态多尺度训练）

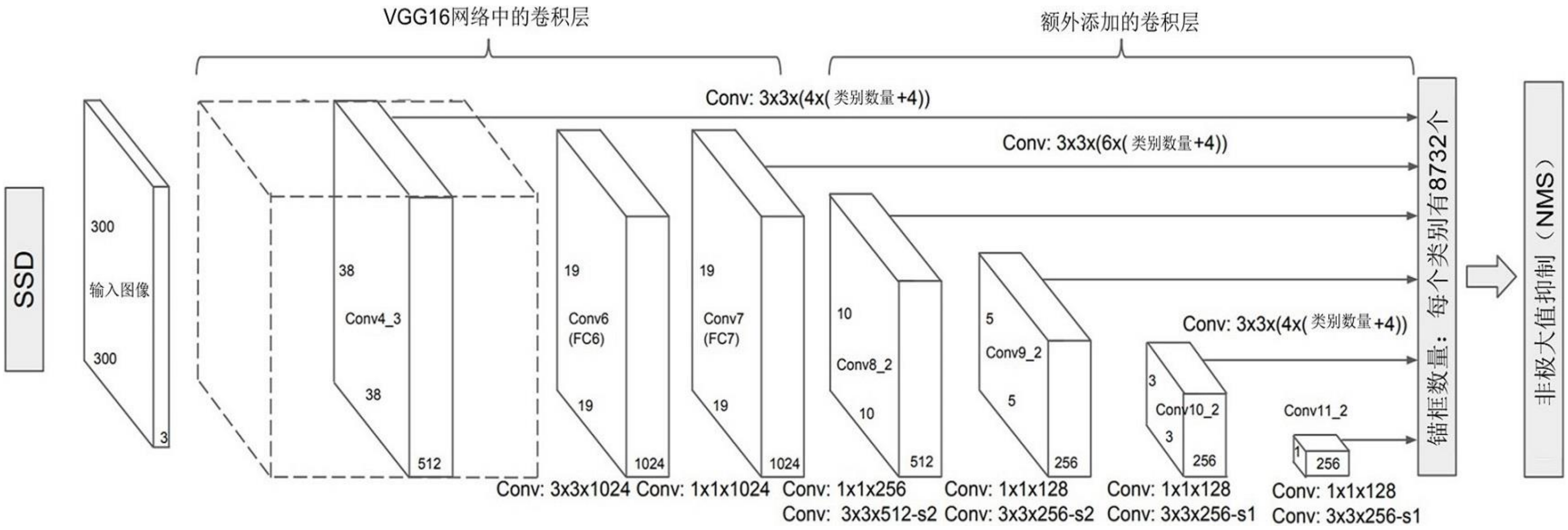
SSD检测算法总结：主要贡献



① 数据增广：RPN只是用了水平翻转和静态多尺度训练，而SSD设计了随机性更强、更丰富的数据增广，例如颜色抖动、随机裁剪+扩充（动态多尺度训练）

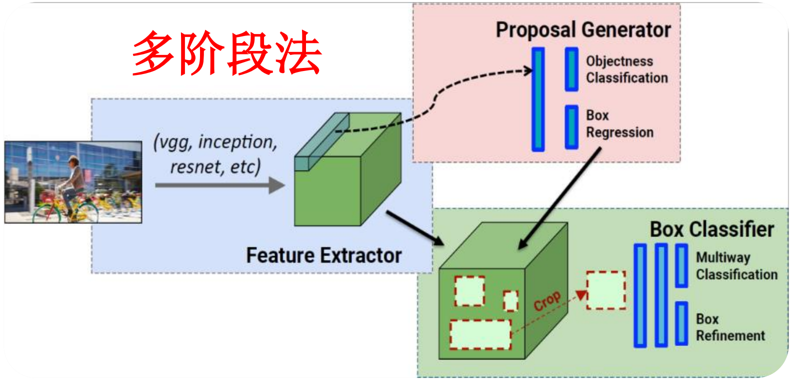
② 多检测层：RPN是在单个检测层上关联锚框进行检测，而SSD在多个检测层上关联适当的锚框进行检测，后续算法基本都采用多检测层这种设计

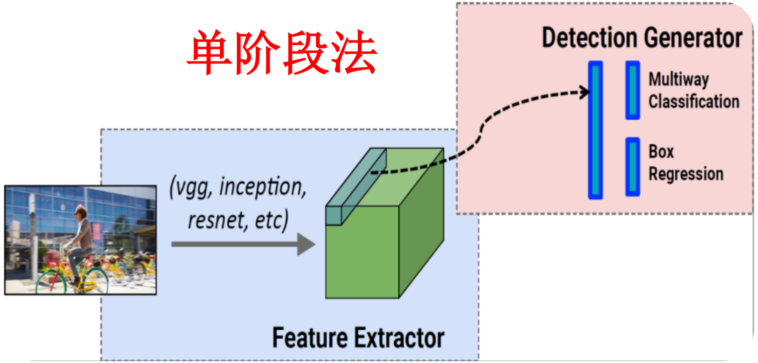
SSD检测算法总结：集大成者



- 单阶段检测算法的集大成者，保持几十FPS的速度，精度跟多阶段检测算法差不多
- 所有代码都是基于Caffe用C++实现，方便工程部署，很多实际产品在使用
- 后续的单阶段检测算法大多都是基于SSD进行改进的

基于锚框的单阶段检测算法：RetinaNet





两类检测算法的对比

- 多阶段法：高精度，但速度较慢
- 单阶段法：速度快，但是准确率不如前者
- 单阶段法精度差的主要原因之一：**正负样本数量的极度不均衡**

Focal Loss for Dense Object Detection, **Best Student Paper Award**, ICCV 2017

基于锚框的单阶段检测算法：RetinaNet

- SSD使用**难负样本挖掘**来解决正负样本比例极度不平衡的问题，而难负样本挖掘有两个问题：

① 样本利用不充分：只使用了一小部分较难的负样本，大部分负样本都没有使用

② 挖掘难以控制：利用正样本数量的3倍来挖负样本，有时挖太多，有时挖太少

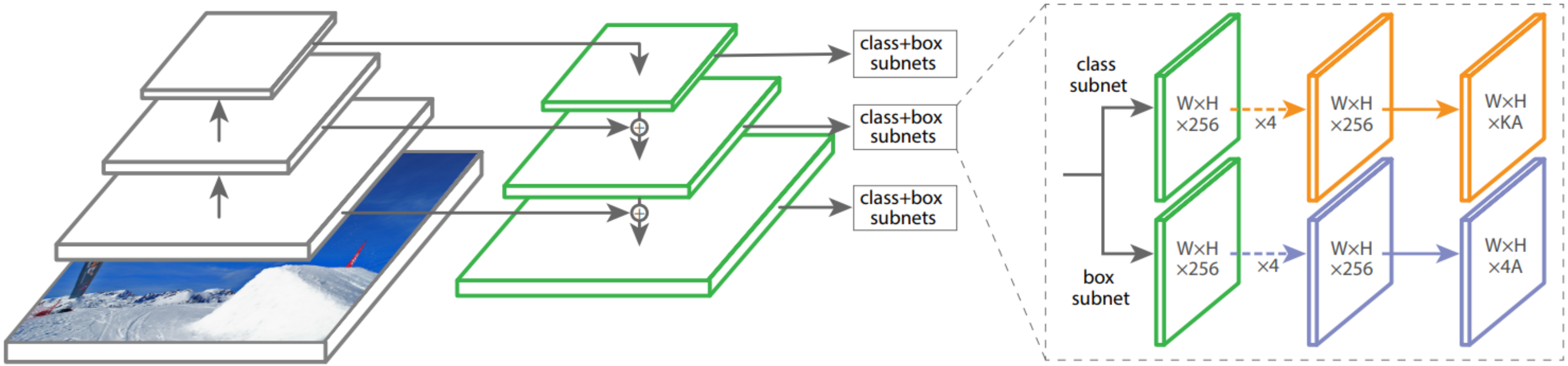
- RetinaNet通过修改标准交叉熵损失，提出了**focal loss损失函数**：

① 通过减少易分类样本的权重，使得模型在训练时更专注于难分类的样本，从而充分地利用所有样本

② 使得单阶段法检测器可以达到多阶段法检测器准确率，同时不影响原有的速度

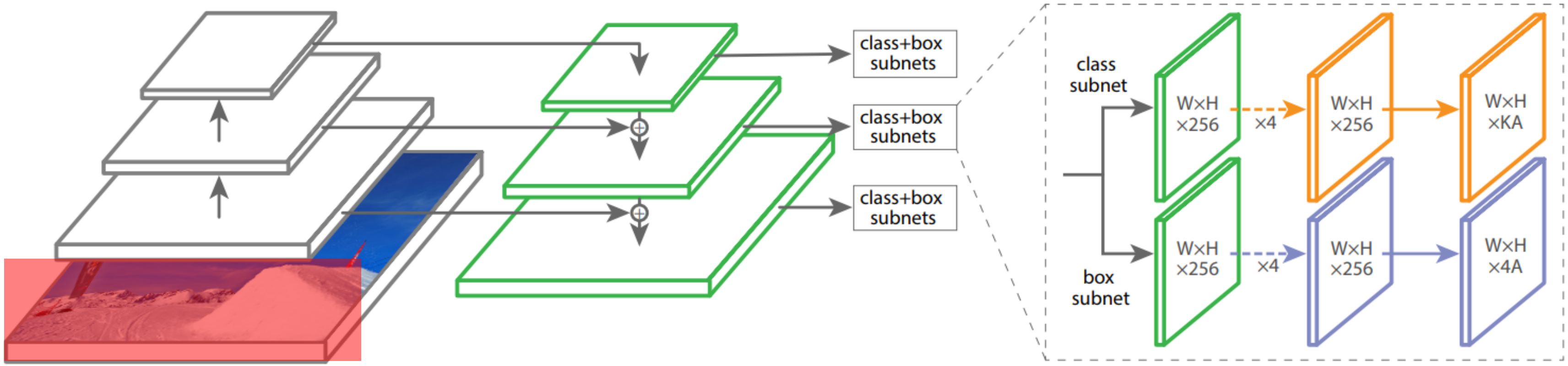
Focal Loss for Dense Object Detection, **Best Student Paper Award**, ICCV 2017

RetinaNet检测算法：整体框架



(a) ResNet (b) feature pyramid net (c) class subnet (top) (d) box subnet (bottom)

RetinaNet检测算法：输入图像



(a) ResNet (b) feature pyramid net (c) class subnet (top) (d) box subnet (bottom)

■ RetinaNet对输入图像的操作：

① 随机水平翻转

② 单尺度训练：图像短边等比例缩放至800，且长边不超过1333

③ 多尺度训练：图像短边等比例缩放至[640, 672, 704, 736, 768, 800]，且长边不超过1333

■ SSD对输入图像的操作：

① 颜色抖动

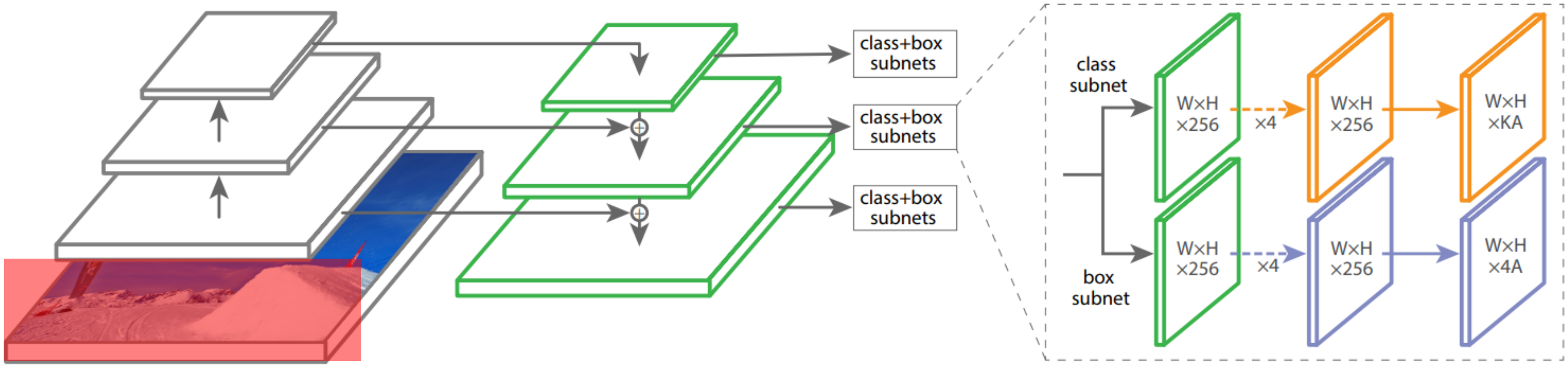
② 随机裁剪

③ 随机扩充

④ 随机水平翻转

⑤ 不等比例地缩放至300x300或512x512

RetinaNet检测算法：输入图像



■ RetinaNet对输入图像的操作 VS ■ SSD对输入图像的操作：

① 输入大小：RetinaNet是~800x1333，而SSD是300x300或512x512

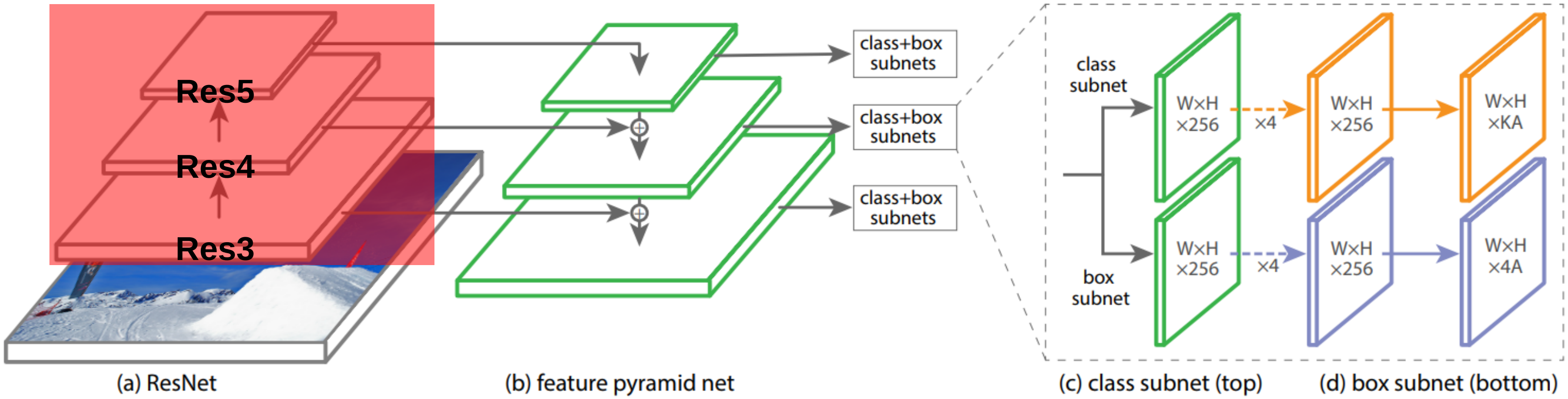
② 物体大小：RetinaNet输入大，物体整体较大，检测难度低，而SSD则相反

③ 多尺度方式：RetinaNet是静态多尺度 (6选1)，而SSD是动态多尺度 (随机扩充+裁剪)

④ 颜色抖动： RetinaNet没有使用，而SSD使用了，在某些任务某些数据上有一定效果

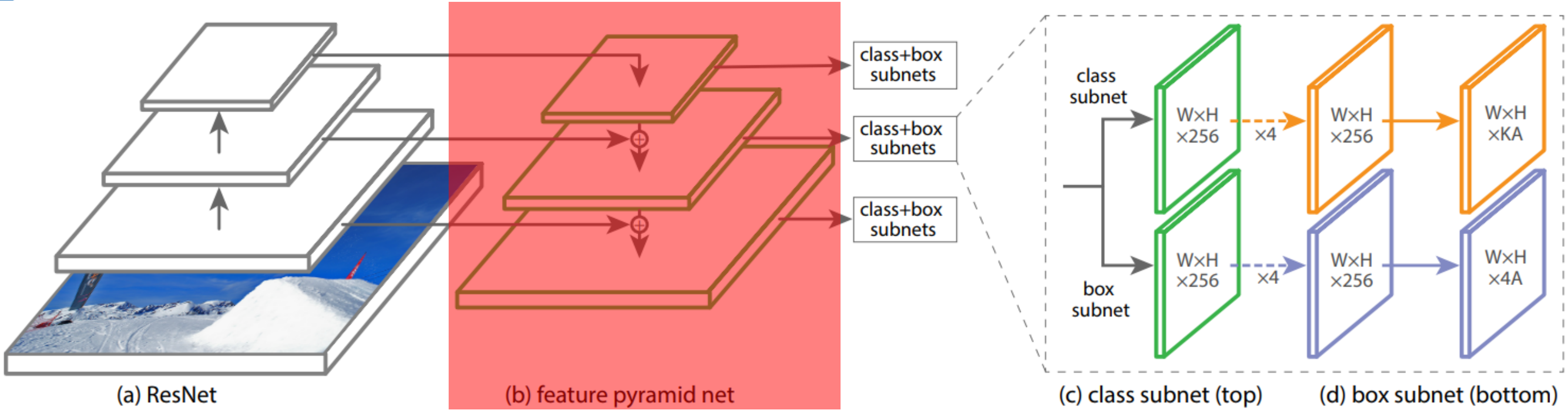
⑤ 迭代次数：数据增广越多，迭代次数就要越多，一般是数据增广+1，迭代次数x2

RetinaNet检测算法：基础网络



- ResNet50/101
- 所有的ResNet网络，都有5个模块组成
- 5个模块：Res1、 Res2、 Res3、 Res4、 Res5
- 下采样率：$2^1=2$、$2^2=4$、 $2^3=8$、 $2^4=16$、$2^5=32$
- RetinaNet选取3个模块来作为初始的检测层
- Res3、 Res4、 Res5

RetinaNet检测算法：特征金字塔



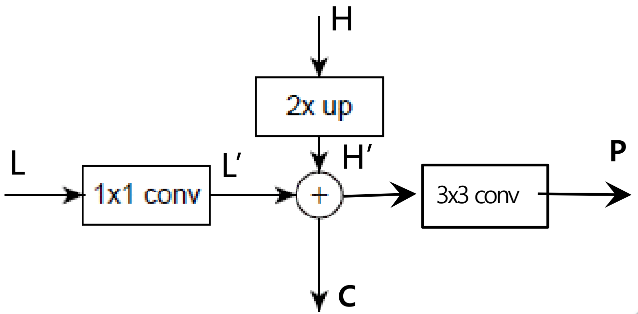
- 利用特征金字塔强化检测层的特征：

① 低层特征L经过1x1卷积，得到L’

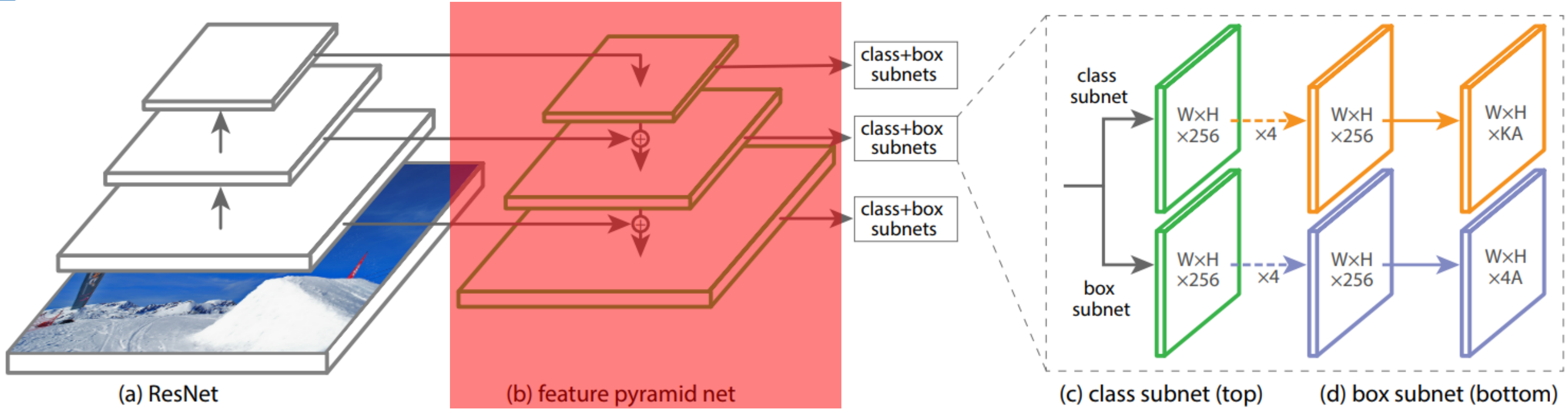
② 高层特征H经过上采样，得到H’

③ 特征融合C = L’ + H’

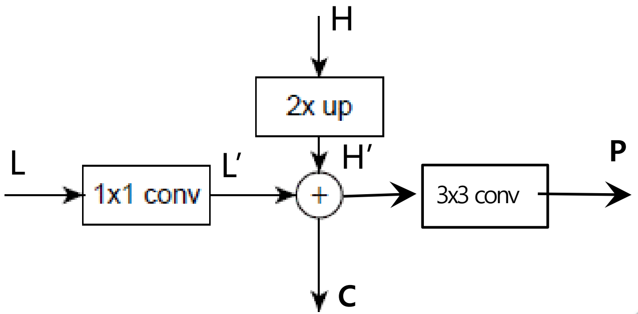
④ 融合特征经过3x3卷积，得到P



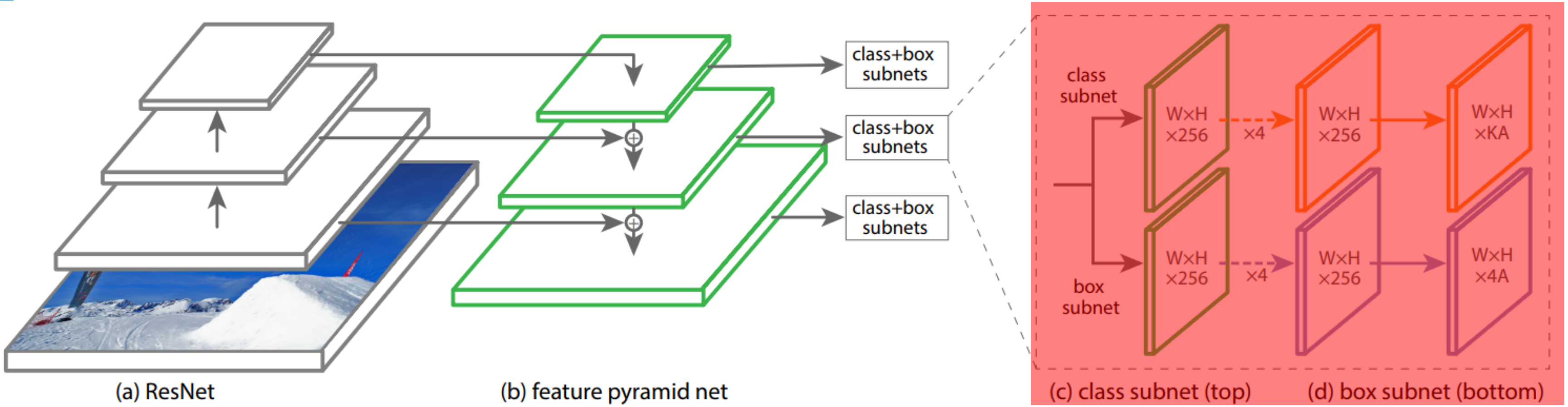
RetinaNet检测算法：特征金字塔



- C是下一轮融合的高层输入，P是用于检测的特征
- 上采样用的是最近邻插值
- 卷积层后不跟BN、ReLU
- 利用FPN对初始的检测层进行强化得到P3，P4，P5
- 再从P5后面使用下采样率为2的卷积层生成P6，P7

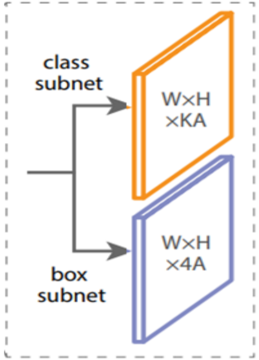


RetinaNet检测算法：检测子网络



- 检测子网络变深：
  1. 分类分支加了4个卷积层
  2. 回归分支加了4个卷积层

- 检测子网络共享
  1. SSD中，每个检测层都有一个检测头
  2. RetinaNet中，所有检测层共用一个检测头



SSD的检测子网络

RetinaNet检测算法：Focal Loss

- 二分类的交叉熵 (cross entropy, CE) 损失函数：

$$\mathrm{CE}(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1-p) & \text{otherwise.} \end{cases}$$

RetinaNet检测算法：Focal Loss

- 二分类的交叉熵 (cross entropy, CE) 损失函数：

$$\mathrm{CE}(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1-p) & \text{otherwise.} \end{cases}$$

- 定义P_t为：

$$p_{\mathrm{t}} = \begin{cases} p & \text{if } y = 1 \\ 1-p & \text{otherwise} \end{cases}$$

RetinaNet检测算法：Focal Loss

- 二分类的交叉熵 (cross entropy, CE) 损失函数：

$$\mathrm{CE}(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1-p) & \text{otherwise.} \end{cases}$$

- 定义P_t为：

$$p_{\mathrm{t}} = \begin{cases} p & \text{if } y = 1 \\ 1-p & \text{otherwise} \end{cases}$$

- 则可以简化为：

$$\mathrm{CE}(p, y) = \mathrm{CE}(p_{\mathrm{t}}) = -\log(p_{\mathrm{t}})$$

RetinaNet检测算法：Focal Loss

- 二分类的交叉熵 (cross entropy, CE) 损失函数：

$$\mathrm{CE}(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1-p) & \text{otherwise.} \end{cases}$$

- 定义P_t为：

$$p_{\mathrm{t}} = \begin{cases} p & \text{if } y = 1 \\ 1-p & \text{otherwise} \end{cases}$$

- 则可以简化为：

$$\mathrm{CE}(p, y) = \mathrm{CE}(p_{\mathrm{t}}) = -\log(p_{\mathrm{t}})$$

P_t 越接近于1，表示分类分的越正确

RetinaNet检测算法：Focal Loss

- 二分类的交叉熵 (cross entropy, CE) 损失函数：

$$\mathrm{CE}(p, y) = \mathrm{CE}(p_{\mathrm{t}}) = -\log(p_{\mathrm{t}})$$

- Focal Loss为：

$$\mathrm{FL}(p_{\mathrm{t}}) = -(1 - p_{\mathrm{t}})^{\gamma} \log(p_{\mathrm{t}}).$$

RetinaNet检测算法：Focal Loss

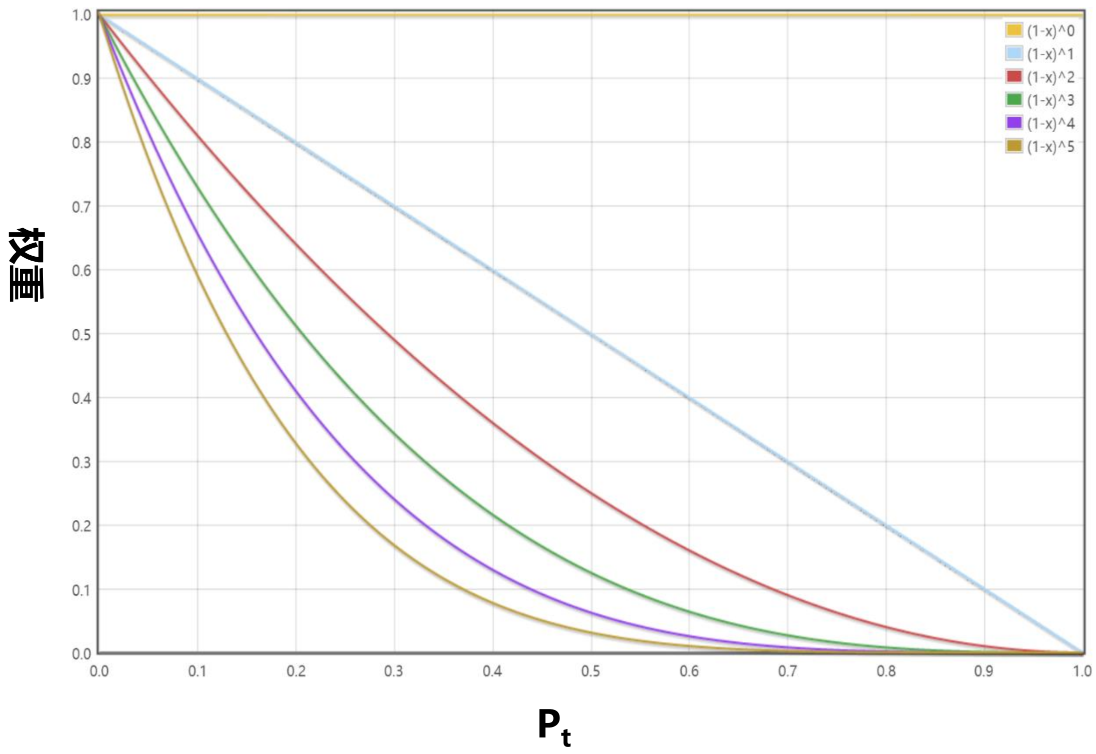
- 二分类的交叉熵 (cross entropy, CE) 损失函数：

$$\mathrm{CE}(p, y) = \mathrm{CE}(p_t) = -\log(p_t)$$

- Focal Loss为：

$$\mathrm{FL}(p_t) = -\boxed{(1 - p_t)^{\gamma}}\log(p_t).$$

RetinaNet检测算法：Focal Loss



$$(1-p_t)^{\gamma}$$

$$\gamma = 0,1,2,3,4,5$$

P_t 越接近于1，

权重越小

RetinaNet检测算法：Focal Loss

- 二分类的交叉熵 (cross entropy, CE) 损失函数：

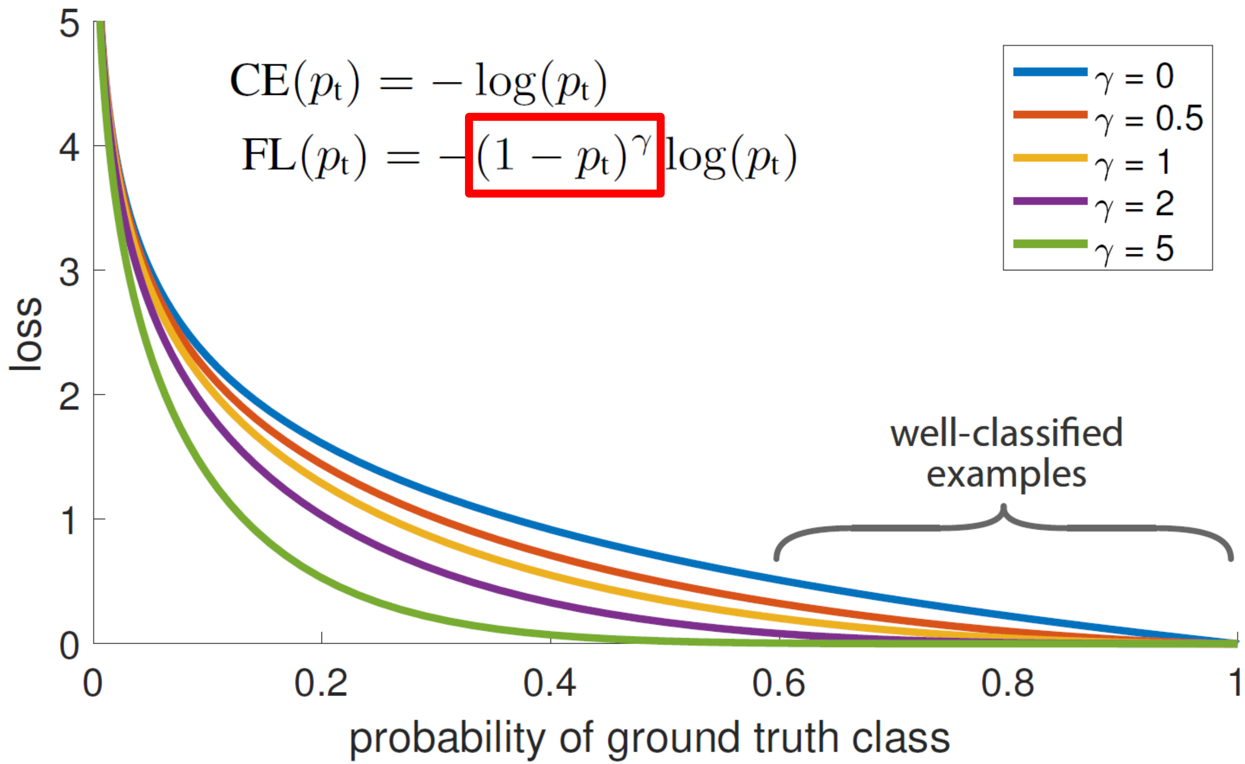
$$\mathrm{CE}(p, y) = \mathrm{CE}(p_t) = -\log(p_t)$$

- Focal Loss为：

$$\mathrm{FL}(p_t) = -\boxed{(1 - p_t)^\gamma}\log(p_t).$$

- **P_t 越接近于1，表示分类分的越正确**
- **分类分的越正确，加权的权重越小**

RetinaNet检测算法：Focal Loss



RetinaNet检测算法：Focal Loss和难样本挖掘

$$\mathrm{CE}(p_t) = -\log(p_t)$$
$$\mathrm{FL}(p_t) = -\boxed{(1-p_t)^\gamma}\log(p_t)$$

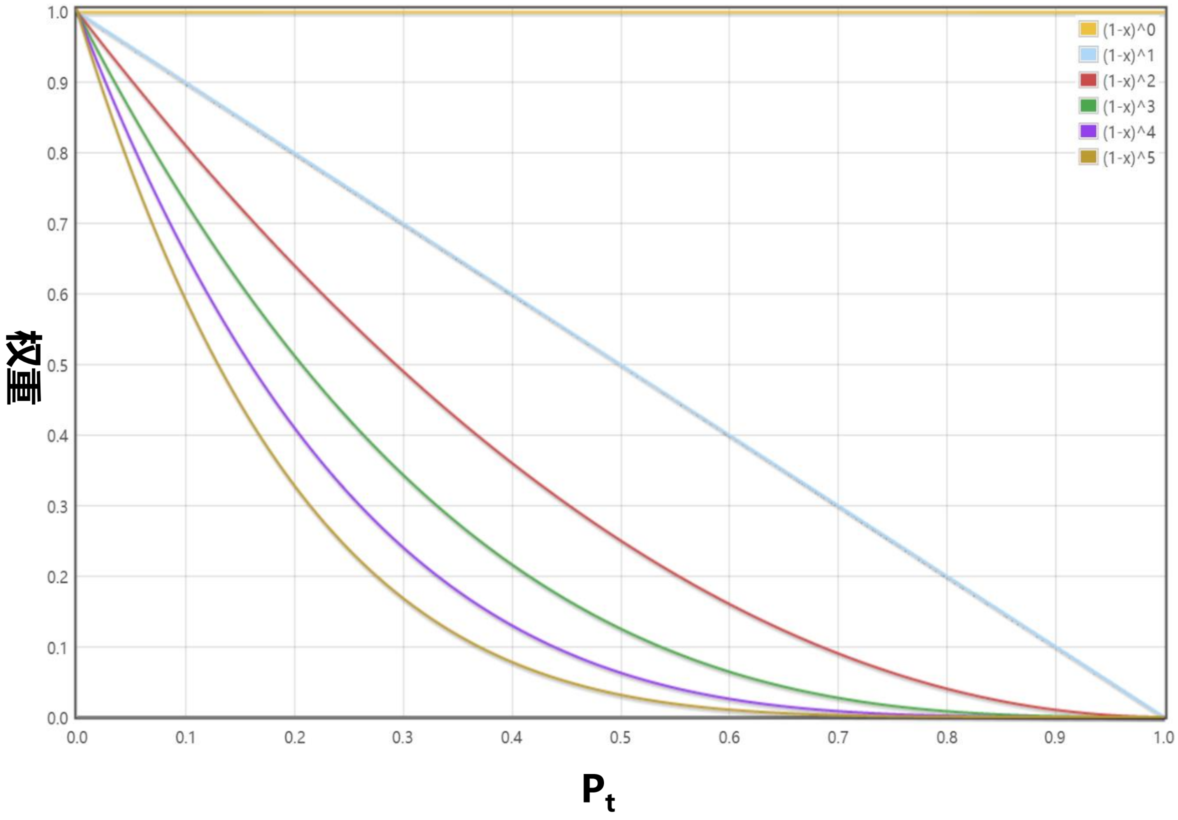
难样本挖掘

- 难样本挖掘是Hard Weight（硬加权）
- 选中的样本，权重为1
- 滤掉的样本，权重为0
- 挖掘固定比例或数量的样本
- 不能充分利用所有的样本

Focal Loss

- Focal Loss是Soft Weight（软加权）
- 分类分的越正确，权重越低
- 分类分的越错误，权重相对越高
- 所有的样本都参与训练
- 能够充分利用所有的样本

RetinaNet检测算法：Focal Loss和难样本挖掘



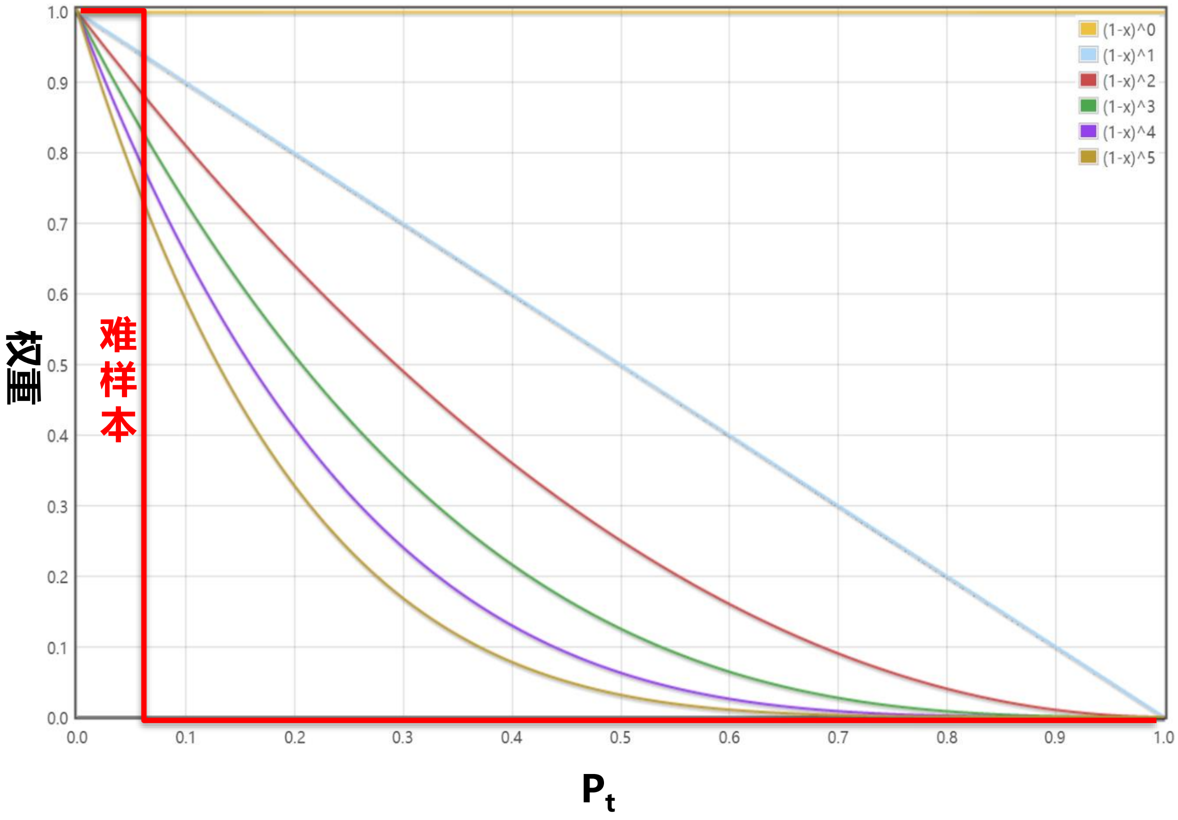
难样本挖掘

- 硬加权
- 选中的样本，权重为1
- 滤掉的样本，权重为0
- 挖掘固定比例或数量的样本
- 不能充分利用所有的样本

Focal Loss

- 软加权
- 分类越正确，权重越低
- 分类越错误，权重相对越高
- 所有的样本都参与训练
- 能够充分利用所有的样本

RetinaNet检测算法：Focal Loss和难样本挖掘



难样本挖掘

- **硬加权**
- **选中的样本，权重为1**
- **滤掉的样本，权重为0**
- **挖掘固定比例或数量的样本**
- **不能充分利用所有的样本**

Focal Loss

- **软加权**
- **分类越正确，权重越低**
- **分类越错误，权重相对越高**
- **所有的样本都参与训练**
- **能够充分利用所有的样本**

RetinaNet检测算法：模型初始化

- ResNet基础网络是用ImageNet预训练的模型进行初始化
- 所有新添加的卷积层都是用σ = 0.01的高斯函数来随机初始化权重w，除了分类子网络中的最后一个卷积层的偏置b，其他卷积层的偏置都初始化为0
- 分类子网络中的最后一个卷积层的偏置b，用下面这个公式进行初始化：

$$b = -\log((1-\pi)/\pi)$$

- 其中π = 0.01，这意味着在训练开始时，每个锚框的前景得分大约是~0.01，目的是降低损失函数的初始值

RetinaNet检测算法：分类子网络初始化

RetinaNet检测算法：分类子网络初始化

$$x \xrightarrow{\text{最后一个卷积层}} y \xrightarrow{\text{Sigmoid函数}} p$$

RetinaNet检测算法：分类子网络初始化

$$x \xrightarrow{\text{最后一个卷积层}} y \xrightarrow{\text{Sigmoid函数}} p$$

$$y = wx + b$$

$$p = \frac{1}{1 + e^{-y}} = \frac{1}{1 + e^{-(wx+b)}}$$

RetinaNet检测算法：分类子网络初始化

$$x \xrightarrow{\text{最后一个卷积层}} y \xrightarrow{\text{Sigmoid函数}} p$$

$$y = wx + b$$

$$p = \frac{1}{1+e^{-y}} = \frac{1}{1+e^{-(wx+b)}}$$

1. 权重w用σ=0.01的Gaussian函数初始化，偏置b初始化为0，那么p为：

$$p = \frac{1}{1+e^{-((w\approx 0)x+0)}} = 0.5$$

RetinaNet检测算法：分类子网络初始化

$$x \xrightarrow{\text{最后一个卷积层}} y \xrightarrow{\text{Sigmoid函数}} p$$

$$y = wx + b$$

$$p = \frac{1}{1 + e^{-y}} = \frac{1}{1 + e^{-(wx+b)}}$$

1. 权重w用σ=0.01的Gaussian函数初始化，偏置b初始化为0，那么p为：

$$p = \frac{1}{1 + e^{-((w\approx 0)x+0)}} = 0.5$$

2. 权重w用σ=0.01的Gaussian函数初始化，偏置b初始化为$-\log((1-\pi)/\pi)$，那么p为：

$$p = \frac{1}{1 + e^{-((w\approx 0)x-\log((1-\pi)/\pi))}} = \pi = 0.01$$

RetinaNet检测算法：分类子网络初始化

P从0.5到0.01的作用

RetinaNet检测算法：分类子网络初始化

P从0.5到0.01的作用

- P=0.5时

$$CE(p,y)=\begin{cases}-\log(p)=-\log(0.5)=0.693, & \text{正样本}\\ -\log(1-p)=-\log(0.5)=0.693, & \text{负样本}\end{cases}$$

RetinaNet检测算法：分类子网络初始化

P从0.5到0.01的作用

- P=0.5时

$$CE(p,y)=\begin{cases}-\log(p)=-\log(0.5)=0.693, & \text{正样本}\\ -\log(1-p)=-\log(0.5)=0.693, & \text{负样本}\end{cases}$$

- P=0.01时

$$CE(p,y)=\begin{cases}-\log(p)=-\log(0.01)=4.605, & \text{正样本}\\ -\log(1-p)=-\log(0.99)=0.010, & \text{负样本}\end{cases}$$

RetinaNet检测算法：分类子网络初始化

P从0.5到0.01的作用

- P=0.5时

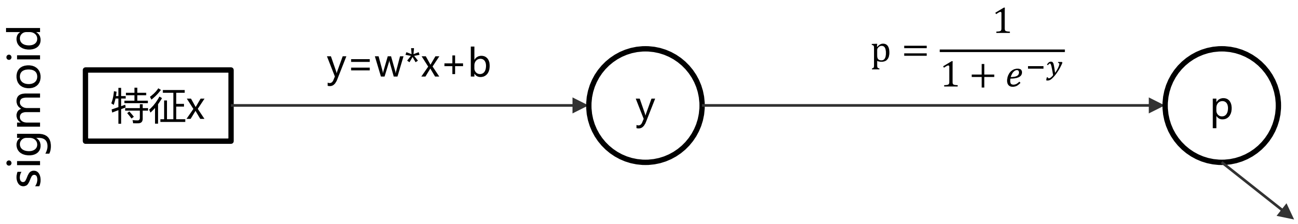
$$CE(p,y)=\begin{cases}-\log(p)=-\log(0.5)=0.693, & \text{正样本} \\ -\log(1-p)=-\log(0.5)=0.693, & \text{负样本}\end{cases}$$

- P=0.01时

$$CE(p,y)=\begin{cases}-\log(p)=-\log(0.01)=4.605, & \text{正样本} \\ -\log(1-p)=-\log(0.99)=0.010, & \text{负样本}\end{cases}$$

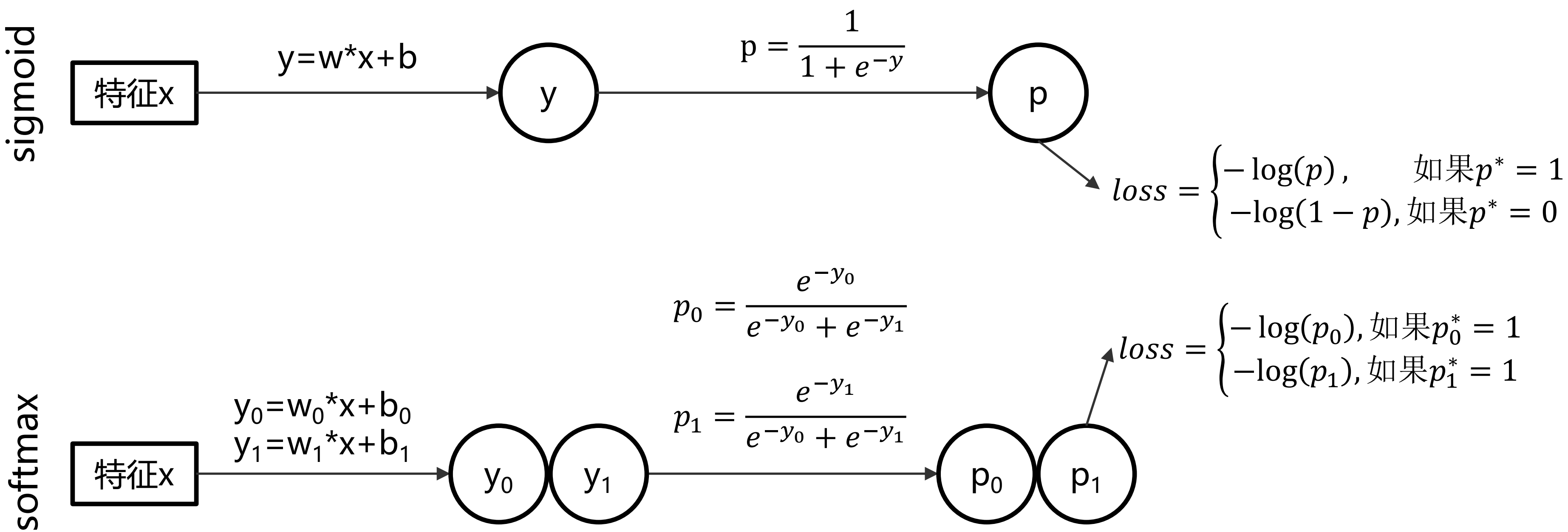
- 当正负样本为1:10时，loss从0.693+0.693*10=7.623变为4.605+0.010*10=4.705，减小1.62倍
- 当正负样本为1:100时，loss从0.693+0.693*100=69.993变为4.605+0.010*100=5.605，减小12.49倍
- 当正负样本为1:1000时，loss从0.693+0.693*1000=693.693变为4.605+0.010*1000=14.605，减小47.50倍
- 当正负样本为1:10000时，loss从0.693+0.693*10000=6930.693变为4.605+0.010*10000=104.605，减小66.26倍

RetinaNet检测算法：Sigmoid和Softmax做二分类

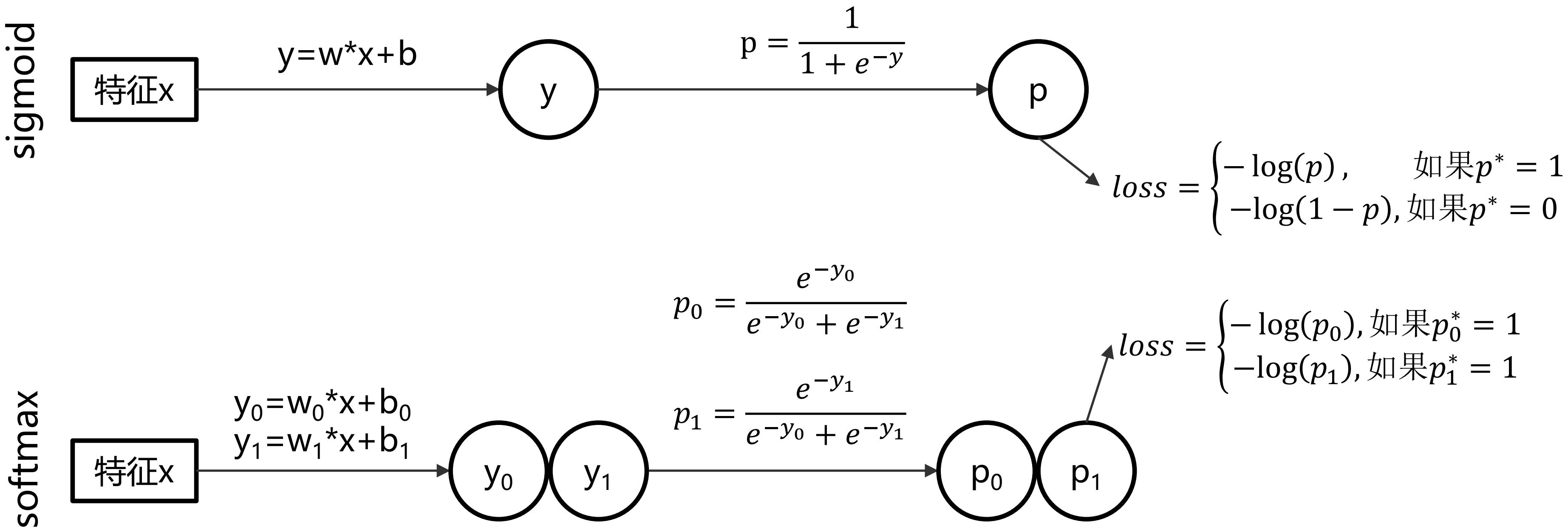


$$loss = \begin{cases} -\log(p)\,, & 如果p^* = 1 \\ -\log(1-p), & 如果p^* = 0 \end{cases}$$

RetinaNet检测算法：Sigmoid和Softmax做二分类



RetinaNet检测算法：Sigmoid和Softmax做二分类



二分类任务时，softmax跟sigmoid等价

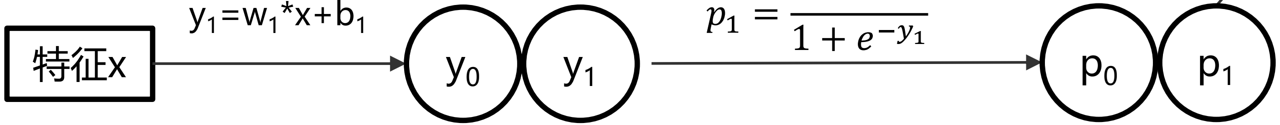
RetinaNet检测算法：Sigmoid和Softmax做多分类

$$loss = \begin{cases} -\log(p_0), & 如果p_0^* = 1 \\ -\log(1-p_0), & 如果p_0^* = 0 \end{cases} + \begin{cases} -\log(p_1), & 如果p_1^* = 1 \\ -\log(1-p_1), & 如果p_1^* = 0 \end{cases}$$

sigmoid

$y_0 = w_0 * x + b_0$
$y_1 = w_1 * x + b_1$

$$p_0 = \frac{1}{1+e^{-y_0}}$$
$$p_1 = \frac{1}{1+e^{-y_1}}$$

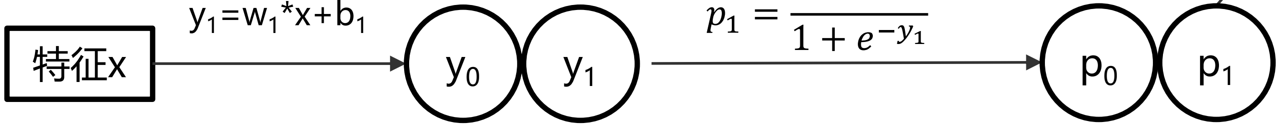


RetinaNet检测算法：Sigmoid和Softmax做多分类

sigmoid

$y_0=w_0*x+b_0$
$y_1=w_1*x+b_1$

$$p_0=\frac{1}{1+e^{-y_0}}$$
$$p_1=\frac{1}{1+e^{-y_1}}$$

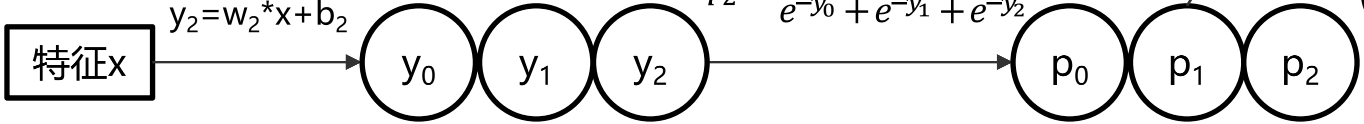


$$loss=\begin{cases}-\log(p_0)\text{，如果}p_0^*=1\\-\log(1-p_0)\text{,如果}p_0^*=0\end{cases}+\begin{cases}-\log(p_1)\text{，如果}p_1^*=1\\-\log(1-p_1)\text{,如果}p_1^*=0\end{cases}$$

softmax

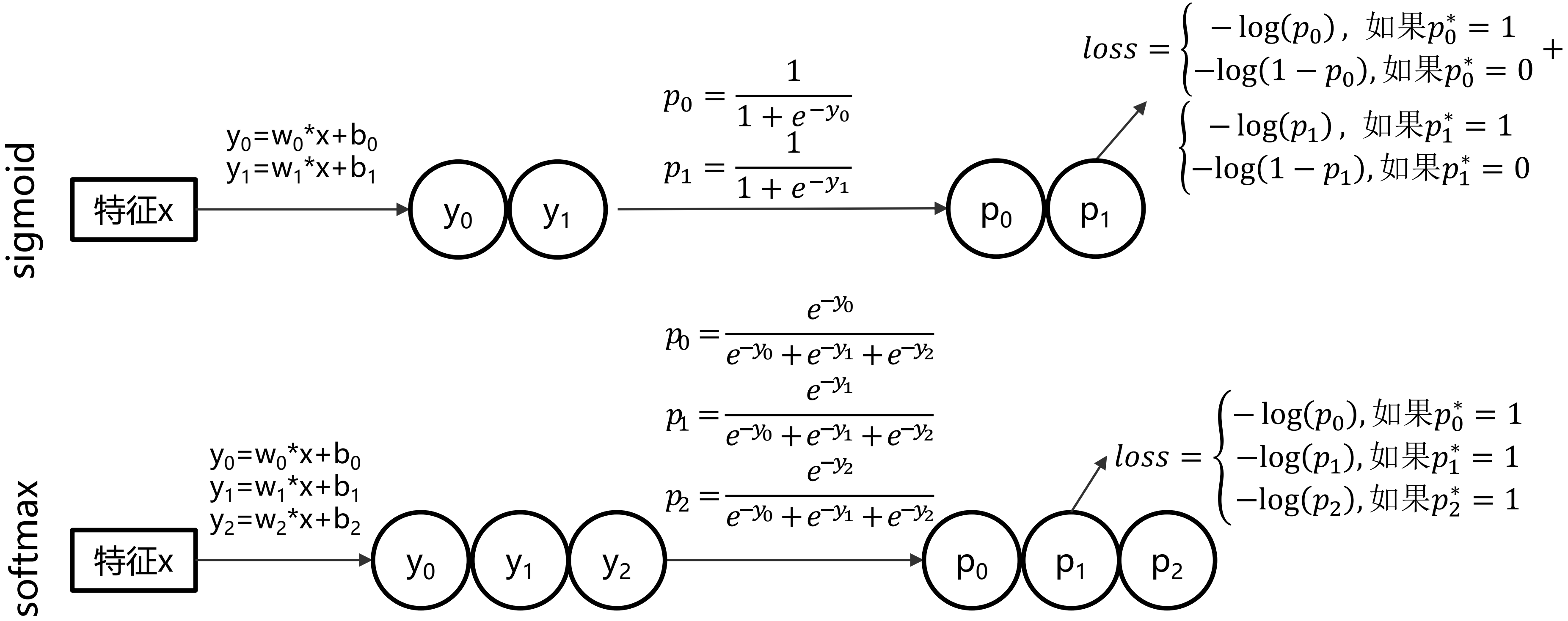
$y_0=w_0*x+b_0$
$y_1=w_1*x+b_1$
$y_2=w_2*x+b_2$

$$p_0=\frac{e^{-y_0}}{e^{-y_0}+e^{-y_1}+e^{-y_2}}$$
$$p_1=\frac{e^{-y_1}}{e^{-y_0}+e^{-y_1}+e^{-y_2}}$$
$$p_2=\frac{e^{-y_2}}{e^{-y_0}+e^{-y_1}+e^{-y_2}}$$



$$loss=\begin{cases}-\log(p_0)\text{,如果}p_0^*=1\\-\log(p_1)\text{,如果}p_1^*=1\\-\log(p_2)\text{,如果}p_2^*=1\end{cases}$$

RetinaNet检测算法：Sigmoid和Softmax做多分类



多（三）分类任务时，softmax跟sigmoid不同

RetinaNet检测算法：Sigmoid和Softmax对比

	Sigmoid	Softmax	异同
检测二分类	1. 最后一层输入是1维度的x 2. 正样本的概率$p_1 = \frac{1}{1+e^{-x}}$ 3. 负样本的概率$p_0 = 1 - p_1$	1. 最后一层输入是2维度的x_0, x_1 2. 正样本的概率$p_1 = \frac{e^{x_1}}{e^{x_0}+e^{x_1}}$ 3. 负样本的概率$p_0 = \frac{e^{x_0}}{e^{x_0}+e^{x_1}}$	1. 换算$x = \frac{x_1}{x_0}$ 2. 二分类时，sigmoid loss与softmax loss等价，只是softmax会多一个维度
检测N分类	1. 最后一层输入是N维度的x_0, x_1, …, x_{N-1} 2. 属于某一类的概率$p_n = \frac{1}{1+e^{-x_n}}$ 3. 不属于某一类的概率$p_n' = 1 - p_n$	1. 最后一层输入是N+1维度的x_0, x_1, …, x_{N-1}, x_N 2. 属于某一类的概率$p_n = \frac{e^{x_n}}{e^{x_0}+e^{x_1}+\cdots+e^{x_n}+\cdots e^{x_N}}$ 3. 不属于某一类的概率$p_n' = 1 - p_n = \frac{e^{x_0}+e^{x_1}+\cdots+e^{x_{n-1}}+e^{x_{n+1}}+\cdots e^{x_N}}{e^{x_0}+e^{x_1}+\cdots+e^{x_n}+\cdots e^{x_N}}$	1. Softmax有类间归一化操作，故有显性的互斥性，即属于某一类，就不能属于其他类 2. Sigmoid没有类间归一化操作，故没有显性的互斥性 3. Softmax会比Sigmoid多一个维度

RetinaNet代码架构解读：配置文件

retinanet_R_50_FPN_1x.yaml

```yaml
_BASE_: "../Base-RetinaNet.yaml"
MODEL:
  WEIGHTS: "detectron2://ImageNetPretrained/MSRA/R-50.pkl"
  RESNETS:
    DEPTH: 50
```

- retinanet_R_50_FPN_1x.yaml

RetinaNet代码架构解读：配置文件

retinanet_R_50_FPN_1x.yaml

```yaml
_BASE_: "../Base-RetinaNet.yaml"
MODEL:
  WEIGHTS: "detectron2://ImageNetPretrained/MSRA/R-50.pkl"
  RESNETS:
    DEPTH: 50
```

- retinanet_R_50_FPN_1x.yaml
- 基础网络是ImageNet预训练ResNet50+FPN

RetinaNet代码架构解读：配置文件

retinanet_R_50_FPN_1x.yaml

```yaml
_BASE_: "../Base-RetinaNet.yaml"
MODEL:
  WEIGHTS: "detectron2://ImageNetPretrained/MSRA/R-50.pkl"
  RESNETS:
    DEPTH: 50
```

Base-RetinaNet.yaml

```yaml
MODEL:
  META_ARCHITECTURE: "RetinaNet"
  BACKBONE:
    NAME: "build_retinanet_resnet_fpn_backbone"
  RESNETS:
    OUT_FEATURES: ["res3", "res4", "res5"]
  ANCHOR_GENERATOR:
    SIZES: !!python/object/apply:eval ["[[x, x * 2**(1.0/3), x * 2**(2.0/3) ] for x in [32, 64, 128, 256, 512 ]]"]
  FPN:
    IN_FEATURES: ["res3", "res4", "res5"]
  RETINANET:
    IOU_THRESHOLDS: [0.4, 0.5]
    IOU_LABELS: [0, -1, 1]
    SMOOTH_L1_LOSS_BETA: 0.0
DATASETS:
  TRAIN: ("coco_2017_train",)
  TEST: ("coco_2017_val",)
SOLVER:
  IMS_PER_BATCH: 16
  BASE_LR: 0.01  # Note that RetinaNet uses a different default learning rate
  STEPS: (60000, 80000)
  MAX_ITER: 90000
INPUT:
  MIN_SIZE_TRAIN: (640, 672, 704, 736, 768, 800)
VERSION: 2
```

- retinanet_R_50_FPN_1x.yaml
- 基础网络是ImageNet预训练ResNet50+FPN

RetinaNet代码架构解读：配置文件

retinanet_R_50_FPN_1x.yaml

```
_BASE_: "../Base-RetinaNet.yaml"
MODEL:
  WEIGHTS: "detectron2://ImageNetPretrained/MSRA/R-50.pkl"
  RESNETS:
    DEPTH: 50
```

Base-RetinaNet.yaml

```
MODEL:
  META_ARCHITECTURE: "RetinaNet"
  BACKBONE:
    NAME: "build_retinanet_resnet_fpn_backbone"
  RESNETS:
    OUT_FEATURES: ["res3", "res4", "res5"]
  ANCHOR_GENERATOR:
    SIZES: !!python/object/apply:eval ["[[x, x * 2**(1.0/3), x * 2**(2.0/3) ] for x in [32, 64, 128, 256, 512 ]]"]
  FPN:
    IN_FEATURES: ["res3", "res4", "res5"]
  RETINANET:
    IOU_THRESHOLDS: [0.4, 0.5]
    IOU_LABELS: [0, -1, 1]
    SMOOTH_L1_LOSS_BETA: 0.0
DATASETS:
  TRAIN: ("coco_2017_train",)
  TEST: ("coco_2017_val",)
SOLVER:
  IMS_PER_BATCH: 16
  BASE_LR: 0.01  # Note that RetinaNet uses a different default learning rate
  STEPS: (60000, 80000)
  MAX_ITER: 90000
INPUT:
  MIN_SIZE_TRAIN: (640, 672, 704, 736, 768, 800)
VERSION: 2
```

- retinanet_R_50_FPN_1x.yaml
- 基础网络是ImageNet预训练ResNet50+FPN
- 初始检测层：res3、res4、res5
- 最终检测层：P3、 P4、 P5、 P6、 P7

RetinaNet代码架构解读：配置文件

retinanet_R_50_FPN_1x.yaml

```yaml
_BASE_: "../Base-RetinaNet.yaml"
MODEL:
  WEIGHTS: "detectron2://ImageNetPretrained/MSRA/R-50.pkl"
  RESNETS:
    DEPTH: 50
```

Base-RetinaNet.yaml

```yaml
MODEL:
  META_ARCHITECTURE: "RetinaNet"
  BACKBONE:
    NAME: "build_retinanet_resnet_fpn_backbone"
  RESNETS:
    OUT_FEATURES: ["res3", "res4", "res5"]
  ANCHOR_GENERATOR:
    SIZES: !!python/object/apply:eval ["[[x, x * 2**(1.0/3), x * 2**(2.0/3) ] for x in [32, 64, 128, 256, 512 ]]"]
  FPN:
    IN_FEATURES: ["res3", "res4", "res5"]
  RETINANET:
    IOU_THRESHOLDS: [0.4, 0.5]
    IOU_LABELS: [0, -1, 1]
    SMOOTH_L1_LOSS_BETA: 0.0
DATASETS:
  TRAIN: ("coco_2017_train",)
  TEST: ("coco_2017_val",)
SOLVER:
  IMS_PER_BATCH: 16
  BASE_LR: 0.01  # Note that RetinaNet uses a different default learning rate
  STEPS: (60000, 80000)
  MAX_ITER: 90000
INPUT:
  MIN_SIZE_TRAIN: (640, 672, 704, 736, 768, 800)
VERSION: 2
```

- retinanet_R_50_FPN_1x.yaml
- 基础网络是ImageNet预训练ResNet50+FPN
- 初始检测层：res3、res4、res5
- 最终检测层：P3、 P4、 P5、 P6、 P7
- 检测层stride：8、 16、 32、 64、 128
- 锚框大小：$32*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、$64*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、$128*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、$256*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、$512*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$
- 锚框比例：每层都是[0.5, 1.0, 2.0]

RetinaNet代码架构解读：配置文件

retinanet_R_50_FPN_1x.yaml

```yaml
_BASE_: "../Base-RetinaNet.yaml"
MODEL:
  WEIGHTS: "detectron2://ImageNetPretrained/MSRA/R-50.pkl"
  RESNETS:
    DEPTH: 50
```

Base-RetinaNet.yaml

```yaml
MODEL:
  META_ARCHITECTURE: "RetinaNet"
  BACKBONE:
    NAME: "build_retinanet_resnet_fpn_backbone"
  RESNETS:
    OUT_FEATURES: ["res3", "res4", "res5"]
  ANCHOR_GENERATOR:
    SIZES: !!python/object/apply:eval ["[[x, x * 2**(1.0/3), x * 2**(2.0/3) ] for x in [32, 64, 128, 256, 512 ]]"]
  FPN:
    IN_FEATURES: ["res3", "res4", "res5"]
  RETINANET:
    IOU_THRESHOLDS: [0.4, 0.5]
    IOU_LABELS: [0, -1, 1]
    SMOOTH_L1_LOSS_BETA: 0.0
DATASETS:
  TRAIN: ("coco_2017_train",)
  TEST: ("coco_2017_val",)
SOLVER:
  IMS_PER_BATCH: 16
  BASE_LR: 0.01  # Note that RetinaNet uses a different default learning rate
  STEPS: (60000, 80000)
  MAX_ITER: 90000
INPUT:
  MIN_SIZE_TRAIN: (640, 672, 704, 736, 768, 800)
VERSION: 2
```

- retinanet_R_50_FPN_1x.yaml
- 基础网络是ImageNet预训练ResNet50+FPN
- 初始检测层：res3、res4、res5
- 最终检测层：P3、 P4、 P5、 P6、 P7
- 检测层stride：8、 16、 32、 64、 128
- 锚框大小：$32*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、$64*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、$128*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、$256*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、$512*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$
- 锚框比例：每层都是[0.5, 1.0, 2.0]
- 锚框匹配：$\theta^+ = 0.5$，$\theta^- = 0.4$

RetinaNet代码架构解读：配置文件

retinanet_R_50_FPN_1x.yaml

```yaml
_BASE_: "../Base-RetinaNet.yaml"
MODEL:
  WEIGHTS: "detectron2://ImageNetPretrained/MSRA/R-50.pkl"
  RESNETS:
    DEPTH: 50
```

Base-RetinaNet.yaml

```yaml
MODEL:
  META_ARCHITECTURE: "RetinaNet"
  BACKBONE:
    NAME: "build_retinanet_resnet_fpn_backbone"
  RESNETS:
    OUT_FEATURES: ["res3", "res4", "res5"]
  ANCHOR_GENERATOR:
    SIZES: !!python/object/apply:eval ["[[x, x * 2**(1.0/3), x * 2**(2.0/3) ] for x in [32, 64, 128, 256, 512 ]]"]
  FPN:
    IN_FEATURES: ["res3", "res4", "res5"]
  RETINANET:
    IOU_THRESHOLDS: [0.4, 0.5]
    IOU_LABELS: [0, -1, 1]
    SMOOTH_L1_LOSS_BETA: 0.0
DATASETS:
  TRAIN: ("coco_2017_train",)
  TEST: ("coco_2017_val",)
SOLVER:
  IMS_PER_BATCH: 16
  BASE_LR: 0.01  # Note that RetinaNet uses a different default learning rate
  STEPS: (60000, 80000)
  MAX_ITER: 90000
INPUT:
  MIN_SIZE_TRAIN: (640, 672, 704, 736, 768, 800)
VERSION: 2
```

- retinanet_R_50_FPN_1x.yaml
- 基础网络是ImageNet预训练ResNet50+FPN
- 初始检测层：res3、res4、res5
- 最终检测层：P3、 P4、 P5、 P6、 P7
- 检测层stride：8、 16、 32、 64、 128
- 锚框大小：$32*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、 $64*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、 $128*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、 $256*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、 $512*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$
- 锚框比例：每层都是[0.5, 1.0, 2.0]
- 锚框匹配：$\theta^{+}=0.5$， $\theta^{-}=0.4$
- 使用数据：COCO 2017的训练集和验证集

RetinaNet代码架构解读：配置文件

retinanet_R_50_FPN_1x.yaml

```yaml
_BASE_: "../Base-RetinaNet.yaml"
MODEL:
  WEIGHTS: "detectron2://ImageNetPretrained/MSRA/R-50.pkl"
  RESNETS:
    DEPTH: 50
```

Base-RetinaNet.yaml

```yaml
MODEL:
  META_ARCHITECTURE: "RetinaNet"
  BACKBONE:
    NAME: "build_retinanet_resnet_fpn_backbone"
  RESNETS:
    OUT_FEATURES: ["res3", "res4", "res5"]
  ANCHOR_GENERATOR:
    SIZES: !!python/object/apply:eval ["[[x, x * 2**(1.0/3), x * 2**(2.0/3) ] for x in [32, 64, 128, 256, 512 ]]"]
  FPN:
    IN_FEATURES: ["res3", "res4", "res5"]
  RETINANET:
    IOU_THRESHOLDS: [0.4, 0.5]
    IOU_LABELS: [0, -1, 1]
    SMOOTH_L1_LOSS_BETA: 0.0
DATASETS:
  TRAIN: ("coco_2017_train",)
  TEST: ("coco_2017_val",)
SOLVER:
  IMS_PER_BATCH: 16
  BASE_LR: 0.01  # Note that RetinaNet uses a different default learning rate
  STEPS: (60000, 80000)
  MAX_ITER: 90000
INPUT:
  MIN_SIZE_TRAIN: (640, 672, 704, 736, 768, 800)
VERSION: 2
```

- retinanet_R_50_FPN_1x.yaml
- 基础网络是ImageNet预训练ResNet50+FPN
- 初始检测层：res3、res4、res5
- 最终检测层：P3、 P4、 P5、 P6、 P7
- 检测层stride：8、 16、 32、 64、 128
- 锚框大小： $32*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、 $64*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、 $128*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、 $256*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、 $512*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$
- 锚框比例：每层都是[0.5, 1.0, 2.0]
- 锚框匹配：$\theta^{+} = 0.5$ ，$\theta^{-} = 0.4$
- 使用数据：COCO 2017的训练集和验证集
- 优化策略：组批次16，lr 0.01，9万次迭代

RetinaNet代码架构解读：配置文件

retinanet_R_50_FPN_1x.yaml

```yaml
_BASE_: "../Base-RetinaNet.yaml"
MODEL:
  WEIGHTS: "detectron2://ImageNetPretrained/MSRA/R-50.pkl"
  RESNETS:
    DEPTH: 50
```

Base-RetinaNet.yaml

```yaml
MODEL:
  META_ARCHITECTURE: "RetinaNet"
  BACKBONE:
    NAME: "build_retinanet_resnet_fpn_backbone"
  RESNETS:
    OUT_FEATURES: ["res3", "res4", "res5"]
  ANCHOR_GENERATOR:
    SIZES: !!python/object/apply:eval ["[[x, x * 2**(1.0/3), x * 2**(2.0/3) ] for x in [32, 64, 128, 256, 512 ]]"]
  FPN:
    IN_FEATURES: ["res3", "res4", "res5"]
  RETINANET:
    IOU_THRESHOLDS: [0.4, 0.5]
    IOU_LABELS: [0, -1, 1]
    SMOOTH_L1_LOSS_BETA: 0.0
DATASETS:
  TRAIN: ("coco_2017_train",)
  TEST: ("coco_2017_val",)
SOLVER:
  IMS_PER_BATCH: 16
  BASE_LR: 0.01  # Note that RetinaNet uses a different default learning rate
  STEPS: (60000, 80000)
  MAX_ITER: 90000
INPUT:
  MIN_SIZE_TRAIN: (640, 672, 704, 736, 768, 800)
VERSION: 2
```

- retinanet_R_50_FPN_1x.yaml
- 基础网络是ImageNet预训练ResNet50+FPN
- 初始检测层：res3、res4、res5
- 最终检测层：P3、 P4、 P5、 P6、 P7
- 检测层stride：8、 16、 32、 64、 128
- 锚框大小：$32*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、 $64*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、 $128*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、 $256*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$、 $512*[1,2^{\frac{1}{3}},2^{\frac{2}{3}}]$
- 锚框比例：每层都是[0.5, 1.0, 2.0]
- 锚框匹配：$\theta^{+} = 0.5$，$\theta^{-} = 0.4$
- 使用数据：COCO 2017的训练集和验证集
- 优化策略：组批次16，lr 0.01，9万次迭代
- 多尺度训练：[648, 672, 704, 736, 768, 800]

RetinaNet代码架构解读：算法流程

构建

- 模型构建
- 优化器构建
- 数据构建

训练

- 数据处理
- 特征提取
- 网络训练

测试

- 数据处理
- 特征提取
- 生成检测结果

RetinaNet代码架构解读：算法流程

构建

- 模型构建
- 优化器构建
- 数据构建

RetinaNet代码架构解读：构建

```
model = self.build_model(cfg)
optimizer = self.build_optimizer(cfg, model)
data_loader = self.build_train_loader(cfg)
```

RetinaNet代码架构解读：模型构建

```
model = self.build_model(cfg)
optimizer = self.build_optimizer(cfg, model)
data_loader = self.build_train_loader(cfg)
```

```
self.backbone = build_backbone(cfg)

backbone_shape = self.backbone.output_shape()
feature_shapes = [backbone_shape[f] for f in self.in_features]
self.head = RetinaNetHead(cfg, feature_shapes)
self.anchor_generator = build_anchor_generator(cfg, feature_shapes)

# Matching and loss
self.box2box_transform = Box2BoxTransform(weights=cfg.MODEL.RPN.BBOX_REG_WEIGHTS)
self.anchor_matcher = Matcher(
    cfg.MODEL.RETINANET.IOU_THRESHOLDS,
    cfg.MODEL.RETINANET.IOU_LABELS,
    allow_low_quality_matches=True,
)
```

RetinaNet代码架构解读：模型构建

```python
model = self.build_model(cfg)
optimizer = self.build_optimizer(cfg, model)
data_loader = self.build_train_loader(cfg)
```

```python
self.backbone = build_backbone(cfg)

backbone_shape = self.backbone.output_shape()
feature_shapes = [backbone_shape[f] for f in self.in_features]
self.head = RetinaNetHead(cfg, feature_shapes)
self.anchor_generator = build_anchor_generator(cfg, feature_shapes)

# Matching and loss
self.box2box_transform = Box2BoxTransform(weights=cfg.MODEL.RPN.BBOX_REG_WEIGHTS)
self.anchor_matcher = Matcher(
    cfg.MODEL.RETINANET.IOU_THRESHOLDS,
    cfg.MODEL.RETINANET.IOU_LABELS,
    allow_low_quality_matches=True,
)
```

```python
def build_retinanet_resnet_fpn_backbone(cfg, input_shape: ShapeSpec):
    """
    Args:
        cfg: a detectron2 CfgNode

    Returns:
        backbone (Backbone): backbone module, must be a subclass of :class:`Backbone`.
    """
    bottom_up = build_resnet_backbone(cfg, input_shape)
    in_features = cfg.MODEL.FPN.IN_FEATURES
    out_channels = cfg.MODEL.FPN.OUT_CHANNELS
    in_channels_p6p7 = bottom_up.output_shape()["res5"].channels
    backbone = FPN(
        bottom_up=bottom_up,
        in_features=in_features,
        out_channels=out_channels,
        norm=cfg.MODEL.FPN.NORM,
        top_block=LastLevelP6P7(in_channels_p6p7, out_channels),
        fuse_type=cfg.MODEL.FPN.FUSE_TYPE,
    )
    return backbone
```

ResNet构建

FPN构建

RetinaNet代码架构解读：模型构建

```python
model = self.build_model(cfg)
optimizer = self.build_optimizer(cfg, model)
data_loader = self.build_train_loader(cfg)
```

```python
self.backbone = build_backbone(cfg)

backbone_shape = self.backbone.output_shape()
feature_shapes = [backbone_shape[f] for f in self.in_features]
self.head = RetinaNetHead(cfg, feature_shapes)
self.anchor_generator = build_anchor_generator(cfg, feature_shapes)

# Matching and loss
self.box2box_transform = Box2BoxTransform(weights=cfg.MODEL.RPN.BBOX_REG_WEIGHTS)
self.anchor_matcher = Matcher(
    cfg.MODEL.RETINANET.IOU_THRESHOLDS,
    cfg.MODEL.RETINANET.IOU_LABELS,
    allow_low_quality_matches=True,
)
```

锚框预测的构建

```python
def build_retinanet_resnet_fpn_backbone(cfg, input_shape: ShapeSpec):
    """
    Args:
        cfg: a detectron2 CfgNode

    Returns:
        backbone (Backbone): backbone module, must be a subclass of :class:`Backbone`.
    """
    bottom_up = build_resnet_backbone(cfg, input_shape)
    in_features = cfg.MODEL.FPN.IN_FEATURES
    out_channels = cfg.MODEL.FPN.OUT_CHANNELS
    in_channels_p6p7 = bottom_up.output_shape()["res5"].channels
    backbone = FPN(
        bottom_up=bottom_up,
        in_features=in_features,
        out_channels=out_channels,
        norm=cfg.MODEL.FPN.NORM,
        top_block=LastLevelP6P7(in_channels_p6p7, out_channels),
        fuse_type=cfg.MODEL.FPN.FUSE_TYPE,
    )
    return backbone
```

ResNet构建

FPN构建

RetinaNet代码架构解读：模型构建

```
model = self.build_model(cfg)
optimizer = self.build_optimizer(cfg, model)
data_loader = self.build_train_loader(cfg)
```

```
self.backbone = build_backbone(cfg)

backbone_shape = self.backbone.output_shape()
feature_shapes = [backbone_shape[f] for f in self.in_features]
self.head = RetinaNetHead(cfg, feature_shapes)
self.anchor_generator = build_anchor_generator(cfg, feature_shapes)

# Matching and loss
self.box2box_transform = Box2BoxTransform(weights=cfg.MODEL.RPN.BBOX_REG_WEIGHTS)
self.anchor_matcher = Matcher(
    cfg.MODEL.RETINANET.IOU_THRESHOLDS,
    cfg.MODEL.RETINANET.IOU_LABELS,
    allow_low_quality_matches=True,
)
```

锚框预测的构建

锚框的构建

```
def build_retinanet_resnet_fpn_backbone(cfg, input_shape: ShapeSpec):
    """
    Args:
        cfg: a detectron2 CfgNode

    Returns:
        backbone (Backbone): backbone module, must be a subclass of :class:`Backbone`.
    """
    bottom_up = build_resnet_backbone(cfg, input_shape)
    in_features = cfg.MODEL.FPN.IN_FEATURES
    out_channels = cfg.MODEL.FPN.OUT_CHANNELS
    in_channels_p6p7 = bottom_up.output_shape()["res5"].channels
    backbone = FPN(
        bottom_up=bottom_up,
        in_features=in_features,
        out_channels=out_channels,
        norm=cfg.MODEL.FPN.NORM,
        top_block=LastLevelP6P7(in_channels_p6p7, out_channels),
        fuse_type=cfg.MODEL.FPN.FUSE_TYPE,
    )
    return backbone
```

ResNet构建

FPN构建

RetinaNet代码架构解读：模型构建

```python
model = self.build_model(cfg)
optimizer = self.build_optimizer(cfg, model)
data_loader = self.build_train_loader(cfg)
```

```python
self.backbone = build_backbone(cfg)

backbone_shape = self.backbone.output_shape()
feature_shapes = [backbone_shape[f] for f in self.in_features]
self.head = RetinaNetHead(cfg, feature_shapes)
self.anchor_generator = build_anchor_generator(cfg, feature_shapes)

# Matching and loss
self.box2box_transform = Box2BoxTransform(weights=cfg.MODEL.RPN.B
self.anchor_matcher = Matcher(
    cfg.MODEL.RETINANET.IOU_THRESHOLDS,
    cfg.MODEL.RETINANET.IOU_LABELS,
    allow_low_quality_matches=True,
)
```

锚框预测的构建

锚框的构建

锚框变换的构建

```python
def build_retinanet_resnet_fpn_backbone(cfg, input_shape: ShapeSpec):
    """
    Args:
        cfg: a detectron2 CfgNode

    Returns:
        backbone (Backbone): backbone module, must be a subclass of :class:`Backbone`.
    """
    bottom_up = build_resnet_backbone(cfg, input_shape)
    in_features = cfg.MODEL.FPN.IN_FEATURES
    out_channels = cfg.MODEL.FPN.OUT_CHANNELS
    in_channels_p6p7 = bottom_up.output_shape()["res5"].channels
    backbone = FPN(
        bottom_up=bottom_up,
        in_features=in_features,
        out_channels=out_channels,
        norm=cfg.MODEL.FPN.NORM,
        top_block=LastLevelP6P7(in_channels_p6p7, out_channels),
        fuse_type=cfg.MODEL.FPN.FUSE_TYPE,
    )
    return backbone
```

ResNet构建

FPN构建

RetinaNet代码架构解读：模型构建

```python
model = self.build_model(cfg)
optimizer = self.build_optimizer(cfg, model)
data_loader = self.build_train_loader(cfg)
```

```python
self.backbone = build_backbone(cfg)

backbone_shape = self.backbone.output_shape()
feature_shapes = [backbone_shape[f] for f in self.in_features]
self.head = RetinaNetHead(cfg, feature_shapes)
self.anchor_generator = build_anchor_generator(cfg, feature_shapes)

# Matching and loss
self.box2box_transform = Box2BoxTransform(weights=cfg.MODEL.RPN.B
self.anchor_matcher = Matcher(
    cfg.MODEL.RETINANET.IOU_THRESHOLDS,
    cfg.MODEL.RETINANET.IOU_LABELS,
    allow_low_quality_matches=True,
)
```

锚框预测的构建

锚框的构建

锚框变换的构建

锚框匹配的构建

```python
def build_retinanet_resnet_fpn_backbone(cfg, input_shape: ShapeSpec):
    """
    Args:
        cfg: a detectron2 CfgNode

    Returns:
        backbone (Backbone): backbone module, must be a subclass of :class:`Backbone`.
    """
    bottom_up = build_resnet_backbone(cfg, input_shape)
    in_features = cfg.MODEL.FPN.IN_FEATURES
    out_channels = cfg.MODEL.FPN.OUT_CHANNELS
    in_channels_p6p7 = bottom_up.output_shape()["res5"].channels
    backbone = FPN(
        bottom_up=bottom_up,
        in_features=in_features,
        out_channels=out_channels,
        norm=cfg.MODEL.FPN.NORM,
        top_block=LastLevelP6P7(in_channels_p6p7, out_channels),
        fuse_type=cfg.MODEL.FPN.FUSE_TYPE,
    )
    return backbone
```

ResNet构建

FPN构建

RetinaNet代码架构解读：优化器构建

```
model = self.build_model(cfg)
optimizer = self.build_optimizer(cfg, model)
data_loader = self.build_train_loader(cfg)
```

```
self.scheduler = self.build_lr_scheduler(cfg, optimizer)
# Assume no other objects need to be checkpointed.
# We can later make it checkpoint the stateful hooks
self.checkpointer = DetectionCheckpointer(
    # Assume you want to save checkpoints together with logs/statistics
    model,
    cfg.OUTPUT_DIR,
    optimizer=optimizer,
    scheduler=self.scheduler,
)
self.start_iter = 0
self.max_iter = cfg.SOLVER.MAX_ITER
self.cfg = cfg
```

学习率策略

训练模型保存

训练迭代次数

RetinaNet代码架构解读：数据构建

```
model = self.build_model(cfg)
optimizer = self.build_optimizer(cfg, model)
data_loader = self.build_train_loader(cfg)
```

```
dataset_dicts = get_detection_dataset_dicts(
    cfg.DATASETS.TRAIN,
    filter_empty=cfg.DATALOADER.FILTER_EMPTY_ANNOTATIONS,
    min_keypoints=cfg.MODEL.ROI_KEYPOINT_HEAD.MIN_KEYPOINTS_PER_IMAGE
    if cfg.MODEL.KEYPOINT_ON
    else 0,
    proposal_files=cfg.DATASETS.PROPOSAL_FILES_TRAIN if cfg.MODEL.LOAD_PROPOSALS else None,
)
dataset = DatasetFromList(dataset_dicts, copy=False)

if mapper is None:
    mapper = DatasetMapper(cfg, True)
dataset = MapDataset(dataset, mapper)

sampler_name = cfg.DATALOADER.SAMPLER_TRAIN
logger = logging.getLogger(__name__)
logger.info("Using training sampler {}".format(sampler_name))
# TODO avoid if-else?
if sampler_name == "TrainingSampler":
    sampler = samplers.TrainingSampler(len(dataset))
elif sampler_name == "RepeatFactorTrainingSampler":
    sampler = samplers.RepeatFactorTrainingSampler(
        dataset_dicts, cfg.DATALOADER.REPEAT_THRESHOLD
    )
else:
    raise ValueError("Unknown training sampler: {}".format(sampler_name))
return build_batch_data_loader(
    dataset,
    sampler,
    cfg.SOLVER.IMS_PER_BATCH,
    aspect_ratio_grouping=cfg.DATALOADER.ASPECT_RATIO_GROUPING,
    num_workers=cfg.DATALOADER.NUM_WORKERS,
```

RetinaNet代码架构解读：算法流程

构建

- 模型构建
- 优化器构建
- 数据构建

训练

- 数据处理
- 特征提取
- 网络训练

RetinaNet代码架构解读：数据处理

读图像

```
image = utils.read_image(dataset_dict["file_name"], format=self.img_format)
```

图像变换

（缩放、水平翻转）

```
image, transforms = T.apply_transform_gens(self.tfm_gens, image)
```

标注变换

```
utils.transform_instance_annotations(
    obj, transforms, image_shape, keypoint_hflip_indices=self.keypoint_hflip_indices
```

过滤标注

```
dataset_dict["instances"] = utils.filter_empty_instances(instances)
```

图像归一化

```
images = [(x - self.pixel_mean) / self.pixel_std for x in images]
```

图像组batch

```
images = ImageList.from_tensors(images, self.backbone.size_divisibility)
```

RetinaNet代码架构解读：特征提取

```python
bottom_up_features = self.bottom_up(x)
x = [bottom_up_features[f] for f in self.in_features[::-1]]
results = []
prev_features = self.lateral_convs[0](x[0])
results.append(self.output_convs[0](prev_features))
for features, lateral_conv, output_conv in zip(
    x[1:], self.lateral_convs[1:], self.output_convs[1:]
):
    top_down_features = F.interpolate(prev_features, scale_factor=2, mode="nearest")
    lateral_features = lateral_conv(features)
    prev_features = lateral_features + top_down_features
    if self._fuse_type == "avg":
        prev_features /= 2
    results.insert(0, output_conv(prev_features))

if self.top_block is not None:
    top_block_in_feature = bottom_up_features.get(self.top_block.in_feature, None)
    if top_block_in_feature is None:
        top_block_in_feature = results[self._out_features.index(self.top_block.in_feature)]
    results.extend(self.top_block(top_block_in_feature))
assert len(self._out_features) == len(results)
return dict(zip(self._out_features, results))
```

ResNet特征提取

FPN特征提取

RetinaNet代码架构解读：网络训练

```
anchors = self.anchor_generator(features)
```

生成锚框

```
pred_logits, pred_anchor_deltas = self.head(features)
```

预测锚框的类别和偏移

```
gt_labels, gt_boxes = self.label_anchors(anchors, gt_instances)
```

计算锚框类别和偏移的真值

```
loss_cls = sigmoid_focal_loss_jit(
    cat(pred_logits, dim=1)[valid_mask],
    gt_labels_target.to(pred_logits[0].dtype),
    alpha=self.focal_loss_alpha,
    gamma=self.focal_loss_gamma,
    reduction="sum",
)
```

计算分类Focal Loss损失函数

```
loss_box_reg = smooth_l1_loss(
    cat(pred_anchor_deltas, dim=1)[pos_mask],
    gt_anchor_deltas[pos_mask],
    beta=self.smooth_l1_loss_beta,
    reduction="sum",
)
```

计算回归SoomthL1损失函数

RetinaNet代码架构解读：算法流程

构建

- 模型构建
- 优化器构建
- 数据构建

训练

- 数据处理
- 特征提取
- 网络训练

测试

- 数据处理
- 特征提取
- 生成检测结果

RetinaNet代码架构解读：数据处理

读图像

```
image = utils.read_image(dataset_dict["file_name"], format=self.img_format)
```

图像变换

（缩放）

```
image, transforms = T.apply_transform_gens(self.tfm_gens, image)
```

图像归一化

```
images = [(x - self.pixel_mean) / self.pixel_std for x in images]
```

图像组batch

```
images = ImageList.from_tensors(images, self.backbone.size_divisibility)
```

RetinaNet代码架构解读：特征提取

```python
bottom_up_features = self.bottom_up(x)
x = [bottom_up_features[f] for f in self.in_features[::-1]]
results = []
prev_features = self.lateral_convs[0](x[0])
results.append(self.output_convs[0](prev_features))
for features, lateral_conv, output_conv in zip(
    x[1:], self.lateral_convs[1:], self.output_convs[1:]
):
    top_down_features = F.interpolate(prev_features, scale_factor=2, mode="nearest")
    lateral_features = lateral_conv(features)
    prev_features = lateral_features + top_down_features
    if self._fuse_type == "avg":
        prev_features /= 2
    results.insert(0, output_conv(prev_features))

if self.top_block is not None:
    top_block_in_feature = bottom_up_features.get(self.top_block.in_feature, None)
    if top_block_in_feature is None:
        top_block_in_feature = results[self._out_features.index(self.top_block.in_feature)]
    results.extend(self.top_block(top_block_in_feature))
assert len(self._out_features) == len(results)
return dict(zip(self._out_features, results))
```

ResNet特征提取

FPN特征提取

RetinaNet代码架构解读：生成检测结果

```
anchors = self.anchor_generator(features)
```

生成锚框

```
pred_logits, pred_anchor_deltas = self.head(features)
```

预测锚框的
类别和偏移

```
predicted_boxes = self.box2box_transform.apply_deltas(box_reg_i, anchors_i.tensor)
```

锚框加上
预测的偏移

```
keep = batched_nms(boxes_all, scores_all, class_idxs_all, self.nms_threshold)
```

NMS过滤
冗余检测结果

RetinaNet代码调试演示

请观看演示视频

课程作业

■ 单步调试RetinaNet代码

1. 利用配好的detectron2物体检测平台，使用PyCharm软件，单步调试如下配置(https://github.com/facebookresearch/detectron2/blob/master/configs/COCO-Detection/retinanet_R_50_FPN_1x.yaml)的RetinaNet代码
2. 把RetinaNet中每个细节与代码对应上，真正弄懂RetinaNet的整个流程
3. (可选) 如果硬件条件允许，可以使用8卡GPU训一个模型，看精度是否与官方一致

结语

感谢各位聆听!

Thanks for Listening