# 第 5 章

# 目 标 检 测

目标检测、物体检测及缺陷检测是卷积神经网络的重要应用场景，其主要任务是找出图像或者视频中所感兴趣的物体(目标)的位置并指明所属类别，在计算机视觉领域中占有重要地位。

本章将详细介绍经典目标检测网络，并结合深度学习框架从零重现 R-CNN、Faster R-CNN、SSD、YOLOv1、YOLOv2、YOLOv3、YOLOv4、YOLOv5、YOLOv7 模型代码。通过阅读本章内容，读者不仅可以掌握经典目标检测算法的原理，同时也具备目标检测算法代码的实现及调优的能力。

## 5.1 标签处理及代码

相对于图像分类任务，目标检测任务需要将真实感兴趣的目标标识出来，然后交给神经网络训练学习。标签处理包括数据标注、标签数据格式转换、标签数据读取等过程。

数据标注可使用开源工具 labelimg，安装方法为 pip install labelimg。安装成功后，在命令行中输入 labelimg 即可打开该软件。选择 Open Dir 配置训练图片的文件夹。选择 Change Save Dir 配置标注目标的保存目录，如图 5-1 所示。

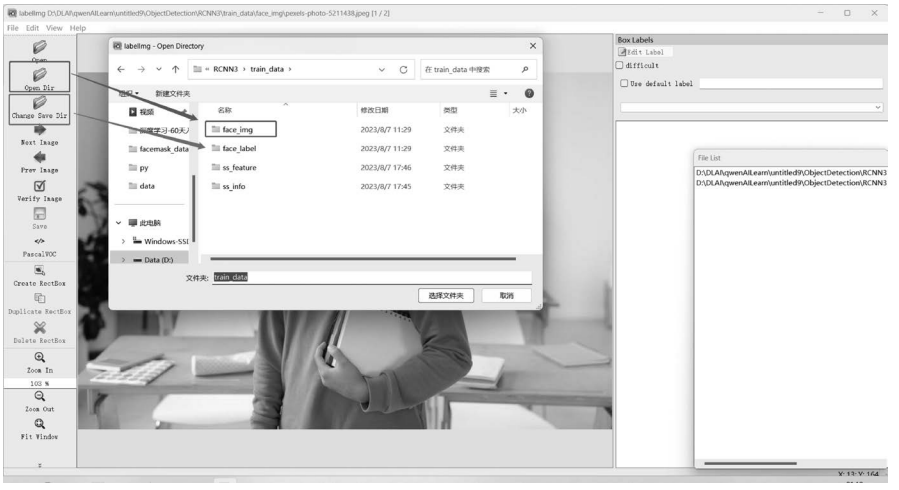



图 5-1 labelimg 选择训练文件夹和标注目标文件夹

加载图片后，选择 Create RectBox 同时默认保留标注文件格式为 PascalVoc 格式，拖动鼠标在目标区域生成一个矩形框后将矩形框的分类填写为 face_mask，如图 5-2 所示。

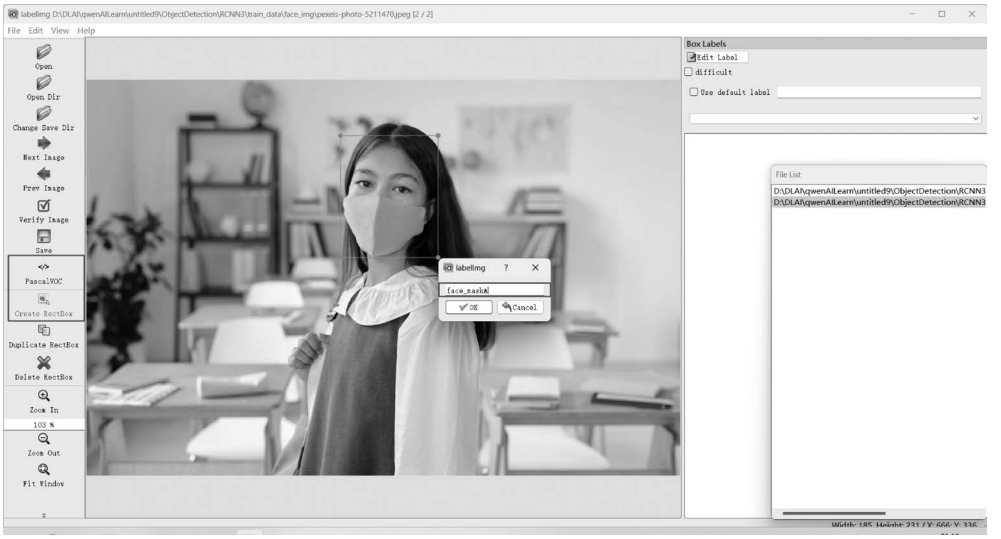



图 5-2　labelimg 数据标注方法

标注完成后会在图 5-1 设置的文件夹中生成与图片名称相同的 XML 文件，其< object >标签表示有一个目标区域，< xmin >、< ymin >为目标图像左上角坐标点的位置，< xmax >、< ymax >为目标图像右下角坐标点的位置，< name >为当前目标图像的类别，如图 5-3 所示。

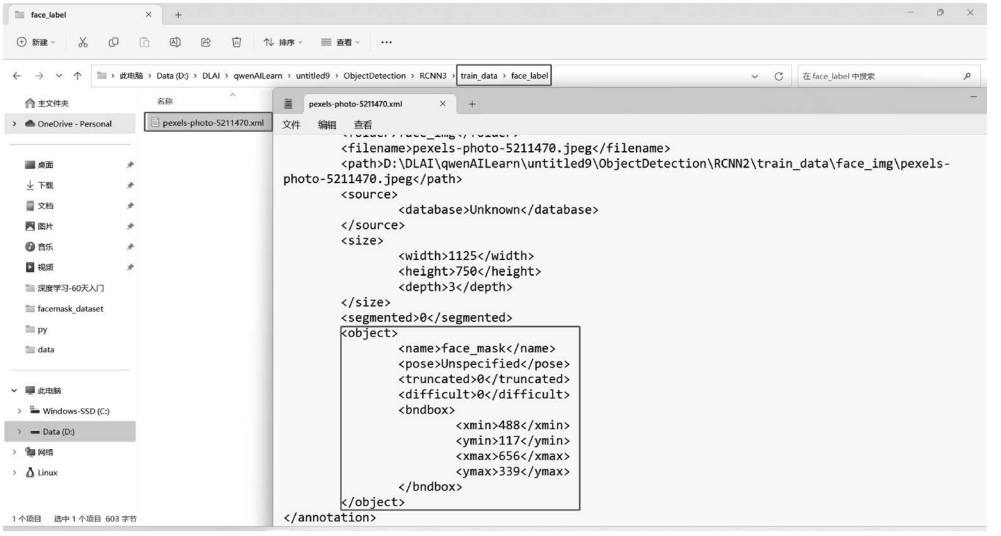



图 5-3　PascalVoc 标注内容

如果有多个感兴趣的目标和图像，就需要逐张图片对感兴趣的区域进行标注，这个过程是比较耗费时间和体力的。

如果选择 YOLO 格式，则会在标注保存文件夹中生成一个 TXT 文件，其中第 1 位为感兴趣的物体类别的下标，假设类别为[face_mask，face]则此时为 0，第 2 位和第 3 位为矩形框的中心点位置(center_x，center_y)，第 4 位和第 5 位为矩形框的宽和高(width，height)，然后针对(center_x，width)/原图的宽、(center_y，height)/原图的高，得到如图 5-4 所示的小数。

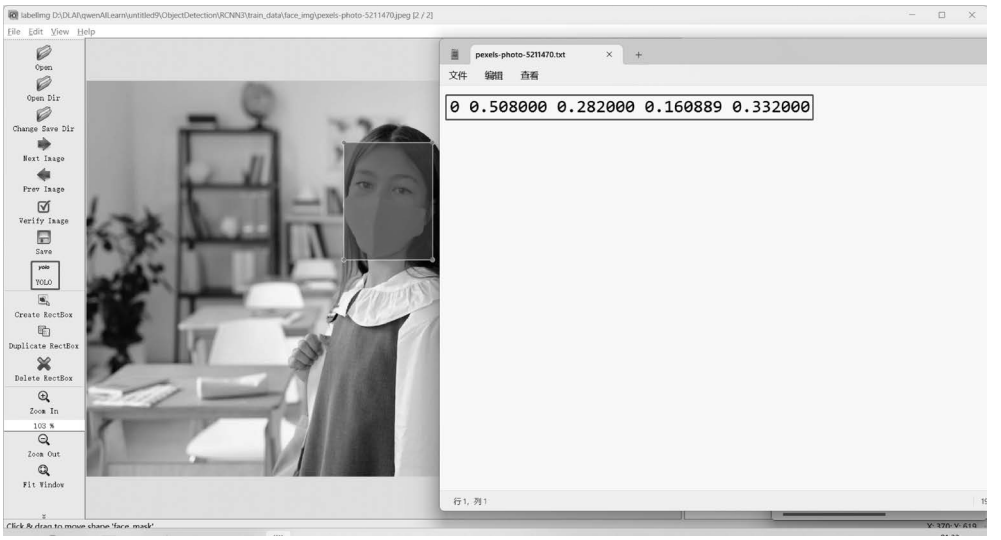



图 5-4 YOLO 标注内容

使用代码可以由 Voc 格式转换成 YOLO 格式，代码如下：

```
#第 5 章/ObjectDetection/LabelConversion/voc2yolo.py
from glob import glob
import os
import xml.etree.cElementTree as ET
import cv2
import numpy as np

def draw_box(image, boxes):
    #在原图中绘出所有的 boxes 和 label
    for i, rect in enumerate(boxes):
        #获得 box
        index, x1, y1, x2, y2 = rect[0:5].astype("int")
        #绘矩形框，需要左上、右下坐标
        cv2.rectangle(image, (x1, y1), (x2, y2), (0, 255, 0), 1, cv2.LINE_AA)
        #绘矩形框的类别
        cv2.putText(image, f"{index}", (x1 - 10, y1 - 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.75, (0, 0, 255), 2)
    cv2.imshow('show box', image)
    cv2.waitKey(0)

def read_annotations(xml_path, all_name=[], image=None):
    #使用 etree 读取 XML 文件
```

```
    et = ET.parse(xml_path)
    element = et.getroot()
    #查找 XML 文件中所有的 object 目标区域
    element_objs = element.findall('object')
    #用来存储矩形框的区域
    results = []
    #遍历所有的 box 区域的 object 标签
    for element_obj in element_objs:
        #获得类名称
        class_name = element_obj.find('name').text
        #如果 XML 文件中类的名称与指定类的名称相同,则获取列表中类的索引值
        for i, n in enumerate(all_name):
            if n == class_name:
                index = i
                break
        #从 bndbox 中获取矩形框的左上、右下坐标并转换成 int 类型
        obj_bbox = element_obj.find('bndbox')
        #lambda 表达式,根据坐标名称获取对应内容,并转换成 float 型
        coord_xy = lambda coord_name: float(obj_bbox.find(coord_name).text)
        #组成[xmin,ymin,xmax,ymax,label index]的数组
        label = [index] + [coord_xy(name) for name in ['xmin', 'ymin', 'xmax', 'ymax']]
        results.append(label)
    #如果 image 不为 None,则可以将获取出来的 label 还原到图像中
    boxes = np.array(results)
    if image is not None:
        draw_box(image, boxes)
    return boxes

def xyxy2cxcywh(boxes, image_shape):
    #由 xmin、ymin、xmax、ymax 分别转换成 cx、cy、w、h
    #width,height = (xmax,ymax)-(xmin,ymin) 得到矩形框的 width 和 height
    boxes[..., 3:5] = boxes[..., 3:5] - boxes[..., 1:3]
    #(center_x,center_y) = (xmin,ymin + wh/2) 中心点在图像中的位置
    boxes[..., 1:3] = boxes[..., 1:3] + boxes[..., 3:5] / 2
    #分别除以 width 和 height 得到归一化后的值
    boxes[..., 1:5] = boxes[..., 1:5] / image_shape
    return boxes

if __name__ == "__main__":
    #根目录
    root = "../train_data"
    #训练图片文件夹
    jpg_path = f"{root}/face_img"
    #标注过的文件夹
    xml_path = f"{root}/face_label"
    #期望转换为 YOLO 格式的文件夹
    save_path = f"{root}/yolo_label"
    #如果指定的保存文件夹不存在,则创建 1 个
    if not os.path.exists(save_path): os.mkdir(save_path)
    #获取指定文件夹中所有以.jpeg 为后缀名的文件
    images_file = glob(os.path.join(jpg_path, '*.jpeg'))
    #遍历所有的图片
```

```
    for image_path in images_file:
        #读取图片以获取图片的 width 和 height
        #不从 XML 文件读取,这是因为有时与实际图片不一致
        image = cv2.imread(image_path)
        image_shape = image.shape[0:2][::-1]
        #组成 width,height,width,height 的数组,方便后续计算
        image_shape = np.array(image_shape + image_shape)
        #根据图片名称找到对应的 Voc XML 文件
        jpg_name = os.path.split(image_path)
        name = jpg_name[-1].split('.')[0]
        name_xml = f"{xml_path}/{name}.xml"
        #获取原始标注类别和位置[index,xmin,ymin,xmax,ymax]
        boxes = read_annotations(name_xml, all_name=['face_mask', 'face'], image=
image)
        #实现从 Voc 格式转换成 YOLO 格式
        boxes = xyxy2cxcywh(boxes, image_shape)
        #设置保存 YOLO 格式的 TXT 文件名
        save_txt = f"{save_path}/{name}.txt"
        #将 box 信息写入 save_path 中
        np.savetxt(save_txt, boxes, fmt='%.4f')
```

代码中 read_annotations()函数实现读取 XML 文件得到[index,xmin,ymin,xmax,ymax]的数组内容,并在 xyxy2cxcywh()函数中实现转换成[index,center_x,center_y,width,height]的数组。draw_box()函数主要将标注 box 还原到图像中,以便进行抽查标注信息是否正确。np.savetxt()实现整个数组保存,由于保存为 float 类型,TXT 文件中第 1 位 index 会显示为 4 位小数,这个 index 在喂入训练数据时需要转换为 int。

也可以由 YOLO 格式转换成 Voc 格式,代码如下:

```
#第 5 章/ObjectDetection/LabelConversion/YOLOv2voc.py
from glob import glob
import os
import cv2
import numpy as np

def cxywh2xyxy(boxes, image_shape):
    #实现从矩形中心点坐标转换成 xmin,ymin,xmax,ymax
    #从小数还原为图像中的位置
    box = boxes[..., 1:5].copy() *image_shape
    #(center_x,center_y) - (width,height/2)就是左上角
    xminymin = box[..., 0:2] - box[..., 2:4] / 2
    #(center_x,center_y) + (width,height/2)就是右下角
    xmaxymax = box[..., 0:2] + box[..., 2:4] / 2
    #合成[xmin,ymin,xmax,ymax]
    boxes[..., 1:5] = np.concatenate([xminymin, xmaxymax], axis=-1)
    #转换为 int 类型
    return boxes.astype('int32')

def generate_voc(boxes, all_name=[]):
    #根据 boxes 信息生成 Voc 格式的文件
    voc_str = ""
```

```
    for i, rect in enumerate(boxes.astype("int32")):
        object_str = f"""
            <object>
                <name>{all_name[0]}</name>
                <pose>Unspecified</pose>
                <truncated>0</truncated>
                <difficult>0</difficult>
                <bndbox>
                    <xmin>{rect[1]}</xmin>
                    <ymin>{rect[2]}</ymin>
                    <xmax>{rect[3]}</xmax>
                    <ymax>{rect[4]}</ymax>
                </bndbox>
            </object>
        """
        voc_str += object_str
    return f"""<annotation>{voc_str}</annotation>"""

if __name__ == "__main__":
    #根目录
    root = "../train_data"
    #训练图片文件夹
    jpg_path = f"{root}/face_img"
    #YOLO 格式的文件夹
    yolo_path = f"{root}/yolo_label"
    #期望保存为 Voc 格式的文件夹
    save_path = f"{root}/voc_label"
    #如果指定的文件夹不存在,则创建 1 个
    if not os.path.exists(save_path): os.mkdir(save_path)
    #获取指定文件夹中所有以 .jpeg 为后缀名的文件
    images_file = glob(os.path.join(jpg_path, '*.jpeg'))
    #遍历所有的图片
    for image_path in images_file:
        #读取图片以获取图片的 width 和 height
        #不从 XML 文件读取,这是因为有时与实际图片不一致
        image = cv2.imread(image_path)
        image_shape = image.shape[0:2][::-1]
        #组成 width,height,width,height 的数组,方便后续计算
        image_shape = np.array(image_shape + image_shape)
        #根据图片名称找到对应的 Voc XML 文件
        jpg_name = os.path.split(image_path)
        name = jpg_name[-1].split('.')[0]
        name_txt = f"{yolo_path}/{name}.txt"
        #读取 YOLO 格式内容
        boxes = np.loadtxt(name_txt)
        #转换成[index,xmin,ymin,xmax,ymax]
        boxes_xyxy = cxywh2xyxy(boxes, image_shape)
        #创建 XML 文件
        save_xml = f"{save_path}/{name}.xml"
                #调用 generate_voc()实现 Voc 格式的写入
        with open(save_xml, 'w', encoding='utf-8') as f:
            f.write(generate_voc(boxes_xyxy, all_name=['face_mask', 'face']))
        #将 YOLO 格式还原到图像中
        draw_box(image, boxes_xyxy)
```

cxywh2xyxy()实现从 YOLO 格式转换为 xmin，ymin，xmax，ymax 的格式。同时 generate_voc()实现 XML 文件的写入，np.loadtxt()实现 TXT 文件的数组读入。

标注文件内容格式的转换为高频操作，本书的代码可以在工作中直接使用。

**总结**

目标检测标注工具 labeling 可标注 YOLO 格式、Voc 格式，并且实现了 YOLO 格式与 Voc 格式的互相转换。

**练习**

寻找一份数据集进行标注，并调试实现 YOLO 格式与 Voc 格式数据的互相转换。

## 5.2 开山之作 R-CNN

### 5.2.1 模型介绍

R-CNN 由 Ross Girshick 等在 2014 年发表的论文 *Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation* 中首次使用深度学习进行目标检测，其网络结构和实现过程对后续模型提供了重要参考。

其网络的主要过程如图 5-5 所示，分为训练过程和预测过程。

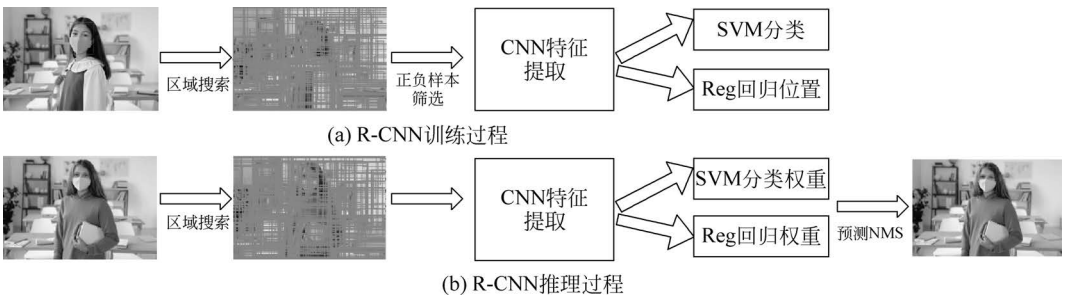



图 5-5 R-CNN 模型过程

(1) 训练时需要输入训练图片和标注 BOX，通过选择区域搜索算法生成多个目标框(取前 2000 个区域)，当区域选择框与标注 BOX 重叠面积较大时认为是有目标的框(正样本)，而重叠面积较小或无重叠时认为是无目标的框(称为负样本)。将正样本图片和负样本图片输入 CNN 中提取特征，例如使用 VGG 中的 FC 层得到 4096 维特征，然后使用 4096 维特征，通过支持向量机(SVM)进行分类训练。由于区域选择框与标注 BOX 之间有一定的偏移，所以这里使用回归算法对这个偏移进行了回归训练。

(2) 预测推理时需要输入未知图片，同样通过区域搜索算法选取 2000 个区域选择框，使用 CNN 提取特征，然后将区域选择框中的图像信息输入 SVM 权重得到预测分类，然后通过阈值设置过滤掉概率较低的区域选择框。将分类概率较高的框输入 Reg 回归权重得到区域选择框与预测框的偏移，通过区域选择框和预测偏移得到多个预测框。多个预测框

之间有可能出现重叠面积较大的框，这会被认为是同一个分类。使用非极大值抑制（NMS）对其进行剔除，得到最后的预测框，将预测框和预测分类信息绘制到原图中，就得到了目标检测的结果。

整个过程步骤较多且涉及较多新概念，详细过程和相关代码将在后续几节中逐步展开。

## 5.2.2　代码实战选择区域搜索

选择区域搜索（Selective Search）算法，首先输入一张图片，通过图像分割的方法获得很多小的区域，然后对这些小的区域采用相似度计算的方法，将相似区域不断地进行合并，一直到无法合并为止。图像相似度计算包括颜色相似度、纹理相似度、尺度相似度、填充相似度等方法，具体可参考原论文。

选择区域搜索算法的实现可调用 OpenCV 中的 createSelectiveSearchSegmentation() 对象实现，代码如下：

```
#第 5 章/ObjectDetection/R-CNN/selectivesearch.py
import cv2
import numpy as np

def start_search(image):
   #开启优化设置
   cv2.setUseOptimized(True)
   #创建选择优化搜索对象
   #pip install opencv-contrib-python
   objSrh = cv2.ximgproc.
segmentation.createSelectiveSearchSegmentation()
   #加载图片
   objSrh.setBaseImage(image)
   #快速搜索
   objSrh.switchToSelectiveSearchFast()
   #提取区域选择框
   objSrhRects = objSrh.process()
   #区域选择框由 x、y、w、h 分别转换成 xmin、ymin、xmax、ymax
   rect = objSrhRects.copy()
objSrhRects[..., 2:4] = rect[..., 0:2] + rect[..., 2:4]
#在原图绘出区域框
   #u.draw_box(image,objSrhRects)
   return objSrhRects[:2000, ...]
```

函数 start_search(image)用于实现选择区域搜索，但是 objSrhRects 提取的是左上角坐标和高宽，通过 objSrhRects[...,2:4]＝rect[...,0:2]＋rect[...,2:4]转换成左上、右下角坐标。

## 5.2.3　代码实战正负样本选择

区域选择框与标注 BOX 重叠面积占比的计算又称为交并比（Intersection over Union，IOU），其计算公式为

$$IOU=\frac{area_{intersection}}{area_{box1}+area_{box2}-area_{intersection}} \tag{5-1}$$

当交集区域最大时，GTBOX 与 SSBOX 会重叠，此时 IOU＝1；当交集区域为 0 时，IOU＝0；求交集则为 $area_{intersection}=abs(GTBOX_{maxx}-SSBOX_{minx})\times abs(GTBOX_{maxy}-SSBOX_{miny})$，加绝对值 abs()是由于 GTBOX 有可能在 SSBOX 的右侧，如图 5-6 所示。


图 5-6　IOU 计算

具体的代码如下：

```
#第 5 章/ObjectDetection/R-CNN/utils.py
import numpy as np

def get_iou(boxes1, boxes2):
    #比较哪个 box 在左边，哪个 box 在右边，此时不再需要 abs
    left = np.maximum(boxes1[..., :2], boxes2[..., :2])
    right = np.minimum(boxes1[..., 2:4], boxes2[..., 2:4])
    #计算交集的 wh
    intersection = np.maximum(0.0, right - left)
    #计算交集的面积
    area_inter = intersection[..., 0] * intersection[..., 1]
    #计算每个 box 的面积
    area_box1 = (boxes1[..., 2] - boxes1[..., 0]) * (boxes1[..., 3] - boxes1[..., 1])
    area_box2 = (boxes2[..., 2] - boxes2[..., 0]) * (boxes2[..., 3] - boxes2[..., 1])
    #计算并集的面积，1e-10 保证分母为 0
    union_square = np.maximum(area_box1 + area_box2 - area_inter, 1e-10)
    #计算交并比。如果比 0 小，则为 0；如果比 1 大，则为 1
    score = np.clip(area_inter / union_square, 0.0, 1.0)
    return score
```

当 IOU＞0.5 时，SSBOX 选取为正样本；当 IOU＜0.3 时，SSBOX 选取为负样本，代码如下：

```
#第 5 章/ObjectDetection/R-CNN/selectivesearch.py

#根据 IOU 的得分，选择正样本和负样本
```

```
def based_iou_sample(objSrhRects, iou_array, g_index, max_threshold, is_
greater=True, image=None):
    #如果 is_greater 为 True,则选择 IOU 大于指定分数的 box 作为正样本
    #如果 is_greater 为 False,则选择 IOU 小于指定分数的 box 作为负样本
    index = np.argwhere(iou_array > max_threshold if is_greater else iou_array <
max_threshold)
    if len(index) > 0:
        #根据 index 获取 ss 的 box 信息
        ss_box = objSrhRects[index].reshape([-1, 4])
        #在原图中绘出这个区域
        u.draw_box(image, ss_box, color=[0, 0, 255])
        #大于 0.5 的框被认为是真实框,然后将这个 label 复制 N 份
        #小于 0.5 的框被认为是负样本,然后将类别置为背景,没有目标
        true_label = np.tile(np.array(g_index), [len(ss_box), 1])
        #对大于 max_threshold 的框增加 label 信息
        ss_box = np.append(ss_box,true_label, axis=-1)
        #返回区域选择框,以及样本的下标
        return ss_box, index
```

函数 based_iou_sample()根据传入的 IOU 分数通过 np.argwhere()函数得到满足条件的索引号,当大于 0.5 时为正样本,当小于 0.3 时为负样本。通过参数 g_index 来区分是正样本还是负样本,如果是负样本,则 g_index 为所有类别数+1。

调试程序,可发现 IOU>0.5 的区域选择框共有 15 个,其中最大值为 0.98190,如图 5-7 所示。

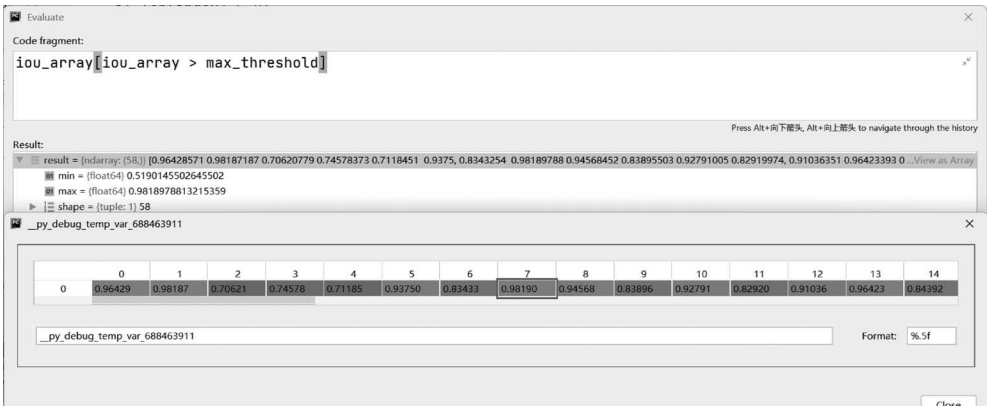



图 5-7　IOU 正样本分数

如果将 IOU>0.5 的区域选择框还原到图像中,则可发现较多框可覆盖标注 BOX。将这些区域选择框送入 CNN 提取特征,也就是正样本的特征,而那些 IOU<0.3 的区域选择框为负样本,CNN 提取的特征为负样本特征。选择负样本来进行训练的一个原因是为了让神经网络学习到某些参数,以便将非目标区域排除。

正样本区域选择框如图 5-8 所示,负样本区域选择框如图 5-9 所示,而中间区域的区域和特征信息将会被丢弃。

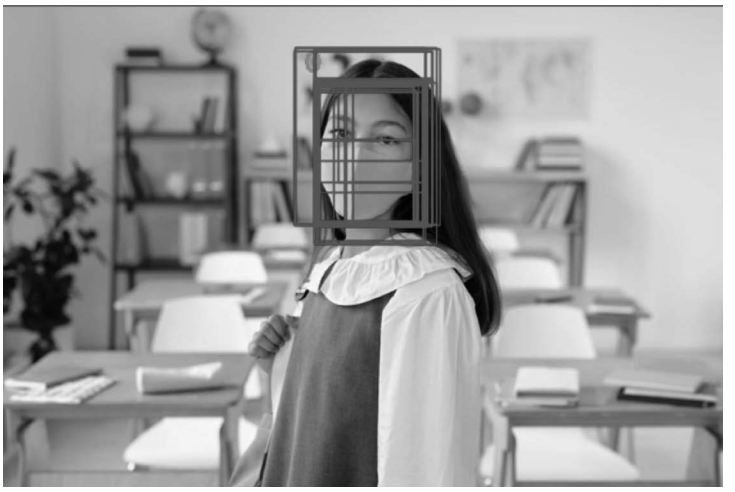


图 5-8　正样本区域选择框(IOU>0.5)

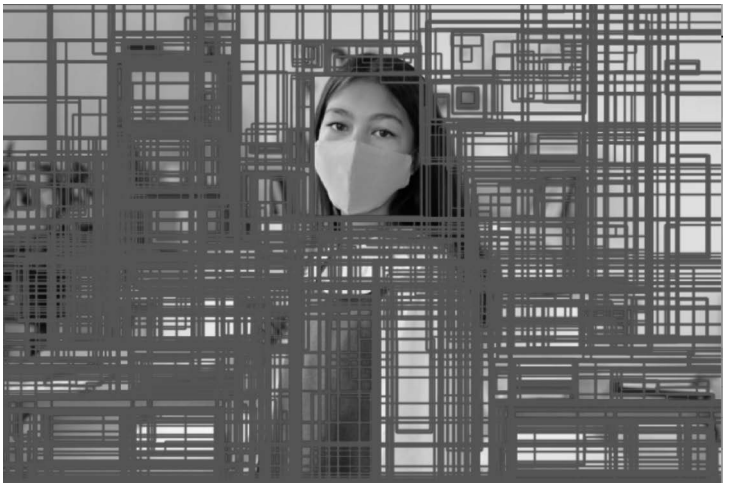


图 5-9　负样本区域选择框(IOU<0.01)

正负样本确定后,需要根据 SSBOX 信息进行切图,并保存到本地以进行 CNN 特征提取,代码如下:

```
#第 5 章/ObjectDetection/R-CNN/selectivesearch.py

#切割图像并转换到 224×224 的大小
def cut_image(image, boxes):
   #image[ymin:ymax, xmin:xmax] 切图的方法
   return [cv2.resize(image[box[0]:box[1], box[2]:box[3]], [224, 224]) for box
in boxes]

cut_positive_box = positive_box.copy()
#正样本的格式为 xmin,ymin,xmax,ymax,其下标分别为 0,1,2,3
```

```
#需要转换为 ymin,ymax,xmin,xmax,其下标分别为 1,3,0,2,并以这些值进行切图
cut_positive_box[..., [0, 1, 2, 3]] = cut_positive_box[..., [1, 3, 0, 2]]
positive_image = cut_image(img.copy(), cut_positive_box[..., 0:4])
```

因为区域选择框得到的格式为 xmin,ymin,xmax,ymax,而切图需要 ymin,ymax,xmin,xmax 格式,所以通过代码 cut_positive_box[...,[0,1,2,3]]=cut_positive_box[...,[1,3,0,2]]进行了转换。同时 cut_image()函数实现了切图,并转换到指定图像的大小。将图像大小设置为[224,224]是由于 VGG 的输入要求,如图 5-10 所示。

图 5-10　切图后将大小转换到 224×224

标注信息使用 5.1 节标签处理及代码中的函数 read_annotations()进行 true_box 的读取,同时随书源码中 ObjectDetection/R-CNN/selectivesearch. p 文件中的 get_positive_negative_samples()函数实现了更完整的代码。

另外在读取图片时将输入图像和标注 BOX 进行了等比例缩放。R-CNN 由于采用的是区域选择框提取,可以不进行图像的大小转换,但是为了网络的训练学习需要统一尺度及提高稳定性,本书将图像大小等比例缩放到 640×640,代码如下:

```
#第 5 章/ObjectDetection/R-CNN/utils.py
def letterbox_image(image, size, box):
    #对图片大小进行转换,使图片不失真。在空缺的地方进行 padding
    #位置不会有偏移的情况
    iw, ih = image.size
    w, h = size
    #如果刚好输入的尺寸与要求的尺寸相同,则原图返回
    if iw == w and ih == h:
        return cv2.cvtColor(np.array(image), cv2.COLOR_RGB2BGR), box
    #计算是宽还是高的比例更小
    scale = min(w / iw, h / ih)
    #按最小的比例进行缩放
    nw = int(iw * scale)
    nh = int(ih * scale)
    #缩放比例距原图的大小,因为需要上下或者左右同时缩放,所以需要整除 2
    dw = (w - nw) //2
    dh = (h - nh) //2
    #等比例缩放
    image = image.resize((nw, nh), Image.BICUBIC)
    #新建一个灰度图
    new_image = Image.new('RGB', size, (128, 128, 128))
    #在缩放的图中增加填充的像素
    new_image.paste(image, (dw, dh))
```

```
        #对 BOX 进行等比例调整
        box_resize = []
        for boxx in box:
                #对 BOX 进行等比例缩减,并加上填充的灰度图
            boxx[0] = int(boxx[0] * scale + dw)
            boxx[1] = int(boxx[1] * scale + dh)
            boxx[2] = int(boxx[2] * scale + dw)
            boxx[3] = int(boxx[3] * scale + dh)
            #略有裁剪,可不使用
            boxx[0] = np.clip(boxx[0], 0, w - 1)
            boxx[2] = np.clip(boxx[2], 0, h - 1)
            boxx[1] = np.clip(boxx[1], 0, w - 1)
            boxx[3] = np.clip(boxx[3], 0, h - 1)
            boxx = boxx.astype("int32")
            box_resize.append(boxx)
        #转回 NumPy 格式
        new_image = cv2.cvtColor(np.array(new_image), cv2.COLOR_RGB2BGR)
    return new_image, np.array(box_resize)
```

函数 letterbox_image()会对输入图像进行等比例缩放,例如原图为 1125×750,按 640×640 进行缩放,则 scale=min(0.568,0.853)按 0.568 进行等比例缩放,而此时(nw,nh)=(640,426),缩放后图像的高不够 640,则需要上下各补 107 像素,即 new_image.paste(image,(0,107))。同理原图输入的标注 BOX 信息,也按 0.568 进行缩放,所以 boxx[1]×scale,但是可能像素不够,所以通过 boxx[1]×scale+107 进行填充,效果如图 5-11 所示。

图 5-11 等比例缩放图像和标注 BOX

### 5.2.4　代码实战特征提取

5.2.3节已提取正样本和负样本图片、正样本SSBOX和标注框GTBOX信息，搭建CNN中的VGG-16网络并使用ImageNet的初始权重，代码如下：

```
#第5章/ObjectDetection/R-CNN/vgg_features.py
def init_vgg():
    #VGG-16模型，从Keras中直接提取
    base_model = VGG-16(weights='ImageNet',include_top=True)
    base_model.summary()
    #构建模型，output参数用于确定使用哪一个网络层作为输出
    model = Model(inputs=base_model.input, outputs=base_model.layers["fc2"].
output)
    return model

#从ImageNet VGG-16中得到图片的特征信息
def vgg_features(model, image):
    x = preprocess_input(image)
    #前向传播得到特征
    features = model.predict(x)
    return features
if __name__ == "__main__":
    model = init_vgg()
    get_feature_map("./train_data/ss_info/正样本图片.npy", "./train_data/ss_
feature/正样本图片特征.npy", model)
    get_feature_map("./train_data/ss_info/负样本图片.npy", "./train_data/ss_
feature/负样本图片特征.npy", model)
```

调用keras. applications模块下已写好的VGG-16模型，并同时使用weights='ImageNet'的初始权重，选择fc2层4096维作为输出特征，然后分别加载正样本图片.py和负样本图片.npy，并保存特征信息。

### 5.2.5　代码实战SVM分类训练

支持向量机(SVM)是一个经典的机器学习分类方法，如图5-12中(a)图红色和蓝色的点显然是可以被一条直线分开的，而能够分开的线不止一条，例如(b)、(c)中的黑线$A$和$B$，如果是一个平面，则称为决策面。

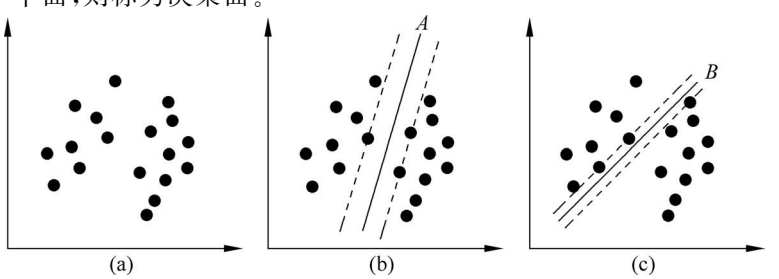



图5-12　最大间隔距离(见彩插)

如果从决策面来观察，则可知 $A$ 和 $B$ 均可，但是从直觉来讲 $A$ 优于 $B$，其原因是 $A$ 的间隔距离比 $B$ 大，所以支持向量机的核心思想就是求解能够正确划分训练数据集并且几何间隔距离最大的超平面。

支持向量机的原理推导较复杂(非重点)，不过调用 sklearn 库中的 SVC()对象可轻松实现，代码如下：

```
#第 5 章/ObjectDetection/R-CNN/train.py
#SVM 训练分类器
def svm_classifier():
    #加载特征提取信息和类别
    pos_X = np.load("./train_data/ss_feature/正样本图片特征.npy")
    pos_Y = np.load("./train_data/ss_info/正样本图片分类 label.npy")
    neg_X = np.load("./train_data/ss_feature/负样本图片特征.npy")
    neg_Y = np.load("./train_data/ss_info/负样本图片分类 label.npy")
    #合并正样本和负样本
    X = np.concatenate([pos_X, neg_X], axis=0)
    #label 信息
    Y = np.concatenate([pos_Y, neg_Y], axis=0)
    #划分训练集和验证集
    x_train, x_test, y_train, y_test = train_test_split(X, Y, random_state=14)
    #SVM 线性分类器
    clf = SVC(C=1.0, kernel='linear', random_state=28, max_iter=1000,
probability=True)
    #训练
    clf.fit(x_train, y_train)
    #验证评分
    pred = clf.predict(x_test)
    print("F1-score: {0:.2f}".format(f1_score(pred, y_test, average='micro')))
    #保存 SVM 模型和参数
    joblib.dump(clf, "face_mask_svm.m")
```

对象 SVC()为一个线性可分对象，其中 C 为惩罚系数，训练完成后使用 joblib.dump()保存模型。

论文的作者在这里使用 SVM 而不是用分类网络的重要原因是由于此时只有少量的正样本，但有大量的负样本，样本数据不平衡对于 CNN 来讲会非常敏感，而 SVM 对于不平衡数据并不敏感。

### 5.2.6 代码实战边界框回归训练

通过 SVM 对区域选择框的背景或者目标类别进行预测，此时正样本区域选择框与标注真实框会有一定的偏差，通过以下公式进行偏移值的计算：

$$
\begin{aligned}
t_x &= (G_x - P_x)/P_w \\
t_y &= (G_y - P_y)/P_h \\
t_w &= \log\left(\frac{G_w}{P_w}\right)
\end{aligned}
\tag{5-2}
$$

$$t_h = \log\left(\frac{G_h}{P_h}\right)$$

其中，$G_x$ 和 $G_y$ 表示标注 GTBOX 的中心点，$P_x$ 和 $P_y$ 为区域选择框 SSBOX 的中心点；$G_x$ 和 $G_y$ 为 GTBOX 的宽和高，$P_w$ 和 $P_h$ 为 SSBOX 的宽和高；$t_x$ 和 $t_y$ 为 GTBOX 中心点与 SSBOX 中心点的偏移，$t_w$ 和 $t_h$ 为 GTBOX 与 SSBOX 的宽和高的偏移。当然 $t_x$、$t_y$、$t_w$ 和 $t_h$ 越小越好，说明 GTBOX 与 SSBOX 的 IOU 分数较高，如图 5-13 所示。


图 5-13　标注 GTBOX 和 SSBOX

计算偏移值的代码如下：

```
#第 5 章/ObjectDetection/R-CNN/train.py
def xyxy2cxcywh(boxes):
    #由 xmin,ymin,xmax,ymax 分别转换成 xmin,ymin,w,h
    #由 xmax-xmin 和 ymax-ymin 得到中心点坐标
    wh = boxes[..., 2:4] - boxes[..., :2]
    center_xy = boxes[..., :2] + wh / 2
    #合并数组
    return np.concatenate([center_xy, wh], axis=-1)

#真实框 cx,cy,w,h。区域选择框 p_cx,p_cy,w,h
def cxywh2offset(g_cxywh_boxes, p_cxywh_boxes):
    #公式(5-2)
    t_xy = (g_cxywh_boxes[..., :2] - p_cxywh_boxes[..., :2]) / p_cxywh_boxes[..., 2:4]
    t_wh = np.log(g_cxywh_boxes[..., 2:4] / p_cxywh_boxes[..., 2:4])
    #合并数组
    return np.concatenate([t_xy, t_wh], axis=-1)

def box_regression(num_class=1):
    #导入正样本的特征信息和 BOX 信息
    p_box = np.load("./train_data/ss_info/正样本 box.npy")
    p_feature = np.load("./train_data/ss_feature/正样本图片特征.npy")
```

```
    g_box = np.load("./train_data/ss_info/真样本 box.npy")
    for i in range(0, num_class):
        #类别号的下标
        index = np.where(p_box[..., -1] == i)
        #根据类别号取不同的特征信息和 box 信息并进行归一化操作
        p_class_feature = p_feature[index]
        #区域选择框,图像输入已统一到 640
        p_class_box = p_box[index] / 640
        #标注 GTBOX
        g_class_box = g_box[index] / 640
        #因为区域选择框和真实框得到的坐标是 xmin,ymin,xmax,ymax
        #为了计算偏移值需要转换成 cx,cy,w,h
        p_class_box = xyxy2cxcywh(p_class_box)
        g_class_box = xyxy2cxcywh(g_class_box)
        #计算偏移值
        offset_box = u.cxywh2offset(g_class_box, p_class_box)
        #输入 4096 维特征,输出 4 位偏移值
        x_train, x_test, y_train, y_test = train_test_split(p_class_feature,
offset_box, random_state=14)
        #sklearn 中的线性回归
        model = LinearRegression()
        #Reg 回归训练
        model.fit(x_train, y_train)
        #预测
        y_pre = model.predict(x_test)
        #计算预测值和真实值之间的平均误差
        loss = mean_squared_error(y_pre, y_test)
        print("误差", loss)
        joblib.dump(model, f"face_mask_lr{i}.m")
        return p_class_feature, offset_box
```

由于 GTBOX 和 SSBOX 格式均为 xmin,ymin,xmax,ymax,但是计算偏移时需要 $cx$,$cy$,$w$,$h$,所以通过函数 xyxy2cxcywh(boxes)进行了实现。函数 cxywh2offset(g_cxywh_boxes,p_cxywh_boxes)传入 GTBOX 和 SSBOX 的信息,套用公式(5-2)实现 GTBOX 与正样本 SSBOX 的偏移量的计算。函数 box_regression(num_class=1)读取正样本特征并调用 cxywh2offset()生成偏移量的数据,以便喂入 Reg 回归网络训练。

LinearRegression()是 sklearn 库中的逻辑回归对象,输入是正样本 4096 维特征,$Y$ 值为 GTBOX 与 SSBOX 的偏移值数据 offset_box,当 model.predict(x_test)预测的偏移与 offset_box 最小时,说明回归权重系数已获得。回归损失使用均方差 mse()函数。

随书代码中还存在一份使用神经网络来做回归权重训练的代码,感兴趣的读者可查阅。

### 5.2.7 代码实战预测推理

根据训练结果得到分类权重文件 face_mask_svm.m,以及回归权重文件 face_mask_lr0.m,读入待预测图片的大小转换到训练所需的 640×640 大小,并调用区域选择搜索算法对预测图片生成 2000 个区域框,然后使用 VGG 提取特征输入分类权重,对这 2000 个区域图片的背景和低概率分类进行过滤,此时得到有预测目标的 SSBOX,代码如下:

```
#第 5 章/ObjectDetection/R-CNN/predictdetect.py
def inference_code(num_class=1,max_threshold=0.5):
    #VGG 特征提取
    model = v.init_vgg()
    #读入推理图片
    org_image = cv2.imread(
        r"../val_data/pexels-photo-5211438.jpeg")
    #由 NumPy 格式转换为 PIL 对象
    old_image = Image.fromarray(np.uint8(org_image.copy()))
    #等比例到 640×640
    img, _ = u.letterbox_image(old_image, [640, 640], [])
    #区域选择,默认得到的是 xmin,ymin,xmax,ymax
    p_box = s.start_search(img.copy())[:2000,...]
    #根据区域选择切图
    cut_box = p_box[..., 0:4].copy()
    cut_box[..., [0, 1, 2, 3]] = cut_box[..., [1, 3, 0, 2]]
    p_img = s.cut_image(img.copy(), cut_box)
    #对切出来的图全部升一维,由原来的[w,h,c]合并成 b,w,h,c
    all_p_img = np.concatenate([np.expand_dims(p, axis=0) for p in p_img], axis=0)
    #提取特征
    features = v.vgg_features(model, all_p_img)
    #调 SVM 进行预测
    clf = joblib.load("./face_mask_svm.m")
    class_result = clf.predict_proba(features)
    #过滤置信度较低的结果
    for i in range(num_class):
        #根据分类的下标进行置信度的过滤,得到分类可能性大于 0.5 的下标
        index = np.where(class_result[..., i] >= max_threshold)
    #.....接下 1 段代码.....
```

在 index 处断点可知 0 为 face_mask 的分类,而 1 为背景的概率。类别概率大于 0.5 的区域选择框共有 17 个,其中最大概率为 0.96,最小概率为 0.03,index 得到满足阈值数组的下标,如图 5-14 所示。

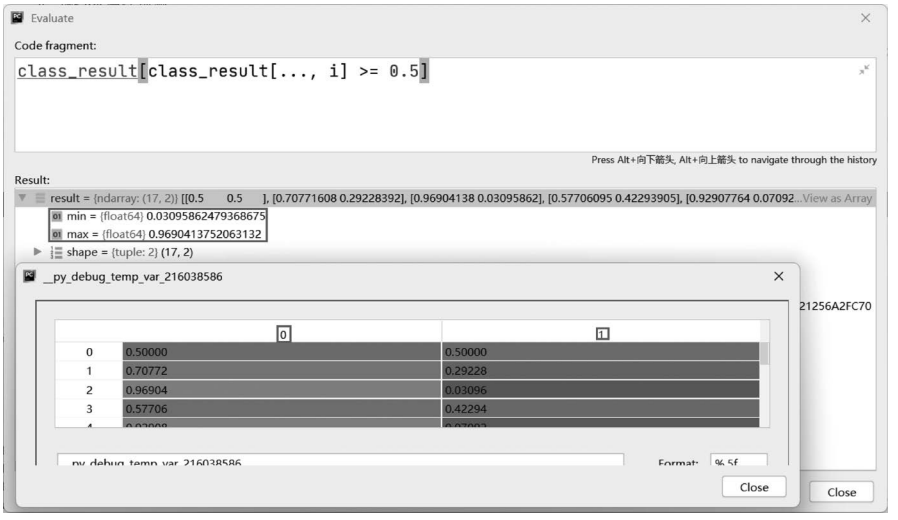



图 5-14　大于类别阈值的概率值

然后将满足分类概率的 SSBOX 和概率分数进行非极大值抑制(NMS)。NMS 先对所有分类概率进行降序排列,将最大概率的数组下标存储下来,然后将最大概率的 BOX 与其他传入的 boxes 进行 IOU 得分,如果有小于阈值的框,则认为是同类别的其他目标,否则剔除,然后对找出来的框再次进行 NMS,直到所有的框都不重叠,并返回 boxes 的下标,代码如下:

```
#第 5 章/ObjectDetection/R-CNN/predictdetect.py
def nms(boxes, nms_thresh=0.2):
    #用非极大值抑制,去掉相似框
    #当前框的分类得分
    scores = boxes[:, 4]
    #对 boxes 的概率得分从大到小进行排序,得到数组下标
    order = scores.argsort()[::-1]
    keep = []
    while order.size > 0:
        #取最大概率的下标
        i = order[0]
        #将最大概率 BOX 的下标存起来
        keep.append(i)
        #计算当前最大概率的 BOX 与其他框 boxes 的 IOU 分数
        iou_score = get_iou(boxes[i, ...], boxes[order[1:], ...])
        #只有小于指定阈值的框才会被认为是不同的框
        index = np.where(iou_score <= nms_thresh)[0]
        #将上一步的 index 存起来
        order = order[index + 1]
    return keep
```

非极大值之后使用 p_boxes=p_boxes[nms_index]获得区域选择框,调用 u.draw_box()函数实现目标区域的绘图,代码如下:

```
#第 5 章/ObjectDetection/R-CNN/predictdetect.py
def inference_code(num_class=1):
    #.....接上 1 段代码.....
    index = np.where(class_result[..., i] >= 0.5)
    if index[0].size:
        #得到 BOX,并归一化
        class_box = p_box[index] / 640
        #得到 BOX 当前类别的分数
        class_score = class_result[index][..., i]
        #将 BOX 和置信度合并成 n * 5
        p_boxes = np.column_stack([class_box, class_score])
        #进行非极大值抑制,得到保留 BOX 的下标
        nms_index = u.nms(p_boxes)
        #保留的区域选择 BOX
        p_boxes = p_boxes[nms_index]
        ss_img = u.draw_box(img.copy(), (p_boxes * 640).astype('int'))
```

直接使用区域选择框的位置也能识别出戴口罩的区域,如图 5-15 所示。

根据 SSBOX 与 Reg 回归器预测的偏移值进行微调以生成预测框,其公式如下:


图 5-15　区域选择框 NMS 后的目标区域

$$\begin{aligned}\hat{G}_x &= P_w d_x(P) + P_x \\ \hat{G}_y &= P_h d_y(P) + P_y \\ \hat{G}_w &= P_w \exp(d_w(P)) \\ \hat{G}_h &= P_h \exp(d_h(P))\end{aligned} \tag{5-3}$$

其中，$P_x$ 和 $P_y$ 为 SSBOX 的中心点坐标，$P_w$ 和 $P_h$ 为宽和高；$d_x(P)$和 $d_y(P)$为 Reg 回归器预测 SSBOX 与预测框中心点的偏移值，$d_w(P)$和 $d_h(P)$为宽和高的偏移。通过式(5-3)解码得到 $\hat{G}_x$、$\hat{G}_y$、$\hat{G}_w$、$\hat{G}_h$ 预测框的中心点、宽和高值。

根据偏移值和 SSBOX 得到预测框的代码实现，代码如下：

```
#第 5 章/ObjectDetection/R-CNN/predictdetect.py

def offset2xyxy(offset_box, p_cxywh_boxes):
    p_cxy = offset_box[..., :2] *p_cxywh_boxes[..., 2:4] + p_cxywh_boxes[..., :2]
    p_wh = np.exp(offset_box[..., 2:4]) *p_cxywh_boxes[..., 2:4]
    boxes = np.concatenate([p_cxy, p_wh], axis=-1)
    #转换成 xmin,ymin,xmax,ymax
    boxes[..., :2] -= boxes[..., 2:] / 2
    boxes[..., 2:] += boxes[..., :2]
    return boxes

def inference_code(num_class=1):
    #.....接上 1 段代码.....
    #调回归器权重进行微调
    lr = joblib.load(f"./face_mask_lr{i}.m")
    pre_offset = lr.predict(features[nms_index])
    #将区域选择框转换成 cx,cy,w,h
p_boxes = u.xyxy2cxcywh(p_boxes)
#解码,从偏移值和区域选择 BOX 转换成 xmin,ymin,xmax,ymax
```

```
        box = u.offset2xyxy(pre_offset, p_boxes)
        rg_img = u.draw_box(img.copy(), box)
        s_img = np.hstack([ss_img, rg_img])
        cv2.imshow('', s_img)
        cv2.waitKey(0)
```

函数 offset2xyxy(offset_box, p_cxywh_boxes)套用式(5-3)实现解码,同时将 boxes 信息转换为左上角、右下角的格式以方便绘图。在 inference_code()函数中,调用线性回归权重 face_mask_lr0.m,输入正样本的 4096 维特征信息,预测得到 pre_offset 的偏移值,然后调用 u.offset2xyxy(pre_offset, p_boxes)将偏移值转换成预测框,如图 5-16 所示。


图 5-16 根据预测偏移值获得的预测框

图 5-16 中的预测框比 SSBOX 要差一些,其原因是本训练集只有 1 张图片,训练数据泛化能力不够。

**总结**

R-CNN 使用区域选择搜索算法,训练过程存在多次文件存储,需要消耗大量计算时间和空间资源。使用 CNN 进行特征提取,偏移值 Reg 回归预测框,IOU 选择正样本,推理的非极大值抑制对于其他网络有极大的启发作用。

**练习**

运行并调试本节代码,理解算法思想与代码的结合。

## 5.3 两阶段网络 Faster R-CNN

### 5.3.1 模型介绍

Ross Girshick 等在 2016 年发表的论文 *Faster R-CNN: Towards Real-Time Object*

*Detection with Region Proposal Networks* 中提出 Faster R-CNN，在结构上 Faster R-CNN 将特征提取、区域提取、边界框 Reg 回归、分类整合在了一个网络中，使检测速度有极大的提高，其网络结构如图 5-17 所示。

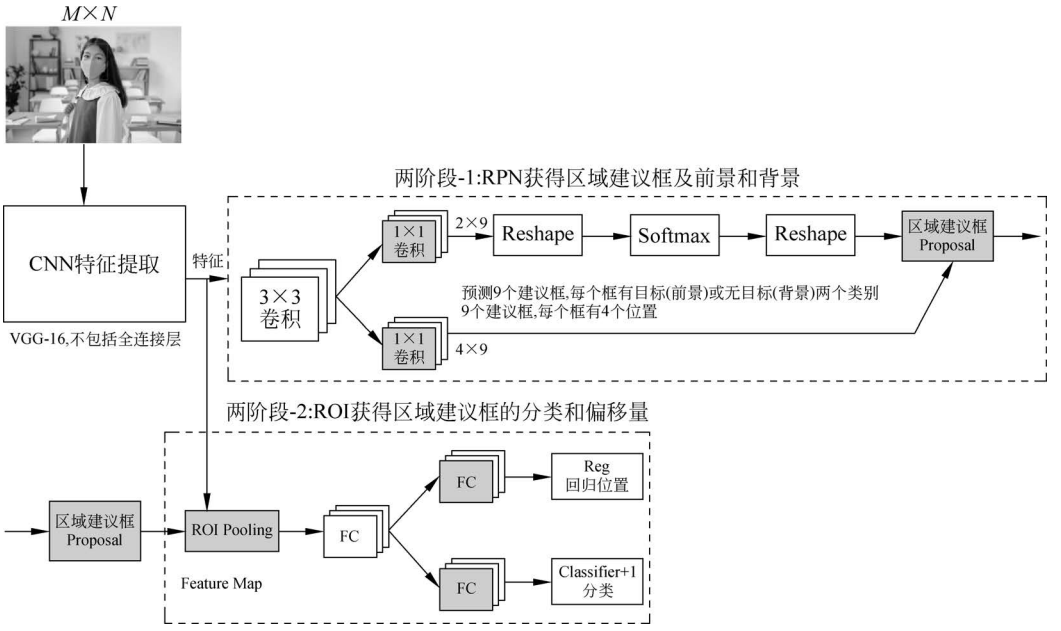



图 5-17　Faster R-CNN 网络结构

特征提取使用 VGG-16，其结构为 2-2-3-3-3-3（2 为卷积池化，最后的 3 为全连接），但不包括最后全连接层，使用 block5_conv3 层的输出作为 RPN、ROI 网络的输入特征。

Faster-R-CNN 为 2 阶段网络，第 1 个阶段为 RPN 网络，block5_conv3 输入后经过 3×3 卷积，分成两个分支，通过 1×1 卷积输出 2×9 个特征，每个特征有 9 个建议框，每个建议框有目标或者没有目标，所以类别数为 2。另一个 1×1 卷积输出 4×9 个特征，代表每个建议框的 4 个位置。RPN 的输出，9 个建议框的有目标、无目标的分类及 9 个建议框的位置。

Faster R-CNN 中的建议框相当于 R-CNN 中的区域搜索框，不同之处在于 R-CNN 是在原图中通过选择区域搜索算法生成，而 Faster R-CNN 通过预设矩形框尺寸和比例在 block5_conv3 上生成。CNN 提取的特征点相对于原图的感受野的区域如图 5-18 所示。

根据网络特征的提取，可以发现 block5_conv3 提取的特征刚好是原图的 1/16，因此可以看成在原图中对图像每隔 16 像素进行均分，并以每个均分点为中心点，以尺寸[8,16,32]和比例[0.5,1,2]生成 9 个建议框，如图 5-19 所示。


图 5-18　特征点在原图中的感受野

图 5-19 中绿色框为真实框，红色框为生成的建议框。图中红色点坐标之间的区域相对于

block5_conv3 特征点信息，以每个红色点的坐标为中心点生成 9 个建议框，假设 block5_conv3 得到的是 40×40，那么共计生成 14 400 个建议框，则 RPN 的输出为 14 400×2 个分类，14 400×4 个框。


图 5-19 Faster R-CNN 建议框的生成(见彩插)

将 14 400 个建议框与真实框之间做 IOU，当 IOU 大于设定值时，则选择此建议框作为正样本、有目标，然后按照式(5-2)计算建议框与真实框的偏移，作为真实值与预测值之间的损失(预测值为预测框基于建议框的偏移)；因为正样本少，负样本较多，选择当 IOU 小于设定值时，将此建议框作为负样本、无目标；将两个阈值之间的建议框丢弃。正负样本数量之和可以设定为 256，如图 5-20 所示，蓝色框为真实框，绿色框为建议框，两个建议框与真实框的 IOU=0.708，则此时有两个正样本，计算 offset 作为 $y$ 值。


图 5-20 Faster R-CNN 正样本的选择(见彩插)

RPN 的损失函数为

$$L(\{p_i\},\{t_i\})=\frac{1}{N_{\text{cls}}}\sum_i L_{\text{cls}}(p_i,p_i^*)+\lambda\frac{1}{N_{\text{reg}}}\sum_i p_i L_{\text{reg}}(t_i,t_i^*)$$

$$L_{\text{reg}}(t_i,t_i^*)=\sum_{i\in(x,y,w,h)}\text{smooth}_{L_1}(t_i-t_i^*) \tag{5-4}$$

$$\text{smooth}_{L_1}(t_i-t_i^*)=\begin{cases}0.5x^2 & \text{如果}\ |x|<1\\ |x|-0.5 & \text{其他}\end{cases}$$

其中，$i$ 表示建议框数组的索引号，$p_i$ 表示预测框有无目标的分类，$p_i^*$ 代表真实框有无目标的分类，分类损失使用交叉熵损失；$t_i$ 代表预测框相对于建议框的偏移，$t_i^*$ 代表真实框相对于建议框的偏移，位置损失使用 $\text{smooth}_{L_1}$ 损失。

回归损失可使用 $L_1$、$L_2$ 损失，$L_1$ 的导数为 $\frac{\partial_{L_1}(x)}{\partial x}=\begin{cases}1 & \text{如果}\ x\geqslant 0\\ -1 & \text{其他}\end{cases}$，$L_2$ 的导数为 $\frac{\partial_{L_2}(x)}{\partial x}=2x$，而 $\text{smooth}_{L_1}$ 的导数为 $\frac{\partial_{\text{smooth}_{L1}}(x)}{\partial x}=\begin{cases}x & \text{如果}\ |x|<1\\ \pm 1 & \text{其他}\end{cases}$，当 $x$ 增大时 $L_2$ 损失对 $x$ 的导数也增大，这就导致训练的初期，当预测值与真实值的差异过大时，损失函数对预测值的梯度较大，训练不稳定。当 $x$ 变小时，$L_1$ 对 $x$ 的导数为常数，在训练后期当预测值与真实值的差异较小时，$L_1$ 损失对预测值的导数的绝对值仍然为 1，如果此时学习率不变，则损失函数将在稳定值附近波动，难以继续收敛以达到更高精度，而 $\text{smooth}_{L_1}$ 损失则避免了 $L_2$、$L_1$ 的缺点，如图 5-21 所示。

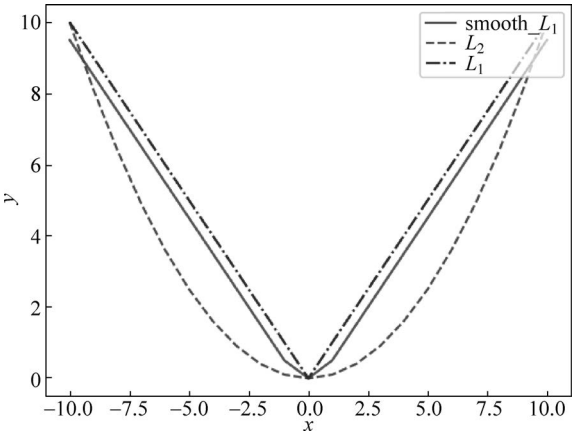



图 5-21　$L_1$、$L_2$、$\text{smooth}_{L_1}$ 损失

从图 5-21 中可知 $\text{smooth}_{L_1}$ 在远离坐标点时下降得非常快，类似 $L_1$，而离坐标点较近时又类似 $L_2$ 有一个转折的平滑效果，下降会慢一些。

第 2 阶段为 ROI 网络，ROI 网络的输入为 block5_conv3 的特征信息，以及 RPN 网络输出的预测建议框和有无目标的分类信息。图 5-17 中 Proposal 是 RPN 预测框信息，但是并没有全部输入 ROI 网络，而是将 RPN 输出信息解码成预测框，然后在预测框与建议框之

间根据有无目标的概率做非极大值抑制，选出 2000 个重叠率不高的框作为 Proposal，然后对 Proposal 的区域进行 ROI Pooling 以得到 7×7 的特征信息，输入全连接后再次微调 BOX 的位置信息，并预测目标区域是什么的分类信息，如图 5-22 所示。

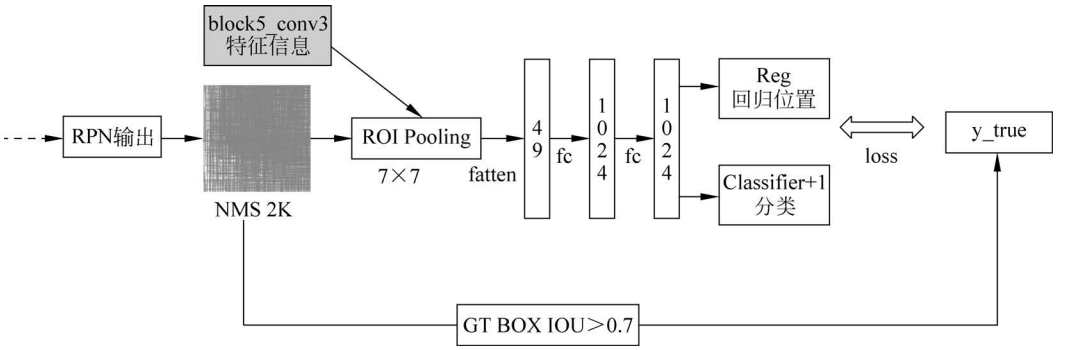



图 5-22 ROI 网络过程

图 5-22 中 y_true 为真实值，训练时需要将真实 BOX 的信息与选出来重叠率较低的 RPN 框做 IOU，当大于阈值时为正样本，当小于阈值时为负样本，正负样本之和可自定义设置，例如 128。

ROI Pooling 对输入的特征进行提取后形成一个固定的区域，Proposal 投影之后左上角的位置为[0,3]，右下角为[7,8]，然后划分成 2×2 的区域，对区域中的最大值进行提取，输出为 2×2 的特征，如图 5-23 所示。

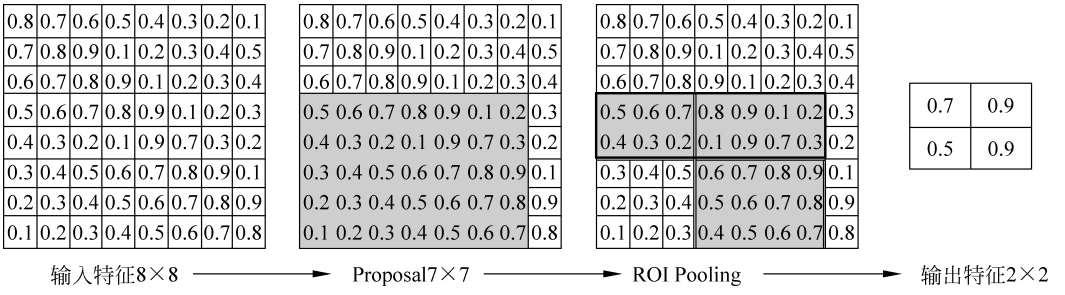



图 5-23 ROI Pooling 过程

因为 Faster R-CNN 要求的输入图像的尺度为 $M \times N$，原图映射到特征图的大小为 $[M/16, N/16]$，不同的图像输入的尺度 $M$、$N$ 值不同，但是通过 ROI Pooling 可以得到相同大小特征图的输出。因为其划分比例为 3∶4，同时也可以看到是不同尺度最大池化信息的融合，这对于分类或回归是有益的。

ROI Pooling 后进入全连接，并通过 Reg 回归再次精修位置，通过 Softmax 进行位置的回归，其损失函数同式(5-4)。

## 5.3.2 代码实战 RPN、ROI 模型搭建

根据图 5-17 中的结构描述，在 CNN 特征提取时使用 VGG-16 模型，同时使用 ImageNet

的权重信息作为初始权重。将 VGG-16 中 block5_conv3 的输出作为 RPN 网络的输入，得到建议框的位置 out_rpn_offset 和有无目标 out_rpn_clf，然后将 block5_conv3 的输出及 roi_input 的输入作为 ROI 网络的输入，输出分类预测和回归预测信息。

前向传播的代码如下：

```
#第 5 章/ObjectDetection/FasterR-CNN/model.py
import tensorflow as tf
from keras.applications.VGG-16 import VGG-16
from keras.layers import Input, Conv2D, Reshape, \
    Layer, Flatten, Dense, TimeDistributed
from keras import Model


def Faster R-CNN(input_shape=(640, 640, 3),
             roi_input_shape=(None, 4),
             num_anchors=9,
             num_class=2 + 1
             ):
    #图片输入 shape
    inputs = Input(shape=input_shape, name='image_input')
    #ROI Pooling 的输入维度
    roi_input = Input(shape=roi_input_shape, name='roi_input')

    #调用 Keras 自带的 VGG-16 作为 backbone，并且不要全连接层，并使用 ImageNet 的权重
    #假设输入为 600 × 600 × 3
    base_model = VGG-16(weights=None, include_top=False)
    base_model.load_weights(
        "../R-CNN/VGG-16_weights_tf_dim_ordering_tf_kernels.h5",
        by_name=True,
        skip_mismatch=True
    )
    #重新构建 backbone 网络，使用 VGG-16 的 block5_conv3 作为特征信息的输出
    base_model = Model(inputs=base_model.input, outputs=base_model.get_layer
('block5_conv3').output)
    #得到 block5_conv3 的特征
    backbone_feature = base_model(inputs)
    #得到 backbone 的输出(None,None,512)，作为 RPN 的输入
    #RPN 的作用：根据先验框，得到建议框
    out_rpn_clf, out_rpn_offset = rpn_proposal(backbone_feature, num_anchors)
    #RPN 模块的网络
    rpn_net = Model(inputs=inputs, outputs=[out_rpn_clf, out_rpn_offset])
    #第 2 个阶段，根据输入的 Feature Map 特征信息，对区域 BOX 进行 offset 微调，同时对
    #BOX 属于哪个类别进行预测
    out_class, out_regbox = roi_pooling_box_cls(backbone_feature, roi_input,
num_class)
    #预测时为整个网络：输入图片和 ROI 框的信息，以及输出类别和 Reg BOX 偏移值
    all_net = Model(inputs=[inputs, roi_input], outputs=[out_class, out_regbox])
    return rpn_net, all_net
```

RPN 网络的实现封装在函数 rpn_proposal(backbone_feature, num_anchors)中，代码如下：

```
#第 5 章/ObjectDetection/FasterR-CNN/model.py
def rpn_proposal(backbone_feature, num_anchors=9):
    #候选区域网络。在 R-CNN 区域选择算法的基础上进行改进,每个像素生成 9 个建议框,有目
    #标或无目标类别
    #RPN 以任意大小的图像作为输入,输出一组矩形的目标 Proposals
    #每个 Proposals 都有一个目标得分,即有目标还是没有目标 None *None *512
    x = Conv2D(512, (3, 3), padding='same', activation='relu',
            name='rpn_conv_3x3')(backbone_feature)
    #在 Feature Map 的基础上,每个特征点输出 9 个 Anchor,每个 Anchor 有 4 个位置
    #None × None × 36
    p_box_x = Conv2D(4 *num_anchors, (1, 1), padding='same',
                  activation='linear', name='rpn_box_conv_1x1')(x)
    #每个 BOX 分为有目标或者没有目标两个类别
    #None × None × 18
    p_conf_x = Conv2D(2 *num_anchors, (1, 1), padding='same',
                   activation='softmax', name='rpn_cnf_conv_1x1')(x)
    #(h×w×2×num_anchors,1),即每个像素只有一个概率,则有无目标
    out_rpn_clf = Reshape((-1, 1), name="rpn_p_conf")(p_conf_x)
    #(h×w×num_anchors,4),即每个像素有 9 个 Anchor,每个 Anchor 有 4 个位置
    out_rpn_offset = Reshape((-1, 4), name="rpn_p_box")(p_box_x)
    return out_rpn_clf, out_rpn_offset
```

VGG-16 网络层 block5_conv3 的特征信息,经过 3×3 的卷积后,分别经过 1×1 的 p_box_x 位置预测,1×1 的 p_conf_x 有无目标 Softmax 的预测。假设输入为 640×640,则 out_rpn_clf 成[40×40×2×9,1]=[28800,1],out_rpn_offset 为[40×40×9,4]=[14400,4]。

ROI 网络封装在 roi_pooling_box_cls(backbone_feature,roi_input,num_class)函数中,输入为 block5_conv3 的特征信息及 RPN 网络预测的 Proposal,将其值赋给 roi_input 变量,经过全连接后输出分类 out_class 和 out_regbox 位置信息,代码如下:

```
#第 5 章/ObjectDetection/FasterR-CNN/model.py
def roi_pooling_box_cls(feature_map_input, roi_input, num_class=2 + 1):
    #实现 roi_pool 的过程
    roi_pool = RoiPooling()([feature_map_input, roi_input])
    #根据 roi_pool 使用两个全连接层,分别输出 BOX 的偏移和类别的概率
    #不加 TimeDistributed 得到的是 None,None。加了之后得到的是 None,None,25088
    #TimeDistributed 实现了在每个 num_rois 上面进行一个全连接操作,实现多对多的功能
    x = TimeDistributed(Flatten(name='flatten'))(roi_pool)
    #batch_size,h,w,1024
    x = Dense(1024, activation='relu', name='fc1')(x)
    x = Dense(1024, activation='relu', name='fc2')(x)
    #分类的概率
    out_class = Dense(num_class, activation='softmax')(x)
    #每个类别的 4 个位置
    out_regbox = Dense(4 *(num_class - 1), activation='linear')(x)
    return [out_class, out_regbox]
```

out_class 中包含“分类+有无目标”的概率,假设类别为 2,Proposal 为 2000 个,则 out_class 输出是 2000×3,out_regbox 为每个类别的 BOX 信息,输出为 2000×8。

ROI Pooling 在 RoiPooling()([feature_map_input, roi_input])中实现,主要调用

tf.image.crop_and_resize()函数实现 7×7 区域的 Pooling,代码如下:

```
#第 5 章/ObjectDetection/FasterR-CNN/model.py
class RoiPooling(Layer):
   #Region of Interest,将不同大小的特征图 ROI Pooling 到同一个尺寸
   def __init__(self):
      super(RoiPooling, self).__init__()
      self.pool_size = (7, 7)

   def build(self, input_shape):
      self.out_channel = input_shape[0][3]

   def compute_output_shape(self, input_shape):
      input_shape2 = input_shape[1]
      return None, input_shape2[1], self.pool_size[0], self.pool_size[1],
self.out_channel

   def call(self, inputs, *args, **kwargs):
      #输入 block5_conv3 特征图及 RPN
      #40×40×512, 2000×4
      feature_map, roi_x = inputs
      #区域选择框的数量 batch,2k
      batch_size, num_roi = tf.shape(feature_map)[0], tf.shape(roi_x)[1]
      #假设 batch_size=2,则[0,1]-->[[0],[1]]
      index = tf.expand_dims(tf.range(0, batch_size), 1)
      #假设 num_roi=2,则[[0],[1]]-->[[0,0],[1,1]]
      index = tf.tile(index, (1, num_roi))
      #由[[0,0],[1,1]]变回[0,0,1,1]
      index = tf.reshape(index, [-1])
      #2000×7×7×512
      roi_pooling_feature = tf.image.crop_and_resize(
         feature_map,                    #特征图
         tf.reshape(roi_x, [-1, 4]),     #BOX 的个数,每个有 4 个位置信息
         index,                          #roi_x 与特征图下标的对应关系
         self.pool_size                  #裁剪到特征图的大小
      )
      #batch_size × 2000 × 7 × 7 × 512
      output = tf.reshape(
         roi_pooling_feature,
         (batch_size, num_roi, self.pool_size[0], self.pool_size[1], self.out_
channel)
      )
      return output
```

如代码输入的是 40×40×512 的特征,2000×4 的 Proposal,经过 ROI Polling 后输出是 2000×7×7×512。整个网络模块搭建的思路是按功能、按算子分别进行封装,然后组建成 Faster R-CNN 模型,并根据需要可调用 RPN 或者 ROI 网络。

### 5.3.3 代码实战 RPN 损失函数及训练

代码实现损失函数通常需要如图 5-24 所示的步骤。

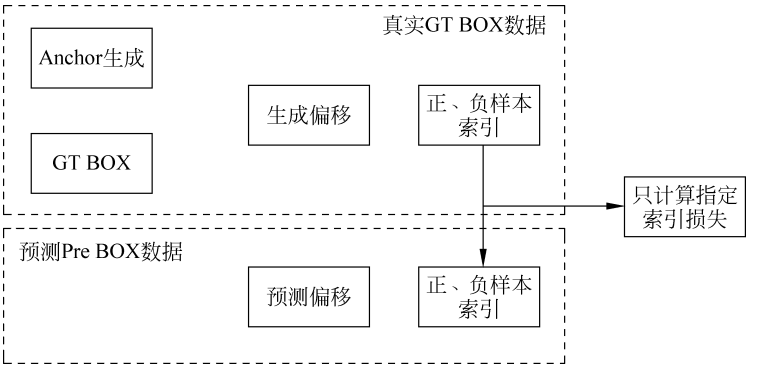



图 5-24 损失函数实现过程

因为要做损失，所以需要先计算 $y$ 值，$y$ 值需要在特征图上计算建议框(Anchor)与真实框 IOU，并将满足 IOU 分数的 Anchor 计算偏移，并记录有目标的 Anchor 的索引 index，及选取为负样本的 index，然后在前向传播预测后，根据 index 信息从预测值中取对应的预测偏移值、预测正负样本的分类信息，按损失函数公式进行计算，然后网络会根据损失函数进行反向传播并学习权重参数。整个过程实现较复杂，但比较关键的是如何计算 $y$ 值。

首先看 Anchor 的生成，代码如下：

```
#第 5 章/ObjectDetection/FasterR-CNN/anchors.py
import numpy as np
import utils as u

#生成 Anchor 默认的各种尺寸
def generate_anchors(base_size=16, ratios=[0.5, 1, 2], anchor_scales=[8, 16, 32]):
    #M×N/16,映射到特征图上的大小,如果目标较小或者 Feature Map 较小,则此值需要调节
    #需要跟 Feature Map 相等
    py, px = base_size / 2., base_size / 2.
    #Anchor 的 3 种比例
    num_ratios = len(ratios)
    #3 种 Anchor 的尺寸
    num_scales = len(anchor_scales)
    #初始全为 0 的矩阵 3× 3 = 9 个 Anchor
    anchor_base = np.zeros([num_ratios *num_scales, 4], dtype=np.float32)
    #每个以不同比例生成不同 h 和 w 的 Anchor
    for i in range(num_ratios):
        for j in range(num_scales):
            #Anchor 的 height 和 width
            #h=16×8×sqrt(0.5)=90。对于小目标来讲,此值较大。需要调节比例或者尺寸
            h = base_size *anchor_scales[j] *np.sqrt(ratios[i])
            #w=16×8×1/sqrt(0.5)=181
            w = base_size *anchor_scales[j] *np.sqrt(1. / ratios[i])
            index = i *num_scales + j
            #计算每个框的 xmin,ymin,xmax,ymax
            anchor_base[index, 0] = py - h / 2.
            anchor_base[index, 1] = px - w / 2.
            anchor_base[index, 2] = py + h / 2.
```

```
            anchor_base[index, 3] = px + w / 2.
    return anchor_base


def mapping_anchor_2_original_image(
        anchor_base,          #基本 Anchor 的尺寸
        feature_hw_size,      #特征图的尺寸
        feature_stride=16,    #M/16
        anchor_num=9,
        image=None,
        gt_boxes=None
):
    #feature_hw_size 特征图的尺寸
    h, w = feature_hw_size
    #每隔 16 生成一个坐标信息,h× feature_stride 即原图的大小。也就是原图被分成了 16 份
    #每隔 16,将原图分为 40 份,宽和高各为 640。也就是此时 1 像素,映射到原图为 16×16 的区域
    shift_y = np.arange(0, h * feature_stride, feature_stride)
    shift_x = np.arange(0, w * feature_stride, feature_stride)
    #组成(x,y)的坐标信息
    shift_x, shift_y = np.meshgrid(shift_x, shift_y)

    shift = np.stack((shift_y.ravel(), shift_x.ravel(),
                      shift_y.ravel(), shift_x.ravel()), axis=1)
    #在每个坐标中都成 9 个 Anchor,并且 9 个 Anchor 根据坐标的位置进行相应调整
    #1×9×4 + 1×64×4
    anchor = anchor_base.reshape([1, anchor_num, 4]) + \
             shift.reshape([1, shift.shape[0], 4]).transpose([1, 0, 2])
    #以第 693 个坐标点为中心,绘 Anchor
    #u.draw_anchor(image, shift_x, shift_y, anchor[693, ...],gt_boxes)
    #reshape 成每个特征点都有 4 个位置,即(num_box,4)
    anchor = anchor.reshape([shift.shape[0] * anchor_num, 4]).astype(np.float32)
    return anchor
```

在函数 generate_anchors(base_size=16,ratios=[0.5,1,2],anchor_scales=[8,16,32])中根据经验值将 Anchor 的比例先验设置为[0.5,1,2],然后每个比例预测尺寸为[8,16,32],通过 base_size×anchor_scales[j]×np.sqrt(ratios[i])和 base_size×anchor_scales[j]×np.sqrt(1./ratios[i])来计算 Anchor 的大小。每个坐标点都会生成 9 个 Anchor。

在 mapping_anchor_2_original_image()函数中 feature_hw_size 为特征图的尺寸,将原图划分为 np.arange(0,h×feature_stride,feature_stride)份,假设输入为 640×640,则为 np.arange(0,40×16,16),然后沿 $x$ 轴和 $y$ 轴进行复制,并且在每个坐标中对 Anchor 共 9 个框进行复制,如果可视化,则可得图 5-18。

然后根据图 5-24 中的结构,计算 offset、正样本、负样本,代码如下:

```
#第 5 章/ObjectDetection/FasterR-CNN/data_processing.py
    def assign(
            self, true_boxes,
            pos_threshold=0.7,
            neg_threshold=0.3,
```

```
            num_sample=256,        #正样本+负样本,一共只能有 256 个
            image=None,
            old_boxes=None
        ):
            anchor_num, feature_stride = 9, 16
            feature_hw_size = np.array(self.input_shape) //feature_stride
            #各种 Anchor 的比例,得到的是 xmin,ymin,xmax,ymax
            anchor_base = a.generate_anchors(base_size=feature_stride)
            #以 640×640 输入时,在每幅图中生成 14400×4 个 BOX 的信息
            anchors = a.mapping_anchor_2_original_image(
                anchor_base, feature_hw_size,
                feature_stride, anchor_num, image,
                gt_boxes=old_boxes
            )
            #Anchor 归一化
            anchors = anchors / self.input_shape[0]
            #初始化 1 个 14400×4 为 0 的 BOX 信息
            true_boxes_assign = np.zeros([anchors.shape[0], 4])
            #[14400,2],初始概率为[0,1]。0 代表背景的概率,1 代表前景的概率
            true_boxes_assign_clf = np.zeros([anchors.shape[0], 2])
            #遍历每个 GT BOX,计算 IOU 后的偏移
            for i in range(len(true_boxes)):
                #计算 Anchor 与 GT BOX 的 IOU
                iou_score = u.get_iou(true_boxes[i], anchors)
                #区域搜索出来的 BOX 与真实框>0.7 的 BOX 信息,当为正样本数据时正样本有目标,
                #所以类别为 1
                #得到的是满足条件的索引 index
                pos_index = np.argwhere(iou_score >= pos_threshold)
                if not pos_index.shape[0]:
                    #如果没有一个 IOU 的值大于 0.5,则获取最大的那个 BOX,并置为有目标
                    pos_index = np.argmax(iou_score)
                    num_pos = 1
                else:
                    num_pos = pos_index.shape[0]
                #根据正样本的 index 取出正样本的 Anchor
                positive_box = anchors[pos_index]
                #正样本可视化
                #anchor_img = u.draw_box(image,positive_box.reshape([-1,4])*640)
                #GT BOX 可视化
                #u.draw_box(anchor_img,true_boxes[i][0:4].reshape([-1,4])*640,
                #color=[255,0,0])
                #offset 计算时,需要转换成 cx,cy,w,h
                a_class_box = u.xyxy2cxcywh(positive_box)       #正样本
                g_class_box = u.xyxy2cxcywh(true_boxes[i])      #GT BOX
                #计算偏移值
                offset_box = u.cxywh2offset(g_class_box, a_class_box)
                #正样本的偏移值,更新到 true_boxes_assign 中
                true_boxes_assign[pos_index] = offset_box
                #RPN 不用读取分类信息。正样本的 np.array([1, 0])代表前景概率为 1,背景概率为 0
                true_boxes_assign_clf[pos_index] = np.array([1, 0])
                #负样本<0.3 并且 256-正样本的数量为负样本
                neg_index = np.argwhere(iou_score < neg_threshold)
```

```
            neg_num = abs(num_sample - num_pos)
            neg_index = neg_index[:neg_num]
            true_boxes_assign_clf[neg_index] = np.array([0, 1])
        #reshape 成[14400×2,1]
        true_boxes_assign_clf = np.reshape(true_boxes_assign_clf, [-1, 1])
        #输出[14400×4],[14400×2,1]
        return true_boxes_assign, true_boxes_assign_clf, anchors
```

代码中 Anchors 的生成调用 mapping_anchor_2_original_image()函数，然后遍历每个 true_boxes 通过 get_iou(true_boxes[i],anchors)计算 IOU 和得分。根据 pos_index=np.argwhere(iou_score>=pos_threshold)得到满足阈值数组的 index，然后由 positive_box=anchors[pos_index]取出正样本的信息，由 offset_box=u.cxywh2offset(g_class_box,a_class_box)得到正样本的偏移值。最后将编码的值赋给 true_boxes_assign，true_boxes_assign_clf 得到真实的偏移值和有无目标的分类概率。

RPN 损失函数的构建，代码如下：

```
#第 5 章/ObjectDetection/FasterR-CNN/loss.py
import tensorflow as tf

def l1_smooth_loss(y_true, y_pred):
    """回归损失"""
    abs_loss = tf.abs(y_true - y_pred)
    sq_loss = 0.5 * (y_true - y_pred) **2
    l1_loss = tf.where(tf.less(abs_loss, 1.0), sq_loss, abs_loss - 0.5)
    return tf.reduce_sum(l1_loss, -1)


def cross_entropy_loss(y_true, y_pred):
    """交叉熵损失"""
    y_pred = tf.maximum(y_pred, 1e-8)
    softmax_loss = -tf.reduce_sum(y_true *tf.math.log(y_pred), axis=-1)
    return softmax_loss

def rpn_loss(y_pre, true_box, true_clf):
    batch_size = true_box.shape[0]
    #由 NumPy 转换成 Tensor 类型
    true_box = tf.cast(true_box, dtype=tf.float32)
    true_clf = tf.cast(true_clf, dtype=tf.float32)
    #预测值 (b,14400,4) (b,28800,1)
    pre_box, pre_clf = y_pre[1], y_pre[0]
    #当 y_true 传过来时，true_clf 已包括正样本和负样本，并且均有设置
    #因为 true_clf 默认为[0,0]，当有目标时是[1,0]，当没有目标时是[0,1]
    #reshape 后变成[[0],[0]],[[1],[0]],[[0],[1]],
    #而交叉熵-y_ture *log(y_pre)，如果 y_true 为 0，则最后的值为 0，所以不再取正样本
    clf_loss = tf.reduce_mean(
        cross_entropy_loss(true_clf, pre_clf)
    )
    #因为如果是负样本，则 true_box 对应的值为 0
    pos_mask = true_box[..., :4] != tf.convert_to_tensor([0., 0., 0., 0.])
```

```
        #只选正样本的偏移值
        box_loss = tf.reduce_mean(
            l1_smooth_loss(true_box[pos_mask], pre_box[pos_mask])
        )
        total = clf_loss + box_loss
        return total / batch_size
```

函数 l1_smooth_loss()封装了 Smooth 损失，cross_entropy_loss()则封装了交叉熵损失，rpn_loss(y_pre,true_box,true_clf)对 RPN 有无目标进行分类损失和正样本 offset 的回归损失。损失函数的构建需要用深度学习框架构建，否则在反向传播时求出的梯度值有可能为 None。rpn_loss()构建的复杂、简易度受 assign()函数的影响。

训练时只需读取 DataProcessingAndEnhancement()类中的 generate()方法，根据参数设置传输数据，代码如下：

```
#第 5 章/ObjectDetection/FasterR-CNN/data_processing.py
    def generate(self, isTraining=True, isRoi=False):
        #训练时将训练集打乱，不训练时使用验证集
        shuffle(self.train_lines)
        lines = self.train_lines if isTraining else self.val_lines
        #batch 存储相关字段
        inputs = []
        true_box_list = []
        true_clf_list = []
        roi_offset_list = []
        roi_class_list = []
        #循环处理数据集中的数据
        for row in lines:
            #将图像和 BOX 缩放到指定的尺寸
            img, y = self.get_image_processing_results(row)
            if len(y) != 0:
                boxes = np.array(y[:, :4], dtype=np.float32)
                #对 label 进行归一化
                old_boxes = boxes.copy()
                boxes = boxes / np.array(self.input_shape[0:2] + self.input_shape[0:2])
                #对 label 构建 one_hot 编码
                one_hot_label = np.eye(self.num_classes)[np.array(y[:, 4], np.int32)]
                #将 max xy - min xy <0 的 label 过滤掉
                if ((boxes[:, 3] - boxes[:, 1]) <= 0).any() and ((boxes[:, 2] - boxes
[:, 0]) <= 0).any():
                    continue
                #组成 xmin,ymin,xmax,ymax,[0,1]
                y = np.concatenate([boxes, one_hot_label], axis=-1)
            true_boxes_assign, true_boxes_assign_clf, anchors = self.assign(y,
image=img, old_boxes=old_boxes)
            if isRoi:
                roi_offset_y, roi_class = self.roi_generate(img, anchors, y)
                roi_offset_list.append(roi_offset_y)
                roi_class_list.append(roi_class)
            inputs.append(img)
            true_box_list.append(true_boxes_assign)
```

```
            true_clf_list.append(true_boxes_assign_clf)
            #按 batch_size 传输 targets
            if len(inputs) == self.batch_size:
               tmp_inp = np.array(inputs, dtype=np.float32)

               if isRoi:
                  tmp_roi = np.array(roi_offset_list)
                  tmp_roi_class = np.array(roi_class_list)
                  roi_offset_list = []
                  roi_class_list = []
                  inputs = []
                  yield tmp_inp, tmp_roi, tmp_roi_class
               else:
                  tmp_box = np.array(true_box_list)
                  tmp_clf = np.array(true_clf_list)
                  true_box_list = []
                  true_clf_list = []
                  inputs = []
                  yield tmp_inp, tmp_box, tmp_clf
```

方法 generate()主要根据 isRoi 的设定返回 RPN 的训练数据或者 ROI 的训练数据，如果是 RNP 数据，则通过 self. assign()方法得到 Anchor 与 GT BOX 的 offset 和 class 分类信息。self. get_image_processing_results()仅实现了输入尺寸 $M \times N$ 的调节及图像的归一化功能。当 len(inputs)==self. batch_size 相等时，通过 yield 生成器返回 tmp_inp 图片、tmp_boxanchor 与 GT BOX 的 offset、tmp_clf 有无目标的分类信息。

训练代码相对容易，只需读取 generate()的数据，然后调用 train_step()实现反向传播，代码如下：

```
#第 5 章/ObjectDetection/FasterR-CNN/rpn_train.py
    def train_step(model, features, true_box, true_clf):
        with tf.GradientTape() as tape:
            #置信度是 batch×28 800×1
            #位置是 batch×14 400×4
            predictions = model(features, training=True)
            #传入 RPN 预测的分类和位置信息，并且传入 true_box 信息
            loss = rpn_loss(predictions, true_box, true_clf)
        #求梯度
        gradients = tape.gradient(loss, model.trainable_variables)
        #反向传播
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
        #更新 loss 信息
        train_loss.update_state(loss)
        train_metric.update_state(true_clf, predictions[0])
        global_steps.assign_add(1)
```

代码 loss=rpn_loss(predictions，true_box，true_clf)中的 predictions 包含 pre_box 预测框基于 Anchor 的偏移，以及 pre_clf 预测框有无目标的概率，此时维度要跟 GT BOX 的信息保持一致，这样就更方便 rpn_loss()函数进行计算。更多更详细的代码可参考随书代码。

### 5.3.4 代码实战 ROI 损失函数及训练

经过 RPN 的训练此时 14 400×4 个预测框基于 Anchor 的偏移，根据图 5-22 的结构需要选出 2000 个重叠率较低的框，然后将 2000 个框与 GT BOX 再进行 1 次偏移作为 roi_loss()中的 y_true，此过程封装在代码 roi_generate()中，代码如下：

```
#第 5 章/ObjectDetection/FasterR-CNN/data_processing.py

    def roi_generate(self, img, anchors, true_box, input_shape=[640, 640, 3]):
        rpn_model, all_model = Faster R-CNN(
            input_shape=input_shape,
            roi_input_shape=[None, 4],
            num_anchors=9,
            num_class=2 + 1
        )
        #加载上一步训练的权重
        rpn_model.load_weights('./weights/last.h5')
        #输入图片的特征，输出
        pre_rpn_cls, pre_rpn_box = rpn_model(np.expand_dims(img, axis=0))
        #如果启动 ROI，则根据 y 值和上一次的权重得到区域选择框
        #得到 2000 个 roi(1,2000,4)
        pre_roi = pre_roi_2_box(pre_rpn_box, pre_rpn_cls, anchors, np.array
(self.input_shape[0:2]))
        #u.draw_box(img *255, pre_roi.reshape([2000, 4]) *640)
        #构建 ROI 网络的 y 值 2000×8
        roi_y = gt_box_2_roi(true_box, pre_roi)
        return roi_y
```

代码 rpn_model.load_weights()调用 RPN 网络的权重，然后前向传播得到 pre_rpn_cls，pre_rpn_box 的输出，然后通过 pre_roi_2_box()函数得到 2000 个建议框，gt_box_2_roi()函数得到 ROI 的 y_true。

函数 pre_roi_2_box()的实现，代码如下：

```
#第 5 章/ObjectDetection/FasterR-CNN/proposal.py
import numpy as np
import utils as u
import tensorflow as tf

def pre_roi_2_box(rpn_box_loc, rpn_box_score,
                  anchor, img_size,
                  num_pre=12000, nms_thresh=0.7, ss_num=2000):
    #根据 RPN 网络输入 BOX Loc 和 BOX Score，以及 Anchor
    #选择输出 2000 个训练 rois。注意此时没有 GT BOX 的事情
    #因为 Anchor 是 xmin,ymin,xmax,ymax 转换成 x,y,w,h 的形式
    anchor = u.xyxy2cxcywh(anchor)
    batch = rpn_box_loc.shape[0]
    #因为 RPN 输出的是 offset 值，所以需要将其解码成 xmin,ymin,xmax,ymax
    roi_box = u.offset2xyxy(rpn_box_loc.NumPy(), anchor)
    #对 roi_box 中的 BOX 进行裁剪
```

```
        roi_box[:, slice(0, 4, 2)] = np.clip(roi_box[:, slice(0, 4, 2)], 0, img_size[0])
        roi_box[:, slice(1, 4, 2)] = np.clip(roi_box[:, slice(1, 4, 2)], 0, img_size[1])
        ########################
        #得到满足目标大小的置信度分数[1, 0]
        rpn_box_score = np.reshape(rpn_box_score, [batch, -1, 2])
        pos_score = rpn_box_score[..., 0] #得到前景的分数
        #对置信度的分数进行降序排列
        order = tf.argsort(pos_score, direction='DESCENDING').NumPy()
        #取前 12000 个预测框
        order = order.ravel()[:num_pre]
        #对 ROI 进行过滤,只要 120000 个置信度较大的框
        roi = roi_box[..., order, :]
        score = pos_score[..., order]
        #使用 NMS 去掉交并比>0.7 的框,留下重叠率不高的框,最多只有 2000 个
        #non_max_suppression()得到的是 ROI 中的 index
        keep = tf.image.non_max_suppression(
            np.reshape(roi, roi.shape[1:]),
            np.reshape(score, score.shape[1:]),
            max_output_size=ss_num,
            iou_threshold=nms_thresh
        )
        #利用 Anchor 和 RPN 网络预测出来的 offset,选取可能有目标最大的前 2000 个框,作为建议框
        roi = roi[..., keep, :]
        return roi
```

因为 RPN 预测输出的是 offset,所以调用 offset2xyxy()进行解码以得到左上、右下坐标,此时 BOX 信息较多,然后根据 pos_score＝rpn_box_score[...,0]有无目标概率的得分进行非极大值抑制,从而保留重叠率不高的框,共计 2000 个,所以此函数的输出为 2000×4,代码如下:

```
#第 5 章/ObjectDetection/FasterR-CNN/proposal.py
def gt_box_2_roi(gt_box, roi, sample_num=128, pos_iou_thresh=0.7, neg_iou_
thresh=0.1, num_class=2 + 1):
    #RPN 产生了 2000 个区域选择框,但是并没有将 2000 个框全用作训练,而是将 2000 个框与
GT BOX 做 IOU,一共选择 128 个正样本
    #假设 IOU>0.7 的正样本选择 32 个。IOU<0.3 的负样本选择 128-32=96 个
    roi = np.reshape(roi, roi.shape[1:])
    #初始为 0 的数组,用来保存计算后的值
    true_offset = np.zeros([roi.shape[0], 4 * (num_class - 1)])
    true_class = np.zeros([roi.shape[0], num_class])

    for box in gt_box:
        #如果此时的 BOX 类别假设为[1,0,0]
        #如果此时的 BOX 类别假设为[0,1,0],offset 则更新到 1 的位置
        #再次计算每个 GT BOX 与 ROI 的 IOU 值
        iou_score = u.get_iou(box[:4], roi[..., :4])
        #如果 IOU>0.7,则全记为正样本
        pos_iou = iou_score > pos_iou_thresh
        if not sum(pos_iou):
            pos_index = np.argmax(iou_score)
        else:
```

```
            pos_index = np.argwhere(pos_iou == True)
        num_pos = pos_index.size
        #如果 IOU<0.3,则为负样本
        neg_iou = iou_score < neg_iou_thresh
        neg_index = np.argwhere(neg_iou == True)
        if pos_index.size != 1:
            pos_index = pos_index[:num_pos]
        #负样本的数量
        neg_index = neg_index[:sample_num - num_pos]
        #取出正样本的 BOX
        pos_box = roi[pos_index]
        pos_box = np.reshape(pos_box, [-1, 4])
        #计算 cx,cy,w,h
        pos_box = u.xyxy2cxcywh(pos_box)
        gt_box = u.xyxy2cxcywh(gt_box)
        #然后对正样本的 BOX 进行编码
        roi_offset = u.cxywh2offset(box, pos_box)
        for i in range(num_class - 1):
            if box[4 + i] == 1:
                start = i *4
                #正样本的 offset
                true_offset[pos_index.flat, start:4 + start] = roi_offset
        #负样本只填充负样本的分类信息
        neg_class = np.zeros(num_class)
        neg_class[-1] = 1 #[0,0,1]代表负样本
        #将计算后的值赋给 true_class
        true_class[neg_index, :] = neg_class
        true_class[pos_index, :] = box[4:]
    return true_offset, true_class
```

函数 gt_box_2_roi()实现将输入的 2000 个建议框与 GT BOX 之间做 IOU，如果 IOU＞0.7，则为正样本并计算 offset，如果 IOU＜0.3，则为负样本，则正负样本之和为 128(可调)。因为 true_offset 的输出为 2000×4×(num_class－1)，在此函数为 2000×8，即每个框可能为两个分类的偏移，所以当 if box[4＋i]＝＝1 时，当前分类才赋值 true_offset[pos_index.flat，start：4＋start]＝roi_offset。

接下来是损失函数的构建，代码如下：

```
#第 5 章/ObjectDetection/FasterR-CNN/loss.py
def roi_loss(y_pre, roi, roi_class, num_class=2 + 1):
    #预测信息 2000×3 和 2000×8
    out_class, out_regbox = y_pre
    #真实值 2000×8 和 2000×3
    y_true = tf.cast(roi, tf.float32)
    true_cls = tf.cast(roi_class, tf.float32)
    batch_size = y_true.shape[0]
    #只要位置信息
    true_box = y_true[..., :8]
    #因为 true_cls 默认为[0,0,0],负样本是[0,0,1],正样本是[1,0,0]
    #所以-y_true *log(y_pred),不用提取出负样本的分类
    clf_loss = tf.reduce_mean(
```

```
            cross_entropy_loss(true_cls, out_class)
        )
        #按每个类别去求正样本 BOX offset 的损失
        box_loss = 0
        for i in range(num_class - 1):
            #因为如果是负样本,true_box 对应的值为 0,所以只选正样本的偏移值
            mask = tf.reshape(true_cls[..., i], [batch_size, -1, 1])
            ind = tf.where(mask == 1)
            #根据 index 取值
            pos_true_box = tf.gather_nd(true_box, ind)
            pos_out_box = tf.gather_nd(out_regbox, ind)
            box_loss += tf.reduce_mean(
                l1_smooth_loss(pos_true_box, pos_out_box)
            )
        total = clf_loss + box_loss
        return total / batch_size
```

函数 roi_loss()与 rpn_loss()与此类似,不同之处在于求解了预测 BOX 的分类信息,同时实现时按每个类别的 offset 损失进行求和。

训练代码 train_step()与 RPN 类似,不同之处在于 predictions=model([features,roi_box[...,:4]],training=True)时传入图像信息及 2000 个 Proposal 信息。更多更详细的代码可参考随书代码。

### 5.3.5　代码实战预测推理

Faster R-CNN 的推理首先要经过 RPN 网络得到 14 400×2 个分类、14 400×4 个框,再经过 NMS 选出 2000×4 个框,然后送入 ROI 网络,假设分类数量为 2,则类别概率为 2000×3(2+1 前景/背景),每个类别的 BOX 信息为 2000×8(有目标的 BOX 和没有目标的 BOX),其详细的推理代码如下:

```
#第 5 章/ObjectDetection/FasterR-CNN/detected.py
class Detected():
    def __init__(self, rpn_path, roi_path, input_size):
    #读取模型和权重
        self.rpn_model = load_model(
            rpn_path,
            custom_objects={'rpn_loss': rpn_loss}
        )
        self.roi_model = load_model(
            roi_path,
            custom_objects={'roi_loss': roi_loss}
        )
        self.confidence_threshold = 0.5              #有无目标置信度
        self.class_prob = [0.5, 0.5]                 #两个类别的分类阈值
        self.nms_threshold = 0.5                     #NMS 的阈值
        self.input_size = input_size

    def generate_anchor(self):
        #默认 Anchor 的生成
```

```
        anchor_num, feature_stride = 9, 16
        feature_hw_size = np.array(self.input_size) //feature_stride
        #各种 Anchor 的比例，得到的是 xmin,ymin,xmax,ymax
        anchor_base = a.generate_anchors(base_size=feature_stride)
        #以 640×640 输入时,在每幅图中生成 14400×4 个 BOX 的信息
        anchors = a.mapping_anchor_2_original_image(
            anchor_base, feature_hw_size,
            feature_stride, anchor_num
        )
        #Anchor 归一化
        anchors = anchors / self.input_size[0]
        self.anchors = anchors

    def readImg(self, img_path=None):
        #读取要预测的图片
        img = cv2.imread(img_path)
        #将图片转换为 640×640
        self.img, _ = u.letterbox_image(img, self.input_size, [])

    def rpn_forward(self):
        #读取图片以进行前向传播
        img_tensor = tf.expand_dims(self.img / 255.0, axis=0)
        #前向传播
        self.output = self.rpn_model.predict(img_tensor)
        self.old_img = self.img.copy()

    def rpn_nms2k(self):
        #预测置信度的结果为 14400×2
        pre_rpn_cls = tf.cast(self.output[0], dtype=tf.float32)
        #预测框的偏移
        pre_rpn_box = tf.cast(self.output[1], dtype=tf.float32)
        #pre_roi_2_box 已集成根据预测框进行解码操作,并根据置信度的得分进行 NMS 去重,
        #得到 2000×4 个框
        #14400× 4, 28800×1
        pre_roi = pre_roi_2_box(pre_rpn_box, pre_rpn_cls, self.anchors,
np.array(self.input_size[0:2]), isReturnAnchor=True)
        #喂入 ROI 网络此时的 BOX 信息
        self.pre_roi = pre_roi[0]
        #喂入 ROI 网络此时 BOX 对应的 Anchor 值
        self.pre_anchor = pre_roi[1]

    def roi_forward(self):
        img_tensor = tf.expand_dims(self.img / 255.0, axis=0)
        #ROI 前向传播,得到 2000×3 和 2000×8
        self.output = self.roi_model.predict([img_tensor, self.pre_roi])
        #分类+置信度的值
        self.pre_class = self.output[0]
        #BOX 的值,但是输出为 2000×8,其中有 4 位是没有目标的框
        #此时仍为偏移值
        self.pre_box = self.output[1]

    def _roi_box_decode(self, pre_box):
```

```
        #转换成 cx,cy,w,h。只取喂入 ROI 网络的 2000 个 Anchor
        anchor = utils.xyxy2cxcywh(self.pre_anchor)
        #解码操作封装在 offset2xyxy 函数中
        decode_box = utils.offset2xyxy(pre_box, anchor)
              #裁剪值域在[0,1]
        decode_box = np.clip(decode_box, 0, 1)
        return decode_box

    def classification_filtering(self):
        #首先根据置信度过滤
        #因为在训练时 [0,0,1]代表负样本,所以只有最后 1 位小于阈值时才代表有目标
        #即背景的概率越小越好
        obj_mask = self.pre_class[..., -1] < self.confidence_threshold
        #将 ROI 输出的预测值 BOX 与 RPN 输出的最佳 2000 个框进行解码操作
        #并且只传前 4 个位置的,因为后面 4 个位置的是背景所处的位置
        boxes = self._roi_box_decode(self.pre_box[..., :4])
        #根据类别的概率过滤
        for i, p in enumerate(self.class_prob):
            if len(obj_mask):
                #根据 obj_mask 过滤分类
                cls_obj = self.pre_class[obj_mask]
                #类别的得分要超过阈值
                cls_obj_mask = cls_obj[..., i] > p
                classification = cls_obj[cls_obj_mask]
                #根据置信度和类别的 mask 过滤出对应 boxes 中的信息
                class_boxes = boxes[obj_mask][cls_obj_mask]
                class_score = classification[..., i]
                #将 BOX 与分类得分合并
                box = np.concatenate([class_boxes, np.reshape(class_score,
[-1, 1])], axis=-1)
                #NMS 得到的索引
                index = u.nms(box, nms_thresh=self.nms_threshold)
                #根据索引取出 boxes 信息
                box = class_boxes[index]
                #绘图
                img = u.draw_box(self.old_img, box)
        u.show(img)
if __name__ == "__main__":
    rpn_path = "./weights/last"
    roi_path = "./weightsRoi/RoiLast"
    img_path = "../val_data/pexels-photo-5211438.jpeg"
    #实例化对象
    det = Detected(rpn_path, roi_path, [640, 640])
    #读取预测图片
    det.readImg(img_path)
    #生成 Anchor
    det.generate_anchor()
    ####################
    #RPN 网络前向传播
    det.rpn_forward()
    #RPN 网络解析得到 2000 个框
    det.rpn_nms2k()
```

```
        ####################
        #ROI 前向传播
        det.roi_forward()
        #对最后的结果解码
        det.classification_filtering()
```

因为 Faster R-CNN 是两阶段网络，所以需要分别加载 RPN、ROI 的权重进行预测。rpn_forward(self)实现读取图片的前向传播，然后调用 rpn_nms2k(self)得到 2000 个候选框，并同时得到此时相对应的 2000 个 Anchor 的值。将 ROI 的 2000 个框在 roi_forward(self)进行预测 self.roi_model.predict([img_tensor,self.pre_roi])，获得分类和置信度的输出 self.pre_class 及最终的 BOX 信息 self.pre_box。在 classification_filtering(self)方法中根据置信度为前景的概率和分类得分进行过滤，并对 self.pre_box 进行由偏移值转向位置值的解码操作，最后经过 NMS 非极大值抑制到得最终的结果。

**总结**

Faster R-CNN 延续了 R-CNN 中的思想，分为两个阶段。第 1 个阶段，RPN 网络用来提取 2000 个前景、背景框；第 2 个阶段，网络输入特征信息和 2000 个框，进行分类＋背景及 BOX 的微调。

**练习**

运行并调试本节代码，理解算法的设计与代码的结合。

## 5.4 单阶段多尺度检测网络 SSD

### 5.4.1 模型介绍

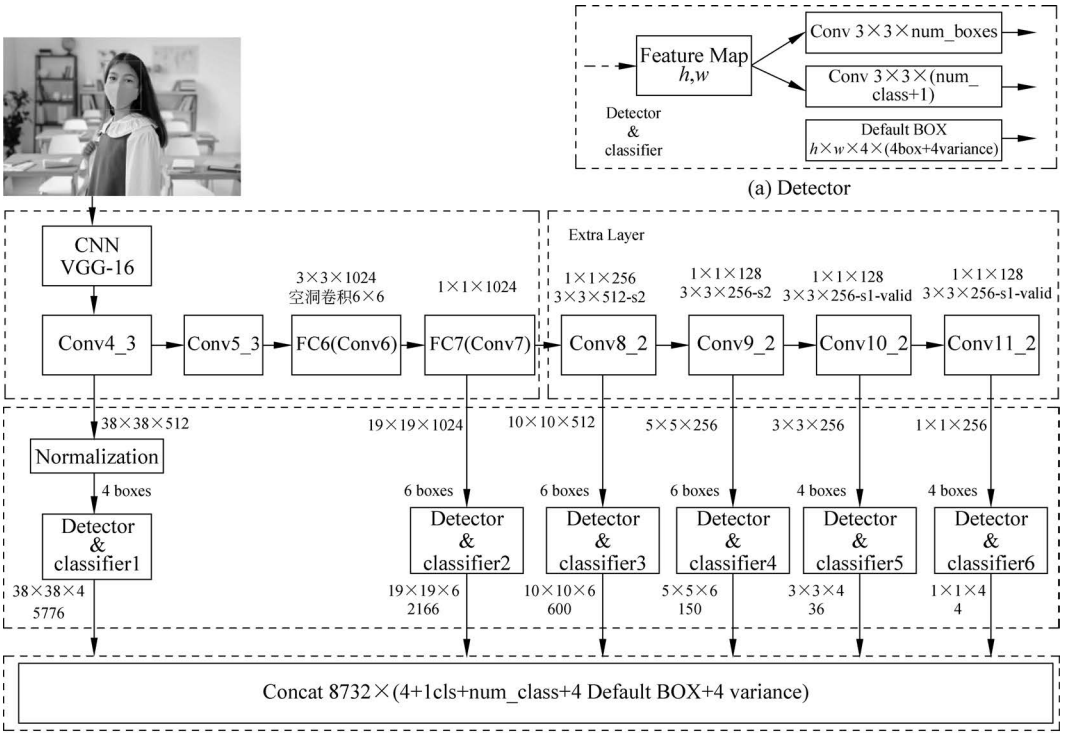
SSD 是于 2015 年由 Wei Liu 等在发表的论文 *SSD: Single Shot MultiBox Detector* 中提出的单阶段、多检测头的目标检测网络，其网络结构如图 5-25 所示。

在特征提取阶段使用 VGG-16(结构为 2—2—3—3—3—3，2 卷积，1 池化，最后 3 为全连接)，并选取 Conv4_3(第 4 个层的最后 1 个卷积)输出 38×38×512 作为 classifier1 检测头的输入。当时没有使用 BN 归一化，而 Conv4_3 提取的特征值较大，所以使用 Normalization 归一化以防止梯度爆炸。

将 VGG-16 中的全连接 FC6 层更换为卷积，并使用空洞卷积，空洞率为 6×6，增大感受野。SSD 的源码在 Conv5_3 后的 Pooling 由 2×2－s2 更换为 3×3－s1，原有 FC6 在 Conv5_3 上的感受野为 14×14，而 FC6 由原来的 7×7 变成 3×3 后，使用空洞卷积 3＋(3＋2($n$－1)－1)×1＝14，算出来 $n$＝5.5，向上取整为 6，使 FC6 的感受野与 VGG-16 时保持一致，然后将 FC7 更换为 1×1×1024 卷积，得到输出 19×19×1024 作为 classifier2 检测头的输入。

在 VGG-16 的基础上添加 Extra Layer，Conv8_2 由 1×1×256、3×3×512－s2 输出 10×10×512 作为 classifier3 检测头的输入；同理 Conv9_2 由 1×1×128、3×3×256－s2 输出为 5×5×256 作为 classifier4 检测头的输入；Conv10_2 由 1×1×128、3×3×256－s1

并且卷积 valid 丢弃，输出为 3×3×256 作为 classifier5 检测头的输入；Conv11_2 与此类似，输出为 1×1×256 作为 classifier6 检测头的输入。




(b) SDD-300模型结构图

图 5-25　SSD 网络结构

共计 6 个特征作为 6 个检测头的输入，并且由 classifier1 检测小目标、classifier6 检测大目标(classifier6 的感受野最大，所以检测大目标)，中间的检测头次之。

得到特征信息后，分别输入两个 3×3 卷积，如 classifier1 输出 38×38×4，即 BOX 的 4 个偏移位置信息，38×38×(1+num_class)的分类信息，其中 1 代表 BOX 有无目标。另外一个分支输出 Default BOX 指在 38×38 的特征层上生成 4 个建议框，即每个特征图对应到原图约 300/38=7.9 的区域，那么一共就有 38×38×4=5776 个建议框，并随着 classifier1 进行输出，SSD 检测头有 4 个，分别是位置偏移值、置信度(有无目标)及分类概率、Default BOX。Default BOX 本来是 5776×4，但是因为 offset 的值较小，作者对(tx,ty)/variance 0.1、(tw,th)/variance 0.2 进行了放大，防止梯度消失，所以其输出变成了 5776×8。

建议框为 classifier1 分配 4 个，为 classifier2、classifier3、classifier4 分配 6 个，为 classifier5、classifier6 分配 4 个，共计 38×38×4+19×19×6+10×10×6+5×5×6+3×3×4+1×1×4=8732 个建议框，所以 SSD 的输出为 8732×4 个框，8732×(1+num)分类概率，8732×8 个 Default BOX，合并后输出为 8732×(4+1cls+num+4Default BOX+4variance)。

综上，SSD 拥有 6 个检测头，其特征信息使用 Conv4_3、Conv7、Conv8_2、Conv9_2、

Conv10_2、Conv11_2 的输出，每个特征信息分配 4、6、6、6、4、4 个框，直接输出 8732×(4+1cls+num+4Default BOX+4variance)，不再经过 Faster R-CNN 中的 RPN 选出建议框，直接预测极大地提高了检测速度。同时由于使用了 6 个不同尺度的特征信息和不同尺度的 Default BOX，3×3 卷积分别进行位置和分类预测，对于模型的精度也有极大的提高。

SSD 的建议框生成过程首先经过以下公式计算最小、最大高宽的尺寸。

$$S_k = S_{\min} + \frac{S_{\max} - S_{\min}}{m-1}(k-1), \quad k \in [1,m] \tag{5-5}$$

公式中 $m$ 指有几个检测头，所以 $m=6$，同时原文初始设定 $S_{\min}=0.2$，$S_{\max}=0.9$。

实际在计算时，由于考虑到第 1 个检测头 $S_k=0.2\times300=60$ 作为 $\text{max\_size}_1$，而将 $\text{min\_size}_1=60\times1/2=30$，所以求其他检测头的尺寸时设置 $m=5$，故 $\frac{S_{\max}-S_{\min}}{m-1}=\frac{0.9-0.2}{5-1}\approx0.17$，然后 $\text{max\_size}_2=(0.2+0.17\times1)\times300=111$，$\text{max\_size}_3=162$，$\text{max\_size}_4=213$，$\text{max\_size}_5=264$，$\text{max\_size}_6=315$，再将上一个检测头的最大值作为下一个检测头的最小值，见表 5-1。

**表 5-1 SSD 建议框默认尺寸**

| 特征层名称 | 特征层尺寸 | min_size(k) | max_size(k) | 比　　例 | step |
|---|---|---|---|---|---|
| Conv4_3 | 38×38 | 30 | 60 | 1∶1,1∶$\sqrt{2}$,$\sqrt{2}$∶1,1∶1 | 8 |
| Conv7 | 19×19 | 60 | 111 | 1∶1,1∶$\sqrt{2}$,$\sqrt{2}$∶1,1∶$\sqrt{3}$,$\sqrt{3}$∶1,1∶1 | 16 |
| Conv8_2 | 10×10 | 111 | 162 | 1∶1,1∶$\sqrt{2}$,$\sqrt{2}$∶1,1∶$\sqrt{3}$,$\sqrt{3}$∶1,1∶1 | 32 |
| Conv9_2 | 5×5 | 162 | 213 | 1∶1,1∶$\sqrt{2}$,$\sqrt{2}$∶1,1∶$\sqrt{3}$,$\sqrt{3}$∶1,1∶1 | 64 |
| Conv10_2 | 3×3 | 213 | 264 | 1∶1,1∶$\sqrt{2}$,$\sqrt{2}$∶1,1∶1 | 100 |
| Conv11_2 | 1×1 | 264 | 315 | 1∶1,1∶$\sqrt{2}$,$\sqrt{2}$∶1,1∶1 | 300 |

第 1 个 1∶1 的比例，宽和高 wh=[30,30]；1∶$\sqrt{2}$ 时为 wh=[30×$\sqrt{2}$,30×1/$\sqrt{2}$]=[42,21]；$\sqrt{2}$∶1 时 wh=[30×1/$\sqrt{2}$,30×$\sqrt{2}$]=[21,42]，最后 1 个 1∶1 为 wh=$\sqrt{30\times60}$∶$\sqrt{30\times60}$=42∶42，其他层与此类似，最后得到建议框的宽和高见表 5-2。

**表 5-2 SSD 建议框宽和高默认值**

| 特征层名称 | 特征层尺寸 | 建议框宽和高 |
|---|---|---|
| Conv4_3 | 38×38 | [30,30]、[42,21]、[21,42]、[42,42] |
| Conv7 | 19×19 | [60,60]、[84,42]、[42,84]、[103,34]、[34,103]、[81,81] |
| Conv8_2 | 10×10 | [111,111]、[156,78]、[78,156]、[192,64]、[64,192]、[134,134] |
| Conv9_2 | 5×5 | [162,162]、[229,114]、[114,229]、[280,93]、[93,280]、[185,185] |
| Conv10_2 | 3×3 | [213,213]、[301,150]、[150,301]、[237,237] |
| Conv11_2 | 1×1 | [264,264]、[373,186]、[186,373]、[288,288] |

与 Faster R-CNN 建议框生成略有不同的是均分不是从图像中的(0,0)坐标开始的，而是采用[0.5×step,300−0.5×step]均分到特征图的尺寸份，例如 Conv7 则为[0.5×16,

300－0.5×16]均分成 19 份。

假设 19×19 特征图中的每个像素映射回原图，$x$ 轴和 $y$ 轴均有 19 个均分点，每两个点之间相差 16×16，并且建议框需要在每个 $x$ 和 $y$ 坐标点为中心生成 6 个红色建议框，然后计算红色框与绿色框之间的 IOU，当 IOU≥0.5 时 Anchor 为正样本，其他为负样本。绿色为真实标注框，可视化效果如图 5-26 所示。

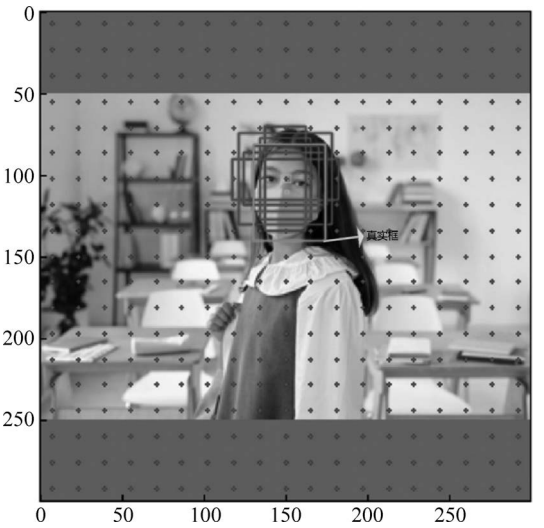



图 5-26　Conv7 特征图中生成的建议框(见彩插)

在计算建议框与真实框之间的偏移时，其计算公式仍采用式(5-2)，得到偏移值后 $t_x$/variance1、$t_y$/variance1、$t_w$/variance2、$t_h$/variance2，其中 variance1＝0.1，variance2＝0.2，对偏移值进行了放大。在推理时使用式(5-3)，然后对于预测值乘以 variance1、variance2 进行相同比例的缩小还原。

SSD 的损失函数分为位置损失、分类误差损失：

$$L(x,c,l,g)=\frac{1}{N}(L_{\text{conf}}(x,c)+\alpha L_{\text{loc}}(x,l,g))$$

$$L_{\text{conf}}(x,c)=-\sum_{i\in\text{Pos}}^{N}x_{ij}^{p}\log(\hat{c}_i^{p})-\sum_{i\in\text{Neg}}^{N}\log(\hat{c}_i^{0}) \tag{5-6}$$

$$\hat{c}_i^{p}=\frac{\exp(c_i^{p})}{\sum_{p}\exp(c_i^{p})}$$

其中，$N$ 为真实边界框配对的建议框数量，对于 $x$，如果 1 个建议框与真实边界框配对为 1，否则为 0；$c$ 为真实物体的预测值；$l$ 用于预测边界框中心位置的长、宽；$g$ 为真实边界框中心位置的长、宽。$L_{\text{loc}}(x,l,g)$使用的是 Smooth 损失函数；$L_{\text{conf}}(x,c)$包含正样本的损失和负样本的损失，并使用 Softmax 求解分类 $\hat{c}_i^{p}$。

因为正样本较少，负样本较多，所以在求解损失时将负样本按背景(无目标)概率从大到

小进行排序，控制负样本与正样本的比例为 3∶1。整个网络的实现，可参见 5.4.2 节。

### 5.4.2 代码实战模型搭建

整个模型的实现可以分为 3 部分，VGG-16 作为特征提取部分，extra_layer 作为 VGG 补充特征提取，detect_head 作为检测头进行多尺度检测并作为输出。

重构 VGG-16 的代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_SSD_Detected/backbone/vggssd.py
def vgg(input_tensor):
    net = {}
    #默认不使用 BN
    is_bn = False
    #将输入内容放入 net 字典中
    net['input'] = input_tensor
    #原 VGG 的配置结构，数字代表输出 channel，M 代表池化
    vgg_config = [
        64, 64, 'M',
        128, 128, 'M',
        256, 256, 256, 'M',
        512, 512, 512, 'M',
        512, 512, 512, 'M'
    ]
    #输入内容
    x = net['input']
    #对配置文件进行解析
    for i, channel in enumerate(vgg_config):
        if channel != 'M':
            #ConvBnRelu 已封装
            x = ConvBnRelu(filters_channels=channel,
                           kernel_size=[3, 3],
                           strides=[1, 1],
                           padding='same',
                           is_bn=is_bn,
                           is_relu=True
                           )(x)
            #每次将卷积内容放入字典中
            net['conv_{}'.format(i + 1)] = x
        else:
            #如果不是最后 1 个池化，则默认为 2×2-s2
            if i != len(vgg_config) - 1:
                pool_size = [2, 2]
                strides = [2, 2]
            else:
                pool_size = [3, 3] #pool5，最后一个池化为 3×3-s=1
                strides = [1, 1]
            x = layers.MaxPooling2D(pool_size=pool_size, strides=strides,
padding='same')(x)
            net['max_pool_{}'.format(i + 1)] = x
    return net
```

重构并通过配置生成模型是为了调优时增加卷积深度和宽度。在构建模型时使用字典来管理各个网络层。ConvBnRelu()是卷积、BN、归一化的封装，is_bn 默认不开启。

在 VGG 的基础上添加 extra_layer 的代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_SSD_Detected/backbone/vggssd.py
def extra_layer(input_tensor):
    """VGG 额外增加的一些 Net """
    net = {}
    is_bn = False
    x = input_tensor
    #格式为 [channel, kernel_size,strides,padding]
    extra_config = [
        [1024, [3, 3], [6, 6], 'same'],                    #fc6
        [1024, [1, 1], [1, 1], 'same'],                    #fc7

        [256, [1, 1], [1, 1], 'same'],                     #conv8_1
        [512, [3, 3], [2, 2], 'same'],                     #conv8_2

        [128, [1, 1], [1, 1], 'same'],                     #conv9_1
        [256, [3, 3], [2, 2], 'same'],                     #conv9_2

        [128, [1, 1], [1, 1], 'same'],                     #conv10_1
        [256, [3, 3], [1, 1], 'valid'],                    #conv10_2

        [128, [1, 1], [1, 1], 'same'],                     #conv11_1
        [256, [3, 3], [1, 1], 'valid'],                    #conv11_2
    ]
    #读配置文件,生成 extra_layer
    for i, cnf in enumerate(extra_config):
        if i == 0:
            #第 1 个是空洞卷积,[6, 6],dilation_rate 指定空洞率
            x = ConvBnRelu(cnf[0], cnf[1], strides=[1, 1],
                           dilation_rate=cnf[-2], padding=cnf[-1],
                           is_bn=is_bn, is_relu=True)(x)
        else:
            x = ConvBnRelu(cnf[0], cnf[1], cnf[2], cnf[-1],
                           is_bn=is_bn, is_relu=True)(x)
        net['conv_extra_{}'.format(i)] = x
    return net

def vgg_extra(input_tensor):
    """对 VGG 和 extra_layer 进行整合"""
    net1 = vgg(input_tensor)
    net2 = extra_layer(net1['max_pool_18'])
    net1.update(net2)
    return net1
```

函数 extra_layer()根据配置文件增加网络层，并且第 1 个网络层的卷积为空洞卷积，vgg_extra(input_tensor)将 VGG 和 extra_layer 进行了整合，构建成整个特征提取网络。net1['max_pool_18']也就是 VGG 的第 5 个卷积后的池化层，如图 5-27 所示。

对 net1 和 net2 进行合并后一共有 29 个网络层，其中网络图与代码的对应关系为 conv4_3＝conv_13、conv7＝conv_extra_1、conv8_2＝conv_extra_3、conv9_2＝conv_extra_5、conv10_2＝conv_extra_7、conv11_2＝conv_extra_9，如图 5-28 所示。

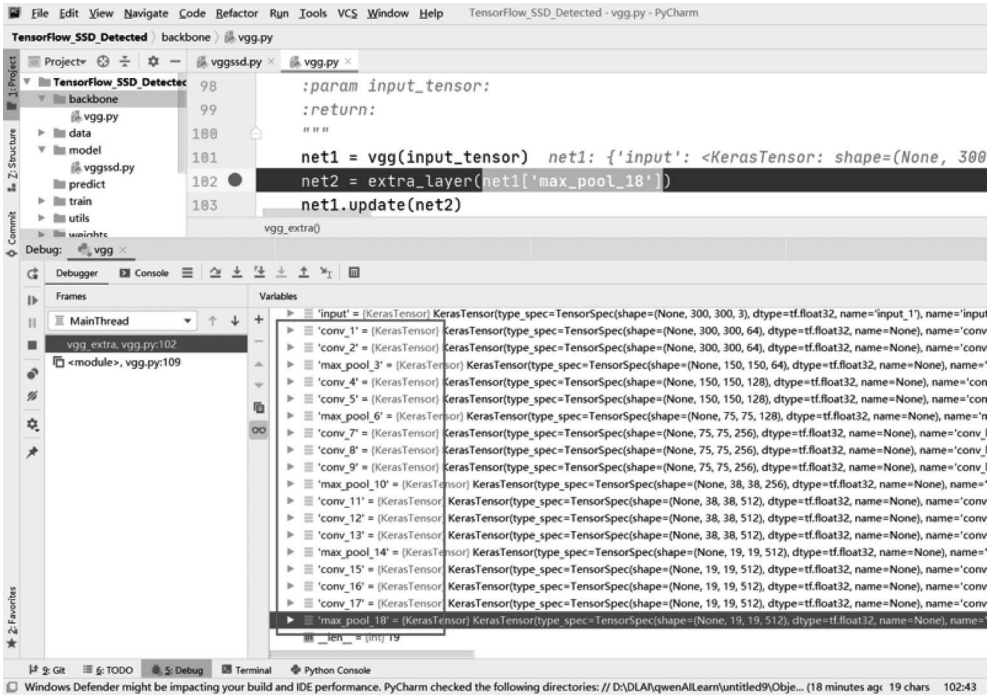



图 5-27 max_pool_18 所处网络层

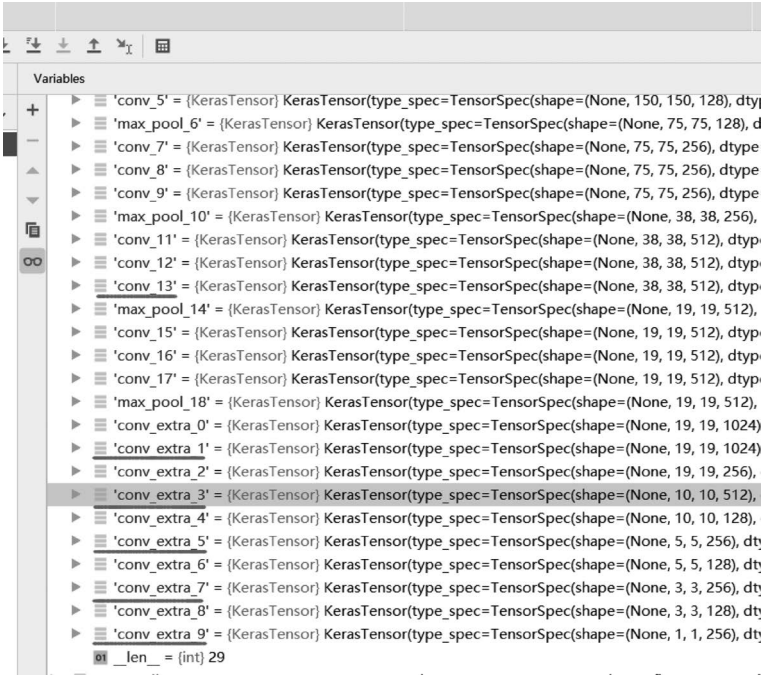



图 5-28 SSD 选择的特征层

检测头的构建分别用了两个 3×3 卷积，并且传入当前特征建议框的宽高比，将在 DefaultBox()中自动生成对应的建议框，代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_SSD_Detected/model/detect_head.py
def vgg_detect_head(input_tensor, num_priors, num_classes,
                    box_layer_name, min_max_size, ratios,
                    img_size=[300, 300]):
    """
    VGG 的检测头，一个用来预测偏移值，一个用来预测分类，另外一个是 Default BOX
    :param input_tensor: 预测的卷积层
    :param num_priors: Default BOX 的数量
    :param num_classes: 类别的数量
    :param box_layer_name: 预测的卷积层的名称
    :param min_max_size: 锚框的最小和最大尺寸
    :param ratios: 建议框的比例
    :param img_size: 输入图像的尺寸
    :return:
    """
    net = {}
    x = input_tensor
    #用来检测 location 的偏移值
    net[box_layer_name + "_loc"] = ConvBnRelu(num_priors * 4,
                                      kernel_size=[3, 3],
                                      strides=[1, 1],
                                      padding='same',
                                      is_bn=False,
                                      is_relu=False)(x)
    #location 位置摊平
    net[box_layer_name + "_loc" + "_flat"] = layers.Flatten()(net[box_layer_name +
"_loc"])
    #用来检测每个 default_box 的 classes 得分
    net[box_layer_name + "_conf"] = ConvBnRelu(num_priors * num_classes,
                                       kernel_size=[3, 3],
                                       strides=[1, 1],
                                       padding='same',
                                       is_bn=False,
                                       is_relu=False)(x)
    net[box_layer_name + "_conf" + "_flat"] = layers.Flatten()(net[box_layer_
name + "_conf"])
    #Default BOX 的生成
    net[box_layer_name + "_default_box"] = DefaultBox(min_max_size, ratios, img_
size)(x)
    return net
```

代码中 num_priors * 4 为每个检测头预测 4 个偏移位置，在 num_priors * num_classes 中，num_priors 为当前检测头分配的建议框的数量，num_classes 为预测的 1＋分类数，1 代表有无目标的分类。DefaultBox(min_max_size，ratios，img_size)传入的当前特征图，生成的建议框的尺寸和比例将在预测时输出建议框的详细信息，具体实现见 5.4.3 节。layers.Flatten()操作是为了对多检测头的输出进行合并。box_layer_name 为检测头输出的字典名称。

将上面的操作在 vgg_ssd_300(input_shape，num_classes＝21)中进行整合，就能实现 SSD 的前向传播，代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_SSD_Detected/model/vggssd.py
def vgg_ssd_300(input_shape, num_classes=21):
    """构建 SSD 模型"""
    #输入
    input_tensor = layers.Input(shape=input_shape)
    #输入宽、高
    img_w, img_h = input_shape[0], input_shape[1]
    net = vgg_extra(input_tensor) #vgg+extra layer
    #对 conv4_3 的输入进行归一化
    net['conv_13_norm'] = Normalize()(net['conv_13']) #38 × 38 × 512,即 conv_4_3
    #依赖的 layer,Default BOX 的数量,输出层的名称,[最小尺寸,最大尺寸],[比例]
    detect_layer = {
        'conv_13_norm': [4, 'conv_13_norm', [30, 60], [2], [38, 38]],
        'conv_extra_1': [6, 'fc7_mbox', [60, 111], [2,3], [19, 19]],
        'conv_extra_3': [6, 'conv8_2_mbox', [111, 162], [2, 3], [10, 10]],
        'conv_extra_5': [6, 'conv9_2_mbox', [162, 213], [2,3], [5, 5]],
        'conv_extra_7': [4, 'conv10_2_mbox', [213, 264], [2], [3, 3]],
        'conv_extra_9': [4, 'conv11_2_mbox', [264, 315], [2], [1, 1]],
    }
    #detect_layer 为配置的检测头的 key 名称
    for k, v in detect_layer.items():
        #传入检测头的信息
        pre_head = vgg_detect_head(net[k], v[0], num_classes, v[1], v[2], v[3],
[img_w, img_h])
        net.update(pre_head)
    #将位置合并在一起
    net['mbox_loc'] = layers.Concatenate(axis=1)(
        [net[v[1] + "_loc_flat"] for v in detect_layer.values()]
    )
    #将置信度合并在一起
    net['mbox_conf'] = layers.Concatenate(axis=1)(
        [net[v[1] + "_conf_flat"] for v in detect_layer.values()]
    )
    #将分类合并在一起
    net['mbox_default_box'] = layers.Concatenate(axis=1)(
        [net[v[1] + "_default_box"] for v in detect_layer.values()]
    )

    #location 8732 * 4
    net['mbox_loc'] = layers.Reshape([-1, 4])(net['mbox_loc'])
    #conf 8732 *num_classes
    net['mbox_conf'] = layers.Reshape([-1, num_classes])(net['mbox_conf'])
    #Softmax 会互斥
    net['mbox_conf'] = layers.Activation('softmax')(net['mbox_conf'])
    #将预测值合并在一起,8732 * 33 = 8732 *[4 + 21 + 4 Default BOX+ 4variances]
    net['predictions'] = layers.Concatenate(axis=2)([
        net['mbox_loc'],
        net['mbox_conf'],
        net['mbox_default_box']
    ])
    return Model(inputs=net['input'], outputs=net['predictions'])
```

代码中 detect_layer 格式分别为传入的特征层 net 字典的名称、Default BOX 的数量、输出层的名称、建议框[最小尺寸,最大尺寸]、建议框[比例]、特征层的大小。通过 for k,v in

detect_layer.items()遍历特征层并进行检测头的构建，然后将检测头的输出在 net['mbox_loc']中进行合并，并进行维度的 Reshape，如图 5-29 所示。

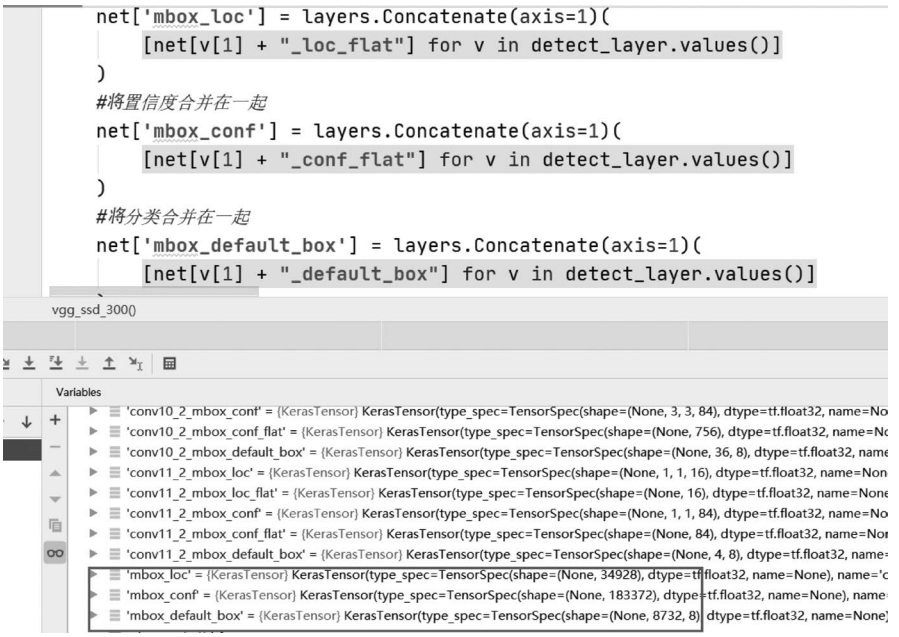



图 5-29　各个检测头输出 Flat

最后对位置、置信度的预测及 Default BOX 进行合并，输出为 8732×33，如图 5-30 所示。

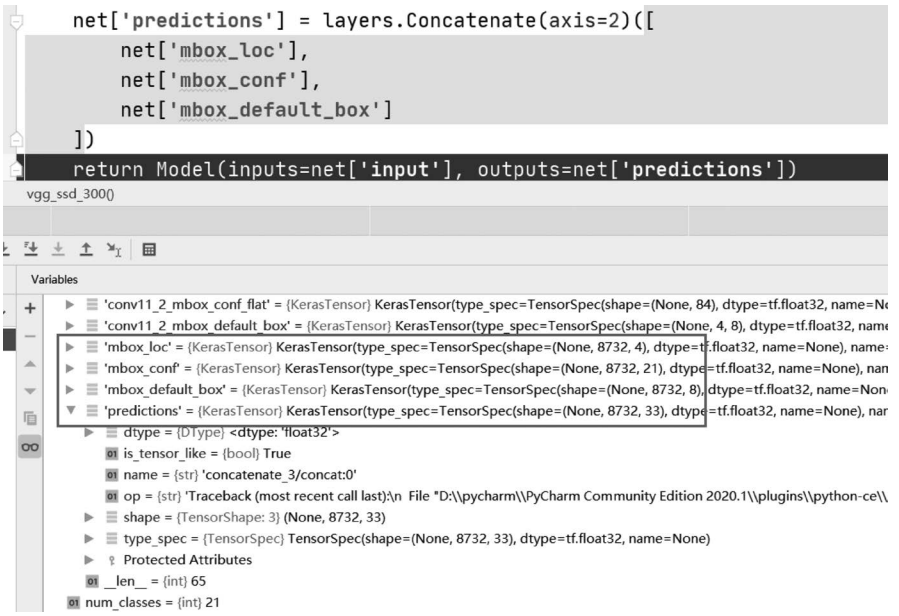



图 5-30　SSD 网络的输出

### 5.4.3 代码实战建议框的生成

代码生成 Default BOX 需要根据表 5-1 中的比例进行计算，假设输入为 38×38×512 的特征层，则比例为 1∶1、1∶$\sqrt{2}$、$\sqrt{2}$∶1、1∶1，然后根据输入的 min_size 和 max_size 生成 Default BOX 的宽和高，然后将原图分为 300/38 份，使用均分指令将原图生成 38×38 的坐标点（将这个坐标点看成特征较上的每个点），在每个坐标点根据计算出来的尺寸，计算 Default BOX 的左上、右下角位置的坐标，详细的代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_SSD_Detected/utils/default_box.py
class BaseDefaultBox(object):
   def __init__(
         self,
         min_max_size: list,                          #Anchor 的最大和最小尺寸
         ratios: list,                                #比例
         img_size: list = [300, 300],                 #原图大小
         variances: list = [0.1, 0.1, 0.2, 0.2],      #缩放尺寸
         **kwargs):
      #属性初始化
      self.variances = np.array(variances)
      self.ratios = aspect_ratios(ratios)
      self.img_size = img_size
      self.min_max_size = min_max_size
      super(BaseDefaultBox, self).__init__(**kwargs)

   def call(self, feature_map, *args, **kwargs):
      #Feature Map 中的 w 和 h
      feature_map_width, feature_map_height = feature_map[0], feature_map[1]
      #原图大小
      img_width, img_height = self.img_size[0], self.img_size[1]
      #存放 Default Box 的 w 和 h
      box_width, box_height = [], []
      #根据 self.ratios 属性选择生成 Anchor 的 w 和 h
      for ar in self.ratios:
         #假设输入为 38×38 的特征，则第 1 个比例为 30∶30
         if ar == 1.0 and len(box_width) == 0:
            box_width.append(self.min_max_size[0])
            box_height.append(self.min_max_size[0])
         elif ar == 1.0 and len(box_width) > 0:
            #第 2 个 1∶1 比例为 sqrt(30 × 60)
            box_width.append(np.sqrt(self.min_max_size[0] *self.min_max_
size[1]))
            box_height.append(np.sqrt(self.min_max_size[0] *self.min_max_
size[1]))
         elif ar != 1.0:
            #第 3 个为 30 *sqrt(2): 30 *1/sqrt(2)，即 1∶2
            #第 4 个为 30 *1/sqrt(2) : 30*sqrt(2) 即 2∶1
            box_width.append(self.min_max_size[0] *np.sqrt(ar))
            box_height.append(self.min_max_size[0] / np.sqrt(ar))
      #求 BOX 中心点
      box_widths = 0.5 *np.array(box_width)
```

```
        box_heights = 0.5 * np.array(box_height)
        #映射到 Feature Map 中的比例,即 300/38=7.8
        step_x = self.img_size[0] / feature_map_width
        step_y = self.img_size[1] / feature_map_height

        #在原图中均分为 38 份,生成每个坐标点
        lin_x = np.linspace(0.5 * step_x, img_width - 0.5 * step_x, feature_map_width)
        lin_y = np.linspace(0.5 * step_y, img_width - 0.5 * step_y, feature_map_height)
        #得到 38 × 38 和 38 × 38 坐标点
        centers_x, centers_y = np.meshgrid(lin_x, lin_y)
        #得到 1444×1 和 1444×1
        centers_x, centers_y = centers_x.reshape(-1, 1), centers_y.reshape(-1, 1)
                                                                    #变成一维
        #每个先验框需要两个(centers_x, centers_y),前一个用来计算左上角,后一个用来
        #计算右下角
        default_box = np.concatenate([centers_x, centers_y], axis=1)  #1444 × 2
        #再复制一份
        num_default_box = len(self.ratios)
        #先沿 x 轴复制 1 倍,再沿 y 轴复制 2 × 4 和 1444 × 16,两个位置预测 xmin,ymin,
        #xmax,ymax,共 4 个锚框
        default_box = np.tile(default_box, (1, 2 * num_default_box))   #1444×16

        #将锚框各个比例的值更新到 Default BOX 中
        default_box[:, 0::4] = default_box[:, 0::4] - box_widths        #xmin
        default_box[:, 1::4] = default_box[:, 1::4] - box_heights       #ymin
        default_box[:, 2::4] = default_box[:, 2::4] + box_widths        #xmax
        default_box[:, 3::4] = default_box[:, 3::4] + box_heights       #ymax

        #转换成浮点数
        default_box[:, ::2] = default_box[:, ::2] / self.img_size[0]
        default_box[:, 1::2] = default_box[:, 1::2] / self.img_size[1]
        #38 × 38 × 4 原比例为(1444,16) = 1444 × 4 个框 × 4 个位置,reshape 之后就是
        #5776 × 4 个位置
        default_box = default_box.reshape([-1, 4])
        #将那些位置信息为负数的值转换成 0
        default_box = np.minimum(np.maximum(default_box, 0.0), 1.0)
        #将 variances 信息复制 Default BOX 这么多份
        variances = np.tile(self.variances, (len(default_box), 1))
        #将 default_box 和 variances 合并
        default_box = np.concatenate([default_box, variances], axis=1)
        return default_box
```

代码中 step_x,step_y 在这里取 7.8,意味着 38×38 的特征图在原图的区域是 7.8,np.linspace(0.5 * step_x,img_width－0.5 * step_x,feature_map_width)表明 Default BOX 的起点从(0,0)偏移了 0.5 * step_x;生成的 BOX 信息归一化后,再通过 default_box=np.concatenate([default_box,variances],axis=1)对缩放因子[0.1,0.1,0.2,0.2]进行了合并。

然后根据 BaseDefaultBox 类的实现,封装 DefaultBox(layers.Layer)算子,使在搭建模型时使用 net[box_layer_name+"_default_box"]=DefaultBox(min_max_size,ratios,img_size)(x)进行调用,并合并到网络层中。DefaultBox(layers.Layer)的代码如下:

```
#第 5 章/ObjectDetection/TensorFlow_SSD_Detected/utils/default_box.py
class DefaultBox(layers.Layer):
   def __init__(
         self,
         min_max_size: list,                    #Anchor 设置的最小尺寸和最大尺寸
         ratios: list,                          #比例
         img_size: list = [300, 300],           #输入图像尺寸
         variances: list = [0.1, 0.1, 0.2, 0.2], #缩放因子
         **kwargs):
      super(DefaultBox, self).__init__()
      #默认框生成类实例化
      self.base = BaseDefaultBox(
         min_max_size,
         ratios,
         img_size,
         variances,
         **kwargs
      )

   def call(self, inputs, *args, **kwargs):
      if hasattr(inputs, '_keras_shape'):
         input_shape = inputs._keras_shape
      elif hasattr(K, 'int_shape'):
         input_shape = K.int_shape(inputs)
      #根据特征图的尺寸调用 self.base 对象,并生成默认框,假设为 38×38,则输出为[5776,8]
      default_box = self.base.call([input_shape[2], input_shape[1]])
      #增维并转换成 Tensor 格式[1,5776,8]
      default_box_tensor = K.expand_dims(tf.cast(default_box, dtype=
tf.float32), 0)
      #在每个 batch_size 中都复制 1 份
      pattern = [tf.shape(inputs)[0], 1, 1]
      #[b,5776,8]
      prior_boxes_tensor = tf.tile(default_box_tensor, pattern)
      return prior_boxes_tensor
```

DefaultBox(layers. Layer)用于获得建议框,接下来就需要在建议框与真实标注框之间做 IOU 的计算,并将 IOU>0.5 的样本设置为正样本,以此计算真实框与建议框的偏移,作为 $y$ 值,代码如下:

```
#第 5 章/ObjectDetection/TensorFlow_SSD_Detected/utils/default_box.py
def assign_boxes(targets, row):
   """
   在图像上生成 Default BOX 及 y 值,构建成 8732×33
   :param targets: GT 格式 xmin,ymin,xmax,ymax,one_hot
   :return:
   """
   #生成一个初始为 0 的 8732×8 的矩阵
   assignment = np.zeros((self.num_priors, 4 + self.num_classes + 4 + 4))
   #是否为背景,默认都为背景
   assignment[:, 4] = 0.
   #如果没有目标,则返回全是 0 的 Tensor
```

```
        if len(targets) == 0: return assignment

        #对真实的 BOX 进行编码
        #在 true_encode_box()函数内实现编码
        encoded_boxes = np.apply_along_axis(true_encode_box, 1,
                                            targets[:, :4],
                                            self.priors_box,
                                            self.iou_overlap_threshold,
                                            True,
                                            row)
        #reshape 成[batch_size,8732,5]
        encoded_boxes = encoded_boxes.reshape(-1, self.num_priors, 5)

        #取重合程度最大的先验框,并且获取这个先验框的 index
        #-1 的位置是 IOU 的得分,即 IOU 最大得分的 BOX 的 index
        best_iou = encoded_boxes[:, :, -1].max(axis=0)
        best_iou_mask = best_iou > 0
        best_iou_idx = encoded_boxes[:, :, -1].argmax(axis=0)
        best_iou_idx = best_iou_idx[best_iou_mask]
        #有物体的先验框的个数
        assign_num = len(best_iou_idx)
        #保留重合程度最大的先验框的应该有的预测结果
        encoded_boxes = encoded_boxes[:, best_iou_mask, :]

        #把有物体的 BOX 更新到 assignment 中[8732, 4 + self.num_classes + 4 + 4]
        assignment[:, :4][best_iou_mask] = encoded_boxes[best_iou_idx, np.arange
    (assign_num), :4]
        assignment[:, 4][best_iou_mask] = 1. #4 为背景的概率,当然为 0;损失要求有样本是 1
        assignment[:, 5:-8][best_iou_mask] = targets[best_iou_idx, 4:]   #one_hot
        return assignment
```

在函数 assign_boxes()中实现了真实框与建议框进行 IOU 的计算并获得正样本的代码,其中 np.apply_along_axis 为关键代码,即将 targets[:, :4]真实框的位置信息和 self.priors_box 建议框信息,根据 self.iou_overlap_threshold 的阈值 0.5 在 true_encode_box()函数中进行编码操作,然后将 true_encode_box()得到的正样本信息赋值到 assignment 中,包括位置 assignment[:, :4][best_iou_mask]=encoded_boxes[best_iou_idx,np.arange(assign_num), :4]、有无目标 assignment[:, 4][best_iou_mask]=1、分类信息 assignment[:, 5:-8][best_iou_mask]=targets[best_iou_idx, 4:]。

计算真实框与建议框的偏移在 true_encode_box()函数中实现,其核心仍然是调用式(5-3)完成,代码如下:

```
#第 5 章/ObjectDetection/TensorFlow_SSD_Detected/utils/default_box.py
def true_encode_box(
        true_box,                                   #真实 BOX
        priors_box,                                 #Default BOX
        overlap_threshold=0.5,                      #阈值
        is_there_any_object=True,
                row=""
```

```
    ):
        #对 GT BOX 进行>0.5 选择为正样本
        iou_score = iou(priors_box, true_box)
        #将 Tensor 初始为 8732 × 5
        encoded_box = np.zeros([len(priors_box), 4 + is_there_any_object]) #[8732,5]
        #将先验框得到的 IOU 与阈值进行对比
        assign_mask = iou_score >= overlap_threshold
        if not assign_mask.any():
            assign_mask[iou_score.argmax()] = True
        if is_there_any_object:
            #将 iou_score 大于阈值的 BOX 的得分设置到 encoded_box 中,即有物体的概率
            encoded_box[:, -1][assign_mask] = iou_score[assign_mask]
        #得到有物体的 BOX 位置信息
        assigned_priors = priors_box[assign_mask] #[any_object_total_num_priors, 8]
        #因为输入的格式为 xmin,ymin,xmax,ymax,所以要转换为中心点来计算
        box_center = 0.5 * (true_box[:2] + true_box[2:])    #xmin,ymin + xmax,ymax
        box_wh = true_box[2:] - true_box[:2]                #xmax,ymax - xmin,ymin

        #计算先验框的中心点
        assigned_priors_center = 0.5 * (assigned_priors[:, :2] + assigned_priors[:, 2:4])
        assigned_priors_wh = assigned_priors[:, 2:4] - assigned_priors[:, :2]

        #encoded_box 为[8732, 5],先验框与真实框 IOU 大于 0.5 的,真实框与先验框中心点的偏移值
        encoded_box[:, :2][assign_mask] = box_center - assigned_priors_center
        #dx 和 dy,代入公式进行计算
        encoded_box[:, :2][assign_mask] /= assigned_priors_wh
        #variances 是[0.1,0.1,0.2,0.2],[-4,-2]就是[0.1,0.1]
        encoded_box[:, :2][assign_mask] /= assigned_priors[:, -4:-2]
        #dw 和 dh 的偏移值,代入 log 函数进行计算
        encoded_box[:, 2:4][assign_mask] = np.log(box_wh / assigned_priors_wh)
        #[-2:]就是[0.2,0.2]
        encoded_box[:, 2:4][assign_mask] /= assigned_priors[:, -2:]
        #encoded_box 为[8732,5]
        return encoded_box.ravel()
```

代码中 assigned_priors[:, −4:−2]为 variances 参数中的[0.1,0.1],assigned_priors[:, −2:]为 variances 参数中的[0.2,0.2],在编码时这里对值进行了放大。

然后新建 DataProcessingAndEnhancement(object)类,并创建 generate(self,isTraining = True)方法,使训练时得到 $x$ 的图片数据 img,$y$ 的数据(真实框与建议框的偏移及分类信息),代码如下:

```
#第 5 章/ObjectDetection/TensorFlow_SSD_Detected/data/data_processing.py
class DataProcessingAndEnhancement(object):
    def generate(self, isTraining=True):
        while True:
            if isTraining:
                shuffle(self.train_lines)
                lines = self.train_lines
            else:
                lines = self.val_lines
```

```
            inputs, targets = [], []
            if len(lines):
               rnd_row = random.choice(lines)
            else:
               rnd_row = None
            label_result = {x: 0 for x in range(self.num_classes - 1)}
            for row in lines:
               #读图片和 BOX 的数据
               img, y = self.get_image_processing_results(row, rnd_row, isTraining)

               if len(y) != 0:
                   boxes = np.array(y[:, :4], dtype=np.float32)
                   #BOX 归一化
                   boxes[:, 0] = boxes[:, 0] / self.input_shape[1]   #xmin
                   boxes[:, 1] = boxes[:, 1] / self.input_shape[0]   #ymin
                   boxes[:, 2] = boxes[:, 2] / self.input_shape[1]   #xmax
                   boxes[:, 3] = boxes[:, 3] / self.input_shape[0]   #ymax
                   #构建 one_hot 编码
                   one_hot_label = np.eye(self.num_classes - 1)[np.array(y[:, 4],
np.int32)]                                                          #[[1,0],[0,1]]
                   if ((boxes[:, 3] - boxes[:, 1]) <= 0).any() and ((boxes[:, 2] -
boxes[:, 0]) <= 0).any():
                       continue
                   #GT 格式 xmin,ymin,xmax,ymax,[0,1]
                   y = np.concatenate([boxes, one_hot_label], axis=-1)
               y = self.assign_boxes(y, row)          #将 y 值统一到 8732×8 这个维度,
                                                      #并且是编码后的内容
               inputs.append(img)
               targets.append(y)

               #按 batch_size 传输 targets
               if len(targets) == self.batch_size:
                  tmp_inp = np.array(inputs, dtype=np.float32)
                  tmp_targets = np.array(targets)
                  inputs = []
                  targets = []
                  #注释以下语句,就可以进行调试
                  yield preprocess_input(tmp_inp), tmp_targets
```

代码中 self.assign_boxes()实现将 $y$ 值统一到 $8732\times33$ 这个维度，np.concatenate([boxes,one_hot_label],axis=－1)增加了真实的分类 one_hot 编码值，if len(targets)==self.batch_size 成立时返回指定 batch_size 的数据。

## 5.4.4　代码实战损失函数的构建及训练

根据式(5-6)实现 SSD 的损失函数的构建，代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_SSD_Detected/utils/loss.py
class MultiAngleLoss(object):
   """
   SSD 损失函数的构建
```

```
    """
    def __init__(self,
                 num_classes=21,
                 negative_sample_ratio=3.0,
                 difficult_sample_top=300,
                 total_regression_loss_ratio=1.,
                 total_classification_loss_ratio=1.0
                 ):
        """
        :param num_classes: 类别数
        :param negative_sample_ratio: 负样本：正样本的比例,SSD 默认为 3：1
        :param difficult_sample_top: 当一个正样本都没有时设置负样本挖掘数量
        :param total_regression_loss_ratio: 最后回归损失的比例
        :param total_classification_loss_ratio: 最后分类损失的比例
        """
        super(MultiAngleLoss, self).__init__()
        self.num_classes = num_classes - 1
        self.negative_sample_ratio = negative_sample_ratio
        self.difficult_sample_top = difficult_sample_top
        self.total_regression_loss_ratio = total_regression_loss_ratio
        self.total_classification_loss_ratio = total_classification_loss_ratio
    def compute_loss(self, y_true, y_pred):
        batch_size = tf.shape(y_true)[0]
        num_boxes = tf.cast(tf.shape(y_true)[1], tf.float32)   #8732

        #conf_loss[0][conf_loss[0]!=0],每个框的损失
        conf_loss = self._softmax_loss(y_true[:, :, 4:-8], y_pred[:, :, 4:-8])
        #每个框的回归损失
        loc_loss = self._l1_smooth_loss(y_true[:, :, :4], y_pred[:, :, :4])
        #统计不为背景的 BOX 的数量
        true_box_num = tf.reduce_sum(y_true[:, :, 4], axis=-1)

        #每个框的回归损失,然后求和
        regression_loss_per_box = tf.reduce_sum(loc_loss *y_true[:, :, 4], axis=1)
        #每个框的分类损失,然后求和
        classification_loss_of_each_box = tf.reduce_sum(conf_loss *y_true[:, :, 4],
axis=1)
        #进行正负样本比例的调节,负样本是正样本的 3 倍
        select_negative_samples_num = tf.minimum(self.negative_sample_ratio *
true_box_num, num_boxes - true_box_num)
        #select_negative_samples_num,有可能一个也没有,因为 true_box_num 可能为 0,
        #即一个正样本也没有
        #与 0 进行比较,得到[True,True,...]或者[True,True,...,False]的情况
        negative_samples_num_mask = tf.greater(select_negative_samples_num, 0)
        #tf.reduce_any 在张量的维度上计算元素的 "逻辑或",如果所有的结果为 False,
        #则 has_min 为 0;只要有一个为 True,则为 1
        #如果图片中没有一个正样本,则 select_negative_samples_num 为 0,也就没有负样本
        #但是这时应该有很多负样本,所以我们要处理一下
        is_a_negative_sample = tf.cast(tf.reduce_any(negative_samples_num_mask),
tf.float32)

        select_negative_samples_num = tf.concat(axis=0, values=[
            select_negative_samples_num,   #如果一个正样本都没有,则此项为 0
```

```
            [(1 - is_a_negative_sample) *self.difficult_sample_top]
                                        #那么此时补 difficult_sample_top;
        ])
        #求本次 batch 中负样本数的平均数
        batch_avg_select_negative_samples_num = tf.reduce_mean(
            tf.boolean_mask(
                select_negative_samples_num,
                tf.greater(select_negative_samples_num, 0)
            )
        )
        #将负样本数转换为 int32 数据类型
        batch_avg_select_negative_samples_num = tf.cast(batch_avg_select_
negative_samples_num, tf.int32

        #取出来预测的最大的分类损失
        pre_max_confs = tf.reduce_max(y_pred[:, :, 5:-8], axis=2)
        #得到 true label 中负样本的置信度损失的下标
        _, top_negative_sample_loss_index = tf.nn.top_k(
            pre_max_confs *(1 - y_true[:, :, 4]),
            k=batch_avg_select_negative_samples_num
        )
        #获取难例样本的损失
        #tf.gather 根据难例样本的下标取 conf_loss 中的损失
         negative_sample_loss = tf.gather(tf.reshape(conf_loss, [-1]), top_
negative_sample_loss_index)
        #对难例样本的损失求总数
        total_negative_sample_loss = tf.reduce_sum(negative_sample_loss, axis=1)

        #如果为真,则为 x,否则为 y
        true_box_num = tf.where(tf.not_equal(true_box_num, 0), true_box_num, tf.
ones_like(true_box_num))
        total_true_box_num = tf.reduce_sum(true_box_num)
        total_conf_loss = tf.reduce_sum(classification_loss_of_each_box) + tf.
reduce_sum(total_negative_sample_loss)
        #总分类损失 = (正样本的损失 + 难例样本的损失) / (正样本数 + 难例样本数)
        #total_conf_loss = total_conf_loss / (total_batch_select_negative_
        #samples_num + total_true_box_num)
        #N 在论文中取正样本的个数
        total_conf_loss = total_conf_loss / (0 + total_true_box_num)
        #总回归损失 = 回归损失 / 正样本数
        total_loc_loss = tf.reduce_sum(regression_loss_per_box) / total_true_box_num
        #总损失 = 总分类损失 + alpha *总回归损失,论文中 alpha 是 1
        total_loss = \
            self.total_classification_loss_ratio *total_conf_loss + self.total_
regression_loss_ratio *total_loc_loss
        return total_loss
```

损失函数的构建,主要在 compute_loss(self,y_true,y_pred)中实现,传入 y_true 为真实框,即上文 assign_boxes(targets,row)函数中处理过的编码值 8732×33,y_pred 预测值也为 8732×33。self._softmax_loss(y_true[:,:,4:-8],y_pred[:,:,4:-8]),4:-8 为置信度+分类的位置,此处使用交叉熵损失。self._l1_smooth_loss(y_true[:,:,:4],y_pred[:,:,:4])为回归 Smooth 损失。在 true_box_num=tf.reduce_sum(y_true[:,:,4],axis=-1)中 4 这个位

置为置信度，因为只有为 1 时才代表有物体，所以 true_box_num 为有目标的正样本的数量。tf.reduce_sum(loc_loss * y_true[:,:,4],axis=1)只对有目标的损失进行求和运算。

损失函数的第 2 部分是实例难例样本的挖掘，即根据正样本的数量，求 3 倍负样本的损失值。batch_avg_select_negative_samples_num 为考虑特殊情况后的负样本的数量。pre_max_confs=tf.reduce_max(y_pred[:,:,5:-8],axis=2)取出预测的分类概率，pre_max_confs * (1-y_true[:,:,4])中因为(1-y_true[:,:,4])得到的是没有目标的分类，所以整个语句得到的就是没有目标但是分类预测最大的概率，并通过 tf.nn.top_k()获得其难例样本的概率排前 $k$ 的索引下标 top_negative_sample_loss_index，然后根据 negative_sample_loss=tf.gather(tf.reshape(conf_loss,[-1]),top_negative_sample_loss_index)获得置信度和分类的损失，求和后将正样本、难例样本的损失相加得到 total_conf_loss，最后实现位置损失+正样本分类损失+负样本分类损失。

训练代码调用 model.fit()函数，将相关参数传入即可，主要代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_SSD_Detected/train/main.py
    #构建模型
    model = vgg_ssd_300(input_shape, num_classes)
    model.build([1, 300, 300, 3])
    model.compile(
        optimizer=Adam(learning_rate=init_lr),
        loss=MultiAngleLoss(num_classes).compute_loss,
        run_eagerly=True,     #是否启用调试模型
    )
    model.fit(
        data_object.generate(True),
        steps_per_epoch=num_train                    //BATCH_SIZE,
        validation_data=data_object.generate(False),
        validation_steps=num_val                     //BATCH_SIZE,
        epochs=end_EPOCH,
        initial_epoch=init_EPOCH,
        callbacks=[logging, checkpoint]
    )
```

更多更详细的代码可参考随书代码。

### 5.4.5 代码实战预测推理

SSD 的推理流程包括加载模型、加载推理图片进行置信度阈值过滤、偏移值解码，以及 NMS 非极大值抑制等操作，详细的代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_SSD_Detected/detected/detected.py
class Detected():
   def __init__(self, model_path, input_size):
      self.model = load_model(
         model_path,
         custom_objects={'compute_loss': MultiAngleLoss(3).compute_loss}
      ) #读取模型和权重
```

```
        self.confidence_threshold = 0.5                          #有无目标置信度
        self.class_prob = [0.5, 0.5]                             #两个类别的分类阈值
        self.nms_threshold = 0.5                                 #NMS 的阈值
              self.input_size = input_size

    def readImg(self, img_path=None):
        img = cv2.imread(img_path)                               #读取要预测的图片
        #将图片转到 300×300
        self.img, _ = u.letterbox_image(img, self.input_size, [])
        self.old_img = self.img.copy()

    def forward(self):
        #升成 4 维
        img_tensor = tf.expand_dims(self.img, axis=0)
        #前向传播
        self.output = self.model.predict(img_tensor)

    def confidence_filtering(self):
        #根据置信度的阈值进行过滤,如果大于>0.5,则为有目标
        self.targeted = self.output[self.output[..., 4] > self.confidence_threshold]

    def classification_filtering(self):
        #根据类别的概率过滤
        for i, p in enumerate(self.class_prob):
            #第 i 个类别
            classification = self.targeted[self.targeted[..., 5 + i] > p]
            if len(classification):
                #偏移值
                offset_box = classification[..., :4]
                #置信度
                confidence = classification[..., [4]]
                #default box
                default_box = classification[..., -8:-4]
                #缩放因子
                variances = classification[..., -4:]
                #对该类别的框进行解码
                #因为 Default BOX 为 xmin,ymin,xmax,ymax,所以需要转换成 cx,cy,w,h
                default_box = u.xyxy2cxcywh(default_box)
                boxes = u.offset2xyxy(offset_box, default_box, variances)
                boxes = np.concatenate([boxes, confidence], axis=-1)
                #NMS 得到的索引
                index = u.nms(boxes, nms_thresh=self.nms_threshold)
                #根据索引取出 boxes 信息
                boxes = boxes[index]
                #绘图
                self.old_img = u.draw_box(self.old_img, boxes)
                u.show(self.old_img)
if __name__ == "__main__":
    d = Detected('../weights')                                   #读权重
    d.readImg('../../val_data/pexels-photo-5211438.jpeg')        #预测图片
    d.forward()                                                  #前向传播
    d.confidence_filtering()                                     #置信度过滤
d.classification_filtering()                                     #分类过滤并进行解码,NMS
```

在 Detected 类中按功能进行了区分，readImg()用来读取图片，并调用 letterbox_image 转换成 SSD 的 300×300 大小，然后在 forward()中进行前向传播以得到预测值，在 confidence_filtering()中进行置信度阈值的过滤，将大于 0.5 的 self.targeted 传递给 classification_filtering()进行类别的概率过滤，将满足类别概率的 boxes 进行解码 u.offset2xyxy 和 u.nms 非极大值抑制操作，从而得到最终预测目标。

**总结**

SSD 有 6 个检测头，每个检测头分配不同的先验框，共计分配 8732 个先验框进行预测，具有多尺度、密集锚框检测的特点。在损失函数构建中，采用了难例样本挖掘的技术。

**练习**

运行并调试本节代码，理解算法的设计与代码的结合，重点梳理本算法的实现方法。

## 5.5 单阶段速度快的检测网络 YOLOv1

### 5.5.1 模型介绍

YOLOv1 由 Joseph Redmon 等在 2015 年发表的论文 *You Only Look Once: Unified, Real-Time Object Detection* 中提出，是一个单阶段目标检测网络，其主要特点为速度快。YOLOv1 的主要结构如图 5-31 所示。

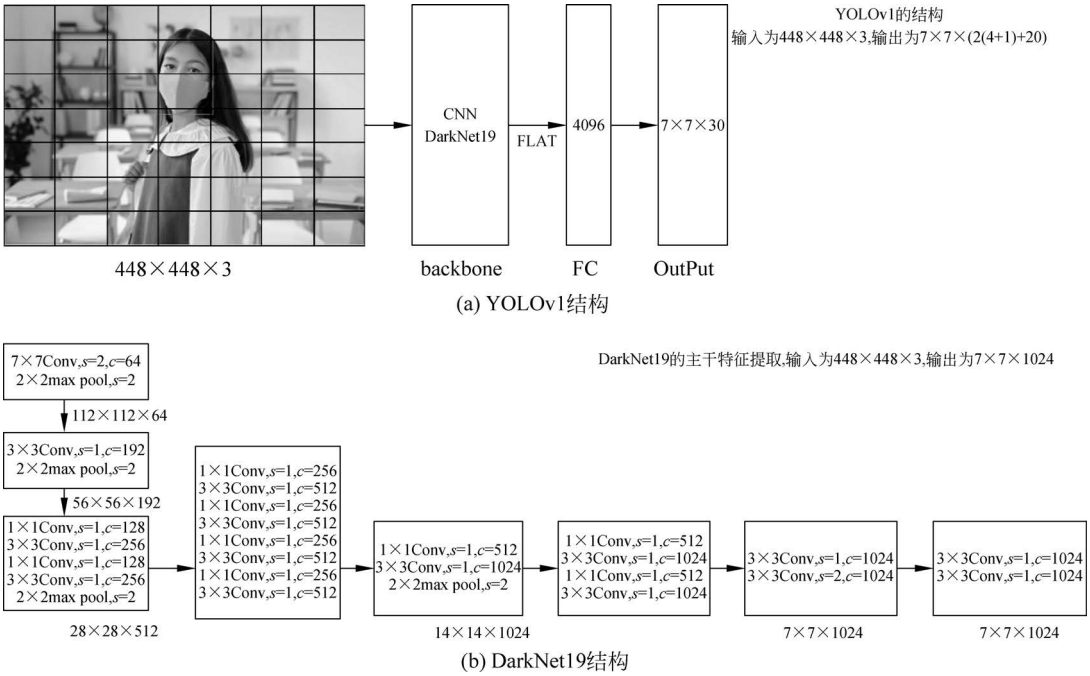



图 5-31 YOLOv1 网络结构

YOLOv1 其主干特征提取网络由 DarkNet19 构成，DarkNet19 的主要特点是由 3×3 和 1×1 卷积构成，共计 8 个块，提取到最后的特征层为 7×7×1024，然后将 7×7×1024 摊平送入 4096 维的全连接层，预测输出为 7×7×(2×(4 个位置+1 有无目标的概率)+20 个类别的概率)=7×7×30。将 7×7×30 中的特征映射到原图就相对于将原图划分成 7×7 个格子，每个格子对应到输出就是每个特征点，这个特征点有 30 个通道，代表预测 30 个特征的输出。

YOLOv1 没有建议框(先验框)，由真实标注物体所在格子进行预测，也就是标注物体所在格子为正样本，每个标注物体所在格子最多预测两个物体，所以 YOLOv1 一共只能预测 7×7×2=98 个目标，而没有标注物体的格子全都是负样本，如图 5-32 所示。

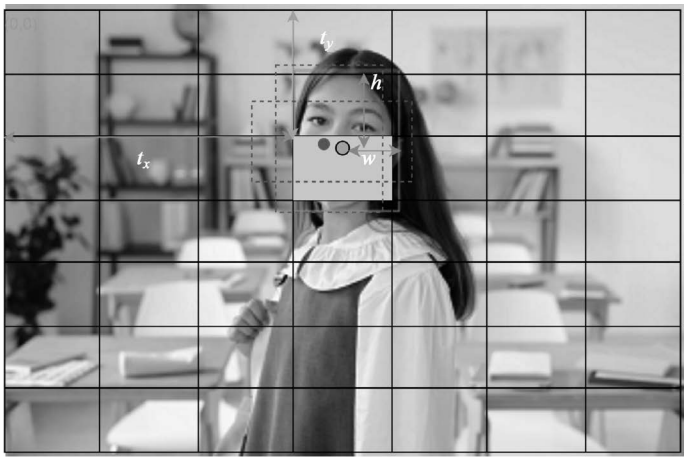



图 5-32　YOLOv1 正样本的选择

图 5-32 中口罩所处中心点为(center_x,center_y)，其所在格子的左上角行 $i=3$、列 $j=2$，所以 $t_x$=center_x−$i$，$t_y$=center_y−$j$，则 $t_x$ 和 $t_y$ 为相对于原点(0,0)的偏移，$w$、$h$ 为标签目标的宽和高，所以 y_true 为 $t_x$，$t_y$，$w$，$h$；y_true 只在高亮背景中有目标，其他格式都没有目标，所以当前格子的置信度为 1，其他为 0，当前格子也负责预测分类的概率。

预测值 y_pred 也为 $t_x$，$t_y$，$w$，$h$，每个格子预测两个偏移值，也就是两个预测框，每个预测框预测前背景、分类的概率。在求损失时，只有高亮背景的格子为正样本，其他格子全是负样本，当预测值 y_true 与真实框 y_pred 很接近时，说明网络预测接近目标。需要注意的是 YOLOv1，宽和高 $wh$ 是直接预测，$t_x$、$t_y$ 是相对于(0,0)点的偏移。没有设置建议框的这种方法被称为 Anchor Free。

在损失函数方面，位置损失、置信度损失(有无目标)、分类损失均使用均方差，其公式如下：

$$\text{Loss}=\lambda_{\text{coord}}\sum_{i=0}^{s^2}\sum_{j=0}^{B}1_{ij}^{\text{obj}}[(x_i-\hat{x}_i)^2+(y_i-\hat{y}_i)^2]+$$

$$\lambda_{\text{coord}}\sum_{i=0}^{s^2}\sum_{j=0}^{B}1_{ij}^{\text{obj}}[(\sqrt{w_i}-\sqrt{\hat{w}_i})^2+(\sqrt{h_i}-\sqrt{\hat{h}_i})^2]+$$

$$\sum_{i=0}^{s^2}\sum_{j=0}^{B}1_{ij}^{obj}[(C_i-\hat{C}_i)^2]+\lambda_{noobi}\sum_{i=0}^{s^2}\sum_{j=0}^{B}1_{ij}^{noobj}[(C_i-\hat{C}_i)^2]+$$

$$\sum_{i=0}^{s^2}1_{ij}^{obj}\sum_{c\in 1}[(p_i(c)-\hat{p}_i(c))^2] \tag{5-7}$$

其中，$\lambda_{coord}$ 为平衡系数，原作者将其设置为 5。$1_{ij}^{obj}$ 代表有物体的格子。$x_i$ 代表真实 tx，$\hat{x}_i$ 代表预测 tx。$\sqrt{w_i}$ 为真实宽，$\sqrt{\hat{w}_i}$ 为预测宽，由于 $w$、$h$ 的值较大，所以开方是为了防止梯度爆炸。$1_{ij}^{noobj}$ 代表没有物体，所以置信度损失由有物体的损失＋没有物体的损失构成，而没有物体的损失占比应该更小，所以超参数 $\lambda_{noobi}=0.5$，最后 $p_i(c)$ 为真实物体的分类，$\hat{p}_i(c)$ 为预测物体的分类。ij 代表每个格子。整个网络的实现，可参见 5.5.2 节。

## 5.5.2 代码实战模型搭建

YOLOv1 模型前向传播的代码如下：

```
#第 5 章/ObjectDetection/TesnsorFlow_YOLO_V1_Detected/backbone/yolo_v1.py
def yolo_v1(input_shape, CLASS_NUM=20, BOX_NUM=2, GRID_NUM=7):
    #输入
    input_tensor = layers.Input(shape=input_shape)
    #默认不启用 BN
    is_bn = False
    #DarkNet19 的结构,Conv 代表卷积,MaxP 即最大池化
        #[算子类型,核,步长,channel]
    darknet_config = [
        ['Conv', 7, 2, 64],
        ['MaxP', 2, 2],                 #112 × 112 × 64

        ['Conv', 3, 1, 192],
        ['MaxP', 2, 2],                 #56 × 56 × 192

        ['Conv', 1, 1, 128],
        ['Conv', 3, 1, 256],
        ['Conv', 1, 1, 256],
        ['Conv', 3, 1, 512],
        ['MaxP', 2, 2],                 #28 × 28 × 512

        ['Conv', 1, 1, 256],
        ['Conv', 3, 1, 512],
        ['Conv', 1, 1, 256],
        ['Conv', 3, 1, 512],
        ['Conv', 1, 1, 256],
        ['Conv', 3, 1, 512],
        ['Conv', 1, 1, 256],
        ['Conv', 3, 1, 512],

        ['Conv', 1, 1, 512],
        ['Conv', 3, 1, 1024],
        ['MaxP', 2, 2],                 #14 × 14 × 1024
```

```
        ['Conv', 1, 1, 512],
        ['Conv', 3, 1, 1024],
        ['Conv', 1, 1, 512],
        ['Conv', 3, 1, 1024],

        ['Conv', 3, 1, 1024],
        ['Conv', 3, 2, 1024],                    #7 × 7 × 1024

        ['Conv', 3, 1, 1024],
        ['Conv', 3, 1, 1024],                    #7 × 7 × 1024
    ]
    x = input_tensor
    #解析配置文件,生成模型结构
    for i, c in enumerate(darknet_config):
        if c[0] == 'Conv':
            x = ConvBnLeakRelu(c[3], c[1], c[2], is_bn=is_bn, padding='same', is_
relu=True)(x)
        elif c[0] == 'MaxP':
            x = layers.MaxPooling2D(c[1], c[2], padding='same')(x)
    x = layers.Flatten()(x)                      #7 × 7 × 1024
    #全连接后激活函数使用 LeakyReLU
    x = layers.LeakyReLU(0.1)(layers.Dense(4096)(x))
    x = layers.DropOut(0.5)(x)                   #防止过拟合
    #输出为 1470 维特征向量
    x = layers.Dense(GRID_NUM *GRID_NUM *(5 *BOX_NUM + CLASS_NUM))(x)
    x = layers.DropOut(0.5)(x)                   #防止过拟合
    x = tf.sigmoid(x)                            #Sigmoid 是为了将值域控制在 0~1
    #将 1470 维特征向量变成 7×7×30,7×7 是把原图像划为 7×7 个格子和(4+1)×2+20 个类别
    x = tf.reshape(x, [-1, GRID_NUM, GRID_NUM, 5 *BOX_NUM + CLASS_NUM])
    return Model(inputs=input_tensor, outputs=x)
```

特征提取使用配置文件 darknet_config 来构建 DarkNet19,具体实现在 for 循环中。ConvBnLeakRelu()即 Conv→BN→LeakRelu 的封装。通过 DarkNet19 后 layers.Flatten()得到 50 176 个神经元,然后输出 layers.Dense(GRID_NUM * GRID_NUM * (5 * BOX_NUM+CLASS_NUM))(x)变成 1470 维特征,对这 1470 维特征归一化后 reshape 成 7×7×30,即每个特征点代表每个格子,每个格子预测(4+1)×2 个框和有无目标,并预测 20 个分类的概率。

### 5.5.3　无建议框时标注框编码

虽然 YOLOv1 为 Anchor Free,但仍然需要将标注框编码成 tx,ty,w,h 作为 y_true,以便与预测 y_pred 做损失,代码如下:

```
#第 5 章/ObjectDetection/TesnsorFlow_YOLO_V1_Detected/utils/tools.py
def yolo_v1_true_encode_box(true_boxes, CLASS_NUM=20, BOX_NUM=2, GRID_NUM=7):
    """
    将图片编码成 YOLOv1 的输出格式
    :param GRID_NUM:
    :param BOX_NUM: 默认为两个
```

```
    :param CLASS_NUM: 预测的类别数
    :param true_boxes: NumPy类型[center_x,center_y,w,h,object] YOLO格式
    :return: 7 × 7 × (5 × 2 + 20)
    """
    #初始7×7×30为0的Tensor
    target_box = np.zeros([GRID_NUM, GRID_NUM, (5 *BOX_NUM + CLASS_NUM)])
    #cell_size = 1.0 /7 每个格子的宽、高,归一化
    cell_size = 1.0 / GRID_NUM
    if len(true_boxes) == 0:
        return target_box
    boxes_wh = true_boxes[:, 2:4]                    #gt wh
    boxes_cxy = true_boxes[:, :2]                    #gt cx,cy
    box_label = true_boxes[:, -1]                    #gt label
    for ibox in range(true_boxes.shape[0]):
        center_xy, wh, label = boxes_cxy[ibox], boxes_wh[ibox], int(box_label[ibox])
        #得到1/ S=7格子中的相对位置。-1.0是为了排除当前格子
        ij = np.ceil(center_xy / cell_size) - 1.0
        i, j = int(ij[0]), int(ij[1])                #第几个格子中
        #由第i、第j个格子的中心点进行预测
        grid_xy = ij *cell_size                      #获得格子的左上角xy
        #(bbox中心坐标 - 网络左上角的坐标) / 网格大小 = tx,ty
        grid_p_center_xy = (center_xy - grid_xy) / cell_size
        for k in range(BOX_NUM):
            s = 5 *k
            target_box[j, i, s:s + 2] = grid_p_center_xy
            target_box[j, i, s + 2:s + 4] = wh
            target_box[j, i, s + 4] = 1.0            #置信度,有没有物体
        #i,j正样本格子的label
        target_box[j, i, 5 *BOX_NUM + label] = label
    return target_box #输出类型为 7 × 7 × 30,与预测的结果保持一致
```

代码中 cell_size=1.0/GRID_NUM,1/7=0.14 为每个格子的大小。true_boxes 传进来时就是 center_x,center_y,w,h。ij=np.ceil(center_xy/cell_size)−1.0,$i=3$、$j=2$ 即标注框的中心点落在此格子中(见图 5-32),那么在计算损失时正样本为 $i=3$、$j=2$,其他样本均为负样本。

在得到 $i=3$、$j=2$ 后,计算当前格子的左上角 xy 的坐标 grid_xy=ij * cell_size,然后通过(center_xy−grid_xy)/cell_size 得到标注中心点相对于(0,0)点的偏移。target_box[j,i,s:s+2]=grid_p_center_xy,即 target_box[2,3,0:2]=grid_p_center_xy 为 tx,ty 的偏移值,target_box[2,3,2:4]=wh 为 wh 的宽和高。因为每个格子预测两个物体,所以当 $k=1$ 时,target_box[2,3,5:7]=grid_p_center_xy,也就是当前格子赋两个相同值作为 y_true。

然后再次编写 generate(self,isTraining=True)函数实现,数据的传输,代码如下:

```
#第5章/ObjectDetection/TesnsorFlow_YOLO_V1_Detected/data/data_processing.py
class DataProcessingAndEnhancement(object):
    def generate(self, isTraining=True):
            while True:
        if isTraining:
            shuffle(self.train_lines)
```

```
                lines = self.train_lines
            else:
                lines = self.val_lines
            inputs, targets = [], []
            if len(lines):
                rnd_row = random.choice(lines)
            else:
                rnd_row = None

            for row in lines:
                #读取 Image 和 BOX 信息
                img, y = self.get_image_processing_results(row, rnd_row, isTraining)
                #将 Voc 格式转换为 YOLO 格式
                y = self.voc_label_convert_to_yolo(y)
                #将 GT BOX 编码成与预测值相同的格式
                y = yolo_v1_true_encode_box(y, self.CLASS_NUM, self.BOX_NUM,
self.GRID_NUM)
                #存储图片和编码后的 BOX 信息
                inputs.append(img)
                targets.append(y)
                #按 batch_size 传输 targets
                if len(targets) == self.batch_size:
                   tmp_inp = np.array(inputs, dtype=np.float32)
                   tmp_targets = np.array(targets)
                   inputs = []
                   targets = []
                   #注释以下语句,就可以进行调试了
                   yield preprocess_input(tmp_inp), tmp_targets
```

因为本数据集的默认格式为 Voc,所以需要调用 self. voc_label_convert_to_yolo(y)实现由 Voc 格式转 YOLO 格式,代码类似 5.1 节标签处理及代码,更多更详细的内容可参考随书代码。

### 5.5.4　代码实现损失函数的构建及训练

第 1 步,根据式(5-7)实现正样本格子 $i$、$j$ 位置的损失、置信度的损失及分类概率的损失,同时对 1 负样本给予较小权重的损失,详细的代码如下:

```
#第 5 章/ObjectDetection/TesnsorFlow_YOLO_V1_Detected/utils/loss.py
class MultiAngleLoss(object):
   def __init__(self, CLASS_NUM=20, BOX_NUM=2, GRID_NUM=7, coord=5.0, no_obj=0.5):
       """
       YOLOv1 的 loss 计算
       """
       self.CLASS_NUM = CLASS_NUM                              #类别数
       self.BOX_NUM = BOX_NUM                                  #每个格子的预测数
       self.GRID_NUM = GRID_NUM                                #一共有多少个格子
       self.coord = coord                                      #坐标损失的系数
       self.no_obj = no_obj                                    #不包含物体的损失系数
       self.output_dim = 5 *self.BOX_NUM + self.CLASS_NUM      #输出维度,即 30
```

```
        super(MultiAngleLoss, self).__init__()

    def compute_loss(self, y_true, y_pred):
        """
        计算损失的函数
        :param y_true:
        :param y_pred:
        :return:
        """
        #(1)第 1 部分,获取有物体和没有物体的 mask,有物体为 true,没有物体为 false
        batch_size = tf.shape(y_true)[0]
        #true 值中有物体的框
        get_object_mask = y_true[:, :, :, 4] > 0
        #true 值中没有物体的框
        get_no_object_mask = y_true[:, :, :, 4] == 0
        #(2)第 2 部分,根据第 1 部分获得的 mask,从预测值获取有物体 get_pre_object_mask、
        #没有物体的 get_pre_no_object_mask
        #及有物体的 bbox_pre、class_pre

        #扩维成 b,7,7,30,好获得整组的值
        get_object_mask = tf.tile(np.expand_dims(get_object_mask, -1), [1, 1, 1,
self.output_dim])
        get_no_object_mask = tf.tile(np.expand_dims(get_no_object_mask, -1), [1,
1, 1, self.output_dim])

        #从预测框中获得置信度框的内容,y_pred[get_object_mask] get_object_mask
        #有物体的 mask
        get_pre_object_mask = tf.reshape(y_pred[get_object_mask], [-1, self.
output_dim])
        #获取两个预测框的值,因为预测输出为 2×(4+1)+20
        bbox_pre = tf.reshape(get_pre_object_mask[..., :5 *self.BOX_NUM], [-1, 5])
        #类别信息的值
        class_pre = get_pre_object_mask[..., 5 *self.BOX_NUM:]

        #获取没有物体的格子 y_pred[get_no_object_mask] get_no_object_mask 没有物体
        #的 mask
        get_pre_no_object_mask = tf.reshape(y_pred[get_no_object_mask], [-1,
self.output_dim])
        #(3)第 1 部分只是获得置信度的 mask,接下来需要根据 mask 获取标注物体为正样本的
        #get_true_object_mask,bbox_true
        #及 class_true,和没有物体的 get_true_no_object_mask

        #y_true[get_object_mask],有物体的 mask。get_true_object_mask 为标注框,
        #有物体的部分
        get_true_object_mask = tf.reshape(y_true[get_object_mask], [-1, self.
output_dim])
        #标注框,有物体的 BOX 信息
        bbox_true = tf.reshape(get_true_object_mask[..., :5 *self.BOX_NUM], [-1, 5])
        #标注框,有物体的分类信息
        class_true = get_pre_object_mask[..., 5 *self.BOX_NUM:]
        #y_true[get_no_object_mask],标注没有物体的 mask
```

```
        get_true_no_object_mask = tf.reshape(y_true[get_no_object_mask], [-1,
self.output_dim])

        #(4)根据预测没有物体的 get_pre_no_object_mask,分别去获取没有物体的预测
        #no_obj_pre_conf
        #及没有物体的标注 no_obj_true_conf

        #初始为 0,没有物体的 mask,此时背景为 1
        get_pre_conf_no_object_mask = np.zeros(get_pre_no_object_mask.shape)
        for b in range(self.BOX_NUM):
            #起到的作用是默认先选择所有的没有物体的格子,为了方便从 get_true_no_
            #object_mask 和 get_pre_no_object_mask 中取值
            get_pre_conf_no_object_mask[:, 4 + b * 5] = 1
        #取出来没有物体预测的格子
        no_obj_pre_conf = tf.gather(get_pre_no_object_mask, get_pre_conf_no_
object_mask.astype(int))
        #取出来没有物体真实的格子
        no_obj_true_conf = tf.gather(get_true_no_object_mask, get_pre_conf_no_
object_mask.astype(int))

        #所有没有物体的格子的损失
        loss_no_obj = tf.reduce_sum(tf.losses.MSE(no_obj_pre_conf, no_obj_true_conf))
        #(5)从预测的 BOX 中获取与真实框最大的 IOU,取最大的为有物体的 mask,然后让预测值
        #与真实值之间做损失
        coord_response_mask = np.zeros(bbox_true.shape)
        coord_not_response_mask = np.ones(bbox_true.shape)
        bbox_target_iou = np.zeros(bbox_true.shape)

        #从预测的 BOX 中获取与真实框最大的 IOU,遍历 batch 下所有的有物体的格子。因为每
        #个格子预测两个框,所以 step 是 2
        for i in range(0, bbox_true.shape[0], self.BOX_NUM):
            pre_box = bbox_pre[i:i + self.BOX_NUM]   #[0,2],预测的每个格子的 2 个框都
                                                     #进行计算
            pre_xy = np.zeros(pre_box.shape)
            #因为预测出来的是 cx,cy,w,h,并且缩放了,所以要还原到原图中,以便进行 IOU 的比较
            #算出 x1,y1,x2,y2
            pre_xy[:, :2] = pre_box[:, :2] / float(self.GRID_NUM) - 0.5 *pre_box[:, 2:4]
            pre_xy[:, 2:4] = pre_xy[:, :2] / float(self.GRID_NUM) + 0.5 *pre_xy[:, 2:4]
            #当为真实值时,因为编码时每个格子的 2 个框赋的值是一样的,所以取 1 个就可以了
            target_true = bbox_true[i]
            target_true = tf.reshape(target_true, [-1, 5])
            true_xy = np.zeros_like(pre_xy)
            #因为传的是 cx,cy,w,h,但是计算 IOU 要使用 x1,y1,x2,y2,所以要转换一下
            true_xy[:, :2] = target_true[:, :2] / float(self.GRID_NUM) - 0.5 *
target_true[:, 2:4]
            true_xy[:, 2:4] = target_true[:, :2] / float(self.GRID_NUM) + 0.5 *
target_true[:, 2:4]
            #获取预测框与真实框之间的 IOU
            get_iou = iou(pre_xy, true_xy)
            #得到所有框中最大的 max
            max_iou, max_index = np.max(get_iou), np.argmax(get_iou)
            coord_response_mask[i + max_index] = 1 #将有物体的 IOU 位置设置为 1,默认为 0
```

```
            coord_not_response_mask[i + max_index] = 0   #将没有物体的设置为 0,默认
                                                         #为 1,为 1 是为了方便取值
            bbox_target_iou[i + max_index, 4] = max_iou  #将 IOU 的值赋为置信度
            #(6)计算损失

        #根据有目标的 mask 取出 pre 的 BOX 值。因为 batch 中设置的都是第 1 个格子有目标,
        #所以这里都是第 1 个格子
        bbox_pred_response = tf.reshape(tf.gather(bbox_pre, coord_response_
mask.astype(int)), [-1, 5])
        bbox_target_response = tf.reshape(tf.gather(bbox_true, coord_response_
mask.astype(int)), [-1, 5])
        target_iou = tf.reshape(bbox_target_iou[coord_response_mask.astype(int)],
[-1, 5])

        #计算 x 和 y 的损失
        loc_loss_xy = tf.reduce_sum(
            tf.losses.MSE(bbox_pred_response[:, :2],
                          bbox_target_response[:, :2])
        )
        #计算 w 和 h 的损失
        loc_loss_wh = tf.reduce_sum(
            tf.losses.MSE(tf.sqrt(bbox_pred_response[:, 2:4]),
                          tf.sqrt(bbox_target_response[:, 2:4]))
        )
        #位置损失
        loc_loss = loc_loss_xy + loc_loss_wh

        #计算置信度损失。预测的置信度与 true 值和 pre 的 IOU 越接近越好
        loss_obj = tf.reduce_sum(tf.losses.MSE(bbox_pred_response[:, 4], target_
iou[:, 4]))
        #分类损失
        class_loss = tf.reduce_sum(
            tf.losses.MSE(class_pre,
                          class_true)
        )
        #位置损失+ 有物体的置信度损失+ 没有物体所有格子的损失+ 有物体的分类损失
        loss = self.coord * loc_loss + \
               tf.cast(loss_obj, dtype=tf.float32) + \
               self.no_obj * loss_no_obj + class_loss
        #总损失/batch_size
        loss = loss / tf.cast(batch_size, dtype=tf.float32)
        return loss
```

此损失计算的代码较长,共由 6 个步骤构成,首先 y_true[:,:,:,:,4]>0 获取有目标的 get_object_mask,y_true[:,:,:,:,4]==0 没有目标的 get_no_object_mask,如图 5-33 所示。

第 2 步,根据 get_object_mask、get_no_object_mask 到 y_pred 预测值中获取有目标的置信度、预测 BOX、预测分类信息,并得到预测值为有物体的 get_pre_object_mask、没有物体的 get_pre_no_object_mask,如图 5-34 所示。

第 3 步,根据第 1 步中的 get_object_mask 计算真实值的 BOX、置信度、分类信息的值,并获取没有目标 get_true_no_object_mask 的值,如图 5-35 所示。

```
get_object_mask = y_true[:, :, :, 4] > 0  get_object_mask: tf.Tensor(\n[[[
# true值中没有物体的框
get_no_object_mask = y_true[:, :, :, 4] == 0  get_no_object_mask: tf.Tenso
```

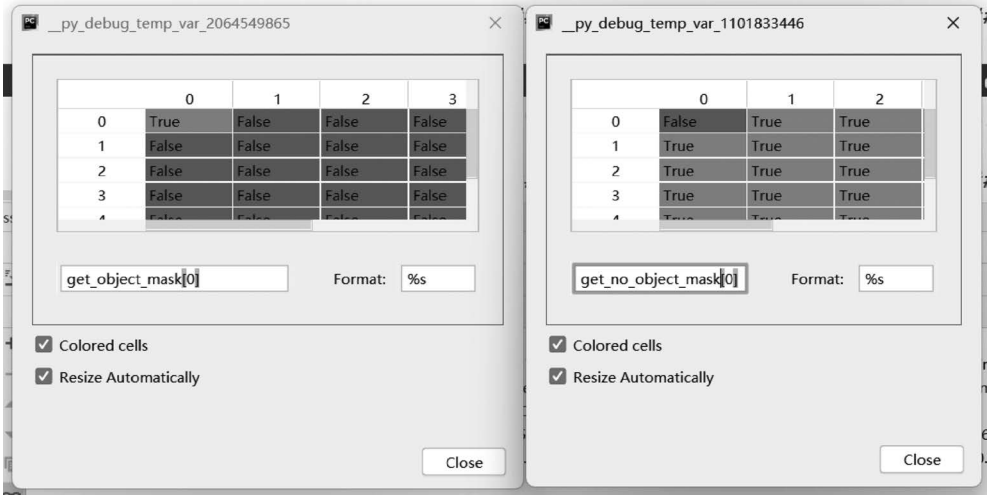



图 5-33　y_true 有无目标 mask

```
# 从预测框中得到置信度框的内容,y_pred[get_object_mask] get_object_mask有物体的mask
get_pre_object_mask = tf.reshape(y_pred[get_object_mask], [-1, self.output_d
# 获取2个预测框的值.因为预测输出为 2*(4+1)+20
bbox_pre = tf.reshape(get_pre_object_mask[..., :5 * self.BOX_NUM], [-1, 5])
# 类别信息的值
class_pre = get_pre_object_mask[..., 5 * self.BOX_NUM:]  class_pre: tf.Tenso

# 获取没有物体的格子 y_pred[get_no_object_mask] get_no_object_mask没有物体的mask
get_pre_no_object_mask = tf.reshape(y_pred[get_no_object_mask], [-1, self.ou
```

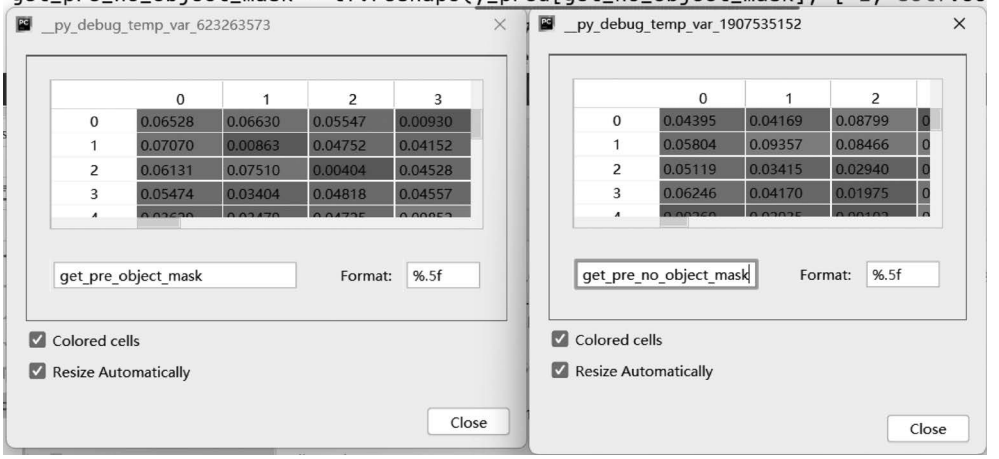



图 5-34　y_pred 有无目标 BOX 信息

```
# y_true[get_object_mask]，有物体的mask。get_true_object_mask为标注框，有物体的部分
get_true_object_mask = tf.reshape(y_true[get_object_mask], [-1, self.output_
#标注框，有物体的 BOX 信息
bbox_true = tf.reshape(get_true_object_mask[..., :5 * self.BOX_NUM], [-1, 5]
#标注框，有物体的分类信息
class_true = get_pre_object_mask[..., 5 * self.BOX_NUM:]  class_true: tf.Ten
# y_true[get_no_object_mask]，标注没有物体的mask
get_true_no_object_mask = tf.reshape(y_true[get_no_object_mask], [-1, self.o
#########################################################################
# 初始为0，没有物体的mask，此时背景为1
get_pre_conf_no_object_mask = np.zeros(get_pre_no_object_mask.shape)
```

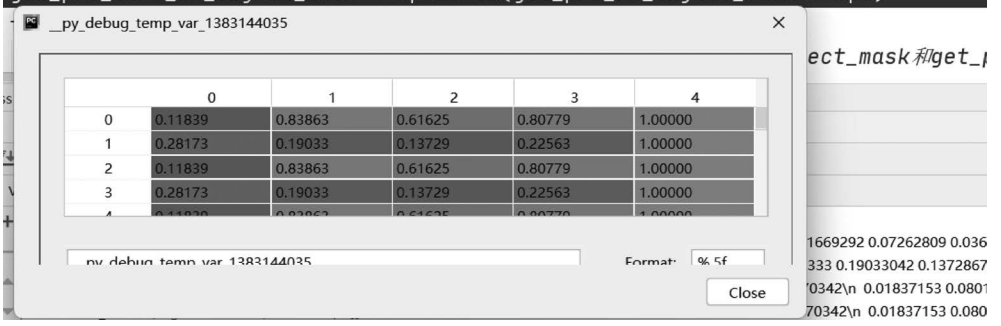



图 5-35　y_true 有无目标 BOX 信息

第 4 步，根据预测没有物体的 get_pre_no_object_mask，分别获取没有物体的预测 no_obj_pre_conf 及没有物体的标注 no_obj_true_conf，然后计算所有没有目标的置信度损失 loss_no_obj＝tf. reduce_sum(tf. losses. MSE(no_obj_pre_conf，no_obj_true_conf))，如图 3-36 所示。

```
#取出来没有物体预测的格子
no_obj_pre_conf = tf.gather(get_pre_no_object_mask, get_pre_conf_no_object_mask.astype(int))
#取出来没有物体真实的格子
no_obj_true_conf = tf.gather(get_true_no_object_mask, get_pre_conf_no_object_mask.astype(int))
#所有没有物体的格子的损失
loss_no_obj = tf.reduce_sum(tf.losses.MSE(no_obj_pre_conf, no_obj_true_conf))  loss_no_obj: tf
```

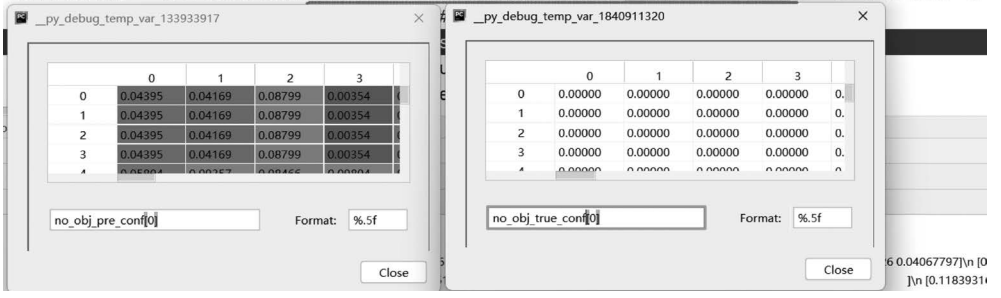



图 5-36　没有目标的置信度损失

第 5 步，从预测的 BOX 中获取与真实 BOX 最大的 IOU，取最大 IOU 作为有物体的 mask，然后让预测值与真实值之间做损失。具体是将 pre_box 与 bbox_true 计算 IOU，取

最大的那个 IOU 所在的 BOX 作为有目标，然后根据 coord_response_mask 取出预测值的 bbox_pred_response，以及真实值 bbox_target_response，如图 5-37 所示。

```
max_iou, max_index = np.max(get_iou), np.argmax(get_iou)  max_iou: 0.000655899
coord_response_mask[i + max_index] = 1  #有物体的IOU位置，设置为1。默认为0
coord_not_response_mask[i + max_index]= 0 #没有物体的，设置为0，默认为1，为1表示有目标
bbox_target_iou[i + max_index, 4] = max_iou  #将IOU的值赋为置信度

#根据有目标的mask取出pre的BOX值。因为batch中设置的都是第1个格子有目标，所以这里都是第1个格子
bbox_pred_response = tf.reshape(tf.gather(bbox_pre, coord_response_mask.astype(int
bbox_target_response = tf.reshape(tf.gather(bbox_true, coord_response_mask.astype(
```

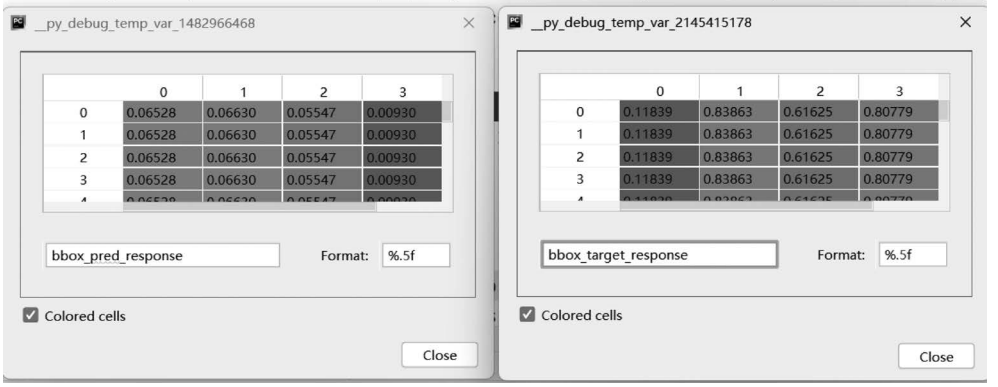



图 5-37　BOX 损失取预测 BOX 与真实 BOX 的最大 IOU

第 6 步，根据前面的步骤按式(5-7)计算损失，完成“5×位置损失＋有物体的置信度损失＋0.5×没有物体所有格子的损失＋有物体的分类损失”。

训练代码可使用 model.fit()函数完成，更多更详细的代码可参考随书代码。

## 5.5.5　代码实战预测推理

YOLOv1 的推理流程包括加载模型、加载推理图片进行置信度阈值过滤、偏移值解码，以及 NMS 非极大值抑制等操作，详细的代码如下：

```
#第 5 章/ObjectDetection/TesnsorFlow_YOLO_V1_Detected/detected/detected.py
class Detected():
   def __init__(self, model_path, input_size):
      self.model = load_model(
          model_path,
          custom_objects={'compute_loss': MultiAngleLoss(3).compute_loss}
      ) #读取模型和权重
      self.confidence_threshold = 0.5                    #有无目标置信度
      self.class_prob = [0.5, 0.5]                       #两个类别的分类阈值
      self.nms_threshold = 0.5                           #NMS 的阈值
      self.input_size = input_size

   def readImg(self, img_path=None):
      img = cv2.imread(img_path)                         #读取要预测的图片
```

```
        #将图片转换到 448×448
        self.img, _ = u.letterbox_image(img, self.input_size, [])
        self.old_img = self.img.copy()

    def forward(self):
        #升成 4 维
        img_tensor = tf.expand_dims(self.img / 255.0, axis=0)
        #前向传播
        self.output = self.model.predict(img_tensor)

    def confidence_filtering(self):
        #根据置信度的阈值进行过滤,如果大于>0.5,则为有目标
        #YOLOv1 的输出是 7×7×[(4+1) ×2+2],所以这里应该输出 7×7×2
        self.targeted = np.concatenate([self.output[..., [4]], self.output
[..., [4 + 5]]], axis=-1)

    def classification_filtering(self):
        self.S = 7                                          #划分的格子数
        self.B = 2                                          #每个格子预测两个框
        cell_size = 1.0 / float(self.S)                     #每个格子的 size 为 1/7
        #用来存储筛选后的内容
        boxes, labels, confidences, class_scores = [], [], [], []
        #遍历每个格子
        for i in range(self.S):
            for j in range(self.S):
                #遍历每个格子中的两个框
                for b in range(self.B):
                    #分类得分
                    class_score = self.output[..., j, i, 5 *self.B:]
                    #取最大的分类得分的下标
                    class_label = np.argmax(class_score)
                    #最大的分类得分值
                    score = class_score[..., class_label]
                    #当前格子的置信度
                    conf = self.targeted[..., j, i, b]
                    #每个格子最后的得分为置信度*分类得分
                    prob = score *conf
                    #如果小于阈值,则跳过
                    if float(prob) < self.confidence_threshold:
                        continue
                    #当前预测 BOX 信息
                    box = self.output[..., j, i, 5 *b:5 *b + 4]
                    #每个格子点的归一化坐标
                    x0y0_normalized = np.array([i, j]) *cell_size
                    #解码操作,x+i,y+j 即预测的 cxcy
                    xy_normalized = box[..., :2] *cell_size + x0y0_normalized
                    #YOLOv1 直接预测 wh
                    wh_normalized = box[..., 2:]
                    #合并 cxcyxy
                    cxcywh = np.concatenate([xy_normalized, wh_normalized], axis=-1)
                    #由 cx,cy,w,h 转换成 xmin,ymin,xmax,ymax
                    xyxy_box = u.cxcy2xyxy(cxcywh)
```

```
                    #对结果进行存储
                    boxes.append(xyxy_box)
                    labels.append([class_label])
                    confidences.append(conf)
                    class_scores.append(class_score)

        #对于得到的 BOX 信息,按类别进行非极大值抑制
        if len(boxes) > 0:
            #由 list 合并成 NumPy
            boxes_normalized_all = np.stack(boxes, 1)
            class_labels_all = np.stack(labels, 1)
            confidences_all = np.stack(confidences, 1)
            class_scores_all = np.stack(class_scores, 1)
            #遍历每个类别
            for label in range(len(self.class_prob)):
                #如果 class_labels_all==label,则取当前 label 中的信息
                mask = class_labels_all == label
                #如果都不是当前 label,则跳过
                if np.sum(mask) == 0: continue
                #当前 label 的 boxes
                boxes_mask = boxes_normalized_all[mask]
                #当前 label 的 class_labels。reshape 是由于得到的是(50,)变成(50,1),
                #方便后面计算
                class_labels_mask = class_labels_all[mask].reshape([-1, 1])
                #当前 label 的 confidences
                confidences_mask = confidences_all[mask].reshape([-1, 1])
                #当前 label 的 class_scores
                class_scores_mask = class_scores_all[mask][..., label].reshape([-1, 1])
                #合并
                cat_boxes = np.concatenate([boxes_mask, confidences_mask], axis=-1)
                #NMS
                index = u.nms(cat_boxes, self.nms_threshold)
                #绘框
                self.old_img = u.draw_box(self.old_img, cat_boxes[index])
        #最后结果
        u.show(self.old_img)


if __name__ == "__main__":
    d = Detected(r'../weights', input_size=[448, 448])
    d.readImg('../../val_data/pexels-photo-5211438.jpeg')

    d.forward()
    d.confidence_filtering()
    d.classification_filtering()
```

相对于其他模型,推理的改变在 confidence_filtering(self)方法中,这是由于 YOLOv1 的输出是 7×7×[(4box+1 置信度)×2 个框+2 个分类],所以 self.output 输出 7×7×12,获取置信度结果在第 4、第 9 位,如图 5-38 所示。

在 classification_filtering(self)中实现了根据每个格子每个框 prob=score * conf,置信度×分类概率的得分得到 box=self.output[...,j,i,5 * b:5 * b+4]信息,然后根据当前所

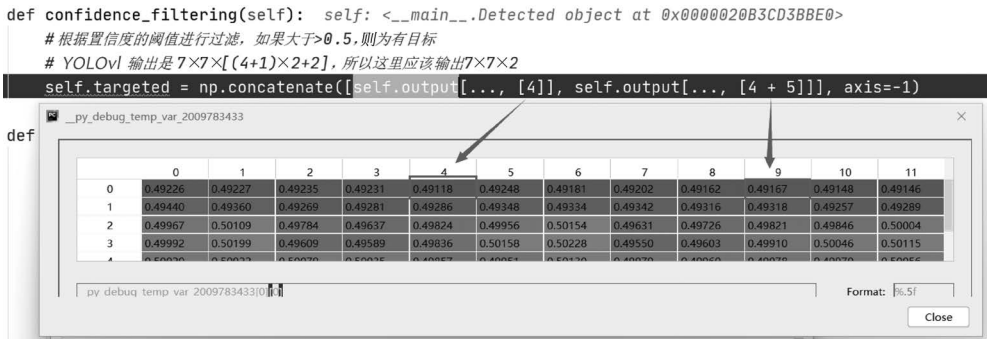



图 5-38 YOLOv1 置信度取值

在的格子进行 xy_normalized＝box[…, :2] * cell_size＋x0y0_normalized 解码操作，得到预测的 cx，cy。由于 YOLOv1 采用的是无锚框机制，所以它直接预测的是 wh_normalized＝box[…,2:]，然后将解码的内容存储到 boxes 中，如图 5-39 所示。

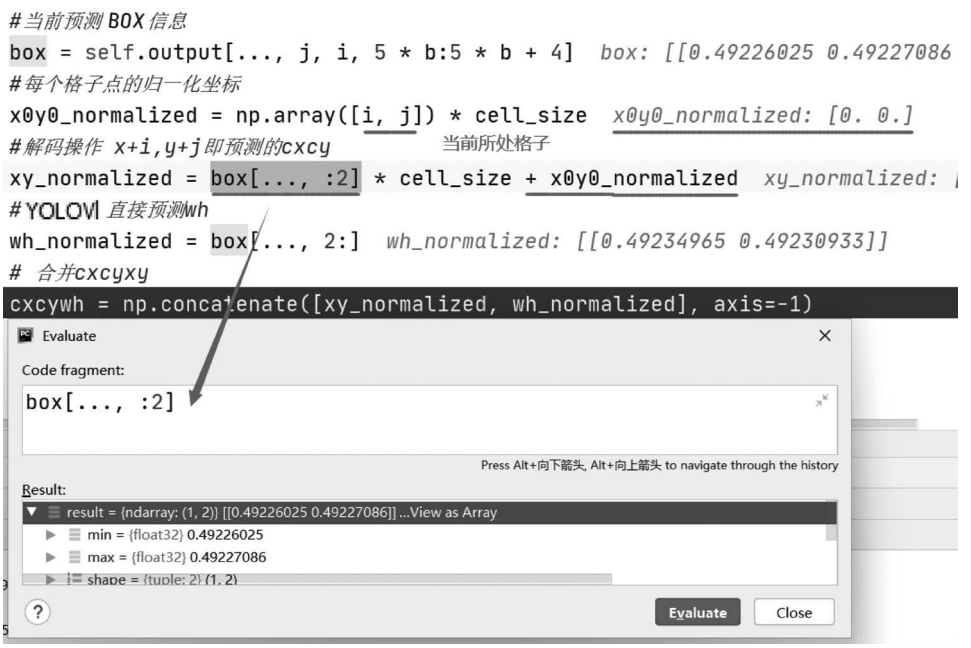



图 5-39 YOLOv1 解码操作

因为 class_labels_all 中存储了所有解码后的 label，如果 mask＝class_labels_all＝＝label，则表明只取当前 label 的信息，经过 u. nms(cat_boxes，self. nms_threshold)非极大值抑制，最后得到当前类别的 cat_boxes[index]，以及最后预测的 BOX 信息，如图 5-40 所示。

**总结**

YOLOv1 为 Anchor Free 机制，通过划分 7×7 个格子，由 GT 落入某个格子的中心点来预测目标，其主要特点为速度快。

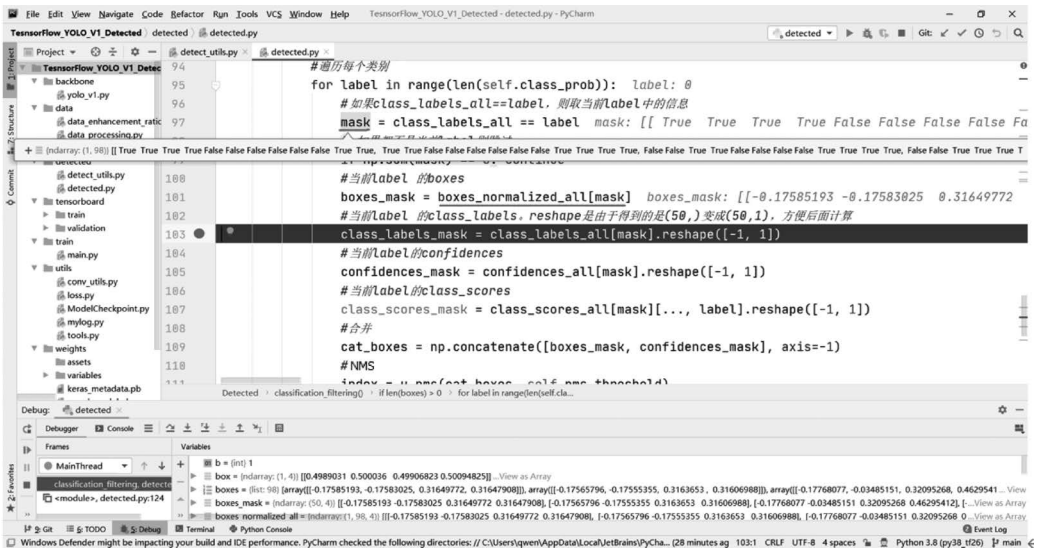



图 5-40　mask 过滤当前 label

**练习**

运行并调试本节代码，理解算法的设计与代码的结合，重点梳理本算法的实现方法。

## 5.6　单阶段速度快的检测网络 YOLOv2

### 5.6.1　模型介绍

YOLOv2 由 Joseph Redmon 等在 2016 年论文 *YOLO9000：Better，Faster，Stronger* 中提出，其主要特点在 YOLOv1 的基础上引入了 BN 归一化、建议框和 PassThrough Layer 层，其网络结构如图 5-41 所示。

在主干特征提取方面将 YOLOv1 的 7×7 卷积替换成 3×3 的卷积，并且在每个卷积后面跟了 BN 归一化。深层的语义信息较丰富，而浅层的几何信息较丰富，为了提高多尺度特征信息的融合，作者在这里使用了 PassThrough 层，经过 PassThrough 层的特征信息与 13×13×1024 进行 Concate，从而得到 13×13×1280 维特征。

PassThrough 具体的实现对输入的 26×26×64 每两个尺度进行重组，促进特征和通道信息的合并交流，如图 5-42 所示。

图 5-42 中输入为 4×3×3，对每两个通道且每个通道的每两组进行拼接，通道与通道之间的信息进行了重组，加强了通道之间的交流，并且没有权重参数。

在 Concate 之后经过 3×3 卷积，然后用 1×1 卷积代替全连接，输出为(4＋1＋20)×5，每个特征点输出 4 个预测 BOX 相对于 Anchor 的偏移，并且预测 BOX 有无目标，20 个分类的概率，每个特征分配 5 个建议框，所以输出为 13×13×125 维向量。

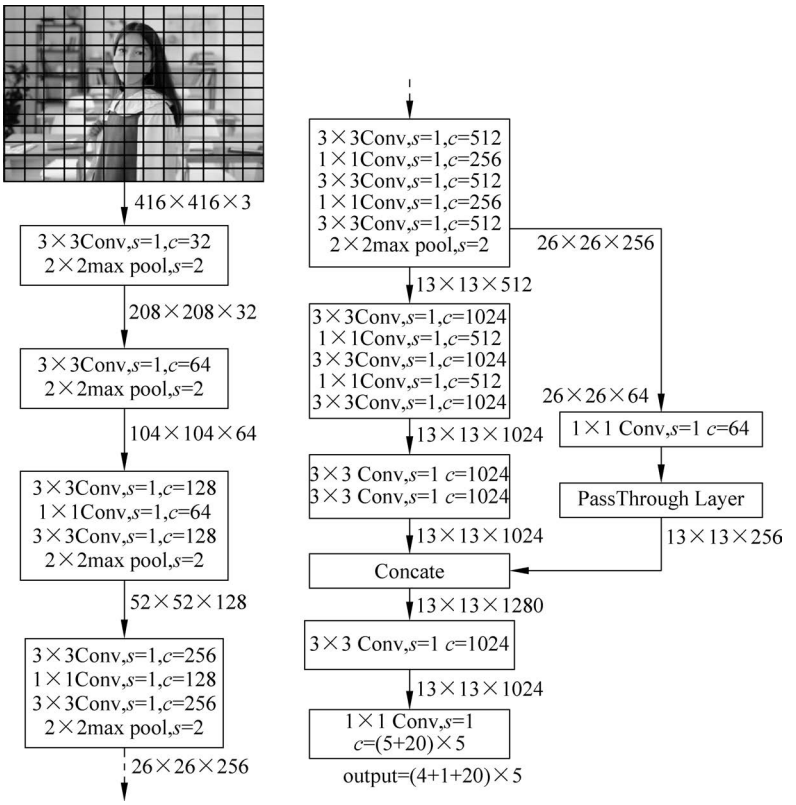



图 5-41　YOLOv2 结构图

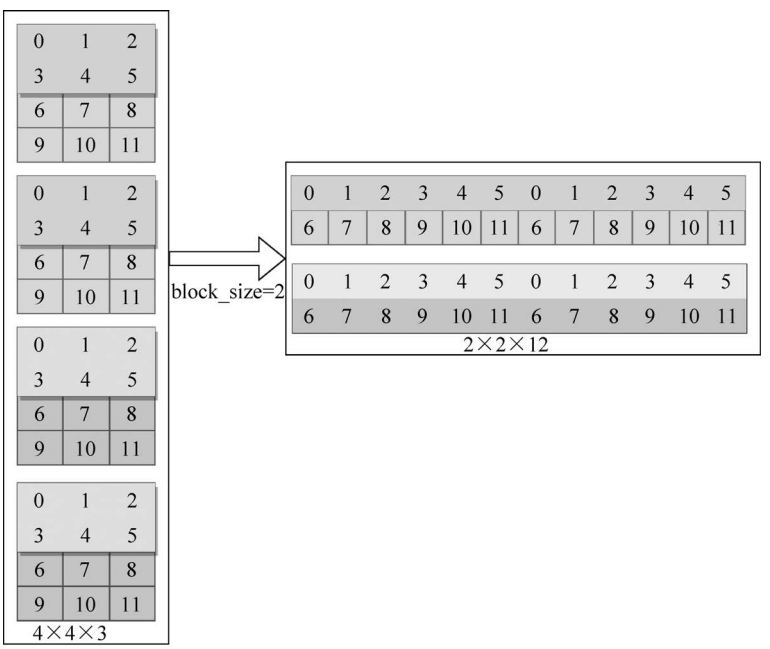



图 5-42　PassThrough 结构

YOLOv2 在计算 $t_x$、$t_y$ 时仍然是基于(0,0)坐标的偏移，但是在计算 $t_w$、$t_h$ 时是基于每个像素生成的建议框与 GT BOX 标注框 wh 的偏移，每个坐标点生成 5 个尺寸的建议框，如图 5-43 所示。

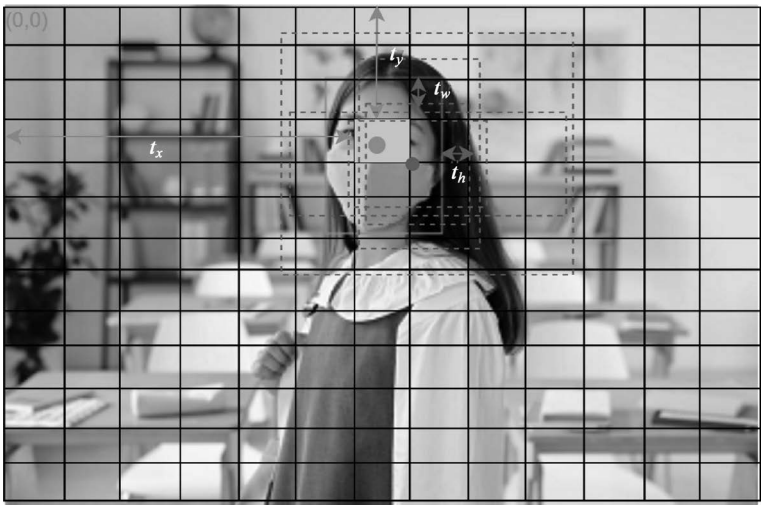



图 5-43　YOLOv2 的正样本

YOLOv2 将原图等比例缩放至 416×416，其输出为 13×13×125，则相对于将原图划分为 13×13 个格子，每个格子对于特征图的输出。每个特征点生成 5 个建议框，这 5 个建议框使用以下公式计算 $t_x$、$t_y$、$t_w$、$t_h$。

$$\begin{aligned} t_x &= G_x - j \\ t_y &= G_y - i \\ t_w &= \log\left(\frac{G_w}{P_w}\right) \\ t_h &= \log\left(\frac{G_h}{P_h}\right) \end{aligned} \tag{5-8}$$

其中，$G_x$、$G_y$ 代表标注 BOX 所在图像的中心点；$j$、$i$ 为标注 BOX 所在的格子；$P_w$、$P_h$ 为预设建议框的宽和高。从公式 5-8 可知 YOLOv2 正样本的选择与标注 BOX 所在的格子及建议框有关，在挑选建议框时将标注 BOX 与建议框计算 IOU，并挑选最大 IOU 作为正样本的 Anchor。

在损失函数方面，仍由正样本 BOX 损失＋正样本置信度损失＋负样本置信度损失＋分类损失构成，其公式可表述如下：

$$\begin{aligned} \text{Loss} = &\sum_{i=0}^{W}\sum_{j=0}^{H}\sum_{k=0}^{A}\lambda_{\text{noobj}}1_{\text{ijk}}^{\text{noobj}}[(C_i-\hat{C}_i)^2]+\lambda_{\text{obj}}1_{\text{ijk}}^{\text{obj}}[(C_i-\hat{C}_i)^2]+ \\ &\lambda_{\text{coord}}1_{\text{ijk}}^{\text{obj}}[(\text{Box}_i-\widehat{\text{Box}}_i)^2]+\lambda_{\text{class}}1_{\text{ijk}}^{\text{obj}}[(p_i(c)-\hat{p}_i(c))^2] \end{aligned} \tag{5-9}$$

其中，$C_i$ 为标注 BOX 与 Anchor 之间 IOU 得分值，如果大于设定的阈值，则为正样本，如果小于设定的预测值，则为负样本。$\hat{C}_i$ 为预测的置信度概率值。$\text{Box}_i$ 为标注 BOX 与 Anchor 的偏移，$\widehat{\text{Box}_i}$ 为预测的偏移值。$p_i(c)$ 为标注 BOX 的分类概率，$\hat{p}_i(c)$ 为预测物体的分类概率。$\sum_{i=0}^{W}\sum_{j=0}^{H}\sum_{k=0}^{A}$ 表明需要遍历每个格子及建议框。

### 5.6.2 代码实战模型搭建

模型代码实现参考图 5-41 完成，PassThrough 层的实现可调用 tf.nn.space_to_depth()，详细的代码如下：

```
#第 5 章/ObjectDetection/TesnsorFlow_YOLO_V2_Detected/backbone/yolo_v2.py
def yolo_v2(input_shape, class_num=20, anchor_num=5):
    """YOLOv2 由两部分构成，一部分是 DarkNet19，另一部分是 YOLO 增加的
    :param anchor_num:Anchor 的数量
    :param class_num:分类的数量
    :param input_shape:输入 shape
    :return:
    """
    input_tensor = layers.Input(shape=input_shape)
    is_bn = True                              #使用 BN
    net = {}
    #结构配置文件
    #[类型, kernel_size, strides, out_channel, 'same']
    darknet_config = [
        ['Conv', 3, 1, 32, 'same'],           #416 × 416
        ['MaxP', 2, 2, 32, 'same'],

        ['Conv', 3, 1, 64, 'same'],           #208 × 208
        ['MaxP', 2, 2, 64, 'same'],

        ['Conv', 3, 1, 128, 'same'],          #104 × 104
        ['Conv', 1, 1, 64, 'same'],
        ['Conv', 3, 1, 128, 'same'],
        ['MaxP', 2, 2, 128, 'same'],

        ['Conv', 3, 1, 256, 'same'],          #52 × 52
        ['Conv', 1, 1, 128, 'same'],
        ['Conv', 3, 1, 256, 'same'],
        ['MaxP', 2, 2, 256, 'same'],

        ['Conv', 3, 1, 512, 'same'],          #26 × 26
        ['Conv', 1, 1, 256, 'same'],
        ['Conv', 3, 1, 512, 'same'],
        ['Conv', 1, 1, 256, 'same'],
        ['Conv', 3, 1, 512, 'same'],
        ['MaxP', 2, 2, 512, 'same'],

        ['Conv', 3, 1, 1024, 'same'],         #13 × 13
```

```
        ['Conv', 1, 1, 512, 'same'],
        ['Conv', 3, 1, 1024, 'same'],
        ['Conv', 1, 1, 512, 'same'],
        ['Conv', 3, 1, 1024, 'same'],          #Conv-18, 13 × 13
    ]
    #concate 前的两个 3×3 卷积
    yolo_add = [
        ['Conv', 3, 1, 1024, 'same'],
        ['Conv', 3, 1, 1024, 'same'],
    ]
    x = input_tensor
    net['input'] = x
    #解析配置文件
    for i, c in enumerate(darknet_config):
        if c[0] == 'Conv':
            #已封装好的 conv→bn→leak_relu
            x = ConvBnLeakRelu(c[3], c[1], c[2], is_bn=is_bn, padding=c[4],
 is_relu=True)(x)
        elif c[0] == 'MaxP':
            #池化
            x = layers.MaxPooling2D(c[1], c[2], padding=c[4])(x)
        net[i] = x
    #构建两个 3×3 卷积
    for j, c in enumerate(yolo_add):
        if c[0] == 'Conv':
            x = ConvBnLeakRelu(c[3], c[1], c[2], is_bn=is_bn, padding=c[4],
 is_relu=True)(x)
        net[j + i] = x
    #将通过 pass_through 得到的 13×13×256 与 13×13×1024 进行合并
    x = layers.concatenate([x, pass_through(net[16], is_bn)], axis=-1)
    net['pass_concatenate'] = x
    x = ConvBnLeakRelu(1024, 3, 1, is_bn=is_bn, padding='same', is_relu=True)(x)
    net['pass_next'] = x
    #输出 13 × 13 × (4+1+num_class) ×anchor,即 13 × 13 × 125
    #直接位置预测,(4+1+20) ×5
    net['output'] = ConvBnLeakRelu((4 + 1 + class_num) *anchor_num, 1, 1, is_bn=
False, is_relu=False)(x)
    #构建模型
    return Model(inputs=net['input'], outputs=net['output'])

 def pass_through(x, is_bn=True, channel=64):
    #pass_through 的封装,由 space_to_depth()函数实现该功能
    cx = ConvBnLeakRelu(channel, 1, 1, is_bn=is_bn, padding='same', is_relu=
True)(x)                                    #26 × 26 × 512
    cx = tf.nn.space_to_depth(cx, 2)        #13×13×256 pass through 起到的作用
                                            #是在各个通道中每隔两个进行合并交流
 return cx
```

代码中 darknet_config 为主干网络的配置，ConvBnLeakRelu()为封装好的卷积、BN、LeakRelu 激活函数，pass_through(x,is_bn＝True,channel＝64)实现 PassThrough 层，网络最后的输出为 13×13×(4＋1＋num_class)×anchor。

### 5.6.3 代码实战聚类得到建议框宽和高

YOLOv2 建议框的宽和高通过聚类得到，其判断距离的公式为

$$d(\text{box},\text{centroid}) = 1 - \text{IOU}(\text{box},\text{centroid}) \tag{5-10}$$

其中，$d$ 代表距离，box 代表标注 BOX，centroid 为聚簇中心，1－IOU(box，centroid)，因为 IOU 越大说明两个 BOX 越近为 1，所以最小距离接近 0，核心代码如下：

```
#第 5 章/ObjectDetection/TesnsorFlow_YOLO_V2_Detected/utils/get_anchors.py
class AnchorKmeans(object):
    """聚类实现，获取 Anchor 的宽和高"""

    def __init__(self, k, max_iter=300, random_seed=None):
        self.k = k                          #设置几个中心点
        self.max_iter = max_iter            #最多迭代多少次
        self.random_seed = random_seed      #随机种子
        self.n_iter = 0
        self.anchors_ = None
        self.labels_ = None
        self.ious_ = None

    def fit(self, boxes):
        """得到 anchors"""
        assert self.k < len(boxes), "K 必须少于 BOX 的数量"
        #迭代次数，保证每次从 0 开始
        if self.n_iter > 0: self.n_iter = 0
        #随机种子
        np.random.seed(self.random_seed)
        #boxes 的数量
        n = boxes.shape[0]
        #从现有 Anchor 中随机选择 K 个 Anchor 作为初始点
        self.anchors_ = boxes[np.random.choice(n, self.k, replace=True)]
        #label 标签
        self.labels_ = np.zeros((n,))
        #开始聚类
        while True:
            #每迭代 1 次 self.n_iter+1
            self.n_iter += 1
            #迭代的次数要小于设置的总次数
            if self.n_iter > self.max_iter: break
            #将其他 BOX 与随机选择的中心点 Anchor 做 IOU
            self.ious_ = self.iou(boxes, self.anchors_)
            #距离 1-IOU→0，如果离得很近，则说明 BOX 与 Anchor 趋近于 1
            distances = 1 - self.ious_
            #取最小距离的下标
            cur_labels = np.argmin(distances, axis=1)
            #如果最小距离的下标与分配的下标一致，则停止
            if (cur_labels == self.labels_).all():
                break
            #更新 Anchor 的位置
            for i in range(self.k):
```

```
                self.anchors_[i] = np.mean(boxes[cur_labels == i], axis=0)
            self.labels_ = cur_labels
if __name__ == "__main__":
    xml_dir = "../face_mask/facemask_dataset_annotations"
    jpg_dir = '../face_mask/facemask_dataset'
    #获取所有标注的 boxes
    boxes = parse_xml(xml_dir, jpg_dir)
    #设置 k=5
    model = AnchorKmeans(5, random_seed=1000)
    model.fit(boxes)
    #获得聚类结果
    print(model.anchors_)
```

代码 distances=1-self.ious_,表示所有 BOX 与聚簇中心越近,且当 if(cur_labels==self.labels_).all():break 时,如果最小距离的下标与分配的下标一致,则停止。如果将 $K$ 的数量从 2 设置到 20,则可以获取 $K$ 越大其 Anchor 与 GT BOX 的 IOU 得分越高,如图 5-44 中 $K=15$ 时平均 IOU=0.75。增大 $K$ 的设置可以提高精度,但同时会降低网络的推理速度,所以可以根据实际情况进行选择。

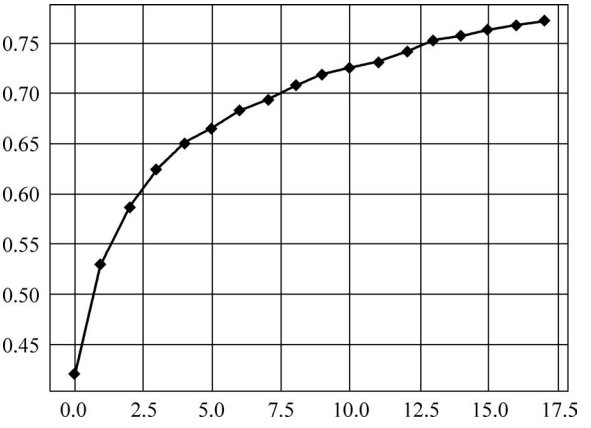



图 5-44　聚类 Anchor 中 $K$ 不同选择 IOU 的变化

## 5.6.4　代码实战建议框的生成

首先,代码需要先计算 GT BOX 在哪个格子中,然后计算 GT BOX 与 Anchor 的 IOU 值,取最大 IOU 值所在的 Anchor 作为正样本,然后在 GT BOX 与挑选出来的 Anchor 之间计算偏移,参考代码如下:

```
#第 5 章/ObjectDetection/TesnsorFlow_YOLO_V2_Detected/utils/tools.py
def yolo_v2_true_encode_box(true_boxes,
                            anchors=None,
                            input_size=(416, 416)):
    """
    YOLOv2 的真值编码。需要编码成 GRID_NUM * GRID_NUM * (4 + 1 + class_num) * anchor_num
```

```
    :param true_boxes:
    两个维度：
    第 1 个维度：一张图片中有几个实际框
    第 2 个维度：[cx, cy, w, h, class],x 和 y 是框中心点坐标,w 和 h 是框的宽度和高度
    x,y,w,h 均是除以图片分辨率(原始图片尺寸 416*416)得到的[0,1]范围的比值
    :param anchors: 实际 anchor boxes 的值,论文中使用了 5 个。[w,h]都是相对于 grid
cell 的比值
    :param input_size: true box 中的输入图片的尺寸要与预测时的保持一致
    :return:
    true_boxes2 13 × 13 × 5 × 4,返回它是为了方便背景损失的计算
    detectors_mask 13 × 13 × 5 × 1,置信度
    matching_true_boxes 13 × 13 × 5 × 5,GT BOX 基于 Anchor 的偏移
    """
    if anchors is None:
        #默认设置的 Anchor 大小,需要乘以 13
        anchors = [[1.08, 1.19], [3.42, 4.41], [6.63, 11.38], [9.42, 5.11], [16.62,
10.52]]
        #anchors×13 为真实 Anchor
        anchors = np.array(anchors) *(input_size[0] //32)
    #输入图像的尺寸
    height, width = input_size
    assert height % 32 == 0 #必须是 32 的倍数
    assert width % 32 == 0
    #特征图的尺寸,也就是 13×13
    conv_height, conv_width = height //32, width //32
    #true box 的数量
    num_box_params = true_boxes.shape[1]
    #5 个 Anchor
    NumPy_boxes = len(anchors)
    #初始一个 Tensor,用来存储 BOX 正样本值 13×13×5×4
    true_boxes2 = np.zeros(
        [conv_height, conv_width, NumPy_boxes, 4], dtype=np.float32
    )
    #初始一个 Tensor,用来存储置信度值 13×13×5×1
    detectors_mask = np.zeros(
        [conv_height, conv_width, NumPy_boxes, 1], dtype=np.float32
    )
    #13 × 13 × 5 × len(true_boxes), BOX+置信度,跟 true boxes 数保持一致
    matching_true_boxes = np.zeros(
        [conv_height, conv_width, NumPy_boxes, num_box_params], dtype=np.float32
    )
    #对所有的 true_boxes 进行编码
    for box in true_boxes:
        #置信度
        box_class = box[4:5]     #这样得到的是一个数组。如果是 box[4],则得到的是一个
                                 #具体值
        #box = (13 × x,13 × y, 13 × w, 13 × h) 换算成相对 grid cell 的值
        #[0.5078125 0.36830357 0.14955357 0.19642857] × 13
        #[6.6015625 4.78794637 1.94419637 2.55357137]
        box = box[0:4] *np.array([
            conv_width, conv_height, conv_width, conv_height
        ])
        box_true = box.copy()
```

```
        #向下取整,计算中心点落在哪个格子中
        i = np.floor(box[1]).astype('int')                    #i=4
        j = np.floor(box[0]).astype('int')                    #j=6

        best_iou = 0
        best_anchor = 0
        #将 true_box 与每个 Anchor 进行 IOU,并取最佳的 Anchor 作为正样本
        for k, anchor in enumerate(anchors):
            #true box 的 wh 1/2
            box_maxes = box[2:4] * 0.5
            box_mines = -box_maxes
            #Anchor BOX 的 wh 1/2
            anchor_maxes = anchor * 0.5
            anchor_mines = -anchor_maxes
            #将真实 wh 与 Anchor 之间进行 IOU 的计算,并获取最佳 IOU 是哪个 Anchor
            intersect_mines = np.maximum(box_mines, anchor_mines)
            intersect_maxes = np.minimum(box_maxes, anchor_maxes)
            intersect_wh = np.maximum(intersect_maxes - intersect_mines, 0.)
            intersect_area = intersect_wh[0] * intersect_wh[1]
            box_area = box[2] * box[3]
            anchor_area = anchor[0] * anchor[1]
            iou_score = intersect_area / (box_area + anchor_area - intersect_area)
            #比较当前 Anchor 的 IOU 是否比上一次的 IOU 值大,如果大,则是最佳 IOU,并记录是
            #第几个 k
            if iou_score > best_iou:
                best_iou = iou_score
                best_anchor = k
        if best_iou > 0: #当前示例 best_iou=0.85,k=2
            #detectors_mask 为 13×13×5×1,将最佳 Anchor 的置信度设置为 1.0
            #detectors_mask[4,6,2]=1.0
            detectors_mask[i, j, best_anchor] = 1.0
            #true_boxes2 为 13×13×5×4,即 BOX 的值
            #true_boxes2[4,6,2]=[6.6015625 4.78794637 1.94419637 2.55357137]
            true_boxes2[i, j, best_anchor] = box_true
            #套公式,算偏移。cx - j, cy - i, np.log(w/w *),np.log(h/h *)
            adjusted_box = np.array([
                box[0] - j, #gt_cx-j,gt_cy-i
                box[1] - i,
                np.log(box[2] / anchors[best_anchor][0]),    #gt_w/anchors[2][0],
                                                             #即 gt_w/a_w
                np.log(box[3] / anchors[best_anchor][1]),    #gt_h/anchors[2][1],
                                                             #即 gt_w/a_h
                box_class                                    #GT BOX 的 label 下标
            ], dtype=np.float32)
            #matching_true_boxes 13 × 13 × 5 × 5
            #matching_true_boxes[4,6,2] = offset 值
            matching_true_boxes[i, j, best_anchor] = adjusted_box
            #返回 GT BOX,Anchor 正样本的置信度,GT BOX 基于 Anchor 的偏移
    return true_boxes2, detectors_mask, matching_true_boxes

def test_yolo_v2_true_encode():
    data = np.array(
        [[0.5078125, 0.36830357, 0.14955357, 0.19642857, 1.]]
```

```
        , dtype=np.float32
    )
    return yolo_v2_true_encode_box(data)
```

代码中 box=box[0:4] * 13,box[0:4]是归一化后的值,乘以 13 得到[6.6015625 4.78794637 1.94419637 2.55357137],即从归一化值后放大到 13×13 的特征图的大小,向下取整则所在格子为 $i=4$、$j=6$。box_maxes=box[2:4] * 0.5 得到 GT BOX 的中心点,因为计算 IOU 还需要左上角的值,所以 box_mines=-box_maxes,得到 best_iou=0.8,best_anchor=2,那么将 detectors_mask[4,6,2]=1.0,表示第 $i=4$、$j=6$ 格子中的第 2 个 Anchor 置信度为 1,然后选择第 2 个 Anchor 并通过公式求与 GT BOX 的偏移值并赋给 adjusted_box,最后 matching_true_boxes[4,6,2]=adjusted_box,返回正样本的偏移值。

得到正样本的偏移值后,仍然像 YOLOv1 中的 DataProcessingAndEnhancement(object)类的 generate(self,isTraining=True)喂给训练数据,具体代码实际基本一致,详细可参考随书代码。

### 5.6.5 代码实现损失函数的构建及训练

根据式(5-9)实现正样本格子 $i$、$j$、$k$ 位置的损失、置信度的损失及分类概率的损失,核心代码如下:

```
#第 5 章/ObjectDetection/TesnsorFlow_YOLO_V2_Detected/utils/loss.py
class MultiAngleLoss(object):
    def yolo_v2_loss(self, y_true, y_pre):
        """
        求 YOLOv2 损失
        :param y_pre: model 预测值,需要解码
        :param y_true: 真实值
        :return:
        """
        #预测值
        #(1)对预测出来的 offset 解码成 xmin,ymin,xmax,ymax,conf,class_score
        #对解码出来的值跟 GT BOX 之间做 IOU,如果 IOU>阈值,则为正样本置信度,否则为负样
        #本置信度
        yolo_output = y_pre
        true_boxes = y_true[..., :4]            #真实坐标 b × 13 × 13 × 5 × 4
        detectors_mask = y_true[..., 4:5]       #置信度 b × 13 × 13 × 5 × 1
        matching_true_boxes = y_true[..., 5:]   #anchor b × 13 × 13 × 5 × 5
        #对预测出来的值进行 offset 解码,解码成 xmin,ymin,xmax,ymax
        boxes, pre_box_confidence, pre_box_class_props = yolo_v2_head_decode(
            yolo_output,
            self.anchors,
            self.num_classes,
            self.input_size
        )
        #预测置信度 b×13×13×5×1
        pre_box_confidence = tf.reshape(
            pre_box_confidence, [-1, self.conv_height, self.conv_width, self.num_
anchors, 1]
```

```
        )
        #预测分类 b×13×13×5×20
        pre_box_class_props = tf.reshape(
            pre_box_class_props, [-1, self.conv_height, self.conv_width, self.
num_anchors, self.num_classes]
        )
        #预测 xmin,ymin,xmax,ymax
        pre_boxes_xy = boxes[..., :2]
        pre_boxes_wh = boxes[..., 2:4]
        ########################################
        #(2)再从 pre_y 中取出偏移值,是为了计算预测偏移与真实 BOX 偏移之间的损失
        #将 YOLO 输出的[b, 13, 13, 125]转换为[b, 13, 13, 5, 25],5 是 Anchor 数量
        yolo_out_shape = yolo_output.shape[1: 3] #[b, 13, 13, 125]
        features = tf.reshape(
            yolo_output,
            [-1, yolo_out_shape[0], yolo_out_shape[1], self.num_anchors, self.
num_classes + 5]
        )
        #预测出来的是偏移值 xy 限制了值域,只能在 0~1
        pre_d_boxes = tf.concat([tf.nn.sigmoid(features[..., 0:2]), features[...,
2:4]], axis=-1)
        ########################################
        #(3)由 xmin,ymin,xmax,ymax 转换成 cx,cy,w,h,这是为了计算 IOU
        #维度调整
        pre_boxes_xy = tf.reshape(pre_boxes_xy, [-1, self.conv_height, self.conv_
width, self.num_anchors, 2])
        pre_boxes_wh = tf.reshape(pre_boxes_wh, [-1, self.conv_height, self.conv_
width, self.num_anchors, 2])
        #预测 cx,cy,w,h,为了计算 IOU
        pre_box = tf.concat([
            (pre_boxes_xy - pre_boxes_wh) *0.5,
            (pre_boxes_xy + pre_boxes_wh) *0.5,
        ], axis=-1)
        ########################################
        #(4)当 GT BOX 与 pre box 之间 IOU>阈值时,才认为 object_detections 有目标
        #算一下 true 的 xmin,ymin,xmax,ymax,方便做 IOU 的计算
        true_box = tf.concat([
            (true_boxes[..., 0:2] - true_boxes[..., 2:4]) *0.5,
            (true_boxes[..., 0:2] + true_boxes[..., 2:4]) *0.5,
        ], axis=-1)
        #得到预测框与真实框之间的 IOU 得分
        iou_result = iou(pre_box, true_box)   #1 × 13 × 13 × 5
        #获得最大得分,即 13 *13 *5 *1
        iou_score = tf.expand_dims(tf.reduce_max(iou_result, axis=-4), axis=-1)
        #过滤 IOU 要大于指定的置信度
        object_detections = iou_score > self.overlap_threshold
        object_detections = tf.cast(object_detections, dtype=iou_score.dtype)
        ########################################
        #(5)根据公式计算损失
        #当预测框的 IOU 与真实框中的 IOU 小于 0.6 时都为背景
        #没有目标物体的损失 1 - object_detections 预测没有目标,1 - detectors_mask
        #真实没有目标
        #tf.square(0-pre_box_confidence)没有目标的置信度
```

```
        no_object_loss = self.no_obj_scale *(
               1 - object_detections
        ) *(1 - detectors_mask) *tf.square(0-pre_box_confidence)

        #有目标物体的损失
        object_loss = self.obj_scale *detectors_mask *tf.square(1 - pre_box_
confidence)
        #置信度损失=没有目标物体的损失+有目标物体的损失
        confidence_loss = tf.reduce_sum(object_loss + no_object_loss)
        #分类损失
        matching_classes = tf.cast(matching_true_boxes[..., 4], 'int32')
        matching_classes = tf.one_hot(matching_classes, self.num_classes)
        classification_loss = tf.reduce_sum(
           self.class_scale *detectors_mask *tf.square(matching_classes - pre_
box_class_props)
        )
        #boxes loss,计算的是偏移值之间的误差
        box_loss = tf.reduce_sum(
            self.coordinates_scale *detectors_mask *tf.square(matching_true_
boxes[..., 0:4] - pre_d_boxes)
        )
        #所有损失
        total_loss = (confidence_loss + classification_loss + box_loss) *0.5
        return total_loss
```

此损失计算的代码较长，共由 5 个步骤构成。第 1 步，对预测出来的 y_pre 进行解码，y_pre 输出的是预测值基于 Anchor 的偏移，根据以下公式可计算出预测框的左上、右下坐标。

$$\begin{aligned} P_x &= \mathrm{Sigmoid}(t_x) + c_x \\ P_y &= \mathrm{Sigmoid}(t_y) + c_y \\ P_w &= A_w \mathrm{e}^{t_w} \\ P_h &= A_h \mathrm{e}^{t_h} \end{aligned} \tag{5-11}$$

其中，$t_x$、$t_y$ 为预测的偏移值，$c_x$、$c_y$ 为每个 13×13 的坐标值，$t_w$、$t_h$ 为预测框与 Anchor 的偏移值。$A_w$、$A_h$ 为 Anchor 的 $w$、$h$。$P_x$、$P_y$、$P_w$、$P_h$ 为解码出来的 $c_x$、$c_y$、$w$、$h$，其具体的代码如下：

```
#第 5 章/ObjectDetection/TesnsorFlow_YOLO_V2_Detected/utils/tools.py
def yolo_v2_head_decode(features, anchors=None, num_classes=20, input_size=
(416, 416)):
    """
    YOLO 预测边界框的中心点相对于网格左上角的偏移值，而每个网格有 5 个 Anchor，然后套用
公式便可得到实际位置
    features:       预测出来的值 conv。预测出来的是偏移值 [None, 13, 13, (4 + 1 + num_
classes) *5]
    anchors:        Anchor 的 widths 和 heights
    num_classes:    分类数
    :return:
    boxes : 返回 xmin,ymin,xmax,ymax,这是为了方便求背景损失的计算，实际上求位置的损
失没用这个返回值
```

```
    box_confidence: 置信度
    box_class_props: 类别,类别是进行了 Softmax 的
    """
    height, width = input_size
    assert height % 32 == 0                    #必须是 32 的倍数
    assert width % 32 == 0
    conv_height, conv_width = height //32, width //32
    if anchors is None:
        anchors = [[1.08, 1.19], [3.42, 4.41], [6.63, 11.38], [9.42, 5.11], [16.62,
10.52]]
    #即 anchors×13
    anchors = np.array(anchors) *(input_size[0] //32)
    anchor_size = tf.constant(len(anchors))
    #将输入的 b×13×13×125 reshape 成 b × 169×5×25
    features = tf.reshape(features, [features.shape[0], conv_height * conv_
width, anchor_size, num_classes + (4 + 1)])
    #因为预测出来的是相对于该左上角的偏移值,Sigmoid 函数归一化到(0,1)之间
    xy_offset = tf.nn.sigmoid(features[..., 0:2])
    #置信度
    box_confidence = tf.sigmoid(features[..., 4:5])
    #wh 偏移
    wh_offset = tf.exp(features[..., 2:4])
    #类别进行 Softmax 输出
    box_class_props = tf.nn.softmax(features[..., 5:])

    #在 feature 上面生成 anchors
    height_index = tf.range(conv_height, dtype=tf.float32)
    width_index = tf.range(conv_width, dtype=tf.float32)
    #得到网格 13×13
    x_cell, y_cell = tf.meshgrid(height_index, width_index)
    #和上面[h,w,num_anchors,num_class+5]对应
    x_cell = tf.reshape(x_cell, [1, -1, 1]) #
    y_cell = tf.reshape(y_cell, [1, -1, 1])

    #根据网格求坐标位置,套公式
    bbox_x = (x_cell + xy_offset[..., 0])                    / conv_height
    bbox_y = (y_cell + xy_offset[..., 1])                    / conv_width
    bbox_w = (anchors[:, 0] *wh_offset[..., 0])              / conv_height
    bbox_h = (anchors[:, 1] *wh_offset[..., 1])              / conv_width

    #由 cx,cy,w,h 转换成 xmin, ymin, xmax, ymax
    boxes = tf.stack(
        [
            bbox_x - bbox_w / 2,
            bbox_y - bbox_h / 2,
            bbox_x + bbox_w / 2,
            bbox_y + bbox_h / 2
        ],
        axis=3
    )
    return boxes, box_confidence, box_class_props
```

在解码代码中 x_cell 和 y_cell 为每个格子的坐标，x_cell＋xy_offset[...,0]为每个偏移值解码后的 $x$ 值，除以 conv_height＝13 是为了归一化操作，如图 5-45 所示。

```
x_cell, y_cell = tf.meshgrid(height_index, width_index)  x_cell: tf.Tensor(\n[[[ 0.]\n  [
#和上面[h,w,num_anchors,num_class+5]对应
x_cell = tf.reshape(x_cell, [1, -1, 1])  #
y_cell = tf.reshape(y_cell, [1, -1, 1])

# 根据网格求坐标位置，套公式
bbox_x = (x_cell + xy_offset[..., 0]) / conv_height  bbox_x:
bbox_y = (y_cell + xy_offset[..., 1]) / conv_width  bbox_y:
```

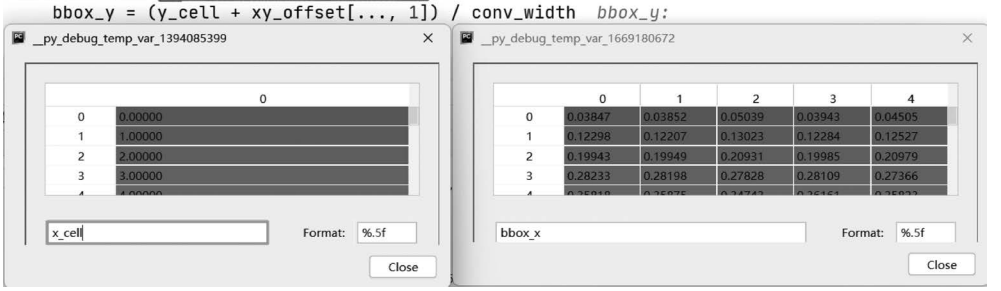



图 5-45　预测偏移值解码

损失函数的第 2、第 3 步是从预测值中得到预测的偏移值 pre_d_boxes＝tx,ty,tw,th，同时从预测解码 boxes 中得到 pre_box＝cx,cy,w,h，以便计算真实 BOX 与预测 BOX 的 IOU，如图 5-46 所示。

```
#预测出来的偏移值xy限制了值域只能在0~1
pre_d_boxes = tf.concat([tf.nn.sigmoid(features[..., 0:2]), features[..., 2:4]], axis=
########################################
# (3) 由xmin,ymin,xmax,ymax转成cx,cy,w,h, 这是为了计算IOU
# 维度调整
pre_boxes_xy = tf.reshape(pre_boxes_xy, [-1, self.conv_height, self.conv_width, self.
pre_boxes_wh = tf.reshape(pre_boxes_wh, [-1, self.conv_height, self.conv_width, self.
#为了计算IOU,需要先预测cx,cy,w,h
pre_box = tf.concat([  pre_box:
    (pre_boxes_xy - pre_boxes_wh) * 0.5,
    (pre_boxes_xy + pre_boxes_wh) * 0.5,
], axis=-1)
```

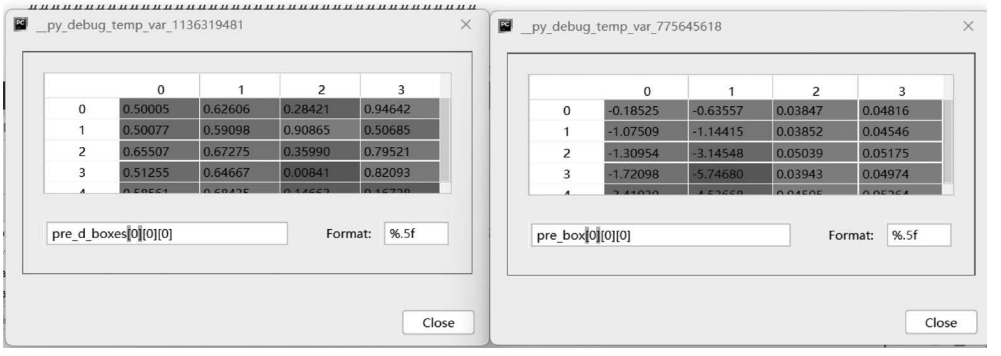



图 5-46　从预测值中获取预测偏移值

第 4 步，将解码出来的 pre_boxes 与 true_boxes 做 IOU，当 IOU＞0.5 时置为正样本 mask 的 object_detections，如图 5-47 所示。

```
true_box = tf.concat([  true_box:
    (true_boxes[..., 0:2] - true_boxes[..., 2:4]) * 0.5,
    (true_boxes[..., 0:2] + true_boxes[..., 2:4]) * 0.5,
], axis=-1)
#得到预测框与真实框之间的IOU得分
iou_result = iou(pre_box, true_box)  # 1 * 13 * 13 * 5  iou_result: tf.
0.0384654  0.04815853]\n [-1.0750892 -1.144149   0.03852105 0.04545987]\n [-1.3095378 -3.1454785  0.05038989 0.0517497
iou_score = tf.expand_dims(tf.reduce_max(iou_result, axis=-4), axis=-1)
#过滤IOU要大于指定的置信度
object_detections = iou_score > self.overlap_threshold  object_detectio
object_detections = tf.cast(object_detections, dtype=iou_score.dtype)
```

0.5

图 5-47　IOU＞0.5 时的 mask

第 5 步，根据式(5-11)计算没有目标的置信度损失、有目标的置信度损失、分类损失和偏移值之间的损失，如图 5-48 所示。

```
no_object_loss = self.no_obj_scale * (  no_object_loss:
        1 - object_detections
) * (1 - detectors_mask) * tf.square(0-pre_box_confidence)

#有目标物体的损失
object_loss = self.obj_scale * detectors_mask * tf.square(1 - pre_box_confidence)  object_loss:
#置信度损失=没有目标物体的损失+有目标物体的损失
confidence_loss = tf.reduce_sum(object_loss + no_object_loss)  confidence_loss: tf.Tensor(325.76306,
#分类损失
matching_classes = tf.cast(matching_true_boxes[..., 4], 'int32')  matching_classes:
matching_classes = tf.one_hot(matching_classes, self.num_classes)
classification_loss = tf.reduce_sum(  classification_loss: tf.Tensor(0.93039364, shape=(), dtype=floa
    self.class_scale * detectors_mask * tf.square(matching_classes - pre_box_class_props)
)
# boxes loss，计算的是偏移量之间的误差
box_loss = tf.reduce_sum(  box_loss: tf.Tensor(0.4022428, shape=(), dtype=float32)
    self.coordinates_scale * detectors_mask * tf.square(matching_true_boxes[..., 0:4] - pre_d_boxes)
)
# 所有损失
total_loss = (confidence_loss + classification_loss + box_loss) * 0.5  total_loss:
```

图 5-48　损失计算

在训练方面，先使用 224×224 训练 DarkNet19 的权重并迁移到 YOLOv2 的主干中，然后使用不同尺度的大小进行训练，例如 320、352、608，然后在 416 上进行微调，这样训练将使模型具备不同分辨率图像的泛化能力，稳健性更强，关键代码如下：

```
#第 5 章/ObjectDetection/TesnsorFlow_YOLO_V2_Detected/utils/train_YOLOv2.py
if __name__ == "__main__":
    input_shape = [
        (320, 320, 3),
        (352, 352, 3),
```

```
            (608, 608, 3),
            (416, 416, 3)
        ]
        #每个尺度的训练次数
        train_EPOCH = [10, 20, 30, 50]
        #每个尺度训练的学习率
        learn = [1e-3, 1e-4, 1e-5, 1e-5]
        old_epoch = 0                    #上一次的 epoch num
        old_name = 0                     #上一次训练的权重名
        #构建不同尺度的图形模型训练
        for im_shape, epoch, lr, save_dir in zip(input_shape, train_EPOCH, learn,
save_weights):
            model = yolo_v2(im_shape, num_classes, is_class=False)
            #如果已经有训练好的其他尺度的权重文件,则作为下一个尺度的初始权重
            if not old_name:
                exist_weights = f"../weights/last_{old_name}.h5"
                if os.path.exists(exist_weights):
                    model.load_weights(exist_weights, by_name=True, skip_mismatch=True)
            else:
                #第 1 个尺寸调入分类的训练权重
                model.load_weights('../weights/darknet_10_224.h5', by_name=True,
skip_mismatch=True)
            #每隔 3 个 epoch 设置检测点并保存最优模型
            checkpoint = ModelCheckpoint(
                save_dir,
                monitor='val_loss',
                save_weights_only=False,
                save_best_only=True,
                period=check_step_epoch
            )
            #设置优化器
            model.compile(
                optimizer=Adam(learning_rate=lr),
                loss=MultiAngleLoss(num_classes=num_classes, input_size=im_shape
[:2]).yolo_v2_loss,
                run_eagerly=False,       #是否启用调试模型
            )
            #加载数据类
            data_object = DataProcessingAndEnhancement(
                train_lines,
                val_lines,
                num_classes,
                batch_size=BATCH_SIZE,
                input_shape=im_shape[0:2]
            )
            #训练
            model.fit(
                data_object.generate(True),
                steps_per_epoch=num_train //BATCH_SIZE,
                validation_data=data_object.generate(False),
                validation_steps=num_val //BATCH_SIZE,
                epochs=epoch,
```

```
            initial_epoch=old_epoch,
            callbacks=[logging, checkpoint]
        )
        old_epoch = epoch
        old_name = im_shape[0]
        #保存模型
        model.save(f"../weights/last_{old_name}.h5")
```

更多更详细的代码可参考随书代码。

## 5.6.6　代码实战预测推理

YOLOv2 引入了 5 个 Anchor 并根据式(5-11)对预测值进行解码操作,代码如下:

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V2_Detected/detected/detected.py
class Detected():
    def __init__(self, model_path, input_size):
        self.model = load_model(
            model_path,
            custom_objects={'compute_loss': MultiAngleLoss(2).compute_loss}
        ) #读取模型和权重
        self.confidence_threshold = 0.01                    #有无目标置信度
        self.class_prob = [0.5, 0.5]                        #两个类别的分类阈值
        self.nms_threshold = 0.5                            #NMS 的阈值
        self.input_size = input_size
        self.anchors = np.array([
            (1.3221, 1.73145), (3.19275, 4.00944),
            (5.05587, 8.09892), (9.47112, 4.84053), (11.2364, 10.0071)
        ])

    def readImg(self, img_path=None):
        img = cv2.imread(img_path)                          #读取要预测的图片
        #将图片转换到 300×300
        self.img, _ = u.letterbox_image(img, self.input_size, [])
        self.old_img = self.img.copy()

    def forward(self):
        #升成 4 维
        img_tensor = tf.expand_dims(self.img / 255.0, axis=0)
        #前向传播
        self.output = self.model.predict(img_tensor)
        self.h, self.w = self.output.shape[1:3] #1*13*13*(4+1+num_class)*5

    def generate_anchor(self):
        #使用均分的方法生成格子。一共有 h×w 个格子,即 169 个格子
        self.lin_x = np.ascontiguousarray(np.linspace(0, self.w - 1, self.w).
repeat(self.h)).reshape(self.h *self.w)
        self.lin_y = np.ascontiguousarray(np.linspace(0, self.h - 1, self.h).
repeat(self.w)).T.reshape(self.h *self.w)
        #得到锚框的 w 和 h,因为一共有 5 个 Anchor,所以 reshape 成[1,5,1]
        self.anchor_w = np.ascontiguousarray(self.anchors[..., 0]).reshape([-1,
5, 1])
```

```
        self.anchor_h = np.ascontiguousarray(self.anchors[..., 1]).reshape([-1,
5, 1])

    def _decode(self):
        #对预测出来的结果解码
        #YOLOv2 的输出(4+1+num_class) ×5 个 Anchor
        self.b = self.output.shape[0]
        #变成[b,5,(4+1+2),169)
        logits = np.reshape(self.output, [self.b, 5, -1, self.h *self.w])
        self.result = np.zeros(logits.shape)
        #套用公式进行解码,Sigmoid()函数进行值域限制
        #根据 Anchor 得到 cx 和 cy
        self.result[:, :, 0, :] = (tf.sigmoid(logits[:, :, 0, :]).NumPy() + self.
lin_x) / self.w
        self.result[:, :, 1, :] = (tf.sigmoid(logits[:, :, 1, :]).NumPy() + self.
lin_y) / self.h
        #根据 Anchor 得到 w 和 h
        self.result[:, :, 2, :] = (tf.exp(logits[:, :, 2, :]).NumPy() *self.anchor_
w) / self.w
        self.result[:, :, 3, :] = (tf.exp(logits[:, :, 3, :]).NumPy() *self.anchor_
h) / self.h
        #得到置信度
        self.result[:, :, 4, :] = tf.sigmoid(logits[:, :, 4, :]).NumPy()
        #得到分类
        self.result[:, :, 5:, :] = tf.nn.softmax(logits[:, :, 5:, :]).NumPy()

    def classification_filtering(self):
        self._decode()
        #取最大的分类得分的下标,此时 result 为[b,5,(4+1+num_class),169]
        class_score = self.result[:, :, 5:, :]          #[1,5,num_class,169]
        class_label = np.argmax(class_score, axis=2)    #[1,5,169]
        #最大的分类得分值
        score = np.max(class_score, axis=2)             #[1,5,169]
        #当前格子的置信度
        conf = self.result[:, :, 4, :]                  #[1,5,169]
        #每个格子最后的得分为置信度*分类得分
        prob = score *conf                              #[1,5,169]
        #根据阈值,得到评分
        score_mask = prob > self.confidence_threshold  #[1,5,169]
        if np.sum(score_mask.reshape([-1])) > 0:
            #只有转置成 [b×5×(4+1+num_class) ×169],才能根据 score_mask 进行取值,
            #最后需要得到[b,5,169,7]中的 7
            self.result = np.reshape(self.result, [self.b, 5, self.h *self.w, -1])
            boxes = self.result[score_mask][..., :4] #根据 score_mask 过滤得到 boxes
            cls_scores = prob[score_mask] #根据 score_mask 过滤得到 cls_scores
            idx = class_label[score_mask] #根据 score_mask 过滤得到 labels index
            #根据类别进行非极大值抑制
            for label in range(len(self.class_prob)):
                #如果 class_labels_all==label,则取当前 label 中的信息
                mask = idx == label
                #如果都不是当前 label,则跳过
                if np.sum(mask) == 0: continue
```

```
                #由 cx,cy,w,h 转换成 xmin,ymin,xmax,ymax
                xyxy_box = u.cxcy2xyxy(boxes[mask][..., :4])
                cat_boxes = np.concatenate([xyxy_box, cls_scores[mask].reshape
([-1, 1])], axis=-1)
                #NMS
                index = u.nms(cat_boxes, self.nms_threshold)
                #绘框
                self.old_img = u.draw_box(self.old_img, cat_boxes[index])

            #最后结果
            u.show(self.old_img)


if __name__ == "__main__":
    d = Detected(r'../weights/chk416', input_size=[416, 416])
    d.readImg('../../val_data/pexels-photo-5211438.jpeg')
    d.forward()
    d.generate_anchor()
    d.classification_filtering()
```

在__init__()中增加 self.anchors 以描述先验框的 wh 值，forward()前向传播后得到的 shape 为[b,13,13,(4+1+num_class)*5]，如图 5-49 所示。

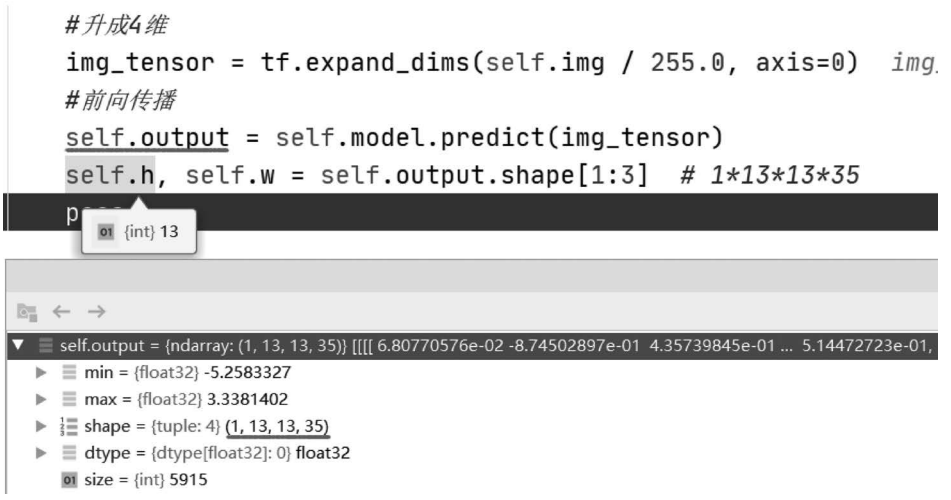



图 5-49　YOLOv2 前向传播输出

在 generate_anchor()中根据 np.linspace(0,self.w−1,self.w)指令将输入图像均分为 13 份，为了组成完整坐标，所以 repeat(self.h)重复了 13 次，摊平后 reshape(self.h*self.w)得到 169 个坐标并存储在 self.lin_x 变量中，以同样的方法得到 self.lin_y。在 self.anchor_w、self.anchor_h 分别得到锚框的宽和高，如图 5-50 所示。

_decode()首先将 self.output 由[b,13,13,(4+1+num_class)*5]变成[b,5,(4+1+num_class),13*13]，然后根据 self.result[:, :, 0, :]的位置对于预测的 offset 值根据式(5-11)进行解码，并采用 tf.sigmoid()函数进行值域限定，如图 5-51 所示。


图 5-50 YOLOv2 画格子

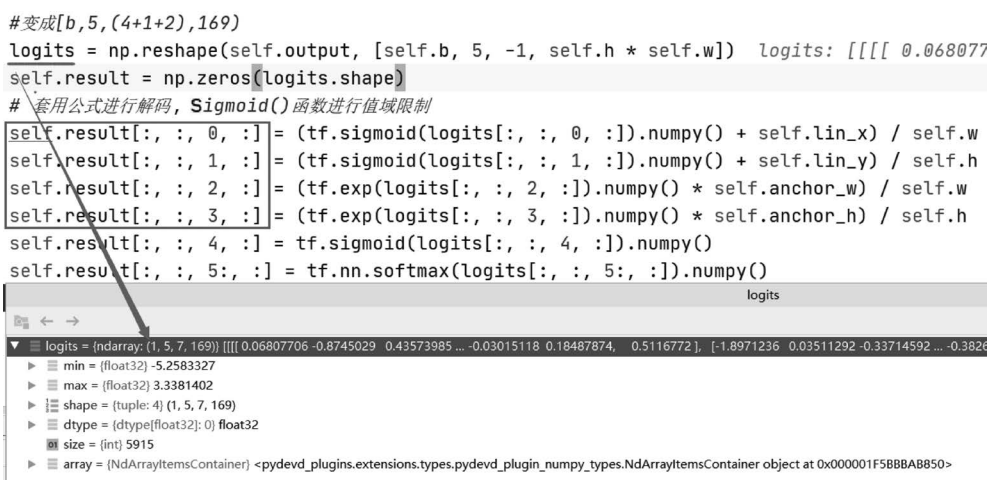



图 5-51 YOLOv2 解码操作

在 classification_filtering()中对于解码出的内容进行 score_mask＝prob＞self.confidence_threshold 的过滤，当置信度×分类概率的值大于 self.confidence_threshold 时，score_mask 为 True，否则为 False。此时由于 self.result 为[b,5,(4＋1＋num_class),169]，而 score_mask 为[1,5,169]，所以需要将 self.result 的 shape 转换 np.reshape(self.result,[self.b,5,self.h＊self.w,－1])变成[b,5,169,(4＋1＋num_class)]，从而 boxes＝self.result[score_mask][...,:4]得到过滤后的 boxes 内容，继而得到分类概率 cls_scores、置信度概率 cls_scores，如图 5-52 所示。

遍历 for label in range(len(self.class_prob))每个类别的下标，当 mask＝idx＝＝label 时从 boxes 中取出当前类别的 boxes，然后通过 xyxy_box＝u.cxcy2xyxy(boxes[mask][...,:4])，由 cx,cy,w,h 转换成 xmin,ymin,xmax,ymax，最后进行 NMS，从而得出预测框的内容，如图 5-53 所示。

**总结**

YOLOv2 采用 Anchor 机制，并划分为 13×13 个格子，每个格子有 5 个 Anchor，由 GT

```
#根据阈值，得到评分
score_mask = prob > self.confidence_threshold  # [1,5,169]  score_mask: [[[False False Fals
if np.sum(score_mask.reshape([-1])) > 0:
    #只有转置成 [b*5*(4+1+num_class)*169],才能根据score_mask进行取值，最后需要得到[b,5,169,7]中的7
    self.result = np.reshape(self.result, [self.b, 5, self.h * self.w, -1])
    boxes = self.result[score_mask][..., :4]  #根据score_mask过滤得到boxes  boxes: [[1.908541
    cls_scores = prob[score_mask]  #根据score_mask过滤得到cls_scores  cls_scores: [0.01579514
    idx = class_label[score_mask]  #根据score_mask过滤得到labels index  idx: [0 1 1 0 1 0 1 1
    #根据类别进行非极大值抑制
```

boxes

boxes = {ndarray: (61, 4)} [[1.90854153e-01 1.83761058e-01 2.02092652e-01 2.05399843e-01], [2.71270523e-01 2.79190224e-01 2.78487893e-01 3.46667583e-01], [9.59
- min = {float64} 4.4993972551310435e-05
- max = {float64} 5.983754461376483
- shape = {tuple: 2} (61, 4)
- dtype = {dtype[float64]: 0} float64
- size = {int} 244
- array = {NdArrayItemsContainer} <pydevd_plugins.extensions.types.pydevd_plugin_numpy_types.NdArrayItemsContainer object at 0x000001F5C4AE2D00>

图 5-52　YOLOv2 过滤大于条件概率值

```
for label in range(len(self.class_prob)):  label: 0
    #如果class_labels_all==label，则取当前label中的信息
    mask = idx == label  mask: [ True False False  True False  True False False False Fa
    #如果都不是当前label，则跳过
    if np.sum(mask) == 0: continue
    #由cx,cy,w,h转换成xmin,ymin,xmax,ymax
    xyxy_box = u.cxcy2xyxy(boxes[mask][..., :4])  xyxy_box: [[ 8.98078267e-02  8.1061136
    cat_boxes = np.concatenate([xyxy_box, cls_scores[mask].reshape([-1, 1])], axis=-1)
    # nms
    index = u.nms(cat_boxes, self.nms_threshold)
```

xyxy_box

xyxy_box = {ndarray: (28, 4)} [[ 8.98078267e-02  8.10611363e-02  2.91900479e-01  2.86460980e-01], [ 2.17705798e-01  2.78295744e-01  7.99659364e-01  8.82276
- min = {float64} -0.723978466964685
- max = {float64} 6.114831115690561
- shape = {tuple: 2} (28, 4)
- dtype = {dtype[float64]: 0} float64
- size = {int} 112
- array = {NdArrayItemsContainer} <pydevd_plugins.extensions.types.pydevd_plugin_numpy_types.NdArrayItemsContainer object at 0x000001F5C4AD8A00>

图 5-53　YOLOv2 根据类别索引过滤

BOX 落在某个格子所在的最佳 Anchor 进行预测。Anchor 的生成采用聚类算法获得，该方法在 YOLO 系列得到延续。

**练习**

运行并调试本节代码，理解算法的设计与代码的结合，重点梳理本算法的实现方法。

## 5.7　单阶段速度快多检测头网络 YOLOv3

### 5.7.1　模型介绍

YOLOv3 由 Joseph Redmon 等在 2018 年发表的论文 *YOLOv3: An Incremental Improvement* 中提出，其主要特点是在 YOLOv2 的基础上将主干特征提取网络换成

DarkNet-53，使用了 FPN。检测头变为 3 个，建议框的数量从 YOLOv2 的 5 个变成 9 个，每个检测头分配 3 个建议框。损失函数方面，分类损失和置信度损失从均方差更换为交叉熵，其结构如图 5-54 所示。

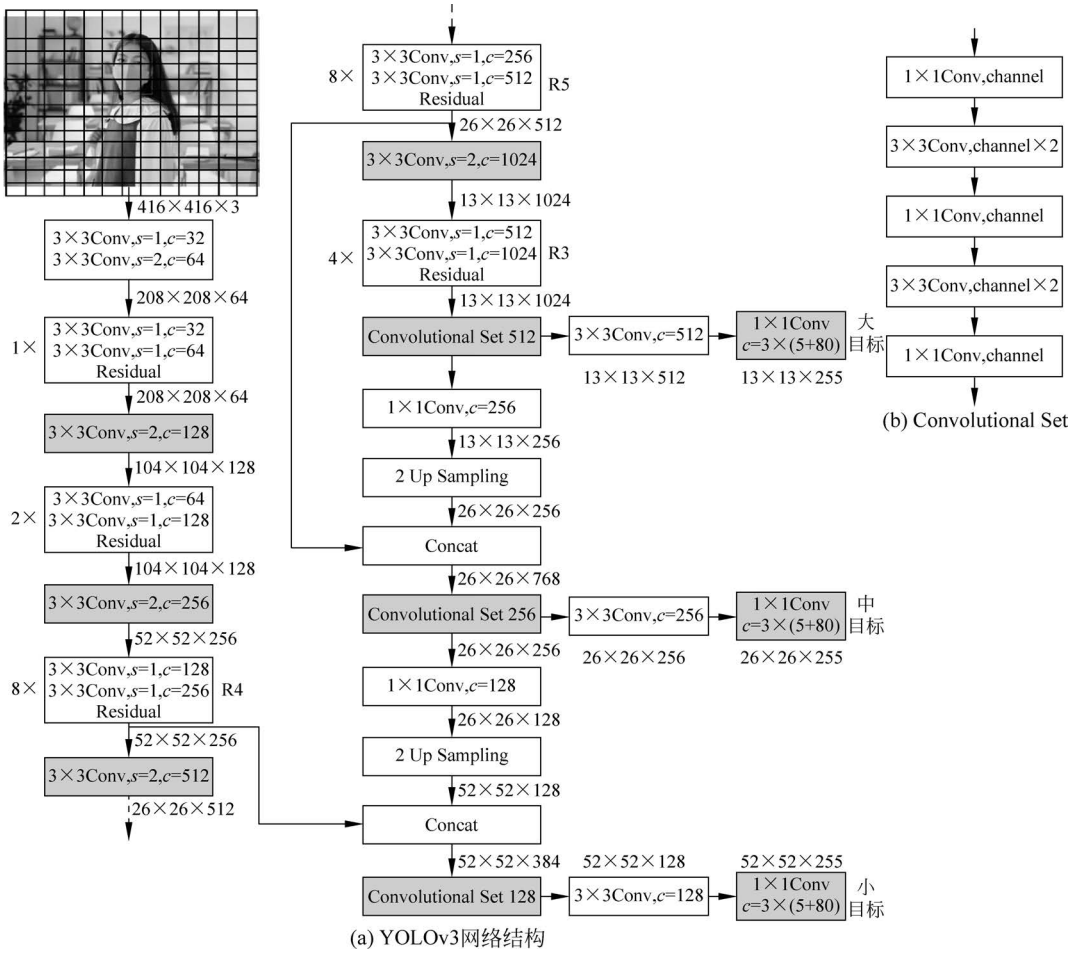



图 5-54　YOLOv3 结构图

YOLOv3 在主干特征提取层引入了 Residual 差结构，可以使网络更稀疏、网络层数更深，以便更好地提取特征信息，如图 5-54 所示，分别重复 1、2、8、8、4 次。输出特征 13×13×1024 后经过 Convolutional Set 512 层，接 3×3、1×1 卷积输出 13×13×3 建议框(4 偏移位置＋1 置信度＋80 个分类)作为第 1 个检测头，检测大目标(13×13×255 的感受野最大，所以检测大目标)。

Convolutional Set 512 后经过 1×1 卷积、2 倍 Up Sampling 上采集得 26×26×256，与 R5 的输出 Concat 得 26×26×768，然后 Convolutional Set 256，接 3×3、1×1 卷积输出 26×26×3 建议框(4 偏移位置＋1 置信度＋80 个分类)作为第 2 个检测头，检测中目标。

Convolutional Set 256 后经过 1×1 卷积、2 倍 Up Sampling 上采集得 52×52×128，与 R5 的输出 Concat 得 26×26×384，然后 Convolutional Set 128，接 3×3、1×1 卷积输出 52×52×3 建议框(4 偏移位置＋1 置信度＋80 个分类)作为第 3 个检测头，检测小目标。

在 CNN 结构中深层网络语义特征信息丰富，浅层特征几何信息丰富，在目标检测任务中特征提取是一个很重要的问题，深层网络虽然能够得到丰富的语义特征信息，但是由于特征图的尺寸较小，所以包含的几何信息较少，不利于物体的位置检测，潜层网络虽然包含了丰富的几何信息，但是图像的语义信息较少，又不利于图像的分类预测，这个问题尤其在小目标检测任务中表现尤为明显。

回顾 Faster R-CNN、YOLOv1 等只使用最深层的特征图信息，单尺度特征图限制了模型的检测能力，尤其是那些较小的样本或者数量较少的建议框尺寸。SSD 利用卷积的层次结构，从 VGG 中的 Conv4_3、Conv7、Conv8_2、Conv9_2、Conv10_2、Conv11_2 得到多尺度特征信息，该方法虽然能提高精度并且检测速度略有下降，但由于没有使用更加深层的特征信息，所以对于检测小目标仍然不够稳健，如图 5-55 所示。

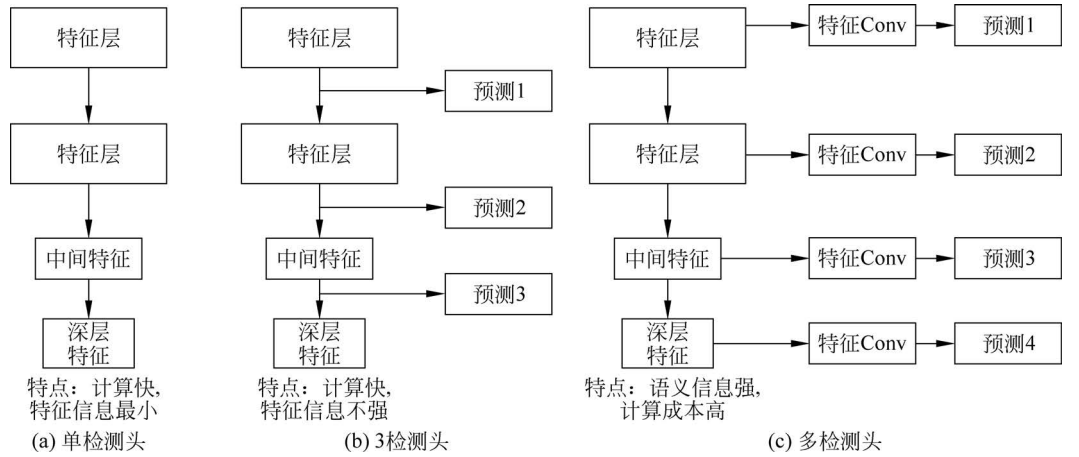



图 5-55　不同的检测头结构

FPN 在 SSD 的结构上不仅使用了深层特征图的信息，并且浅层网络的特征信息也被使用，并通过自底向上、自顶向下及横向连接的方式对这些特征图的信息进行整合，在提升精度的同时检测速度也没有较大降低，其结构如图 5-56 所示。

FPN 自上而下、由底而上进行了特征信息的融合，将语义信息与几何信息进行融合，有助于小目标的特征信息提取，能够显著地提高小目标的检测能力。

在 FPN 进行上采样时，可选择最近邻插值、双线性插值的方法放大图像。最近邻插值在放大图时补充的像素是最近邻的像素值，由于方法简单，所以处理速度很快，但是当放大图像时会有锯齿，画质较低，如图 5-57 所示。

已知 $A=(x_0,y_0)$、$B=(x_1,y_1)$，将 $(x_0,y_0)$ 和 $(x_1,y_1)$ 连成一条直线，求区间 $(x_0,x_1)$ 上某一点 $x$ 在该直线上的 $y$ 值，这个求解过程就是单线性插值的过程，如图 5-58 所示。

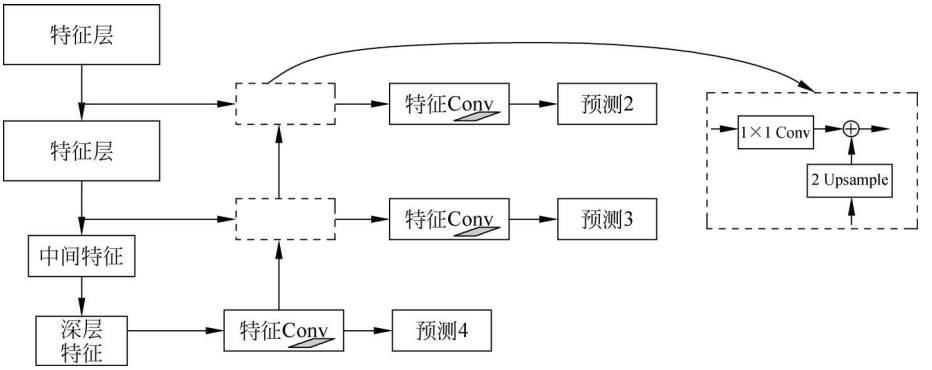



图 5-56 FPN 结构

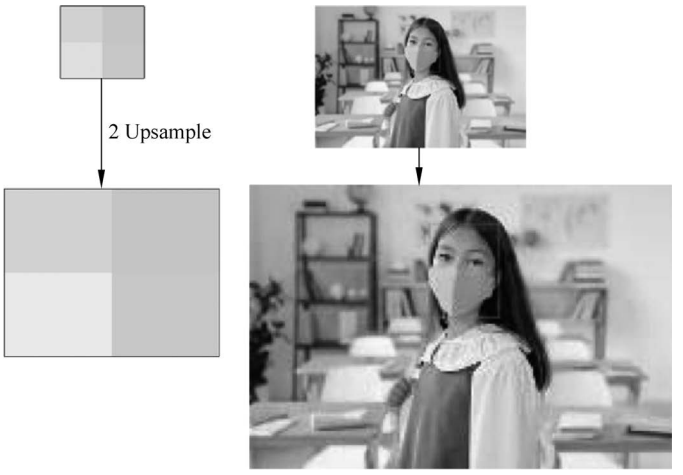



图 5-57 最近邻插值

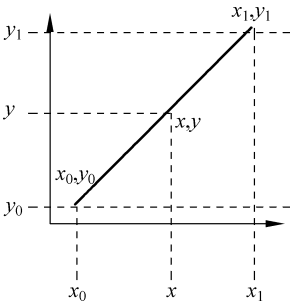



图 5-58 单线性插值

因为$\frac{y_1-y}{x_1-y}=\frac{y-y_0}{x-x_0}$,所以 $y=y_0+\frac{x_1-y}{x_1-x}(x-x_0)=\frac{x-x_0}{x_1-x_0}y_1+\frac{x_1-x}{x_1-x_0}y_0$,同样已知 $y$ 也可求出 $x$,代码如下:

```
#第 5 章/ObjectDetection/TesnsorFlow_YOLO_V3_Detected/utils/interp.py
def interp():
    #x 值
    x = np.arange(0, 10, 0.5)
    y = x **2
    #插入点为 xvals
    x_vals = np.linspace(0, 10, 5)
    y_interp = np.interp(x_vals, x, y)
    for ix,iy in zip(x_vals,y_interp):
        plt.text(ix,iy+1,f"{ix},{iy}")
    plt.plot(x, y, 'o',label="xy 轴")
    plt.plot(x_vals, y_interp, '-x',label='线性插值')
    plt.legend()
    plt.show()
```

调用代码，运行后单线性插值如图 5-59 所示。

在卷积图像中图像的维度是三维的，在进行上采样时就需要用到双线性插值，如图 5-60 所示。

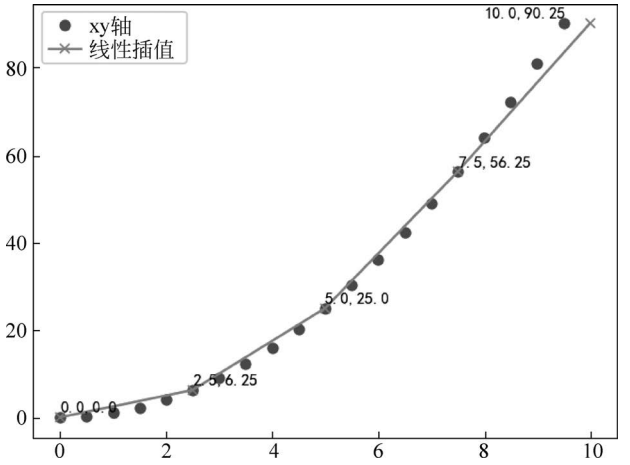



图 5-59　单线性插值随 x_vals 变化

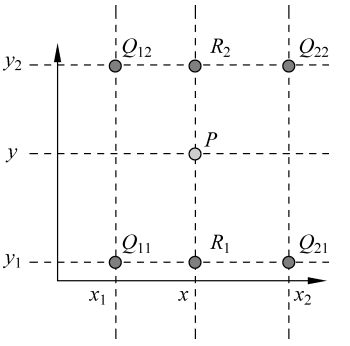



图 5-60　双线性插值算法

所谓双线性插值，原理与线性插值相同，如图 5-60 中已知点 $Q_{11}=(x_1,y_1)$、$Q_{12}=(x_1,y_2)$、$Q_{21}=(x_2,y_1)$和 $Q_{22}=(x_2,y_2)$共四个点的值，求函数 $f(x,y)$在 $P=(x,y)$的值。

首先做两次线性插值，分别求出点 $R_1=(x,y_1)$和 $R_2=(x,y_2)$的像素值，然后用这两个点再做 1 次线性插值以求出 $P=(x,y)$的像素值。对于 $Q_{11}$ 和 $Q_{12}$ 来讲，它们连成线的纵坐标是相同的，所以可以忽略这个纵坐标的影响(但这个影响是存在的，所以线性插值是近似的)，而用当前点的像素值直接代替纵坐标，所以 $Q_{11}=(x,f(Q_{11}))$。$f(R_1)\approx\dfrac{x-x_1}{x_2-x_1}f(Q_{21})+\dfrac{x_2-x}{x_2-x_1}f(Q_{11})$，$f(R_2)\approx\dfrac{x-x_1}{x_2-x_1}f(Q_{22})+\dfrac{x_2-x}{x_2-x_1}f(Q_{12})$，$f(P)\approx\dfrac{y-y_1}{y_2-y_1}f(R_2)+\dfrac{y_2-y}{y_2-y_1}f(R_1)=\begin{bmatrix}\dfrac{y-y_1}{y_2-y_1},\dfrac{y_2-y}{y_2-y_1}\end{bmatrix}\begin{bmatrix}f(Q_{22}), & f(Q_{12})\\ f(Q_{22}), & f(Q_{12})\end{bmatrix}\begin{bmatrix}\dfrac{x-x_1}{x_2-x_1},\dfrac{x_2-x}{x_2-x_1}\end{bmatrix}^{\mathrm{T}}$，在双线性插值上采样时 $x_2-x_1=1$、$y_2-y_1=1$。

双线性插值是一种比较好的图像缩放算法，它充分利用一源图像中虚拟的点四周的 4 个真实存在的像素来共同决定目标图像中的一像素，这样所生成的新图效果更好，过渡更自然，边缘更光滑，如图 5-61 所示的效果比图 5-57 要好得多。

YOLOv3 虽然有 3 个检测头，但是它正样本的选取仍然跟 YOLOv2 一样，即从检测头 13×13、26×26、52×52 生成的 Anchor 与标注 BOX 做 IOU，取最大的 IOU 得分作为正样本，其他样本作为负样本。稍有不同之处，3 个检测头分配的建议框尺寸有变化，13×13 分配大一些的建议框，如[116，90]、[156，198]、[373，326]；26×26 分配中等大小的建议框，如[30，61]、[62，45]、[59，119]；52×52 分配小目标的建议框，如[10，13]、[16，30]、[33，23]。这些建议框仍然通过聚类得到，一共有 9 个 Anchor。


图 5-61 双线性插值效果图

在损失函数方面，将 YOLOv2 中的分类损失更改为二元交叉熵、位置损失，仍然使用均方差损失，置信度损失使用交叉熵，其公式如下：

$$\begin{aligned}\text{Loss}=&\lambda_{\text{coord}}\sum_{i=0}^{s^2}\sum_{j=0}^{B}l_{i,j}^{\text{obj}}[(b_x-\hat{b}_x)^2+(b_y-\hat{b}_y)^2+(b_w-\hat{b}_w)^2+\\&(b_h-\hat{b}_h)^2]+\sum_{i=0}^{s^2}\sum_{j=0}^{B}l_{i,j}^{\text{obj}}[-\log(p_c)]+\\&\lambda_{\text{noobj}}\sum_{i=0}^{s^2}\sum_{j=0}^{B}l_{i,j}^{\text{noobj}}[-\log(1-p_c)]+\sum_{i=1}^{n}\text{BCE}(\hat{c}_i,c_i)\end{aligned}\tag{5-12}$$

其中，$l_{i,j}^{\text{obj}}$ 表示每个格子中有目标；$b_x$、$b_y$、$b_w$、$b_h$ 代表真实 BOX 的位置，$\hat{b}_x$、$\hat{b}_y$、$\hat{b}_w$、$\hat{b}_h$ 代表预测的 BOX 位置；$-\log(p_c)$表示有目标的置信度、$-\log(1-p_c)$代表没有目标的置信度；$\text{BCE}(\hat{c}_i,c_i)$为二次交叉熵，即逻辑回归的损失函数，每个类为 0 或者 1；原作者论文并没有说明具体的损失函数，不同的代码作者所使用的损失函数可能有所不同。

## 5.7.2 代码实战模型搭建

因为 YOLOv3 使用了残差结构，所以需要对 Residual 进行实现，代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V3_Detected/utils/conv_utils.py

class ResidualBlock(layers.Layer):
```

```
    def __init__(self, num_filters, num_blocks, **kwargs):
        """
        基本残差模块,即 a 3 × 3 channels,接着 b 1 × 1 channels/2,再 c 3 × 3 channels
        然后 a + c
        :param num_filters: channels 数
        :param num_blocks: 重复次数
        """
        super(ResidualBlock, self).__init__(**kwargs)
        #3×3 卷积,s=2 因为 padding=same,所以尺寸不变
        self.conv1 = ConvBnLeakRelu(num_filters, kernel_size=3, strides=2)
        #1×1 卷积,s=1,outchannel 是输入的 1/2
        self.block1 = ConvBnLeakRelu(num_filters //2, kernel_size=1, strides=1,
padding='valid')
        #1×1 卷积,s=1
        self.block2 = ConvBnLeakRelu(num_filters, kernel_size=3, strides=1)
        self.add = layers.Add()
        self.num_blocks = num_blocks

    def call(self, inputs, *args, **kwargs):
        #输入的卷积
        x1 = self.conv1(inputs)
        #残差可能会执行多次
        for i in range(self.num_blocks):
            #1×1
            y = self.block1(x1)
            #1×1
            y = self.block2(y)
            #进行 add 残差
            x1 = self.add([x1, y])
        return x1
```

ConvBnLeakRelu 中已对 Conv、BN、LeakRelu 进行了封装。self. num_blocks 用来控制残差重复的次数。

然后根据结构图 5-54 实现 DarkNet-53,代码如下:

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V3_Detected/backbone/YOLOv3.py
def darknet_body(input_x):
    """根据结构图实现 DarkNet-53 部分"""
    #input 416 × 416 × 3
    x = ConvBnLeakRelu(32, 3, 1)(input_x)
    #重复的次数是 1,2,8,8,4
    #-> 208 × 208 × 64
    x = ResidualBlock(64, 1)(x)
    #-> 104 × 104 × 128
    x = ResidualBlock(128, 2)(x)
    #-> 52 × 52 × 256
    result4 = ResidualBlock(256, 8)(x)
    #-> 26 × 26 × 512
    result5 = ResidualBlock(512, 8)(result4)
    #-> 13 × 13 × 1024
    result6 = ResidualBlock(1024, 4)(result5)
    return result4, result5, result6
```

因为 YOLOv3 使用 13×13×1024 作为第 1 个检测头，使用 26×26×512、52×52×256 作为 FPN 上采样 Concat 层，所以输出为 result4、result5 和 result6。

在图 5-54 中还存在 Convolutional Set 结构，该结构主要是由 1×1、3×3、1×1、3×3、1×1 卷积构成的，代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V3_Detected/utils/conv_utils.py
class ConvolutionSet(layers.Layer):
    """
    即 1 × 1 -> 3 × 3 -> 1 × 1 -> 3 × 3 -> 1 × 1
    """
    def __init__(self, number_filters, **kwargs):
        super(ConvolutionSet, self).__init__(**kwargs)
        self.conv1x1_1 = ConvBnLeakRelu(number_filters, kernel_size=1, strides=1)
        self.conv3x3_2 = ConvBnLeakRelu(number_filters *2, kernel_size=3,
strides=1)
        self.conv1x1_3 = ConvBnLeakRelu(number_filters, kernel_size=1, strides=1)
        self.conv3x3_4 = ConvBnLeakRelu(number_filters *2, kernel_size=3,
strides=1)
        self.conv1x1_5 = ConvBnLeakRelu(number_filters, kernel_size=1, strides=1)

    def call(self, inputs, *args, **kwargs):
        x = self.conv1x1_1(inputs)
        x = self.conv3x3_2(x)
        x = self.conv1x1_3(x)
        x = self.conv3x3_4(x)
        x = self.conv1x1_5(x)
        return x
```

在检测头方面由 3×3、1×1 卷积组成，并且 1×1 卷积的输出采用线性模型，不经过 ReLU 也不经过 BN，代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V3_Detected/utils/conv_utils.py
class Yolo_V3_Pre_Head(layers.Layer):
    """预测的头部，即先经过一个 3×3 卷积，再经过一个 1×1 卷积"""
    def __init__(self, number_filters, classes_filters, **kwargs):
        super(Yolo_V3_Pre_Head, self).__init__(**kwargs)
        self.conv3 = ConvBnLeakRelu(number_filters, kernel_size=3, strides=1)
        self.out = ConvBnLeakRelu(classes_filters, kernel_size=1, is_relu=
False, is_bn=False, strides=1)

    def call(self, inputs, *args, **kwargs):
        x = self.conv3(inputs)
        x = self.out(x)
        return x
```

基于以上结构的封装，然后组合在 yolo-v3()函数中实现前向传播，代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V3_Detected/backbone/yolo_v3.py
def yolo_v3(input_shape, anchors=None, class_num=80, is_class=False):
    if anchors is None:
```

```
        anchors_mask = np.array([
            [[10, 13], [16, 30], [33, 23]],                    #小目标
            [[30, 61], [62, 45], [59, 119]],                   #中目标
            [[116, 90], [156, 198], [373, 326]]                #大目标
        ])
    else:
        anchors_mask = anchors
    #输入层
    input_x = layers.Input(shape=input_shape, dtype='float32')
    #调用 DarkNet-53
    r4, r5, r6 = darknet_body(input_x)
    #经过基本的 DarkNet 后,先经过一个 ConvolutionalSet,进入第 1 个预测
    x = ConvolutionSet(512)(r6)
    #第 1 个预测结果,小目标
    p5 = Yolo_V3_Pre_Head(512, len(anchors_mask[0]) *(class_num + 4 + 1))(x)
                                        #1 个置信度 + 4 个 bbox,每个 anchors 有 3 个尺度
    #第 1 个上采样
    p5_1x1 = ConvBnLeakRelu(256, kernel_size=1, strides=1)(p5)
    p5_up = layers.UpSampling2D(size=2)(p5_1x1)            #2 倍上采样
    x = layers.Concatenate()([p5_up, r5])                  #26×26
    x = ConvolutionSet(256)(x)
    #第 1 个预测,中目标
    p6 = Yolo_V3_Pre_Head(256, len(anchors_mask[1]) *(class_num + 4 + 1))(x)
    #第 1 个上采样
    p6_1x1 = ConvBnLeakRelu(128, kernel_size=1, strides=1)(p6)
    p6_up = layers.UpSampling2D(size=2)(p6_1x1)
    x = layers.Concatenate()([p6_up, r4])                  #52×52
    x = ConvolutionSet(128)(x)
    #第 2 个预测,大目标
    p7 = Yolo_V3_Pre_Head(128, len(anchors_mask[2]) *(class_num + 4 + 1))(x)
    return Model(inputs=input_x, outputs=[p5, p6, p7])
```

建议框 anchors_mask 默认初始设置为 9 个,其中每个检测头为 3 个。r4、r5 和 r6 分别对应 52×52×256、26×26×512 和 13×13×1024,所以 r6 经过 ConvolutionSet 后成为 p5 的输入,p5 的输出为 len(anchors_mask[0]) *(class_num+4+1) * 512; p5 经过 2 倍上采样(默认为最近邻),然后由 layers.Concatenate()([p5_up,r5])得到 26×26×768 并经过 ConvolutionSet 后成为 p6 的输入,p6 的输出为 len(anchors_mask[0]) *(class_num+4+1) * 256; p6 经过 2 倍上采样,然后由 layers.Concatenate()([p6_up,r4])得到 52×52×384,并经过 ConvolutionSet 后成为 p7 的输入,p7 的输出为 len(anchors_mask[0]) *(class_num+4+1) * 128。

### 5.7.3　代码实战建议框的生成

将真实框转换为 GT BOX 与 Anchor 的偏移,其实现思路是遍历每个检测头和每个 Anchor 以获取所在 $j$、$i$ 格子最佳 IOU 得分的 Anchor 作为正样本,计算偏移后赋给 true_box,代码如下:

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V3_Detected/utils/tools.py

def YOLOv3_v3_true_encode_box(
        true_boxes,                                 #传 GT BOX
        anchors=None,                               #新的 Anchor
        class_num=80,                               #类别数
        input_size=(416, 416),                      #默认尺寸
        ratio=[32, 16, 8]                           #416//13, 416//26, 416//52
):
    #将 3 个检测头 Anchor 与 GT BOX 最大 IOU 设为正样本
    if anchors is None:
        anchors = np.array([
            [[10, 13], [16, 30], [33, 23]],         #小目标
            [[30, 61], [62, 45], [59, 119]],        #中目标
            [[116, 90], [156, 198], [373, 326]]     #大目标
        ])
    #Anchor 的数量
    detected_num = anchors.shape[0]
    #用来存储 3 个检测头正样本结果
    grid_shape = []
    #用来存储 3 个检测头 GT BOX 结果
    true_boxes2 = []
    #升维
    true_boxes = np.expand_dims(true_boxes, axis=0)
    #batch size
    batch_size = true_boxes.shape[0]
    #每个检测头分配为 0 的 grid 矩阵
    for i in range(detected_num):
        #input_size[0] //ratio[i] = 13,获得格子数
        grid = np.zeros(
            [batch_size, input_size[0] //ratio[i], input_size[1] //ratio[i],
             len(anchors[i]), 4 + 1 + class_num], dtype=np.float32)
        grid_shape.append(grid)
        #
        true_boxes2.append(
            np.zeros([batch_size, input_size[0] //ratio[i], input_size[1] //ratio[i],
                      len(anchors[0]), 4 + 1 + class_num], dtype=np.float32))
    #3 个检测头的 grid
    grid = grid_shape
    #遍历每个 GT BOX
    for box_index in range(true_boxes.shape[1]):
        #根据下标取是第几个 GT BOX
        box = true_boxes[:, box_index, :]
        box = np.squeeze(box, axis=0) #降维
        box_class = box[4].astype('int32')
        best_choice = {}
        #将每个 GT BOX 与每个检测头中的 Anchor 进行 IOU 的计算
        for index in range(0, 3, 1):
            #即 input_size[0] //ratio[i] = 13
            ratio_input = grid[index].shape[-3]
            #因为 BOX 是归一化的值 × 13,放大到 13×13 的特征图的尺寸
            box2 = box[0:4] *np.array([
```

```
                ratio_input, ratio_input, ratio_input, ratio_input
            ])
            #内存复制
            box_true = box2.copy()
            box2 = box2.copy()
            #算是第 ij 个格子
            i = np.floor(box2[1]).astype('int')
            j = np.floor(box2[0]).astype('int')
            #如果 ij>13 就继续下一个 BOX,说明这个 BOX 可能标注错误
            if i > ratio_input or j > ratio_input: continue
            #最佳 IOU
            best_iou = 0
            #最佳 Anchor; 接下来,将 true_box 与 Anchor 进行 IOU,并取最佳 Anchor 用来与
            #预测的内容进行 loss
            best_anchor = 0
            #遍历每 1 个 Anchor 以获取最佳 IOU 作为正样本
            for k, anchor in enumerate(anchors[2 - index]):
                #wh center
                box_maxes = box2[2:4] *0.5
                box_mines = -box_maxes
                #
                anchor_maxes = anchor *0.5
                anchor_mines = -anchor_maxes
                #将真实 wh 与 Anchor 之间进行 IOU 的计算,并获取最佳 IOU 是哪个 Anchor
                intersect_mines = np.maximum(box_mines, anchor_mines)
                intersect_maxes = np.minimum(box_maxes, anchor_maxes)
                intersect_wh = np.maximum(intersect_maxes - intersect_mines, 0.)
                intersect_area = intersect_wh[0] *intersect_wh[1]
                box_area = box2[2] *box2[3]
                anchor_area = anchor[0] *anchor[1]
                #IOU 得分
                iou_score = intersect_area / (box_area + anchor_area - intersect_area)
                #如果 IOU 得分>上 1 次的得分,就更改 best_iou,并记录 k
                if iou_score > best_iou:
                    best_iou = iou_score
                    best_anchor = k
            #记录下来与最佳 best_iou 相关的参数
            #[历史 GT BOX, 第几个检测头, 最佳 IOU 得分,GT BOX,格子 j,格子 i,最佳第几个
            #Anchor,历史 GT BOX,历史 GT BOX]
            best_choice[best_iou] = [box_true, index, best_iou, box_index, j, i,
best_anchor, box2, box]
        #按最佳 IOU 的得分进行排序,获得最大 IOU
        best_iou_choice = best_choice[sorted(best_choice.keys(), reverse=True)[0]]
        #从 best_iou_choice 中获得最佳 box,j,i 的信息
        box = best_iou_choice[-1]
        j = best_iou_choice[-5]
        i = best_iou_choice[-4]
        box2 = best_iou_choice[-2]
        index = best_iou_choice[1]
        best_anchor = best_iou_choice[-3]
        box_true = best_iou_choice[0]
        #对最佳 box2 进行偏移值的求解
```

```
            adjusted_box = np.array([
                box2[0] - j,
                box2[1] - i,
                np.log(box2[2] / anchors[2 - index][best_anchor][0]),
                np.log(box2[3] / anchors[2 - index][best_anchor][1]),
            ], dtype=np.float32)
            #检测头[第几个][...,j 格子,i 格子,最佳 anchor,:4] = 偏移值
            grid[index][..., j, i, best_anchor, 0:4] = adjusted_box
            #置信度
            grid[index][..., j, i, best_anchor, 4] = 1
            #分类信息 box_class 分类的下标,假设为 1,则为 2
            grid[index][..., j, i, best_anchor, 5 + box_class] = 1

            #获取真值,方便后面计算损失时做 IOU
            true_boxes2[index][..., j, i, best_anchor, 0:4] = box_true
        return true_boxes2, grid

    def test_yolo_v3_true_encode_box():
        data = np.array(
            [[0.50, 0.47, 0.05, 0.12, 0], [0.50, 0.47, 0.05, 0.12, 1]]
            , dtype=np.float32
        )
        return YOLOv3_v3_true_encode_box(data)
```

代码中 for i in range(detected_num)初始化设置 3 个 grid,即设置 3 个检测头为 0 的矩阵,用来存放正样本相关值。for box_index in range(true_boxes. shape[1])循环每个 true box(可能有多个),for index in range(0, 3, 1)循环每个检测头,for k, anchor in enumerate(anchors[2－index])循环每个检测头分配的第 $k$ 个 Anchor,best_iou＝iou_score 得到最佳的得分,best_anchor＝k 得到最佳 Anchor,best_choice[best_iou]＝[box_true,index,best_iou,box_index,j,i,best_anchor,box2,box]记录下来与最佳 best_iou 相关的参数,因为 best_choice 是一个字典,如果 best_iou 有相同得分,则将会被替换掉,如果没有,则会保留下来。

3 个检测头和 Anchor 循环结束后,best_iou_choice＝best_choice[sorted(best_choice. keys(),reverse＝True)[0]]根据 best_choice. keys()即 IOU 的得分进行降序排列,从而得到当前最佳的 best_iou_choice 信息,如图 5-62 所示。

然后根据 best_iou_choice 中的信息,计算 adjusted_box 偏移值,并同时赋给 grid[index]相关值。此时就实现某个 GT BOX 从 3 个检测头中得到最佳 IOU 分配到某个检测头中,如图 5-63 所示。

代码中的 GT BOX 将分配到第 2 个检测头(26×26),第 $j$＝26、$i$＝24 的第 1 个 Anchor 为正样本,其他都为负样本。

得到正样本的偏移值后,仍然像 YOLOv1 中的 DataProcessingAndEnhancement(object)类的 generate(self,isTraining＝True)喂给训练数据,具体代码实现基本一致,详细可参考随书代码。

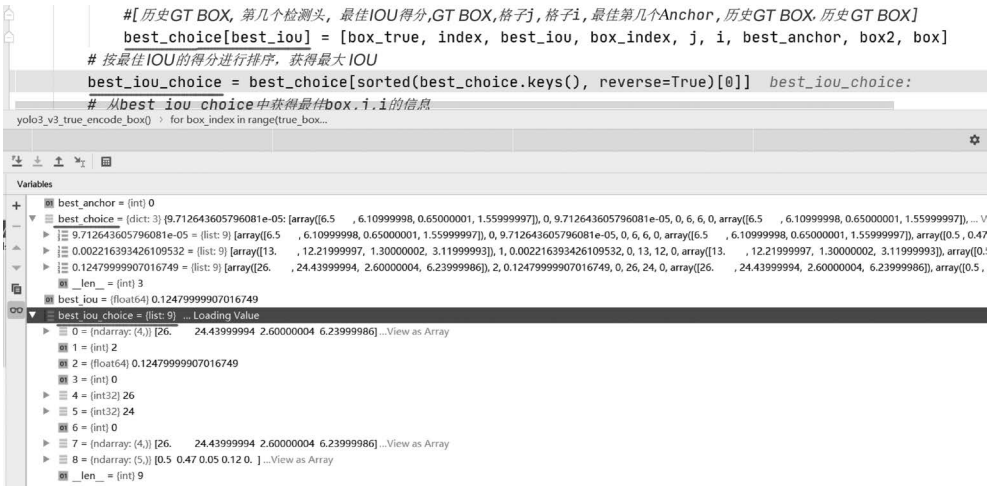



图 5-62　best_iou 的计算示例

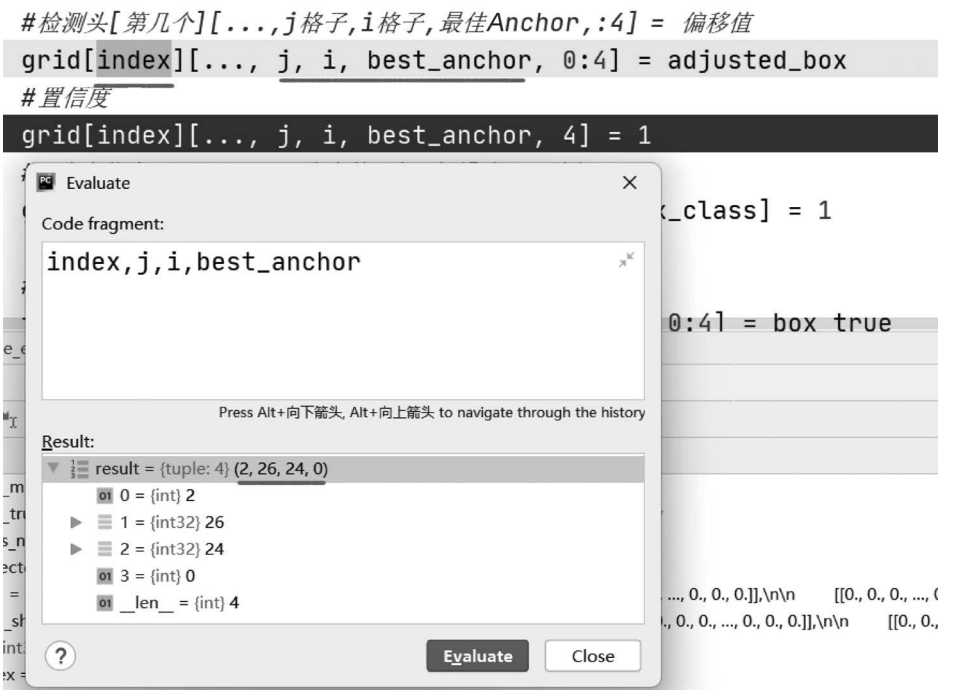



图 5-63　正样本赋值

## 5.7.4　代码实现损失函数的构建及训练

YOLOv3 的损失函数与 YOLOv2 的损失函数基本类似，不同之处在于需要将 3 个检测头的损失相加，另外在计算分类损失时使用二元交叉熵损失，关键代码参考如下：

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V3_Detected/utils/loss.py
class MultiAngleLoss(object):
    def __init__(self, wh_scale=0.5, overlap_threshold=0.5, num_layers=3, num_
class=80, anchors=None):
        #置信度
        self.overlap_threshold = overlap_threshold
        #wh 损失的比例
        self.wh_scale = wh_scale
        self.num_layers = num_layers
        self.num_class = num_class
        if anchors is None:
            self.anchor = np.array([
                [[10, 13], [16, 30], [33, 23]],                    #小目标
                [[30, 61], [62, 45], [59, 119]],                   #中目标
                [[116, 90], [156, 198], [373, 326]]                #大目标
            ])
        else:
            self.anchor = anchors

    def yolo_v3_loss(self, y_true, y_pred, true_box2):
        #true_box2 传入的是原标签值
        #有几个检测头
        num_layers = self.num_layers
        #将数据类型转换成 Tensor
        mf = tf.cast(y_pred[0].shape[0], dtype=y_pred[0].dtype)
        loss = 0 #累加每个检测头中的损失
        for index in range(num_layers):
            y_pre1 = y_pred[index]                                 #第 1 个检测头
            #维度从 b×13×13×255 转换为 b×13×13×3×85,3 是 Anchor,85 是 4+1+80
            y_pre = tf.reshape(
                y_pre1,
                [
                    -1, y_pre1.shape[1], y_pre1.shape[2],
                    len(self.anchor[0]), 4 + 1 + self.num_class
                ])
            #所有的同层 true_box concat 在一起
            true_boxes = true_box2[0][index]
            #传入的是编码过的 3 个检测头的真值 tx,ty,tw,th
            true_grid = y_true[0][index]
            #获取置信度信息,在真值中只有一个检测头是用来预测的
            object_mask = true_grid[..., 4:5]
            #true_class_probs = true_grid[..., 5:] #获取分类信息
            #预测返回的是 xmin,ymin, xmax, ymax 位置信息,置信度信息,分类信息
            pre_boxes, pre_box_confidence, pre_box_class_props = yolo_v3_head_
decode(y_pre, 2 - index,num_class=self.num_class)
            #因为 true_box 中的位置信息是 cx,cy,w,h,所以要转换成 xmin,ymin,xmax,
            #ymax 做 IOU 计算
            #计算 true 的 xmin,ymin,xmax,ymax 以方便做 IOU 的计算
            true_box = tf.concat([
                (true_boxes[..., 0:2] - true_boxes[..., 2:4]) *0.5,
                (true_boxes[..., 0:2] + true_boxes[..., 2:4]) *0.5,
```

```
            ], axis=-1)
            #2 - (w × h) 如果 wh 较少,则惩罚学习小框。因为值放大了
            #如果小目标较多,则可以在 box_loss_scale 的基础上再乘以 1.5
            box_loss_scale = 2 - true_boxes[..., 2] *true_boxes[..., 3]
            #将预测值与真值之间做 IOU,通过 IOU 进行过滤
            iou_result = iou(pre_boxes, true_box)
            iou_score = tf.reduce_max(iou_result, axis=-4)
            iou_score = tf.expand_dims(iou_score, axis=-1)
            #预测值与真值之间的 IOU,如果大于指定阈值,但是又不是最大 IOU 的内容则会被
            #忽略处理
            #真实的框只有一个,而小于阈值的都作为负样本
            object_detections = iou_score > self.overlap_threshold
            object_detections = tf.cast(object_detections, dtype=iou_score.dtype)
            #没有物体的损失,即背景的损失
            #置信度损失,预测框的 IOU 与真实框中的 IOU 小于 0.6 的都为背景
            no_object_loss = (1 - object_detections) *(
                  1 - object_mask) *tf.expand_dims(tf.keras.losses.binary_
crossentropy(
                object_mask, pre_box_confidence, from_logits=True), axis=-1)
            object_loss = object_mask *tf.expand_dims(tf.keras.losses.binary_
crossentropy(
                object_mask, pre_box_confidence, from_logits=True), axis=-1)
            confidence_loss = object_loss + no_object_loss

            #有文章说使用 binary_crossentropy 有助于抑制 exp 指数溢出,所以这里更改了
            #位置损失
            #https://github.com/qqwweee/keras-yolov3/blob/master/yolov3/model.py
            xy_loss = object_mask *tf.expand_dims(box_loss_scale, axis=-1) *tf.
expand_dims(
                tf.keras.losses.binary_crossentropy(
                   true_grid[..., :2], y_pre[..., :2], from_logits=True
                ), axis=-1)

            wh_loss = object_mask *tf.expand_dims(box_loss_scale, axis=-1) *tf.
square(
                true_grid[..., 2:4] - y_pre[..., 2:4]) *self.wh_scale
            #分类损失,使用 binary_crossentropy 交叉熵
            class_loss = object_mask *tf.expand_dims(tf.keras.losses.binary_
crossentropy(
                true_grid[..., 5:], y_pre[..., 5:], from_logits=True), axis=-1)
            #统计损失
            xy_loss = tf.reduce_sum(xy_loss) / mf
            wh_loss = tf.reduce_sum(wh_loss) / mf
            confidence_loss = tf.reduce_sum(confidence_loss) / mf
            class_loss = tf.reduce_sum(class_loss) / mf
            #将 3 个检测头的损失合并在一起
            loss += xy_loss + wh_loss + confidence_loss + class_loss
        return loss
```

代码中 for index in range(num_layers)循环了 3 个检测头以进行损失的计算。yolo_v3_head_decode()解码函数的实现逻辑与 YOLOv2 是一致的,求 GT BOX 与 Anchor BOX 的偏移,然后将 iou_result＝iou(pre_boxes, true_box)预测 BOX 与真实 BOX 做 IOU,根据

object_detections=iou_score>self.overlap_threshold 得到预测有目标，然后由 1-object_detections 得到预测，没有目标，1-object_mask 是真值，有目标，所以将 binary_crossentropy(object_mask,pre_box_confidence)做交叉熵后得到没有目标的置信度损失。将有目标和没有目标相加 confidence_loss=object_loss+no_object_loss，得到总的置信度损失。类别损失这里使用交叉熵损失，class_loss=object_mask * binary_crossentropy(true_grid[...,5:],y_pre[...,5:])，最后将 xy_loss+wh_loss+confidence_loss+class_loss 相加得到总损失。求损失的整体过程跟 YOLOv2 类似，故不再赘述。训练脚本也无重要更新，更多更详细的代码可参考随书代码。

### 5.7.5 代码实战预测推理

YOLOv3 的推理过程与 YOLOv2 基本相同，不同之处在于 YOLOv3 需要遍历 3 个检测层的结果，并根据不同的检测层生成不同的锚框，详细的代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V3_Detected/detected/detected.py
class Detected():
    def __init__(self, model_path, input_size):
        self.model = load_model(
            model_path,
            custom_objects={'compute_loss': MultiAngleLoss(2).compute_loss}
        ) #读取模型和权重
        self.confidence_threshold = 0.5                    #有无目标置信度
        self.class_prob = [0.5, 0.5]                       #两个类别的分类阈值
        self.nms_threshold = 0.5                           #NMS 的阈值
        self.input_size = input_size
                #锚框初始值
        self.anchors = np.array([
            [[10, 13], [16, 30], [33, 23]],
            [[30, 61], [62, 45], [59, 119]],
            [[116, 90], [156, 198], [373, 326]]
        ]) / 416

    def readImg(self, img_path=None):
        img = cv2.imread(img_path)                         #读取要预测的图片
        #将图片转换到 416×416
        self.img, _ = u.letterbox_image(img, self.input_size, [])
        self.old_img = self.img.copy()

    def forward(self):
        #升成 4 维
        img_tensor = tf.expand_dims(self.img / 255.0, axis=0)
        #前向传播
        self.output = self.model.predict(img_tensor)

    def _generate_anchor(self):
        #根据当前预测 layer 得到特征图的 height、width
        self.h, self.w = self.output[self.currentLayer].shape[1:3] #1×13×13×21
        #使用均分的方法生成格子。一共有 h×w 个格子，即 169 个格子
```

```
        self.lin_x = np.ascontiguousarray(np.linspace(0, self.w - 1, self.w).
repeat(self.h)).reshape(self.h *self.w)
        self.lin_y = np.ascontiguousarray(np.linspace(0, self.h - 1, self.h).
repeat(self.w)).T.reshape(self.h *self.w)
        #得到锚框的 w 和 h,因为一共有 3 个 Anchor,所以 reshape 成[1,3,1]
        self.anchor_w = np.ascontiguousarray(self.anchors[2 - self.currentLayer]
[..., 0] *(416 //self.w)).reshape([-1, 3, 1])
        self.anchor_h = np.ascontiguousarray(self.anchors[2 - self.currentLayer]
[..., 1] *(416 //self.h)).reshape([-1, 3, 1])

    def _decode(self):
        #根据当前预测 layer 对预测出来的结果解码
        #YOLOv3 的输出(4+1+num_class) ×3 个 Anchor
        self.b = self.output[self.currentLayer].shape[0]
        #变成[b,3,(4+1+2),169)
        logits = np.reshape(self.output[self.currentLayer], [self.b, 3, -1,
self.h *self.w])
        self.result = np.zeros(logits.shape)
        #套用公式进行解码,Sigmoid()函数进行值域限制
        self.result[:, :, 0, :] = (tf.sigmoid(logits[:, :, 0, :]).NumPy() + self.
lin_x) / self.w
        self.result[:, :, 1, :] = (tf.sigmoid(logits[:, :, 1, :]).NumPy() + self.
lin_y) / self.h
        self.result[:, :, 2, :] = (tf.exp(logits[:, :, 2, :]).NumPy() *self.anchor_
w) / self.w
        self.result[:, :, 3, :] = (tf.exp(logits[:, :, 3, :]).NumPy() *self.anchor_
h) / self.h
        self.result[:, :, 4, :] = tf.sigmoid(logits[:, :, 4, :]).NumPy()
        self.result[:, :, 5:, :] = tf.nn.softmax(logits[:, :, 5:, :]).NumPy()

    def classification_filtering(self):
        #根据 3 个预测 layer 对预测结果进行解析
        all_boxes = []
        all_cls_scores = []
        all_idx = []
        for i in range(3):
            #当前预测 layer
            self.currentLayer = i
            #根据当前预测 layer 进行 Anchor 的生成
            self._generate_anchor()
            #根据当前预测 layer 进行解码操作,解码后此时的内容存储在 self.result 属性中
            self._decode()
            #取最大的分类得分的下标,此时 result 为[b,3,(4+1+num_class),169]
            class_score = self.result[:, :, 5:, :]        #[1,3,num_class,169]
            class_label = np.argmax(class_score, axis=2) #[1,3,169]
            #最大的分类得分值
            score = np.max(class_score, axis=2)           #[1,3,169]
            #当前格子的置信度
            conf = self.result[:, :, 4, :]                #[1,3,169]
            #每个格子最后的得分为置信度*分类得分
            prob = score *conf                            #[1,3,169]
            #根据阈值,得到评分
```

```
            score_mask = prob > self.confidence_threshold   #[1,3,169]
            num = np.sum(score_mask.reshape([-1]))
            if num > 0:
                #只有转置成 [b * 3*(4+1+num_class)*169],才能根据 score_mask 进行取
                #值,最后需要得到[b,3,169,7]中的 7
                self.result = np.reshape(self.result, [self.b, 3, self.h *self.w, -1])
                boxes = self.result[score_mask][..., :4]
                                                #根据 score_mask 过滤得到 boxes
                cls_scores = prob[score_mask]  #根据 score_mask 过滤得到 cls_scores
                idx = class_label[score_mask]  #根据 score_mask 过滤得到 labels index
                #将当前 layer 满足 box、cls、idx 进行存储
                all_boxes.append(boxes)
                all_cls_scores.append(cls_scores)
                all_idx.append(idx)

        if len(all_boxes):
          #遍历 3 个 layer 中的预测内容并进行合并
          all_boxes = np.concatenate(all_boxes, axis=0)
          all_cls_scores = np.concatenate(all_cls_scores, axis=0)
          all_idx = np.concatenate(all_idx, axis=0)
          #根据类别进行非极大值抑制
          for label in range(len(self.class_prob)):
             #如果 class_labels_all==label,则取当前 label 中的信息
             mask = all_idx == label
             #如果都不是当前 label,则跳过
             if np.sum(mask) == 0: continue
             #由 cx,cy,w,h 转换成 xmin,ymin,xmax,ymax
             xyxy_box = u.cxcy2xyxy(all_boxes[mask][..., :4])
             cat_boxes = np.concatenate([xyxy_box, all_cls_scores[mask].reshape
([-1, 1])], axis=-1)
             #NMS
             index = u.nms(cat_boxes, self.nms_threshold)
             #绘框
             self.old_img = u.draw_box(self.old_img, cat_boxes[index])
           #最后结果
           u.show(self.old_img)


if __name__ == "__main__":
   d = Detected(r'../weights', input_size=[416, 416])
   d.readImg('../../val_data/pexels-photo-5211438.jpeg')
   d.forward()
   d.classification_filtering()
```

主要变化在__init__()中 self. anchors 变为 9 个锚框,每个预测层分配 3 个锚框。在_generate_anchor(self)中根据 self. currentLayer 的结果取当前预测层的结果,如图 5-64 所示。

在解码_decode(self)方法中,根据 self. output[self. currentLayer]获取当前层的输出,并且解码公式仍然与 YOLOv2 的解码公式相同。

在 classification_filtering(self)方法中,for i in range(3)循环 3 个预测层,然后将 i 值赋给 self. currentLayer 属性并传递给 self. _generate_anchor()方法生成锚框、self. _decode()

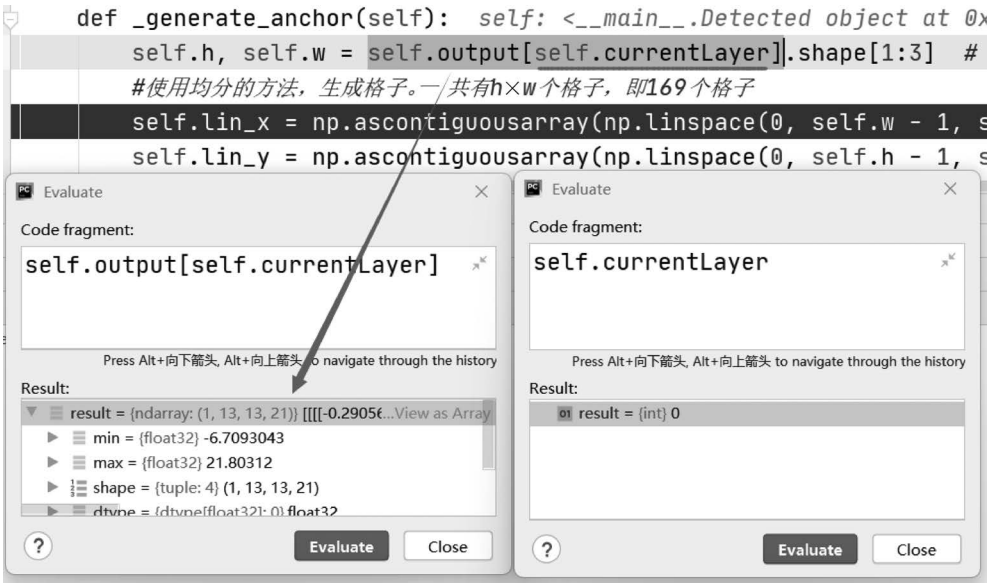



图 5-64　YOLOv3 根据预测层生成锚框

方法进行当前预测 Layer 的解码，将满足 score_mask＝prob＞self. confidence_threshold 阈值的结果存储到 all_boxes、all_cls_scores、all_idx 变量中，如图 5-65 所示。


图 5-65　YOLOv3 阈值过滤

然后根据 all_boxes 的结果，根据 mask＝all_idx＝＝label 进行 mask 的过滤，最后通过 NMS 算法就可以得到预测结果，如图 5-66 所示。

## 总结

YOLOv3 有 3 个检测并且每个检测头分配 3 个 Anchor，分别检测大、中、小物体。同时引入 FPN 以加强几何信息与语义信息的融合，这样更有利于检测小目标。

## 练习

运行并调试本节代码，理解算法的设计与代码的结合，重点梳理本算法的实现方法。

```
#遍历3个layer中的预测内容并进行合并
all_boxes = np.concatenate(all_boxes, axis=0)
all_cls_scores = np.concatenate(all_cls_scores, axis=0)
all_idx = np.concatenate(all_idx, axis=0)
#根据类别进行非极大值抑制
for label in range(len(self.class_prob)):  label: 0
    # 如果class_labels_all==label，则取当前label中的信息
    mask = all_idx == label  mask: [ True  True  True]
    #  {ndarray: (3,)} [0 0 0]
    # 由cx,cy,w,h转换成xmin,ymin,xmax,ymax
    xyxy_box = u.cxcy2xyxy(all_boxes[mask][..., :4])  xyxy_box: [[ 0.0479721   0.03377668
    cat_boxes = np.concatenate([xyxy_box, all_cls_scores[mask].reshape([-1, 1])], axis=-1)
    # nms
    index = u.nms(cat_boxes, self.nms_threshold)  index: [2, 1, 0]
    # 绘框
    self.old_img = u.draw_box(self.old_img, cat_boxes[index])
```

图 5-66　YOLOv3 根据类别索引过滤

## 5.8　单阶段速度快多检测头网络 YOLOv4

### 5.8.1　模型介绍

YOLOv4 由 Bochkovskiy 等在 2020 年论文 *YOLOv4: Optimal Speed and Accuracy of Object Detection* 中提出，其主要特点在 YOLOv3 的基础上将主干特征提取网络换成 CSPDarkNet-53，增加了 SPP 和 PANet。引入数据增强 Mosic 和 DropBlock 抑制过拟合。在损失函数方面将位置 BOX 损失更改为 CIOU 损失，其结构如图 5-67 所示。

首先看 CBM 结构，引入了 Mish 激活函数，其公式如下：

$$f(x) \equiv x \times \tanh(\ln(1+e^{x})) \tag{5-13}$$

Mish 函数的优点是无上界、有下界。无上界是任何激活函数都需要的特征，因为它避免了导致训练速度急剧下降的梯度饱和。Mish 函数具有非单调性，这种性质有助于保持小的负值，从而稳定网络梯度流。Mish 函数是光滑函数，具有较好的泛化能力和结果，可以提高训练的准确率，但是 Mish 函数的计算量相比 ReLU 要大，需要的资源更多，其值域如图 5-68 所示。

主干提取网络由 CSP 构成，原作者认为在推理过程中计算量过高的问题是由于网络优化中的梯度信息重复导致的，CSPNet 通过将梯度的变化从头到尾地集成在特征图中，在减少计算量的同时可以保证准确率。进入 CSP 之后，一条分支进行残差并重复指定次数，另外一条分支经过卷积后与其他分支 concat，使用残差可以使网络更稀疏的同时保留更好的特征信息，另一个分支可以保留不同尺度的特征，使 concat 之后能够增强主干特征的提取能力。ShuffLeNet 的思想，分支结构只有两个时也可降低内存消耗。图 5-67 中 CSP 模块残差处重复的次数分别是 1、2、8、8、4 次。

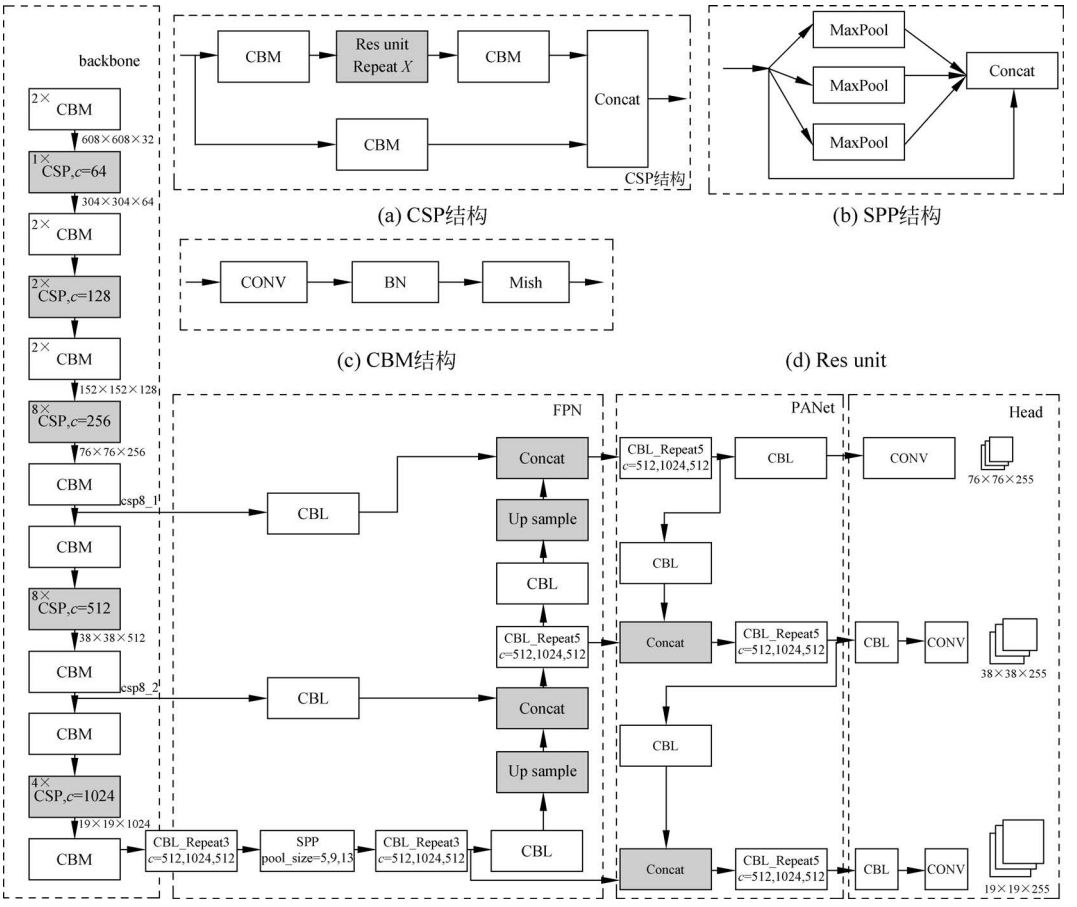



图 5-67　YOLOv4 网络结构

SPP 模块将输入直接作为输出，这为第 1 个分支，第 2 个分支为 5×5 的最大池化，第 3 个分支为 9×9 的最大池化，第 4 个分支为 13×13 的最大池化，其步距都为 1，进行 padding 填充，然后将 4 个分支的特征信息 concat，实现多尺度特征信息的融合，更有助于提升检测的性能。

经过 SPP 模块之后，进行 2 倍上采样为 38×38 并与 csp8_2 的输入进行 concat，再进行 1 次上采样与 csp8_1 的输入进行 concat 得到 76×76 的特征信息。沿 76×76 的特征进行 2 次下采样得到 19×19 的特征，从而用来预测较大目标。FPN 是自底向上的，将深层的语义特征传递过来，对整个特征金字塔进行了增强，不过更多地是增强了语义信息，对定位更有益的几何信息没有传递，PAN 针对这一点，在 FPN 的后面进行了补充，将浅层的定位特征传递到深层，更有助于语义和几何特征信息的融合，如图 5-69 所示。

在计算位置损失时 YOLOv1～3 都使用了均方差损失，这是因为 IOU（GT BOX 与预测 BOX）损失当 IOU＝0 时并不能反映两个框的距离，此时损失函数不可导，无法优化两个框不相交的情况；另一种情况虽然 IOU 的值相同，但是从 IOU 的角度也无法区分两者相交情况的不同，如图 5-70 所示。

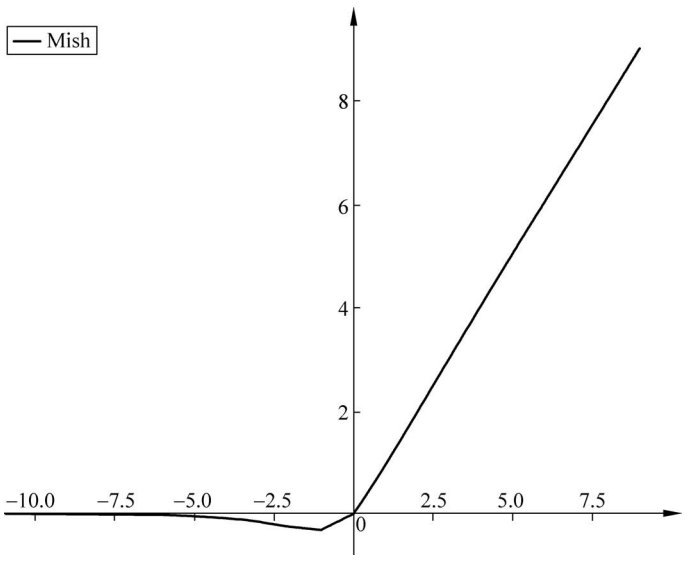



图 5-68 Mish 函数值域

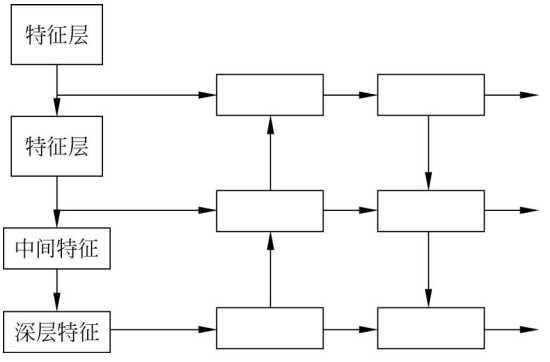



图 5-69 FPN+PAN 结构

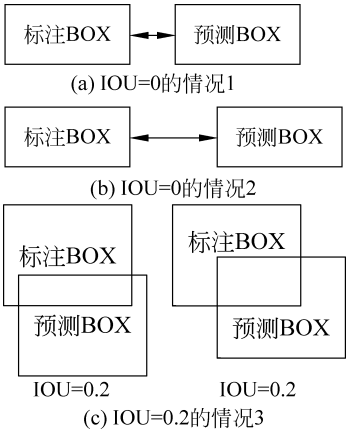


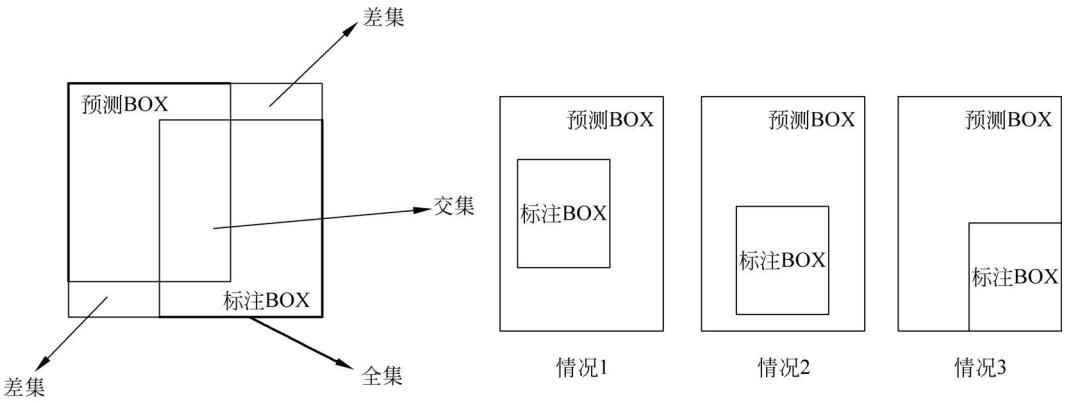

图 5-70 IOU 损失

为了缓解 IOU 损失的不足，在 GIOU 损失中增加相交尺度的衡量方式，其公式为

$$\mathrm{GIOULoss}=1-\mathrm{GIOU}=1-\left(\mathrm{IOU}-\frac{|D|}{|C|}\right) \tag{5-14}$$

其中，$C$ 代表全集，$D$ 代表差集。当差集 $D=0$ 时，此时为 IOU 损失。当 IOU=0 时，$1-\frac{|D|}{|C|}$仍然能够进行求导计算损失。不过，如果某个 BOX 在另外一个 BOX 的中间时，则差集为 0，此时也就退化成 IOU 损失，仍然无法区分相对位置关系，如图 5-71 所示。

DIOU 损失考虑了重叠面积和中心点距离，其公式：

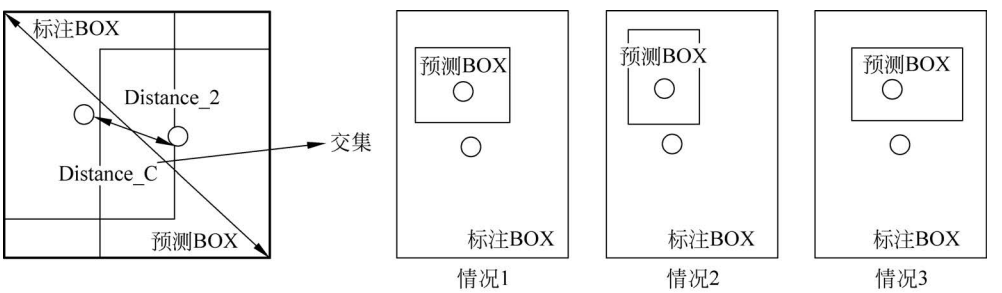



(a) GIOU　　(b) GIOU仍无法度量的情况

图 5-71　GIOU 损失

$$\mathrm{DIOULoss}=1-\mathrm{DIOU}=1-\left(\mathrm{IOU}-\frac{\mathrm{Distance}_{2^2}}{\mathrm{Distance}_{C^2}}\right) \tag{5-15}$$

其中，$\mathrm{Distance}_{2^2}$ 代表两个框中心点的距离，$\mathrm{Distance}_{C^2}$ 代表最小外接矩形(对角线)的距离。当两个框重叠时，$\mathrm{Distance}_{2^2}=0$，IOU＝1，损失为 0；当离得很远时，IOU＝0，损失为 $1-\left(0-\frac{\mathrm{Distance}_{2^2}}{\mathrm{Distance}_{C^2}}\right)$ 仍然能够进行求导计算损失，如图 5-72 所示。




(a) DIOU　　(b) DIOU仍无法度量的情况

图 5-72　DIOU 损失

DIOU 损失：当目标框包住预测 BOX 时，改为直接度量两个框的距离，但是如图 5-72 中的情况，当预测框的中心点都一样时，没有考虑到宽高比的情况，仍然不能更好地做出选择。

CIOU 损失在 DIOU 损失的基础上考虑了宽高比，其公式如下：

$$\mathrm{CIOULoss}=1-\mathrm{CIOU}=1-\left(\mathrm{IOU}-\frac{\mathrm{Distance}_{2^2}}{\mathrm{Distance}_{C^2}}-\frac{v^2}{(1-\mathrm{IOU})+v}\right) \tag{5-16}$$

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

其中，$v$ 为衡量宽高比一致性的参数。对 $v$ 进行求导可得

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

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

图 5-73　DropBlock 可视化效果

当预测到 $w^{p^2}+h^{p^2}$ 很小时，$\frac{1}{w^{p^2}+h^{p^2}}$ 的值会很大，为了避免梯度爆炸，在反向传播时此项值设置为 1。

YOLOv4 在主干特征提取时使用了 DropBlock 按空间位置进行丢弃，强迫网络去学习特征图的其他地方的特征。DropBlock 作用在卷积层，而 DropOut 则作用在全连接层，在 RGB 通道进行 DropBlock 的效果如图 5-73 所示。

关于正负样本匹配，YOLOv4 仍然有 3 个检测头，每个检测头有 3 个 Anchor，当 Anchor 与标注 GT BOX 之间的 IOU 大于阈值时都设定为正样本，其他样本为负样本。在设置置信度的值时使用了标签平滑，允许标签的类别有一点错误，其公式为

$$\text{smooth}_{\text{labels}}=y_{\text{true}}\times(1.0-\text{error}_{\text{rate}})+0.5\times\text{error}_{\text{rate}} \tag{5-17}$$

其中，$\text{error}_{\text{rate}}$ 为允许出现错误的百分比，$y_{\text{true}}=1$。

在损失函数方面，在训练时将位置损失改为 CIOU_loss，推理时使用 DIOU_loss。

## 5.8.2　代码实战模型搭建

从结构图 5-67 中可知，在每次进行 CSP 结果之前有两个 CBM 模块，所以代码中使用 self.split_conv0、self.split_conv1 存储这两个结构。在每次进入 CSP 结构时会经过卷积核为 3×3 且步长 $s=2$ 的下采样。在 1×CSP，$c=64$ 时，输入 channel 与输出 channel 相等，当为其他 CSP 结构时，输出 channel 是输入 channel 的 1 半，关键代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V4_Detected/utils/conv_utils.py

class CSPResBlock(layers.Layer):
    def __init__(self, in_channel, out_channel, num_block, first=False, **kwargs):
        super(CSPResBlock, self).__init__(**kwargs)
        #CSP 中第 1 个 layer1 是用来做下采样的，s=2
        #ConvBNMish 是 Conv->BN->Mish 函数的封装
        self.downSample = ConvBNMish(in_channel, kernel_size=3, strides=2) #32

        if first:
```

```
                #在 CSPRes1 中输入的是 64,残差之后输出的仍然是 64
            self.split_conv0 = ConvBNMish(out_channel, kernel_size=1, strides=1)
            self.split_conv1 = ConvBNMish(out_channel, kernel_size=1, strides=1)
            #CSP 结构中的残差
                self.bocks_conv = Sequential([
                ResBlockMish(in_channel, out_channel),  #残差结构
                ConvBNMish(out_channel, kernel_size=1, strides=1)
            ])
            self.cat_conv = ConvBNMish(out_channel, kernel_size=1, strides=1)
        else:
            #假设:
            #CSPRes2 128->64
            #CSPRes3 256->128
            #CSPRes4 512->256
            #CSPRes5 1024->512
            #从第 2 个 CSP 开始,输出 channel 是输入 channel 的 1 半
            self.split_conv0 = ConvBNMish(out_channel //2, kernel_size=1, strides=1)
                                                            #128 //2
            self.split_conv1 = ConvBNMish(out_channel //2, kernel_size=1, strides=1)
            #*[ResBlockMish for ...] *的作用是将列表反射成参数。num_block 重复的次数
                self.bocks_conv = Sequential([
                 *[ResBlockMish(out_channel //2, out_channel //2) for _ in range
(num_block)],
                ConvBNMish(out_channel //2, kernel_size=1, strides=1)
            ])
                #合并
            self.cat_conv = ConvBNMish(out_channel, kernel_size=1, strides=1)

    def call(self, inputs, *args, **kwargs):
        #layer_1 进行下采样
        x = self.downSample(inputs)
        #两个分支
        x0 = self.split_conv0(x)
        x1 = self.split_conv1(x)
        #残差
        x0 = self.bocks_conv(x0)
        out = layers.Concatenate()([x1, x0])
        #再进行一个 1x1
        out = self.cat_conv(out)
        return out
```

然后根据配置文件中的参数进行设置,构建成 cspdarknet 模块,代码如下:

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V4_Detected/backbone/YOLOv4.py

def csp_darknet_body(input_x):
    """"""
    x = ConvBNMish(32, kernel_size=3, strides=1)(input_x)
    #CPS 模块的输入 channel 和输出 channel,重复次数
    items = [
        [64, 64, 1],
        [128, 128, 2],
```

```
            [256, 256, 8],                  #76 × 76 × 255
            [512, 512, 8],                  #38 × 38 × 255
            [1024, 1024, 4]                 #19 × 19 × 255
        ]
        result = []
        for i, item in enumerate(items):
                #i = 0 first=True
            first = True if i == 0 else False
                #按结构图传递参数
            x = CSPResBlock(item[0], item[1], item[2], first=first)(x)
        return x
```

根据 items 中描述的输入 channel 和输出 channel 循环调用 CSPResBlock(item[0], item[1], item[2], first=first)函数,实现 cspdarknet 特征提取网络的构建。

然后根据结构图构建 CBL_Repeat3 模块,代码如下:

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V4_Detected/utils/conv_utils.py
class CBL_Repeat3(layers.Layer):
    def __init__(self, list_channels: list):
        super(CBL_Repeat3, self).__init__()
        self.list_channels = list_channels
        self.cbl1 = ConvBnLeakRelu(self.list_channels[0], 1, 1)
        self.cbl2 = ConvBnLeakRelu(self.list_channels[1], 3, 1)
        self.cbl3 = ConvBnLeakRelu(self.list_channels[2], 1, 1)

    def call(self, inputs, *args, **kwargs):
        inputs = self.cbl1(inputs)
        inputs = self.cbl2(inputs)
        inputs = self.cbl3(inputs)
        return inputs
```

根据传入的 3 个 channel 数进行卷积、BN、LeakRelu 的计算。与 CBL_Repeat5 的计算类似,只是增加了 self.cbl4、self.cbl5。

对检测头进行封装,输入[channel,3 * (4+1+class_num)]数进行输出,不同检测头的通道 channel 略有不同,代码如下:

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V4_Detected/utils/conv_utils.py
class Yolo_V4_Head(layers.Layer):

    def __init__(self, list_channels: list):
        super(Yolo_V4_Head, self).__init__()
        #list_channels=[1024, len(anchors_mask[0]) *(4 + 1 + class_num)]
        self.list_channels = list_channels
        #3×3 卷积
        self.cbl1 = ConvBnLeakRelu(self.list_channels[0], 3, 1)
        #1×1 卷积,输出
        self.conv = layers.Conv2D(self.list_channels[1], 1, 1)
    def call(self, inputs, *args, **kwargs):
        inputs = self.cbl1(inputs)
        inputs = self.conv(inputs)
        return inputs
```

对 CspDarknet、CBL_Repeat3、CBL_Repeat5、Yolo_V4_Head 等结构进行封装，构建完成 YOLOv4 的主体结构，代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V4_Detected/backbone/YOLOv4.py

def YOLOv4_body(input_shape, anchors=None, class_num=80, is_class=False):
    """构建 YOLO 的网络模型"""
    if anchors is None:
        anchors_mask = np.array([
            [[10, 13], [16, 30], [33, 23]],                    #小目标
            [[30, 61], [62, 45], [59, 119]],                   #中目标
            [[116, 90], [156, 198], [373, 326]]                #大目标
        ])
    else:
        anchors_mask = anchors
    input_x = layers.Input(shape=input_shape, dtype='float32')
    #backbone
    #76 × 76, 38 × 38, 19 × 19
    result = csp_darknet_body(input_x)
    csp8_1, csp8_2, csp4 = result[0]
    #19 × 19 的分支处理
    csp4_19 = CBL_Repeat3(list_channels=[512, 1024, 512])(csp4) #19 *19 *512
    csp4_19_spp = SPP()(csp4_19)                               #19 × 19 × 2048
    #供 19 × 19 输出头 concatenate
    csp4_19_spp_cbl3 = CBL_Repeat3(list_channels=[512, 1024, 512])(csp4_19_spp)
                                                               #19 *19 *512
    #上采样，38 × 38 × 256
    csp4_19_spp_cbl3_for_middle_upsample = ConvUpsample(256)(csp4_19_spp_cbl3)
    #csp8_2 分支的处理 38 × 38 × 256
    csp8_2_cbl = ConvBnLeakRelu(filters_channels=256, kernel_size=1, strides=1)
(csp8_2)
    middle_concat1 = layers.Concatenate()([csp4_19_spp_cbl3_for_middle_
upsample, csp8_2_cbl])                                         #38 × 38 × 512
    #38 × 38 × 256
    middle_concat_cbl5 = CBL_Repeat5(list_channels=[256, 512, 256, 512, 256])
(middle_concat1)                                               #等待第 2 个 concat
    #上采样 76 × 76 × 128
    middle_concat_cbl5_for_76_upsample = ConvUpsample(128)(middle_concat_cbl5)
    #csp8_1 分支的处理 76 × 76 × 128
    csp8_1_cbl = ConvBnLeakRelu(filters_channels=128, kernel_size=1, strides=1)
(csp8_1)
    #76 × 76 × 256
    last_concat = layers.Concatenate()([csp8_1_cbl, middle_concat_cbl5_for_76_
upsample])
    #供中间层进行 middle_concat2，并且也是 76 × 76 的检测头的来源
    #最后一层的输出 76 × 76
    #76 × 76 × 128
    last_concat_cbl5 = CBL_Repeat5(list_channels=[128, 256, 128, 256, 128])(last_
concat)                                                        #p5
    #cbl5 后面还有一个下采样
    #38 × 38 × 256
    last_concat_cbl5_down_sample = ConvBnLeakRelu(filters_channels=256, kernel_
size=3, strides=2)(last_concat_cbl5)
```

```
    #中间层的输出 38 × 38 × 512
    middle_concat2 = layers.Concatenate()([last_concat_cbl5_down_sample,
middle_concat_cbl5])
    #38 × 38 的检测头来源,并且也给 19 × 19 进行 concatenate 38 × 38 × 256
    middle_head_cbl5 = CBL_Repeat5(list_channels=[256, 512, 256, 512, 256])
(middle_concat2)                    #p4
    #下采样 19 × 19 × 512
    middle_head_cbl5_down_sample = ConvBnLeakRelu(filters_channels=512, kernel_
size=3, strides=2)(middle_head_cbl5)
    #第 1 个层的输出 19 × 19 × 1024
    first_concat = layers.Concatenate()([csp4_19_spp_cbl3, middle_head_cbl5_
down_sample])
    #19 × 19 × 512
    first_cbl5 = CBL_Repeat5(list_channels=[512, 1024, 512, 1024, 512])(first_
concat)                             #p3
    #构建检测头
    #19 × 19 × 255 用于检测大图
    out1 = Yolo_V4_Head([1024, len(anchors_mask[0]) *(4 + 1 + class_num)])(first_
cbl5)
    #38 × 38 × 255 用于检测中图
    out2 = Yolo_V4_Head([512, len(anchors_mask[1]) *(4 + 1 + class_num)])(middle_
head_cbl5)
    #76 × 76 × 255 用于检测小图
    out3 = Yolo_V4_Head([256, len(anchors_mask[2]) *(4 + 1 + class_num)])(last_
concat_cbl5)
    return Model(inputs=input_x, outputs=[out1, out2, out3])
```

csp8_1、csp8_2 和 csp4 为 cspdarknet 中获取的特征输出,经过 SPP、FPN、PAN 之后得到 19×19×255、38×38×255、76×76×255 的 3 个检测头,然后通过 outputs=[out1,out2,out3]进行输出。在 FPN、PAN、Head 部分使用激活函数 LeakRule 是由于 Mish 函数的运算量过大,能够加快模型的收敛。

### 5.8.3 代码实战建议框的生成

在 YOLOv4 中如果真实 GT BOX 与 Anchor 的 IOU>0.5,则设为正样本,这样有可能会导致某个 GT BOX 均能在 3 个检测头中找到正样本,其实现代码如下:

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V4_Detected/utils/tools.py
def yolo_v4_true_encode_box(
        true_boxes,
        anchors=None,
        class_num=80,
        input_size=(416, 416),
        ratio=[32, 16, 8],
        iou_threshold=0.213
):
    """当 YOLOv4 使用真值编码时,如果 IOU>0.26,则为正样本。也就说一个 GT 可能会有多个
正样本,而 YOLOv3 是一个 GT 只有一个正样本"""
    if anchors is None:
        anchors = np.array([
```

```
            [[10, 13], [16, 30], [33, 23]],                    #小目标
            [[30, 61], [62, 45], [59, 119]],                   #中目标
            [[116, 90], [156, 198], [373, 326]]                #大目标
        ])
    #Anchor 的数量
    detected_num = anchors.shape[0]
    #用来存储结果
    grid_shape = []
    true_boxes2 = []
    #用来存储结果,对输入的 boxes 进行了升维
    true_boxes = np.expand_dims(true_boxes, axis=0)
    #因为每个 img 都要转换一下,所以应该是 1。
    batch_size = true_boxes.shape[0]
    #跟 YOLOv3 一样,初始 3 个检测头的矩阵
    for i in range(detected_num):
            #13 × 13
        sw, sh = input_size[0] //ratio[i], input_size[1] //ratio[i]
        grid = np.zeros(
            [batch_size, sw, sh, len(anchors[i]), 4 + 1 + class_num],
            dtype=np.float32)
        grid_shape.append(grid)
        true_boxes2.append(
            np.zeros(
                [batch_size, sw, sh, len(anchors[0]), 4 + 1 + class_num],
                dtype=np.float32))
    grid = grid_shape
    #遍历每个 BOX 进行 IOU 的选择
    for box_index in range(true_boxes.shape[1]):
        #BOX 信息
        box = true_boxes[:, box_index, :]
        box = np.squeeze(box, axis=0)                          #降维
        #类别
        box_class = box[4].astype('int32')
        best_choice = {}
        for index in range(0, 3, 1):
            #anchors = anchors *grid[index].shape[-3]
            #box = (13 × x,13 × y, 13 × w, 13 × h) 换算成相对 grid cell 的值
            sa = grid[index].shape[-3]
            #box[0:4]是归一化后的结果 × 13,将 BOX 信息映射到某个特征层
            box2 = box[0:4] *np.array([sa, sa, sa, sa])
            box_true = box2.copy()
            box2 = box2.copy()
            #在 i,j 个格子
            i = np.floor(box2[1]).astype('int')
            j = np.floor(box2[0]).astype('int')
            if i > grid[index].shape[-3] or j > grid[index].shape[-3]:
                continue
            #遍历每个 Anchor
            for k, anchor in enumerate(anchors[2 - index]):
                #wh
                box_maxes = box2[2:4] *0.5
                box_mines = -box_maxes
                #anchor wh
```

```
            anchor_maxes = anchor * 0.5
            anchor_mines = -anchor_maxes
            #将真实 wh 与 Anchor 之间进行 IOU 的计算,并获取最佳 IOU 是哪个 Anchor
            intersect_mines = np.maximum(box_mines, anchor_mines)
            intersect_maxes = np.minimum(box_maxes, anchor_maxes)
            intersect_wh = np.maximum(intersect_maxes - intersect_mines, 0.)
            intersect_area = intersect_wh[0] * intersect_wh[1]
            #计算 IOU
            box_area = box2[2] * box2[3]
            anchor_area = anchor[0] * anchor[1]
            iou_score = intersect_area / (box_area + anchor_area - intersect_area)
            #只要 IOU 大于 0.26 都认为是正样本
            if iou_score >= iou_threshold:
                #计算偏移值
                adjusted_box = np.array([
                    box2[0] - j, #x 和 y 都是相对于 gird cell 的位置,左上角为[0,0],右
                                 #下角为[1,1]
                    box2[1] - i,
                    np.log(box2[2] / anchors[2 - index][k][0]),
                    np.log(box2[3] / anchors[2 - index][k][1]),
                ], dtype=np.float32)
                #存储偏移值
                grid[index][..., j, i, k, 0:4] = adjusted_box
                #置信度
                grid[index][..., j, i, k, 4] = 1
                #标签平滑处理
                grid[index][..., j, i, k, 5 + box_class] = smooth_labels(1, 0.1)
                #获取真值,方便后面计算损失时做 IOU
                true_boxes2[index][..., j, i, k, 0:4] = box_true
    return true_boxes2, grid
```

与 YOLOv3 的代码类似,for box_index in range(true_boxes.shape[1])循环每个 GT BOX,for index in range(0, 3, 1)循环每个检测头,for k, anchor in enumerate(anchors[2-index])循环每个 Anchor,当 iou_score>=iou_threshold(大于设定的阈值)时,grid[index][...,j,i,k,0:4]=adjusted_box 设置为 j,i 个格子中的第 k 个 Anchor 的偏移值为 adjusted_box。与 YOLOv3 不同之处在于可能有多个正样本,如图 5-74 所示。通常来说提高正样本的数量对于模型的训练有益,正样本越多越容易学到更多的特征。

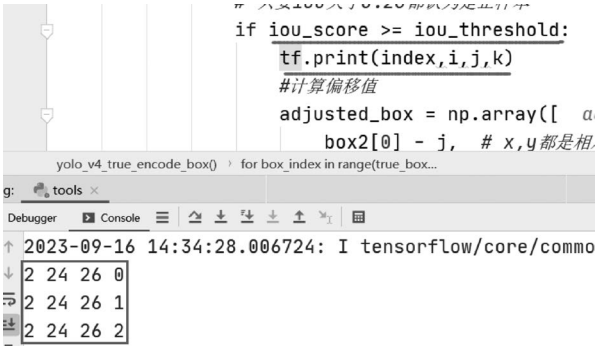



图 5-74 YOLOv4 多个正样本

### 5.8.4　代码实现损失函数的构建及训练

YOLOv4 损失函数的代码实现与 YOLOv3 损失函数的代码实现基本相同，不同之处仅在于计算 BOX 损失时使用 CIOU 损失，CIOU 的计算代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V4_Detected/utils/iou_help.py
def c_iou(priors_box, true_box):
    """
    不仅考虑到重叠面积和中心点距离，还考虑到纵横比。c_iou 一般用于训练
    :param priors_box: cx, cy, w,h
    :param true_box: cx, cy, w,h
    :return:
    """
    #box1 的信息
    b1_xy = priors_box[..., :2]
    b1_wh = priors_box[..., 2:4]
    b1_wh_half = b1_wh / 2.                    #中心点
    b1_mins = b1_xy - b1_wh_half               #xmin, ymin
    b1_maxes = b1_xy + b1_wh_half              #xmax, ymax
    #box2 的信息
    b2_xy = true_box[..., :2]
    b2_wh = true_box[..., 2:4]
    b2_wh_half = b2_wh / 2.
    b2_mins = b2_xy - b2_wh_half               #xmin, ymin
    b2_maxes = b2_xy + b2_wh_half              #xmax, ymax
    #计算 IOU
    intersect_mins = tf.maximum(b1_mins, b2_mins)
    intersect_maxes = tf.minimum(b1_maxes, b2_maxes)
    intersect_wh = tf.maximum(intersect_maxes - intersect_mins, 0.)
    intersect_area = intersect_wh[..., 0] *intersect_wh[..., 1]
    b1_area = b1_wh[..., 0] *b1_wh[..., 1]
    b2_area = b2_wh[..., 0] *b2_wh[..., 1]
    union_area = b1_area + b2_area - intersect_area
    #calculate IoU, add epsilon in denominator to avoid dividing by 0
    iou = intersect_area / (union_area + tf.keras.backend.epsilon())   #iou

    #计算两个 BOX 的中心点距离
    center_distance = tf.keras.backend.sum(tf.square(b1_xy - b2_xy), axis=-1)
    #获得全集
    enclose_mins = tf.minimum(b1_mins, b2_mins)
    enclose_maxes = tf.maximum(b1_maxes, b2_maxes)
    enclose_wh = tf.maximum(enclose_maxes - enclose_mins, 0.0)
    #计算全集中心点的距离
    enclose_diagonal = tf.keras.backend.sum(tf.square(enclose_wh), axis=-1)

    #       d**2
    #求 Diou = iou - ---------
    #       c**2
    diou = iou - 1.0 *(center_distance)/(enclose_diagonal + tf.keras.backend.epsilon())

    #计算 w、h 宽高比
    v = 4 *tf.keras.backend.square(
        tf.math.atan2(b1_wh[..., 0], b1_wh[..., 1]) - tf.math.atan2(b2_wh[..., 0],
b2_wh[..., 1])
```

```
    ) / (tf.cast(np.pi **2, dtype=tf.float32))
    #公式中的 alpha 为了防止梯度爆炸,alpha 不进行梯度计算
    alpha = v / (1.0 - iou + v)
    K.stop_gradient(alpha) #此句不进行梯度求解
    #得到 CIOU
    ciou = diou - alpha *v
    ciou = tf.expand_dims(ciou, -1)
    return ciou
```

在计算 CIOU 时先计算 iou=intersect_area/(union_area+tf. keras. backend. epsilon()),再计算 diou=iou-1.0 * (center_distance)/(enclose_diagonal+tf. keras. backend. epsilon()),然后计算 v,最后得 ciou=diou-alpha * v 的损失结果。计算损失的核心代码如下:

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V4_Detected/utils/loss.py

class MultiAngleLoss(object):
    def yolo_v4_loss(self, y_true, y_pred, true_box2):
        """整体过程跟 YOLOv3 区别不大。主要是在计算 BOX 时,使用的是 CIOU 进行计算"""
        #网络层数
        num_layers = self.num_layers
        loss = 0
        for index in range(num_layers):
                #每层的预测值
            y_pre1 = y_pred[index]
            y_pre = tf.reshape(y_pre1,
                               [-1, y_pre1.shape[1], y_pre1.shape[2], len(self.
anchor[0]), 4 + 1 + self.num_class])

            #y_pre = tf.nn.sigmoid(y_pre) #把预测出来的值限定在 0~1
            #所有的同层 true_box concat 在一起
            true_boxes = tf.concat([t[index] for t in true_box2], axis=0)

            true_grid = tf.concat([t[index] for t in y_true], axis=0)
                #获取置信度信息,在真值中只有一个检测头是用来预测的
            object_mask = true_grid[..., 4:5]
            num_pos = tf.maximum(K.sum(K.cast(object_mask, tf.float32)), 1)
            #预测返回的是位置信息、置信度信息和分类信息,与 YOLOv3 相同
            pre_boxes, pre_box_confidence, pre_box_class_props = yolo_v4_head_
decode(y_pre, 2 - index,

num_class=self.num_class)
            #平衡系数
            #2 - (w × h) 如果 wh 较小,则惩罚学习小框。也可人为设定
            box_loss_scale = 2 - true_boxes[..., 2] *true_boxes[..., 3]

            #使用 CIOU 直接求 BOX 的 loss
            ciou_score = c_iou(pre_boxes, true_boxes[0:4])
            ciou_loss = object_mask *tf.expand_dims(box_loss_scale, axis=-1) *
(1 - ciou_score)

            #预测值与真值之间的 IOU,如果大于指定阈值,但是又不是最大 IOU 的内容,则会被
            #忽略处理
            iou_score = tf.reduce_max(ciou_score, axis=-1)
            iou_score = tf.expand_dims(iou_score, axis=-1)
```

```
            object_detections = iou_score > self.overlap_threshold
            #正样本的 mask
            object_detections = tf.cast(object_detections, dtype=iou_score.dtype)
            #负样本置信度损失
            no_object_loss = (1 - object_detections) *(
                    1 - object_mask) *tf.expand_dims(tf.keras.losses.binary_
crossentropy(
            object_mask, pre_box_confidence, from_logits=True), axis=-1)
            #正样本损失
               object_loss = object_mask *tf.expand_dims(tf.keras.losses.binary_
crossentropy(
            object_mask, pre_box_confidence, from_logits=True), axis=-1)
            #负样本+正样本的损失
            confidence_loss = object_loss + no_object_loss
            #分类损失
            class_loss = object_mask *tf.expand_dims(tf.keras.losses.binary_
crossentropy(
                    true_grid[..., 5:], y_pre[..., 5:], from_logits=True), axis=-1)
            #位置平均损失
            location_loss = tf.abs(tf.reduce_sum(ciou_loss)) / num_pos
            confidence_loss = tf.reduce_mean(confidence_loss)
            class_loss = tf.reduce_sum(class_loss) / num_pos
               #求合
            loss += location_loss + confidence_loss + class_loss
        return loss
```

跟 YOLOv3 相比区别仅在 ciou_score=c_iou(pre_boxes,true_boxes[0:4])的计算上,即位置损失使用 ciou_score,同时根据 object_detections=iou_score>self. overlap_threshold 的得分计算正样本的 mask,然后由 confidence_loss=object_loss+no_object_loss 计算正样本、负样本的置信度损失,最后再将 location_loss+confidence_loss+class_loss 相加得到总损失。

### 5.8.5　代码实战预测推理

YOLOv4 的预测推理流程与 YOLOv3 的预测推理流程一致,即需要先遍历 3 个检测头,接着对每个检测头的输出结果进行置信度、分类概率的过滤,然后对 3 个检测头中满足条件的结果进行合并,再通过 NMS 去除重复的框,详细代码可参考 YOLOv3 中的内容。

**总结**

YOLOv4 在主干特征提取方面使用了 CSP 结构,引用 SPP、FPN 和 PAN 进行更细粒度的特征融合,仍然由 3 个检测头构成多尺度的预测。在损失函数方面,引入 CIOU 来计算位置损失。

**练习**

运行并调试本节代码,理解算法的设计与代码的结合,重点梳理本算法的实现方法。

## 5.9　单阶段速度快多检测头网络 YOLOv5

### 5.9.1　模型介绍

YOLOv5 由 ultralytics 公司开源在 GitHub 上面,到本书成稿时没有发表论文。此模

型得以流行，主要在于 GitHub 上代码工程化的易用性，并且精度和速度较佳，其主要结构如图 5-75 所示。

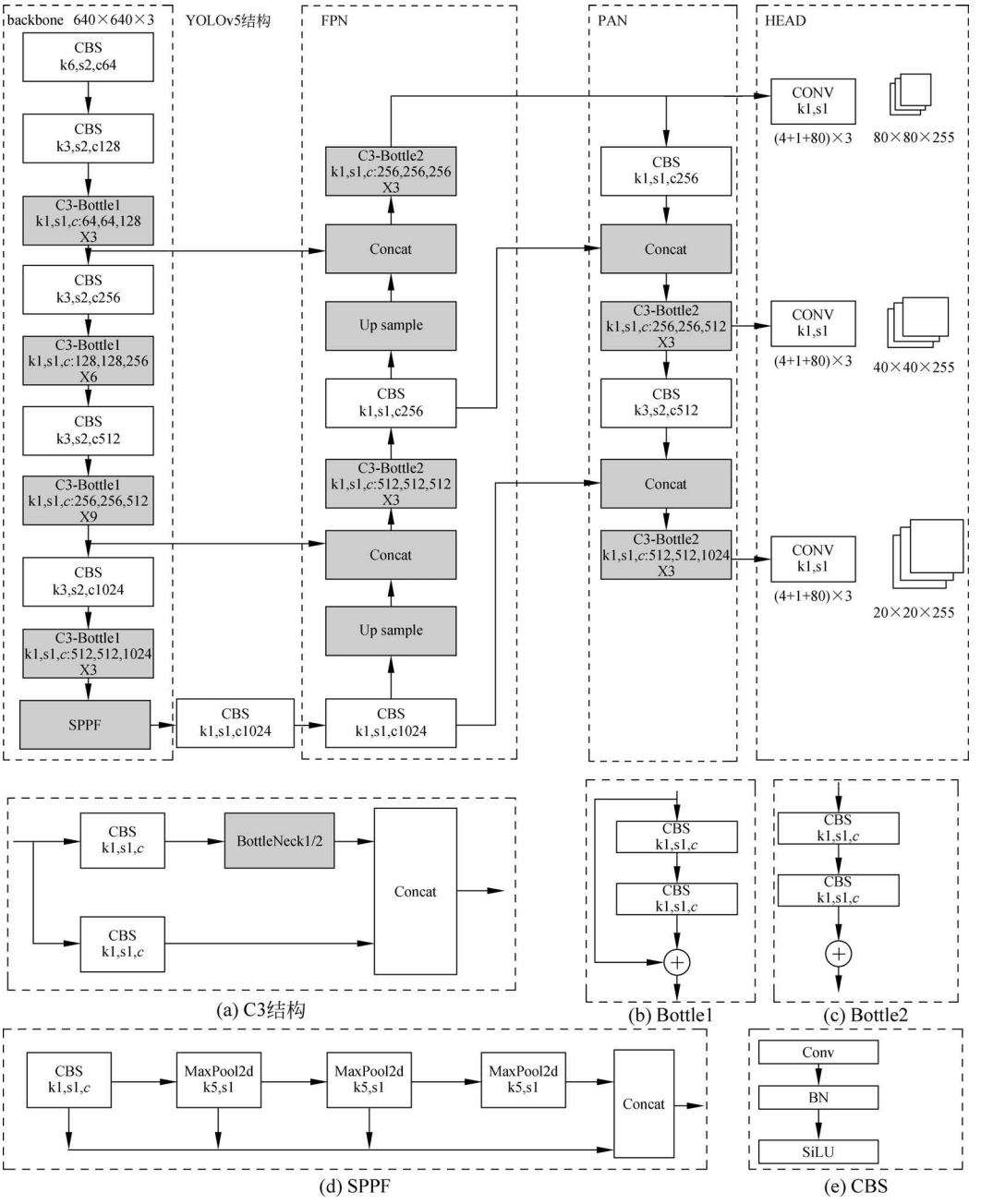



图 5-75　YOLOv5 的结构

图像的输入尺寸设定为 640×640，CBS 为卷积、池化、SiLU 的激活函数，其公式为

$$\mathrm{SiLU}_{(x)} = x * \mathrm{Sigmoid}(x) \tag{5-18}$$

SiLU 函数在接近 0 时具有更平滑的曲线，并且由于 Sigmoid($x$)函数，可以使网络输出在一个较小值的范围，其函数的值域如图 5-76 所示。

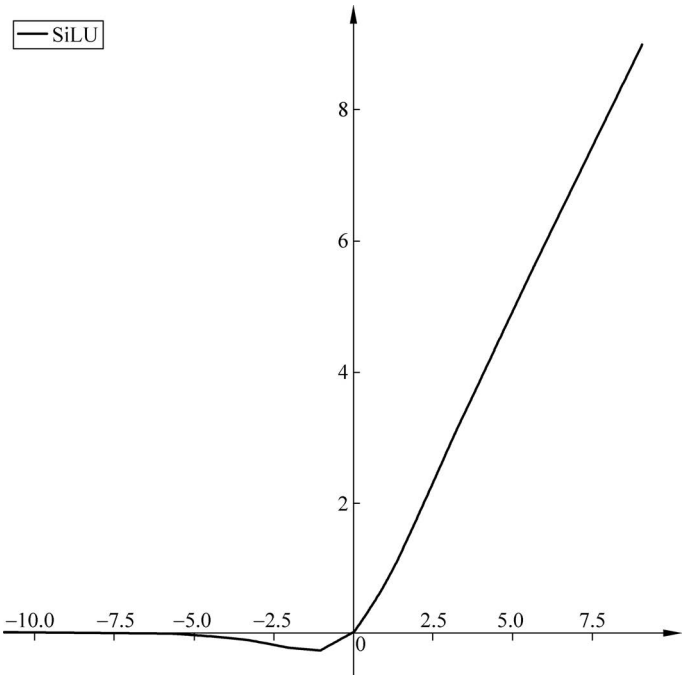



图 5-76　SiLU 激活函数值域

C3 模块分两种情况，在主干特征提取部分为 C3-Bottle1，其结构与 YOLOv4 中的 CSP 结构相同，并且在 YOLOv5 中将 C3-Bottle2 应用于 FPN、PAN 结构之中(YOLOv4 没有)。C3-Bottle2 为正常卷积结构，使用 C3 为一个普通残差结构。将 YOLOv4 中的 SPP 结构更换为 SPPF，由原来的并联更改为了 3 个 5×5 卷积的串联，据作者说能够提高运行速度。

因为输入的分辨率增大了，所以网络的 3 个检测头的输出也有变化，并且不同尺寸的输出分别用于检测小、中、大目标。输出尺寸为 80×80×(4box+1conf+num_class)×3Anchor、40×40×255、20×20×255。

在正负样本匹配方面，YOLOv5 改为将真实 GT BOX 宽和高与 Anchor BOX 宽和高的比例值小于 4 设为正样本，如图 5-77 所示，只要在这个范围内的 Anchor 均被选取，如果超过这个范围，则不选取。

在计算偏移值时，围绕 GT BOX 的中心点 cx 和 cy 所在的格子选择了附近的 3 个格子，并且都计算偏移值，并设定增加新的 ij 作为有目标的格子，由原来 1 个格子，变为 3 个格子，结合宽高比小于 4 的 Anchor，极大地增加了正样本的数量，更有利于网络的学习，如图 5-78 所示。

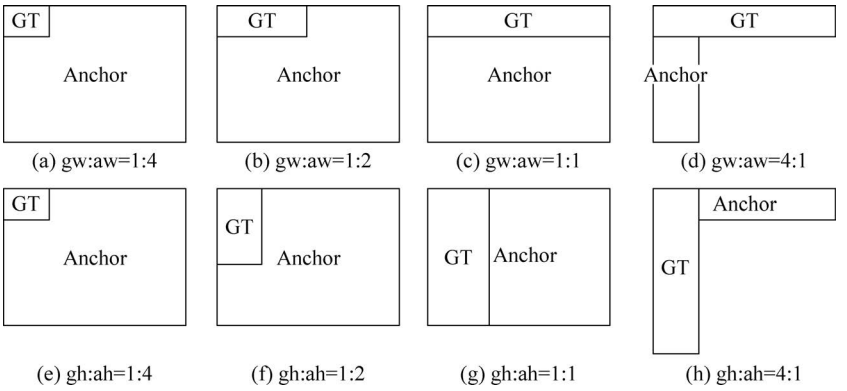



图 5-77 YOLOv5 正样本选取

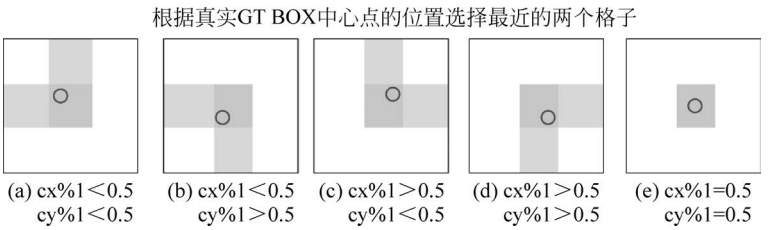



图 5-78 单 GT 多锚点

图 5-78 中 cx 和 cy 为真实 GT BOX 的中心点，因为每个格子的值域为[0,1]，所以由 cx%1 和 cy%1 可得 GT BOX 中心点所在区域，然后根据这个区域选择附近的格子来计算偏移值。

根据预测偏移值和 Anchor，计算预测 $P_x$、$P_y$、$P_w$、$P_h$ 时解码式(5-11)，YOLOv4 的作者指出当真实目标中心点非常靠近网格的左上角时，Sigmoid($t_x$)、Sigmoid($t_y$)会趋近于 0，如果在右下角，则会趋近于 1，网络的预测值需要在负无穷或者正无穷时才能取得，而这种极端值网络一般无法达到，GT BOX 在黄色、红色处时 X 值需要很大才能接近，如图 5-79 所示。

为了解决这个问题，YOLOv4 的作者将偏移值从原来的[0,1]缩放到[−0.5,1.5]，所以其公式变为

$$\begin{aligned} P_x &= (2 * \text{Sigmoid}(t_x) - 0.5) + c_x \\ P_y &= (2 * \text{Sigmoid}(t_y) - 0.5) + c_y \\ P_w &= A_w (2 * \text{Sigmoid}(t_w))^2 \\ P_h &= A_h (2 * \text{Sigmoid}(t_h))^2 \end{aligned} \tag{5-19}$$

虽然 YOLOv4 提出了这个改进策略，但是代码中没有实现，而这一点在 YOLOv5 中被应用。在 YOLOv5 中除了调整 $P_x$、$P_y$ 外，还对 $P_w$、$P_h$ 进行了限制，这样可以避免出现梯度爆炸、训练不稳定的情况。

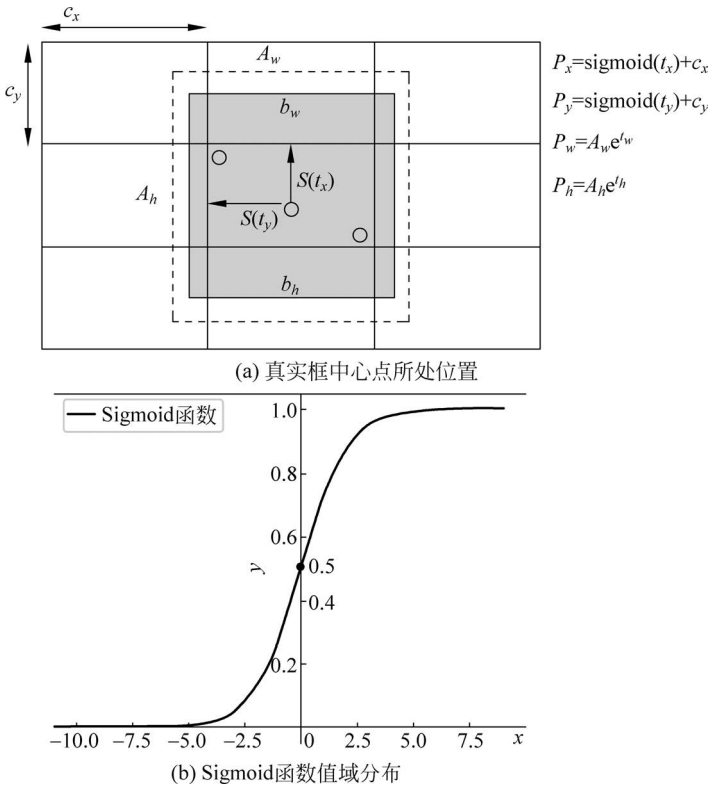



图 5-79　Grid 敏感度(见彩插)

在损失函数方面没有大的变化,位置损失使用 CIOU,而置信度和分类损失则使用交叉熵,不同的检测头有不同的权重比例。

## 5.9.2　代码实战模型搭建

首先根据如图 5-75 所示的结构图实现 CSP_Bottle1,即 CSP 结构,代码如下:

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V5_Detected/utils/conv_utils.py

class Yolo5_BottleNeck1(layers.Layer):
    #残差结构,输入 channel,输出 channel
    def __init__(self, input_channels1, input_channels2):
        super(Yolo5_BottleNeck1, self).__init__()
        self.silu1 = ConvBNSiLU(input_channels1, 1, 1)#conv->bn->silu
        self.silu2 = ConvBNSiLU(input_channels2, 3, 1)

    def call(self, inputs, *args, **kwargs):
        return layers.add([inputs, self.silu2(self.silu1(inputs))])

#CSP bottle1 的结构
class Yolo5_CSP_Bottleneck1(layers.Layer):
```

```
        #CSP bottle1 的结构
        def __init__(self,
                     left_channel, right_channel1,
                     bottleneck_channel: list,
                     bottlenect_repeat: int, out_channel):
            super(Yolo5_CSP_Bottleneck1, self).__init__()
            #左侧 channel
            self.left = ConvBNSiLU(left_channel, 1, 1)
            #右侧 channel
            self.right1 = ConvBNSiLU(right_channel1, 1, 1)
            #残差结构
             self.bottle = Yolo5_BottleNeck1(bottleneck_channel[0], bottleneck_
    channel[1])
            #输出结构
            self.out = ConvBNSiLU(out_channel, 1, 1)
            self.repeat = bottlenect_repeat

        def call(self, inputs, *args, **kwargs):
            left = self.left(inputs)
            right = self.right1(inputs)
            #重复次数
            for _ in range(self.repeat):
                right = self.bottle(right)
            out = layers.concatenate([left, right], axis=-1)
            return self.out(out)
```

代码中 Yolo5_BottleNeck1()实现了残差的封装，如果是第 1 个 C3-Bottle1，则通道值 input_channels1＝64，input_channels2＝64。Yolo5_CSP_Bottleneck1()实现了 C3-Bottle1 整个的封装，left_channel 和 right_channel1 分别为左侧输入、右侧输入通道数，bottleneck_channel 为残差通道数，bottlenect_repeat 为残差重复的次数，out_channel 为输出的通道数，第 1 个 C3-Bottle1 重复的次数为 3，输入通道为 64，输出为 128，则第 1 个 C3-Bottle1 应传入的参数值为 Yolo5_CSP_Bottleneck1(64,64,[64,64],3,128)。

然后根据结构图实现 C3-Bottle2 的结构，C3-Bottle1 主要用在 FPN、PAN 中，代码如下：

```
    #第 5 章/ObjectDetection/TensorFlow_Yolo_V5_Detected/utils/conv_utils.py

    class Yolo5_BottleNeck2(layers.Layer):
        #bottle2,没有残差
        def __init__(self, input_channels1, input_channels2):
            super(Yolo5_BottleNeck2, self).__init__()
            self.silu1 = ConvBNSiLU(input_channels1, 1, 1)
            self.silu2 = ConvBNSiLU(input_channels2, 3, 1)

        def call(self, inputs, *args, **kwargs):
            return self.silu2(self.silu1(inputs))

    class Yolo5_CSP_Bottleneck2(layers.Layer):
        #neck 部分使用 csp_bottle2 结构
        def __init__(self, left_channel, right_channel1,
```

```
                bottleneck_channel: list,
                bottlenect_repeat: int, out_channel):
        super(Yolo5_CSP_Bottleneck2, self).__init__()
        self.left = ConvBNSiLU(left_channel, 1, 1)
        self.right1 = ConvBNSiLU(right_channel1, 1, 1)
              #此时就是一个串行的卷积
         self.bottle = Yolo5_BottleNeck2(bottleneck_channel[0], bottleneck_
channel[1])
        self.out = ConvBNSiLU(out_channel, 1, 1)
        self.repeat = bottlenect_repeat

    def call(self, inputs, *args, **kwargs):
        left = self.left(inputs)
        right = self.right1(inputs)
              #重复多次
        for _ in range(self.repeat):
            right = self.bottle(right)
              #concat
        out = layers.concatenate([left, right], axis=-1)
        return self.out(out)
```

Yolo5_BottleNeck2()是没有残差的，只是一个串行的卷积结构。按指定的重复次数bottlenect_repeat进行解析。Yolo5_CSP_Bottleneck2()的形参的含义相同，根据传入的参数如Yolo5_CSP_Bottleneck2(512,512,[512,512],3,512)进行解析。

完成了相关模块的封装后，对主干特征提取网络进行实现，代码如下：

```
#第5章/ObjectDetection/TensorFlow_Yolo_V5_Detected/backbone/yolo_v5.py

def yolo5_csp_darknet(input_x):
    """DarkNet部分，也是backbone的部分"""
    x = ConvBNSiLU(64, 6, 2)(input_x)
    x = ConvBNSiLU(128, 3, 2)(x)
    x = Yolo5_CSP_Bottleneck1(64, 64, [64, 64], 3, 128)(x)
    p3 = ConvBNSiLU(256, 3, 2)(x)                    #P3

    neck1 = Yolo5_CSP_Bottleneck1(128, 128, [128, 128], 6, 256)(p3)
    p4 = ConvBNSiLU(512, 3, 2)(neck1)                #P4

    neck2 = Yolo5_CSP_Bottleneck1(256, 256, [256, 256], 9, 512)(p4)
    p5 = ConvBNSiLU(1024, 3, 2)(neck2)               #P5
    x = Yolo5_CSP_Bottleneck1(512, 512, [512, 512], 3, 1024)(p5)
    return x, neck2, neck1
```

函数yolo5_csp_darknet()实现了backbone的封装，每个C3-Bottle1结构之后经过ConvBNSiLU()得到结构图中的P3、P4、P5，而neck1、neck2、x会与FPN模块进行结合。对x进行SPPF，以便进行多尺度池化提取，代码如下：

```
#第5章/ObjectDetection/TensorFlow_Yolo_V5_Detected/utils/conv_utils.py

class Yolo5_SPPF(layers.Layer):
```

```
    #SPPF 模块
    def __init__(self, input_channel=512, out_channel=1024):
        super(Yolo5_SPPF, self).__init__()
        self.con_si = ConvBNSiLU(input_channel, 1, 1)
        #都是 5×5 的池化
        self.maxpool = layers.MaxPool2D(5, 1, padding='same')
        self.con_out = ConvBNSiLU(out_channel, 1, 1)

    def call(self, inputs, *args, **kwargs):
        inputs = self.con_si(inputs)  #输入
        pool1 = self.maxpool(inputs)
        pool2 = self.maxpool(pool1)
        pool3 = self.maxpool(pool2)
        #输入与 3 个 5×5 串联的池化 concat
        out1 = layers.concatenate([pool3, pool2, pool1, inputs], axis=-1)
        return self.con_out(out1)
```

Yolo5_SPPF()中对 inputs 进行 pool1，然后对 pool1 最大池化得到 pool2，对 poo2 最大池化得到 pool3，然后将 inputs 与 pool1、pool2、pool3 进行 concat 后由 ConvBNSiLU()输出，从而得到多尺度的特征池化特征信息。

将上面的模块按 YOLOv5 结构图组合得到 YOLOv5 的前向传播，代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V5_Detected/backbone/yolo_v5.py

def yolo_v5(input_shape, anchors=None, class_num=80):
    """YOLOv5 的前向传播"""
    if anchors is None:
        anchors_mask = np.array([
            [[10, 13], [16, 30], [33, 23]],                    #小目标
            [[30, 61], [62, 45], [59, 119]],                   #中目标
            [[116, 90], [156, 198], [373, 326]]                #大目标
        ])
    else:
        anchors_mask = anchors
    input_x = layers.Input(shape=input_shape, dtype='float32')
    #neck1 80 × 80 × 256
    #neck2 40 × 40 × 512
    #neck3 20 × 20 × 1024
    x, neck2, neck1 = yolo5_csp_darknet(input_x)
    #SPPF 实现多尺度池化
    neck3 = Yolo5_SPPF(512, 1024)(x)
    #输入 head
    head3 = ConvBNSiLU(512, 1, 1)(neck3)
    #2 倍上采样，这里用转置卷积代替 upsample
    up3 = layers.Conv2DTranspose(512, 2, 2, padding='same')(head3)
    #将 up3 与 neck2 进行 concat
    up3_neck2 = layers.concatenate([neck2, up3], axis=-1)
    #经过 csp2
    up3_neck2_bot1 = Yolo5_CSP_Bottleneck2(512, 512, [512, 512], 3, 512)(up3_neck2)
    #输入 head2
    head2 = ConvBNSiLU(256, 1, 1)(up3_neck2_bot1)
```

```
    up2_head2 = layers.Conv2DTranspose(256, 2, 2, padding='same')(head2)
    #将 up2 与 neck1 进行 concat
    up2_neck1 = layers.concatenate([neck1, up2_head2], axis=-1)

    #输入 head1
    pred1 = Yolo5_CSP_Bottleneck2(256, 256, [256, 256], 3, 256)(up2_neck1)

    #进行 PAN 下采样
    pred1_down1 = ConvBNSiLU(256, 3, 2)(pred1)
    pred1_down1_cat = layers.concatenate([pred1_down1, head2], axis=-1)
    pred2 = Yolo5_CSP_Bottleneck2(256, 256, [256, 256], 3, 512)(pred1_down1_cat)

    pred2_down2 = ConvBNSiLU(512, 3, 2)(pred2)
    pred2_down2_cat = layers.concatenate([pred2_down2, head3], axis=-1)
    pred3 = Yolo5_CSP_Bottleneck2(512, 512, [512, 512], 3, 1024)(pred2_down2_cat)

    #构建检测头
    out1 = layers.Conv2D((4 + 1 + class_num) *len(anchors_mask[0]), 1, 1)(pred1) #小
    out2 = layers.Conv2D((4 + 1 + class_num) *len(anchors_mask[1]), 1, 1)(pred2) #中
    out3 = layers.Conv2D((4 + 1 + class_num) *len(anchors_mask[2]), 1, 1)(pred3) #大

    return Model(inputs=input_x, outputs=[out3, out2, out1]) #大,中,小
```

代码中 layers. Conv2DTranspose( ) 替换了 layers. UpSample(2),输出内容分别为 out1、out2、out3。

### 5.9.3　代码实战建议框的生成

YOLOv5 中正样本选择分为两步,即先对真实 GT BOX 与 Anchor 进行宽高比的计算,当宽高比小于 4 时选择此 Anchor 作为正样本,同时还实现了单个真实 GT BOX 挑选多个锚点的操作,代码如下:

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V5_Detected/util/tools.py

def yolo_v5_true_encode_box(
        true_boxes,                    #GT BOX 的值
        anchors=None,                  #传入的 Anchor
        class_num=80,                  #num class
        input_size=(640, 640),         #输入尺度
        ratio=[32, 16, 8],             #特征图的尺寸比
        max_wh_threshold=4.0           #阈值
):
    """
    YOLOv5 不再采用 IOU 匹配,YOLOv5 采用基于宽高比例的匹配策略,GT 的宽和高与 Anchors 的宽和高对应相除得到 ratio1;
    Anchors 的宽和高与 GT 的宽和高对应相除得到 ratio2;
    取 ratio1 和 ratio2 的最大值作为最后的宽高比,该宽高比和设定的阈值(默认为 4)比较,小于设定阈值的 Anchor 则为匹配到的 Anchor
    """
    if anchors is None:
```

```
        anchors = np.array([
            [[10, 13], [16, 30], [33, 23]],                    #小目标
            [[30, 61], [62, 45], [59, 119]],                   #中目标
            [[116, 90], [156, 198], [373, 326]]                #大目标
        ]) / 255.
    #Anchor 的数量
    detected_num = anchors.shape[0]
    #存储结果
    grid_shape = []
    true_boxes2 = []
    true_boxes = np.expand_dims(true_boxes, axis=0)
    batch_size = true_boxes.shape[0]
    #初始 3 个检测头的矩阵
    for i in range(detected_num):
            sw,sh = input_size[0] //ratio[i],input_size[1] //ratio[i]
        grid = np.zeros(
            [batch_size, sw, sh, len(anchors[i]), 4 + 1 + class_num],
            dtype=np.float32)
        grid_shape.append(grid)
        true_boxes2.append(
            np.zeros(
                [batch_size, sw, sh, , len(anchors[0]), 4 + 1 + class_num],
                dtype=np.float32))
    grid = grid_shape
    #遍历每个 GT BOX
    for box_index in range(true_boxes.shape[1]):
        #box 信息
        box = true_boxes[:, box_index, :]
        box = np.squeeze(box, axis=0)                          #降维
            #类别
        box_class = box[4].astype('int32')
        best_choice = {}
        for index in range(0, 3, 1):

            #box = (13 × x,13 × y, 13 × w, 13 × h) 换算成特征图上的值
            box2 = box[0:4] *np.array([
                grid[index].shape[-3], grid[index].shape[-3], grid[index].shape
[-3], grid[index].shape[-3]
            ])
            box_true = box2.copy()
            box2 = box2.copy()
            #GT BOX 本来所处的格子
            i = np.floor(box2[1]).astype('int')                #y 轴
            j = np.floor(box2[0]).astype('int')                #x 轴
            if i > grid[index].shape[-3] or j > grid[index].shape[-3]:
                continue
            #循环每个 Anchor
            for k, anchor in enumerate(anchors[2 - index]):
                box_maxes = box2[2:4]
                anchor_maxes = anchor

                #(1)计算每个 GT BOX 与对应的 Anchor Template 模板的高宽比例
```

```
                #(2)统计这些比例和它们倒数之间的最大值
                #这里可以理解成计算 GT BOX 和 Anchor Template 分别在宽度及高度方向的
                #最大差异(当相等时比例为 1,差异最小)
                #假设 r_wh_ratio = 1,则 r_wh_max 就是 max(1, 1),即为 1
                #假设 r_wh_ratio = 0.5, 则 r_wh_max 就是 max(0.5, 1/0.5)--> max(0.5,2)-->2
                #假设 r_wh_ratio = 0.25.则 r_wh_max 就是 max(0.25, 1/0.25)-->
                #max(0.5,4)-->4
                #假设 r_wh_ratio = 0.1, 则 r_wh_max 就是 max(0.1, 1/0.1)-->
                #max(0.1,10) --> 10
                #离得越近,比值越接近 1;离得越远,比值越大(正无穷)

                #每层的 Anchor 相比较
                r_wh_ratio = box_maxes / anchor_maxes #r_w, r_h = w_gt, h_gt / w_ach, h_ach
                r_wh_max = np.maximum(r_wh_ratio, 1 / r_wh_ratio)
                #(3)获取宽度方向与高度方向的最大差异之间的最大值
                r_max = np.max(r_wh_max)
                #如果最大的比例小于 4,则将 GT BOX 分配给该 Anchor Template 模板
                if r_max >= 4:
                    continue
                ################################
                #下一步需要计算 i, j, 2-index(第几组 Anchor),k(第几组的第几个 Anchor)
                #按规则获取偏移范围
                offset = get_near_points(box2[0], box2[1])
                #计算偏移位置,假设 offset 有 3 个偏移位置,则都认为是正样本
                for off_xy in offset:
                   #原来的位置 + 偏移后的位置
                   off_xy_value = np.array([j, i]) + np.array(off_xy)
                            #新的第 j,i 个格子
                   new_j = int(off_xy_value[0])
                   new_i = int(off_xy_value[1])

                   if (new_j >= grid[index].shape[-3] or new_j <= 0) or (new_i >=
grid[index].shape[-3] or new_i <= 0):
                       continue
                   #计算新的偏移
                   adjusted_box = np.array([
                       box2[0] - new_j, #x 和 y 都是相对于 grid cell 的位置,左上角为
                                        #[0,0],右下角为[1,1]
                       box2[1] - new_i,
                       np.log(box2[2] / anchor[0]),
                        np.log(box2[3] / anchor[1])
                    ], dtype=np.float32)
                    #正样本赋值
                    grid[index][..., new_j, new_i, k, 0:4] = adjusted_box
                    grid[index][..., new_j, new_i, k, 4] = 1
                    #标签平滑处理
                    grid[index][..., new_j, new_i, k, 5 + box_class] = smooth_labels
(1, 0.1)
                    true_boxes2[index][..., j, i, k, 0:4] = box_true
    return true_boxes2, grid
```

```
def get_near_points(x, y):
    """根据 cx 和 cy 计算新的锚点"""
    gx = x % 1
    gy = y % 1
    g = 0.5
    #先水平,再垂直
    if gx < g and gy < g:
        return [[0, 0], [-1, 0], [0, -1]]
    elif gx > g and gy > g:
        return [[0, 0], [1, 0], [0, 1]]
    elif gx > g and gy < g:
        return [[0, 0], [1, 0], [0, -1]]
    elif gx < g and gy > g:
        return [[0, 0], [-1, 0], [0, 1]]
    else:
        return [[0, 0]]
```

代码较长但是与 YOLOv3、YOLOv4 类似,重点看不同之处是将 GT BOX 与 Anchor 做宽高比,如图 5-80 所示。

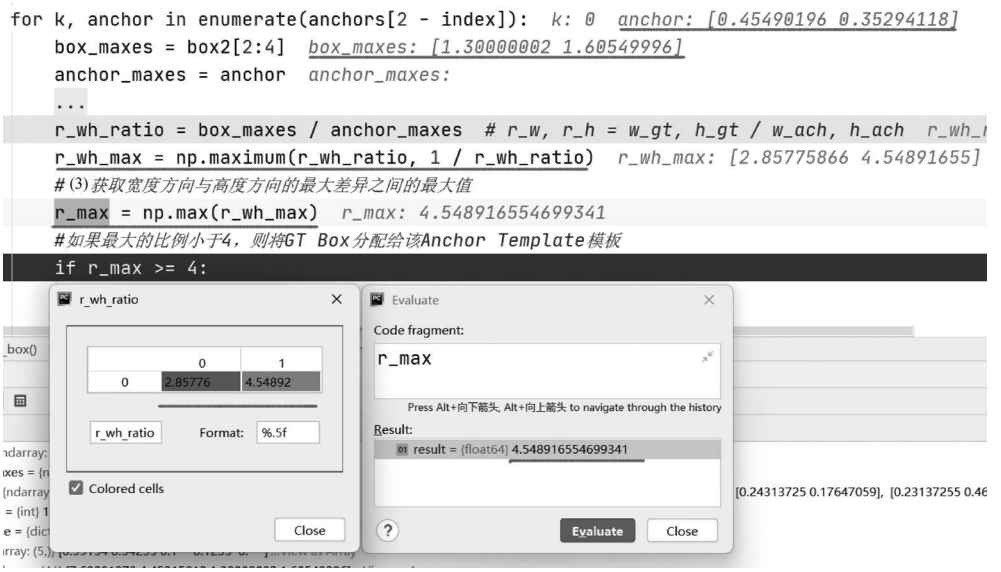



图 5-80 宽高比选择正样本

代码中 box_maxes 的真实 BOX 的宽和高为[1.3,1.6],此时选择的 Anchor 的宽和高为[0.45,0.35],box_maxes/anchor_maxes=[2.85,4.54],然后如果 np.max(r_wh_max)=4.54,宽高比>4 则不会选取,循环下 1 个 Anchor 的值。

多次循环后,当前 GT BOX 所在的格子为 i=4,j=7 且第 3 个检测头中的第 2 个 Anchor 此时 r_max<4,然后根据 cx=7.69 和 cy=4.45 在 get_near_points(box2[0],box2[1])中计算最近的 3 个锚点,如图 5-81 所示。

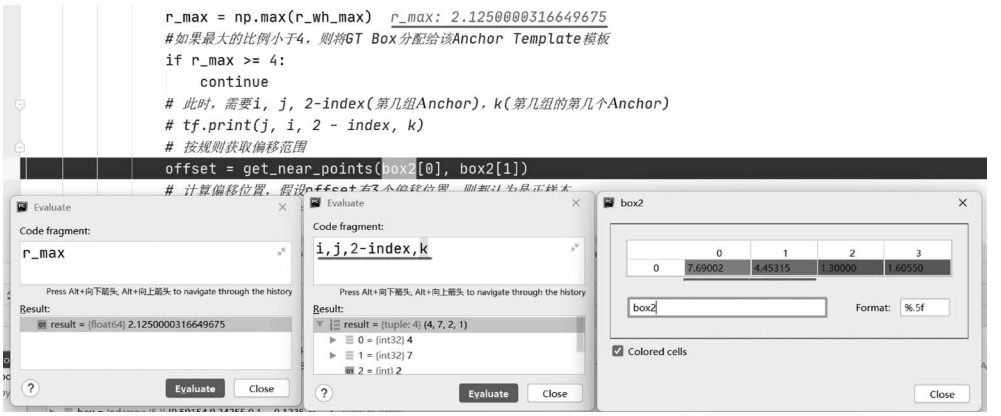



图 5-81　根据 GT BOX 选择锚点

因为 cx＝7.69 和 cy＝4.45，所以 gx＝0.69 和 gy＝0.45，其 GT BOX 的锚点在格子的右上方，所以新锚点格子需要[7,4]＋[[0,0],[1,0],[0,－1]]，其锚点由原来的 1 个变成了 3 个，如图 5-82 所示。

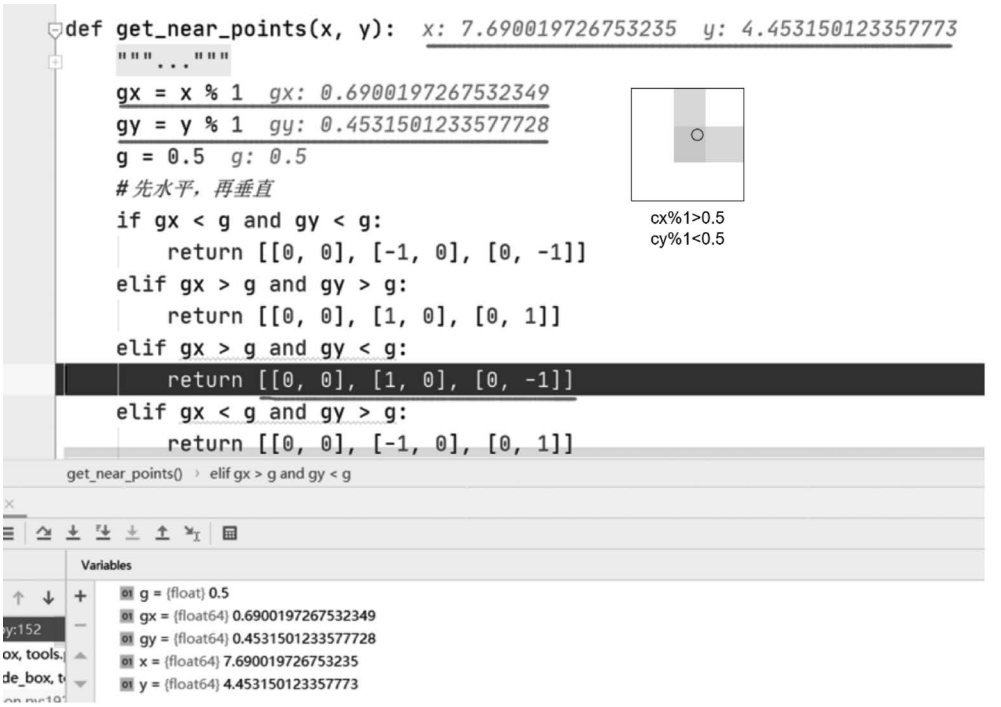



图 5-82　根据 GT BOX 计算 3 个新锚点

然后得到的新的格子为[[7,4],[8,4],[7,3]]，根据新的锚点计算偏移值，box2[0]－new_j、box2[1]－new_i，如图 5-83 所示。

```
            for off_xy in offset:  off_xy: [0, -1]
                #原来的位置 + 偏移后的位置
                # 7, 4 | 8, 4 | 7, 3
                off_xy_value = np.array([j, i]) + np.array(off_xy)  off_xy_value: [7 3]
                new_j = int(off_xy_value[0])  new_j: 7
                new_i = int(off_xy_value[1])  new_i: 3
                # tf.print(new_i, new_j)
                if (new_j >= grid[index].shape[-3] or new_j <= 0) or (new_i >= grid[index].shape
                    continue
                # YOLOv4代码好像是用 BOX 来进行偏移的
                adjusted_box = np.array([  adjusted_box: [0.6900197 1.4531502 1.3000001 1.6055
                    box2[0], box2[1],  # x,y都是相对于grid cell的位置，左上角为[0,0]，右下角为[1,1]
                    box2[2],  #w和h取的是true值中的内容
                    box2[3]
                ], dtype=np.float32)
                #tf.print(f'正样本:index={index},new_j={new_j},new_i={new_i},k={k}')
                grid[index][..., new_j, new_i, k, 0:4] = adjusted_box
                grid[index][..., new_j, new_i, k, 4] = 1
                #标签平滑处理
                grid[index][..., new_j, new_i, k, 5 + box_class] = smooth_labels(1, 0.1)
                true_boxes2[index][..., j, i, k, 0:4] = box_true
```

+ ≣ {ndarray: (4,)} [0.6900197 1.4531502 1.3000001 1.6055 ]

图 5-83　新锚点的正样本赋值

## 5.9.4　代码实现损失函数的构建及训练

YOLOv5 的损失基本与 YOLOv4 的损失相同，主要代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V5_Detected/util/tools.py
class MultiAngleLoss(object):
    def yolo_v5_loss(self, y_true, y_pred, true_box2):
        """YOLOv5 的损失的整体计算过程跟 YOLOv4 区别不大"""
        #3 个检测头
        num_layers = self.num_layers
        loss = 0
        for index in range(num_layers):
            y_pre1 = y_pred[index]
            y_pre = tf.reshape(y_pre1, [
                -1, y_pre1.shape[1], y_pre1.shape[2],
                len(self.anchor[0]), 4 + 1 + self.num_class
            ])
            #真实值合并
            true_grid = tf.concat([t[index] for t in y_true], axis=0)
            #获取置信度信息
            object_mask = true_grid[..., 4:5]
            #对每个检测头进行解码，预测返回的是 xmin,ymin,xmax,ymax 位置信息、置信度
            #信息和分类信息
            pre_boxes, pre_box_confidence, pre_box_class_props = yolo_v5_head_decode(
                y_pre, 2 - index, anchor=self.anchor,
                num_class=self.num_class
            )
            #使用 CIOU 直接求 BOX 的 loss
            ciou_score = c_iou(pre_boxes, true_grid)
```

```
            ciou_loss = object_mask *tf.abs((1 - ciou_score))

            #确定正样本
            tobj = tf.where(
                tf.equal(object_mask, 1),
                tf.maximum(ciou_score, tf.zeros_like(ciou_score)),
                tf.zeros_like(ciou_score)
            )
            #置信度损失
            confidence_loss = K.binary_crossentropy(tobj, pre_box_confidence,
from_logits=True)
            #分类损失
            class_loss = object_mask *K.binary_crossentropy(
                true_grid[..., 5:], y_pre[..., 5:], from_logits=True
            )
            #正样本数量
            num_pos = tf.maximum(K.sum(K.cast(object_mask, tf.float32)), 1)
            #位置损失
            location_loss = tf.abs(K.sum(ciou_loss)) *self.box_ratio / num_pos
            #置信度损失
            confidence_loss = K.mean(confidence_loss) * self.balance[index] *
self.obj_ratio
            #分类损失
            class_loss = K.sum(class_loss) *self.cls_ratio / num_pos / self.num_class
            loss += location_loss + confidence_loss + class_loss
        return loss
```

代码中没有求负样本的置信度损失，而是使用 K. binary_crossentropy(tobj，pre_box_confidence)求正样本的置信度损失，位置损失使用 ciou_loss 进行求解。总损失仍然是位置损失、置信度损失、分类损失之和 location_loss＋confidence_loss＋class_loss。

与 YOLOv4 实现不同的是在解码函数 yolo_v5_head_decode()中对 Grid 敏感度进行了限制，代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V5_Detected/util/tools.py
def yolo_v5_head_decode(features, index, anchor=None, num_class=80, scale_x_y=2):
    """对预测出来的值进行解码，得到 xmin,ymin,w,h"""
    #features 1 × 13 × 13 × 3 × 85
    if anchor is None:
        anchor = np.array([
            [[10, 13], [16, 30], [33, 23]],                    #小目标
            [[30, 61], [62, 45], [59, 119]],                   #中目标
            [[116, 90], [156, 198], [373, 326]]                #大目标
        ])/255.
    anchor_size = len(anchor)
    features = tf.reshape(features, [-1, features.shape[1], features.shape[2],
len(anchor[0]), 4 + 1 + num_class])
    conv_height, conv_width = features.shape[-4], features.shape[-3]
    #2 ×sigmoid(xy) - 0.5,值域就变成了-0.5~1.5
    xy_offset = scale_x_y *tf.nn.sigmoid(features[..., 0:2]) - 0.5 *(scale_x_y - 1)
    #2 × pow(sigmoid(wh),2),限定值域
```

```
        wh_offset = tf.square(scale_x_y *tf.sigmoid(features[..., 2:4]))
        #置信度
        box_confidence = tf.sigmoid(features[..., 4:5])
        #
        box_class_props = tf.nn.sigmoid(features[..., 5:])
        #在 feature 上面生成 anchors
        height_index = tf.range(conv_height, dtype=tf.float32)
        width_index = tf.range(conv_width, dtype=tf.float32)
        #得到网格
        x_cell, y_cell = tf.meshgrid(height_index, width_index)
        x_cell = tf.reshape(x_cell, [x_cell.shape[0], x_cell.shape[1], 1])
        y_cell = tf.reshape(y_cell, [x_cell.shape[0], x_cell.shape[1], 1])
        #根据网格求坐标位置,套公式
        bbox_x = (x_cell + xy_offset[..., 0]) / conv_height
        bbox_y = (y_cell + xy_offset[..., 1]) / conv_width
        bbox_w = (anchor[index][:, 0] *wh_offset[..., 0]) / conv_height
        bbox_h = (anchor[index][:, 1] *wh_offset[..., 1]) / conv_width
        boxes = tf.stack(
            [
                bbox_x,
                bbox_y,
                bbox_w,
                bbox_h
            ],
            axis=3
        )
        boxes = tf.reshape(boxes, [boxes.shape[0], boxes.shape[1], boxes.shape[2],
    anchor_size, 4])
        return boxes, box_confidence, box_class_props
```

在代码 xy_offset 和 wh_offset 中根据公式使值域为[－0.5,1.5],通过 bbox_x、bbox_y、bbox_w 和 bbox_h 解码得到预测值。

### 5.9.5 代码实战预测推理

YOLOv5 的预测推理流程仍然与 YOLOv3 一致,不同之处是在将预测值解码时的公式变为公式 5-18,所以在 YOLOv3 的基础上应对解码函数_decode(self)进行修改,修改后的代码如下:

```
    def _decode(self):
            #根据当前预测 layer 对预测出来的结果解码
            #YOLOv3 的输出(4+1+num_class) ×3 个 Anchor
            self.b = self.output[self.currentLayer].shape[0]
            #变成[b,3,(4+1+2),169)
            logits = np.reshape(self.output[self.currentLayer], [self.b, 3, -1,
    self.h *self.w])
            self.result = np.zeros(logits.shape)
            #套用公式进行解码,使用 Sigmoid()函数对值域进行限制
            self.result[:, :, 0, :] = ((2 *tf.nn.sigmoid(logits[:, :, 0, :]) - 0.5 *
    (2 - 1)).NumPy() + self.lin_x) / self.w
```

```
        self.result[:, :, 1, :] = ((2 *tf.nn.sigmoid(logits[:, :, 1, :]) - 0.5 *
    (2 - 1)).NumPy() + self.lin_y) / self.h
        self.result[:, :, 2, :] = (tf.square(2 *tf.sigmoid(logits[:, :, 2, :]))).
    NumPy() *self.anchor_w) / self.w
        self.result[:, :, 3, :] = (tf.square(2 *tf.sigmoid(logits[:, :, 3, :]))).
    NumPy() *self.anchor_h) / self.h
        self.result[:, :, 4, :] = tf.sigmoid(logits[:, :, 4, :]).NumPy()
        self.result[:, :, 5:, :] = tf.nn.softmax(logits[:, :, 5:, :]).NumPy()
```

**总结**

YOLOv5 没有发表论文(截至本节撰写时),其主要结构与 YOLOv4 类似。在代码中主要使用了 GT 和 Anchor 宽高比小于 4,并且由某个 GT 中心点附近的 3 个格子进行预测,极大地提高了正样本的数量。

**练习**

运行并调试本节代码,理解算法的设计与代码的结合,重点梳理本算法的实现方法。

## 5.10　单阶段速度快多检测头网络 YOLOv7

### 5.10.1　模型介绍

YOLOv7 于 2022 年由 Chien-Yao Wang 等在发表的论文 *YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors* 中提出。相对于 YOLOv5,主要引入了 ELAN、MP、SPPCSPC、REPConv 等模块,正负样本匹配时使用了 Better simOTA 的计算方法,并在训练时增加了 Aux Head 检测,其网络结构如图 5-84 所示。

图像的输入尺寸设定为 640×640,进行两次 CBS,其中 CBS 分别为卷积、BN、SiLU 激活函数。第 1 个 CBS 的步长为 1,改变通道数,第 2 个 CBS 的步长为 2,进行下采样,重复后得到 160×160×128 的特征图。

ELAN 模块是一个高效的网络结构,它通过控制最短和最长的梯度路径,使网络能够学习到更多的特征,并且具有更强的稳健性。ELAN 有两个分支,第 1 个分支经过 1×1 卷积做通道数的变化;第 2 个分支首先经过 1×1 卷积模块做通道数的变化,接着经过 4 个 3×3 的卷积做特征提取,然后将 4 个特征叠加在一起再经过 CBS 模块输出特征。ELAN-W 模块与 ELAN 模块类似,只是融合的特征层多了两个分支。

MP 模块进一步做下采样,由两个分支构成。第 1 个分支先经过最大池化后接 1×1 卷积以改变通道数。第 2 个分支经过 1×1 卷积做通道数的变化后再经过步长为 2 的 3×3 卷积也进行下采样,然后将两个分支进行 Concat 得到不同尺度,以及不同特征值的下采样特征图。MP1 在 backbone 中输入通道数减半,而 head 时 MP2 的通道数不变。

在 YOLOv4 中 SPP 的作用是能够增大感受野,使算法适应不同分辨率的图像,获取不同的感受野。在 SPPCSPC 中首先将特征分为两部分,第 1 个分支经过 5、9、13 的最大池

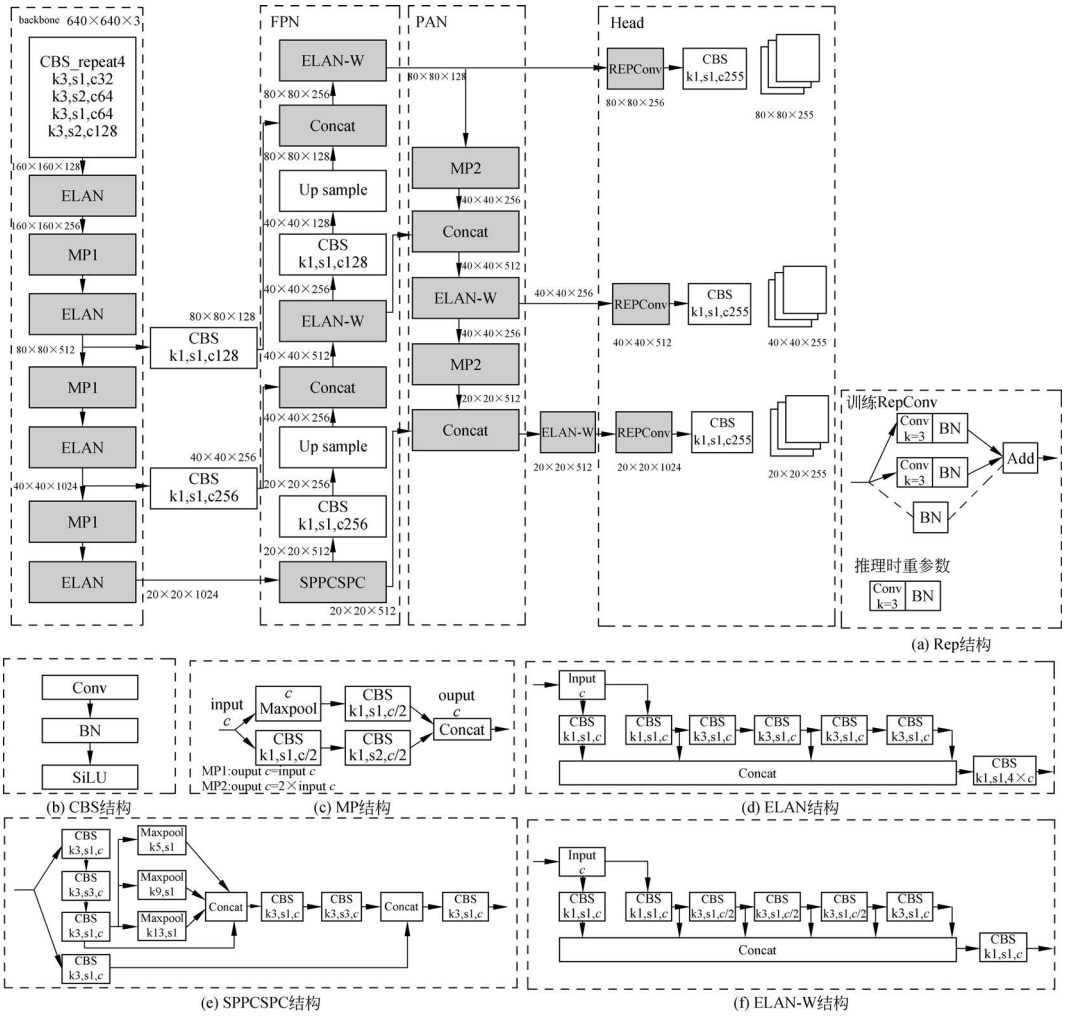



图 5-84 YOLOv7 的结构

化，并与 s=1 的 CBS 模块进行 Concat，使该结构能够处理不同尺度的感受野，利于区分大目标和小目标。另 1 个分支进行常规处理，并与经过最大池化的分支合并，使该结构能够减少计算量，从而使精度得到提升。

模块重参数 RepConv 在训时将一个整体模块分割成多个不同的模块分支，而在推理过程将多个分支模块集成到一个完全等价的模块中，在保证精度的条件下使推理效率更高。

正负样本匹配继承了 YOLOv5 中的宽高比的匹配方法，并根据标注框 GT BOX 的中心位置获取临近的两个网格并作为正样本，即 1 个 GT BOX 由 3 个网格来预测。默认 Anchor 有 9 个，1 个 GT BOX 如果与 Anchor 的宽高比都小于 4，则一共有 3×9=27 个正样本，并且此时正样本将分布在 3 个检测头中。有可能存在一个 GT BOX 对应多个正样本

的情况，而且一个正样本有可能对应多个 GT BOX。这种多 Anchor、多 Grid 网格、多检测头 Head 匹配的方法是 YOLOv7 中的初选，如图 5-85 所示。

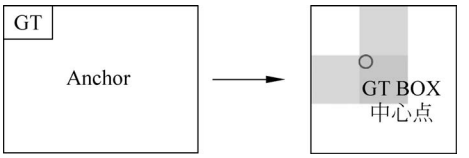


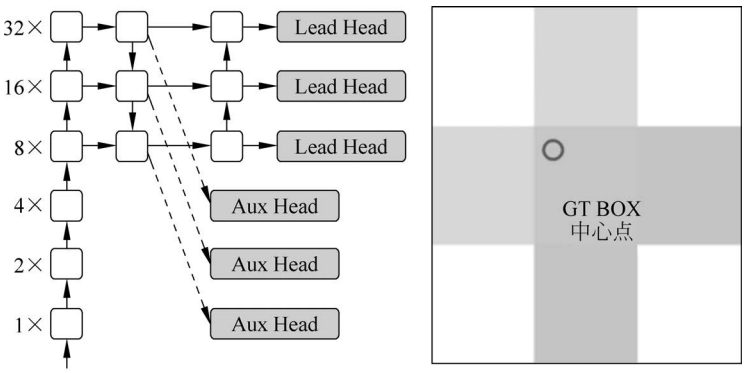

图 5-85　YOLOv7 正样本选择

然后进入复选 OTA 算法，动态地确定每个 GT BOX 真正需要的正样本数量。首先计算 GT BOX 与预测框的 IOU，然后根据 IOU 的得分从大到小进行排序，取 Top 10 个 IOU 的得分进行求和，求和所得值设为当前动态正样本数 $K$（$K$ 最小取 1）。这个 $K$ 值就是 GT BOX 所要选取的正样本的数量。

然后计算每个 GT BOX 中的置信度与预测结果中的置信度×分类的交叉熵损失 pair_wise_cls_loss，再将初选框正样本的回归 Reg IOU Loss 加上 pair_wise_cls_loss，并为每个 GT BOX 取 loss 最小的前 $K$ 个样本作为正样本。

因为一个 GT BOX 可以有多个正样本，一个正样本应该只能对应一个 GT BOX，所以如果一个正样本对应多个 GT BOX，就能找到它跟多个 GT BOX 的损失值，用最小的那个损失所在的正样本进行预测。这个过程可以称为精选，更详细的代码实现过程可参考后文。

YOLOv7 论文中使用一辅助头 Aux Head 检测，需要注意的是正样本在粗选时，辅助头中每个网络与 GT BOX 匹配后选择附近的 4 个网格，而 Lead Head 是两个。在精选时 Aux Head 选择 Top 20 个进行 GT BOX 与初选正样本的求和，而 Lead Head 是 10 个，然后在平衡 Aux Head loss 和 Lead Head loss 时，需要按照 0.25∶1 的比例进行，否则会导致 Lead Head 精度变低，如图 5-86 所示。




(a) 辅助头检测　　(b) 辅助头初选正样本

图 5-86　辅助头检测

在损失函数方面没有大的变化，位置损失使用 CIOU，置信度和分类损失使用交叉熵，不同的检测头有不同的权重比例。

## 5.10.2　代码实战模型搭建

根据模型结构图 5-84 使用 PyTorch 生成结构 ELAN，代码如下：

```
#第 5 章/ObjectDetection/Pytorch_Yolo_V7_Detected/conv_utils.py
def autopad(k, p=None):
    #根据步长自动补 0
    #Pad to 'same'
    if p is None:
        p = k //2 if isinstance(k, int) else [x //2 for x in k] #auto-pad
    return p
class CBS(nn.Module):
    #标准卷积模块
    #输入 cannel,输出 channel,核长,步长,padding 数量,groups 分组卷积数量
    def __init__(self, c1, c2, k=1, s=1, p=None, g=1, act=True):
        super(CBS, self).__init__()
        #autopad(k,p)根据卷积核自动补 0
        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 fuseforward(self, x):
        return self.act(self.conv(x))

class ELAN(nn.Module):
    #ELAN 模块
    def __init__(self, c1, c2):
        #C1 为输入通道,C2 为输出通道
        super(ELAN, self).__init__()
        #CBS 为 Conv->BN->SiLU 的封装
        #c1、c2、1、1 分别为输入、输出、kernelSize、strides
        self.cbs1 = CBS(c1, c2, 1, 1)
        self.cbs2 = CBS(c1, c2, 1, 1)
        #cbs3->cbs6 输入与输出通道相同
        self.cbs3 = CBS(c2, c2, 3, 1)
        self.cbs4 = CBS(c2, c2, 3, 1)
        self.cbs5 = CBS(c2, c2, 3, 1)
        self.cbs6 = CBS(c2, c2, 3, 1)
        #因为会将 out1,out2,out3,out4 合并,所以输入/输出通道是 4*c2
        self.cbs7 = CBS(4 *c2, c2 *4, 1, 1)

    def forward(self, x):
        out1 = self.cbs1(x) #假设 x 为 128×160×160,则经 cbs1 后输出为 64×160×160
        out2 = self.cbs2(x) #cbs2 为另一个分支,输出为 64×160×160
        out3 = self.cbs4(self.cbs3(out2)) #out2 进行 cbs3 和 cbs4
        out4 = self.cbs6(self.cbs5(out3)) #out3 进行 cbs5 和 cbs6
        #合并后使用 cbs7
        return self.cbs7(torch.cat([out1, out2, out3, out4], dim=1))
```

PyTorch 在创建子模块时继承了 nn.Module,然后在__init__()中描述算子的属性,在 forward()中实现前向传播。ELAN 模块在 cbs3、cbs4、cbs5、cbs6 时,输入与输出的通道相同。在 forward()中将 out1、out2、out3、out4 进行 cat 操作,通道数变为 4 倍,所以 cbs7 的

输出为 4 * c2。另外，PyTorch 的输入 shape 为[batch_size，channel，height，width]，所以在 cat 操作时 dim＝1(在 1 轴)，而 TensorFlow 的输入 shape 为[batch_size，height，width，channel]，所以在 cat 操作时 axis＝－1。

ELAN-W 模块与 ELAN 模块类似，将 out1、out2、out3、out4、out5、out6 的输出特征进行了融合，cbs3、cbs4、cbs5、cbs6 是输入通道数的一半，详细的代码如下：

```
#第 5 章/ObjectDetection/Pytorch_Yolo_V7_Detected/conv_utils.py
class ELAN_W(nn.Module):
    #ELAN_W 模块
    def __init__(self, c1, c2):
        super(ELAN_W, self).__init__()
        self.cbs1 = CBS(c1, c2, 1, 1)
        self.cbs2 = CBS(c1, c2, 1, 1)
        #输出是 c2//2 的一半
        self.cbs3 = CBS(c2, c2 //2, 3, 1)
        self.cbs4 = CBS(c2 //2, c2 //2, 3, 1)
        self.cbs5 = CBS(c2 //2, c2 //2, 3, 1)
        self.cbs6 = CBS(c2 //2, c2 //2, 3, 1)
        self.cbs7 = CBS(4 *c2, c2, 1, 1)

    def forward(self, x):
        out1 = self.cbs1(x) #假设 x 为 512 × 40 × 40,则经 cbs1 后输出为 512 × 40 × 40
        out2 = self.cbs2(x)
        out3 = self.cbs3(out2) #256 × 40 × 40
        out4 = self.cbs4(out3)
        out5 = self.cbs5(out4)
        out6 = self.cbs6(out5)
        #将每个 cbs 的输出都进行了合并
        return self.cbs7(torch.cat([out1, out2, out3, out4, out5, out6], dim=1))
```

接下来对 MP 模块进行实现，代码如下：

```
#第 5 章/ObjectDetection/Pytorch_Yolo_V7_Detected/conv_utils.py
class MP(nn.Module):
    #MP 模块
    def __init__(self, c1, c2, k=2, p=None):
        #C1 为输入通道，C2 为输出通道
        super(MP, self).__init__()
        self.k = k
        self.pool1_1 = nn.MaxPool2d(self.k, self.k)
        #输入与输出通道相同
        self.cbs1_2 = CBS(c1, c2, 1, 1)
        self.cbs2_1 = CBS(c1, c2, 1, 1)
        self.cbs2_2 = CBS(c2, c2, 3, 2)

    def forward(self, x):
        #先池化后 cbs
        x1 = self.cbs1_2(self.pool1_1(x)) #128 × 80 × 80
        #另一个分支进行两次 cbs
        x2 = self.cbs2_2(self.cbs2_1(x)) #128 × 80 × 80
        #池化和 cbs 进行融合
        return torch.cat([x1, x2], dim=1)
```

当采用 MP(2 * c1,c1)时,即 c1 是 c2 的两倍时为 MP1 结构;当采用 MP(c1,c1)时,即 c1==c2 时为 MP2 结构。

对 SPPCSPC 模块进行封装,代码如下:

```
#第 5 章/ObjectDetection/Pytorch_Yolo_V7_Detected/conv_utils.py
class SPPCSPC(nn.Module):
    #SPPCSPC 模块
    def __init__(self, c1, c2, e=0.5, k=(5, 9, 13)):
        #c1 为输入,c2 为输出,e 为通道扩展倍数
        #k=(5, 9, 13)表明进行 SPP 时尺寸的变化
        super(SPPCSPC, self).__init__()
        c_ = int(2 *c2 *e) #输出 channel
        self.cv1 = CBS(c1, c_, 1, 1)
        self.cv2 = CBS(c1, c_, 1, 1)
        self.cv3 = CBS(c_, c_, 3, 1)
        self.cv4 = CBS(c_, c_, 1, 1)
        #对 k=(5, 9, 13)进行 3 个尺度的池化
        self.m = nn.ModuleList([nn.MaxPool2d(kernel_size=x, stride=1, padding=
x //2) for x in k])
        #因为 cv5 是对 cv4 和 m 的 cat,所以输入通道扩展了 4 倍
        self.cv5 = CBS(4 *c_, c_, 1, 1)
        #对 cv5 进行卷积,所以输入、输出又降为 c
        self.cv6 = CBS(c_, c_, 3, 1)
        #对 cv2 和 cv6 进行 cat,所以输入变为 2c
        self.cv7 = CBS(2 *c_, c2, 1, 1)

    def forward(self, x):
        #先进行串列卷积
        x1 = self.cv4(self.cv3(self.cv1(x)))
        #不同尺度池化,进行 cv5、cv6 的卷积
        y1 = self.cv6(self.cv5(torch.cat([x1] + [m(x1) for m in self.m], 1)))
        #对原输入 x 进行卷积
        y2 = self.cv2(x)
        #合并后卷积
        return self.cv7(torch.cat((y1, y2), dim=1))
```

SPPCSPC 模块对于输入 x 进行 3 次串列 cv1、cv3、cv4 的输出,从而得到 x1,然后对 x1 进行 5、9、13 的不同尺度池化 cat,从而得到 x2,将 x1 和 x2 合并进行两次串联 cv5、cv6,从而得到 y1。另 1 个分支对于 x 进行 1 次 cv2,从而得到 y2,将 y1 和 y2 进行 cat 后 cv7 得到最终的输出。该结构能够处理不同尺度的感受野,利于区分大目标和小目标。RepConv 模块可参考 4.12 节重参数网络 RepVGGNet 的实现。

将 CBS、ELAN、MP、ELAN-W、SPPCSPC 模块按模型图进行组装就能实现 YOLOv7 的前向传播,代码如下:

```
#第 5 章/ObjectDetection/Pytorch_Yolo_V7_Detected/yolo7.py
class Yolo7(nn.Module):
    #YOLOv7 模型的实现
    def __init__(self, num_class=80, layer_num_anchors=3):
        #num_class:预测类别数
```

```
        #layer_num_anchors:锚框数量
        super(Yolo7, self).__init__()
        self.num_class = num_class
        ####bakcbone
        #重复 4 次 cbs,即结构图中的 cbs_repeat4
        self.cbs1 = CBS(3, 32, 3, 1)
        self.cbs2 = CBS(32, 64, 3, 2)
        self.cbs3 = CBS(64, 64, 3, 1)
        self.cbs4 = CBS(64, 128, 3, 2)
        #第 1 个 ELAN->MP,输出是 b×(64×4) ×160×160
        #因为 ELAN 会对 out1、out2、out3 和 out4 进行合并,所以输出是 64×4
        self.elan_b1 = ELAN(128, 64)
        #mp1,输出是 b×(128×2) ×80×80,因为 MP 会对 x1 和 x2 合并,所以输出是 128×2
        self.mp1_b1 = MP(64 *4, 128)
        #第 2 个 ELAN->MP
        self.elan_b2 = ELAN(128 *2, 128)
        self.mp1_b2 = MP(128 *4, 256)                  #out b×(256×2) ×40 × 40
        #第 3 个 ELAN->MP
        self.elan_b3 = ELAN(256 *2, 256)
        self.mp1_b3 = MP(256 *4, 512)                  #out b×(512×2) ×20 × 20
        #ELAN->SPPCSP
        self.elan_b4 = ELAN(512 *2, 256)               #out b×(256×4) ×20 × 20
        self.sppcsp = SPPCSPC(512 *2, 512)             #out b×512×20 × 20

        ####FPN 结构,cbs->upsample->cat->ELAN_W...
        self.cbs_f1 = CBS(512, 256, 1, 1)
        self.up_f1 = nn.Upsample(scale_factor=2)
        self.cbs_route2 = CBS(1024, 256, 1, 1)
        self.cbs_route1 = CBS(512, 128, 1, 1)
        self.elan_w_f1 = ELAN_W(512, 256)
        self.cbs_f2 = CBS(256, 128, 1, 1)
        self.up_f2 = nn.Upsample(scale_factor=2)
        self.elan_w_f2 = ELAN_W(256, 128)
        ####PAN 结构下采样
        self.mp2_p1 = MP(128, 128)
        self.elan_w_p1 = ELAN_W(512, 256)
        self.mp2_p2 = MP(256, 256)
        self.elan_w_p2 = ELAN_W(1024, 512)
        ####head 输出头 4 代表位置,1 为置信度,num_class 为分类的概率,每个检测头有 3 个锚框
        out_c = (4 + 1 + num_class) *layer_num_anchors
        #第 1 个检测头,进行重参数化 255 × 80 ×80
        self.repcov1 = RepConv(128, 256)
        #预测输出
        self.cbs_out1 = CBS(256, out_c, act=False)
        #第 2 个检测头,进行重参数化,预测输出 255 × 40 ×40
        self.repcov2 = RepConv(256, 512)
        self.cbs_out2 = CBS(512, out_c)
        #第 3 个检测头,进行重参数化,预测输出 255 × 20 ×20
        self.repcov3 = RepConv(512, 1024)
        self.cbs_out3 = CBS(1024, out_c)

    def forward(self, x):
```

```
        #cbs repeat4->elan->mp1
        x = self.cbs4(self.cbs3(self.cbs2(self.cbs1(x)))) #1×128×160×160
        x = self.mp1_b1(self.elan_b1(x))                   #256 × 80 × 80

        #route1 会经过 cbs 进入 FPN 结构进行 cat
        route1 = self.elan_b2(x)                           #512 × 80 × 80
        x = self.mp1_b2(route1)                            #512 × 40 × 40

        #route2 会经过 cbs 进入 FPN 结构进行 cat
        route2 = self.elan_b3(x)                           #1024 × 40 × 40
        x = self.mp1_b3(route2)                            #1024 × 20 × 20

        #route3 会经过 cbs 进入 FPN 结构进行 cat
        x = self.elan_b4(x)                                #1024 × 20 × 20
        route3 = self.sppcsp(x)                            #512 × 20 × 20

        #FPN 上采样的构建 512 × 40 × 40
        #上采样,并实现浅深层特征合并,然后 ELAN-W
        x = torch.concat([self.up_f1(self.cbs_f1(route3)), self.cbs_route2
(route2)], dim=1)
        #这一层特征会进入 PAN 进行特征合并
        fpn_route1 = self.elan_w_f1(x)                     #256 × 40 × 40

        #继续上采样,并实现浅深层特征合并,然后 ELAN-W
        x = torch.concat([self.up_f2(self.cbs_f2(fpn_route1)), self.cbs_route1
(route1)], dim=1)                                          #256 × 80 × 80
        #上采样,并实现浅深层特征合并,然后 ELAN-W,并作为第 1 个检测层来检测小目标
        out1 = self.elan_w_f2(x) #128 × 80 × 80

        #PAN 下采样的构建 256 × 40 × 40
        #MP->CAT-ELAN-W 作为第 2 个检测头,检测中目标
        out2 = self.elan_w_p1(torch.concat([self.mp2_p1(out1), fpn_route1], dim=1))
        #继承 MP->CAT-ELAN-W 作为第 3 个检测头,检测大目标
        out3 = self.elan_w_p2(torch.concat([self.mp2_p2(out2), route3], dim=1))
                                                           #512 × 20 × 20

        #根据 out1、out2 和 out3 输出进行重参数化和检测
        p1 = self.cbs_out1(self.repcov1(out1))             #255 × 80 ×80
        p1_shape = p1.shape
        #维度(batch,3,height,width, (4 + 1 + self.num_class))
        #表示每个特征图有 3 个锚框,每个锚框有 4 个位置、1 个置信度和 num_class 个类别
        p1 = p1.view(p1_shape[0], 3, p1_shape[2], p1_shape[3], (4 + 1 + self.num_class))

        p2 = self.cbs_out2(self.repcov2(out2))             #255 × 40 ×40
        p2_shape = p2.shape
        p2 = p2.view(p2_shape[0], 3, p2_shape[2], p2_shape[3], (4 + 1 + self.num_class))

        p3 = self.cbs_out3(self.repcov3(out3))             #255 *20 *20
        p3_shape = p3.shape
        p3 = p3.view(p3_shape[0], 3, p3_shape[2], p3_shape[3], (4 + 1 + self.num_class))
        #返回 3 个输出
        return p1, p2, p3
```

在 YOLOv7 类中对模型进行了实现，__init__()为每个结构类初始化，在 forward()中进行了实现。提取 Backbone 特征，输出 route1、route2、route3，然后在 FPN 中对 route3 进行上采样，并与 self. cbs_route2(route2)进行特征合并，通过 self. elan_w_f1 输出 fpn_route1，并对 fpn_route1 继续上采样，然后与 self. cbs_route1(route1)进行特征合并，经过 self. elan_w_f2()后得到 out1。

在 out1 之后就是 PAN 结构，通过 self. mp2_p1(out1)进行下采样，并跟 fpn_route1 进行特征合并，然后经过 self. elan_w_p1 作为 out2；通过 self. mp2_p2(out2)进行下采样，并与 route3 进行特征合并，通过 self. elan_w_pw()输出 out3；在 out1、out2、out3 之后经过 self. repcov()重参数和 self. cbs_out 卷积，输出 p1、p2、p3 并通过 view(p1_shape[0],3,p1_shape[2],p1_shape[3],(4+1+self. num_class))方法输出(batch,3,height,width,(4+1+self. num_class))，表示每个特征图有 3 个锚框，每个锚框有 4 个位置、1 个置信度和 num_class 个类别。

## 5.10.3　代码实战建议框的生成

YOLOv7 的正样本提取参考了 YOLOv5 中的方法进行初选，然后由 OTA 算法进行精选，其代码实现过程较复杂，在此参考 YOLOv7 官方代码进行实现。

训练标签数据的文件格式的内容如下：

```
文件路径 xmin,ymin,xmax,ymax,label xmin,ymin,xmax,ymax,label
./face_train/19_Couple_Couple_19_461.jpg 441,432,525,528,0 592,435,676,546,0
```

PyTorch 读取自定义标签文件需要继承 Dataset 类，并在__init__()中进行属性的初始化，在__len__()中返回标签的数量，在__getitem__()中实现按 batch size 生成指定的内容，详细的代码如下：

```
#第 5 章/ObjectDetection/Pytorch_Yolo_V7_Detected/dataloader.py
class FaceDataSet(Dataset):
    def __init__(self, data_root, transform=None, size=(640, 640)):
        """
        自定义读取数据格式初始化
        :param data_dir: 路径
        :param transform: 数据预处理
        """
        self.transform = transform
        self.data_root = data_root
        self.img_size = size
        #获取训练图片和 label 信息
        self.data_info = self.get_data()

    def __len__(self):
        #类返回数据容量
        return len(self.data_info)

    def __getitem__(self, item):
```

```
        #从所有数据中获取指定长度的内容,只能返回维度相等的内容
        #所以需要指定 collate_fn 进行合并拼接
        datas = self.data_info[item]
        img, true_box, path = self.get_data_rows(datas)
        if self.transform is not None:
            #数据格式转换等
            img = self.transform(img)
        return img, true_box, path

    @staticmethod
    def collate_fn(batch):
        #根据指定 batch 返回 img 和 boxes 的信息
        img, true_box, path = zip(*batch)
        #将 true_box [n,6]中的第 1 位标明是哪一张图片
        for i, l in enumerate(true_box):
            l[..., 0] = i
        img = torch.stack(img, 0)
        true_box = torch.cat(true_box, 0)
        return img, true_box, path

    def get_data(self):
        with open(self.data_root, encoding='utf-8') as f:
            rows = f.readlines()
            return rows

    def get_data_rows(self, row):
        #存储 image path 和 boxes
        #按格式进行解析
        #./face_train/19_Couple_Couple_19_461.jpg 441,432,525,528,0 592,435,
        #676,546,0
        #解析后为['./face_train/19_Couple_Couple_19_461.jpg', '441,432,525,528,0',
        #'592,435,676,546,0']
        data = row.strip().split(' ')
        path = data[0]
        #读图片和 boxes
        img = cv2.imread(path)
        true_box = np.array([np.array(list(map(int, box1.split(',')))) for box1
in data[1:]])
        #转换为指定大小的图片和 boxes 信息
        img, true_box = u.letterbox_image(
            Image.fromarray(np.uint8(img.copy())),
            self.img_size, true_box
        )
        #print(true_box)
        #将 true_box 由 xmin,ymin,xmax,ymax 转换为 cx,cy,w,h
        true_box[..., :4] = u.xyxy2cxcywh(true_box)
        #[len,6]是为了在 collate_fn 中的第 1 个位置标明是第几张图片
        true_box_out = torch.zeros([len(true_box), 6])
        true_box = torch.from_numpy(true_box)
        if true_box[0] != 0:
            true_box_out[..., 1] = true_box[..., -1]
            true_box_out[..., 2:6] = true_box[..., 0:4] / self.img_size[0]
```

```
        return torch.tensor(img / 255., dtype=torch.float32).transpose(0, -1),
true_box_out, path
if __name__ == "__main__":
    data_root = r"../2021_train_yolo.txt"
    t = FaceDataSet(data_root)
    train_loader = DataLoader(
        dataset=t,
        batch_size=4,
        shuffle=True,
        collate_fn=FaceDataSet.collate_fn
    )
    #每次得到的是 batch size 的矩阵。在 data 中应保留 inputs 和 labels
    for i, (img, truebox, path) in enumerate(train_loader):
        #truebox 返回的是[4,6],第 1 位表明是第几张图片
        inputs, labels = img, truebox
```

在自定义 FaceDataSet 类的__init__()中 self.data_info 属性用于返回所有训练标签文件,如图 5-87 所示。

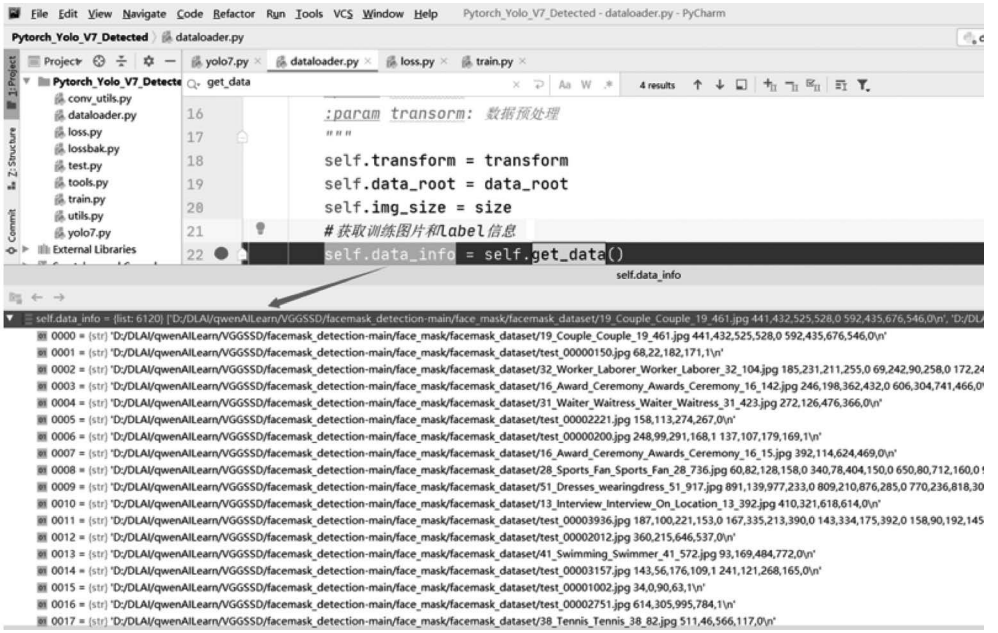



图 5-87　self.data_info 属性值

在__getitem__()中根据随机的 item 从 self.get_data_row 中获取数据,datas 为标签文件中的第 3902 行数据,如图 5-88 所示。

在 get_data_rows(row)中根据传入的标签内容,先对标签的数据进行清洗,然后经过 letterbox_image()对图片和 GT BOX 进行等比例缩放,并将输入的标签格式转换成 cx,cy,w,h,然后使用 true_box_out 矩阵的第 1 位标明当前图片是 batch size 中的第几张图片,使输出格式的矩阵变为[batch index,label,cx,cy,w,h]。

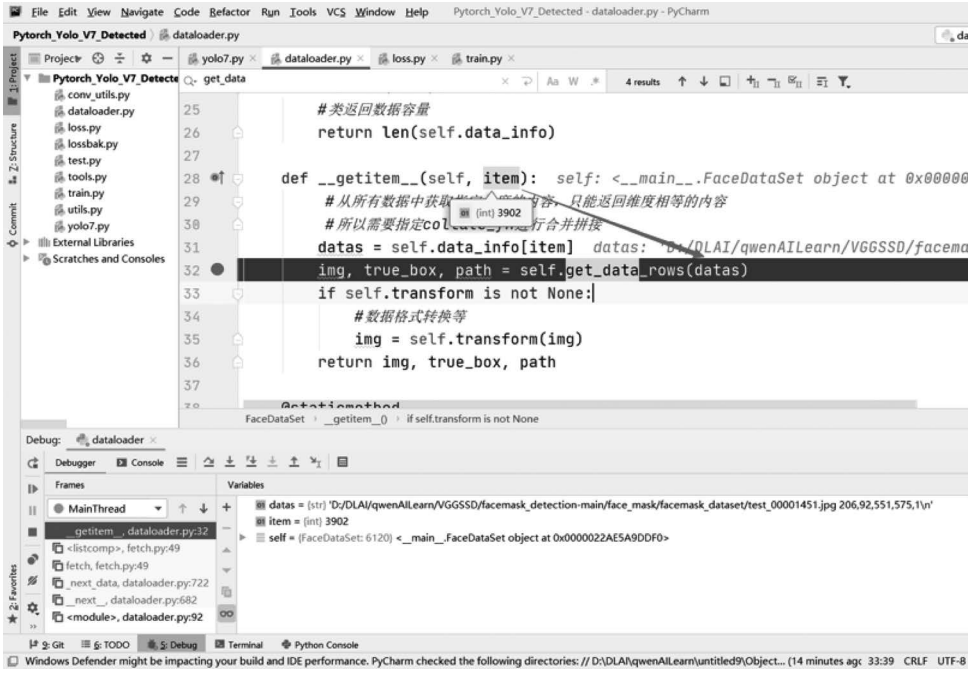



图 5-88 __getitem__()构造函数

torch.tensor(img/255.,dtype=torch.float32).transpose(0, -1)是由于 OpenCV 读取的格式是[height,width,channel],而 PyTorch 格式是[channel,height,width],所以图片的 shape 要进行改变,如图 5-89 所示。

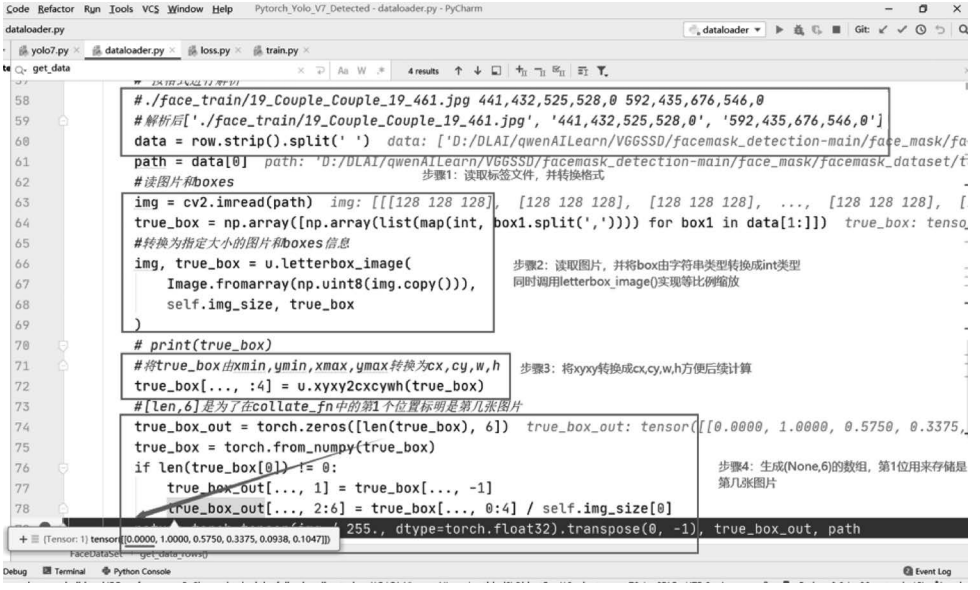



图 5-89 self.get_data_rows 的实现

在 collate_fn(batch)中根据 batch size 返回数据。batch 变量用于存储 img、true_box、path 的信息，如图 5-90 所示。

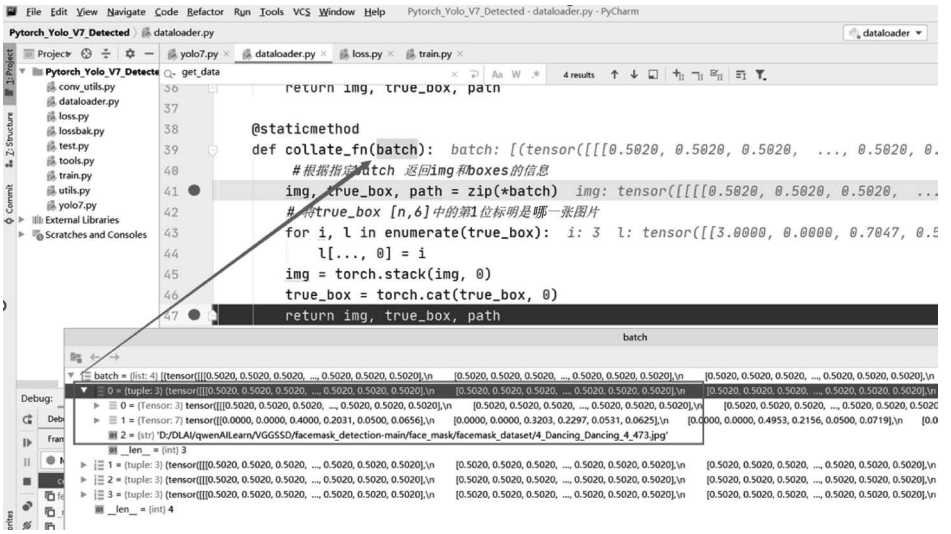


图 5-90　batch 参数的值

然后根据 enumerate(true_box)中的内容，通过 l[...,0]=i 修改第 1 位的值，以此来表明是 batch size 中的第几张图片。最后通过 img=torch.stack(img,0)对图片数据进行合并，true_box=torch.cat(true_box,0)实现标签数据的合并，如图 5-91 所示。

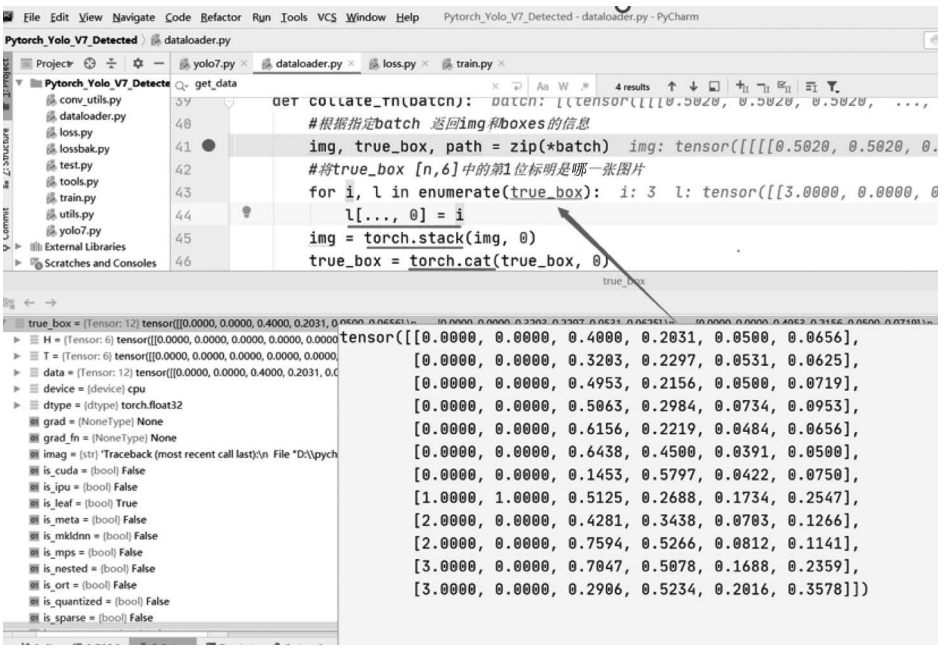


图 5-91　collate_fn 的实现

然后使用 DataLoader 构建生成器 train_loader，经 enumerate(train_loader)之后便可获得指定 batch size 的图片和 label 信息，如图 5-92 所示。

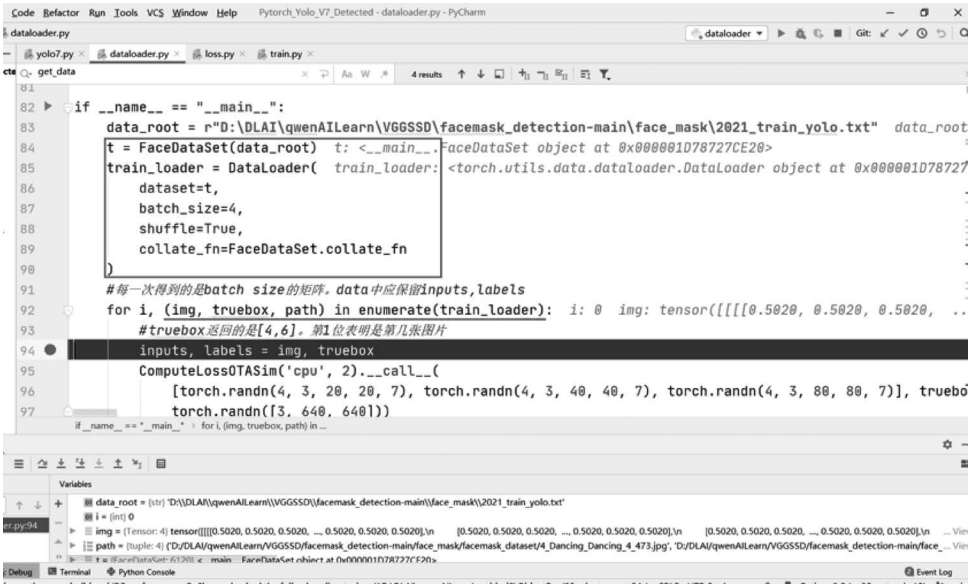



图 5-92　DataLoader 的实现

读取完数据后根据 YOLOv5 中的实现，先将 GT BOX 与 Anchor BOX 进行高宽比，如果小于 4，则入选，同时再以 GT BOX 为中心点挑选附近的 3 个格子作为正样本，此函数被封装在 find_3_positive(self,p,targets)中，p 为 3 个检测头的预测值，targets 为 GT BOX 的标签值，详细的代码如下：

```
#第 5 章/ObjectDetection/Pytorch_Yolo_V7_Detected/loss.py
def find_3_positive(self, p, targets):
    na = len(self.anchor)                          #Anchor 的数量
    nt = targets.shape[0]                          #GT BOX 的数量,4 × 6
    #存放 indices 和 anchors
    indices, anch = [], []
    #序号 2:6 为特征层的高宽
    gain = torch.ones(7, device=targets.device).long()
    #GT BOX 的下标
    ai = torch.arange(na, device=targets.device).float().view(na, 1).repeat(1, nt)
    #实现每个 Anchor 都复制一份 GT 的信息
    targets = torch.cat((targets.repeat(na, 1, 1), ai[:, :, None]), 2)
    #targets [0.0000, 1.0000, 0.4344, 0.2141, 0.7750, 0.6594, 2.0000]
    #第几张图, 类别, gt_cx, gt_cy, gt_h, gt_w, 第几个 Anchor
    g = 0.5 #偏置
    off = torch.tensor([
        [0, 0],
        [1, 0], [0, 1], [-1, 0], [0, -1],          #j,k,l,m
    ], device=targets.device).float() *g           #offsets
    #遍历每个检测头
```

```
        for i in range(len(p)):
            #每个检测头对应的 Anchor
            anchors = self.anchor[2 - i] *p[i].shape[2]
            #4×3×20×20×7-->20×20×20×20
            #gain 本来是[1,1,1,1,1,1,1],此时变为[1,1,20,20,20,20,1]
            gain[2:6] = torch.tensor(p[i].shape)[[3, 2, 3, 2]]   #xyxy gain
            #将 targets 乘以 gain,将真实框映射到特征层上。每个格子都存有 targets 值
            t = targets *gain
            #如果存在 GT
            if nt:
                #4:6 为 GT BOX 的高宽,在 YOLOv5 中根据高宽比来确定正样本
                r = t[:, :, 4:6] / anchors[:, None]
                #高宽比小于 4.0 的 mask
                j = torch.max(r, 1. / r).max(2)[0] < 4.0
                t = t[j]                                    #通过 j 的布尔值过滤出 t 的正样本

                #gxy 用于获得 t 对应的正样本的 x 坐标和 y 坐标
                gxy = t[:, 2:4]
                #gxi 用于获取每个格子的 xy 坐标 20*20
                gxi = gain[[2, 3]] - gxy                    #gxy 离每个格子左上角点的距离
                #根据 gxi 的值,计算附近的格子的 mask
                j, k = ((gxy % 1. < g) & (gxy > 1.)).T
                l, m = ((gxi % 1. < g) & (gxi > 1.)).T
                j = torch.stack((torch.ones_like(j), j, k, l, m))
                #t 重复 5 次,使用满足条件的 j 进行框的提取
                #假设 t 本来有 17 个,则先扩充 5 倍,变成[5,17,4+1+num_class]
                #j 代表当前特征点在[0, 0], [1, 0], [0, 1], [-1, 0], [0, -1]方向是否存在
                t = t.repeat((5, 1, 1))[j] #在[5,17,4+1+num_class]个样本中根据 mask j 提
                                           #取正样本,就变成了[50,7]
                offsets = (torch.zeros_like(gxy)[None] + off[:, None])[j] #附近新正样本
                                                                          #的偏移值
            else:
                #没有目标当负样本
                t = targets[0]
                offsets = 0
            #b 为第几张图片,c 为类别
            b, c = t[:, :2].long().T
            gxy = t[:, 2:4]                                 #正样本 xy
            #gwh = t[:, 4:6]                                #正样本 wh
            gij = (gxy - offsets).long()                    #偏移值
            gi, gj = gij.T                                  #得到在 gi,gj 个格子中存在目标

            #a 为第几个 Anchor
            a = t[:, 6].long()
            #返回 indices [第几张图片,第几个 anchor,第 j 个列,第 i 行]
            indices.append(
                (b, a, gj.clamp_(0, gain[3] - 1),           #gj.clamp_(0, gain[3] - 1) 限定格
                                                            #子的位置在 0~19
                 gi.clamp_(0, gain[2] - 1))
            )                                               #image, anchor, grid indices
            anch.append(anchors[a])                         #此时正样本的 anchors 值是多少
        return indices, anch
```

代码较难理解，先看第 1 部分，targets 为输入 GT BOX 的信息，因为 batch size＝4，所以这里的 shape＝4＊6。ai 这个变量根据设置的 anchor＝3 数量，生成 Anchor 的下标索引值，因为 Anchor 也有 4 个位置，所以 ai 的值为 tensor([[0.,0.,0.,0.],[1.,1.,1.,1.],[2.,2.,2.,2.]])；targets.repeat(na,1,1)将每个 Anchor 都分配 targets 的内容，所以 shape＝3＊4＊6，然后与 ai[…,None]升维合并后变成 shape＝3＊4＊7，如内容[0.0000,1.0000,0.4344,0.2141,0.7750,0.6594,2.0000]的含义为[第几张图，类别，gt_cx，gt_cy，gt_h，gt_w，第几个 anchor]，如图 5-93 所示。

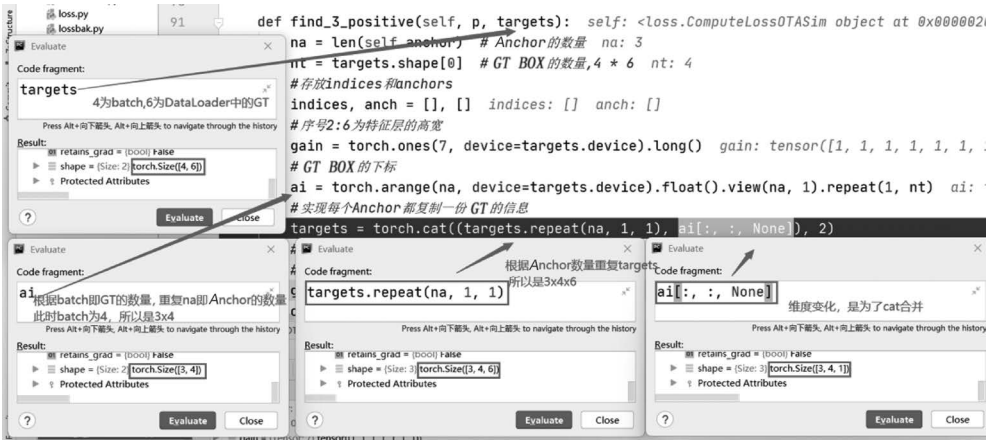



图 5-93　find_3_positive 中的 targets

第 2 部分，offset 描述了每个当前格子，如果假设为[0,0]，则附近的格式的位置为[1,0]，[0,1]，[－1,0]，[0,－1]。anchors＝self.anchor[2－i]＊p[i].shape[2]根据当前预测 p 的 shape 计算 Anchor 在当前特征图上的大小。t＝targets＊gain 为计算 GT BOX 在当前特征图上的大小；gain 变量用来存储计算后的值。r＝t[:,:,4:6] / anchors[:,None]用于得到当前特征图上 GT BOX 与 anchors 的高宽比，并通过 j＝torch.max(r,1./r).max(2)[0]＜4.0 来判断宽高比或者高宽比是否小于 4，如果小于 4，则 j 为 True，否则为 False。t[j]用于取出满足小于 4.0 的 GT BOX 在此特征图上的值，关键代码如图 5-94 所示。

第 3 部分，t[:,2:4]为 targets 的 $x$ 和 $y$ 坐标，假设特征图为 20＊20，则 gain[[2,3]]＝20＊20，则 gxi＝gain[[2,3]]－gxy 用于获取 targets 离每个格子左上角所在的格子数。(gxy％1.＜g)&(gxy＞1.)用于计算当前格子偏离每个格子中心点的位置，得到 $j$、$i$ 个格子的布尔值，然后将 t＝t.repeat((5,1,1))[j]复制 5 份，通过 $j$ 的布尔值得到离 GT 中心点最近的 3 个框，假设原有 11 个正样本 BOX，那么此时就扩展为 33 个正样本，如图 5-95 所示。

第 4 部分，根据 offsets 值 gij＝(gxy－offsets).long()计算所在格子的位置，根据 t[:,6].long()计算 Anchor 的位置，indices 返回[第几张图片，第几个 anchor，第 j 个列，第 i 行]，anch.append(anchors[a])返回此时正样本的 anchors 值，如图 5-96 所示。

根据上面初选的正样本，进入复选流程，复选的代码在 build_targets()中，详细的代码如下：

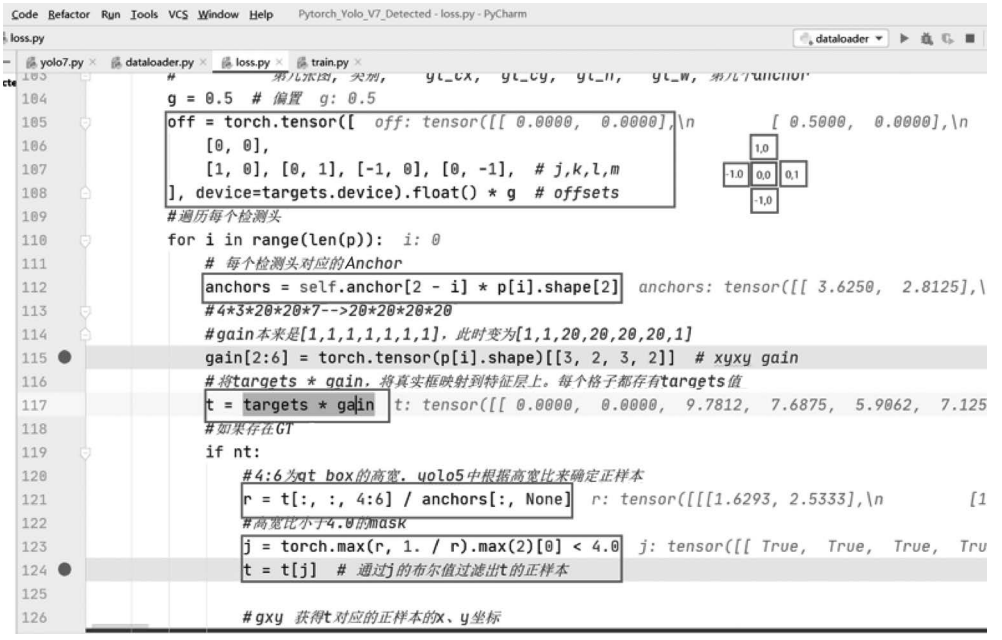



图 5-94　find_3_positive 中的 GT BOX 与 Anchors 的高宽比

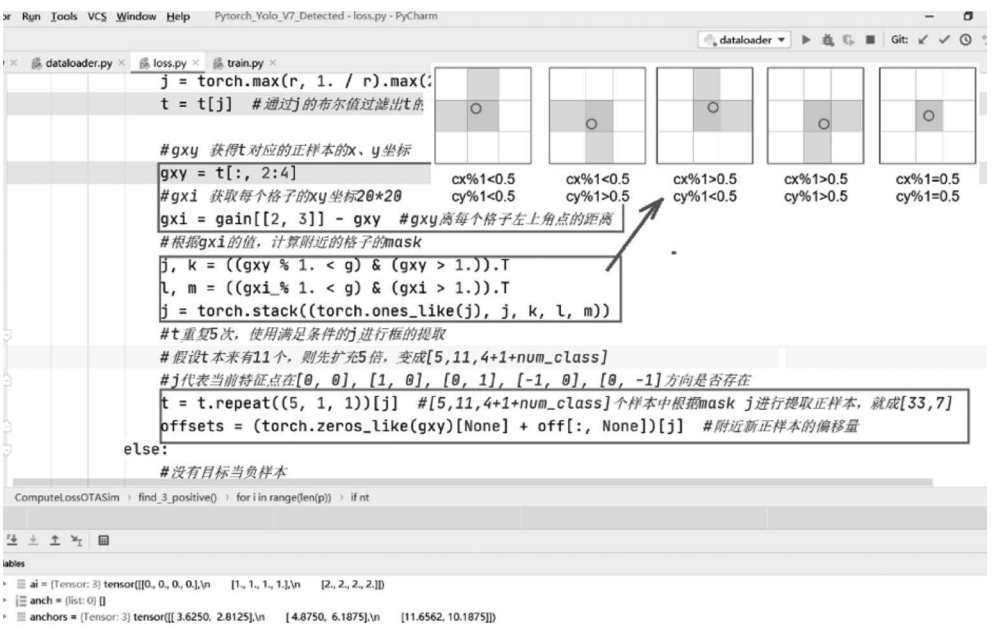



图 5-95　find_3_positive 中正样本的扩展

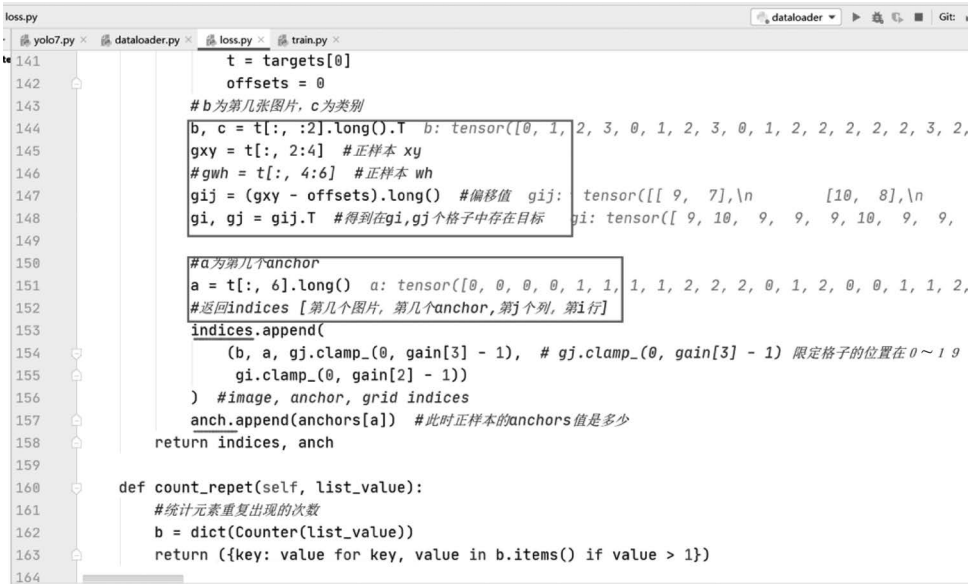



图 5-96 find_3_positive 返回 indices

```
#第 5 章/ObjectDetection/Pytorch_Yolo_V7_Detected/loss.py

def build_targets(self, p, targets, imgs):
    device = torch.device(targets.device)
    #初选,寻找目标 GT 中心点位置附近的 3 个格子作为正样本
    indices, anch = self.find_3_positive(p, targets)

    #匹配,[batch img,anchor,j,i,gt box, anchors 此时的比例]
    matching_bs = [[] for pp in p]
    matching_as = [[] for pp in p]
    matching_gjs = [[] for pp in p]
    matching_gis = [[] for pp in p]
    matching_targets = [[] for pp in p]
    matching_anchs = [[] for pp in p]
    #检测头
    nl = len(p)
    #p[0].shape[0]即 batch size,遍历每幅图来计算
    for batch_idx in range(p[0].shape[0]):
        #targets [0.0000, 1.0000, 0.4344, 0.2141, 0.7750, 0.6594, 2.0000]
        #第几张图, 类别, gt_cx, gt_cy, gt_h, gt_w, 第几个 Anchor
        #只取当前图的 targets 信息
        b_idx = targets[:, 0] == batch_idx
        this_target = targets[b_idx]
        #如果没有 GT BOX,则不处理
        if this_target.shape[0] == 0:
            continue
        #因为 target 是归一化后的值,所以如果乘原图的尺寸,则返回真实的 txywh
        #这样算的原因是后面要计算 IOU
        txywh = this_target[:, 2:6] * imgs[batch_idx].shape[1]
```

```
            #将 xywh 转换成 xmin,ymin,xmax,ymax,也是为了计算 IOU
            txyxy = tools.xywh2xyxy(txywh)
            #预测 xyxy,预测分类,预测置信度
            pxyxys = []
            p_cls = []
            p_obj = []
            #在哪个预测层
            from_which_layer = []
            #存储所有的[batch img,anchor,j,i,gt box, anchors 此时的比例]
            all_b = []
            all_a = []
            all_gj = []
            all_gi = []
            all_anch = []
            #遍历每个预测层
            for i, pi in enumerate(p):
               b, a, gj, gi = indices[i] #根据检测头,从初选中获取
               #获取当前图片
               idx = (b == batch_idx)
               #进一步获取当前图片
               b, a, gj, gi, anchi = b[idx], a[idx], gj[idx], gi[idx], anch[i][idx]
               all_b.append(b)
               all_a.append(a)
               all_gj.append(gj)
               all_gi.append(gi)
               all_anch.append(anchi)
               from_which_layer.append((torch.ones(size=(len(b),)) *i).to(device))

               #在预测结果中根据真实 b,a,gj,gi 进行筛选
               fg_pred = pi[b, a, gj, gi]
               #存储预测结果置信度和分类
               p_obj.append(fg_pred[:, 4:5])
               p_cls.append(fg_pred[:, 5:])
               #所在格子合并
               grid = torch.stack([gi, gj], dim=1)
               #因为预测出来的是偏移值,所以需要解码成 xyxy
               #即式(2×sigmoid(txy)-0.5 + grid) × 32
               pxy = (fg_pred[:, :2].sigmoid() *2. - 0.5 + grid) *self.stride[i]   #32
               pwh = (fg_pred[:, 2:4].sigmoid() *2) **2 *anchi *self.stride[i]   #32
               pxywh = torch.cat([pxy, pwh], dim=-1)
               pxyxy = tools.xywh2xyxy(pxywh)
               pxyxys.append(pxyxy)
            #合并 3 个检测头的预测值
            pxyxys = torch.cat(pxyxys, dim=0)
            if pxyxys.shape[0] == 0:
               continue
            #对每个层预测的相关信息进行合并
            p_obj = torch.cat(p_obj, dim=0)
            p_cls = torch.cat(p_cls, dim=0)
            from_which_layer = torch.cat(from_which_layer, dim=0)
            all_b = torch.cat(all_b, dim=0)
            all_a = torch.cat(all_a, dim=0)
```

```
        all_gj = torch.cat(all_gj, dim=0)
        all_gi = torch.cat(all_gi, dim=0)
        all_anch = torch.cat(all_anch, dim=0)

        #每张图片和预测值进行 IOU
        pair_wise_iou = tools.box_iou(txyxy, pxyxys) #GT BOX 和预测值进行 IOU
        #-log(pair_wise_iou),pair_wise_iou 越大,-log(y)就越小。反之离得越远,重合度越低
        pair_wise_iou_loss = -torch.log(pair_wise_iou + 1e-8)
        #假设 pair_wise_iou.shape[1]=66,则取 10 个。如果 pair_wise_iou.shape[1]=1,则取 1
        top_k, _ = torch.topk(pair_wise_iou, min(10, pair_wise_iou.shape[1]), dim=1)
        #得到动态 k,假设为[2,2,3,3],代表 4 个 GT BOX 中的每个与 pre box 的前 10 个 IOU
        #得分之和的值
        dynamic_ks = torch.clamp(top_k.sum(1).int(), min=1)
        #根据标签类别进行 one_hot 编码并升维与 pxyxys 一致[预测 box 的个数与正样本个数相同]
        gt_cls_per_image = (
           F.one_hot(this_target[:, 1].to(torch.int64), self.num_class)
              .float()
              .unsqueeze(1)
              .repeat(1, pxyxys.shape[0], 1)
        )
        #GT 的数量
        num_gt = this_target.shape[0]
        #预测 置信度 *分类
        cls_preds_ = (
              p_cls.float().unsqueeze(0).repeat(num_gt, 1, 1).sigmoid_()
              *p_obj.unsqueeze(0).repeat(num_gt, 1, 1).sigmoid_()
        )
        #开平方根
        y = cls_preds_.sqrt_()
        #将 GT BOX 的置信度与预测 log(y/1-y)进行求交叉熵损失,从而得到 pair_wise_cls_
        #loss 损失
        pair_wise_cls_loss = F.binary_cross_entropy_with_logits(
           torch.log(y / (1 - y)), gt_cls_per_image, reduction="none"
        ).sum(-1)
        del cls_preds_
        #精选总损失
        cost = (
           #置信度的损失
              pair_wise_cls_loss
              #GT BOX 与 Pre BOX 重合度的损失。3.0 表明 cost 更多地学习 BOX 之间的重叠
              + 3.0 *pair_wise_iou_loss
        )
        #根据 cost 初始矩阵,假设 pxyxys 是 12 个,则 cost 也是 12 个
        #pxyxys 的个数与正样本的 i,j,k 有关。根据正样本的 i,j,k 取对应的预测框
        matching_matrix = torch.zeros_like(cost, device=device)
        new_pos_idx = []
        for gt_idx in range(num_gt): #循环每个 GT BOX,gt_idx 为其下标
           _, pos_idx = torch.topk(
              cost[gt_idx], k=dynamic_ks[gt_idx].item(), largest=False
           ) #dynamic_ks 按 batch size 生成 tensor([2, 2, 3, 3, 3, 3], dtype=torch.int32)
           #如果 gt_idx=0,则 k = 2。cost[0]即损失 12 个预测框中的第 1 组。topk 则取第 1 组
           #的两个损失,并且 largest 是从小到大
```

```
            matching_matrix[gt_idx][pos_idx] = 1.0  #matching_matrix 将对应位置设置为 1
            #如果 match[0][[17,17]] = 1.0,则表明此时有一个 Anchor 被分配到多个 GT
            new_pos_idx += list(pos_idx.NumPy())

        del top_k, dynamic_ks
        #假设 matching_matrix 4×21,将变为 21
        anchor_matching_gt = matching_matrix.sum(0)
        if (anchor_matching_gt > 1).sum() > 0:    #Anchor 匹配 GT 的个数,如果大于 1,
                                                  #则需要去重
            print(self.count_repet(new_pos_idx)) #pos_idx 如果有重复项,就会去重
            #anchor_matching_gt > 1,只留 1 个 Anchor 匹配 1 个 GT 的内容。
            #anchor_matching_gt 如果有 2,则会舍弃
            #cost[:, anchor_matching_gt > 1]选出不重复 Anchor 匹配 GT 的 cost 损失
            #torch.min 表明取 cost 里面的最小值。返回最小值_和其索引 cost_argmin_,
            #cost_argmin = torch.min(cost[:, anchor_matching_gt > 1], dim=0)
            #将 anchor_matching_gt>1 在 matching_matrix 中的值设置为 0;如果原来是 2,
            #则现在变成 0
            matching_matrix[:, anchor_matching_gt > 1] *= 0.0
            #将 anchor_matching_gt>1 且可以令 cost 最小的位置重新设置为 1,表明只取
            #1 个 anchor
            matching_matrix[cost_argmin, anchor_matching_gt > 1] = 1.0
        #matching_matrix 已去掉重复的 Anchor 分配到某个 GT 上。再次求 sum(0)>0.0 是否
        #存在 Anchor
        fg_mask_inboxes = (matching_matrix.sum(0) > 0.0).to(device)
        #最后确定有多少个正样本,并确定其编号
        matched_gt_inds = matching_matrix[:, fg_mask_inboxes].argmax(0)
        #根据最后的正样本的 index 去筛选相关信息
        from_which_layer, all_b, \
        all_a, all_gj, all_gi, all_anch = [
            x[fg_mask_inboxes] for x in
            [from_which_layer, all_b, all_a, all_gj, all_gi, all_anch]
        ]
        #根据正样本对 GT BOX 的对应的位置进行赋值。使两者维度保持一致
        this_target = this_target[matched_gt_inds]
        #遍历 3 个检测头。根据检测头的编号重新整理 all_b 的值。使 b,a,j,i 等值分配到
        #正确的 layer 头上
        #matching_bs,最后正样本分配在 3 个头的存储
        for i in range(nl):
            layer_idx = from_which_layer == i
            matching_bs[i].append(all_b[layer_idx])
            matching_as[i].append(all_a[layer_idx])
            matching_gjs[i].append(all_gj[layer_idx])
            matching_gis[i].append(all_gi[layer_idx])
            matching_targets[i].append(this_target[layer_idx])
            matching_anchs[i].append(all_anch[layer_idx])
        #按 batch size 重新整合
        for i in range(nl):
            if matching_targets[i] != []:
                matching_bs[i] = torch.cat(matching_bs[i], dim=0)
                matching_as[i] = torch.cat(matching_as[i], dim=0)
                matching_gjs[i] = torch.cat(matching_gjs[i], dim=0)
                matching_gis[i] = torch.cat(matching_gis[i], dim=0)
```

```
                matching_targets[i] = torch.cat(matching_targets[i], dim=0)
                matching_anchs[i] = torch.cat(matching_anchs[i], dim=0)
            else:
                matching_bs[i] = torch.tensor([], device=device, dtype=torch.int64)
                matching_as[i] = torch.tensor([], device=device, dtype=torch.int64)
                matching_gjs[i] = torch.tensor([], device=device, dtype=torch.int64)
                matching_gis[i] = torch.tensor([], device=device, dtype=torch.int64)
                matching_targets[i] = torch.tensor([], device=device, dtype=torch.int64)
                matching_anchs[i] = torch.tensor([], device=device, dtype=torch.int64)
        #最后返回 3 个检测头中的相关信息
        return matching_bs, matching_as, matching_gjs, matching_gis, matching_
    targets, matching_anchs
```

代码较长且实现较复杂，先看第 1 部分代码中 matching_的相关变量，用来存储经过复选的正样本内容，包括每个 batch 中的图片，anchor 值、第 ji 个格子，GT BOX 的值，以及此时 anchor 值，[[]for pp in p]是由于有 3 个检测头，所以需要循环接收。在循环每个 batch 的图片中，通过 b_idx＝targets[:,0]＝＝batch_idx 获取当前图片的 b_idx，按每张图片的 b_idx 去获取 this_target[:,2:6]在当前图片中的 txywh 值并转换成 xmin，ymin，xmax，ymax 值，以此来计算 GT BOX 与 Pre BOX 预测值的 IOU，如图 5-97 所示。


图 5-97　build_targets 获取当前 batch idx 的真实值

第 2 部分，根据预测值当前所在 batch 及 find_3_positive()返回的[batch img，anchor，j，i，gt box，anchors 值]去预测值 fg_pred 中获取预测的置信度和分类信息的概率，然后使用 pxy＝(fg_pred[:,:2].sigmoid()＊2.－0.5＋grid)＊self.stride[i]对于预测的偏移值进行解码操作，转换成 cx，cy，w，h。tools.xywh2xyxy(pxywh)将预测值又转换成 xmin，ymin，xmax，ymax 的内容，循环 enumerate(p)，从而得到 3 个检测头的预测值，如图 5-98 所示。

第 3 部分，计算正样本 txyxy 与预测 pxyxys 的 IOU，存储在 pair_wise_iou 中，然后由 torch.topk(pair_wise_iou，min(10，pair_wise_iou.shape[1])，dim＝1)计算 pair_wise_iou

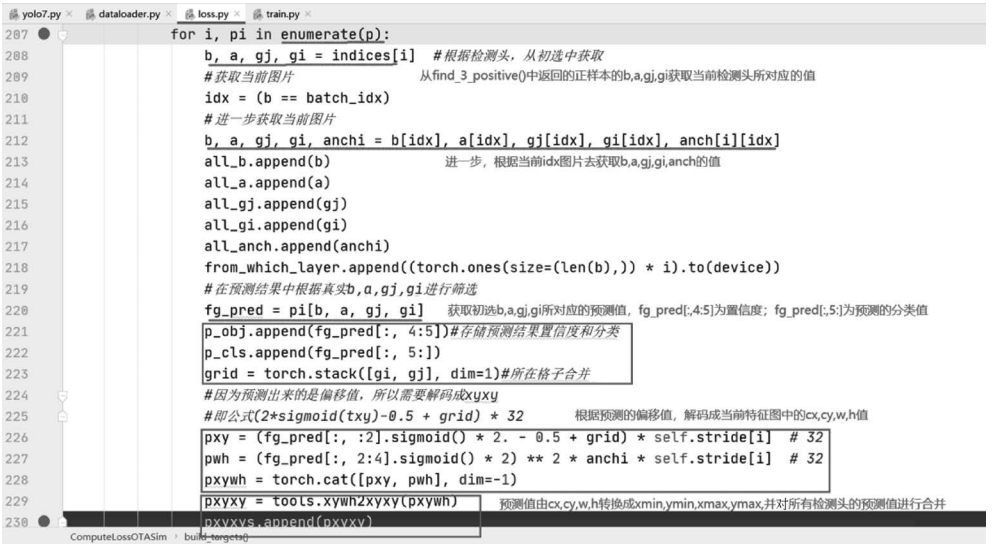



图 5-98　build_targets 中 pxyxys 的获取

中最大 IOU 值前 10 个 top_k. sum(1). int()之和并作为正样本的 dynamic_ks 个数。gt_cls_per_image 根据标签类别进行 one_hot 编码，升维后跟 pxyxys 一致。cls_preds_是预测置信度 p_obj * 分类 p_cls 的概率。－torch. log(pair_wise_iou＋1e-8)表明 pair_wise_iou 值越大，－torch. log(pair_wise_iou)就越小，反之离得越远，重合度越低，如图 5-99 所示。

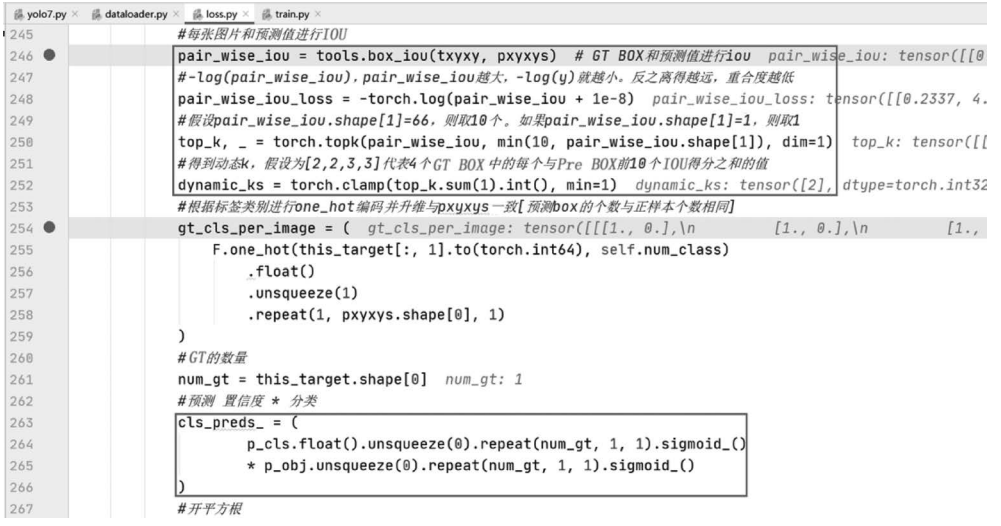



图 5-99　从 build_targets 中获取动态 k 的值决定正样本

第 4 部分，将预测置信度 * 分类概率 cls_preds_与每张图片中分类的概率 gt_cls_per_image 做 binary_cross_entropy_with_logits()交叉熵损失，并对 pair_wise_cls_loss＋3. 0 * pair_wise_iou_loss 进行求和，得到可以令置信度损失和回归损失最小的 cost，也就是

dynamic_ks 个能够令选出的正样本的置信度损失与回归损失最小，如图 5-100 所示。

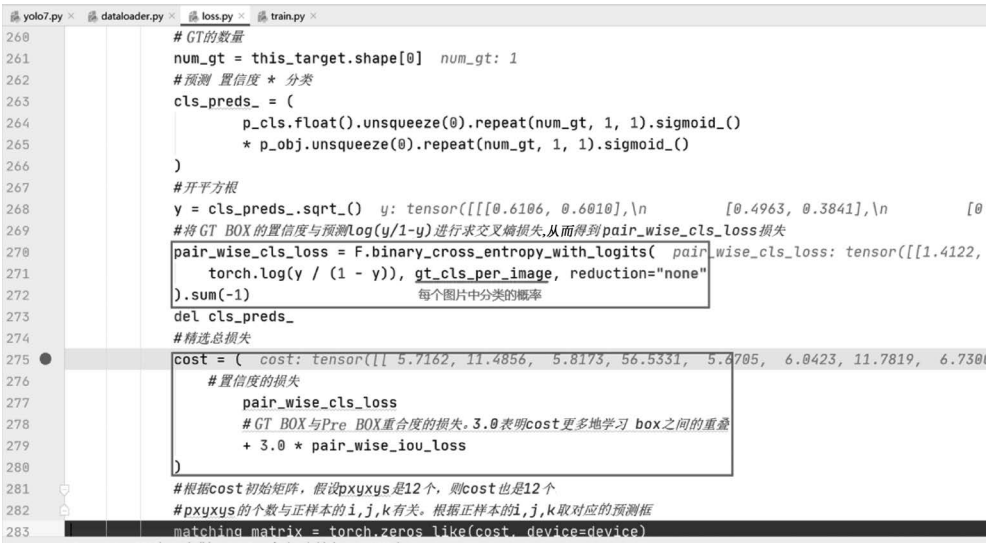



图 5-100 build_targets 中的 dynamic_ks 决定 cost 最小

第 5 部分，根据 cost 值，遍历每个 GT BOX，取出 dynamic_ks 动态 K 能够令 cost 最小的正样本索引 pos_idx，并且如果此时存在，则将 matching_matrix[gt_idx][pos_idx]=1.0，表明当前 gt_idx 和 pos_idx 都进行选择。new_pos_idx 用来统计每个 pos_idx 的索引号，如图 5-101 所示。


图 5-101 从 build_targets 中获取正样本 pos_idx

第 6 部分，如果 pos_idx 存在重复，则表明某个 Anchor 被分配给多个 GT BOX。在理想情况下，一个 GT BOX 可以有多个正样本，但是一个 Anchor 应该只分配 1 个 GT BOX。

在 matching_matrix 中存储 gt_idx 和 pos_idx 的值，假设 matching_matrix[0][[17,17]]=1.0，则表明 pos_idx 被分配给多个 GT BOX，所以如果 matching_matrix.sum(0)>1，则表明有重复 Anchor 被分配给 GT BOX，所以 matching_matrix[:,anchor_matching_gt>1] *= 0.0 可以令 anchor_matching_gt>1 的位置在 matching_matrix 中将值设置为 0，从而剔除重复项，并得到最后的正样本 fg_mask_inboxes，如图 5-102 所示。

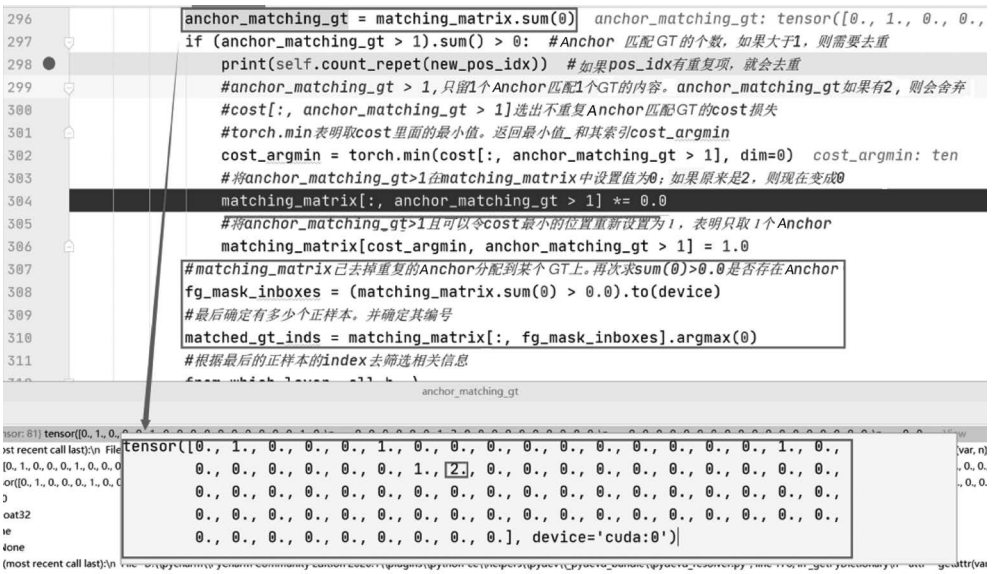



图 5-102　从 build_targets 中去除重复 pos_idx

最后遍历 3 个检测头，根据检测头的编号重新整理 all_b 的值，使 b,a,j,i 值分配到正确的 layer 头上，并使 matching_bs 分配正确，至此 build_targets()完成正样本的提取。

整个正样本的代码比较复杂，建议下载源码后根据注释和解释进行调试。

## 5.10.4　代码实现损失函数的构建及训练

损失函数的构建根据 self.build_targets(p,targets,imgs)返回的 bs、as_、gjs、gis、targets 和 anchors 值遍历 3 个检测头进行位置损失计算，分类和置信度损失进行计算，并对 3 个检测头的损失按指定的比例进行相加，详细的代码如下：

```
#第 5 章/ObjectDetection/Pytorch_Yolo_V7_Detected/loss.py
class ComputeLossOTASim(object):
    def __call__(self, p, targets, imgs):
        device = targets.device
        #分类、位置和置信度的初始值
        lcls, lbox, lobj = [torch.zeros(1, device=device) for _ in range(3)]
        #在 build_targets 中进行正样本的匹配，分为精选和复选
        bs, as_, gjs, gis, targets, anchors = self.build_targets(p, targets, imgs)
        #因为 p 是 3 个检测头的输出，而 pp 则是 1*3anchor *80*80*num_class 等
```

```
        #所以[3, 2, 3, 2]得到的 Tensor 为 80*80*80*80
        pre_gen_gains = [torch.tensor(pp.shape, device=device)[[3, 2, 3, 2]] for
    pp in p]
        #i 表明是第几个检测头。pi 为取的特征
        for i, pi in enumerate(p):
            #b 为第几张图,a 表示第几个 Anchor,gj 表示第 j 个格子,gi 表示第 i 格子
            b, a, gj, gi = bs[i], as_[i], gjs[i], gis[i]
            #初始 GT 的矩阵跟预测出来的结果保持一致
            tobj = torch.zeros_like(pi[..., 0], device=device)
            n = b.shape[0]
            if n:
                ps = pi[b, a, gj, gi]       #根据 build_targets 返回的 index 在预测值
                                            #中取对应位置的值
                #在哪个格子
                grid = torch.stack([gi, gj], dim=1)
                #对预测值限制值域
                pxy = ps[:, :2].sigmoid() *2. - 0.5
                pwh = (ps[:, 2:4].sigmoid() *2) **2 *anchors[i]
                pbox = torch.cat((pxy, pwh), 1) #合并
                #将真实框映射到特性图上
                selected_tbox = targets[i][:, 2:6] *pre_gen_gains[i]
                #计算真实框的偏移值
                selected_tbox[:, :2] -= grid
                #求预测 BOX 与真实 BOX 之间的 loss,这里使用的是 CIOU 损失

                iou = tools.bbox_iou(pbox.T, selected_tbox, x1y1x2y2=False, CIoU=True)
                lbox += (1.0 - iou).mean() #
                #tobj 置信度损失,self.gr obj loss 的权重。原作者默认写为 1
                #tobj[b, a, gj, gi],即对应 build_targets 的位置的置信度值
                tobj[b, a, gj, gi] = (1.0 - self.gr) + self.gr *iou.detach().clamp(0).
    type(tobj.dtype)
                #GT 的分类值
                selected_tcls = targets[i][:, 1].long()
                if self.num_class > 1:
                    t = torch.full_like(ps[:, 5:], self.cn, device=device)
                    t[range(n), selected_tcls] = self.cp
                    #分类的损失
                    lcls += self.BCEcls(ps[:, 5:], t)
                #置信度损失
                obji = self.BCEobj(pi[..., 4], tobj)
                lobj += obji *self.balance[i]     #不同检测头的权重不同
        lbox *= self.hyp['box']
        lobj *= self.hyp['obj']
        lcls *= self.hyp['cls']
        bs = tobj.shape[0] #batch size
        #总损失
        loss = lbox + lobj + lcls
        return loss *bs, torch.cat((lbox, lobj, lcls, loss)).detach()
```

第 1 部分代码根据 self. build_targets(p,targets,imgs)获得 bs、as_、gjs、gis、targets 和 anchors,然后遍历 3 个检测头,得到当前检测头中的 grid=torch. stack([gi,gj],dim=1)所在的格子,限制预测值的值域后计算真实框的偏移值 selected_tbox[:, :2]--grid,然后将

预测框 pbox 与真实框 selected_tbox 做 CIOU 损失，如图 5-103 所示。

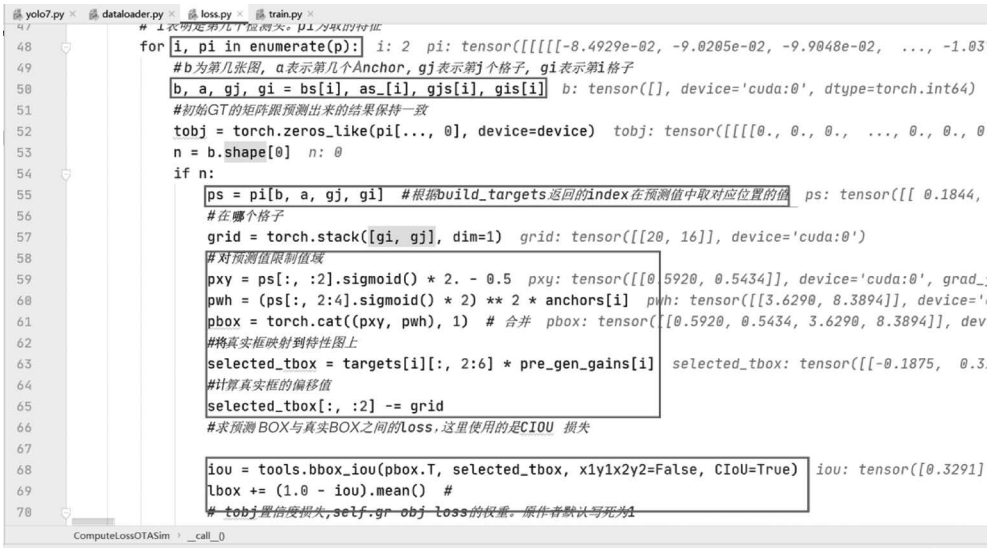



图 5-103　CIOU 位置损失

然后对分类损失、置信度损失、回归损失进行求和，从而得到总损失，如图 5-104 所示。

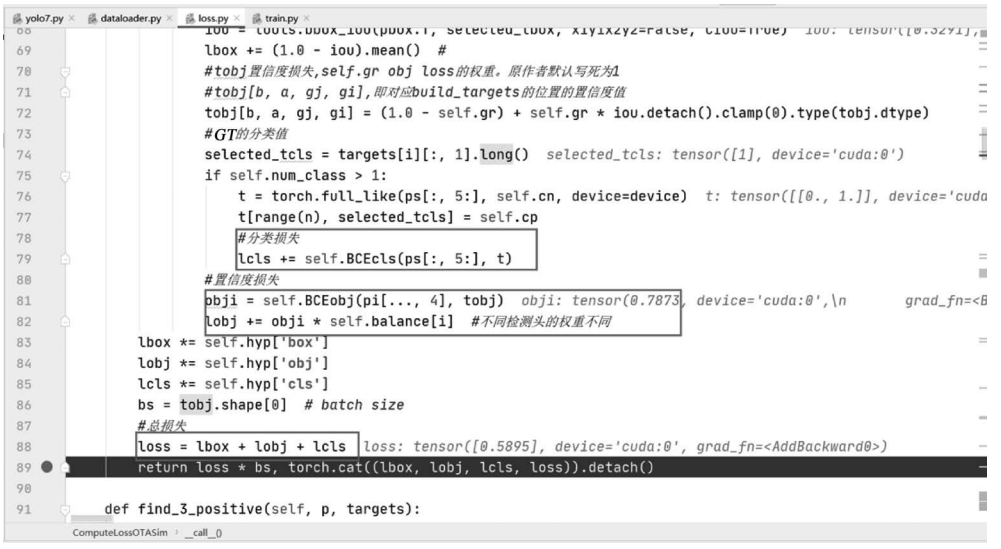



图 5-104　总损失(求和)

## 5.10.5　代码实战预测推理

YOLOv7 的预测推理流程与 YOLOv5 一致，即在 YOLOv3 的基础上只变更了解码公式，其流程仍然是先遍历 3 个检测头，接着对每个检测头的输出结果进行置信度、分类概率

的过滤，然后对 3 个检测头中满足条件的结果进行合并，再通过 NMS 去除重复的框，详细代码可参考 YOLOv5 中的解码代码和 YOLOv3 的推理代码。

**注意**：YOLOv7 使用 PyTorch 框架进行了模型的搭建、训练。

**总结**

YOLOv7 在 YOLOv4 的基础上引入了 MP、ELAN、SPPCSPC 模块，保护 FPN、PAN、3 个检测头的预测，同时在损失函数方面使用了 OTASim 进行更精细化的正样本提取。

**练习**

运行并调试本节代码，理解算法的设计与代码的结合，重点梳理本算法的实现方法。

## 5.11 数据增强

### 5.11.1 数据增强的作用

数据增强是一种基于原有数据，通过一些技术手段生成新数据的方法，通过数据增强可以提高模型的泛化能力，缓解过拟合，提高模型的稳健性。

目标检测中常见的数据增强有旋转、翻转、HSV 等，而在 YOLOv4、YOLOv5 中使用 Mosic 数据增强提高了模型的精确率。

本节重点介绍 CutOut、MixUp、Mosic、随机复制 label 等增强手段，其他数据增强如图 5-105 所示。

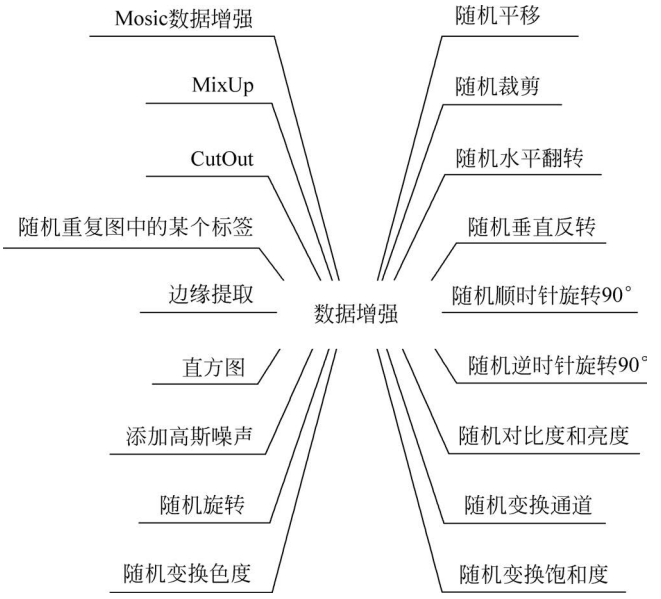



图 5-105 数据增强相关方法

## 5.11.2　代码实现 CutOut 数据增强

CutOut 是在 2017 年提出的一种数据增强方法，即在训练时随机裁剪掉图像的一部分，起到类似 DropOut 正则化的效果，在论文 *Improved Regularization of Convolutional Neural Networks with CutOut* 中表明在原有数据的基础上精度均有提高，如图 5-106 所示。

| Model | STL10 | STL10+ |
|---|---|---|
| WideResNet | 23.48±0.68 | 14.21±0.29 |
| WideResNet+CutOut | **20.77±0.38** | **12.74±0.23** |

图 5-106　CutOut 论文效果

原论文表明，增加 CutOut 数据增强在 STL10 上面精度提高了 0.38。CutOut 的代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V5_Detected/data/enhancement.py
def CutOut(img, gt_boxes, amount=0):
    '''CutOut 数据增强
    img: image
    gt_boxes: 格式[[x1 y1 x2 y2,obj],...]
    amount: 概率
    '''
    out = img.copy()
   #随机选择 CutOut 区域
    ran_select = [random.randint(0, int(len(gt_boxes) - 1 * amount)) for i in
range(int(len(gt_boxes) - 1 * amount))]
    #根据区域进行操作
    for i in ran_select:
    #选择哪个 GT BOX 进行 CutOut
        box = gt_boxes[i]
        x1 = int(box[0])
        y1 = int(box[1])
        x2 = int(box[2])
        y2 = int(box[3])
    #在原有 GT BOX 的基础上裁一定的 BOX
        mask_w = int((x2 - x1) * 0.5)
        mask_h = int((y2 - y1) * 0.5)
        mask_x1 = random.randint(x1, x2 - mask_w)
        mask_y1 = random.randint(y1, y2 - mask_h)
        mask_x2 = mask_x1 + mask_w
        mask_y2 = mask_y1 + mask_h
        #绘框
        cv2.rectangle(out, (mask_x1, mask_y1), (mask_x2, mask_y2), (0, 0, 0),
thickness=-1)
        #位置 CutOut
        gt_boxes[i][0:4] = [mask_x1, mask_y1, mask_x2, mask_y2]
    return out, gt_boxes
```

传入图片和 BOX 信息调用后的效果如图 5-107 所示。

图 5-107　CutOut 示意效果图

### 5.11.3　代码实现 MixUp 数据增强

MixUp 是在论文 *MixUp*：*BEYOND EMPIRICAL RISK MINIMIZATION* 中提出的，其实际上是将两张图片按一定透明度进行叠加的操作。在论文中在 ERM 数据集中使用 VGG-11，使用 MixUp 数据增强对于分类网络错误率约降低了 0.1，如图 5-108 所示。

| Model | Method | Validation set | Test set |
|---|---|---|---|
| LeNet | ERM | **9.8** | **10.3** |
| | MixUP($\alpha$=0.1) | 10.1 | 10.8 |
| | MixUP($\alpha$=0.2) | 10.2 | 11.3 |
| VGG-11 | ERM | 5.0 | 4.6 |
| | MixUP($\alpha$=0.1) | 4.0 | 3.8 |
| | MixUP($\alpha$=0.2) | **3.9** | **3.4** |

图 5-108　MixUp 论文效果

MixUp 的实现代码如下：

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V5_Detected/data/enhancement.py

def mixup(im, labels, im2, labels2):
    #传入两张图的 image 和 label 信息
    r = np.random.beta(32.0, 32.0) #mixup ratio, alpha 和 beta=32.0
    #resize 到相同的大小
    if im.shape[0] > im2.shape[0]:
        im2 = cv2.resize(im2, (im.shape[1], im.shape[0]))
    else:
        im = cv2.resize(im, (im2.shape[1], im2.shape[0]))
    #两张图按一定比例进行融合
    im = (im *r + im2 *(1 - r)).astype(np.uint8)
```

```
    #对两个 BOX 信息进行融合
    if len(labels) != 0 and len(labels2) != 0:
        labels = np.concatenate((labels, labels2), 0)
    elif len(labels) == 0:
        labels = labels2
    elif len(labels2) == 0:
        labels = labels
    return im, labels
```

调用运行代码后其效果如图 5-109 所示。

图 5-109　MixUp 示意效果图

## 5.11.4　代码实现随机复制 Label 数据增强

在完成某些任务(例如缺陷检测)时,由于某些类别的缺陷数量较少,所以可以通过复制标注 BOX 实现特定目标分类的重采样,从而提高网络的稳健性,代码如下:

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V5_Detected/data/enhancement.py
def replicate(im, labels, label_index=0, repetitions=1):
    """标签复制"""
    #得到当前图像的宽和高
    h, w = im.shape[:2]
    #得到 BOX 的位置
    boxes = labels[:, 0:4].astype("int")
    x1, y1, x2, y2 = boxes.T
    #计算所处坐标的位置
    s = ((x2 - x1) + (y2 - y1)) / 2
    for i in s.argsort()[:round(s.size * 0.5)]:
        #如果指定的类别不为空
        if labels[i][-1] != None:
            #判断当前的标签是否与指定的类别一致
            if labels[i][-1] == label_index:
                #重复标签的次数
```

```
            for x in range(repetitions):
                #得到 4 个坐标点
                x1b, y1b, x2b, y2b = boxes[i]
                #得到 BOX 的高、宽
                bh, bw = y2b - y1b, x2b - x1b
                #随机原位置偏移位置
                yc, xc = int(random.uniform(0, h - bh)), int(random.uniform(0, w - bw))
                x1a, y1a, x2a, y2a = [xc, yc, xc + bw, yc + bh]
                #从当前图像中切图并改变透明度,赋给新指定的区域
                im[y1a:y2a, x1a:x2a] = im[y1b:y2b, x1b:x2b]*rand()
                #保存 BOX 信息
                labels = np.append(labels, [[x1a, y1a, x2a, y2a, labels[i, -1]]],
axis=0)
    return im, labels
```

调用运行后红色为原目标、绿色为重复目标,如图 5-110 所示,实现目标框增加两次。

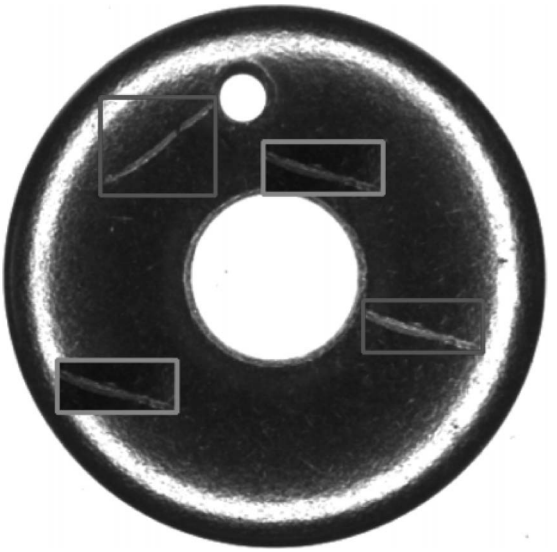


图 5-110 Replicate 数据增强(见彩插)

## 5.11.5 代码实现 Mosic 数据增强

Mosic 数据增强在 YOLOv4、YOLOv5、YOLOv7 中均有使用,使用 Mosic 数据增强可以涨点,其实现思路是将经过翻转、随机裁剪、翻转、色域变换等其他数据增强的 4 张图合并成 1 张新的图,代码如下:

```
#第 5 章/ObjectDetection/TensorFlow_Yolo_V5_Detected/data/enhancement.py
def getList_img_box(rnd_all_lines: list, num: int = 4):
    """
    获得随机的图片和地址
    :param rnd_all_lines: 所有训练的 lines 中的地址,包括图片和位置
    :param num:
```

```
    :return: 返回对应 lines 的下标
    """
    idx = random.sample(range(len(rnd_all_lines)), num)
    return idx


def rand():
    np.random.seed(1000)
    return np.random.uniform()

#合并 4 张图片
def mosic_join_img(rnd_all_lines: list, output_size=None, scale_range=None):
    """Mosic 合并 4 张图片"""
    if output_size is None:
        output_size = [1024, 1024] #设定图像尺寸
    if scale_range is None:
        scale_range = [0.5, 0.5]
    #新建 1 个为 0 的图像
    output_img = np.zeros([output_size[0], output_size[1], 3], dtype=np.uint8)
    #图像缩小的比例
    scale_x = scale_range[0] + random.random() *(scale_range[1] - scale_range[0])
    scale_y = scale_range[0] + random.random() *(scale_range[1] - scale_range[0])
    #贴的图像大小
    point_x = int(scale_x *output_size[1])
    point_y = int(scale_y *output_size[0])
    new_bbox = []
    #从所有的 lines 中获取随机的 4 张图片进行 Mosic
    idx = getList_img_box(rnd_all_lines, 4)
    for i, ix in enumerate(idx):
        #对选择的 4 张照片进行处理,得到 boxes 信息
        line = rnd_all_lines[ix].split()
        img_path = line[0]
        img_boxes = np.array([np.array(list(map(int, box1.split(',')))) for box1
in line[1:]])

        #读图片
        img = cv2.imread(img_path)
        #对图像进行加工操作,调用已有函数
        #翻转
        if rand() < 0.5:
            img, img_boxes = random_horizontal_flip(img, img_boxes)
        #色域变换
        if rand() < 0.3:
            img, img_boxes = random_hue(img, img_boxes)
        #CutOut
        if rand() < 0.1:
            img, img_boxes = CutOut(img, img_boxes)
        #随机裁剪
        if rand() < 0.2:
            img, img_boxes = random_crop(img, img_boxes)

        #左上角图片的处理
```

```
        if i == 0:
            #用 letter_box 进行替换,得到指定大小的图片和 boxes
            img2, img_boxes = letterbox_image(Image.fromarray(np.uint8(img.
copy())), (point_x, point_y),
                                              np.array(img_boxes.copy()))
            #更新到要保存的图像中
            output_img[:point_x, :point_y, :] = img2
            #处理 bbox
            for bbox in img_boxes:
                xmin = bbox[0]                    #第 1 张图的位置不变
                ymin = bbox[1]
                xmax = bbox[2]
                ymax = bbox[3]
                new_bbox.append([xmin, ymin, xmax, ymax, bbox[-1]])
        elif i == 1:
            #第 2 张图的 x 轴的位置发生了变化
            img2, img_boxes = letterbox_image(
                Image.fromarray(np.uint8(img.copy())), (output_size[1] - point_x,
point_y),
                np.array(img_boxes.copy())
            )
            #更新到要保存的图像中
            output_img[:point_y, point_x:output_size[1], :] = img2
            for bbox in img_boxes:
                xmin = point_x + bbox[0]          #第 2 张图的 x 发生了变化
                ymin = bbox[1]
                xmax = point_x + bbox[2]
                ymax = bbox[3]
                new_bbox.append([xmin, ymin, xmax, ymax, bbox[-1]])
        elif i == 2:
            #第 3 张图的 x 轴的位置发生了变化
            img2, img_boxes = letterbox_image(
                Image.fromarray(np.uint8(img.copy())), (point_x, output_size[0] -
point_y),
                np.array(img_boxes.copy())
            )

            output_img[point_y:output_size[0], :point_x, :] = img2
            #x 不变,y 轴变了
            for bbox in img_boxes:
                xmin = bbox[0]                    #第 3 张图的 y 发生了变化
                ymin = point_y + bbox[1]
                xmax = bbox[2]
                ymax = point_y + bbox[3]
                new_bbox.append([xmin, ymin, xmax, ymax, bbox[-1]])
        elif i == 3:
            img2, img_boxes = letterbox_image(
                Image.fromarray(np.uint8(img.copy())), (output_size[1] - point_x,
output_size[0] - point_y),
                np.array(img_boxes.copy())
            )
            output_img[point_y:output_size[0], point_x:output_size[1], :] = img2
```

```
            for bbox in img_boxes:
                xmin = point_x + bbox[0]        #第 4 张图的 x、y 方向同时变化
                ymin = point_y + bbox[1]
                xmax = point_x + bbox[2]
                ymax = point_y + bbox[3]
                new_bbox.append([xmin, ymin, xmax, ymax, bbox[-1]])
        return output_img, np.array(new_bbox, dtype=np.int)
```

代码实现思路是先通过 np.zeros()得到一张 1024×1024 全黑的图片，然后根据传入的 lines 信息随机获得第 idx 张图片，然后解析出 img_boxes。根据 rand()随机值调用翻转、色域变换、CutOut、随机裁剪的数据增强函数，以 1024×1024 的中心点为坐标，在第 1 个位置将图像更新上去，即通过代码 output_img[:point_x,:point_y,:]=img2 实现，同时计算 bbox 的位置变化，调用该代码实现的效果如图 5-111 所示。

图 5-111　Mosic 数据增强

更多数据增强的代码，可参考随书代码。

**总结**

数据增强不仅能提高算法的稳健性，同时也能有效地缓解过拟合，是模型训练调优的重要组成方法。

**练习**

动手实现 Mosic 和标签随机复制的数据增强代码。