# 第5章

# 目标检测

目标检测(Object Detection)的任务是找出图像中所有感兴趣的目标(物体),确定它们的类别和位置,是计算机视觉领域的核心问题之一。由于各类物体有不同的外观、形状和姿态,加上成像时光照、遮挡等因素的干扰,目标检测一直是计算机视觉领域最具有挑战性的问题。

## 5.1 目标检测概述

### 5.1.1 计算机视觉任务

计算机视觉中关于图像识别有四大类任务,如图 5-1 所示。

(1) 分类(Classification):解决"是什么?"的问题,即给定一幅图片或一段视频判断里面包含什么类别的目标。

(2) 定位(Location):解决"在哪里?"的问题,即定位出这个目标的位置。

(3) 检测(Detection):解决"在哪里? 是什么?"的问题,即定位出这个目标的位置并且知道目标物是什么。

(4) 分割(Segmentation):分为实例(Instance)分割和场景(Scene)分割,解决"每一个像素属于哪个目标物或场景"的问题。


图 5-1 图像识别过程

因此,目标检测是一个分类问题、回归问题的叠加。

### 5.1.2 目标检测的核心问题

目标检测的核心问题主要有4类，分别说明如下。

(1) 分类问题：图片(或某个区域)中的图像属于哪个类别。

(2) 定位问题：目标可能出现在图像的任何位置。

(3) 大小问题：目标有各种不同的大小。

(4) 形状问题：目标可能有各种不同的形状。

### 5.1.3 目标检测算法分类

基于深度学习的目标检测算法主要分为两类：Two stage 和 One stage。

**1. Tow Stage**

Tow Stage(先进行区域生成)区域称为 Region Proposal(简称 RP，一个有可能包含待检物体的预选框)，再通过卷积神经网络进行样本分类。常见的 tow stage 目标检测算法有 R-CNN、SPP-Net、Fast R-CNN、Faster R-CNN 和 R-FCN 等。

**2. One Stage**

One Stage 直接在网络中提取特征来预测物体分类和位置。常见的 One Stage 目标检测算法有 OverFeat、YOLOv1、YOLOv2、YOLOv3、SSD 和 RetinaNet 等。

### 5.1.4 目标检测模型

深度学习目标检测方法分为 Anchor-Based(锚框法)和 Anchor-Free(无锚框)两大类，根据有无区域提案阶段划分为双阶段模型和单阶段模型。

- 双阶段模型：区域检测模型将目标检测任务分为区域提案生成、特征提取和分类预测三个阶段。在区域提案生成阶段，检测模型利用搜索算法，如选择性搜索(Selective Search，SS)、EdgeBoxes、区域提案网络(Region Proposal Network，RPN)等在图像中搜寻可能包含物体的区域。在特征提取阶段，模型利用深度卷积网络提取区域提案中的目标特征。在分类预测阶段，模型从预定义的类别标签对区域提案进行分类和边框信息预测。
- 单阶段模型：单阶段检测模型联合区域提案和分类预测，输入整幅图像到卷积神经网络中提取特征，最后直接输出目标类别和边框位置信息。这类代表性的方法有 YOLO、SSD 和 CenterNet 等。

### 5.1.5 目标检测的研究方向

目标检测方法可分为检测部件、数据增强、优化方法和学习策略4个方面。其中检测部件包含基准模型和基准网络；数据增强包含几何变换、光学变换等；优化方法包括特征图、上下文模型、边框优化、区域提案方法、类别不平衡和训练策略6个方面；学习策略涵盖监督学习、弱监督学习和无监督学习。

### 5.1.6 边界框

图片分类一般默认图片中只有一个主体，而目标检测任务中，图片通常含有多个主体。在目标检测中，不仅想知道它们的类别，还想得到它们在图像中的具体位置。

在目标检测中，通常使用边界框(Bounding Box)来描述对象的空间位置。边界框是矩形

的，表示方式有三种：

- （左上 $x$，左上 $y$，右下 $x$，右下 $y$）。
- （左上 $x$，左上 $y$，宽，高）。
- （中心 $x$，中心 $y$，宽，高）。

目标检测数据集的常见表示：每一行表示一个物体，对于每一个物体而言，用“图片文件名，物体类别，边缘框”表示，由于边缘框用 4 个数值表示，因此对于每一行的任意一个物体而言，需要用 6 个数值表示。

**提示**：COCO 目标数据集包含 80 个物体类别、超过 33 万幅图像和 150 万个标注实例。

**【例 5-1】** 边界框实例演示。

具体实现步骤如下。

（1）从实例加载的图片（见图 5-2）可以看到，图像左边是一只狗，右边是一只猫，它们是这幅图像中的两个主要目标。

```
import torch
from d2l import torch as d2l
#加载图片
d2l.set_figsize()
img = d2l.plt.imread('catdog.jpg')
d2l.plt.imshow(img);
d2l.plt.show()
```

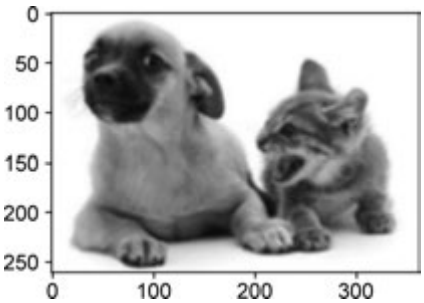


图 5-2 加载的图片

（2）将根据坐标信息定义图像中狗和猫的边界框。图像中坐标的原点是图像的左上角，向右的方向为 $x$ 轴的正方向，向下的方向为 $y$ 轴的正方向。

```
#bbox 是边界框的英文缩写
dog_bbox, cat_bbox = [16.0, 11.0, 190.0, 216.0], [213.0, 59.0, 332.0, 200.0]
```

（3）可以将边界框在图中画出，以检查其是否准确。画之前，定义一个辅助函数 bbox_to_rect，它将边界框表示成 matplotlib 的边界框格式。

```
def bbox_to_rect(bbox, color):
    #将边界框(左上 x,左上 y,右下 x,右下 y)格式转换成 matplotlib 格式
    #((左上 x,左上 y),宽,高)
    return d2l.plt.Rectangle(
        xy = (bbox[0], bbox[1]), width = bbox[2] - bbox[0], height = bbox[3] - bbox[1],
        fill = False, edgecolor = color, linewidth = 2)
```

（4）在图像上添加边界框之后，可以看到两个物体的主要轮廓基本上在两个框内，如图 5-3 所示。

```
fig = d2l.plt.imshow(img)
fig.axes.add_patch(bbox_to_rect(dog_bbox, 'blue'))
```

```
fig.axes.add_patch(bbox_to_rect(cat_bbox, 'red'));
d2l.plt.show()      % 显示如图 5-3 所示的效果
def box_corner_to_center(boxes):
    """从(左上,右下)转换到(中间,宽度,高度)"""
    x1, y1, x2, y2 = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
```

还可以在两种常用的边界框表示(中间,宽度,高度)和(左上,右下)坐标之间进行转换。

```
cx = (x1 + x2) / 2
 cy = (y1 + y2) / 2
 w = x2 - x1
 h = y2 - y1
 boxes = torch.stack((cx, cy, w, h), axis = -1)
 return boxes
def box_center_to_corner(boxes):
    """从(中间,宽度,高度)转换到(左上,右下)"""
    cx, cy, w, h = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    x1 = cx - 0.5 * w
    y1 = cy - 0.5 * h
    x2 = cx + 0.5 * w
    y2 = cy + 0.5 * h
    boxes = torch.stack((x1, y1, x2, y2), axis = -1)
    return boxes
```

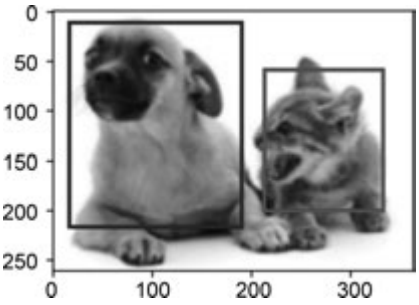


图 5-3 添加的两个框图

## 5.2 锚框

目标检测算法通常会在输入图像中采样大量的区域,然后判断这些区域中是否包含感兴趣的目标,并调整区域边缘,从而更准确地预测目标的真实边界框(Ground-truth Bounding Box)。不同的模型使用的区域采样方法可能不同。此处介绍其中的一种方法:它以每个像素为中心生成多个大小和宽高比(Aspect Tatio)不同的边界框。这些边界框被称为锚框(Anchor Box)。

下面将设计一个基于锚框的目标检测模型。

修改输出精度,以获得更简洁的输出。

```
import torch
from d2l import torch as d2l
torch.set_printoptions(2)      #精简输出精度
```

### 5.2.1 生成多个锚框

设输入图像的高度为 $h$,宽度为 $w$,以图像的每个像素为中心生成不同形状的锚框:缩放比为 $s\in(0,1]$,宽度比为 $r>0$。那么锚框的宽度和高度分别是 $ws\sqrt{r}$ 和 $hs\sqrt{r}$。需要注意,当

中心位置给定时，已知宽和高的锚框是确定的。

要生成多个不同形状的锚框，先设置许多缩放比(scale)取值 $s_1,s_2,\cdots,s_n$ 和许多宽高比(aspect ration)取值 $r_1,r_2,\cdots,r_m$。当以每个像素为中心，使用这些比例和长宽比的所有组合时，输入图像总共有 $whnm$ 个锚框。尽管这些锚框可能会覆盖所有真实边界框，但计算复杂性很容易过高。在实践中，只考虑包含 $s_1$ 或 $r_1$ 的组合：

$$(s_1,r_1),(s_1,r_2),\cdots,(s_1,r_m),(s_2,r_1),(s_3,r_1),\cdots,(s_m,r_1)$$

也就是说，以同一像素为中心的锚框的数量是 $n+m-1$。对于整幅输入图像，将共生成 $wh(n+m-1)$ 个锚框。

上述生成锚框的方法在下面的 multibox_prior 函数中实现。该函数指定输入图像、尺寸列表和宽高比列表，并返回所有的锚框。

```
def multibox_prior(data, sizes, ratios):
    """生成以每个像素为中心具有不同形状的锚框"""
    in_height, in_width = data.shape[-2:]
    device, num_sizes, num_ratios = data.device, len(sizes), len(ratios)
    boxes_per_pixel = (num_sizes + num_ratios - 1)
    size_tensor = torch.tensor(sizes, device=device)
    ratio_tensor = torch.tensor(ratios, device=device)

    #为了将锚点移动到像素的中心，需要设置偏移量
    #一个像素的高为1、宽为1，即选择偏移的中心为0.5
    offset_h, offset_w = 0.5, 0.5
    steps_h = 1.0 / in_height                  #在y轴上缩放步长
    steps_w = 1.0 / in_width                   #在x轴上缩放步长

    #生成锚框的所有中心点
    center_h = (torch.arange(in_height, device=device) + offset_h) * steps_h
    center_w = (torch.arange(in_width, device=device) + offset_w) * steps_w
    shift_y, shift_x = torch.meshgrid(center_h, center_w, indexing='ij')
    shift_y, shift_x = shift_y.reshape(-1), shift_x.reshape(-1)

    #生成boxes_per_pixel个高和宽
    #之后用于创建锚框的四角坐标(xmin,xmax,ymin,ymax)
    w = torch.cat((size_tensor * torch.sqrt(ratio_tensor[0]),
                   sizes[0] * torch.sqrt(ratio_tensor[1:])))\
                    * in_height / in_width     #处理矩形输入
    h = torch.cat((size_tensor / torch.sqrt(ratio_tensor[0]),
                   sizes[0] / torch.sqrt(ratio_tensor[1:])))
    #除以2来获得半高和半宽
    anchor_manipulations = torch.stack((-w, -h, w, h)).T.repeat(
                                        in_height * in_width, 1) / 2

    #每个中心点都将有boxes_per_pixel个锚框
    #所以生成含所有锚框中心的网格，重复了boxes_per_pixel次
    out_grid = torch.stack([shift_x, shift_y, shift_x, shift_y],
                dim=1).repeat_interleave(boxes_per_pixel, dim=0)
    output = out_grid + anchor_manipulations
    return output.unsqueeze(0)

#可以查看返回的锚框变量Y的形状(批量大小，锚框的数量，4)
img = d2l.plt.imread('catdog.jpg')
h, w = img.shape[:2]

print(h, w)
```

```
X = torch.rand(size=(1, 3, h, w))
Y = multibox_prior(X, sizes=[0.75, 0.5, 0.25], ratios=[1, 2, 0.5])
Y.shape
```

输出为：

```
261 363
```

将锚框变量$Y$的形状更改为(图像高度,图像宽度,以同一像素为中心的锚框的数量4)后,可以获得以指定像素的位置为中心的所有锚框。在接下来的内容中,访问以(150,150)为中心的第一个锚框。它有4个元素：锚框左上角的$(x_1, y_1)$轴坐标和右下角的$(x_2, y_2)$轴坐标。将两个轴的坐标各分别除以图像的宽度和高度后,所得的值介于0和1之间。

```
boxes = Y.reshape(h, w, 5, 4)
boxes[150,150, 0, :]
```

为了显示在图像中以某个像素为中心的所有锚框,定义了show_bboxes()函数在图像上绘制多个边界框。

```
def show_bboxes(axes, bboxes, labels=None, colors=None):
    """显示所有边界框"""
    def _make_list(obj, default_values=None):
        if obj is None:
            obj = default_values
        elif not isinstance(obj, (list, tuple)):
            obj = [obj]
        return obj

    labels = _make_list(labels)
    colors = _make_list(colors, ['b', 'g', 'r', 'm', 'c'])
    for i, bbox in enumerate(bboxes):
        color = colors[i % len(colors)]
        rect = d2l.bbox_to_rect(bbox.detach().numpy(), color)
        axes.add_patch(rect)
        if labels and len(labels) > i:
            text_color = 'k' if color == 'w' else 'w'
            axes.text(rect.xy[0], rect.xy[1], labels[i],
                      va='center', ha='center', fontsize=9, color=text_color,
                      bbox=dict(facecolor=color, lw=0))
```

正如所看到的,变量boxes中$x$轴和$y$轴的坐标值已分别除以图像的宽度和高度。绘制锚框时,需要恢复它们原始的坐标值。因此,在下面定义了变量bbox_scale。现在,可以绘制出图像中所有以(150,150)为中心的锚框了。如图5-4所示,缩放比为0.75且宽高比为1的蓝色锚框很好地围绕着图像中的狗。

```
d2l.set_figsize()
bbox_scale = torch.tensor((w, h, w, h))
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, boxes[150, 150, :, :] * bbox_scale,
            ['s=0.75, r=1', 's=0.5, r=1', 's=0.25, r=1', 's=0.75, r=2',
             's=0.75, r=0.5'])
d2l.plt.show()
```

### 5.2.2 交并比

在5.2.1节提到的某个锚框“较好地”覆盖了图像中的狗。如果已知目标的真实边界框,

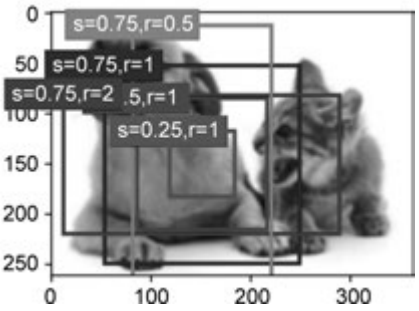


图 5-4　为图绘制锚框

那么此处的“好”该如何量化呢？直观地说，可以衡量锚框和真实边界框之间的相似性。杰卡德(Jaccard)系数可以衡量两组之间的相似性。给定集合 $A$ 和 $B$，杰卡德系数是它们的交集的大小除以它们的并集的大小：

$$J(A,B)=\frac{|A \cap B|}{|A \cup B|}$$

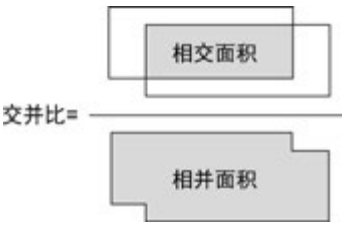


图 5-5　交并比效果图

实质上，可以将任何边界框的像素区域视为一组像素。通过这种方式，可以通过其像素的杰卡德系数来测量两个边界框的相似性。对于两个边界框，通常将它们的杰卡德系数称为交并比(Intersection over Union，IoU)，即两个边界框相交面积与相并面积之比，如图 5-5 所示。交并比的取值范围为 0～1：0 表示两个边界框无重合像素，1 表示两个边界框完全重合。

接着，将使用交并比来衡量锚框和真实边界框之间，以及不同锚框之间的相似度。给定两个锚框或边界框的列表，以下 box_iou()函数将在这两个列表中计算它们成对的交并比。

```
def box_iou(boxes1, boxes2):
    """计算两个锚框或边界框列表中成对的交并比"""
    box_area = lambda boxes: ((boxes[:, 2] - boxes[:, 0]) *
                              (boxes[:, 3] - boxes[:, 1]))
    #boxes1,boxes2,areas1,areas2 的形状:
    #boxes1:(boxes1 的数量,4),
    #boxes2:(boxes2 的数量,4),
    #areas1:(boxes1 的数量,),
    #areas2:(boxes2 的数量,)
    areas1 = box_area(boxes1)
    areas2 = box_area(boxes2)
    #inter_upperlefts,inter_lowerrights,inters 的形状:(boxes1 的数量,boxes2 的数量,2)
    inter_upperlefts = torch.max(boxes1[:, None, :2], boxes2[:, :2])
    inter_lowerrights = torch.min(boxes1[:, None, 2:], boxes2[:, 2:])
    inters = (inter_lowerrights - inter_upperlefts).clamp(min=0)
    #inter_areasandunion_areas 的形状:(boxes1 的数量,boxes2 的数量)
    inter_areas = inters[:, :, 0] * inters[:, :, 1]
    union_areas = areas1[:, None] + areas2 - inter_areas
    return inter_areas / union_areas
```

### 5.2.3　标注锚框

在训练集中，将每个锚框视为一个训练样本。为了训练目标检测模型，需要每个锚框的类别(class)和偏移量(offset)标签，其中前者是与锚框相关的对象的类别，后者是真实边界框相

对于锚框的偏移量。在预测时，为每幅图像生成多个锚框，预测所有锚框的类别和偏移量，根据预测的偏移量调整它们的位置以获得预测的边界框，最后只输出符合特定条件的预测边界框。

我们知道，目标检测训练集带有“真实边界框”的位置及其包围物体类别的标签。要标记任何生成的锚框，可以参考分配到的最接近此锚框的真实边界框的位置和类别标签。

**1. 将真实边界框分配给锚框**

在训练集中，将每个锚框视为一个训练样本。为了训练目标检测模型，需要每个锚框的类别(class)和偏移量(offset)标签。预测时，为每幅图像生成多个锚框，预测所有锚框的类别和偏移量，最后只输出符合特定条件的预测边界框。

定义 assign_anchor_to_bbox()函数，将真实边界框分配给锚框：

```
def assign_anchor_to_bbox(ground_truth, anchors, device, iou_threshold = 0.5):
    """将最接近的真实边界框分配给锚框
   ground_truth:边界框;anchors:锚框
   iou_threshold = 0.5 表示某个锚框,任何其他锚框小于 0.5,即删掉"""
    num_anchors, num_gt_boxes = anchors.shape[0], ground_truth.shape[0]
    #计算所有锚框和边界框的 IoU。位于第 i 行和第 j 列的元素 x_ij 是锚框 i 和真实边界框 j 的 IoU
    jaccard = box_iou(anchors, ground_truth)
    #对于每个锚框,分配的真实边界框的张量
    anchors_bbox_map = torch.full((num_anchors,), -1, dtype = torch.long, device = device)
    #根据阈值决定是否分配真实边界框
    max_ious, indices = torch.max(jaccard, dim = 1)
    anc_i = torch.nonzero(max_ious >= 0.5).reshape(-1)
    box_j = indices[max_ious >= 0.5]
    anchors_bbox_map[anc_i] = box_j
    col_discard = torch.full((num_anchors,), -1)
    row_discard = torch.full((num_gt_boxes,), -1)

    #每次找出最大的 IoU
    for _ in range(num_gt_boxes):
        max_idx = torch.argmax(jaccard)
        box_idx = (max_idx % num_gt_boxes).long()
        anc_idx = (max_idx / num_gt_boxes).long()
        anchors_bbox_map[anc_idx] = box_idx
        #删除行和列
        jaccard[:, box_idx] = col_discard
        jaccard[anc_idx, :] = row_discard
    return anchors_bbox_map
```

**2. 标记偏移量**

假设一个锚框 $A$ 被分配了一个真实边界框 $B$。那么，锚框 $A$ 的类别将被标记为与 $B$ 相同。另外，锚框 $A$ 的偏移量将根据 $B$ 和 $A$ 中心坐标的相对位置以及这两个框的相对位置进行标记。给定框 $A$ 和 $B$，中心坐标分别为 $(x_a, y_a)$ 和 $(x_b, y_b)$，宽度分别为 $w_a$ 和 $w_b$，长度分别为 $h_a$ 和 $h_b$。可以将 $A$ 的偏移量标记为

$$\left(\frac{\frac{x_b - x_a}{w_a} - \mu_x}{\sigma_x}, \frac{\frac{y_b - y_a}{h_a} - \mu_y}{\sigma_y}, \frac{\log\frac{w_b}{w_a} - \mu_w}{\sigma_w}, \frac{\log\frac{h_b}{h_a} - \mu_h}{\sigma_h}\right)$$

其中，常量的默认值为 $\mu_x = \mu_y = \mu_h = 0$，$\sigma_x = \sigma_y = 0.1$，$\sigma_w = \sigma_h = 0.2$，这样做使数值分得比较开，均值方差都比较好进行预测。

在下面的 offset_boxes()函数中可以实现这种转换：

```
def offset_boxes(anchors, assigned_bb, eps = 1e - 6):
    """对锚框偏移量的转换"""
    c_anc = d2l.box_corner_to_center(anchors)
    c_assigned_bb = d2l.box_corner_to_center(assigned_bb)
    offset_xy = 10 * (c_assigned_bb[:, :2] - c_anc[:, :2]) / c_anc[:, 2:]
    offset_wh = 5 * torch.log(eps + c_assigned_bb[:, 2:] / c_anc[:, 2:])
    offset = torch.cat([offset_xy, offset_wh], axis = 1)
    return offset
```

**3. 标记类别和偏移量**

如果一个锚框没有被分配真实边界框，只需将锚框的类别标记为“背景”(background)。背景类别的锚框通常被称为“负类”锚框，其余的被称为“正类”锚框。以下使用 multibox_target()函数标记锚框的类别和偏移量(anchors 参数)。此函数将背景类别的索引设置为零，然后将新类别的整数索引递增 1：

```
def multibox_target(anchors, labels):
    """使用真实边界框标记锚框"""
    batch_size, anchors = labels.shape[0], anchors.squeeze(0)
    batch_offset, batch_mask, batch_class_labels = [], [], []
    device, num_anchors = anchors.device, anchors.shape[0]
    for i in range(batch_size):
        label = labels[i, :, :]
        anchors_bbox_map = assign_anchor_to_bbox(
            label[:, 1:], anchors, device)
        bbox_mask = ((anchors_bbox_map >= 0).float().unsqueeze( - 1)).repeat(1, 4)
        # 将类标签和分配的边界框坐标初始化为零
        class_labels = torch.zeros(num_anchors, dtype = torch.long,
                                   device = device)
        assigned_bb = torch.zeros((num_anchors, 4), dtype = torch.float32,
                                  device = device)
        # 使用真实边界框来标记锚框的类别
        # 如果一个锚框没有被分配，标记其为背景(值为零)
        indices_true = torch.nonzero(anchors_bbox_map >= 0)
        bb_idx = anchors_bbox_map[indices_true]
        class_labels[indices_true] = label[bb_idx, 0].long() + 1
        assigned_bb[indices_true] = label[bb_idx, 1:]
        # 偏移量转换
        offset = offset_boxes(anchors, assigned_bb) * bbox_mask
        batch_offset.append(offset.reshape( - 1))
        batch_mask.append(bbox_mask.reshape( - 1))
        batch_class_labels.append(class_labels)
    bbox_offset = torch.stack(batch_offset)
    bbox_mask = torch.stack(batch_mask)
    class_labels = torch.stack(batch_class_labels)
    return (bbox_offset, bbox_mask, class_labels)
    # 锚框对应真实边界框的偏移量 offset，bbox_mask = 0 表示锚框是背景框，不需要预测类别
    # bbox_mask = 1 表示锚框对应一个真实边缘框。class_labels 表示锚框对应的类别
```

下面通过一个具体的例子来说明锚框标签。已知为加载图像中的狗和猫定义了真实边界框，其中第一个元素是类别(0 代表狗，1 代表猫)，其余 4 个元素是左上角和右下角的$(x,y)$轴坐标(范围介于 0 和 1 之间)。还构建了 5 个锚框，用左上角和右下角的坐标进行标记 $A_0$，$A_1$，…，$A_4$(索引从 0 开始)。最后在图像中绘制这些真实边界和锚框，如图 5-6 所示。

```
ground_truth = torch.tensor([[0, 0.1, 0.08, 0.52, 0.92],
                             [1, 0.55, 0.2, 0.9, 0.88]])
```

```
anchors = torch.tensor([[0, 0.1, 0.2, 0.3], [0.15, 0.2, 0.4, 0.4],
                        [0.63, 0.05, 0.88, 0.98], [0.66, 0.45, 0.8, 0.8],
                        [0.57, 0.3, 0.92, 0.9]])

fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, ground_truth[:, 1:] * bbox_scale, ['dog', 'cat'], 'k')
show_bboxes(fig.axes, anchors * bbox_scale, ['0', '1', '2', '3', '4']);
```

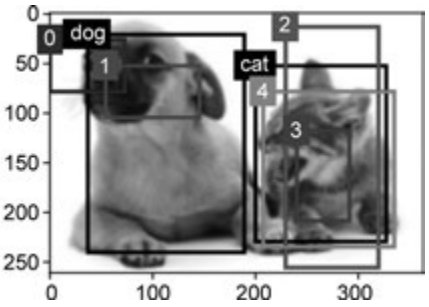


图 5-6 绘制真实边界框和锚框

使用上面定义的 multibox_target()函数，我们可以根据狗和猫的真实边界框标注这些锚框的分类和偏移量。在这个例子中，背景、狗和猫的类索引分别为 0、1 和 2。下面为锚框和真实边界框样本添加一个维度。

```
#unsqueeze 表示增加一个维度
labels = multibox_target(anchors.unsqueeze(dim=0),
                         ground_truth.unsqueeze(dim=0))
```

返回的结果中有三个元素，都是张量格式。第三个元素包含标记的输入锚框的类别。

下面根据图像中的锚框和真实边界框的位置来分析返回的类别标签。首先，在所有的锚框和真实边界框配对中，锚框 $A_4$ 与猫的真实边界框的 IoU 是最大的。因此，$A_4$ 的类别被标记为猫。去除包含 $A_4$ 或猫的真实边界框的配对，在剩下的配对中，锚框 $A_1$ 与猫的真实边界框有最大的 IoU，因此，$A_1$ 的类别被标记为狗。接着，需要遍历剩下的三个未标记的锚框：$A_0$、$A_2$ 和 $A_3$。对于 $A_0$，与其拥有最大 IoU 的真实边界的类别是狗，但 IoU 低于预定义的阈值(0.5)，因此该类别被标记为背景；对于 $A_2$，与其拥有最大 IoU 的真实边界框的类别是猫，IoU 超过阈值，因此该类别被标记为猫；对于 $A_3$，与其拥有最大 IoU 的真实边界框的类别是猫，但值低于阈值，因此该类别被标记为背景。

```
print(labels[2])
```

输出为：

```
tensor([[0, 1, 2, 0, 2]])
```

返回的第二个元素是掩码(mask)变量，形状为(批量大小，锚框数的 4 倍)。掩码变量中的元素与每个锚框的 4 个偏移量一一对应。由于我们不关心对背景的检测，负类的偏移量不应影响目标函数。通过元素乘法，掩码变量中的零将在计算目标函数之前过滤掉负类偏移量。

```
print(labels[1])
```

输出为：

```
tensor([[0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 1., 1.,
         1., 1.]])
```

返回的第一个元素包含为每个锚框标记的 4 个偏移值。注意，负类锚框的偏移量被标记

为零。

```
print(labels[0])
```

输出为：

```
tensor([[ -0.00e+00, -0.00e+00, -0.00e+00, -0.00e+00,  1.40e+00,  1.00e+01,
          2.59e+00,  7.18e+00, -1.20e+00,  2.69e-01,  1.68e+00, -1.57e+00,
         -0.00e+00, -0.00e+00, -0.00e+00, -0.00e+00, -5.71e-01, -1.00e+00,
          4.17e-06,  6.26e-01]])
```

### 5.2.4 使用非极大值抑制预测边界框

在预测时，先为图像生成多个锚框，再为这些锚框一一预测类别和偏移量。一个“预测好的边界框”则根据其中某个带有预测偏移量的锚框而生成。下面自定义编写 offset_inverse() 函数，该函数将锚框和偏移量预测作为输入，并应用逆偏移变换来返回预测的边界框坐标。

```
def offset_inverse(anchors, offset_preds):
    """根据带有预测偏移量的锚框来预测边界框"""
    anc = d2l.box_corner_to_center(anchors)
    pred_bbox_xy = (offset_preds[:, :2] * anc[:, 2:] / 10) + anc[:, :2]
    pred_bbox_wh = torch.exp(offset_preds[:, 2:] / 5) * anc[:, 2:]
    pred_bbox = torch.cat((pred_bbox_xy, pred_bbox_wh), axis=1)
    predicted_bbox = d2l.box_center_to_corner(pred_bbox)
    return predicted_bbox
```

当有许多锚框时，可能会输出许多相似的具有明显重叠的预测边界框，都围绕着同一目标。为了简化输出，可以使用非极大值抑制(Non-Maximum Suppression，NMS)合并属于同一目标的类似的预测边界框。

非极大值抑制的工作原理为：对于一个预测边界框 $B$，目标检测模型会计算每个类别的预测概率。假设最大的预测概率为 $p$，则该概率所对应的类别 $B$ 即为预测的类别。具体来说，将 $p$ 称为预测边界框的置信度(confidence)。在同一幅图像中，所有预测的非背景边界框都按置信度降序排序，以生成列表 $L$。然后通过以下步骤操作排序列表 $L$：

(1) 从 $L$ 中选取置信度最高的预测边界框 $B_1$ 作为基准，然后将所有与 $B_1$ 的 IoU 超过预定阈值 $\varepsilon$ 的非基准预测边界框从 $L$ 中移除。这时，$L$ 保留了置信度最高的预测边界框，去除了与其太过相似的其他预测边界框。简而言之，那些具有非极大值置信度的边界框被抑制了。

(2) 从 $L$ 中选取置信度第二高的预测边界框 $B_2$ 作为又一个基准，然后将所有与 $B_2$ 的 IoU 大于 $\varepsilon$ 的非基准预测边界框从 $L$ 中移除。

(3) 重复上述过程，直到 $L$ 中的所有预测边界框都曾被用作基准。此时，$L$ 中任意一对预测边界框的 IoU 都小于阈值 $\varepsilon$，因此，没有一对边界框过于相似。

(4) 输出列表 $L$ 中的所有预测边界框。

以下 nms()函数按降序对置信度进行排序并返回其索引。

```
def nms(boxes, scores, iou_threshold):
    """对预测边界框的置信度进行排序"""
    B = torch.argsort(scores, dim=-1, descending=True)     #按照 scores 进行置信度排序
    keep = []                                              #保留预测边界框的指标
    while B.numel() > 0:
        i = B[0]
        keep.append(i)
        if B.numel() == 1: break
        iou = box_iou(boxes[i, :].reshape(-1, 4),boxes[B[1:], :].reshape(-1, 4)).reshape
```

```
(-1)
        inds = torch.nonzero(iou <= iou_threshold).reshape(-1)
        B = B[inds + 1]
    return torch.tensor(keep, device = boxes.device)
```

定义以下 multibox_detection()函数将非极大值抑制应用于预测边界框。

```
def multibox_detection(cls_probs, offset_preds, anchors, nms_threshold = 0.5,
                       pos_threshold = 0.009999999):
    """
    #anchor 表示成归一化(xmin, ymin, xmax, ymax)
        cls_prob: 经过 softmax 后得到的各个锚框的预测概率, shape:(bn, 预测总类别数 + 1, 锚框
个数)
        loc_pred: 预测的各个锚框的偏移量, shape:(bn, 锚框个数 * 4)
        anchor: MultiBoxPrior 输出的默认锚框, shape: (1,锚框个数, 4)
        nms_threshold: 非极大值抑制中的阈值
    Returns:
        所有锚框的信息, shape: (bn, 锚框个数, 6)
        每个锚框信息由[class_id, confidence, xmin, ymin, xmax, ymax]表示
        class_id = -1 表示背景或在非极大值抑制中被移除
    """
    """使用非极大值抑制来预测边界框"""
    device, batch_size = cls_probs.device, cls_probs.shape[0]
    anchors = anchors.squeeze(0)
    num_classes, num_anchors = cls_probs.shape[1], cls_probs.shape[2]
    out = []
    for i in range(batch_size):
        cls_prob, offset_pred = cls_probs[i], offset_preds[i].reshape(-1, 4)
        conf, class_id = torch.max(cls_prob[1:], 0)
        predicted_bb = offset_inverse(anchors, offset_pred)
        keep = nms(predicted_bb, conf, nms_threshold)

        #找到所有的 non_keep 索引,并将类设置为背景
        all_idx = torch.arange(num_anchors, dtype = torch.long, device = device)
        combined = torch.cat((keep, all_idx))
        uniques, counts = combined.unique(return_counts = True)
        non_keep = uniques[counts == 1]
        all_id_sorted = torch.cat((keep, non_keep))
        class_id[non_keep] = -1
        class_id = class_id[all_id_sorted]
        conf, predicted_bb = conf[all_id_sorted], predicted_bb[all_id_sorted]
        #pos_threshold 是一个用于非背景预测的阈值
        below_min_idx = (conf < pos_threshold)
        class_id[below_min_idx] = -1
        conf[below_min_idx] = 1 - conf[below_min_idx]
        pred_info = torch.cat((class_id.unsqueeze(1),
                               conf.unsqueeze(1),
                               predicted_bb), dim = 1)
        out.append(pred_info)
    return torch.stack(out)
```

现在将上述算法应用到一个带有 4 个锚框的具体实例中。为简单起见,假设预测的偏移量都是零,这意味着预测的边界框就是锚框。对于背景、狗和猫中的每个类,定义了它的预测概率。

```
anchors = torch.tensor([[0.1, 0.08, 0.52, 0.92], [0.08, 0.2, 0.56, 0.95],
                        [0.15, 0.3, 0.62, 0.91], [0.55, 0.2, 0.9, 0.88]])
offset_preds = torch.tensor([0] * anchors.numel())
```

```
cls_probs = torch.tensor([[0] * 4,          #背景的预测概率
                          [0.9, 0.8, 0.7, 0.1],  #狗的预测概率
                          [0.1, 0.2, 0.3, 0.9]]) #猫的预测概率
```

可以在图像上绘制这些预测边界框和置信度，如图 5-7 所示。

```
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, anchors * bbox_scale,
            ['dog=0.9', 'dog=0.8', 'dog=0.7', 'cat=0.9'])
```

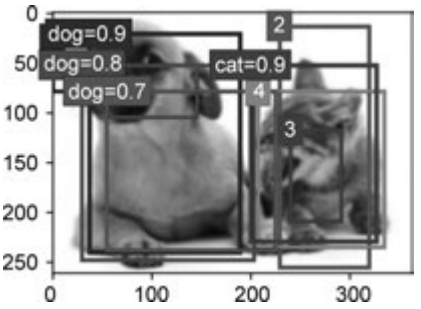


图 5-7　预测边界框和置信度

调用 multibox_detection()函数来执行非极大值抑制，其中阈值设置为 0.5。注意，在实例的张量输入中添加了维度。

```
output = multibox_detection(cls_probs.unsqueeze(dim=0),
                            offset_preds.unsqueeze(dim=0),
                            anchors.unsqueeze(dim=0),
                            nms_threshold=0.5)
print(output)
tensor([[[ 0.00,  0.90,  0.10,  0.08,  0.52,  0.92],
         [ 1.00,  0.90,  0.55,  0.20,  0.90,  0.88],
         [-1.00,  0.80,  0.08,  0.20,  0.56,  0.95],
         [-1.00,  0.70,  0.15,  0.30,  0.62,  0.91]]])
```

由结果可以看到：

- output 形状是(批量大小，锚框的数量，6)，6 表示同一预测边界框的输出信息有 6 个元素。
- 第一个元素是预测的类索引，从 0 开始(0 代表狗，1 代表猫)，值−1 表示背景或在非极大值抑制中被移除。第二个元素是预测的边界框的置信度。其余 4 个元素分别是预测边界框左上角和右下角的($x$，$y$)轴坐标(范围介于 0～1)。

删除−1 类别(背景)的预测边界框后，可以输出由非极大值抑制保存的最终预测边界框，如图 5-8 所示。

```
fig = d2l.plt.imshow(img)
for i in output[0].detach().numpy():
    if i[0] == -1:
        continue
    label = ('dog=', 'cat=')[int(i[0])] + str(i[1])
    show_bboxes(fig.axes, [torch.tensor(i[2:]) * bbox_scale], label)
```

在实践中，在执行非极大值抑制前，我们甚至可以将置信度较低的预测边界框移除，从而减少此算法中的计算量，也可以对非极大值抑制的输出结果进行后处理。例如，只保留置信度更高的结果作为最终输出。

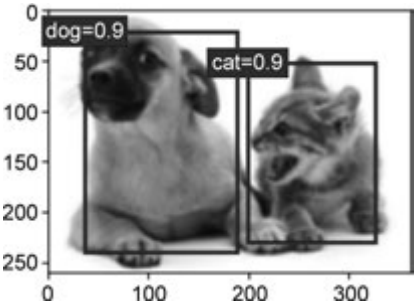


图 5-8 删除－1 类后的预测边界框

## 5.3 区域卷积神经网络系列

传统的目标检测效率很低，太耗时。那么有没有高效的检测方法呢？下面对几种常用的高效方法进行介绍。

### 5.3.1 R-CNN 算法

R-CNN 首先对图像选取若干提议区域（如锚框就是一种选取方法）并标注它们的类别和边界框（如偏移量）。然后，用卷积神经网络对每个提议区域进行前向计算抽取特征。之后，用每个提议区域的特征预测类别和边界框。图 5-9 描述了 R-CNN 模型。

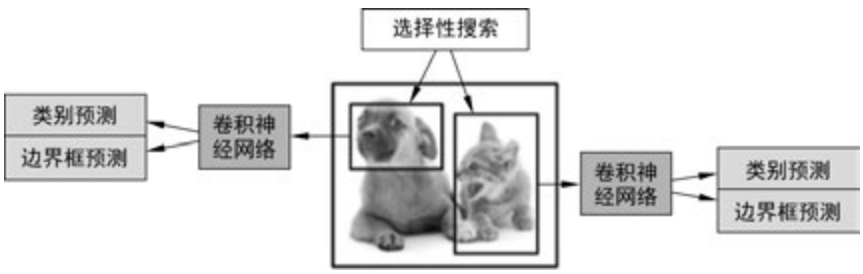


图 5-9 R-CNN 模型

具体来说，R-CNN 主要由以下 4 步构成。

（1）对输入图像使用选择性搜索（selective search）来选取多个高质量的提议区域。这些提议区域通常是在多个尺度下选取的，并具有不同的形状和大小。每个提议区域将被标注类别和真实边界框。

（2）选取一个预训练的卷积神经网络，并将其在输出层之前截断。将每个提议区域变形为网络需要的输入尺寸，并通过前向计算输出抽取的提议区域特征。

（3）将每个提议区域的特征连同其标注的类别作为一个样本，训练多个支持向量机对目标分类。其中每个支持向量机用来判断样本是否属于某一个类别。

（4）将每个提议区域的特征连同其标注的边界框作为一个样本，训练线性回归模型来预测真实边界框。

R-CNN 虽然通过预训练的卷积神经网络有效抽取了图像特征，但它的主要缺点是速度慢。想象一下，我们可能从一幅图像中选出上千个提议区域，对该图像进行目标检测将导致上千次的卷积神经网络的前向计算。这个巨大的计算量令 R-CNN 难以在实际应用中被广泛采用。

### 5.3.2 Fast R-CNN 算法

R-CNN 的主要性能瓶颈在于需要对每个提议区域独立抽取特征。由于这些区域通常有大量重叠，独立的特征抽取会导致大量的重复计算。Fast R-CNN 对 R-CNN 的一个主要改进在于只对整幅图像进行卷积神经网络的前向计算。图 5-10 描述了 Fast R-CNN 模型。

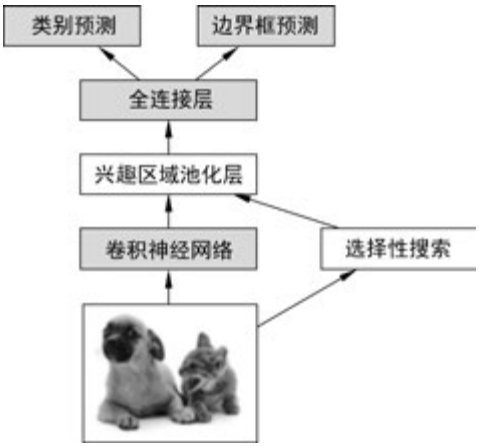


图 5-10 Fast R-CNN 模型

Fast R-CNN 模型的主要计算步骤为：

(1) 与 R-CNN 相比，Fast R-CNN 用来提取特征的卷积神经网络的输入是整幅图像，而不是各个提议区域。而且，这个网络通常会参与训练，即更新模型参数。设输入为一幅图像，将卷积神经网络的输出的形状记为 $1\times c\times h_1\times w_1$。

(2) 假设选择性搜索生成 $n$ 个提议区域。这些形状各异的提议区域在卷积神经网络的输出上分别标出形状各异的兴趣区域。这些兴趣区域需要抽取出形状相同的特征(假设高和宽均分别指定为 $h_2$ 和 $w_2$)以便于连接后输出。Fast R-CNN 引入兴趣区域池化(Region of Interest Pooling，RoI 池化)层，将卷积神经网络的输出和提议区域作为输入，输出连接后的各个提议区域抽取的特征，形状为 $n\times c\times h_2\times w_2$。

(3) 通过全连接层将输出形状变换为 $n\times d$，其中超参数 $d$ 取决于模型设计。

(4) 预测类别时，将全连接层的输出的形状再变换为 $n\times q$ 并使用 softmax 回归($q$ 为类别个数)。预测边界框时，将全连接层的输出的形状变为 $n\times 4$。也就是说，为每个提议区域预测类别和边界框。Fast R-CNN 中提出的兴趣区域池化层对每个区域的输出形状是可以直接指定的，例如，指定每个区域输出的高和宽分别为 $h_2$ 和 $w_2$。假设某一兴趣区域窗口的高和宽分别为 $h$ 和 $w$，该窗口将被划分形状为 $h_2\times w_2$ 的子窗口网格，且每个子窗口的大小约为 $(h/h_2)\times(w/w_2)$。任一子窗口的高和宽要取整，其中的最大元素作为该子窗口的输出。因此，兴趣区域池化层可从形状各异的兴趣区域中均抽取出形状相同的特征。

图 5-11 中，在 4×4 的输入上选取了左上角的 3×3 区域作为兴趣区域。对于该兴趣区域，通过 2×2 兴趣区域池化层得到一个 2×2 的输出。4 个划分后的子窗口分别含有元素 0、1、4、5(5 最大)，2、6(6 最大)，8、9(9 最大)，10。

下面使用 ROIPooling()函数来演示兴趣区域池化层的计算。假设卷积神经网络抽取的特征 $X$ 的高和宽均为 4 且只有单通道。

```
import torch
import torchvision
```


图 5-11 2×2 兴趣区域池化层

```
X = torch.arange(16, dtype = torch.float).view(1, 1, 4, 4)
print(X)
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

假设图像的高和宽均为 40 像素。再假设选择性搜索在图像上生成了两个提议区域：每个区域由 5 个元素表示，分别为区域目标类别、左上角的 $x$ 和 $y$ 轴坐标以及右下角的 $x$ 和 $y$ 轴坐标。

```
rois = torch.tensor([[0, 0, 0, 20, 20], [0, 0, 10, 30, 30]], dtype = torch.float)
```

由于 $X$ 的高和宽是图像的高和宽的 1/10，以上两个提议区域中的坐标先按 spatial_scale 自乘 0.1，然后在 $X$ 上分别标出兴趣区域 $X[:,:,0:3,0:3]$ 和 $X[:,:,1:4,0:4]$。最后对这两个兴趣区域分别划分子窗口网格并抽取高和宽为 2 的特征。

```
torchvision.ops.roi_pool(X, rois, output_size = (2, 2), spatial_scale = 0.1)
tensor([[[[ 5.,  6.],
          [ 9., 10.]]],

        [[[ 9., 11.],
          [13., 15.]]]])
```

### 5.3.3 Faster R-CNN 算法

Fast R-CNN 通常需要在选择性搜索中生成较多的提议区域，以获得较精确的目标检测结果。

Faster R-CNN 提出将选择性搜索替换成区域提议网络（Region Proposal Network，RPN），从而减少提议区域的生成数量，并保证目标检测的精度。

Faster R-CNN 的模型结构如图 5-12 所示。

与 Fast R-CNN 相比，只有生成提议区域的方法从选择性搜索变成了区域提议网络，而其他部分均保持不变。具体来说，区域提议网络的计算步骤如下：

（1）用填充为 1 的 3×3 卷积层变换卷积神经网络的输出，并将输出通道数记为 $c$。这样，卷积神经网络为图像抽取的特征图中的每个单元均得到一个长度为 $c$ 的新特征。

（2）以特征图每个单元为中心，生成多个不同大小和宽高比的锚框并标注它们。

（3）用锚框中心单元长度为 $c$ 的特征分别预测该锚框的二元类别（含目标还是背景）和边界框。

（4）使用非极大值抑制，从预测类别为目标的预测边界框中移除相似的结果。最终输出的预测边界框即兴趣区域池化层所需要的提议区域。

值得一提的是，区域提议网络作为 Faster R-CNN 的一部分，是和整个模型一起训练得到

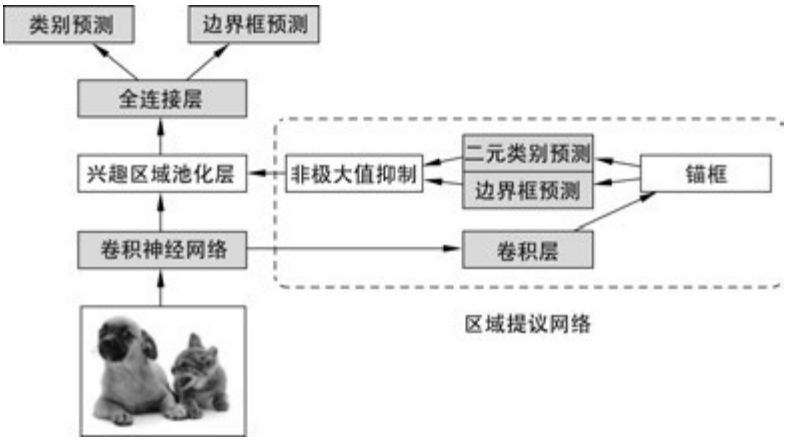


图 5-12　Faster R-CNN 的模型结构

的。也就是说，Faster R-CNN 的目标函数既包括目标检测中的类别和边界框预测，又包括区域提议网络中锚框的二元类别和边界框预测。最终，区域提议网络能够生成高质量的提议区域，从而在减少提议区域数量的情况下也能保证目标检测的精度。

### 5.3.4　Mask R-CNN 算法

如果训练数据还标注了每个目标在图像上的像素级位置，那么 Mask R-CNN 能有效利用这些详尽的标注信息进一步提升目标检测的精度。Mask R-CNN 的模型结构如图 5-13 所示。

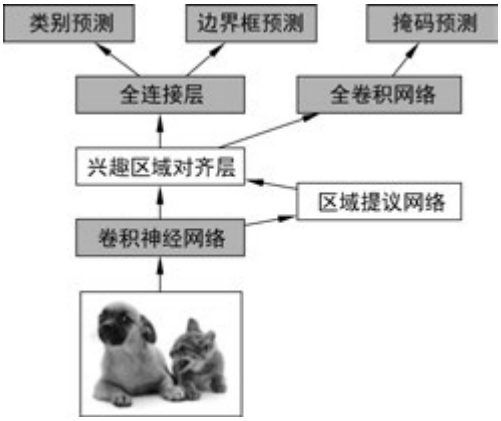


图 5-13　Mask R-CNN 的模型结构

Mask R-CNN 在 Faster R-CNN 的基础上做了修改。Mask R-CNN 将兴趣区域池化层替换成了兴趣区域对齐层，即通过双线性插值来保留特征图上的空间信息，从而更适用于像素级预测。

## 5.4　SPP-Net 算法

针对卷积神经网络重复运算问题，微软研究院提出一种 SPP-Net 算法，通过在卷积层和全连接层之间加入空间金字塔池化结构(Spatial Pyramid Pooling，SPP)代替 R-CNN 算法，在输入卷积神经网络前对各个候选区域进行剪裁、缩放操作，使其图像子块尺寸一致。

利用空间金字塔池化结构有效避免了 R-CNN 算法对图像区域裁剪、缩放操作导致的图像物体裁剪不全以及形状扭曲等问题，更重要的是解决了卷积神经网络对图像重复特征提取

的问题，大大提高了产生候选框的速度，且节省了计算成本。

SPP-Net 算法流程为：

(1) 区域提名。用 Selective Search 从原图中生成 2000 个左右的候选窗口，这一步和 R-CNN 一样。

(2) 区域大小缩放。SPP-Net 不再进行区域大小归一化，而是缩放到 $\min(w,h)=s$，即统一长宽的最短边长度，$s$ 选自{480,576,688,864,1200}中的一个，选择的标准是使得缩放后的候选框大小与 224×224 最接近。

(3) 特征提取。利用 SPP-Net 网络结构提取特征：把整幅待检测的图片输入 CNN 中进行一次性特征提取，得到特征图，然后在特征图中找到各个候选框的区域，再对各个候选框采用金字塔空间池化，提取出固定长度的特征向量。

(4) 分类与回归。类似于 R-CNN，利用 SVM 基于上面的特征训练分类器模型，用边框回归来微调候选框的位置。

SSP-Net 针对 R-CNN 的瓶颈提出两大核心创新，显著提升了目标检测的效率和精度：

(1) 利用空间金字塔池化结构。

(2) 对整幅图片只进行了一次特征提取，加快运算速度。

空间金字塔池化是什么？即把原来的特征图分成 4×4=16 块，2×2=4 块，1×1=1 块(不变)，总共 21 块，取每块的最大值作为代表，即每幅特征图就有 21 维的参数，总共卷积出 256 个特征图，则送入全连接层的维度就是 21256。这样就解决了输入数据大小任意的问题。

这样就把一幅任意大小的图片转换成了一个固定大小的 21 维特征(也可以设计其他维数的输出，增加金字塔的层数，或者改变划分网格的大小)。上面的三种不同刻度的划分，每一种刻度称为金字塔的一层，每一幅图片的块大小称为窗口尺寸(windows size)。

下面的代码是采用 PyTorch 深度学习框架构建一个 SPP 层：

```
#coding = utf - 8
import math
import torch
import torch.nn.functional as F

#构建 SPP 层(空间金字塔池化层)
class SPPLayer(torch.nn.Module):

    def __init__(self, num_levels, pool_type = 'max_pool'):
        super(SPPLayer, self).__init__()
        self.num_levels = num_levels
        self.pool_type = pool_type

    def forward(self, x):
        num, c, h, w = x.size() # num:样本数量,c:通道数,h:高,w:宽
        for i in range(self.num_levels):
            level = i + 1
            kernel_size = (math.ceil(h / level), math.ceil(w / level))
            stride = (math.ceil(h / level), math.ceil(w / level))
            pooling = (math.floor((kernel_size[0] * level - h + 1)/2), math.floor((kernel_size
[1] * level - w + 1)/2))

            #选择池化方式
            if self.pool_type == 'max_pool':
                tensor = F.max_pool2d(x, kernel_size = kernel_size, stride = stride, padding =
pooling).view(num, -1)
```

```
            else:
                tensor = F.avg_pool2d(x, kernel_size = kernel_size, stride = stride, padding =
pooling).view(num, -1)

            #展开、拼接
            if (i == 0):
                x_flatten = tensor.view(num, -1)
            else:
                x_flatten = torch.cat((x_flatten, tensor.view(num, -1)), 1)
        return x_flatten
```

## 5.5 单发多框检测算法

R-CNN 有两个网络，一个是区域提议网络(RPN)，用于抽取锚框特征，另一个主网络，用于训练。SSD(Single Shot MultiBox Detector)只用一个网络来训练模型，其过程如图 5-14 所示。

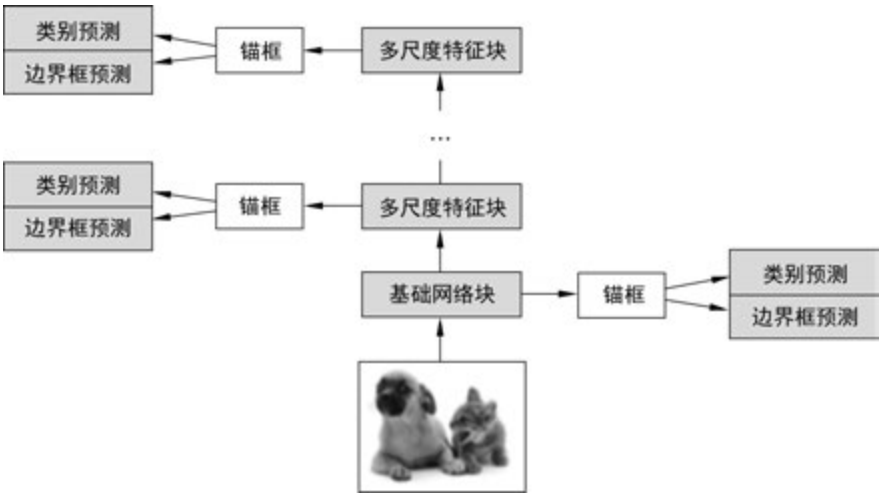


图 5-14 SSD 训练模型

SSD 中使用一个基础网络从原始图像中抽取特征，然后多个卷积层将高宽减半。在每段中都生成锚框，然后对这些锚框进行类别预测和边界框预测。

- 锚框生成方式：对于每个像素，生成 $n+m-1$ 个锚框(缩放比 scale 取 $s_1,s_2,\cdots,s_n$，宽高比取值 $r_1,r_2,\cdots,r_m$)。对于整幅输入图像，将共生成 $wh(n+m-1)$个锚框。

SSD 预测速度比 Faster R-CNN 系列快，但是精度会差一些。

### 5.5.1 多尺度锚框

以输入图像的每个像素为中心，生成了多个锚框。基本而言，这些锚框代表了图像不同区域的样本。然而，如果为每个像素都生成锚框，最终可能会得到太多需要计算的锚框。想象一个 261×363 的输入图像，如果以每个像素为中心生成 5 个形状不同的锚框，就需要在图像上标记和预测超过 470 000 个锚框(261×363×5)。

减少图像上的锚框数量并不困难，例如，可以在输入图像中均匀采样一小部分像素，并以它们为中心生成锚框。此外，在不同尺度下，可以生成不同数量和不同大小的锚框。直观地说，比起较大的目标，较小的目标在图像上出现的可能性更多样。例如，1×1、1×2 和 2×2 的目标可以分别以 4、2 和 1 种可能的方式出现在 2×2 的图像上。因此，当使用较小的锚框检测

较小的物体时，可以采样更多的区域，而对于较大的物体，可以采样较少的区域。

为了演示如何在多个尺度下生成锚框，让我们先读取一幅图像。它的高度和宽度分别为261和363像素：

```
import torch
from d2l import torch as d2l

img = d2l.plt.imread('catdog.jpg')
h, w = img.shape[:2]
print(h, w)
261 363
```

定义函数display_anchors()，用于在特征图(fmap)上生成锚框(anchors)，每个锚框以像素作中心。假设有 $c$ 幅形状为 $h \times w$ 的特征图，那么共生成 $hw$ 组锚框，每一组锚框有 $n+m-1$ 个。

```
def display_anchors(fmap_w, fmap_h, s):
    """根据特征图的高宽以及缩放比 scale,生成锚框,并打印出来。ratios 固定为[1, 2, 0.5]"""
    d2l.set_figsize()
    #构造一幅特征图,batch_size = 1,channnel = 10。这两个维度上的值不影响输出
    fmap = torch.zeros((1, 10, fmap_h, fmap_w))
    anchors = d2l.multibox_prior(fmap, sizes = s, ratios = [1, 2, 0.5])
    bbox_scale = torch.tensor((w, h, w, h))
    d2l.show_bboxes(d2l.plt.imshow(img).axes,
                    anchors[0] * bbox_scale)

display_anchors(fmap_w = 4, fmap_h = 4, s = [0.15])      #s = 0.15,锚框大小是特征图面积的
                                                         #0.15 * 0.15 = 0.225 倍
d2l.plt.show()
```

运行结果如图5-15所示。

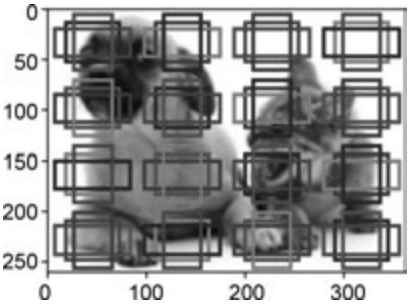


图5-15 多尺度锚框

然后，将特征图的高和宽分别减半，并用更大的锚框检测更大的目标。当锚框大小设为0.4时，有些锚框的区域有重合，效果如图5-16所示。

```
display_anchors(fmap_w = 2, fmap_h = 2, s = [0.4])
```

最后，将特征图的宽度进一步减半至1，并将锚框大小增至0.8。此时锚框中心即图像中心，效果如图5-17所示。

```
display_anchors(fmap_w = 1, fmap_h = 1, s = [0.8])
```

既然已在多个尺度上生成了不同大小的锚框，相应地，需要在不同尺度下检测不同大小的目标。下面介绍一种基于卷积神经网络的方法。

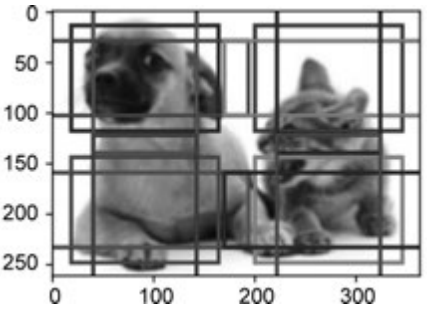


图 5-16　特征图的高和宽分别减半

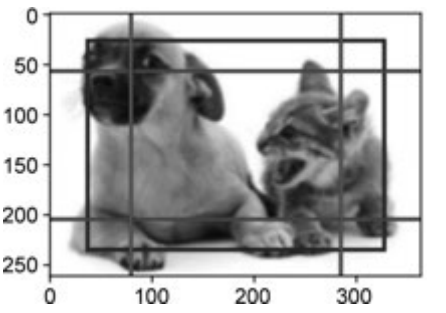


图 5-17　特征图的宽进一步减半至 1

### 5.5.2　多尺度检测

在某个尺度下，假设 $c_i$ 张特征图形状有 $h \times w$ 组不同中心的锚框的类别和偏移量，且每组的锚框个数为 $a$。例如，在 5.5.1 节实例的第一个尺度下，依据 10(通道数)幅形状为 $4 \times 2$ 的特征图生成了 8 组不同中心的锚框，且每组含 3 个锚框。

接着，依据真实边界框的类别和位置，每个锚框将被标注类别和偏移量。在当前的尺度下，目标检测模型需要根据输入图像预测 $h \times w$ 组不同中心的锚框的类别和偏移量。

假设此处的 $c_i$ 张特征图为卷积神经网络根据输入图像进行前向计算所得的中间输出。既然每幅特征图上都有 $h \times w$ 个不同的空间位置，那么相同空间位置可以看作含有 $c_i$ 个单元。

根据二维卷积层中感受野的定义，特征图在相同空间位置的 $c_i$ 个单元在输入图像上的感受野相同，并表征了同一感受野内的输入图像信息。

因此，可以将特征图在相同空间位置的 $c_i$ 个单元变换为以该位置为中心生成的 $a$ 个锚框的类别和偏移量。不难发现，本质上，是用输入图像在某个感受野区域内的信息来预测输入图像上与该区域位置相近的锚框的类别和偏移量。

当不同层的特征图在输入图像上分别拥有不同大小的感受野时，它们将分别用来检测不同大小的目标。例如，可以通过设计网络，令较接近输出层的特征图中的每个单元拥有更广阔的感受野，从而检测输入图像中更大尺寸的目标。