学习笔记-7: Task 7

19 - 目标检测基础

目标检测和边界框

边界框 bounding box 的表示：bbox = [左上x, 左上y, 右下x, 右下y]

使用 matplotlib 绘制边界框时传入 xy = (bbox[0], bbox[1]), width = bbox[2]-bbox[0], height = bbox[3]-bbox[1]

锚框

以输入图像的每个像素为中心生成多个大小和宽高比(aspect ratio)不同的边界框。这些边界框被称为锚框(anchor box)

生成多个锚框

假设输入图像高为 h，宽为w。现在分别以图像的每个像素为中心生成不同形状的锚框。设大小为$s \in (0,1]$且宽高比为$r > 0$，那么锚框的宽和高将分别为$ws\sqrt{r}$和$hs/\sqrt{r}$。当中心位置给定时，已知宽和高的锚框是确定的。

这里的s实际上是锚框面积与输入图像面积之比的算术平方根。

下面分别设定好一组大小$s_1,\ldots,s_n$和一组宽高比$r_1,\ldots,r_m$。如果以每个像素为中心时使用所有的大小与宽高比的组合，输入图像将一共得到$whnm$个锚框。虽然这些锚框可能覆盖了所有的真实边界框，但计算复杂度容易过高。因此，我们通常只对包含s_1或r_1的大小与宽高比的组合感兴趣，即

$$(s_1,r_1),(s_1,r_2),\ldots,(s_1,r_m),(s_2,r_1),(s_3,r_1),\ldots,(s_n,r_1).$$

也就是说，以相同像素为中心的锚框的数量为$n+m-1$。对于整个输入图像，一共生成$wh(n+m-1)$个锚框。

以上生成锚框的方法在MultiBoxPrior函数中实现。指定输入、一组大小和一组宽高比，该函数将返回输入的所有锚框。

```
def MultiBoxPrior(feature_map, sizes=[0.75, 0.5, 0.25], ratios=[1, 2, 0.5]):
    """
    # 按照上文所述方法实现, anchor表示成(xmin, ymin, xmax, ymax).
    Args:
        feature_map: torch tensor, shape: [N, C, H, W]
        sizes: list of sizes (0~1) of generated anchor boxes
        ratios: list of aspect ratios (non-negative) of generated anchor boxes
    Returns:
        anchors of shape (1, num_anchors, 4). 由于batch里每个都一样, 所以第一维为1
    """
    pairs = [] # pair of (size, sqrt(ratio))

    # 生成n+m-1个框
    for r in ratios:
```

```
            pairs.append([sizes[0], math.sqrt(r)])
        for s in sizes[1:]:
            pairs.append([s, math.sqrt(ratios[0])])

        pairs = np.array(pairs)

        # 生成相对于坐标中心点的框(x, y, x, y)
        # 这里已经分别除以了w和h, 进行标准化
        ss1 = pairs[:, 0] * pairs[:, 1] # size * sqrt(ratio), all width
        ss2 = pairs[:, 0] / pairs[:, 1] # size / sqrt(ratio), all height

        base_anchors = np.stack([-ss1, -ss2, ss1, ss2], axis=1) / 2

        # 将坐标点和anchor组合起来生成hw(n+m-1)个框输出
        h, w = feature_map.shape[-2:]
        shifts_x = np.arange(0, w) / w
        shifts_y = np.arange(0, h) / h
        shift_x, shift_y = np.meshgrid(shifts_x, shifts_y)

        shift_x = shift_x.reshape(-1)
        shift_y = shift_y.reshape(-1)
        # shifts里包含了所有中心点的坐标
        shifts = np.stack((shift_x, shift_y, shift_x, shift_y), axis=1)
        anchors = shifts.reshape((-1, 1, 4)) + base_anchors.reshape((1, -1, 4))

        return torch.tensor(anchors, dtype=torch.float32).view(1, -1, 4)
```

```
X = torch.Tensor(1, 3, h, w)  # 构造输入数据
Y = MultiBoxPrior(X, sizes=[0.75, 0.5, 0.25], ratios=[1, 2, 0.5])
Y.shape
>>> torch.Size([1, 2042040, 4])
```

这里的2042040就是通过$wh(n+m-1)$计算得到的，即728×561×(3+3-1)=2042040

可以看出，返回锚框变量y的形状为(1，锚框个数，4)。将锚框变量y的形状变为(图像高，图像宽，以相同像素为中心的锚框个数，4)后(这里高和宽的顺序一定不能反)，就可以通过指定像素位置来获取所有以该像素为中心的锚框了。下面的例子访问了以(250，250)为中心的第一个锚框。它有4个元素，分别是锚框左上角的x和y轴坐标和右下角的x和y轴坐标，其中x和y轴的坐标值已分别除以图像的宽和高，因此值域均为0和1之间。

```
# 展示某个像素点的anchor
boxes = Y.reshape((h, w, 5, 4))
boxes[250, 250, 0, :]   # torch.tensor([w, h, w, h], dtype=torch.float32)
>>> tensor([-0.0316,  0.0706,  0.7184,  0.8206])
```

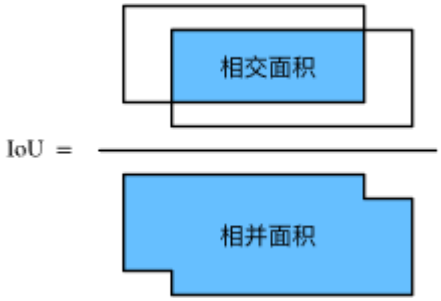
> 可以验证一下以上输出对不对: size和ratio分别为0.75和1, 则(归一化后的)宽高均为0.75, 所以输出是正确的(0.75 = 0.7184 + 0.0316 = 0.8206 - 0.0706).

交并比

随机生成的锚框中，某个锚框较好地覆盖了图像中的目标。如果该目标的真实边界框已知，这里的“较好”该如何量化呢？一种直观的方法是衡量锚框和真实边界框之间的相似度。Jaccard系数(Jaccard index)可以衡量两个集合的相似度。给定集合$\mathcal{A}$和$\mathcal{B}$，它们的Jaccard系数即二者交集大小除以二者并集大小：

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

实际上，我们可以把边界框内的像素区域看成是像素的集合。如此一来，我们可以用两个边界框的像素集合的Jaccard系数衡量这两个边界框的相似度。当衡量两个边界框的相似度时，通常将Jaccard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，如下图所示。交并比的取值范围在0和1之间：0表示两个边界框无重合像素，1表示两个边界框相等。




```
def compute_intersection(set_1, set_2):
    """
    计算anchor之间的交集
    Args:
        set_1: a tensor of dimensions (n1, 4), anchor表示成(xmin, ymin, xmax,
ymax)
        set_2: a tensor of dimensions (n2, 4), anchor表示成(xmin, ymin, xmax,
ymax)
    Returns:
        intersection of each of the boxes in set 1 with respect to each of the
boxes in set 2, shape: (n1, n2)
    """
    # PyTorch auto-broadcasts singleton dimensions
    """
    用[:, :2]取出4个坐标里的前两个，是xmin和ymin
    用[:, 2:]取出4个坐标里的后两个，是xmax和ymax
    对tensor使用.unsqueeze(i)，就是将tensor在第i维展开，且将其第i维的size设为1
    """
    lower_bounds = torch.max(set_1[:, :2].unsqueeze(1), set_2[:, :2].unsqueeze(0))
# (n1, n2, 2)
    upper_bounds = torch.min(set_1[:, 2:].unsqueeze(1), set_2[:, 2:].unsqueeze(0))
# (n1, n2, 2)
    # torch.clamp(input, min, max)将input的所有元素限制在(min, max)的范围内，>max的
值换成max，<min的值换成min
    intersection_dims = torch.clamp(upper_bounds - lower_bounds, min=0)  # (n1,
n2, 2)
    return intersection_dims[:, :, 0] * intersection_dims[:, :, 1]  # (n1, n2)
```

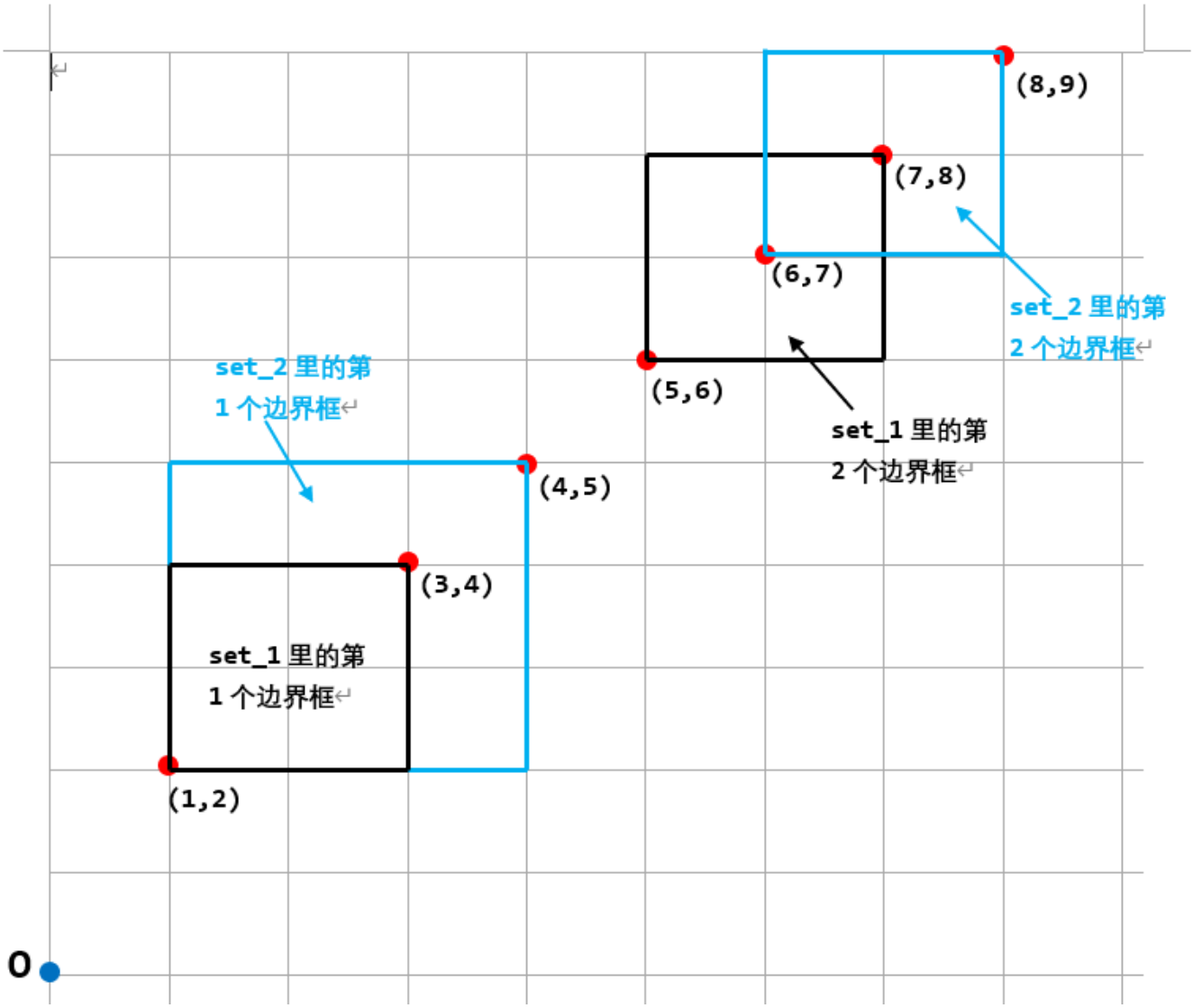
```
def compute_jaccard(set_1, set_2):
    """
    计算anchor之间的Jaccard系数(IoU)
    Args:
        set_1: a tensor of dimensions (n1, 4), anchor表示成(xmin, ymin, xmax,
ymax)
        set_2: a tensor of dimensions (n2, 4), anchor表示成(xmin, ymin, xmax,
ymax)
    Returns:
        Jaccard Overlap of each of the boxes in set 1 with respect to each of the
boxes in set 2, shape: (n1, n2)
    """
    # Find intersections
    intersection = compute_intersection(set_1, set_2)  # (n1, n2)

    # Find areas of each box in both sets
    areas_set_1 = (set_1[:, 2] - set_1[:, 0]) * (set_1[:, 3] - set_1[:, 1])  #
(n1)
    areas_set_2 = (set_2[:, 2] - set_2[:, 0]) * (set_2[:, 3] - set_2[:, 1])  #
(n2)

    # Find the union
    # PyTorch auto-broadcasts singleton dimensions
    union = areas_set_1.unsqueeze(1) + areas_set_2.unsqueeze(0) - intersection  #
(n1, n2)

    return intersection / union  # (n1, n2)
```

以上compute_intersection的实现过程比较抽象，以下为一个简单例子

```
set_1 = torch.tensor([[1,2,3,4],[5,6,7,8]])
set_2 = torch.tensor([[1,2,4,5],[6,7,8,9]])
a = torch.max(set_1[:, :2].unsqueeze(1), set_2[:, :2].unsqueeze(0))
a
>>> tensor([[[1, 2],
         [6, 7]],

        [[5, 6],
         [6, 7]]])
b = torch.min(set_1[:, 2:].unsqueeze(1), set_2[:, 2:].unsqueeze(0))
b
>>> tensor([[[3, 4],
         [3, 4]],

        [[4, 5],
         [7, 8]]])
b-a
>>> tensor([[[ 2,  2],
         [-3, -3]],

        [[-1, -1],
         [ 1,  1]]])
```

```
c = torch.clamp(b-a, min=0)
c
>>>tensor([[[2, 2],
          [0, 0]],

         [[0, 0],
          [1, 1]]])
c[:, :, 0] * c[:, :, 1]
>>> tensor([[4, 0],
         [0, 1]])
```

可以看到，输出的tensor的第i行第j列的值就是set_1中第i个边界框与set_2中第j个边界框的重叠面积。如下图所示，set_1中第1个边框和set_2中第1个边框的重叠面积为4；set_1中第1个边框和set_2中第2个边框的重叠面积为0；set_1中第2个边框和set_2中第1个边框的重叠面积为0；set_1中第2个边框和set_2中第2个边框的重叠面积为1。



标注训练集的锚框

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，需要为每个锚框标注两类标签：一是锚框所含目标的类别，简称类别；二是真实边界框相对锚框的偏移量，简称偏移量(offset)。在目标检测时，首先生成多个锚框，然后为每个锚框预测类别以及偏移量，接着根据预测的偏移量调整锚框位置从而得到预测边界

框，最后筛选需要输出的预测边界框。

在目标检测的训练集中，每个图像已标注了真实边界框的位置以及所含目标的类别。在生成锚框之后，我们主要依据与锚框相似的真实边界框的位置和类别信息为锚框标注。那么，该如何为锚框分配与其相似的真实边界框呢？

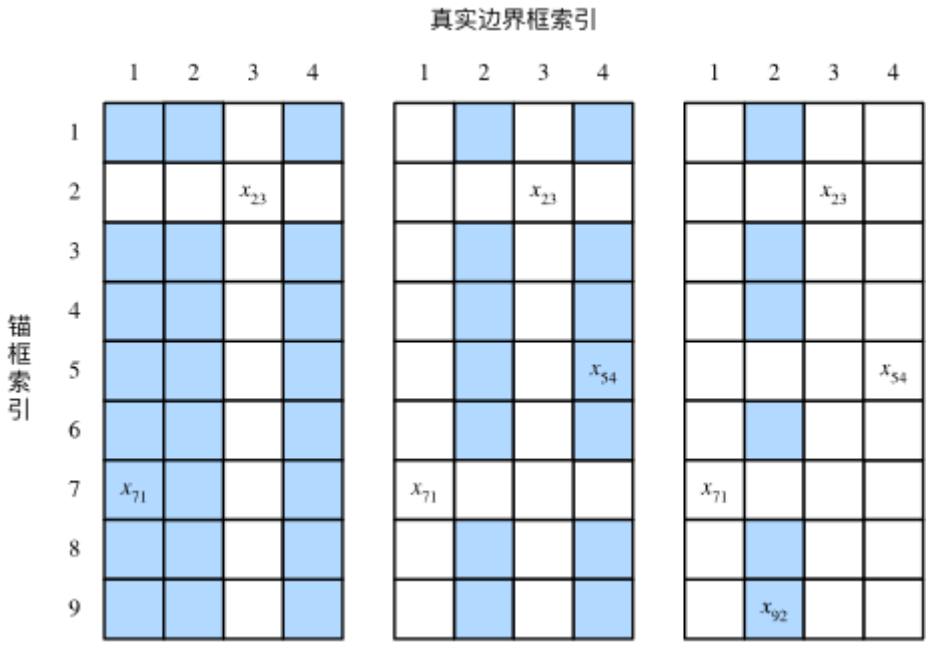
假设图像中锚框分别为$A_1, A_2, \ldots, A_{n_a}$，真实边界框分别为$B_1, B_2, \ldots, B_{n_b}$，且$n_a \geq n_b$。定义矩阵$\boldsymbol{X} \in \mathbb{R}^{n_a \times n_b}$，其中第$i$行第$j$列的元素$x_{ij}$为锚框$A_i$与真实边界框$B_j$的交并比。

首先找出矩阵$\boldsymbol{X}$中最大的元素，并将该元素的行索引与列索引分别记为i_1, j_1。为锚框A_{i_1}分配真实边界框B_{j_1}。显然，锚框A_{i_1}和真实边界框B_{j_1}在所有的“锚框—真实边界框”的配对中相似度最高。接下来，将矩阵$\boldsymbol{X}$中第i_1行和第j_1列上的所有元素丢弃。找出矩阵$\boldsymbol{X}$中剩余的最大元素，并将该元素的行索引与列索引分别记为i_2, j_2。为锚框A_{i_2}分配真实边界框B_{j_2}，再将矩阵$\boldsymbol{X}$中第i_2行和第j_2列上的所有元素丢弃。此时矩阵$\boldsymbol{X}$中已有两行两列的元素被丢弃。

依此类推，直到矩阵$\boldsymbol{X}$中所有n_b列元素全部被丢弃。这个时候已为n_b个锚框各分配了一个真实边界框。

接下来只遍历剩余的$n_a - n_b$个锚框：给定其中的锚框A_i，根据初始的矩阵$\boldsymbol{X}$的第i行找到与A_i交并比最大的真实边界框B_j，且只有当该交并比大于预先设定的阈值时，才为锚框A_i分配真实边界框B_j。

如下图左所示，假设矩阵$\boldsymbol{X}$中最大值为x_{23}，因此为锚框A_2分配真实边界框B_3。然后，丢弃矩阵中第2行和第3列的所有元素，找出剩余阴影部分的最大元素x_{71}，为锚框A_7分配真实边界框B_1。接着如下图中所示，丢弃矩阵中第7行和第1列的所有元素，找出剩余阴影部分的最大元素x_{54}，为锚框A_5分配真实边界框B_4。最后如下图右所示，丢弃矩阵中第5行和第4列的所有元素，找出剩余阴影部分的最大元素x_{92}，为锚框A_9分配真实边界框B_2。之后只需遍历除去A_2, A_5, A_7, A_9的剩余锚框，并根据阈值判断是否为剩余锚框分配真实边界框。



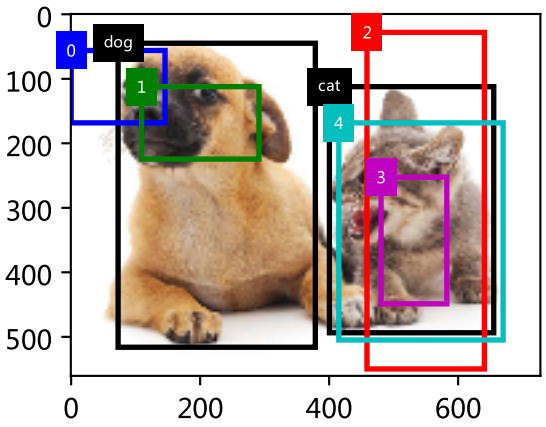
现在可以标注锚框的类别和偏移量了。如果一个锚框A被分配了真实边界框B，将锚框A的类别设为B的类别，并根据B和A的中心坐标的相对位置以及两个框的相对大小为锚框A标注偏移量。由于数据集中各个框的位置和大小各异，因此这些相对位置和相对大小通常需要一些特殊变换，才能使偏移量的分布更均匀从而更容易拟合。设锚框A及其被分配的真实边界框B的中心坐标分别为(x_a, y_a)和(x_b, y_b)，A和B的宽分别为w_a和w_b，高分别为h_a和h_b，一个常用的技巧是将A的偏移量标注为

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

其中常数的默认值为$\mu_x = \mu_y = \mu_w = \mu_h = 0, \sigma_x = \sigma_y = 0.1, \sigma_w = \sigma_h = 0.2$ 。如果一个锚框没有被分配真实边界框，只需将该锚框的类别设为背景。类别为背景的锚框通常被称为负类锚框，其余则被称为正类锚框。

下面演示一个具体的例子。为读取的图像中的猫和狗定义真实边界框，其中第一个元素为类别(0为狗，1为猫)，剩余4个元素分别为左上角的x和y轴坐标以及右下角的x和y轴坐标(值域在0到1之间)。这里通过左上角和右下角的坐标构造了5个需要标注的锚框，分别记为$A_0, \ldots, A_4$ (程序中索引从0开始)。先画出这些锚框与真实边界框在图像中的位置。

```
bbox_scale = torch.tensor((w, h, w, h), dtype=torch.float32)
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']);
```




```
# 验证一下写的compute_jaccard函数
compute_jaccard(anchors, ground_truth[:, 1:])
>>> tensor([[0.0536, 0.0000],
        [0.1417, 0.0000],
        [0.0000, 0.5657],
        [0.0000, 0.2059],
        [0.0000, 0.7459]])
```

根据锚框与真实边界框在图像中的位置来分析这些标注的类别。首先，在所有的“锚框—真实边界框”的配对中，锚框A_4与猫的真实边界框的交并比最大，因此锚框A_4的类别标注为猫。不考虑锚框A_4或猫的真实边界框，在剩余的“锚框—真实边界框”的配对中，最大交并比的配对为锚框A_1和狗的真实边界框，因此锚框A_1的类别标注为狗。接下来遍历未标注的剩余3个锚框：与锚框A_0交并比最大的真实边界框的类别为狗，但交并比小于阈值(默认为0.5)，因此类别标注为背景；与锚框A_2交并比最大的真实边界框的类别为猫，且交并比大于阈值，因此类别标注为猫；与锚框A_3交并比最大的真实边界框的类别为猫，但交并比小于阈值，因此类别标注为背景。

下面实现MultiBoxTarget函数来为锚框标注类别和偏移量。该函数将背景类别设为0，并令从零开始的目标类别的整数索引自加1(1为狗，2为猫)。

```
def assign_anchor(bb, anchor, jaccard_threshold=0.5):
    """
    按照上面讲的方法为每个anchor分配真实的bounding box
    anchor表示成归一化(xmin, ymin, xmax, ymax)
    Args:
        bb: 真实边界框(bounding box), shape: (nb, 4)
        anchor: 待分配的anchor, shape: (na, 4)
        jaccard_threshold: 预先设定的阈值
    Returns:
        assigned_idx: shape: (na, ), 每个anchor分配的真实bb对应的索引, 若未分配任何bb
则为-1
    """
    na = anchor.shape[0]
    nb = bb.shape[0]
    jaccard = compute_jaccard(anchor, bb).detach().cpu().numpy() # shape: (na, nb)
    assigned_idx = np.ones(na) * -1  # 存放标签初始全为-1

    # 先为每个bb分配一个anchor(不要求满足jaccard_threshold)
    jaccard_cp = jaccard.copy()
    for j in range(nb):
        i = np.argmax(jaccard_cp[:, j])
        assigned_idx[i] = j
        jaccard_cp[i, :] = float("-inf") # 赋值为负无穷, 相当于去掉这一行

    # 处理还未被分配的anchor, 要求满足jaccard_threshold
    for i in range(na):
        if assigned_idx[i] == -1:
            j = np.argmax(jaccard[i, :])
            if jaccard[i, j] >= jaccard_threshold:
                assigned_idx[i] = j

    return torch.tensor(assigned_idx, dtype=torch.long)


def xy_to_cxcy(xy):
    """
    将(x_min, y_min, x_max, y_max)形式的anchor转换成(center_x, center_y, w, h)形式
的
    Args:
        xy: bounding boxes in boundary coordinates, a tensor of size (n_boxes, 4)
    Returns:
        bounding boxes in center-size coordinates, a tensor of size (n_boxes, 4)
    """
    return torch.cat([(xy[:, 2:] + xy[:, :2]) / 2,  # c_x, c_y
                      xy[:, 2:] - xy[:, :2]], 1)  # w, h

def MultiBoxTarget(anchor, label):
    """
    按照"生成多个锚框"所讲的实现, anchor表示成归一化(xmin, ymin, xmax, ymax)
```

```
    Args:
        anchor: 输入的锚框，一般是通过MultiBoxPrior生成，shape为(1，锚框总数，4)
        label: 真实标签，shape为(bn，每张图片最多的真实锚框数，5)，这里bn指的
batch_size
        第二维中，如果给定图片没有这么多锚框，可以先用-1填充空白
        最后一维中的元素为[类别标签，四个坐标值]
    Returns:
        列表，[bbox_offset，bbox_mask，cls_labels]
        bbox_offset: 每个锚框的标注偏移量，形状为(bn，锚框总数*4)
        bbox_mask: 形状同bbox_offset，每个锚框的掩码，一一对应上面的偏移量，负类锚框
(背景)对应的掩码均为0，正类锚框的掩码均为1
        cls_labels: 每个锚框的标注类别，其中0表示为背景，形状为(bn，锚框总数)
    """
    assert len(anchor.shape) == 3 and len(label.shape) == 3
    bn = label.shape[0]

    def MultiBoxTarget_one(anc, lab, eps=1e-6):
        """
        MultiBoxTarget函数的辅助函数，处理batch中的一个
        Args:
            anc: shape of (锚框总数，4)
            lab: shape of (真实锚框数，5)，5代表[类别标签，四个坐标值]
            eps: 一个极小值，防止log0
        Returns:
            offset: (锚框总数*4，)
            bbox_mask: (锚框总数*4，)，0代表背景，1代表非背景
            cls_labels: (锚框总数，)，0代表背景
        """
        an = anc.shape[0]
        # 变量的意义
        assigned_idx = assign_anchor(lab[:, 1:], anc) # (锚框总数，)
        print("a: ",  assigned_idx.shape)
        print(assigned_idx)
        bbox_mask = ((assigned_idx >= 0).float().unsqueeze(-1)).repeat(1, 4) # (锚
框总数，4)
        print("b: " , bbox_mask.shape)
        print(bbox_mask)

        cls_labels = torch.zeros(an, dtype=torch.long) # 0表示背景
        assigned_bb = torch.zeros((an, 4), dtype=torch.float32) # 所有anchor对应的
bb坐标
        for i in range(an):
            bb_idx = assigned_idx[i]
            if bb_idx >= 0: # 即非背景
                cls_labels[i] = lab[bb_idx, 0].long().item() + 1 # 注意要加一
                assigned_bb[i, :] = lab[bb_idx, 1:]
        # 计算偏移量
        center_anc = xy_to_cxcy(anc) # (center_x, center_y, w, h)
        center_assigned_bb = xy_to_cxcy(assigned_bb)

        offset_xy = 10.0 * (center_assigned_bb[:, :2] - center_anc[:, :2]) /
center_anc[:, 2:]
        offset_wh = 5.0 * torch.log(eps + center_assigned_bb[:, 2:] /
center_anc[:, 2:])
```

```
        offset = torch.cat([offset_xy, offset_wh], dim = 1) * bbox_mask # (锚框总
数, 4)

        return offset.view(-1), bbox_mask.view(-1), cls_labels
    # 组合输出
    batch_offset = []
    batch_mask = []
    batch_cls_labels = []
    # 对每一个batch调用MultiBoxTarget_one函数
    for b in range(bn):
        offset, bbox_mask, cls_labels = MultiBoxTarget_one(anchor[0, :, :],
label[b, :, :])

        batch_offset.append(offset)
        batch_mask.append(bbox_mask)
        batch_cls_labels.append(cls_labels)

    bbox_offset = torch.stack(batch_offset)
    bbox_mask = torch.stack(batch_mask)
    cls_labels = torch.stack(batch_cls_labels)

    return [bbox_offset, bbox_mask, cls_labels]
```

通过unsqueeze函数为锚框和真实边界框添加样本维

```
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]])
labels = MultiBoxTarget(anchors.unsqueeze(dim=0),
                        ground_truth.unsqueeze(dim=0))
>>> a:  torch.Size([5])
tensor([-1,  0,  1, -1,  1])
b:  torch.Size([5, 4])
tensor([[0., 0., 0., 0.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [0., 0., 0., 0.],
        [1., 1., 1., 1.]])
```

返回的结果里有3项，均为tensor。第三项表示的是锚框标注的类别。

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

返回值的第二项为掩码(mask)变量，形状为(批量大小, 锚框个数的四倍)。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，有关负类的偏移量不应影响目标函数。通过按元素乘法，掩码变量中的 0 可以在计算目标函数之前过滤掉负类的偏移量。

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

返回的第一项是为每个锚框标注的四个偏移量，其中负类锚框的偏移量标注为0。

```
labels[0]
>>> tensor([[-0.0000e+00, -0.0000e+00, -0.0000e+00, -0.0000e+00,  1.4000e+00,
          1.0000e+01,  2.5940e+00,  7.1754e+00, -1.2000e+00,  2.6882e-01,
          1.6824e+00, -1.5655e+00, -0.0000e+00, -0.0000e+00, -0.0000e+00,
         -0.0000e+00, -5.7143e-01, -1.0000e+00,  4.1723e-06,  6.2582e-01]])
```

输出预测边界框

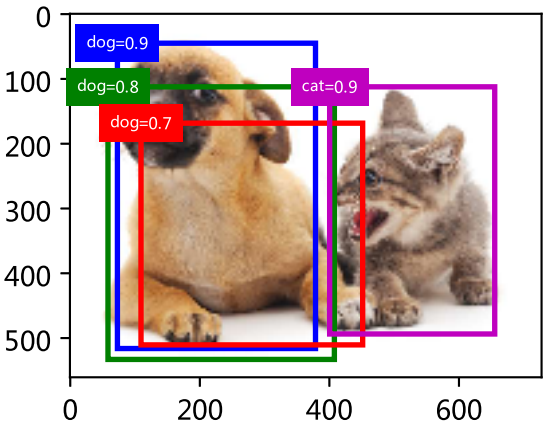
在模型预测阶段，先为图像生成多个锚框，并为这些锚框一一预测类别和偏移量。随后，根据锚框及其预测偏移量得到预测边界框。当锚框数量较多时，同一个目标上可能会输出较多相似的预测边界框。为了使结果更加简洁，可以移除相似的预测边界框。常用的方法叫作非极大值抑制(non-maximum suppression，NMS)。
下面描述了非极大值抑制的工作原理。对于一个预测边界框B，模型会计算各个类别的预测概率。设其中最大的预测概率为p，该概率所对应的类别即B的预测类别。p称为预测边界框B的置信度。在同一图像上，将预测类别非背景的预测边界框按置信度从高到低排序，得到列表L。从L中选取置信度最高的预测边界框B_1作为基准，将所有与B_1的交并比大于某阈值的非基准预测边界框从L中移除。这里的阈值是预先设定的超参数。此时，L保留了置信度最高的预测边界框并移除了与其相似的其他预测边界框。
接下来，从L中选取置信度第二高的预测边界框B_2作为基准，将所有与B_2的交并比大于某阈值的非基准预测边界框从L中移除。重复这一过程，直到L中所有的预测边界框都曾作为基准。此时L中任意一对预测边界框的交并比都小于阈值。最终，输出列表L中的所有预测边界框。

下面来看一个具体的例子。先构造4个锚框，简单起见，假设预测偏移量全是0：预测边界框即锚框。最后，构造每个类别的预测概率。

```
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([0.0] * (4 * len(anchors)))
cls_probs = torch.tensor([[0., 0., 0., 0.,],  # 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
```

在图像上打印预测边界框和它们的置信度

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



下面实现MultiBoxDetection函数来执行非极大值抑制

```
from collections import namedtuple
Pred_BB_Info = namedtuple("Pred_BB_Info", ["index", "class_id", "confidence",
"xyxy"])

def non_max_suppression(bb_info_list, nms_threshold = 0.5):
    """
    非极大抑制处理预测的边界框
    Args:
        bb_info_list: 由Pred_BB_Info(类型为namedtuple)组成的列表, 包含预测类别、置信
度等信息
        nms_threshold: 阈值
    Returns:
        output: Pred_BB_Info的列表, 只保留过滤后的边界框信息
    """
    output = []
    # 先根据置信度从高到低排序
    sorted_bb_info_list = sorted(bb_info_list, key = lambda x: x.confidence,
reverse=True)

    # 循环遍历删除冗余输出
    while len(sorted_bb_info_list) != 0:
        best = sorted_bb_info_list.pop(0)
        output.append(best)

        if len(sorted_bb_info_list) == 0:
            break

        bb_xyxy = []
        for bb in sorted_bb_info_list:
            bb_xyxy.append(bb.xyxy)
        # 计算交并比
        iou = compute_jaccard(torch.tensor([best.xyxy]),
                              torch.tensor(bb_xyxy))[0]
        # shape: (len(sorted_bb_info_list), )

        n = len(sorted_bb_info_list)
```

```
        sorted_bb_info_list = [sorted_bb_info_list[i] for i in range(n) if iou[i]
<= nms_threshold]
    return output

def MultiBoxDetection(cls_prob, loc_pred, anchor, nms_threshold = 0.5):
    """
    # 按照"生成多个锚框"所讲的实现, anchor表示成归一化(xmin, ymin, xmax, ymax)
    Args:
        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了
    """
    assert len(cls_prob.shape) == 3 and len(loc_pred.shape) == 2 and
len(anchor.shape) == 3
    bn = cls_prob.shape[0]

    def MultiBoxDetection_one(c_p, l_p, anc, nms_threshold = 0.5):
        """
        MultiBoxDetection的辅助函数, 处理batch中的一个
        Args:
            c_p: (预测总类别数+1, 锚框个数)
            l_p: (锚框个数*4, )
            anc: (锚框个数, 4)
            nms_threshold: 非极大抑制中的阈值
        Return:
            output: (锚框个数, 6)
        """
        pred_bb_num = c_p.shape[1]
        anc = (anc + l_p.view(pred_bb_num, 4)).detach().cpu().numpy() # 加上偏移量

        confidence, class_id = torch.max(c_p, 0)
        confidence = confidence.detach().cpu().numpy()
        class_id = class_id.detach().cpu().numpy()

        pred_bb_info = [Pred_BB_Info(
                            index = i,
                            class_id = class_id[i] - 1, # 正类label从0开始
                            confidence = confidence[i],
                            xyxy=[*anc[i]]) # xyxy是个列表
                        for i in range(pred_bb_num)]

        # 正类的index
        obj_bb_idx = [bb.index for bb in non_max_suppression(pred_bb_info,
nms_threshold)]

        output = []
        for bb in pred_bb_info:
            output.append([
```

```
                (bb.class_id if bb.index in obj_bb_idx else -1.0),
                bb.confidence,
                *bb.xyxy
            ])

        return torch.tensor(output) # shape: (锚框个数, 6)

    batch_output = []
    for b in range(bn):
        batch_output.append(MultiBoxDetection_one(cls_prob[b], loc_pred[b],
anchor[0], nms_threshold))

    return torch.stack(batch_output)
```

注：torch.max(input_tensor, dim = i)对input_tensor在第i维上排序，返回第i维上的最大值和它在第i维上的索引

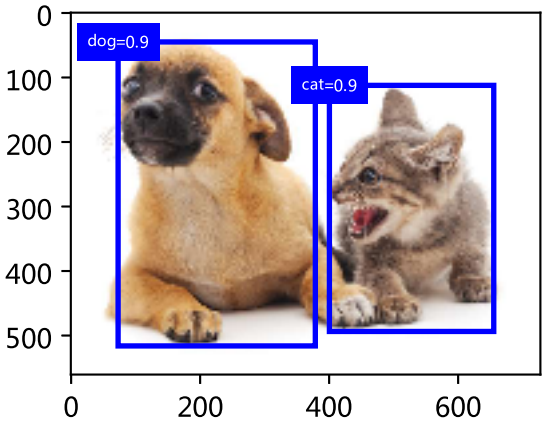
e.g.

```
input_tensor = torch.tensor([[0., 0., 0., 0.,], [0.9, 0.8, 0.7, 0.1], [0.1, 0.2,
0.3, 0.9]])
a, b = torch.max(input_tensor, 0)
print(a)
>>> tensor([0.9000, 0.8000, 0.7000, 0.9000])
print(b)
>>> tensor([1, 1, 1, 2])
```

运行MultiBoxDetection函数并设阈值为0.5。这里为输入都增加了样本维。可以看到，返回的结果的形状为(批量大小, 锚框个数, 6)。其中每一行的6个元素代表同一个预测边界框的输出信息。第一个元素是索引从0开始计数的预测类别(0为狗，1为猫)，其中-1表示背景或在非极大值抑制中被移除。第二个元素是预测边界框的置信度。剩余的4个元素分别是预测边界框左上角的x和y轴坐标以及右下角的x和y轴坐标(值域在0到1之间)。

```
output = MultiBoxDetection(
    cls_probs.unsqueeze(dim=0), offset_preds.unsqueeze(dim=0),
    anchors.unsqueeze(dim=0), nms_threshold=0.5)
output
>>> tensor([[[ 0.0000,  0.9000,  0.1000,  0.0800,  0.5200,  0.9200],
         [-1.0000,  0.8000,  0.0800,  0.2000,  0.5600,  0.9500],
         [-1.0000,  0.7000,  0.1500,  0.3000,  0.6200,  0.9100],
         [ 1.0000,  0.9000,  0.5500,  0.2000,  0.9000,  0.8800]]])
```

```
fig = d2l.plt.imshow(img)
for i in output[0].detach().cpu().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)
```



实践中，我们可以在执行非极大值抑制前将置信度较低的预测边界框移除，从而减小非极大值抑制的计算量。我们还可以筛选非极大值抑制的输出，例如，只保留其中置信度较高的结果作为最终输出。

小结

- 以每个像素为中心，生成多个大小和宽高比不同的锚框
- 交并比是两个边界框相交面积与相并面积之比
- 在训练集中，为每个锚框标注两类标签：一是锚框所含目标的类别；二是真实边界框相对锚框的偏移量
- 预测时，可以使用非极大值抑制来移除相似的预测边界框，从而令结果简洁

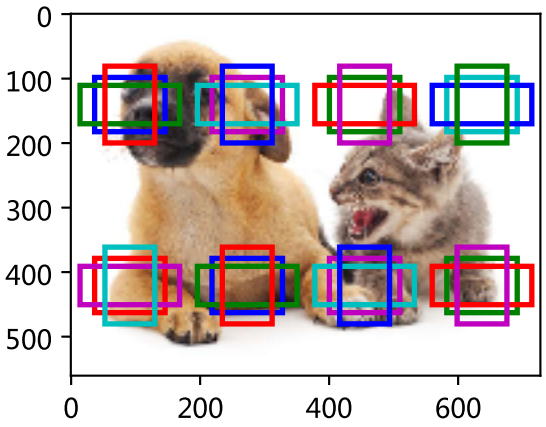
多尺度目标检测

在上节(锚框)中，以输入图像的每个像素为中心生成多个锚框。这些锚框是对输入图像不同区域的采样。然而，如果以图像每个像素为中心都生成锚框，很容易生成过多锚框而造成计算量过大。举个例子，假设输入图像的高和宽分别为561像素和728像素，如果以每个像素为中心生成5个不同形状的锚框，那么一张图像上则需要标注并预测**200**多万个锚框($561 \times 728 \times 5$)。
减少锚框个数并不难。一种简单的方法是在输入图像中均匀采样一小部分像素，并以采样的像素为中心生成锚框。此外，在不同尺度下，可以生成不同数量和不同大小的锚框。值得注意的是，较小目标比较大目标在图像上出现位置的可能性更多。举个简单的例子：形状为1×1、1×2和2×2的目标在形状为2×2的图像上可能出现的位置分别有4、2和1种。因此，当使用较小锚框来检测较小目标时，可以采样较多的区域；而当使用较大锚框来检测较大目标时，可以采样较少的区域。
为了演示如何多尺度生成锚框，下面先读取一张图像。它的高和宽分别为561像素和728像素。

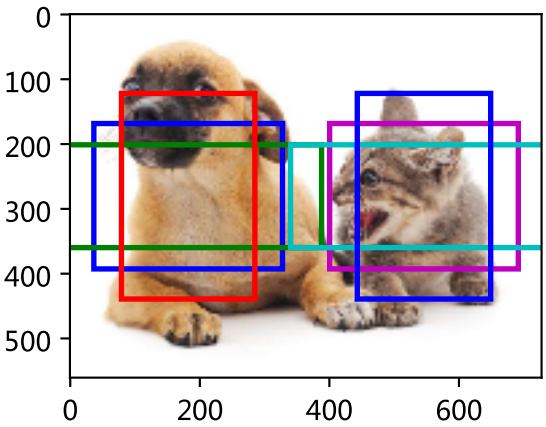
```
d2l.set_figsize()

def display_anchors(fmap_w, fmap_h, s):
    # 前两维的取值不影响输出结果
    fmap = torch.zeros((1, 10, fmap_h, fmap_w), dtype=torch.float32)

    # 平移所有锚框使均匀分布在图片上
    offset_x, offset_y = 1.0/fmap_w, 1.0/fmap_h
    anchors = d2l.MultiBoxPrior(fmap, sizes=s, ratios=[1, 2, 0.5]) + \
        torch.tensor([offset_x/2, offset_y/2, offset_x/2, offset_y/2])

    bbox_scale = torch.tensor([[w, h, w, h]], dtype=torch.float32)
```

```
    d2l.show_bboxes(d2l.plt.imshow(img).axes,
                    anchors[0] * bbox_scale)
```

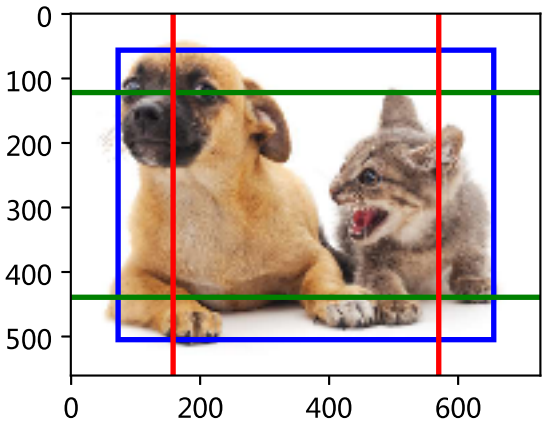
```
display_anchors(fmap_w=4, fmap_h=2, s=[0.15])
```




```
display_anchors(fmap_w=2, fmap_h=1, s=[0.4])
```

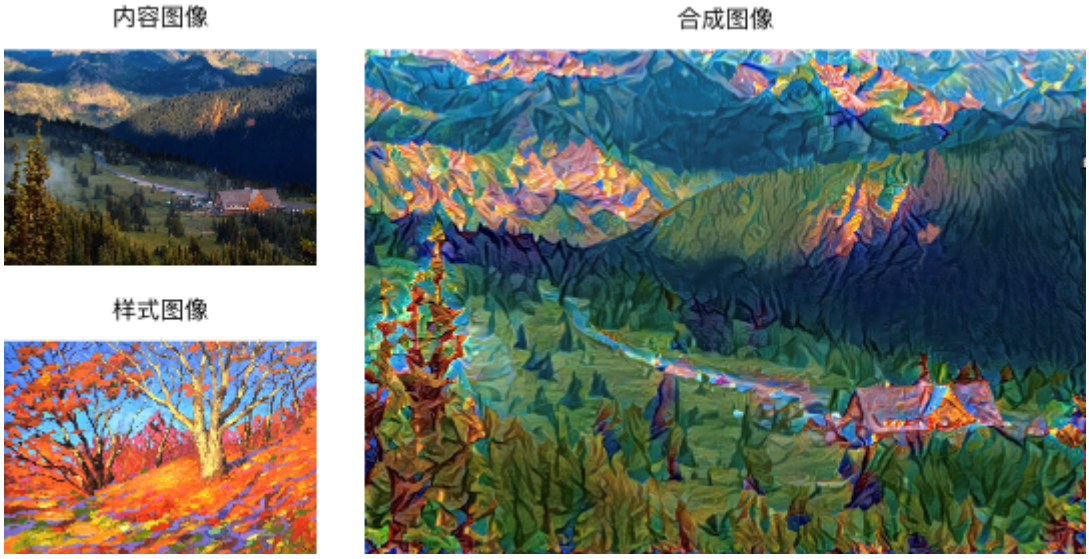



```
display_anchors(fmap_w=1, fmap_h=1, s=[0.8])
```



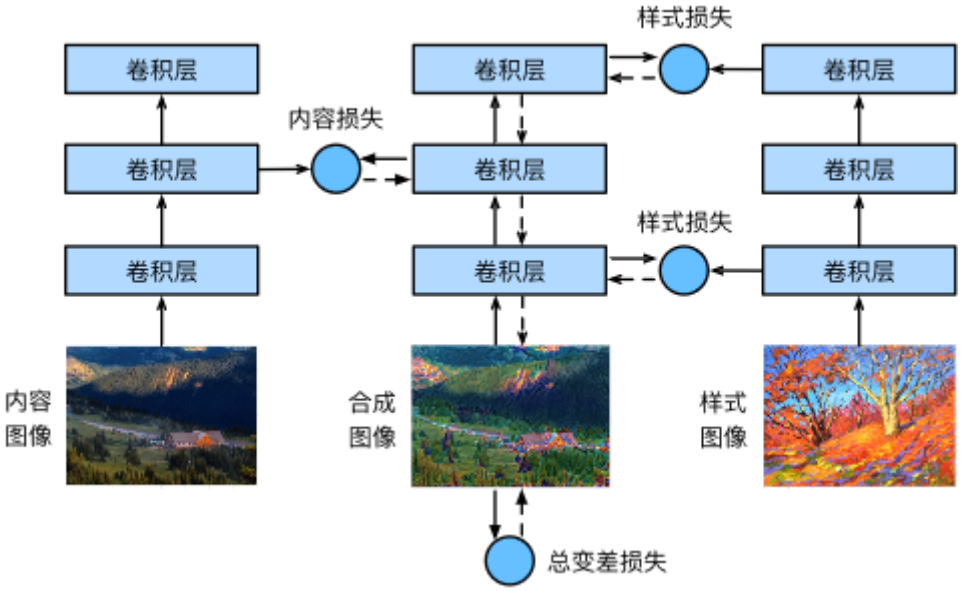
20 - 图像风格迁移

本节将介绍如何使用卷积神经网络自动将某图像中的样式应用在另一图像之上，即样式迁移(style transfer)。这里需要两张输入图像，一张是内容图像，另一张是样式图像，然后使用神经网络修改内容图像使其在样式上接近样式图像。下图中的内容图像是在西雅图郊区的雷尼尔山国家公园(Mount Rainier National Park)拍摄的风景照，而样式图像则是一副主题为秋天橡树的油画。最终输出的合成图像在保留了内容图像中物体主体形状的情况下应用了样式图像的油画笔触，同时也让整体颜色更加鲜艳。



方法

下用一个例子来阐述基于卷积神经网络的样式迁移方法。首先初始化合成图像，例如将其初始化成内容图像。该合成图像是样式迁移过程中唯一需要更新的变量，即样式迁移所需迭代的模型参数。然后选择一个预训练的卷积神经网络来抽取图像的特征，其中的模型参数在训练中无须更新。深度卷积神经网络凭借多个层逐级抽取图像的特征。可以选择其中某些层的输出作为内容特征或样式特征。以下图为例，这里选取的预训练的神经网络含有3个卷积层，其中第2层输出图像的内容特征，而第1层和第3层的输出被作为图像的样式特征。接下来通过正向传播(实线箭头方向)计算样式迁移的损失函数，并通过反向传播(虚线箭头方向)迭代模型参数，即不断更新合成图像。样式迁移常用的损失函数由3部分组成：内容损失(content loss)使合成图像与内容图像在内容特征上接近，样式损失(style loss)让合成图像与样式图像在样式特征上接近，而总变差损失(total variation loss)则有助于减少合成图像中的噪点。最后，当模型训练结束时，输出样式迁移的模型参数，即得到最终的合成图像。



具体实现

读取内容图像和样式图像

内容图像和样式图像的尺寸并不一样

预处理和后处理图像

下面定义了图像的预处理函数和后处理函数。预处理函数`preprocess`对输入图像在`RGB`三个通道分别做标准化，并将结果变换成卷积神经网络接受的输入格式。后处理函数`postprocess`则将输出图像中的像素值还原回标准化之前的值。由于图像打印函数要求每个像素的浮点数值在`0`到`1`之间，使用`clamp`函数对小于`0`和大于`1`的值分别取`0`和`1`。

```
rgb_mean = np.array([0.485, 0.456, 0.406])
rgb_std = np.array([0.229, 0.224, 0.225])

def preprocess(PIL_img, image_shape):
    process = torchvision.transforms.Compose([
        torchvision.transforms.Resize(image_shape),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mean=rgb_mean, std=rgb_std)])

    return process(PIL_img).unsqueeze(dim = 0) # (batch_size, 3, H, W)

def postprocess(img_tensor):
    inv_normalize = torchvision.transforms.Normalize(
        mean= -rgb_mean / rgb_std,
        std= 1/rgb_std)
    to_PIL_image = torchvision.transforms.ToPILImage()
    return to_PIL_image(inv_normalize(img_tensor[0].cpu()).clamp(0, 1))
```

抽取特征

使用基于ImageNet数据集预训练的VGG-19模型来抽取图像特征
为了抽取图像的内容特征和样式特征，可以选择VGG网络中某些层的输出。一般来说，越靠近输入层的输出越容易抽取图像的细节信息，反之则越容易抽取图像的全局信息。为了避免合成图像过多保留内容图像的细节，选择VGG较靠近输出的层，也称内容层，来输出图像的内容特征。同时还从VGG中选择不同层的输出来匹配局部和全局的样式，这些层也叫样式层。在“使用重复元素的网络(VGG)”一节中曾介绍过，VGG网络使用了5个卷积块。实验中选择第四卷积块的最后一个卷积层作为内容层，以及每个卷积块的第一个卷积层作为样式层。这些层的索引可以通过打印pretrained_net实例来获取。

```
style_layers, content_layers = [0, 5, 10, 19, 28], [25]
```

在抽取特征时只需要用到VGG从输入层到最靠近输出层的内容层或样式层之间的所有层。下面构建一个新的网络net，它只保留需要用到的VGG的所有层。之后将使用net来抽取特征。

```
net_list = []
for i in range(max(content_layers + style_layers) + 1):
    net_list.append(pretrained_net.features[i])
net = torch.nn.Sequential(*net_list)
```

给定输入X，如果简单调用前向计算net(X)，只能获得最后一层的输出。由于还需要中间层的输出，因此这里逐层计算，并保留内容层和样式层的输出。

```
def extract_features(X, content_layers, style_layers):
    contents = []
    styles = []
    for i in range(len(net)):
        X = net[i](X)
        if i in style_layers:
            styles.append(X)
        if i in content_layers:
            contents.append(X)
    return contents, styles
```

下面定义两个函数，其中get_contents函数对内容图像抽取内容特征，而get_styles函数则对样式图像抽取样式特征。因为在训练时无须改变预训练的VGG的模型参数，所以可以在训练开始之前就提取出内容图像的内容特征，以及样式图像的样式特征。因为合成图像是样式迁移所需迭代的模型参数，所以只能在训练过程中通过调用extract_features函数来抽取合成图像的内容特征和样式特征。

```
def get_contents(image_shape, device):
    content_X = preprocess(content_img, image_shape).to(device)
    contents_Y, _ = extract_features(content_X, content_layers, style_layers)
    return content_X, contents_Y

def get_styles(image_shape, device):
    style_X = preprocess(style_img, image_shape).to(device)
```

```
    _, styles_Y = extract_features(style_X, content_layers, style_layers)
    return style_X, styles_Y
```

定义损失函数

下面描述样式迁移的损失函数。它由内容损失、样式损失和总变差损失三部分组成。

内容损失

与线性回归中的损失函数类似，内容损失通过平方误差函数衡量合成图像与内容图像在内容特征上的差异。平方误差函数的两个输入均为extract_features函数计算所得到的内容层的输出。

```
def content_loss(Y_hat, Y):
    return F.mse_loss(Y_hat, Y)
```

样式损失

样式损失也一样通过平方误差函数衡量合成图像与样式图像在样式上的差异。为了表达样式层输出的样式，先通过extract_features函数计算样式层的输出。假设该输出的样本数为1，通道数为c，高和宽分别为h和w，可以把输出变换成c行hw列的矩阵$\boldsymbol{X}$。矩阵$\boldsymbol{X}$可以看作是由c个长度为hw的向量$\boldsymbol{x}_1, \ldots, \boldsymbol{x}_c$ 组成的。其中向量$\boldsymbol{x}_i$代表了通道i上的样式特征。这些向量的格拉姆矩阵(Gram matrix) $\boldsymbol{X}\boldsymbol{X}^\top \in \mathbb{R}^{c \times c}$ 中i行j列的元素x_{ij}即向量$\boldsymbol{x}_i$与$\boldsymbol{x}_j$的内积，它表达了通道i和通道j上样式特征的相关性。我们用这样的格拉姆矩阵表达样式层输出的样式。需要注意的是，当hw的值较大时，格拉姆矩阵中的元素容易出现较大的值。此外，格拉姆矩阵的高和宽皆为通道数c。为了让样式损失不受这些值的大小影响，下面定义的gram函数将格拉姆矩阵除以了矩阵中元素的个数，即chw。

```
def gram(X):
    num_channels, n = X.shape[1], X.shape[2] * X.shape[3]
    X = X.view(num_channels, n)
    return torch.matmul(X, X.t()) / (num_channels * n)
```

自然地，样式损失的平方误差函数的两个格拉姆矩阵输入分别基于合成图像与样式图像的样式层输出。这里假设基于样式图像的格拉姆矩阵gram_Y已经预先计算好了

```
def style_loss(Y_hat, gram_Y):
    return F.mse_loss(gram(Y_hat), gram_Y)
```

总变差损失

有时候，学习到的合成图像里面有大量高频噪点，即有特别亮或者特别暗的颗粒像素。一种常用的降噪方法是总变差降噪(total variation denoising)。假设$x_{i,j}$表示坐标为(i,j)的像素值，降低总变差损失

$$\sum_{i,j} |x_{i,j} - x_{i+1,j}| + |x_{i,j} - x_{i,j+1}|$$

能够尽可能使邻近的像素值相似。

```
def tv_loss(Y_hat):
    return 0.5 * (F.l1_loss(Y_hat[:, :, 1:, :], Y_hat[:, :, :-1, :]) +
                  F.l1_loss(Y_hat[:, :, :, 1:], Y_hat[:, :, :, :-1]))
```

损失函数

样式迁移的损失函数即内容损失、样式损失和总变差损失的加权和。通过调节这些权值超参数，可以权衡合成图像在保留内容、迁移样式以及降噪三方面的相对重要性。

```
content_weight, style_weight, tv_weight = 1, 1e3, 10

def compute_loss(X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram):
    # 分别计算内容损失、样式损失和总变差损失
    contents_l = [content_loss(Y_hat, Y) * content_weight for Y_hat, Y in zip(
        contents_Y_hat, contents_Y)]
    styles_l = [style_loss(Y_hat, Y) * style_weight for Y_hat, Y in zip(
        styles_Y_hat, styles_Y_gram)]
    tv_l = tv_loss(X) * tv_weight
    # 对所有损失求和
    l = sum(styles_l) + sum(contents_l) + tv_l
    return contents_l, styles_l, tv_l, l
```

创建和初始化合成图像

在样式迁移中，合成图像是唯一需要更新的变量。因此可以定义一个简单的模型`GeneratedImage`，并将合成图像视为模型参数。模型的前向计算只需返回模型参数即可。

```
class GeneratedImage(torch.nn.Module):
    def __init__(self, img_shape):
        super(GeneratedImage, self).__init__()
        self.weight = torch.nn.Parameter(torch.rand(*img_shape))

    def forward(self):
        return self.weight
```

下面定义`get_inits`函数。该函数创建了合成图像的模型实例，并将其初始化为图像`X`。样式图像在各个样式层的格拉姆矩阵`styles_Y_gram`将在训练前预先计算好。

```
def get_inits(X, device, lr, styles_Y):
    gen_img = GeneratedImage(X.shape).to(device)
    gen_img.weight.data = X.data
    optimizer = torch.optim.Adam(gen_img.parameters(), lr=lr)
```

```
    styles_Y_gram = [gram(Y) for Y in styles_Y]
    return gen_img(), styles_Y_gram, optimizer
```

训练

在训练模型时，不断抽取合成图像的内容特征和样式特征，并计算损失函数。

```
def train(X, contents_Y, styles_Y, device, lr, max_epochs, lr_decay_epoch):
    print("training on ", device)
    X, styles_Y_gram, optimizer = get_inits(X, device, lr, styles_Y)
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, lr_decay_epoch,
gamma=0.1)
    for i in range(max_epochs):
        start = time.time()

        contents_Y_hat, styles_Y_hat = extract_features(
                X, content_layers, style_layers)
        contents_l, styles_l, tv_l, l = compute_loss(
                X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram)

        optimizer.zero_grad()
        l.backward(retain_graph = True)
        optimizer.step()
        scheduler.step()

        if i % 50 == 0 and i != 0:
            print('epoch %3d, content loss %.2f, style loss %.2f, '
                  'TV loss %.2f, %.2f sec'
                  % (i, sum(contents_l).item(), sum(styles_l).item(), tv_l.item(),
                     time.time() - start))
    return X.detach()
```

合成的图片

合成图像里面不仅有大块的类似样式图像的油画色彩块，色彩块中甚至出现了细微的纹理。

总结

- 样式迁移常用的损失函数由3部分组成：内容损失使合成图像与内容图像在内容特征上接近，样式损失令合成图像与样式图像在样式特征上接近，而总变差损失则有助于减少合成图像中的噪点。
- 可以通过预训练的卷积神经网络来抽取图像的特征，并通过最小化损失函数来不断更新合成图像。
- 用格拉姆矩阵表达样式层输出的样式。

21 - 图像分类案例1

Kaggle上的图像分类（CIFAR-10)

运用在前面几节中学到的知识来参加Kaggle竞赛，该竞赛解决了CIFAR-10图像分类问题

获取和组织数据集

比赛数据分为训练集和测试集。训练集包含50,000张图片。测试集包含300,000张图片。两个数据集中的图像格式均为PNG，高度和宽度均为32像素，并具有三个颜色通道(RGB)。图像涵盖10个类别：飞机，汽车，鸟类，猫，鹿，狗，青蛙，马，船和卡车。为了更容易上手，这里使用上述数据集的小样本。`train_tiny.zip`包含80个训练样本，而`test_tiny.zip`包含100个测试样本。它们的未压缩文件夹名称分别是`train_tiny`和`test_tiny`。

图像增强

包括图像裁剪和随机翻转

导入数据集

定义模型

`ResNet-18`网络结构：`ResNet`全名`Residual Network`(残差网络)。`Kaiming He`的`Deep Residual Learning for Image Recognition`获得了`CVPR`最佳论文。他提出的深度残差网络在2015年可以说是洗刷了图像方面的各大比赛，以绝对优势取得了多个比赛的冠军。而且它在保证网络精度的前提下，将网络的深度达到了152层，后来又进一步加到1000的深度。

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

训练和测试