

```

179     with tf.Session() as sess:
180         sess.run(load_ops)
181
182         detected_boxes = sess.run(boxes, feed_dict={inputs:
183 [np.array(img_resized, dtype=np.float32)]})
184
185         #对 10647 个预测框进行去重
186         filtered_boxes = non_max_suppression(detected_boxes,
187         confidence_threshold=conf_threshold,
188         iou_threshold=iou_threshold)
189
190         draw_boxes(filtered_boxes, img, classes, (size, size))
191
192     img.save(output_img)
193
194 if __name__ == '__main__':
195     main()

```

代码第 165 行，先将输入的图片统一成固定大小，然后放入模型中进行识别，再将最终的结果画到图片上，并保存起来（见代码第 189 行）。

## 8.5.12 运行程序

在本地代码文件下随便放一张图片（例如“timg.jpg”）。运行代码之后，生成以下结果：

```
wine glass 95.74% [257.179009107443, 120.12802956654475]
wine glass 95.74% [352.22361010771533, 128.20337944764358]
bowl 93.57% [419.31336153470556, 222.11435362008902]
bowl 93.57% [166.07221649243283, 233.63491428815402]
banana 52.60% [560.0561892436101, 198.93592790456918]
apple 80.51% [478.8216531460102, 221.5187714283283]
```

结果中的每一行都分为 3 部分，代表着所识别出来的物体：

- 类别名称。
- 置信度（所属类别的评分）。
- 类别对应的坐标。

同时，会在本地目录下生成名为“out.jpg”的图片，如图 8-19 所示。

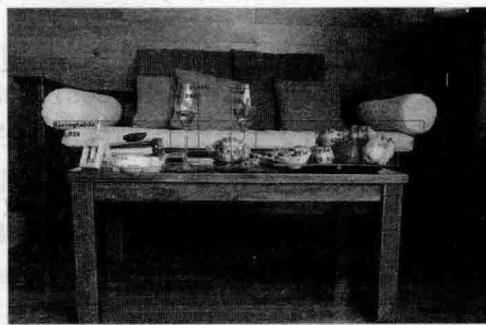


图 8-19 YOLO V3 模型的识别结果

## 8.6 实例 44：用 YOLO V3 模型识别门牌号

本节将用自定义数据集训练 YOLO V3 模型，并用训练好的模型进行目标识别。

### 实例描述

准备一个带有门牌号图片的数据集，里面含有具体图片和与图片上具体门牌数字的位置标注。这个数据集训练 YOLO V3 模型，让模型能够识别图中门牌的数字内容及坐标。

本实例是在动态图框架中用 `tf.keras` 接口来实现的。数据集使用的是 SVHN (Street View House Numbers, 街道门牌号码) 数据集。加载预训练模型，并在其基础上进行二次训练。下面讲解具体操作。

### 8.6.1 工程部署：准备样本

SVHN 数据集是斯坦福大学发布的一个真实图像数据集。该数据集的作用类似于 MNIST，在图像算法领域经常使用。具体下载地址：

<http://ufldl.stanford.edu/housenumbers/>

在目标识别任务中，光有图片是不够的。例如 COCO 数据集，每张图片都有对应的标注信息。在随书的配套资源里，也为每张 SVHN 图片提供了对应的标注文件（配套资源中提供的样本量不多，只是为了演示案例），其格式与对应关系如图 8-20 所示。

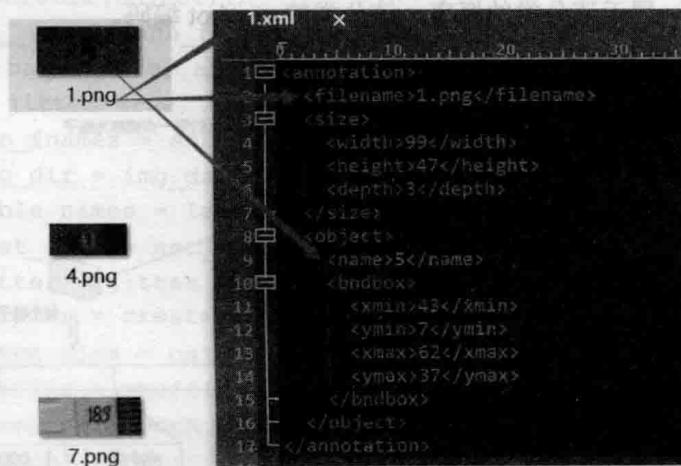


图 8-20 样本与标注

如图 8-20 所示，每张图片对应一个与其同名的 XML 文档。该文档里会放置图片的尺寸数据（高、宽），以及内容（例如图中的数字 5）对应的位置坐标。

### 8.6.2 代码实现：读取样本数据，并制作标签

本小节分为两步实现：读取样本与制作标签。

## 1. 读取样本

读取原始样本数据的代码是在代码文件“8-13 annotation.py”中实现的。该代码主要通过 `parse_annotation` 函数解析 XML 文档，并返回图片与内容的对应关系。例如，其返回值为：

```
G:/python3/8-20 yolov3numbers\data\img\9.png      #图片文件路径
[[27 8 39 26]                                     #图片中的坐标、高、宽
 [40 5 53 23]
 [52 7 67 25]]
[1, 4, 4]                                         #图片中的数字
```

该文件中的代码功能单一，可以直接被当作工具使用，不需要过多研究。

## 2. 制作标签

该步骤需要将原始数据转为 YOLO V3 模型需要的标签格式。YOLO V3 模型中的标签格式是与内部模型结构相关的，具体描述如下：

- YOLO V3 模型的标签由 3 个矩阵组成。
- 3 个矩阵的高、宽分别与 YOLO V3 模型的 3 个输出尺度相同。
- 每种尺度的矩阵对应 3 个候选框。
- 矩阵在高、宽维度上的每个点被称为格子。
- 每个格子中有 3 个同样的结构，对应所在矩阵的 3 个候选框。
- 每个结构中的内容都是候选框信息。
- 每个候选框信息的内容包括中心点坐标、高（相对候选框的缩放值）、宽（相对候选框的缩放值）、属于该分类的概率、该分类的 one-hot 编码。

整体结构如图 8-21 所示。

将原始图片缩小为 3 个尺寸的矩阵，每个矩阵对应 3 个候选框，矩阵上的每一点被称为格子

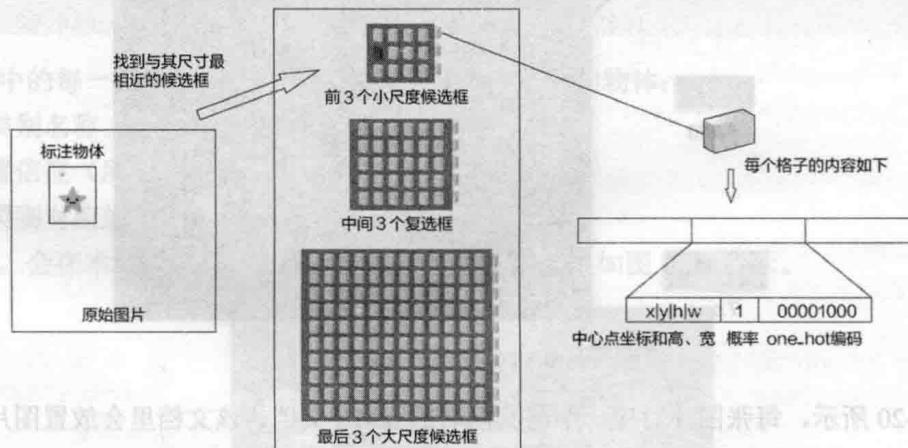


图 8-21 制作 YOLO V3 的样本标签

从图 8-21 中的结构可以看出，3 个不同尺度的矩阵分别存放原始图片中不同大小的标注物体。矩阵中的格子，可以理解为是原图像中对应区域的映射。

具体实现的代码文件为“8-14 generator.py”中的 `BatchGenerator` 类。其步骤如下：

(1) 根据原始图片，构造3个矩阵当作放置标签的容器（如图8-21中间的3个方块），并向这3个矩阵填充0作为初始值。见代码第67行的\_create\_empty\_xy函数。

(2) 根据标注中物体的高、宽尺寸，在候选框中找到最接近的框。见代码第96行\_find\_match\_anchor函数。

(3) 根据\_find\_match\_anchor函数返回的候选框索引，可以定位对应的矩阵。调用函数\_encode\_box，计算物体在该矩阵上的中心点位置，以及自身尺寸相对于该候选框的缩放比例。见代码第57行。

(4) 调用\_assign\_box函数，根据最相近的候选框索引定位到格子里的具体结构，并将步骤

(3) 算出来的值与分类信息填入，见代码第58行。

完整代码如下：

#### 代码8-14 generator

```

01 import numpy as np
02 from random import shuffle
03 annotation = __import__("8-13_annotation")
04 parse_annotation = annotation.parse_annotation
05 ImgAugment = annotation.ImgAugment
06 box = __import__("8-15_box")
07 find_match_box = box.find_match_box
08 DOWNSAMPLE_RATIO = 32
09 def find_match_box(annotation, anchors, net_size=416):
10     class BatchGenerator(object):
11         def __init__(self, ann_fnames, img_dir, labels,
12                      batch_size, anchors, net_size=416,
13                      jitter=True, shuffle=True):
14             self.ann_fnames = ann_fnames
15             self.img_dir = img_dir
16             self.labels = labels
17             self.net_size = net_size
18             self.jitter = jitter
19             self.anchors = create_anchor_boxes(anchors) #按照候选框尺寸生成坐标
20             self.batch_size = batch_size
21             self.shuffle = shuffle
22             self.steps_per_epoch = int(len(ann_fnames) / batch_size)
23             self._epoch = 0
24             self._end_epoch = False
25             self._index = 0
26
27         def next_batch(self):
28             xs, ys_1, ys_2, ys_3 = [], [], [], []
29             for _ in range(self.batch_size): #按照指定的批次获取样本数据，并做成标签
30                 x, y1, y2, y3 = self._get()
31                 xs.append(x)
32                 ys_1.append(y1)
33                 ys_2.append(y2)

```

```

34         ys_3.append(y3)
35     if self._end_epoch == True:
36         if self.shuffle:
37             shuffle(self.ann_fnames)
38         self._end_epoch = False
39         self._epoch += 1
40     return np.array(xs).astype(np.float32),
41           np.array(ys_1).astype(np.float32), np.array(ys_2).astype(np.float32),
42           np.array(ys_3).astype(np.float32)
43
44     def _get(self):          # 获取一条样本数据并做成标签
45         net_size = self._net_size
46         # 解析标注文件
47         fname, boxes, coded_labels =
48             parse_annotation(self.ann_fnames[self._index], self.img_dir,
49             self.label_names)
50
51         # 读取图片，并按照设置修改图片的尺寸
52         img_augmenter = ImgAugment(net_size, net_size, self.jitter)
53         img, boxes_ = img_augmenter.imread(fname, boxes)
54
55         # 生成 3 种尺度的格子
56         list_ys = _create_empty_xy(net_size, len(self.label_names))
57         for original_box, label in zip(boxes_, coded_labels):
58             # 在 anchors 中，找到与其面积区域最匹配的候选框 max_anchor、对应的尺度索引、
59             # 该尺度下的第几个锚点
60             max_anchor, scale_index, box_index =
61                 _find_match_anchor(original_box, self.anchors)
62             # 计算在对应尺度上的中心点坐标，以及对应候选框的长宽缩放比例
63             _coded_box = _encode_box(list_ys[scale_index], original_box,
64             max_anchor, net_size, net_size)
65             _assign_box(list_ys[scale_index], box_index, _coded_box, label)
66
67         self._index += 1
68         if self._index == len(self.ann_fnames):
69             self._index = 0
70             self._end_epoch = True
71     return img/255., list_ys[2], list_ys[1], list_ys[0]
72
73     # 初始化标签
74     def _create_empty_xy(net_size, n_classes, n_boxes=3):
75         # 获得最小矩阵格子
76         base_grid_h, base_grid_w = net_size//DOWNSAMPLE_RATIO,
77         net_size//DOWNSAMPLE_RATIO
78         # 初始化 3 种不同尺度的矩阵，用于存放标签
79         ys_1 = np.zeros((1*base_grid_h, 1*base_grid_w, n_boxes, 4+1+n_classes))
80         ys_2 = np.zeros((2*base_grid_h, 2*base_grid_w, n_boxes, 4+1+n_classes))

```

```

73     ys_3 = np.zeros((4*base_grid_h, 4*base_grid_w, n_boxes, 4+1+n_classes))
74     list_ys = [ys_3, ys_2, ys_1]
75     return list_ys
76
77 def _encode_box(yolo, original_box, anchor_box, net_w, net_h):
78     x1, y1, x2, y2 = original_box
79     _, _, anchor_w, anchor_h = anchor_box
80     #取出格子在高和宽方向上的个数
81     grid_h, grid_w = yolo.shape[:2]
82
83     #根据原始图片到当前矩阵的缩放比例，计算当前矩阵中物体的中心点坐标
84     center_x = .5*(x1 + x2)
85     center_x = center_x / float(net_w) * grid_w
86     center_y = .5*(y1 + y2)
87     center_y = center_y / float(net_h) * grid_h
88
89     #计算物体相对于候选框的尺寸缩放值
90     w = np.log(max((x2 - x1), 1) / float(anchor_w))
91     h = np.log(max((y2 - y1), 1) / float(anchor_h))
92     box = [center_x, center_y, w, h]#将中心点和缩放值打包返回
93     return box
94
95 #找到与物体尺寸最接近的候选框
96 def _find_match_anchor(box, anchor_boxes):
97     x1, y1, x2, y2 = box
98     shifted_box = np.array([0, 0, x2-x1, y2-y1])
99     max_index = find_match_box(shifted_box, anchor_boxes)
100    max_anchor = anchor_boxes[max_index]
101    scale_index = max_index // 3
102    box_index = max_index%3
103    return max_anchor, scale_index, box_index
104 #将具体的值放到标签矩阵里，作为真正的标签
105 def _assign_box(yolo, box_index, box, label):
106     center_x, center_y, _, _ = box
107     #向下取整，得到的就是格子的索引
108     grid_x = int(np.floor(center_x))
109     grid_y = int(np.floor(center_y))
110     #填入所计算的数值，作为标签
111     yolo[grid_y, grid_x, box_index] = 0.
112     yolo[grid_y, grid_x, box_index, 0:4] = box
113     yolo[grid_y, grid_x, box_index, 4] = 1.
114     yolo[grid_y, grid_x, box_index, 5+label] = 1.
115
116 def create_anchor_boxes(anchors):  #将候选框变为box
117     boxes = []
118     n_boxes = int(len(anchors)/2)
119     for i in range(n_boxes):

```

```

120     boxes.append(np.array([0, 10, anchors[2*i], anchors[2*i+1]]))
121     return np.array(boxes)

```

代码第 10 行定义了 BatchGenerator 类，用来实现数据集的输入功能。在实际使用时，可以用 BatchGenerator 类的 next\_batch 方法（见代码第 27 行）来获取一批次的输入样本和标签数据。

在 next\_batch 方法中，用 \_get 函数读取样本和转化标注（见代码第 30 行）。

代码第 90 行是计算物体相对于候选框的尺寸缩放值。代码解读如下：

(1) “ $x_2 - x_1$ ” 代表计算该物体的宽度。

(2) 在其外层又加了一个 max 函数，取 “ $x_2 - x_1$ ” 和 1 中更大的那个值。



### 提示：

代码第 90 行中的 max 函数可以保证计算出的宽度值永远大于 1，这样可以增强程序的健壮性。

## 8.6.3 代码实现：用 tf.keras 接口构建 YOLO V3 模型，并计算损失

用 tf.keras 接口构建 YOLO V3 模型，并计算模型的输出结果与标签（见 8.6.2 小节）之间的 loss 值，训练模型。

### 1. 构建 YOLO V3 模型

在本书的配套资源里有代码文件“8-15\_box.py”，该文件实现了 YOLO V3 模型中边框处理相关的功能，可以被当作工具代码使用。

YOLO V3 模型分为 4 个代码文件来完成，具体如下。

- “8-16\_darknet53.py”：实现了 Darknet-53 模型的构建。
- “8-17\_yolohead.py”：实现了 YOLO V3 模型多尺度特征融合部分的构建。
- “8-18\_yolov3.py”：实现 YOLO V3 模型的构建。
- “8-19\_weights.py”：实现加载 YOLO V3 的预训练模型功能。

在代码文件“8-18\_yolov3.py”中定义了 Yolonet 类，用来实现 YOLO V3 模型的网络结构。Yolonet 类在对原始图片进行计算之后，会输出一个含有 3 个矩阵的列表，该列表的结构与 8.6.2 小节中的标签结构一致。

YOLO V3 模型的正向网络结构在 8.5 节已经介绍，这里不再详细说明。

### 2. 计算值

YOLO V3 模型的输出结构与样本标签一致，都是一个含有 3 个矩阵的列表。在计算值时，需要对这 3 个矩阵依次计算 loss 值，并将每个矩阵的 loss 值结果相加再开平方得到最终结果，见代码第 118 行的 loss\_fn 函数。

定义函数 loss\_fn，用来计算损失值（见代码 118 行）。在函数 loss\_fn 中，具体的计算步骤如下：

(1) 遍历 YOLO V3 模型的预测列表与样本标签列表（如图 8-21 的中间部分所示，列表中一共有 3 个矩阵）。

- (2) 从两个列表(预测列表和标签列表)中取出对应的矩阵。
- (3) 将取出的矩阵和对应的候选框一起传入 lossCalculator 函数中进行 loss 值计算。
- (4) 重复第(2)步和第(3)步,依次对列表中的每个矩阵进行 loss 值计算。
- (5) 将每个矩阵的 loss 值结果相加,再开平方,得到最终结果。

具体代码如下:

### 代码 8-20 yololoss

```

1  import tensorflow as tf
2
3  def _create_mesh_xy(batch_size, grid_h, grid_w, n_box): #生成带序号的网格
4      mesh_x = tf.cast(tf.reshape(tf.tile(tf.range(grid_w), [grid_h]), (1,
5          grid_h, grid_w, 1, 1)),tf.float)
6      mesh_y = tf.transpose(mesh_x, (0,2,1,3,4))
7      mesh_xy = tf.tile(tf.concat([mesh_x,mesh_y],-1), [batch_size, 1, 1,
8          n_box, 1])
9      return mesh_xy
10
11
12  def adjust_pred_tensor(y_pred):#将网格信息融入坐标,置信度做 sigmoid 运算,并重新
13      grid_offset = _create_mesh_xy(*y_pred.shape[:4])
14      pred_xy    = grid_offset + tf.sigmoid(y_pred[..., :2]) #计算该尺度矩阵上
15      的坐标 sigma(t_xy) + c_xy
16      pred_wh   = y_pred[..., 2:4] #取出预测物体的尺寸 t_wh
17      pred_conf  = tf.sigmoid(y_pred[..., 4]) #对分类概率(置信度)做 sigmoid 转化
18      pred_classes = y_pred[..., 5:] #取出分类结果
19      #重新组合
20      preds = tf.concat([pred_xy, pred_wh, tf.expand_dims(pred_conf, axis=-1),
21          pred_classes], axis=-1)
22      return preds
23
24
25  def _create_mesh_anchor(anchors, batch_size, grid_h, grid_w, n_box):
26      mesh_anchor = tf.tile(anchors, [batch_size*grid_h*grid_w])
27      mesh_anchor = tf.reshape(mesh_anchor, [batch_size, grid_h, grid_w, n_box,
28          2]) #每个候选框有两个值
29      mesh_anchor = tf.cast(mesh_anchor, tf.float32)
30      return mesh_anchor
31
32  def conf_delta_tensor(y_true, y_pred, anchors, ignore_thresh):
33
34      pred_box_xy, pred_box_wh, pred_box_conf = y_pred[..., :2], y_pred[...,
35          2:4], y_pred[..., 4]
36      #创建带有候选框的格子矩阵
37      anchor_grid = _create_mesh_anchor(anchors, *y_pred.shape[:4])
38      true_wh = y_true[::,:,:,:,2:4]
39      true_wh = anchor_grid * tf.exp(true_wh)

```

```

33     true_wh = true_wh * tf.expand_dims(y_true[:, :, :, :, 4], 4) #还原真实尺寸
34     anchors_ = tf.constant(anchors, dtype='float',
35                             shape=[1, 1, 1, y_pred.shape[3], 2])          #y_pred.shape[3]是候选框个数
36     true_xy = y_true[:, :, :, :, 0:2]           #获取中心点
37     true_wh_half = true_wh / 2.                  #计算起始坐标
38     true_mins = true_xy - true_wh_half           #计算尾部坐标
39     true_maxes = true_xy + true_wh_half          #计算尾部坐标
40
41     pred_xy = pred_box_xy
42     pred_wh = tf.exp(pred_box_wh) * anchors_
43
44     pred_wh_half = pred_wh / 2.                  #计算起始坐标
45     pred_mins = pred_xy - pred_wh_half           #计算尾部坐标
46     pred_maxes = pred_xy + pred_wh_half          #计算尾部坐标
47
48     intersect_mins = tf.maximum(pred_mins, true_mins)
49     intersect_maxes = tf.minimum(pred_maxes, true_maxes)
50
51     #计算重叠面积
52     intersect_wh = tf.maximum(intersect_maxes - intersect_mins, 0.)
53     intersect_areas = intersect_wh[:, 0] * intersect_wh[:, 1]
54
55     true_areas = true_wh[:, 0] * true_wh[:, 1]
56     pred_areas = pred_wh[:, 0] * pred_wh[:, 1]
57
58     #计算不重叠面积
59     union_areas = pred_areas + true_areas - intersect_areas
60     best_ious = tf.truediv(intersect_areas, union_areas)    #计算 iou
61     #如 iou 小于阈值，则将其作为负向的 loss 值
62     conf_delta = pred_box_conf * tf.cast(best_ious < ignore_thresh, tf.float)
63
64     def wh_scale_tensor(true_box_wh, anchors, image_size):
65         image_size_ = tf.reshape(tf.cast(image_size, tf.float32), [1, 1, 1, 1, 2])
66         anchors_ = tf.constant(anchors, dtype='float', shape=[1, 1, 1, 3, 2])
67
68         #计算高和宽的缩放范围
69         wh_scale = tf.exp(true_box_wh) * anchors_ / image_size_
70         #物体尺寸占整个图片的面积比
71         wh_scale = tf.expand_dims(2 - wh_scale[:, 0] * wh_scale[:, 1], axis=4)
72
73     def loss_coord_tensor(object_mask, pred_box, true_box, wh_scale,
74                           xywh_scale): #计算基于位置的损失值：将 box 的差与缩放比相乘，所得的结果再进行平方和运算
75
76         xy_delta = object_mask * (pred_box - true_box) * wh_scale * xywh_scale
77
78         loss_xy = tf.reduce_sum(tf.square(xy_delta), list(range(1, 5)))

```

```

77     return loss_xy
78
79 def loss_conf_tensor(object_mask, pred_box_conf, true_box_conf, obj_scale,
80                      noobj_scale, conf_delta):
81     object_mask_ = tf.squeeze(object_mask, axis=-1)
82     #计算置信度 loss 值，分为正向与负向的之和
83     conf_delta_ = object_mask_* (pred_box_conf-true_box_conf) * obj_scale
84     + (1-object_mask_)* conf_delta * noobj_scale
85     #按照 1、2、3（候选框）归约求和，0 为批次
86     loss_conf = tf.reduce_sum(tf.square(conf_delta), list(range(1,4)))
87     return loss_conf
88
89 #分类损失直接用交叉熵
90 def loss_class_tensor(object_mask, pred_box_class, true_box_class,
91                       class_scale):
92     true_box_class_ = tf.cast(true_box_class, tf.int64)
93     class_delta = object_mask * \
94                 tf.expand_dims(tf.nn.softmax_cross_entropy_with_logits_v2(
95                             labels=true_box_class_, logits=pred_box_class), 4) * \
96                 class_scale
97
98     loss_class = tf.reduce_sum(class_delta, list(range(1,5)))
99     return loss_class
100
101 ignore_thresh=0.5      #小于该阈值的 box，被认为没有物体
102 grid_scale=1           #每个不同矩阵的总 loss 值缩放参数
103 obj_scale=5            #有物体的 loss 值缩放参数
104 noobj_scale=1           #没有物体的 loss 值缩放参数
105 xywh_scale=1           #坐标 loss 值缩放参数
106 class_scale=1           #分类 loss 值缩放参数
107
108 def lossCalculator(y_true, y_pred, anchors, image_size):
109     y_pred = tf.reshape(y_pred, y_true.shape) #统一形状
110
111     object_mask = tf.expand_dims(y_true[..., 4], 4) #取置信度
112     preds = adjust_pred_tensor(y_pred)           #将 box 与置信度数值变化后重新组合
113     conf_delta = conf_delta_tensor(y_true, preds, anchors, ignore_thresh)
114     wh_scale = wh_scale_tensor(y_true[..., 2:4], anchors, image_size)
115
116     loss_box = loss_coord_tensor(object_mask, preds[..., :4],
117                                   y_true[..., :4], wh_scale, xywh_scale)
118     loss_conf = loss_conf_tensor(object_mask, preds[..., 4], y_true[..., 4],
119                                 obj_scale, noobj_scale, conf_delta)
120     loss_class = loss_class_tensor(object_mask, preds[..., 5:], y_true[...,
121                                    5:], class_scale)
122
123     loss = loss_box + loss_conf + loss_class
124     return loss*grid_scale

```

```

117
118 def loss_fn(list_y_trues, list_y_preds, anchors, image_size):
119     inputanchors = [anchors[12:], anchors[6:12], anchors[:6]]
120     losses = [lossCalculator(list_y_trues[i], list_y_preds[i],
121         inputanchors[i], image_size) for i in range(len(list_y_trues)) ]
122     return tf.sqrt(tf.reduce_sum(losses)) #将3个矩阵的loss值相加再开平方

```

代码第 104 行，`lossCalculator` 函数用于计算预测结果中每个矩阵的 loss 值。`lossCalculator` 函数内部的计算步骤如下。

- (1) 定义掩码变量 `object_mask`：通过获取样本标签中的置信度值（有物体为 1，没物体为 0）来标识有物体和没有物体的两种情况（见代码第 107 行）。
- (2) 用 `loss_coord_tensor` 函数计算位置损失：计算标签位置与预测位置相差的平方。
- (3) 用 `loss_conf_tensor` 函数计算置信度损失：分别在有物体和没有物体的情况下，计算标签与预测置信度的差，并将二者的和进行平方。
- (4) 用 `loss_class_tensor` 函数计算分类损失：计算标签分类与预测分类的交叉熵。
- (5) 将第 (2)、(3)、(4) 的结果加起来，作为该矩阵的最终损失返回。

其中，在求其他的损失时只对有物体的情况进行计算。

代码第 112 行，在用 `loss_coord_tensor` 函数计算位置损失时传入了一个缩放值 `wh_scale`。该值代表标签中的物体尺寸在整个图像上的面积占比。

`wh_scale` 值是在函数 `wh_scale_tensor` 中计算的（见代码第 68 行）。具体步骤如下。

- (1) 对标签尺寸 `true_box_wh` 做 `tf.exp(true_box_wh) * anchors` 计算（`anchors` 为候选框的尺寸），得到了该物体的真实尺寸（该计算正好是 8.6.2 小节代码 90、91 行的逆运算）。
- (2) 用物体的真实尺寸除以 `image_size_`（`image_size_` 是图片的真实尺寸），得到物体在整个图上的面积占比。

在函数 `loss_conf_tensor` 中计算置信度损失是在代码第 82 行实现的，该代码解读如下。

- 前半部分：`object_mask_ * (pred_box_conf - true_box_conf) * obj_scale` 是有物体情况下置信度的 loss 值。
- 后半部分：`(1-object_mask_) * conf_delta * noobj_scale` 是没有物体情况下置信度的 loss 值。执行完“`1-object_mask_`”操作后，矩阵中没有物体的自信度字段都会变为 1，而 `conf_delta` 是由 `conf_delta_tensor` 得来的。在 `conf_delta_tensor` 中，先计算真实与预测框（`box`）的重叠度（IOU），并通过阈值来控制是否需要计算。如果低于阈值，就将其置信度纳入没有物体情况的 loss 值中来计算。

代码第 97~102 行，定义了训练中不同 loss 值的占比参数。这里将 `obj_scale` 设为 5，是让模型对有物体情况的置信度准确性偏大一些。在实际训练中，还可以根据具体的样本情况适当调整该值。

## 8.6.4 代码实现：在动态图中训练模型

在训练过程中，需要使用候选框和预训练文件。其中，候选框来自 COCO 数据集聚类后的结果；预训练文件与 8.5 节中使用的预训练文件一样。下面介绍具体细节。

## 1. 建立类信息，加载数据集

因为样本中的分类全部是数字，所以手动建立一个0~9的分类信息，见代码第27行。接着用BatchGenerator类实例化一个对象generator，作为数据集。具体代码如下：

代码 8-21 mainyolo

```

01 import os
02 import tensorflow as tf
03 import glob
04 from tqdm import tqdm
05 import cv2
06 import matplotlib.pyplot as plt
07 import tensorflow.contrib.eager as tfe
08 generator = __import__("8-14 generator")
09 BatchGenerator = generator.BatchGenerator
10 box = __import__("8-15 box")
11 draw_boxes = box.draw_boxes
12 yolov3 = __import__("8-18 yolov3")
13 YOLONet = yolov3.YOLONet
14 yoloLoss = __import__("8-20 yoloLoss")
15 loss_fn = yoloLoss.loss_fn
16
17 tf.enable_eager_execution()
18
19 PROJECT_ROOT = os.path.dirname(__file__) # 获取当前目录
20 print(PROJECT_ROOT)
21
22 # 定义 coco 锚点的候选框
23 COCO_ANCHORS = [10, 13, 16, 30, 33, 23, 30, 61, 62, 45, 59, 119, 116, 90, 156, 198,
24 373, 326]
25 # 定义预训练模型的路径
26 YOLOV3_WEIGHTS = os.path.join(PROJECT_ROOT, "yolov3.weights")
27 # 定义分类
28 LABELS = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
29 # 定义样本路径
30 ann_dir = os.path.join(PROJECT_ROOT, "data", "ann", "*.xml")
31 img_dir = os.path.join(PROJECT_ROOT, "data", "img")
32
33 train_ann_fnames = glob.glob(ann_dir) # 获取该路径下的 XML 文件
34
35 imgsize = 416 # 定义输入图片大小
36 batch_size = 2 # 定义批次
37 # 制作数据集
38 generator = BatchGenerator(train_ann_fnames, img_dir,
39 net_size=imgsize,
40 anchors=COCO_ANCHORS,
41 history=History(),
42 if loss_value == min_loss:
43     history.append(history.history)
44
45 if loss_value - min_loss < 0.001:
46     break

```

```

41         batch_size=2,
42         labels=LABELS,
43         jitter = False) #随机变化尺寸，数据增强

```

代码第 35 行，定义图片的输入尺寸为 416 pixel×416 pixel。这个值必须大于 COCO\_ANCHORS 中的最大候选框，否则候选框没有意义。

由于使用了 COCO 数据集的候选框，所以在选择输入尺寸时，尽量也使用与 COCO 数据集中训练的 YOLOV3 模型一致的输入尺寸。这样会有相对较好的训练效果。



### 提示：

在实例中，直接用 COCO 数据集的候选框作为模型的候选框，这么做只是为了演示方便。在实际训练中，为了得到更好的精度，建议使用训练数据集聚类后的结果作为模型的候选框。

## 2. 定义模型及训练参数

定义两个循环处理函数：

- `_loop_validation` 函数用于循环所有数据集，进行模型的验证。
- `_loop_train` 函数用于对全部的训练数据集进行训练。

为了演示方便，这里只用一个数据集，既做验证用，也做训练用。具体代码如下：

代码 8-21 mainyolo（续）

```

44 learning_rate = 1e-4           # 定义学习率
45 num_epochs = 85                # 定义迭代次数
46 save_dir = "./model"           # 定义模型路径
47
48 # 循环整个数据集，进行 loss 值验证
49 def _loop_validation(model, generator):
50     n_steps = generator.steps_per_epoch
51     loss_value = 0
52     for _ in range(n_steps):      # 按批次循环获取数据，并计算 loss 值
53         xs, yolo_1, yolo_2, yolo_3 = generator.next_batch()
54         xs=tf.convert_to_tensor(xs)
55         yolo_1=tf.convert_to_tensor(yolo_1)
56         yolo_2=tf.convert_to_tensor(yolo_2)
57         yolo_3=tf.convert_to_tensor(yolo_3)
58         ys = [yolo_1, yolo_2, yolo_3]
59         ys_ = model(xs)
60         loss_value += loss_fn(ys, ys_, anchors=COCO_ANCHORS,
61                               image_size=[imgsize, imgsize] )
62     loss_value /= generator.steps_per_epoch
63     return loss_value
64
65 # 循环整个数据集，进行模型训练
66 def _loop_train(model,optimizer, generator,grad):
67     n_steps = generator.steps_per_epoch

```

```

68     for _ in tqdm(range(n_steps)): #按批次循环获取数据，并进行训练
69         xs, yolo_1, yolo_2, yolo_3 = generator.next_batch()
70         xs=tf.convert_to_tensor(xs)
71         yolo_1=tf.convert_to_tensor(yolo_1)
72         yolo_2=tf.convert_to_tensor(yolo_2)
73         yolo_3=tf.convert_to_tensor(yolo_3)
74         ys = [yolo_1, yolo_2, yolo_3]
75         optimizer.apply_gradients(grad(model, xs, ys))
76
77 if not os.path.exists(save_dir):
78     os.makedirs(save_dir)
79 save_fname = os.path.join(save_dir, "weights")
80
81 yolo_v3 = Yolonet(n_classes=len(LABELS)) #实例化yolo模型的类对象
82 #加载预训练模型
83 yolo_v3.load_darknet_params(YOLOV3_WEIGHTS, skip_detect_layer=True)
84
85 #定义优化器
86 optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
87
88 #定义函数计算loss值
89 def _grad_fn(yolo_v3, images_tensor, list_y_trues):
90     logits = yolo_v3(images_tensor)
91     loss = loss_fn(list_y_trues, logits, anchors=COCO_ANCHORS,
92                    image_size=[imgsize, imgsize])
93     return loss
94
95 grad = tfe.implicit_gradients(_grad_fn) #获得计算梯度的函数

```

代码第 77~95 行，实现了在动态图里建立梯度函数、优化器及 YOLO V3 模型的操作。有关动态图的使用方式可以参考第 6 章内容，这里不再详述。

### 3. 启用循环训练模型

按照指定的迭代次数循环，并用 history 列表接收测试的损失值，将损失值最小的模型保存起来。具体代码如下：

代码 8-21 manyolo（续）

```

96 history = []
97 for i in range(num_epoches):
98     _loop_train(yolo_v3, optimizer, generator, grad) #训练
99
100    loss_value = _loop_validation(yolo_v3, generator) #验证
101    print("{}-th loss = {}".format(i, loss_value))
102
103    #收集loss值
104    history.append(loss_value)
105    if loss_value == min(history): #只有在loss值创新低时才保存模型

```

```

106     print("update weight {}".format(loss_value))
107     yolo_v3.save_weights("{}{}.h5".format(save_fname))

```

代码运行后，输出以下结果：

```

100%|██████████| 16/16 [00:23<00:00, 1.46s/it]
0-th loss = 16.659032821655273
    update weight 16.659032821655273
.....
100%|██████████| 16/16 [00:22<00:00, 1.42s/it]
81-th loss = 0.8185760378837585
    update weight 0.8185760378837585
100%|██████████| 16/16 [00:22<00:00, 1.42s/it]
.....
85-th loss = 0.9106661081314087
100%|██████████| 16/16 [00:22<00:00, 1.42s/it]

```

从结果中可以看到，模型在训练时 loss 值会发生一定的抖动。在第 81 次时，loss 值为 0.81 达到了最小，程序将当时的模型保存了起来。

在真实训练的环境下，可以使用更多的样本数据，设置更多的训练次数，来让模型达到更好的效果。

同时，还可以在代码第 43 行将变量 jitter 设为 True，对数据进行尺度变化（这是数据增强的一种方法），以便让模型有更好的泛化效果。一旦使用了数据增强，模型会需要更多次数的迭代训练才可以收敛。

## 8.6.5 代码实现：用模型识别门牌号

编写代码，载入 test 目录下的测试样本，并输入模型进行识别。具体代码如下：

代码 8-21 manyolo（续）

```

108 IMAGE_FOLDER = os.path.join(PROJECT_ROOT, "data", "test", "*.png")
109 img_fnames = glob.glob(IMAGE_FOLDER)
110
111 imgs = [] #存放图片
112 for fname in img_fnames: #读取图片
113     img = cv2.imread(fname)
114     img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
115     imgs.append(img)
116
117 yolo_v3.load_weights(save_fname+".h5") #载入训练好的模型
118 import numpy as np
119 for img in imgs: #依次传入模型
120     boxes, labels, probs = yolo_v3.detect(img, COCO_ANCHORS, imgsize)
121     print(boxes, labels, probs)
122     image = draw_boxes(img, boxes, labels, probs, class_labels=LABELS,
123     desired_size=400)
123     image = np.asarray(image, dtype= np.uint8)

```

```
124     plt.imshow(image)
125     plt.show()
```

代码运行后，输出以下结果（见图 8-22~图 8-27）：

```
[[ 72.  24.  94.  66. ]
 [ 71.5 26.5 94.5 69.5]
 [ 93.  22.  119.  72. ]] [5 1 6] [0.1293204  0.83631355  0.94269735]
5: 12.93203979730606% 1: 83.6313545703888% 6: 94.26973462104797%
```

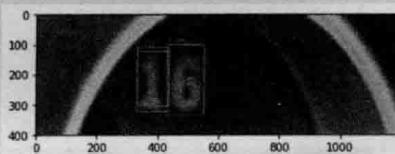


图 8-22 YOLO V3 结果 1

```
[[44.5 11. 55.5 33. ]] [6] [0.8771134]
6: 87.7113401889801%
```

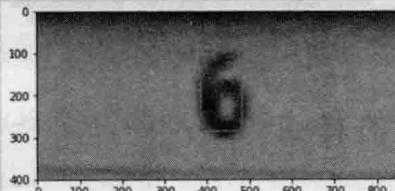


图 8-23 YOLO V3 结果 2

```
[[35.  6.5 45. 25.5]] [5] [0.6734172]
5: 67.34172105789185%
```

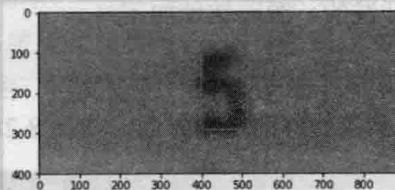


图 8-24 YOLO V3 结果 3

```
[[65. 16. 85. 50.]] [8] [0.49630296]
8: 49.63029623031616%
```

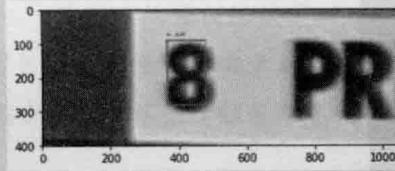


图 8-25 YOLO V3 结果 4

```
[[105.5 14.5 126.5 49.5]] [9] [0.719958]
9: 71.99580073356628%
```

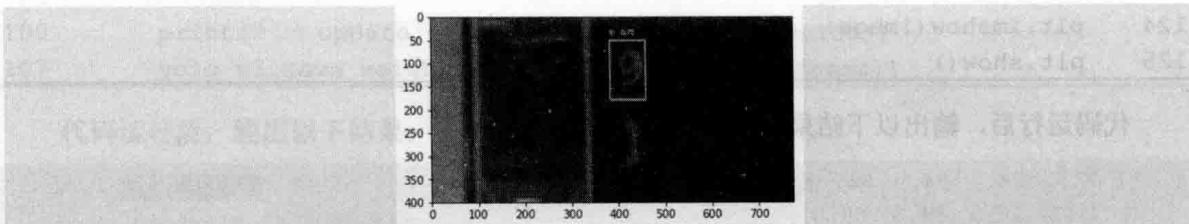


图 8-26 YOLO V3 结果 5

```
[ [60. 30. 74. 58.]
```

```
[75.5 34. 90.5 60. ] ] [6 9] [0.62158585 0.95006496]
```

```
6: 62.15858459472656%
```

```
9: 95.00649571418762%
```

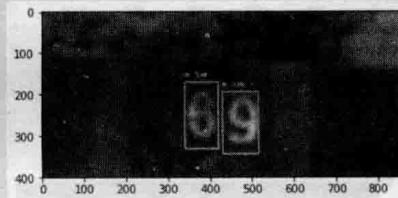


图 8-27 YOLO V3 结果 6

## 8.6.6 扩展：标注自己的样本

本小节介绍两个标注样本的工具。可以利用它们对自己的数据进行标注，然后按照本节的例子训练自己的模型。

### 1. Label-Tool

该工具是用 Python Tkinter 开发的。源码地址如下：

<https://github.com/puzzledqs/BBox-Label-Tool>

在上面链接的页面中可以看到该软件的操作界面，如图 8-28 所示。

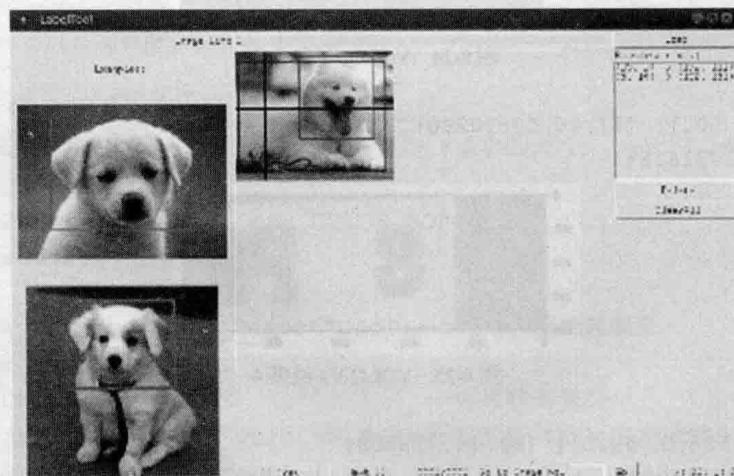


图 8-28 Label-Tool 工具

## 2. labelImg

该工具是用 Python 和 Qt 开发的。源码地址如下：

<https://github.com/tzutalin/labelImg>

从上面链接的页面中可以看到该软件的操作界面，如图 8-29 所示。



图 8-29 Label Img 工具

另外，在以下链接中还可以找到该软件的安装包：

<https://tzutalin.github.io/labelImg/>

## 8.7 实例 45：用 Mask R-CNN 模型定位物体的像素点

Mask R-CNN 模型是一个简单、灵活、通用的对象实例分割框架。它能够有效地检测图像中的对象，并为每个实例生成高质量的分割掩码，还可以通过增加不同的分支完成不同的任务。它可以完成目标分类、目标检测、语义分割、实例分割、人体姿势识别等多种任务。具体细节可以参考以下论文：

<https://arxiv.org/abs/1703.06870>

本节就通过实例来演示具体的做法。

### 实例描述

搭建 Mask R-CNN 模型，并加载现有的预训练权重。对任意一张图片进行计算，并在图上标出识别出来的物体名称、位置矩形框和精确的像素点。在程序正常执行之后，将 Mask R-CNN 模型的关键节点提取出来，并图示化。尝试根据结果及本书 8.7.3 小节介绍的模型结构，更深刻地理解 Mask R-CNN 模型。

本实例是用 tf.keras 接口实现的。先从 COCO 数据集的特点开始介绍，接着介绍 Mask R-CNN 模型的原理，并实现网络的搭建，然后加载 COCO 数据集上的预训练模型，最终完成对图片的检测。

## 8.7.1 下载 COCO 数据集及安装 pycocotools

COCO 数据集是微软发布的一个可以用来做图像识别训练的数据集，官方网址：<http://mscoco.org>。

图像主要从复杂的日常场景中截取，图像中的目标通过矩形框进行位置的标定。目前被广泛地用于图片分割任务中。在官网还为该数据集提供了配套的读取 API 工具——pycocotools。用户可以直接用该 API 载入数据。它帮助用户将精力更多地聚焦在模型上。下面就来完成数据的下载及 pycocotools 的安装。

### 1. 下载 COCO 数据集

COCO 数据集可以从以下链接下载：

<http://cocodataset.org/#download>

本实例使用 2014 年的 COCO 数据集。包含图片：训练集 82783 张、验证集 40504 张、测试集 40775 张，共分成 80 个类别。并配有目标检测的矩形框坐标标注、语义分割的散点标注、基于人物的关键点标注、对图片的整体文本描述标注。具体的下载界面如图 8-30 所示。

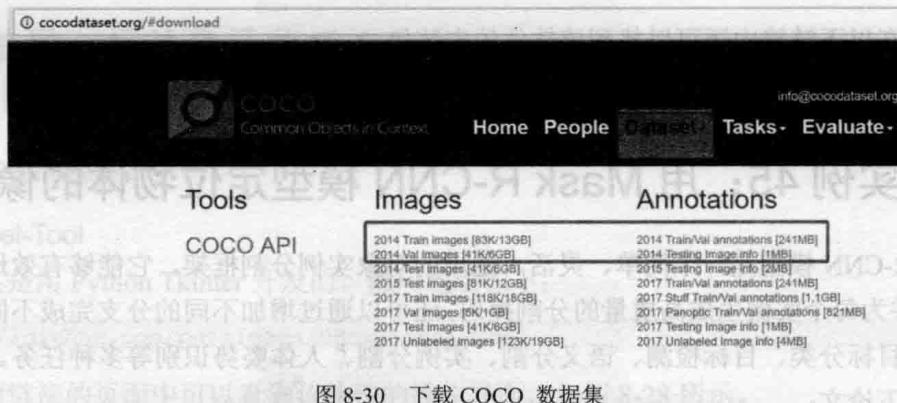


图 8-30 下载 COCO 数据集

在图 8-30 中一共有 4 个压缩文档：3 个图片数据集文档和 1 个标注数据集文档。



### 提示：

本章使用的图片都是在线获取的。如果只是跟着本书例子学习，可以只下载标注文档，不用下载其他的图片样本。

### 2. 安装 pycocotools

在安装 pycocotools 之前，还需要先安装两个工具软件 GIT 与 visualcppbuildtools，然后再用 pip 命令安装 pycocotools。具体步骤如下：

#### (1) 安装 GIT 软件。

GIT 为一个代码版本管理软件，可以与 GitHub 网站的代码库进行交互。安装该软件之后，就可以从 GitHub 网站上下载对应的 pycocotools 源代码。具体下载地址如下：

<https://git-scm.com/>

2 下载后直接将其安装即可。

### (2) 安装 visualcppbuildtools 软件。

该软件是 Visual studio 系列的编译工具。安装完之后，就可以用该工具对 pycocotools 源代码进行编译。下载地址如下：

```
https://download.microsoft.com/download/5/f/7/5f7acaeb-8363-451f-9425-68a90f98b238/visualcppbuildtools\_full.exe
```

### (3) 用 pip 命令安装 pycocotools。

直接在命令行里输入以下命令来安装 pycocotools。

```
pip install git+https://github.com/philferriere/cocoapi.git#subdirectory=PythonAPI
```

如果看到如图 8-31 所示的界面，则表示已经安装成功。

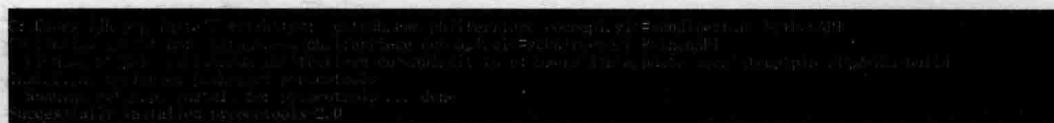


图 8-31 pycocotools 安装成功

## 8.7.2 代码实现：验证 pycocotools 及读取 COCO 数据集

在数据集的标注压缩包“annotations\_trainval2014.zip”中有以下 3 个标注文件。

- instances\_train2014.json：包含全部的分类信息、全部图片的坐标及分类标注信息。
- person\_keypoints\_train2014.json：包含基于人的关键点标注信息。
- captions\_train2014.json：包含基于全部图片的文本描述标注。

将数据集的标注压缩包“annotations\_trainval2014.zip”解压缩到本地代码文件夹 cocos2014 下，并编写代码进行验证。

### 1. 获得数据集的分类信息

用 pycocotools 接口中的 COCO 函数，将包含分类信息的文档“instances\_train2014.json”载入并解析。具体代码如下：

#### 代码 8-22 数据集验证

```
01 from pycocotools.coco import COCO
02 import numpy as np
03 import skimage.io as io
04 import matplotlib.pyplot as plt
05
06 annFile='./cocos2014/annotations_trainval2014/annotations/
07 instances_train2014.json'
08 coco=COCO(annFile) #加载注解的 JSON 格式数据
09 cats = coco.loadCats(coco.getCatIds()) #提取分类信息
```

```

10 print(cats, len(cats))          # 显示 80 个分类
11 nmcats=[cat['name'] for cat in cats]
12 print('COCO categories: \n{}'.format(' '.join(nmcats)))
13
14 nms = set([cat['supercategory'] for cat in cats])
15 print('COCO supercategories: \n{}'.format(' '.join(nms)))
16 print("supercategory len",len(nms))    # 显示 12 个超级分类
17
18 # 分类并不连续，例如：没有 26，第 1 个是 1，最后一个 90
19 catIds = coco.getCatIds(catNms=nmcats)
20 print(catIds)                  # 打印出分类的 ID

```

代码运行后，输出以下信息。

(1) 输出分类的信息，包括了每一类对应的 ID、名字及所属的超级类。一共 80 个，具体如下：

```
[{'supercategory': 'person', 'id': 1, 'name': 'person'}, {'supercategory': 'vehicle',
'id': 2, 'name': 'bicycle'}, ... ..., 'indoor', 'id': 87, 'name': 'scissors'},
{'supercategory': 'indoor', 'id': 88, 'name': 'teddy bear'}, {'supercategory': 'indoor',
'id': 89, 'name': 'hair drier'}, {'supercategory': 'indoor', 'id': 90, 'name':
'toothbrush'}] 80
```

(2) 输出 80 个分类的名称，具体如下：

COCO categories:

```
person bicycle car motorcycle airplane bus train truck boat traffic light hydrant
stop sign parking meter bench bird cat dog horse sheep cow elephant bear zebra giraffe
backpack umbrella handbag tie suitcase frisbee skis snowboard sports ball kite baseball
bat baseball glove skateboard surfboard tennis racket bottle wine glass cup fork knife
spoon bowl banana apple sandwich orange broccoli carrot hot dog pizza donut cake chair
couch potted plant bed dining table toilet tv laptop mouse remote keyboard cell phone
microwave oven toaster sink refrigerator book clock vase scissors teddy bear hair drier
toothbrush
```

(3) 输出 12 个超级类的名称，具体如下：

COCO supercategories:

```
furniture vehicle animal kitchen accessory outdoor food indoor sports electronic
person appliance
```

supercategory len 12

(4) 输出所有类的 ID，具体如下：

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 27, 28, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 46, 47, 48, 49, 50,
51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 67, 70, 72, 73, 74, 75, 76,
77, 78, 79, 80, 81, 82, 84, 85, 86, 87, 88, 89, 90]
```

从输出结果中可以看到，类的 ID 从 1 开始，而且不连续，例如，没有 26、29 等。

## 2. 加载并显示数据集的坐标标注

用 pycocotools 接口中的 COCO 函数，将包含图片坐标标注信息的文档“instances\_train2014.json”载入并解析，具体的代码如下：

代码 8-22 数据集验证（续）

```

21 catIds = coco.getCatIds(catNms=['person'])          #根据类名获得类 ID
22 imgIds = coco.getImgIds(catIds=catIds )           #根据类 ID 获得对应的图片列表
23 print(catIds,len(imgIds),imgIds[:5])
24
25 index = imgIds[np.random.randint(0,len(imgIds))] #从指定列表中取一张图片
26 print(index)
27 img = coco.loadImgs(index)[0]                      #index 可以是数组，会返回多个图片
28 print(img)
29 I = io.imread(img['coco_url'])                     #直接从网络获得该文件
30 plt.axis('off')                                     #工具箱内的工具图标或命令行提示符处显示的当前命令
31 plt.imshow(I)
32 plt.show()
33 plt.imshow(I); plt.axis('off')                      #获得标注的分割信息，并叠加到原图显示出来
34 annIds = coco.getAnnIds(imgIds=img['id'], catIds=catIds,
35                           iscrowd=None)                  #参数 iscrowd 代表是否是一群
36 #一条标注 ID 对应的信息——segmentation（分割）、bbox（框）、category_id（类别）
37 anns = coco.loadAnns(annIds)
38 print(annIds,anns)
39 coco.showAnns(anns) #将分割的信息叠加到图像上

```

代码第 21 行，让 coco.getCatIds 函数按照指定的类名返回对应的类 ID。

代码第 22 行，让 coco.getImgIds 函数按照指定的类 ID 返回对应的图片索引列表。

代码第 27 行，让 coco.loadImgs 函数按照指定的图片索引返回对应的标注信息。

代码运行后，输出结果大致分为以下两部分。

(1) 输出图片的坐标标注信息及内容。具体如下：

```

[1] 45174 [262145, 262146, 524291, 393223, 393224]
187519
{'license': 5, 'file_name': 'COCO_train2014_000000187519.jpg', 'coco_url':
'http://images.cocodataset.org/train2014/COCO_train2014_000000187519.jpg', 'height':
640, 'width': 367, 'date_captured': '2013-11-23 01:14:03', 'flickr_url':
'http://farm7.staticflickr.com/6014/5958747831_c486a37977_z.jpg', 'id': 187519}

```

上面显示的结果为图片的标注信息，对应代码第 21~32 行输出的内容。具体解读如下所示。

- 结果的第 1 行显示的内容为：person 类的 ID 为 1，person 类共有 45174 个图片，person 类中前 5 个图片的 ID 值。
- 第 2 行显示的内容为：从 45174 张图片中随机抽取了一个 ID 为 187519 的图片。
- 第 3 行到最后，显示的内容为 ID 为 187519 的图片所对应的标注信息。其中包括了文件名、URL、高、宽等信息。

接着，输出了图片的内容，如图 8-32 所示。

```

10 print(cats,len(cats))
11 print('类别信息：')
12 print(coco.cats)
13 print('bboxcats：')
14 anns = set([cat['id']for cat in coco.cats])
15 print('GT类别数：',len(anns))
16 print('GT类别名：',list(anns))
17 print('长图幅标注框数：',len(coco.anns))
18 print('长图幅标注框类名：',list(coco.anns))
19 cat_ids = coco.getCatIds(catNms=['baseball player'])
20 perImAnnIds = coco.getAnnIds(catIds=cat_ids)
21 perImAnn = coco.loadAnns(ids=perImAnnIds)
22 print('长图幅标注框数：',len(perImAnn))
23 print('长图幅标注框类名：',list(perImAnn))
24 print('长图幅标注框ID：',perImAnn[0]['id'])
25 print('长图幅标注框坐标：',perImAnn[0]['bbox'])
26 print('长图幅标注框分割掩码：',perImAnn[0]['segmentation'])
27 print('长图幅标注框面积：',perImAnn[0]['area'])
28 print('长图幅标注框是否是人群：',perImAnn[0]['iscrowd'])
29 print('长图幅标注框ID：',perImAnn[0]['id'])

# 取出某张图片的所有标注
30 anns = coco.loadAnns(coco.getAnnIds(imgIds=[187519]))
31 annId = anns[0]
32 print('长图幅标注框ID：',annId)
33 print('长图幅标注框坐标：',anns[0]['bbox'])
34 print('长图幅标注框分割掩码：',anns[0]['segmentation'])
35 print('长图幅标注框面积：',anns[0]['area'])
36 print('长图幅标注框是否是人群：',anns[0]['iscrowd'])
37 print('长图幅标注框ID：',anns[0]['id'])
38 annIds = [ann['id'] for ann in anns]
39 print('长图幅标注框ID列表：',annIds)
40 print('长图幅标注框数：',len(annIds))
41 print('长图幅标注框坐标：',anns[0]['bbox'])
42 print('长图幅标注框分割掩码：',anns[0]['segmentation'])
43 print('长图幅标注框面积：',anns[0]['area'])
44 print('长图幅标注框是否是人群：',anns[0]['iscrowd'])
45 print('长图幅标注框ID：',anns[0]['id'])

```



图 8-32 COCO 数据集图例

(2) 输出图片的坐标信息及将坐标信息叠加到图片上的内容。具体如下：

```

[447039, 1219580, 2167032]
[{'segmentation': [[168.85, 32.27, ..., 64.48, 146.7, 58.73, 161.08, 34.28]], 'area': 75920.5941499999, 'iscrowd': 0, 'image_id': 187519, 'bbox': [21.57, 32.27, 331.51, 593.11], 'category_id': 1, 'id': 447039},
 {'segmentation': [[60.7, 263.98, ..., 265.01], [56.58, ..., 101.95, 280.48]], 'area': 5370.287899999999, 'iscrowd': 0, 'image_id': 187519, 'bbox': [55.55, 195.92, 60.84, 196.95], 'category_id': 1, 'id': 1219580},
 {'segmentation': [[1.18, 200.27, ..., 5.47, 260.31], [5.47, ..., 1.18, 296.46]], 'area': 5143.66055, 'iscrowd': 0, 'image_id': 187519, 'bbox': [1.18, 200.27, 64.94, 227.92], 'category_id': 1, 'id': 2167032}]

```

输出结果的第 1 行是图 8-32 对应的坐标标注 ID（对应代码第 38 行中的变量 annIds），该坐标标注 ID 是含有 3 个元素的列表，表示图 8-32 中共有 3 条坐标标注信息。

输出结果的第 2~3 行、第 4~5 行、第 6~7 行分别为图 8-32 中 3 条坐标标注的具体信息。每条坐标标注信息都包括以下几个属性。

- segmentation：语义分割坐标。由若干个点坐标  $x$  和  $y$  组成，个数不定。
- area：所分割的面积。
- iscrowd：是否是一群个体，取值 0 或 1，用来指定 Segmentation 属性的格式。
- image\_id：所对应的图片 ID。
- bbox：位置所在的矩形框，由左上角的  $x$  和  $y$  坐标与右下角的  $x$  和  $y$  坐标组成，一共 4 个值。
- category\_id：物体的类别。
- id：该标注的 ID。

其中，segmentation 字段可以有 3 种格式来表示。

- poly 格式：坐标点组成的列表。
- uncompress RLE 格式：没有压缩的 Run Length Encoding。
- compact RLE 格式：压缩的 Run Length Encoding。

如果 iscrowd 为 0，则 segmentation 为 poly 格式；如果 iscrowd 为 1，则 segmentation 为 RLE 格式。

将坐标标注信息叠加到图片上之后，如图 8-33 所示，可以看到其对应的语义分割区域。

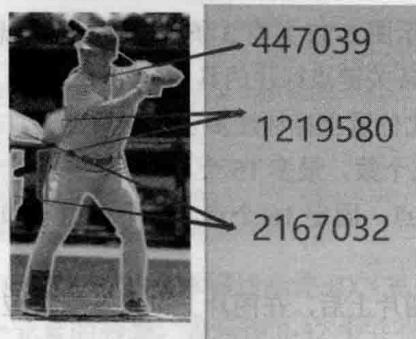


图 8-33 COCO 数据集坐标图例

在输出结果的第 4~5 行和第 6~7 行，可以看到其 Segmentation 字段为两个数组。所以对应于图 8-33 中 ID 为 1219580、2167032 的标注分别有两个区域。

### 3. 加载并显示基于人的关键点标注信息

用 pycocotools 接口中的 COCO 函数将包含人物关键点标注信息的文档“person\_keypoints\_train2014.json”载入并解析。具体的代码如下：

#### 代码 8-22 数据集验证（续）

```
40 annFile =
    './annotations_trainval2014/annotations/person_keypoints_train2014.json'
41 coco_kps=COCO(annFile)
42 plt.imshow(I); plt.axis('off')
43 ax = plt.gca()
44 annIds = coco_kps.getAnnIds(imgIds=img['id'], catIds=catIds, iscrowd=None)
45 anns = coco_kps.loadAnns(annIds)#超级类 person 的每条标注，包括了关键点及
    segmentation 和 bbox、category_id
46 print(annIds, anns)
47 coco_kps.showAnns(anns)
```

代码运行后，输出以下结果：

(1) 输出图片的关键点标注信息，并将关键点信息叠加到图片上的内容。具体如下：

```
[447039, 1219580, 2167032]
[{'segmentation': [[168.85, 32.27, ..., 34.28]], 'num_keypoints': 16, 'area':
75920.59415, 'iscrowd': 0, 'keypoints': [148, 96, 2, ..., 582, 2], 'image_id': 187519,
'bbox': [21.57, 32.27, 331.51, 593.11], 'category_id': 1, 'id': 447039},
 {'segmentation': [[60.7, 263.98, ..., 265.01], [56.58, ..., 101.95, 280.48]], 'num_
keypoints': 8, 'area': 5370.2879, 'iscrowd': 0, 'keypoints': [0, 0, 0, 0, 0, ..., 1, 0, 0, 0, 0, 0], 'image_id': 187519, 'bbox': [55.55, 195.92, 60.84, 196.95],
'category_id': 1, 'id': 1219580},
```

```
{'segmentation': [[1.18, ..., 5.47, 260.31], [5.47, ..., 296.46]], 'num_keypoints': 4, 'area': 5143.66055, 'iscrowd': 0, 'keypoints': [0, 0, ..., 0, 0], 'image_id': 187519, 'bbox': [1.18, 200.27, 64.94, 227.92], 'category_id': 1, 'id': 2167032}]
```

输出结果的第 1 行是图 8-32 的关键点标注 ID（见代码第 44 行的 annIds），该关键点标注 ID 是含有 3 个元素的列表，表示图 8-32 共有 3 条关键点标注信息。

输出结果的是图 8-32 中 3 条关键点标注的具体信息（2 和 3 是一条，4 和 5 是一条，6 和 7 是一条）。每条关键点标注都比位置坐标标注多了两个属性。

- **num\_keypoints**: 关键点个数，最多 16 个。
- **keypoints**: 具体的关键点，固定 16 个点，每个点由  $x$  和  $y$  两个值组成。如果个数不足 16，则需要补 0。

将关键点标注信息叠加到图片上后，在图片上可以看到对应的语义分割及关键点区域，如图 8-34 所示。

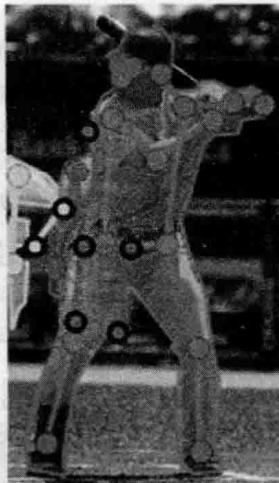


图 8-34 COCO 数据集人物关键点图例

#### 4. 加载并显示文本描述标注信息

使用 pycocotools 接口中的 COCO 函数将含有文本描述标注信息的文件“captions\_train2014.json”载入并解析。

具体代码如下：

#### 代码 8-22 数据集验证（续）

```
48 annFile =
    './annotations_trainval2014/annotations/captions_train2014.json'
49 coco_caps=COCO(annFile)           #加载 Json 文件，获取图片描述
50 annIds = coco_caps.getAnnIds(imgIds=img['id']) #每一个图片 ID 对应于多条描述
51 anns = coco_caps.loadAnns(annIds)      #根据描述 ID 载入每条描述
52 print(annIds,anns)                  #每条描述包括图片 ID 和一段文字
53 coco_caps.showAnns(anns)
```

代码运行后，输出以下结果：

```
[270682, 275047, 275455, 276205, 279877]
```

```
[{'image_id': 187519, 'id': 270682, 'caption': 'A man standing on home plate holding a baseball bat.'}, {'image_id': 187519, 'id': 275047, 'caption': 'A man swinging a baseball bat on a field.'}, {'image_id': 187519, 'id': 275455, 'caption': 'A baseball player is standing with his bat raised.'}, {'image_id': 187519, 'id': 276205, 'caption': 'A baseball player up at bat in a game in a stadium.'}, {'image_id': 187519, 'id': 279877, 'caption': 'A ball player is preparing to take a swing.'}]
```

A man standing on home plate holding a baseball bat.

A man swinging a baseball bat on a field.

A baseball player is standing with his bat raised.

A baseball player up at bat in a game in a stadium.

A ball player is preparing to take a swing.

输出结果的第1行是图8-32对应的文本描述标注ID(见代码第50行的annIds变量)，该文本描述标注ID是含有5个元素的列表，表示图8-32中共有5条文本描述标注信息。

输出结果的第2~5行是这5条文本描述标注的具体信息。

输出结果的第6~10行是描述图8-32的5条具体文本。该信息由COCO对象的showAnns方法输出(见代码第53行)。

### 8.7.3 拆分Mask R-CNN模型的处理步骤

Mask R-CNN模型属于两阶段(2-stage)检测模型，即该模型会先检测包含实物的区域，再对该区域内的实物进行分类识别。

#### 1. 检测实物区域的步骤

具体步骤如下：

- (1) 按照算法将一张图片分成多个子框。这些子框被叫作锚点(anchors)，锚点是不同尺度的矩形框，彼此间存在部分重叠。

- (2) 在图片中为具体的实物标注位置坐标(所属的位置区域)。

- (3) 根据实物标注的位置坐标与锚点区域的面积重合度(Intersection over Union, IOU)计算出哪些锚点属于前景、哪些锚点属于背景(重叠度高的就是前景，重叠度低的就是背景，重叠度一般的就忽略掉)。

- (4) 根据第(3)步结果中属于前景的锚点坐标和第(2)步结果中实物标注的位置坐标，计算出二者的相对位移和长宽的缩放比例。

最终，检测区域中的任务会被转化成对一堆锚点框的分类(前景和背景)和回归任务(偏移和缩放)。如图8-35所示，每张图片都会将其自身标注的信息转化为与锚点对应的标签，让模型对已有的锚点进行训练或识别。

在Mask R-CNN模型中，担当区域检测功能的网络被称作RPN(Region Proposal Network)。

在实际处理过程中，会从RPN的输出结果中选取前景概率较高的一定数量锚点作为靠谱区域(Region Of Interest, ROI)，送到第2阶段的网络中进行计算。

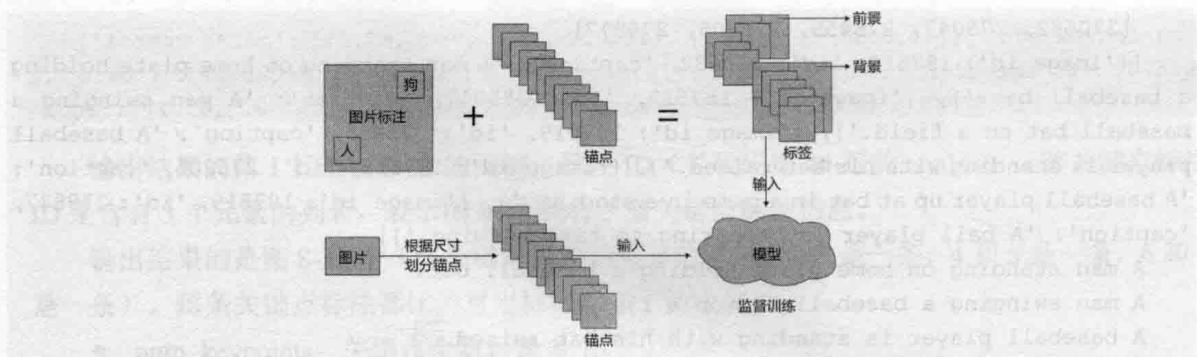


图 8-35 区域检测图例

## 2. Mask R-CNN 模型的完整步骤

Mask R-CNN 模型可以拆分成以下 5 个子步骤。

(1) 提取主特征：这部分的模型又被叫作骨干网络。它用来从图片中提取出一些不同尺度的重要特征，通常用于一些预训练好的网络（如 VGG 模型、Inception 模型、Resnet 模型等）。这些获得的特征数据被称作 feature map。

(2) 特征融合：用特征金字塔网络（Feature Pyramid Network, FPN）整合骨干网络中不同尺度的特征。最终的特征信息用于后面的 RPN 网络和最终的分类器网络。

(3) 提取靠谱区域：主要通过 RPN 来实现。该网络的作用是，在众多锚点中计算出前景和背景的预测值，并算出基于锚点的偏移，然后对前景概率较大的靠谱区域用 NMS 算法去重，并从最终结果中取出指定个数的 ROI 用于后续网络的计算。

(4) ROI 池化：用区域对齐（ROIAlign）的方式进行。将第（2）步的结果当作图片，按照 ROI 中的区域框位置从图中取出对应的内容，并将形状统一成指定大小，用于后面的计算。

(5) 最终检测：将第 4 步的结果输入依次送入分类器网络（classifier）进行分类与边框坐标的计算。再将带有精确边框坐标的分类结果一起送到检测器网络（detectioner）进行二次去重（过滤掉类别分数较小且重复度高于指定阈值的 ROI），以实现实物矩形检测功能。最后再将前面检测器的结果与第（2）步结果一起送入掩码检测器（Mask\_Detectioner）进行实物像素分割。

完整的架构如图 8-36 所示。

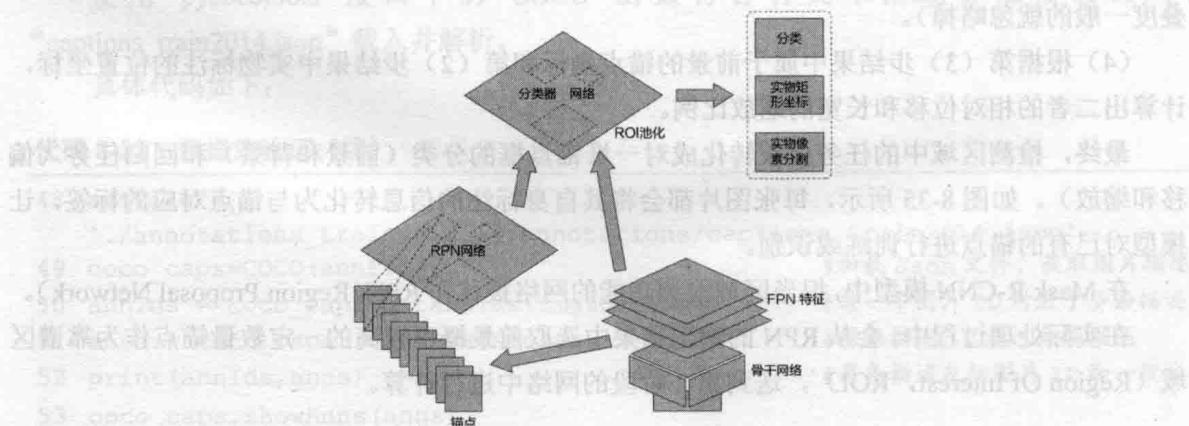


图 8-36 Mask-RCNN 架构图

## 8.7.4 工程部署：准备代码文件及模型

Mask R-CNN 模型的预训练模型的下载地址如下：

```
https://github.com/matterport/Mask\_RCNN/releases/download/v2.0/mask\_rcnn\_coco.h5
```

将预训练模型下载之后，放到本地代码的同级目录下。再从本书配套资源里将该项目的源代码文件取出，放到本地路径下，完成项目的部署。

该项目由 5 个代码文件组成，具体说明如下。

- “8-23 Mask\_RCNN 应用.py”：放置使用模型的全流程代码，以及讲解模型内部过程的示例代码。
- “8-24 mask\_rcnn\_model.py”：放置 Mask-RCNN 模型的具体代码。
- “8-25 mask\_rcnn\_utils.py”：放置模型所需要的辅助工具代码。
- “8-26 mask\_rcnn\_visualize.py”：放置可视化部分的显示代码。
- “8-27 othernet.py”：放置 Mask\_RCNN 中使用的具体模型，包括 RPN 模型、FPN 模型、分类器模型（用于图片分类）、检测器模型（用于目标检测）、Mask 模型（用于图片分割）。

## 8.7.5 代码实现：加载数据构建模型，并输出模型权重

编写代码完成以下步骤：

- (1) 载入必要的代码模块，并将本实例中用到的其他代码文件载入。
- (2) 用 pycocotools 工具将 COCO 数据集中的类名提取出来。
- (3) 实例化 MaskRCNN 类，并构建 Mask R-CNN 模型。
- (4) 将预训练模型 Mask R-CNN 的权重文件载入。
- (5) 用 html\_weight\_stats 函数将权重文件的内容保存成网页形式，并显示出来。

具体的代码如下：

代码 8-23 Mask\_RCNN 应用

```

01 import numpy as np
02 import tensorflow as tf
03 import matplotlib.pyplot as plt
04 from pycocotools.coco import COCO
05 import skimage.io as io #载入必要的模块
06
07 mask_rcnn_model = __import__("8-24 mask_rcnn_model")
08 MaskRCNN = mask_rcnn_model.MaskRCNN
09 utils = __import__("8-25 mask_rcnn_utils")
10 visualize = __import__("8-26 mask_rcnn_visualize")
11
12 #加载数据集
13 annFile='./cocos2014/annotations_trainval2014/annotations/
instances_train2014.json'
```

```

14 coco=COCO(annFile)          #加载注解的 JSON 格式数据
15
16 class_ids = sorted(coco.getCatIds())      #获得分类 ID
17 class_info = coco.loadCats(coco.getCatIds())  #提取分类信息
18 class_name=[n["name"] for n in class_info]
19
20 class_ids.insert(0,0)           #所有的类索引
21 class_name.insert(0,"BG")      #所有的类名
22
23 print(class_ids)
24 print(class_name)
25
26 #载入模型
27 BATCH_SIZE = 1                #批次
28 MODEL_DIR = "./log"
29 #指定模型运行的设备
30 DEVICE = "/cpu:0"  #指定模型在第 0 块 CPU 上运行（也可以指定在 GPU 上运行）
31 #以 inference 模式构建模型
32 with tf.device(DEVICE):
33     model = MaskRCNN(mode="inference", model_dir=MODEL_DIR,
34                         num_class=len(class_ids),batch_size = BATCH_SIZE)  #指定分类个数 (81)
35 #模型权重文件路径
36 weights_path = "./mask_rcnn_coco.h5"
37
38 #载入权重文件
39 print("Loading weights ", weights_path)
40 model.load_weights(weights_path, by_name=True)
41
42 #将所有的可训练权重显示出来
43 utils.html_weight_stats(model)    #显示权重

```

运行代码后，输出以下结果：

```
[0, 1, 2, 3, .....89, 90]
['BG', 'person',..... 'toothbrush']
```

在输出结果中：

- 第 1 行显示的是类的 ID。
- 第 2 行显示的类的名称。

在代码第 20、21 行，对原始的数据类进行变换，加入一个背景类（ID 为 0，名称为 BG）。



### 提示：

因为本实例使用的预训练模型就是按照含有 ID 为 0 的背景类结构进行训练的，所以在构建 Mask R-CNN 模型时也必须添加这个背景类。

另外，在训练自己的数据时也建议使用这种技巧，它可以让模型训练出更好的效果。

00 POST\_MERGE\_INTEGRATION = 1000

在代码运行之后，会在本地目录下生成一个名为 a.html 的文件。双击打开该文件，可以看到如图 8-37 所示的权重列表。

64	BN2A_BRANCH2A/Moving_Variance:0	(64,)	+0.0000	+8.9258	+2.0314
65	RES2A_BRANCH2B/Kernel:0	(3, 3, 64, 64)	-0.3878	+0.5070	+0.0323
66	RES2A_BRANCH2B/Bias:0	(64,)	-0.0037	+0.0026	+0.0010
67	BN2A_BRANCH2B/Gamma:0	(64,)	+0.3165	+1.7010	+0.3042
68	BN2A_BRANCH2B/Beta:0	(64,)	-1.9348	+4.5429	+1.5113
69	BN2A_BRANCH2B/Moving_Mean:0	(64,)	-6.7752	+4.5769	+2.2594
70	BN2A_BRANCH2B/Moving_Variance:0	(64,)	+0.0000	+5.5085	+1.0835
71	RES2A_BRANCH2C/Kernel:0	(1, 1, 64, 256)	-0.4468	+0.3615	+0.0410
72	RES2A_BRANCH2C/Bias:0	(256,)	-0.0041	+0.0052	+0.0016
73	RES2A_BRANCH1/Kernel:0	(1, 1, 64, 256)	-0.8674	+0.7588	+0.0703
74	RES2A_BRANCH1/Bias:0	(256,)	-0.0034	+0.0025	+0.0009
75	BN2A_BRANCH2C/Gamma:0	(256,)	-0.5782	+3.1806	+0.6192
76	BN2A_BRANCH2C/Beta:0	(256,)	-1.1422	+1.4273	+0.4229
77	BN2A_BRANCH2C/Moving_Mean:0	(256,)	-4.2602	+3.0864	+1.0168
78	BN2A_BRANCH2C/Moving_Variance:0	(256,)	+0.0000	+2.6688	+0.3827
79	BN2A_BRANCH1/Gamma:0	(256,)	+0.2411	+3.4973	+0.6241
80	BN2A_BRANCH1/Beta:0	(256,)	-1.1422	+1.4274	+0.4229
81	BN2A_BRANCH1/Moving_Mean:0	(256,)	-8.0883	+8.6554	+2.0289
82	BN2A_BRANCH1/Moving_Variance:0	(256,)	+0.0000	+8.7306	+1.5526
83	RES2B_BRANCH2A/Kernel:0	(1, 1, 256, 64)	-0.2536	+0.2319	+0.0358
84	RES2B_BRANCH2A/Bias:0	(64,)	-0.0027	+0.0028	+0.0012

图 8-37 Mask-RCNN 权重列表

## 8.7.6 代码实现：搭建残差网络 ResNet

搭建一个残差网络(ResNet 模型)作为 Mask R-CNN 模型中的骨干网结构。

在具体实现时，将 ResNet 模型封装成 API，以便程序调用，具体步骤如下。

### 1. 载入模块，定义模型参数

整个 Mask R-CNN 模型的网络结构都是在代码文件“8-24 mask\_rcnn\_model.py”中实现的。在代码开始处，先引入全部的模块，并定义需要的参数。具体的代码如下：

#### 代码 8-24 mask\_rcnn\_model

```

01 import os
02 import random
03 import datetime
04 import re
05 import math
06 import logging
07 import numpy as np
08 import skimage.transform
09 import tensorflow as tf
10 from tensorflow import keras
11 from tensorflow.keras import backend as K      #载入 keras 的后端实现
12 from tensorflow.keras import layers as KL
13 from tensorflow.keras import models as KM      #载入模块
14

```

```

15 utils = __import__("8-25 mask_rcnn_utils")
16 log = utils.log
17 compose_image_meta = utils.compose_image_meta
18 othernet = __import__("8-27 othernet")
19 build_rpn_model = othernet.build_rpn_model
20 ProposalLayer = othernet.ProposalLayer
21 fpn_classifier_graph = othernet.fpn_classifier_graph
22 DetectionLayer = othernet.DetectionLayer
23 build_fpn_mask_graph = othernet.build_fpn_mask_graph
24 parse_image_meta_graph = othernet.parse_image_meta_graph
25 #要求TensorFlow的版本在1.8以上，这样MNS算法才会表现稳定
26 from distutils.version import LooseVersion
27 assert LooseVersion(tf.__version__) >= LooseVersion("1.8")
28
29 #定义全局输入图片大小(二选一)，图片会被下采样6次，必须能够被2的6次方整除
30 IMAGE_MIN_DIM = 800
31 IMAGE_MAX_DIM = 1024
32 IMAGE_DIM = IMAGE_MAX_DIM          #选择1024
33 IMAGE_RESIZE_MODE = "square"       #统一成IMAGE_MAX_DIM
34
35 #对图片变化尺寸时，定义的最小缩放范围。0代表不限制最小缩放范围
36 IMAGE_MIN_SCALE = 0
37
38 BACKBONE = "resnet101"           #主干网络使用ResNet
39
40 #骨干网络返回的每一层特征，对原始图片的缩小比例代表着输出特征的5种尺度
41 #在计算锚点时，BACKBONE_STRIDES的每个元素代表按照该像素值划分网格
42 #骨干网络输出的特征，其尺度分别为256、128、64、32、16，代表输出的网格个数分别为256、
43   128、64、32、16
44 BACKBONE_STRIDES = [4, 8, 16, 32, 64]
45 #扫描网格的步长。按照该步长获取网格，用于计算锚点。网格中的第1个像素坐标被当作锚点的中
46   心点
47 RPN_ANCHOR_STRIDE = 1
48 #每个锚点的边长初始值
49 RPN_ANCHOR_SCALES = (32, 64, 128, 256, 512)
50
51 #锚点的边长比例(width/height)，将初始值和边长比例一起计算，得到锚点的真实边长
52 RPN_ANCHOR RATIOS = [0.5, 1, 2]
53
54 RPN_TRAIN_ANCHORS_PER_IMAGE = 256      #训练RPN时选取锚点的个数
55 TRAIN_ROIS_PER_IMAGE = 200                #在训练过程中，将选取多少个ROI放到FPN层中
56 ROI_POSITIVE_RATIO = 0.33                 #训练过程中选取的正向ROI比例，用于送往FPN
57
58 #对应于训练或是使用时，RPN最终需要最大保留多少个ROI
59 POST_NMS_ROIS_TRAINING = 2000

```

```

60 POST_NMS_ROIS_INFERENCE = 1000      #特征金字塔层的深度
61 RPN_NMS_THRESHOLD = 0.7
62 FPN_FEATURE = 256
63 DETECTION_MAX_INSTANCES = 100       #FPN最终检测的实例个数
64 #在制作样本的标签时,从一张图片中最多只读取100个实例
65 MAX_GT_INSTANCES = 100
66 #分类时的置信度阈值
67 DETECTION_MIN_CONFIDENCE = 0.7
68 #检测时的Non-maximum suppression阈值
69 DETECTION_NMS_THRESHOLD = 0.3
70
71 #定义池化ROI的相关参数
72 POOL_SIZE = 7                         #金字塔对齐池化后的ROI形状
73 MASK_POOL_SIZE = 14
74 MASK_SHAPE = [28, 28]
75 #定义RPN和最终检测的边界框细化标准偏差
76 RPN_BBOX_STD_DEV = np.array([0.1, 0.1, 0.2, 0.2])
77 BBOX_STD_DEV = np.array([0.1, 0.1, 0.2, 0.2])
78
79 #是否对掩码进行压缩
80 USE_MINI_MASK = True
81 MINI_MASK_SHAPE = (56, 56)             #压缩后的掩码大小(height, width)

```

代码中每个参数的定义,都做了详细的注释。读者需要理解这些定义,并与具体的算法规则结合起来,才能更好地理解代码。



### 提示:

在代码第24行,用断言函数判断TensorFlow的版本,要求TensorFlow的版本号要在1.8以上。原因在于,本实例直接使用了TensorFlow中的NMS算法库。

如果使用的是1.8以下的版本,则不建议使用TensorFlow中的NMS的算法库。可以将使用TensorFlow中的NMS算法库的代码(见8.7.12小节)改成使用8.5.9小节自定义的NMS算法函数。

## 2. 搭建残差块

残差网络中最核心的部分是通过短链接实现的残差块。

在ResNet101模型中实现了两种不同的残差块结构:

- 不带卷积操作的短链接结构。
- 带卷积操作的短链接结构。

这两种残差块的实现代码如下:

### 代码 8-24 mask\_rcnn\_model(续)

```

82 def compute_backbone_shapes(image_shape):    #计算ResNet返回的形状
83     returnshape = [[int(math.ceil(image_shape[0] / stride)),
84     (image_shape[1] * stride), (image_shape[2] * stride)] for
85     stride in backbone_strides]

```

```

84         int(math.ceil(image_shape[1] / stride))) for stride in backbone_strides]
85     return np.array(returnshape)
86
87 #ResNet 中的 identity_block(不带卷积的短链接)
88 def identity_block(input_tensor, kernel_size, filters, stage, block,
89 use_bias=True, train_bn=True): #kernel_size 是第 2 层卷积核的大小。Filters 是每层
90   #卷积核的个数, stage 和 block 用于命名
91   nb_filter1, nb_filter2, nb_filter3 = filters #解析出每层卷积核个数
92   conv_name_base = 'res' + str(stage) + block + '_branch' #为卷积层命名
93   bn_name_base = 'bn' + str(stage) + block + '_branch' #为 BN 层命名
94
95   x = KL.Conv2D(nb_filter1, (1, 1), name=conv_name_base + '2a',
96   use_bias=use_bias)(input_tensor)
97   x = KL.BatchNormalization(name=bn_name_base + '2a')(x,
98   training=train_bn)
99   x = KL.Activation('relu')(x)
100
101   x = KL.Conv2D(nb_filter2, (kernel_size, kernel_size), padding='same',
102   name=conv_name_base + '2b', use_bias=use_bias)(x)
103   x = KL.BatchNormalization(name=bn_name_base + '2b')(x,
104   training=train_bn)
105   x = KL.Activation('relu')(x)
106
107   x = KL.Conv2D(nb_filter3, (1, 1), name=conv_name_base + '2c',
108   use_bias=use_bias)(x)
109   x = KL.BatchNormalization(name=bn_name_base + '2c')(x,
110   training=train_bn)
111
112   x = KL.Add()([x, input_tensor]) #短链接
113   x = KL.Activation('relu', name='res' + str(stage) + block + '_out')(x)
114   return x
115
116 #ResNet 中的 conv_block(带卷积的短链接)
117 def conv_block(input_tensor, kernel_size, filters, stage, block, strides=(2,
118 2), use_bias=True, train_bn=True): #strides 为第 1 层的步长, 进行了下采样, 所以
119 短链接时也得下采样
120
121   nb_filter1, nb_filter2, nb_filter3 = filters
122   conv_name_base = 'res' + str(stage) + block + '_branch'
123   bn_name_base = 'bn' + str(stage) + block + '_branch'
124
125   #第 1 层, 1x1 卷积
126   x = KL.Conv2D(nb_filter1, (1, 1), strides=strides, name=conv_name_base
127   + '2a', use_bias=use_bias)(input_tensor)
128   x = KL.BatchNormalization(name=bn_name_base + '2a')(x, training=train_bn)

```

```

119     x = KL.Activation('relu')(x)
120
121     # 第 2 层，按照指定卷积核卷积
122     x = KL.Conv2D(nb_filter2, (kernel_size, kernel_size), padding='same',
123                   name=conv_name_base + '2b', use_bias=use_bias)(x)
124     x = KL.BatchNormalization(name=bn_name_base + '2b')(x, training=train_bn)
125     x = KL.Activation('relu')(x)
126
127     # 第 3 层，1×1 卷积
128     x = KL.Conv2D(nb_filter3, (1, 1), name=conv_name_base + '2c',
129                   use_bias=use_bias)(x)
130     x = KL.BatchNormalization(name=bn_name_base + '2c')(x, training=train_bn)
131
132     # 带卷积的短链接
133     shortcut = KL.Conv2D(nb_filter3, (1, 1), strides=strides,
134                           name=conv_name_base + '1', use_bias=use_bias)(input_tensor)
135     shortcut = KL.BatchNormalization(name=bn_name_base + '1')(shortcut,
136                           training=train_bn)
137     x = KL.Add()([x, shortcut])
138     x = KL.Activation('relu', name='res' + str(stage) + block + '_out')(x)
139
140     return x

```

代码第 88 行定义了 identity\_block 层，实现了不带卷积的残差块，主要是用于识别图像特征。

代码第 110 行定义了 conv\_block 层，实现了带卷积的残差块（见代码第 131 行）。在识别图像特征的同时，又对原有图片进行了下采样。残差网络主要是将这两种单元结构按照一定顺序串联起来，形成了深层的神经网络，从而具有分析特征的能力。

### 3. 搭建 ResNet 模型

ResNet 模型常被用在复杂模型中，实现特征提取功能。经典的 ResNet 模型有两种结构：ResNet50 和 ResNet101。

- ResNet50 一共有 50 层，属于较小型网络，精度稍低一些，但运算速度更快。
- ResNet101 一共有 50 层，属于较大型网络，精度稍高一些，但运算速度较慢。

下面代码中用 resnet\_graph 函数来搭建 ResNet 模型。函数 resnet\_graph 可以同时支持 ResNet101 和 ResNet50 两种模型的实现。

在整个 Mask R-CNN 模型中，仅获取残差网络输出的最终特征是不够的，还需要将其中间状态的部分特征抽取出来。

在代码实现时，按照整个网络对原始图片的缩放尺度（每个带卷积的残差块都会将尺寸缩小原来的一半）将不同尺寸的特征层抽取出来。

具体代码如下：

## 代码 8-24 mask\_rcnn\_model (续)

```

137 #组建残差网络，支持 resnet50 和 resnet101 两种。参数 stage5 表示是否将第 5 特征层的结果输出
138 def resnet_graph(input_image, architecture, stage5=False, train_bn=True):
139
140     assert architecture in ["resnet50", "resnet101"]
141     #第 1 特征层
142     x = KL.ZeroPadding2D((3, 3))(input_image)
143     x = KL.Conv2D(64, (7, 7), strides=(2, 2), name='conv1', use_bias=True)(x)
144     x = KL.BatchNormalization(name='bn_conv1')(x, training=train_bn)
145     x = KL.Activation('relu')(x)
146     C1 = x = KL.MaxPooling2D((3, 3), strides=(2, 2), padding="same")(x)
147     #第 2 特征层
148     x = conv_block(x, 3, [64, 64, 256], stage=2, block='a', strides=(1, 1),
149     train_bn=train_bn)
150     x = identity_block(x, 3, [64, 64, 256], stage=2, block='b',
151     train_bn=train_bn)
152     C2 = x = identity_block(x, 3, [64, 64, 256], stage=2, block='c',
153     train_bn=train_bn)
154     #第 3 特征层
155     x = conv_block(x, 3, [128, 128, 512], stage=3, block='a', train_bn=train_bn)
156     x = identity_block(x, 3, [128, 128, 512], stage=3, block='b',
157     train_bn=train_bn)
158     x = identity_block(x, 3, [128, 128, 512], stage=3, block='c',
159     train_bn=train_bn)
160     C3 = x = identity_block(x, 3, [128, 128, 512], stage=3, block='d',
161     train_bn=train_bn)
162     #第 4 特征层
163     x = conv_block(x, 3, [256, 256, 1024], stage=4, block='a',
164     train_bn=train_bn)
165     block_count = {"resnet50": 5, "resnet101": 22}[architecture]
166     for i in range(block_count):
167         x = identity_block(x, 3, [256, 256, 1024], stage=4, block=chr(98+i),
168         train_bn=train_bn)
169         C4 = x
170         #第 5 特征层
171         if stage5:
172             x = conv_block(x, 3, [512, 512, 2048], stage=5, block='a',
173             train_bn=train_bn)
174             x = identity_block(x, 3, [512, 512, 2048], stage=5, block='b',
175             train_bn=train_bn)
176             C5 = x = identity_block(x, 3, [512, 512, 2048], stage=5, block='c',
177             train_bn=train_bn)
178         else:
179             C5 = None
180     return [C1, C2, C3, C4, C5]

```

在上述代码的最后一行，返回了 ResNet 模型中每个特征层所抽取的特征数据。其中，第 1 特征层至第 5 特征层分别用张量 C1~C5 表示。

每个特征层都是通过对上层数据进行下采样处理得来的。假如输入图片的尺寸为 [1024,2048,3]，则 C1 到 C5 的尺寸依次为：[256,256]、[128,128]、[64,64]、[32,32]、[16,16]。

## 8.7.7 代码实现：搭建 Mask R-CNN 模型的骨干网络 ResNet

下面通过 MaskRCNN 类搭建 Mask R-CNN 模型。在 MaskRCNN 类中，实现模型的两种使用方式：训练（training）方式和接口调用（inference）方式。因为本实例是直接使用预训练模型进行实现，所以只实现其接口功能即可。

### 1. 在 MaskRCNN 类中搭建 ResNet 模型

在 MaskRCNN 类中，用成员变量 keras\_model 来创建 Mask R-CNN 模型。基本思路是：首先通过 MaskRCNN 类的初始化方法（\_\_init\_\_）为其添加基本设置；接着通过 build 方法为 keras\_model 构建模型。在 build 方法中，用 resnet\_graph 函数构建 ResNet 模型，并返回其中 5 种尺度的特征。

在构建模型之前，需要实现一个 mold\_inputs 方法，以便对输入的图片进行预处理。在 mold\_inputs 方法中，将图片等比例缩放到 [1024,1024,3] 大小，并将尺寸不足的地方补 0。具体实现代码如下：

代码 8-24 mask\_rcnn\_model（续）

```

170 class MaskRCNN():
171     def __init__(self, mode, model_dir, num_class, batch_size):
172         """
173             mode: 可以是 training 或 inference 两种模式
174             model_dir: 保存模型的路径
175         """
176         assert mode == 'inference'
177         self.mode = mode
178         self.num_class = num_class
179         self.batch_size = batch_size
180         self.model_dir = model_dir
181         self.set_log_dir()
182         self.keras_model = self.build(mode=mode)      #keras_model 是真正模型
183
184     def mold_inputs(self, images):
185         molded_images = []
186         image_metas = []
187         windows = []
188         for image in images:
189             #window 是缩放后有效图片的坐标
190             #scale 是缩放比例
191

```

```

192     molded_image, window, scale, padding, crop = utils.resize_image(
193         image,
194         min_dim=IMAGE_MIN_DIM,
195         min_scale=IMAGE_MIN_SCALE,
196         max_dim=IMAGE_MAX_DIM,
197         mode=IMAGE_RESIZE_MODE)
198     molded_image = mold_image(molded_image) #均值化
199
200     #把图片配套的信息也打包好
201     image_meta = utils.compose_image_meta(
202         0, image.shape, molded_image.shape, window, scale,
203         np.zeros([self.num_class], dtype=np.int32))
204     #将信息添加到列表
205     molded_images.append(molded_image)
206     windows.append(window)
207     image_metas.append(image_meta)
208
209     #转成 np 数组
210     molded_images = np.stack(molded_images)
211     image_metas = np.stack(image_metas)
212     windows = np.stack(windows)
213
214     return molded_images, image_metas, windows
215
216 def build(self, mode): #构建 Mask R-CNN 模型的网络架构
217
218     #检查尺寸合法性
219     h, w = IMAGE_DIM, IMAGE_DIM;
220     if h / 2**6 != int(h / 2**6) or w / 2**6 != int(w / 2**6):
221         raise Exception("必须要被2的6次方整除.例如: 256, 320, 384, 448,
512, ... 等. ")
222
223     input_image = KL.Input( shape=[None, None, 3], name="input_image") # 定义输入节点
224
225     input_image_meta = KL.Input(shape=[img_meta_size], name="input_image_meta")
226
227     if mode == "inference": #将全局的锚点框输入
228         input_anchors = KL.Input(shape=[None, 4], name="input_anchors")
229
230     #构建骨干网络。返回最后5层的特征(5种尺度)，不使用BN，因为批次=1，非常小
231     _, C2, C3, C4, C5 = resnet_graph(input_image, BACKBONE, stage5=True,
train_bn=False)

```

从代码第 231 行可以看到，并没有将 5 种尺度的特征全部使用，而是将第 1 层的特征层的特征丢掉。原因是：第 1 层的特征相对变化较小，虽然信息丰富，但是相对精度较低。

## 2. 实现 utils 模块中相关的函数

在 MaskRCNN 类中用到了 3 个函数： resize\_image、compose\_image\_meta 和 mold\_image。实现方式请看以下代码：

代码 8-25 mask\_rcnn\_utils

```

01 import numpy as np
02 import tensorflow as tf
03 from tensorflow.keras import backend as K #载入Keras的后端实现
04 from collections import OrderedDict
05 import skimage.color
06 import skimage.io
07 import skimage.transform
08 mask_rcnn_model = __import__("8-24 mask_rcnn_model")
09 model = mask_rcnn_model
10
11 #Image mean (RGB)
12 MEAN_PIXEL = np.array([123.7, 116.8, 103.9])
13 def mold_image(images): #将图片均值化
14     return images.astype(np.float32) - MEAN_PIXEL
15
16 def unmold_image(normalized_images): #将均值化的图片还原
17     return (normalized_images + MEAN_PIXEL).astype(np.uint8)
18
19 #改变图片形状，mode 为 square 表示填充为正方形，大小为 max_dim
20 def resize_image(image, min_dim=None, max_dim=None, min_scale=None,
21                 mode="square"): #mode 为 pad64，支持被 64 整除；mode 为 crop，表示按
22     照 min_dim 变形
23
24     .....#由于代码过长，这里略过。请参考随书的配套代码
25     else:
26         raise Exception("Mode {} not supported".format(mode))
27     return image.astype(image_dtype), window, scale, padding, crop
28
29 #定义函数将图片信息组合起来
30 def compose_image_meta(image_id, original_image_shape, #原始图片尺寸
31                       image_shape, #image_shape 转化后图片尺寸
32                       window, #转化后的图片，除去补 0 后剩下的坐标
33                       scale, active_class_ids):
34     meta = np.array([
35         [image_id] + #size=1
36         list(original_image_shape) + #size=3
37         list(image_shape) + #size=3
38         list(window) + #size=4 (y1, x1, y2, x2)
39         [scale] + #size=1
40         list(active_class_ids) #size=num_classes
41     ])
42     return meta

```

## 8.7.8 代码实现：可视化 Mask R-CNN 模型骨干网络的特征输出

为了可以清晰地了解 ResNet 模型所输出的内容，通过代码向模型输入图片，并将结果显示出来。

### 1. 实现 utils 模块中相关的函数

在 utils 模块中实现了函数 run\_graph，用于输出 MaskRCNN 类中的指定模型节点信息。在函数 run\_graph 中，使用的是 tf.keras 接口的 function 函数将指定的网络节点输出（见代码第 64 行）。

函数 tf.keras.function 的用法与函数 tf.keras.model 的用法类似，具体如下：

- (1) 构建输入与输出的网络节点。
- (2) 将输入与输出的网络节点传入 tf.keras.function 函数，得到一个 kf 对象。kf 对象具有可调用（\_\_call\_\_）属性。
- (3) 调用对象 kf，并向里面传入具体的输入数据，这样便可实现指定节点的输出。

在本例中，直接将 MaskRCNN 类里的输入层作为函数 tf.keras.function 的输入节点，将参数 outputs 作为函数 tf.keras.function 的输出节点，并调用函数 tf.keras.function 构造出可调用对象 kf。接着便构造出输入数据，调用 kf 对象，输出 outputs 节点的计算结果。具体代码如下：

代码 8-25 mask\_rcnn\_utils（续）

```

42 def log(text, array=None): #输出 numpy 类型的对象信息
43     if array is not None:
44         text = text.ljust(25)
45         text += ("shape: {:20} min: {:10.5f} max: {:10.5f} {}".format(
46             str(array.shape),
47             array.min() if array.size else "",
48             array.max() if array.size else "",
49             array.dtype))
50     print(text)
51
52 #定义函数，运行子图
53 def run_graph(MaskRCNNobj, images, outputs, BATCH_SIZE, image_metas=None):
54
55     model = MaskRCNNobj.keras_model #取得模型
56     outputs = OrderedDict(outputs) #检查参数
57     for o in outputs.values():
58         assert o is not None
59
60     #通过 tf.keras 接口的 function 函数来运行图中的一部分
61     inputs = model.inputs
62     if model.uses_learning_phase and not isinstance(K.learning_phase(), int):
63         inputs += [K.learning_phase()]
64     kf = K.function(model.inputs, list(outputs.values()))
65
66     if image_metas is None: #检查 image_metas 参数

```

```

67     molded_images, image_metas, _ = MaskRCNNObj.mold_inputs(images)
68     else:
69         molded_images = images
70         image_shape = molded_images[0].shape
71
72     #根据图片形状获得锚点信息
73     anchors = MaskRCNNObj.get_anchors(image_shape) #根据图片大小获得锚点
74
75     #一张图片的锚点变成batch张图片，复制batch份
76     anchors = np.broadcast_to(anchors, (BATCH_SIZE,) + anchors.shape)
77     model_in = [molded_images, image_metas, anchors]
78
79     #运行模型
80     if model.uses_learning_phase and not isinstance(K.learning_phase(), int):
81         model_in.append(0.)
82     outputs_np = kf(model_in)
83
84     #将结果打包成字典
85     outputs_np = OrderedDict([(k, v) for k, v in zip(outputs.keys(), outputs_np)])
86
87     for k, v in outputs_np.items():
88         log(k, v)
89     return outputs_np

```

代码第 66~77 行，实现了列表对象 model\_in 的构建。该列表对象 model\_in 将作为可调用对象 kf 的输入数据在计算输出节点中使用。

可调用对象 kf 的输入数据格式应与 MaskRCNN 类里输入层节点的格式一致，它们由图片 (molded\_images)、图片元数据 (image\_metas)、锚点信息 (anchors) 这 3 个数据组成。

- **molded\_images:** 预处理过的图片数据。在代码第 66 行对参数 image\_metas 进行判断。

如果参数 image\_metas 为 None，则表示输入的图片 images 是原始图片，需要调用模型的 mold\_inputs 方法对图片进行预处理，并将缩放后的图片信息打包到参数 image\_metas 里；如果参数 image\_metas 不为 None，则表示输入的图片 images 已被预处理过，可以直接使用。

- **image\_metas:** 图片的元数据，记录着图片在预处理过程中的附属信息。
- **anchors:** 图片的锚点信息。它根据是输入图片的形状计算得来的，由模型的 get\_anchors 方法生成。生成规则见 8.7.10 小节的详细介绍。

在代码第 82 行，将列表对象 model\_in 传入可调用对象 kf 中，进行输出节点的计算（得到结果 outputs\_np）。接着将最终的计算结果 outputs\_np 输出，并返回。

## 2. 获取图片

从数据集中随机取出一张图片，作为模型的原始输入数据输入 MaskRCNN 类中，用来计算 ResNet 层所抽取的特征。代码如下：

**代码 8-23 Mask\_RCNN 应用（续）**

```

44 #从数据集中获取一个图片用于测试
45 catIds = coco.getCatIds(catNms=['person'])          #根据类名获得对应的图片列表
46 imgIds = coco.getImgIds(catIds=catIds )
47 print(catIds,len(imgIds),imgIds[:5])
48
49 #从指定列表中取一张图片
50 index = imgIds[np.random.randint(0,len(imgIds))]
51 print(index)
52 img = coco.loadImgs(index)[0]                         #index 可以是数组，会返回多个图片
53 print(img)
54 image = io.imread(img['coco_url'])
55 plt.axis('off')
56 plt.imshow(image)
57 plt.show()

```

代码运行后，输出以下结果：

```

[1] 45174 [262145, 262146, 524291, 393223, 393224]
227612
{'license': 2, 'file_name': 'COCO_train2014_000000227612.jpg', 'coco_url':
'http://images.cocodataset.org/train2014/COCO_train2014_000000227612.jpg', 'height':
333, 'width': 500, 'date_captured': '2013-11-19 23:55:59', 'flickr_url':
'http://farm3.staticflickr.com/2646/3916774397_6f358fa220_z.jpg', 'id': 227612}

```

输出结果的第 1 行的意义是：在 COCO 数据集中，person 类的索引为 1。该类有 45174 条标注信息，以及 person 类中前 5 条标注的索引值。

输出结果的第 2 行，对应于代码第 51 行的运行结果。意思是：在 person 类的标注数据中，随机取出一条索引值为 227612 的标注数据。

输出结果的第 3 行显示索引值为 227612 的标注数据的具体内容。

在代码第 54 行，调用函数 `io.imread`，并根据标注信息中的网址将图片下载到内存中并显示出来，如图 8-38 所示。



图 8-38 COCO 数据集中的人物图片

### 3. 运行 MaskRCNN 子图，验证 ResNet 输出

下面用 run\_graph 函数将 ResNet 的最后两层网络的特征值打印出来，并进行可视化。代码如下：

代码 8-23 Mask\_RCNN 应用（续）

```

58 ResNetFeatures = utils.run_graph(model, [image], [
59     ("res4w_out",
60      model.keras_model.get_layer("res4w_out").output),
61     ("res5c_out",
62      model.keras_model.get_layer("res5c_out").output),
63 ], BATCH_SIZE)
64
65 # 可视化
66 visualize.display_images(np.transpose(ResNetFeatures["res4w_out"][:, :, :, :4], [2, 0, 1]))
67
68 visualize.display_images(np.transpose(ResNetFeatures["res5c_out"][:, :, :, :4], [2, 0, 1]))

```

在构建 ResNet 模型的代码中（见 8.7.6 小节），为每个特征层的残差块都定义了一个名字。代码第 59 行，用 model.keras\_model.get\_layer 方法，根据残差块的名字取出第 4 特征层输出的张量。该张量将传入 run\_graph 函数，计算出具体的特征数据。

代码第 64、65 行，用 visualize 模块的 display\_images 函数，分别从第 4、5 特征层的输出结果中取出 4 张特征数据，并将其可视化（visualize 模块是一个可视化代码模块，本书不做具体讲解）。

整个代码运行后，输出以下结果：

```

res4w_out      shape: (1, 64, 64, 1024)      min:  0.00000  max:  78.48668  float32
res5c_out      shape: (1, 32, 32, 2048)      min:  0.00000  max:  70.40952  float32

```

输出结果中的第 1、2 行分别是第 4、5 特征层的输出。从形状上可以看到，第 4 特征层将原始图片（1024 pixel×1024 pixel）做了 4 次缩小一半的操作（ $1024 \div 4^2 = 64$ ），而第 5 特征层将原始图片做了 5 次缩小一半的操作。

接着还会看到输出的特征图片，如图 8-39 所示。



图 8-39 ResNet 模型的第 4、5 特征层的输出结果

从图 8-39 中可以看到，模型中第 4、5 层的特征数据能够关注到图片中的某些特殊区域。

### 8.7.9 代码实现：用特征金字塔网络处理骨干网络特征

在特征提取过程中，骨干网模型的最终层特征与中间层特征有以下特点：

- 最终特征层，输出的特征语义信息比较少，但指向收敛目标的特征相对精准。
- 中间特征层，含有的特征语义信息比较丰富，但指向收敛目标的特征相对比较粗略。

特征金字塔网络（Feature Pyramid Networks, FPN）是目标检测模型中的一个经典网络，它可以对骨干网络模型做更好的特征提取。用 FPN 提取出来的特征能够兼顾最终层和中间层特征的优点，使预测效果更好。

FPN 的原理是：将骨干网络最终特征层和中间特征层的多个尺度的特征以类似金字塔的形式融合在一起。最终的特征可以兼顾两个特点——指向收敛目标的特征准确、特征语义信息丰富。更多信息可以参考论文：

<https://arxiv.org/abs/1612.03144>

具体方式如图 8-40 所示。

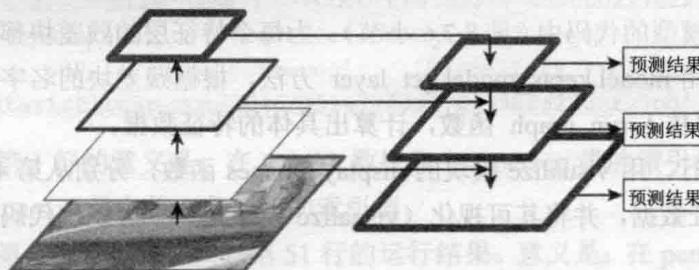


图 8-40 FPN 的结构

#### 1. FPN 的代码实现

接着 8.7.7 小节，在 MaskRCNN 类中添加以下代码：

#### 代码 8-24 mask\_rcnn\_model (续)

```

232     # 实现特征金字塔层 FPN
233     P5 = KL.Conv2D(256, (1, 1), name='fpn_c5p5')(C5)
234     P4 = KL.Add(name="fpn_p4add") ([KL.UpSampling2D(size=(2, 2),
235                                         name="fpn_p4upsampled")(P5),
236                                         KL.Conv2D(256, (1, 1),
237                                         name='fpn_c4p4')(C4)])
236     P3 = KL.Add(name="fpn_p3add") ([KL.UpSampling2D(size=(2, 2),
237                                         name="fpn_p4upsampled")(P4),
238                                         KL.Conv2D(256, (1, 1),
239                                         name='fpn_c3p3')(C3)])
238     P2 = KL.Add(name="fpn_p2add") ([KL.UpSampling2D(size=(2, 2),
239                                         name="fpn_p3upsampled")(P3),
240                                         KL.Conv2D(256, (1, 1),
241                                         name='fpn_c2p2')(C2)])
240
241     # 为每层添加一个分类头
242     P5 = KL.Dense(1, activation='sigmoid')(P5)
243     P4 = KL.Dense(1, activation='sigmoid')(P4)
244     P3 = KL.Dense(1, activation='sigmoid')(P3)
245     P2 = KL.Dense(1, activation='sigmoid')(P2)
246
247     return Model(inputs=inputs, outputs=[P5, P4, P3, P2])

```

```

239                               KL.Conv2D(256, (1, 1),
240   name='fpn_c2p2'))(C2) ] )
241   #依次对融合后的特征进行卷积操作
242   P2 = KL.Conv2D(FPN_FEATURE, (3, 3), padding="SAME",
243   name="fpn_p2")(P2)
244   P3 = KL.Conv2D(FPN_FEATURE, (3, 3), padding="SAME",
245   name="fpn_p3")(P3)
246   P4 = KL.Conv2D(FPN_FEATURE, (3, 3), padding="SAME",
247   name="fpn_p4")(P4)
248   P5 = KL.Conv2D(FPN_FEATURE, (3, 3), padding="SAME",
249   name="fpn_p5")(P5)
250   #额外再将 P5 进行下采样，生成一个 P6 特征，用于 RPN
251   P6 = KL.MaxPooling2D(pool_size=(1, 1), strides=2, name="fpn_p6")(P5)
252   #将特征准备好后，分别放到两个列表里，用于 RPN 处理及最终的分类器
253   rpn_feature_maps = [P2, P3, P4, P5, P6]#定义列表，用于 rpn 处理
254   mrcnn_feature_maps = [P2, P3, P4, P5]#定义列表，用于分类器

```

在代码的最后两行，分别将做好的特征放入 `rpn_feature_maps` 与 `mrcnn_feature_maps` 两个列表中。

代码第 242~245 行，用 `KL.Conv2D` 函数依次对 `P2`、`P3`、`P4`、`P5` 这 4 个融合后的特征数据做卷积操作，将这 4 个特征数据的深度都变为 `FPN_FEATURE`(256) 个通道。

## 2. FPN 的结果查看

用 `utils.run_graph` 函数将 FPN 中的各个尺度特征都打印出来。

具体代码如下：

代码 8-23 Mask\_RCNN 应用（续）

```

66 roi_align_mask = utils.run_graph(model, [image], [
67   ("fpn_p2",           model.keras_model.get_layer("fpn_p2").output), #输出 fpn_p2 层
68   ("fpn_p3",           model.keras_model.get_layer("fpn_p3").output), #输出 fpn_p3 层
69   ("fpn_p4",           model.keras_model.get_layer("fpn_p4").output), #输出 fpn_p4 层
70   ("fpn_p5",           model.keras_model.get_layer("fpn_p5").output), #输出 fpn_p5 层
71   ("fpn_p6",           model.keras_model.get_layer("fpn_p6").output), #输出 fpn_p6 层
72 ], BATCH_SIZE)

```

代码运行后，输出以下结果：

<code>fpn_p2</code>	<code>shape: (1, 256, 256, 256)</code>	<code>min: -27.07274</code>	<code>max: 28.28360</code>	<code>float32</code>
<code>fpn_p3</code>	<code>shape: (1, 128, 128, 256)</code>	<code>min: -33.81137</code>	<code>max: 31.17183</code>	<code>float32</code>
<code>fpn_p4</code>	<code>shape: (1, 64, 64, 256)</code>	<code>min: -40.54855</code>	<code>max: 36.91861</code>	<code>float32</code>
<code>fpn_p5</code>	<code>shape: (1, 32, 32, 256)</code>	<code>min: -35.25085</code>	<code>max: 44.75595</code>	<code>float32</code>

```
fpn_p6      shape: (1, 16, 16, 256)      min: -33.35557 max: 43.78230 float32
```

从输出结果中可以看出，不同的尺度的特征层有相同的深度，但高度  $h$  和宽度  $w$  是不断减半的。

在两阶段的识别模型中，经过 FPN 后的特征数据会被放在 mrcnn\_feature\_maps 列表里，并按以下步骤进行使用：

- (1) 通过 RPN 对 rpn\_feature\_maps 列表进行计算，得出相对靠谱的 ROI 区域。
- (2) 在 mrcnn\_feature\_maps 列表中，为每个 ROI 区域找到与其匹配的特征数据。
- (3) 在 ROI 区域对应的特征数据上，按照 ROI 区域的尺寸取出相应的 ROI 特征数据。
- (4) 将第 (3) 步的结果输入分类器中，算出最终结果。

### 8.7.10 计算 RPN 中的锚点

计算 RPN 的锚点分为两步：计算中心点和计算边长。具体如下：

#### 1. 计算中心点

计算中心点的计算方式有以下几点需说明：

- rpn\_feature\_maps 列表中的 5 个特征都是  $w$  与  $h$  相等的正方形，边长值依次是 256、128、64、32、16。
- 这 5 个特征的边长对应于原始图片（边长值是 1024）的缩小比例分别是 4、8、16、32、64。
- 假设把缩小的比例当作图片上的像素点，那么这 5 个特征可以理解成：将  $1024 \times 1024$  pixel 大小的图片按照具体的像素点分割成多个网格。

以第 5 特征为例，在图片上分了  $16 \times 16$  个网格，每个网格的大小是  $64 \text{ pixel} \times 64 \text{ pixel}$ 。步长与中心点的关系如下：

- 如果步长为 1，则每个网格的左上角第 1 个元素被当作锚点的中心点。
- 如果步长为 2，则每隔一个网格的下一个网格左上角第 1 个元素被当作锚点的中心点。
- 如果步长为 3，则每隔两个网格的下一个网格左上角第 1 个元素被当作锚点的中心点。
- 如果步长为 4，以此类推。

#### 2. 计算边长

计算边长的具体步骤如下：

- (1) 给出一个预设值，即宽与高的比例。这里使用的比例 ( $w/h$ ) 为 RPN\_ANCHOR\_RATIOS = [0.5, 1]，即 3 种形状。
- (2) 将全部的网格复制 3 份，每种形状各对应一份网格。

#### 3. 锚点的组成与作用

锚点的基本信息由中心点和边长组成，一共四个值 ( $x, y, h, w$ )。

在实际应用中，锚点也会被用左上和右下两个点来表示，即 ( $x_1, y_1, x_2, y_2$ )。

在网络运算的中间状态，还会根据锚点的基本位置信息，通过计算其偏移量（中心点的平

移量与边长的缩放量)来修正位置。

在本实例中,总的锚点个数为:

$$(256 \times 256 + 128 \times 128 + 64 \times 64 + 32 \times 32 + 16 \times 16) \times 3 = 261888$$

### 8.7.11 代码实现: 构建 RPN

RPN的工作原理如下:

- (1) 从预先设定好的锚点(261888个)中找出可能包含实例的锚点区域(在8.7.3小节中称它为靠谱区域ROI)。
- (2) 计算出其中心点和对应的边长。
- (3) 用NMS算法对第(2)步的结果进行去重。

#### 1. 将RPN接入MaskRCNN类

继续在MaskRCNN类中添加代码。构建RPN,并将rpn\_feature\_maps列表中的特征依次传入RPN中进行计算。具体代码如下:

代码 8-24 mask\_rcnn\_model(续)

```

251     # 定义 RPN 模型, 该模型会生成前后景的概率
252     rpn = build_rpn_model(RPN_ANCHOR_STRIDE, len(RPN_ANCHOR_RATIOS),
253                           FPN_FEATURE)                                # 每个尺度的特征都是 256
254     layer_outputs = []                               # 定义列表, 用来保存 RPN 结果
255     for p in rpn_feature_maps:                      # 依次将特征送入 RPN 中进行计算
256         layer_outputs.append(rpn([p]))              # 将 RPN 的输出结果放到 layer_outputs 中
257
258     # 用 concat 函数将结果连在一起
259     # 例如 [[a1, b1, c1], [a2, b2, c2]] => [[a1, a2], [b1, b2], [c1, c2]]
260     output_names = ["rpn_class_logits", "rpn_class", "rpn_bbox"]
261     outputs = list(zip(*layer_outputs))
262     outputs = [KL.Concatenate(axis=1, name=n)(list(o)) for o, n in
263                zip(outputs, output_names)]
264     rpn_class_logits, rpn_class, rpn_bbox = outputs

```

代码第252行,用build\_rpn\_model函数生成RPN。接着,依次将特征放入RPN中(见代码第256行),最终将结果连接到一起。

#### 2. 构建RPN

定义build\_rpn\_model函数,构建RPN。具体代码如下:

代码 8-27 othernet

```

01 import tensorflow as tf
02 from tensorflow.keras import layers as KL
03 from tensorflow.keras import models as KM

```

```

04 from tensorflow.keras import backend as K #载入Keras的后端实现框架
05 import numpy as np
06 utils = __import__("8-25 mask_rcnn_utils")
07 mask_rcnn_model = __import__("8-24 mask_rcnn_model")
08
09 #构建RPN图结构一共分为两部分：1-计算分数，2-计算边框
10 def rpn_graph(feature_map, #输入的特征，其宽和高所围成区域的个数为锚点的个数
11                 anchors_per_location, #每个待计算锚点的网格需要划分为几种形状的矩形
12                 anchor_stride):#扫描网格的步长
13
14     #通过一个卷积得到共享特征
15     shared = KL.Conv2D(512, (3, 3), padding='same', activation='relu',
16                         strides=anchor_stride, name='rpn_conv_shared')(feature_map)
17
18     #第1部分计算锚点的分数（前景和背景）[batch, height, width, anchors per
19     #location * 2]
20     x = KL.Conv2D(2 * anchors_per_location, (1, 1), padding='valid',
21                   activation='linear', name='rpn_class_raw')(shared)
22
23     #将 feature_map 展开，得到 [batch, anchors, 2]。anchors 的值是 feature_map 形
24     #状的 h、w 与 anchors_per_location 这三个维度的乘积
25     rpn_class_logits = KL.Lambda(lambda t: tf.reshape(t, [tf.shape(t)[0], -1,
26 ]))(x)
27
28     #用 Softmax 来分类前景和背景 BG/FG，结果代表预测的分数
29     rpn_probs = KL.Activation(
30         "softmax", name="rpn_class_xxx")(rpn_class_logits)
31
32     #第2部分计算锚点的边框，每个网格划分 anchors_per_location 种矩形框，每种 4 个值
33     x = KL.Conv2D(anchors_per_location * 4, (1, 1), padding="valid",
34                   activation='linear', name='rpn_bbox_pred')(shared)
35
36     #将 feature_map 展开，得到 [batch, anchors, 4]
37     rpn_bbox = KL.Lambda(lambda t: tf.reshape(t, [tf.shape(t)[0], -1, 4]))(x)
38
39     return [rpn_class_logits, rpn_probs, rpn_bbox]
40
41 def build_rpn_model(anchor_stride, #扫描网格的步长
42                     anchors_per_location, #每个待计算锚点的网格需要划分为几种形状的
43                     矩形
44                     depth): #输入的特征有多少个
45
46     input_feature_map = KL.Input(shape=[None, None,
47                                     depth], name="input_rpn_feature_map")
48     outputs = rpn_graph(input_feature_map, anchors_per_location,
49                         anchor_stride)
50
51     return KM.Model([input_feature_map], outputs, name="rpn_model")

```

代码第38行，在`build_rpn_model`函数中定义了一个输入层`input_feature_map`，用于输入特征，然后用`rpn_graph`函数构建RPN。

在`rpn_graph`函数中分为两部分：计算锚点的前景与背景概率值（见代码第18~27行），计算锚点的边框`rpn_bbox`，（见代码第30~34行）。

其中，边框`rpn_bbox`表示实物坐标相对于原始锚点坐标的偏移量（相对中心点的偏移）与缩放值（相对边长的缩放比例）。

### 8.7.12 代码实现：用非极大值抑制算法处理RPN的结果

定义`ProposalLayer`类，将非极大值抑制算法封装起来，并用`ProposalLayer`类对RPN的结果进行去重。

`ProposalLayer`类会从RPN的结果中选取前景分值最大的n个结果（n可以事先指定）保留下来，并传入下一层。具体实现如下。

#### 1. 在MaskRCNN类中处理RPN的结果

在MaskRCNN类中，具体操作如下：

(1) 对参数`proposal_count`赋值，使其等于`POST_NMS_ROIS_INFERENCE`。

(2) 定义非极大值抑制算法的处理对象：向`ProposalLayer`类中传入参数`proposal_count`和`RPN_NMS_THRESHOLD`，得到实例化后的非极大值抑制算法处理对象。

其中，传入的参数如下。

- `proposal_count`: 在去重时需要保留的结果个数。
- `RPN_NMS_THRESHOLD`: NMS算法对结果去重时的阈值。

该对象可以返回前景概率值最大的`proposal_count`个ROI（靠谱区域）。

(3) 用非极大值抑制算法对RPN的结果进行去重：将参数`rpn_class`、`rpn_bbox`、`anchors`传入非极大值抑制算法的处理对象，得到去重后的RPN的结果该结果存储在`rpn_rois`对象中。

其中，传入的参数如下。

- `rpn_class`: RPN的分类结果，即前景和背景的分类分值。
- `rpn_bbox`: RPN的矩形框结果，即每个边框的修正值。
- `anchors`: 原始锚点的矩形框信息。

具体代码如下：

代码8-24 mask\_rcnn\_model(续)

```

265     proposal_count = POST_NMS_ROIS_TRAINING if mode == "training" else
266     POST_NMS_ROIS_INFERENCE
267
268     # 定义锚点输入
269     if mode == "inference":
270         anchors = input_anchors
271
272     # 返回NMS去重后前景概率值最大的n个ROI

```

```

271     rpn_rois = ProposalLayer(proposal_count=proposal_count,
272                               nms_threshold=RPN_NMS_THRESHOLD, batch_size=self.batch_size,
273                               name="ROI")([rpn_class, rpn_bbox, anchors])

```

代码最后一行，将 RPN 结果中的 proposal\_count (1000) 个前景概率值最高的 ROI (靠谱区域) 返回到 rpn\_rois 对象中。

## 2. 实现 RPN 的结果处理类 ProposalLayer

ProposalLayer 类是在代码文件“8-27othernet.py”中实现的。ProposalLayer 类是用 tf.keras 接口实现的一个网络层（有关使用 tf.keras 接口自定义网络层的详细说明见 8.4 节）。在 ProposalLayer 类的 call 方法里实现了该网络层的处理流程：

- (1) 将概率值的由高到低排序，取出前 6000 个结果。
- (2) 用函数 apply\_box\_deltas\_graph，对每个结果的偏移坐标 deltas 在对应的锚点框 anchors 上做偏移运算，合成矩形框坐标（见代码第 116 行）。
- (3) 用函数 clip\_boxes\_graph 对合成的坐标进行二次处理，剪掉坐标中超出边界的部分。
- (4) 用 NMS（参考 8.7.9 小节）算法去重。



### 提示：

在上述过程中，第（2）、（3）步都是基于标准化坐标进行操作的。这是由于，在程序中锚点 anchors 的坐标是以标准化坐标的形式存在的。

在 MaskRCNN 类的 get\_anchors 方法中计算出锚点 anchors 之后，又将其像素坐标转化成了标准化坐标。具体代码在 8.7.22 小节。

具体代码如下：

代码 8-27 othernet（续）

```

45 #按照给定的框与偏移量计算最终的框
46 def apply_box_deltas_graph(boxes,    #[N, (y1, x1, y2, x2)]
47                           deltas):      #[N, (dy, dx, log(dh), log(dw))]
48
49     #转换成中心点和h、w格式
50     height = boxes[:, 2] - boxes[:, 0]
51     width = boxes[:, 3] - boxes[:, 1]
52     center_y = boxes[:, 0] + 0.5 * height
53     center_x = boxes[:, 1] + 0.5 * width
54
55     #计算偏移
56     center_y += deltas[:, 0] * height
57     center_x += deltas[:, 1] * width
58     height *= tf.exp(deltas[:, 2])
59     width *= tf.exp(deltas[:, 3])
60
61     #转成左上、右下两个点：y1, x1, y2, x2
62     y1 = center_y - 0.5 * height
63     x1 = center_x - 0.5 * width
64     y2 = y1 + height

```

```

63     x2 = x1 + width
64     result = tf.stack([y1, x1, y2, x2], axis=1, name="apply_box_deltas_out")
65     return result
66
67 #将框坐标限制在0~1之间
68 def clip_boxes_graph(boxes,      #计算完的box[N, (y1, x1, y2, x2)]
69             window):        #y1, x1, y2, x2[0, 0, 1, 1]
70
71     #获取坐标
72     wy1, wx1, wy2, wx2 = tf.split(window, 4)
73     y1, x1, y2, x2 = tf.split(boxes, 4, axis=1)
74     #剪辑
75     y1 = tf.maximum(tf.minimum(y1, wy2), wy1)
76     x1 = tf.maximum(tf.minimum(x1, wx2), wx1)
77     y2 = tf.maximum(tf.minimum(y2, wy2), wy1)
78     x2 = tf.maximum(tf.minimum(x2, wx2), wx1)
79     clipped = tf.concat([y1, x1, y2, x2], axis=1, name="clipped_boxes")
80     clipped.set_shape((clipped.shape[0], 4))
81     return clipped
82
83 class ProposalLayer(tf.keras.layers.Layer):      #定义RPN的最终处理层
84
85     def __init__(self, proposal_count, nms_threshold, batch_size, **kwargs):
86         super(ProposalLayer, self).__init__(**kwargs)
87         self.proposal_count = proposal_count
88         self.nms_threshold = nms_threshold
89         self.batch_size = batch_size
90
91     def call(self, inputs):
92         """
93             输入字段 input 描述
94             rpn_probs: [batch, num_anchors, 2]
95             rpn_bbox: [batch, num_anchors, (dy, dx, log(dh), log(dw))]
96             anchors: [batch, (y1, x1, y2, x2)]
97             """
98             #从形状为 [batch, num_anchors, 1] 的数据中取出前景概率值
99             scores = inputs[0][:, :, 1] #scores 的形状为 [batch, num_anchors]
100            #取出位置偏移量 [batch, num_anchors, 4]
101            deltas = inputs[1]
102            deltas = deltas * np.reshape(mask_rcnn_model.RPN_BBOX_STD_DEV, [1, 1,
103                                         4])
103            #取出锚点 Anchors
104            anchors = inputs[2]
105
106            #获得前 6000 个分值最大的数据
107            pre_nms_limit = tf.minimum(6000, tf.shape(anchors)[1])

```

```

108     ix = tf.nn.top_k(scores, pre_nms_limit, sorted=True,
109     name="top_anchors").indices
110     #获取 scores 中索引为 ix 的值
111     scores = utils.batch_slice([scores, ix], lambda x, y: tf.gather(x,
112         y), self.batch_size)
113     deltas = utils.batch_slice([deltas, ix], lambda x, y: tf.gather(x,
114         y), self.batch_size)
115     pre_nms_anchors = utils.batch_slice([anchors, ix], lambda a, x:
116         tf.gather(a, x),
117             4 self.batch_size, names=["pre_nms_anchors"])
118
119     #得出最终的框坐标。其形状为 [batch, N, 4]
120     boxes = utils.batch_slice([pre_nms_anchors, deltas],
121         lambda x, y: apply_box_deltas_graph(x, y), self.batch_size,
122             names=["refined_anchors"])
123
124     #对出界的 box 进行剪辑，范围控制在 0.0~1.0，其形状为 [batch, N, (y1, x1, y2,
125     x2)]
126     window = np.array([0, 0, 1, 1], dtype=np.float32)
127     boxes = utils.batch_slice(boxes, lambda x: clip_boxes_graph(x, window),
128         self.batch_size,
129             names=["refined_anchors_clipped"])
130
131     #Non-max suppression 算法
132     def nms(boxes, scores):
133         indices = tf.image.non_max_suppression(boxes, scores,
134         self.proposal_count,
135         self.nms_threshold, name="rpn_non_max_suppression")#计算NMS，并获得索引
136         proposals = tf.gather(boxes, indices) #从boxes中取出indices索引所
137         指的值
138         #如果 proposals 的个数小于 proposal_count，则剩下的补0
139         padding = tf.maximum(self.proposal_count - tf.shape(proposals)[0],
140         0)
141         proposals = tf.pad(proposals, [(0, padding), (0, 0)])
142         return proposals
143
144         proposals = utils.batch_slice([boxes, scores], nms, self.batch_size)
145
146         return proposals
147
148     def compute_output_shape(self, input_shape):
149         return (None, self.proposal_count, 4)

```

代码第 91 行是 ProposalLayer 类的 call 方法实现。call 方法的输入参数 inputs 是一个数组。inputs 数组的第一个元素是 RPN 的返回结果 (rpn\_probs 对象)。

在 rpn\_probs 对象的形状 [batch,num\_anchors,2] 中，最后一维有两个元素，代表背景概率值和前景概率值。

代码第 99 行的操作可以理解成以下步骤：

- (1) 从 inputs 数组中取出第 1 个元素 rpn\_probs 对象。
- (2) 从 rpn\_probs 对象中取出最后一维索引值是 1 的元素，得到前景概率值。



**提示：**

在 rpn\_probs 对象中，最后一维索引值与前景、背景的对应关系是在模型训练时设置的。在用训练好的模型进行预测时，这个索引值必须与训练时的设置一致，否则模型将无法输出正确的结果。

代码第 108 行，用函数 tf.nn.top\_k 从前景概率值 scores 中找出分值最大的前 pre\_nms\_limit 个索引。



**提示：**

函数 tf.nn.top\_k 的作用是：在多维数组的最后一维中找出最大的 k 个元素。该函数会以列表的方式返回元素的值和索引。具体用法见以下代码：

```
import tensorflow as tf
import numpy as np
tf.enable_eager_execution()      #启动动态图（在TensorFlow 2.x中可以去掉该句）

scores = np.array([[4, 5, 3, 4],    #假设 scores 的 batch 为 2，每行有 4 个分值
                  [10, 60, 80, 50]])
print(np.shape(scores))          #输出 scores 的形状：(2, 4)
top_k=tf.nn.top_k(scores,2)       #在 scores 的每行中查找最大的两个数
print(top_k.values.numpy())       #输出[[ 5  4] [80 60]]
print(top_k.indices.numpy())      #输出 [[1 0] [2 1]]
```

代码第 110 行，在前景概率值 scores 中按照索引 ix 取值。

代码第 111 行，在前景的位移偏移量 deltas 中按照索引 ix 取值。



**提示：**

代码第 110、111 行都用函数 tf.gather 按照指定的索引从数据中取值。因为函数 tf.gather 不支持批量处理数据，所以又在外层用 utils.batch\_slice 函数进行转换。utils.batch\_slice 函数可以将数据按照批次拆开，单独放到指定的函数里去处理，并将处理后的结果组合成批次数据。下面对函数 tf.gather 和函数 utils.batch\_slice 进行详细说明。

- (1) 函数 tf.gather 的详细说明。

函数 tf.gather 可以在张量中按照指定的索引获取数据，与 Python 中的切片操作类似。具体使用见以下代码：

```
import tensorflow as tf
```

```

import numpy as np
tf.enable_eager_execution()          #启动动态图(在TensorFlow 2.x中可以去掉该句)
#假设 deltas 的 batch 为 2，则每行有 4 个坐标
deltas = np.array([[[1,2,3,4], [2,2,3,4], [3,2,3,4], [4,2,3,4]],
                  [[5,6,7,8], [2,6,7,8], [3,6,7,8], [4,6,7,8]]])
ix = np.array([[1,0],[2,1]])          #定义索引
for data,i in zip(deltas,ix):        #模拟 utils.batch_slice 的处理，将批次拆开
    print(data[i])                  #用切片获取数据，输出：[[2 2 3 4] [1 2 3 4]]
    print(tf.gather(data,i).numpy())  #调用获取数据，输出：[[2 2 3 4] [1 2 3 4]]
    break

```

从程序的输出结果中可以看到，用 `tf.gather` 函数取出的值与 Python 切片方式取出的值完全一样。不过函数 `tf.gather` 只能指定一个维度进行取值。如果要指定多维度进行取值，则可以用 `tf.gather_nd` 函数（该函数的用法见 8.7.20 小节）。

### (2) 函数 `utils.batch_slice` 的详细说明。

函数 `utils.batch_slice` 的作用是：将第 1 个参数中的元素按照批次个数，依次输入第 2 个参数所代表的函数中（具体代码实现请参考配套资源中的代码文件“8-25\_mask\_rcnn\_utils”）。它可以让程序兼容输入批次小于 2 和大于等于 2 的两种情况，但是以牺牲效率为代价的。如果模型节点中的所有网络层都支持批次大于 2 的输入，则可以直接将 `utils.batch_slice` 函数去掉。

代码第 102 行，将偏移量 `deltas` 与标准差 `RPN_BBOX_STD_DEV` 相乘，并将得到的值再次赋给变量 `deltas`。此时的变量 `deltas` 变成了基于标准化的偏移坐标。



#### 提示：

代码第 102 行使用的是标准差 `RPN_BBOX_STD_DEV` 是模型在训练时设置的。因为在训练模型过程中，制作 RPN 标签时将每个前景标签的偏移坐标进行了归一化处理（即除以了标准差 `RPN_BBOX_STD_DEV`，见 8.8.4 小节），所以在模型输出偏移量之后，还需要将其乘以 `RPN_BBOX_STD_DEV`，还原成归一化处理之前的真实偏移坐标。

### 8.7.13 代码实现：提取 RPN 的检测结果

调用 `run_graph` 函数，并传入图片 `image`。该函数会计算出 RPN 的检测结果 `rpn_class` 和 `ProposalLayer` 层返回的结果。

具体代码如下：

## 代码 8-23 Mask\_RCNN 应用(续)

```

73 pillar = model.keras_model.get_layer("ROI").output #获得 ROI 节点, 即
    ProposalLayer 层
74
75 rpn = utils.run_graph(model, [image], [
76     ("rpn_class", model.keras_model.get_layer("rpn_class").output), #(1,
    261888, 2)
77     ("pre_nms_anchors", model.ancestor(pillar, "ROI/pre_nms_anchors:0")),
78     ("refined_anchors", model.ancestor(pillar, "ROI/refined_anchors:0")),
79     ("refined_anchors_clipped", model.ancestor(pillar,
    "ROI/refined_anchors_clipped:0")),
80     ("post_nms_anchor_ix", model.ancestor(pillar,
    "ROI/rpn_non_max_suppression/NonMaxSuppressionV3:0") ),#shape: (1000,)
81     ("proposals", model.keras_model.get_layer("ROI").output),
82 ],BATCH_SIZE)

```

代码运行后, 输出以下结果:

rpn_class	shape: (1, 261888, 2)	min: 0.00000	max: 1.00000
float32			
pre_nms_anchors	shape: (1, 6000, 4)	min: -0.35390	max: 1.29134
float32			
refined_anchors	shape: (1, 6000, 4)	min: -2.48665	max: 3.46406
float32			
refined_anchors_clipped	shape: (1, 6000, 4)	min: 0.00000	max: 1.00000
float32			
post_nms_anchor_ix	shape: (1000,)	min: 0.00000	max: 3886.00000
int32			
proposals	shape: (1, 1000, 4)	min: 0.00000	max: 1.00000
float32			

在结果的第1行中, rpn\_class 是 RPN 对所有锚点的前景/背景进行分类的分值, 其形状是(1, 261888, 2), 表示一共 261888 个 ROI (靠谱区域)、2 个分类。

第2、3、4 行是将 rpn\_class 中前景分值最大的前 6000 个 ROI (靠谱区域) 取出, 并按照其索引找到前 6000 个 ROI 对应的原始锚点 pre\_nms\_anchors、修正后的边框 refined\_anchors、剪辑后的边框 refined\_anchors\_clipped。

最后两行是通过 NMS 算法处理后的结果: post\_nms\_anchor\_ix 是 NMS 算法所返回的 proposal\_count (1000) 个索引值; proposals 是按照 post\_nms\_anchor\_ix 索引值返回的被剪辑后的 bbox 框, 见 8.7.12 小节“2. 实现 RPN 的结果处理类 ProposalLayer”的代码第 129 行。



## 提示:

这里解释一个疑点:在 8.7.12 小节定义了一个输入层,用于将原始锚点输入 ProposalLayer 层;但是在 8.7.13 小节运行子图 ProposalLayer 时只输入了一个图片,并没有输入原始锚点(见 8.7.13 小节代码第 68 行)。为什么程序可以工作呢?

原因是:在 utils.run\_graph 函数的内部已经实现了获取原始锚点的操作(见 8.7.8 小节的

“1. 实现 utils 模块中相关的函数”代码第 73 行)。在运行子图 ProposalLayer 时, 会将锚点与图片的信息一起组成输入参数传入模型中。

其中, 生成锚点部分调用了 MaskRCNN 类中的 get\_anchors 方法, 具体代码在 8.6.22 小节的“2. 实现 MaskRCNN 类锚点生成”。

在 get\_anchors 方法里, 将计算好的锚点框(像素坐标)存入 MaskRCNN 类的成员变量 anchors 中, 并将像素坐标转化为标准化坐标并返回用于输入 ProposalLayer 层。

## 8.7.14 代码实现: 可视化 RPN 的检测结果

下面通过代码可视化 RPN 中各个环节的检测结果。

### 1. 可视化 RPN 返回的前景锚点

在 RPN 字典里的 pre\_nms\_anchors 元素中存放着 6000 个锚点。这 6000 个锚点是按照前景概率值从大到小排列的。

编写代码, 将 RPN 字典里的 pre\_nms\_anchors 元素中的前 50 个锚点取出, 并在图中显示出来。具体代码如下:

代码 8-23 Mask\_RCNN 应用(续)

```

83 def get_ax(rows=1, cols=1, size=16):#设置显示图片的位置及大小
84     _, ax = plt.subplots(rows, cols, figsize=(size*cols, size*rows))
85     return ax
86 #将分值高的前 50 个锚点显示出来
87 limit = 50
88 h, w = mask_rcnn_model.IMAGE_DIM,mask_rcnn_model.IMAGE_DIM;
89 pre_nms_anchors = rpn['pre_nms_anchors'][0, :limit] * np.array([h, w, h, w])
90 print(image.shape)
91 image2, window, scale, padding, _ = utils.resize_image( image,
92                                         min_dim=mask_rcnn_model.IMAGE_MIN_DIM,
93                                         max_dim=mask_rcnn_model.IMAGE_MAX_DIM,
94                                         mode=mask_rcnn_model.IMAGE_RESIZE_MODE)
95 print(image2.shape)
96 visualize.draw_boxes(image2, boxes=pre_nms_anchors, ax=get_ax())

```

代码第 89 行, 将标准坐标的锚点框转化成像素坐标, 然后在图像上显示出来。

代码第 91 行, 将原始图片变为统一大小。这样才可以与显示的锚点框对应。

代码运行后, 输出以下信息:

```
(640, 480, 3)
(1024, 1024, 3)
```

第 1 行是原始的图片形状, 第 2 行是转化后的图片形状。显示的图如图 8-41 所示。

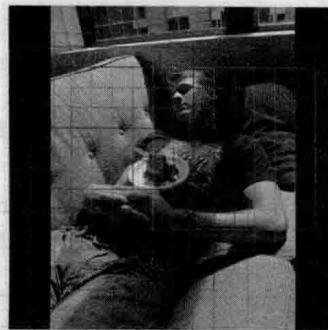


图 8-41 RPN 锚点的可视化结果

从图 8-41 可以看到，两边黑色的部分就是补 0 的部分。

## 2. 根据像素坐标可视化 RPN 返回的前景锚点

在 8.7.13 小节中的“提示”部分介绍过，锚点的像素坐标保存在 MaskRCNN 类的成员变量 anchors 中，所以也可以直接从 rpn\_class 元素中取出前景概率值最高的 50 个锚点的索引。根据索引从 MaskRCNN 类的成员变量 anchors 中取值，并在图中显示出来。代码如下：

### 代码 8-23 Mask\_RCNN 应用（续）

```
97 #从 rpn 的 rpn_class 元素中取出前景，并按由大到小排列
98 sorted_anchor_ids = np.argsort(rpn['rpn_class'][:, :, 1].flatten())[::-1]
99 visualize.draw_boxes(image2,
    boxes=model.anchors[sorted_anchor_ids[:limit]], ax=get_ax())
```

将代码第 98 行中的链式表达式展开，含义如下。

- (1) rpn['rpn\_class']: 代表从 rpn 中取出 rpn\_class 元素。其形状为(1, 261888, 2)。
- (2) rpn\_class 元素的最后一维为 softmax 后的背景和前景。0 代表背景，1 代表前景。
- (3) rpn['rpn\_class'][:, :, 1]: 将 rpn\_class 元素中的前景取出。
- (4) np.argsort(rpn['rpn\_class'][:, :, 1].flatten()): 对前景按照从小到大排序。
- (5) np.argsort(rpn['rpn\_class'][:, :, 1].flatten())[::-1]: 将“从小到大”排序后的前景进行倒序转化，变为“从大到小”排序。
- (6) 将最终的结果赋值给 sorted\_anchor\_ids 变量。

代码第 99 行中，sorted\_anchor\_ids[:limit] 的含义是：从列表 sorted\_anchor\_ids 中取出前 50 条记录（变量 limit 的值为 50）。

代码运行后生成的结果与图 8-41 一样。这也验证了像素坐标与标准化坐标的转化正确。

## 3. 可视化坐标调整前后的效果

将 RPN 中的 pre\_nms\_anchors 数据、refined\_anchors 数据与 refined\_anchors\_clipped 数据在图像上显示出来。代码如下：

### 代码 8-23 Mask\_RCNN 应用（续）

```
100 ax = get_ax(1, 2)
101 pre_nms_anchors = rpn['pre_nms_anchors'][0, :limit] * np.array([h, w, h, w])
```

```

102 refined_anchors = rpn['refined_anchors'][0, :limit] * np.array([h, w, h, w])
103 refined_anchors_clipped = rpn['refined_anchors_clipped'][0, :limit] *
    np.array([h, w, h, w])
104 #将 nms 之前的数据、边框调整后的数据和边框剪辑后的数据显示出来
105 visualize.draw_boxes(image2,
    boxes=pre_nms_anchors, refined_boxes=refined_anchors, ax=ax[0])
106 visualize.draw_boxes(image2, refined_boxes=refined_anchors_clipped,
    ax=ax[1])#边框剪辑后的数据

```

代码第 105 行，在用 `visualize.draw_boxes` 方法显示图片时传入了两个参数——`boxes` 与 `refined_boxes`。前者用于虚线显示，后者用于实线显示。

代码运行后，生成的图片如图 8-42 所示。

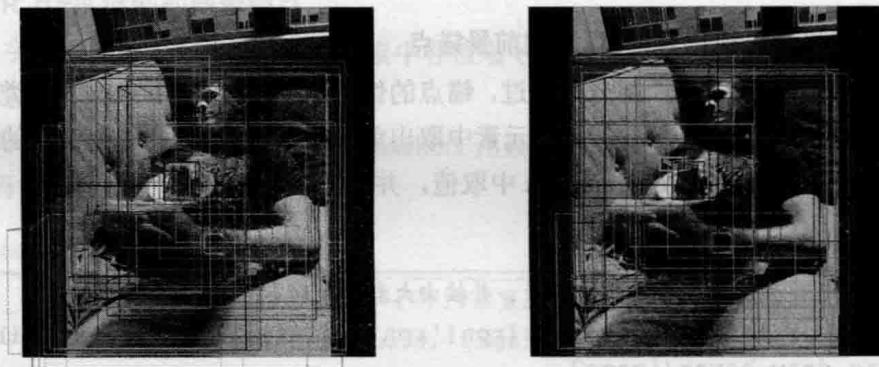


图 8-42 RPN 锚点边框调整后的可视化图片

图 8-42 左侧图有两种边框：虚线框与实线框。虚线框是模型中 ROI（靠谱区域）对应的锚点框，实线框是每个锚点经过偏移计算后的修正边框。二者的左上角通过直线连接起来。

图 8-42 右侧的图表示对出界部分的边框进行了剪辑。

#### 4. 可视化 NMS 之后的结果

经过 NMS 去重之后，锚点个数会变成 1000 个。将前景概率值最高的 50 个锚点取出，在图中显示出来。代码如下：

#### 代码 8-23 Mask\_RCNN 应用（续）

```

107 post_nms_anchor_ix = rpn['post_nms_anchor_ix'][ :limit]
108 refined_anchors_clipped = rpn["refined_anchors_clipped"][0,
    post_nms_anchor_ix] * np.array([h, w, h, w])
109 visualize.draw_boxes(image2, refined_boxes=refined_anchors_clipped,
    ax=get_ax())
110
111
112 #将 rpn 对象中的数据转化成原始图像尺寸，用于显示
113 proposals = rpn['proposals'][0, :limit] * np.array([h, w, h, w])
114 visualize.draw_boxes(image2, refined_boxes=proposals, ax=get_ax())

```

代码运行后，生成的图片如图 8-43 所示。

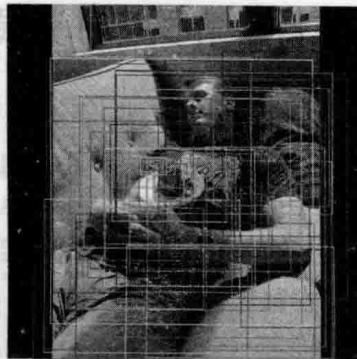


图 8-43 RPN 结果经过 NMS 算法处理后的锚点边框

对比图 8-42 右侧的图片，图 8-43 上的边框稀疏了一些，这说明去重算法还是有效果的。

### 8.7.15 代码实现：在 MaskRCNN 类中对 ROI 区域进行分类

经过 NMS 算法后的 RPN 结果被叫作 ROI（靠谱区域）。

在 MaskRCNN 类中，将 ROI（靠谱区域）与 8.7.9 小节的金字塔网络结果 mrcnn\_feature\_map 张量一起输入分类器网络中进行分类处理。

具体代码如下：

代码 8-24 mask\_rcnn\_model（续）

```

273      #下面式子中的数字，从左到右的意义依次是：1 代表 image_id, 3 代表
274      #original_image_shape, 3 代表 image_shape, 3 代表坐标, 1 代表缩放
275      img_meta_size = 1 + 3 + 3 + 4 + 1 + self.num_class #定义图片附加信息
276
277      input_image_meta = KL.Input(shape=[img_meta_size],
278                                     name="input_image_meta")#定义图片附加信息
279
280      #FPN 对 rpn_rois 区域与特征数据 mrcnn_feature_maps 进行计算，识别出分类、边
281      #框和掩码
282
283      if mode == "inference":
284          #定义网络的头部
285          #对 rpn_rois 区域内的 mrcnn_feature_maps 做分类，并微调 box_Proposal
286          classifier and BBox regressor heads
287          #mrcnn_class 是分类结果，mrcnn_bbox 是中心点长宽变化量
288          mrcnn_class_logits, mrcnn_class, mrcnn_bbox =
289              fpn_classifier_graph(rpn_rois, mrcnn_feature_maps,
290              input_image_meta,
291                               POOL_SIZE, self.num_class ,
292                               train_bn=False,#不用 BN 算法
293                               fc_layers_size=1024)#全连接层 1024 个节点

```

在代码第 284 行可以看到，分类器是用 fpn\_classifier\_graph 网络来实现的。下面 8.7.16 小节就来介绍具体内容。

## 8.7.16 代码实现：金字塔网络的区域对齐层（ROIAlign）中的区域框与特征的匹配算法

在分类器 fpn\_classifier\_graph 网络的第 1 层，用 ROIAlign 层进行特征抽取。具体步骤如下：

- (1) 把 mrcnn\_feature\_maps 列表中每个尺度的特征都当作一副图片。
- (2) 用 rpn\_rois 中的矩形框坐标在图片上找到对应区域，并将该区域的内容取出。
- (3) 统一变化到  $7 \times 7$  大小的特征数据（feature map）。

因为 rpn\_rois 区域中的位置框大小各有不同，mrcnn\_feature\_maps 列表中的内容也是各种尺度。如何在 mrcnn\_feature\_maps 列表中选取特征元素？需要按照 rpn\_rois 区域中的哪个框来提取内容？这便是在 ROIAlign 层中需要解决的问题。

### 1. ROIAlign 层中匹配算法的实现方法

ROIAlign 层中的匹配算法来自于一篇 FPN 论文，链接如下：

<https://arxiv.org/abs/1612.03144>

该算法的核心思想是：先用一个算法将 rpn\_rois 区域中的每个框与 mrcnn\_feature\_maps 列表中的具体特征对应起来，然后用 rpn\_rois 区域中的每个框从与其自身对应的特征中提取内容。

因为特征列表 mrcnn\_feature\_maps 中的特征是 P2~P5，所以该算法也将 rpn\_rois 区域中的所有框按照 2~5 来划分等级。

由于特征列表 mrcnn\_feature\_maps 中的每个特征尺度都不同，rpn\_rois 区域中的每个框的面积也不同，所以在该论文中，设计了一个根据 rpn\_rois 区域中的单个区域框的尺寸来划分 2~5 等级的算法，见式 (8.5)。

$$k = k_0 + \log_2(\sqrt{wh}/224) \quad (8.5)$$

在式 (8.5) 中， $k$  代表返回的级别， $k_0$  代表一个基准的级别值（在本实例中，值为 4）， $w$  与  $h$  分别代表区域框的宽和高。

这里的  $k_0=4$  与 224 代表了一个基准。因为在模型中使用的骨干网是 ResNet 模型，该模型在 ImgNet 数据集上训练时，输入的尺寸是 224，输出的特征尺寸与 P4 一致。所以，如果在 rpn\_rois 对象中某个框的大小为 224，则其所对应的特征必定是 P4。

至于其他尺寸的特征，可以根据  $\log_2(\sqrt{wh}/224)$  算出其与 224 的差别，再将需要调整的差别作用在基准的级别值  $k_0$  上，得到对应的级别。在式 (8.5) 中，使用  $\log_2$  只是进行了数值转换而已，这样会保证当边框发生较小的变化时，变差会有较大的值；而当边框发生过大的变化时，变差不会产生过大的值（ $\log$  的特性）。

### 2. 实现 ROIAlign 层中的匹配算法

用 tf.keras 接口定义一个 PyramidROIAlign 类，完成 ROIAlign 层的工作。在 PyramidROIAlign 类的 call 方法中，实现了 ROIAlign 层的匹配算法和区域提取功能。

其中，匹配算法的实现部分见以下代码：

## 代码 8-27 othernet(续)

```

139 #PyramidROIAlign 处理
140 class PyramidROIAlign(tf.keras.layers.Layer):
141
142     def __init__(self, batch_size, pool_shape, **kwargs):
143         super(PyramidROIAlign, self).__init__(**kwargs)
144         self.pool_shape = tuple(pool_shape)
145         self.batch_size = batch_size
146
147     def log2_graph(self, x):          #计算 log2
148         return tf.log(x) / tf.log(2.0)
149
150     def call(self, inputs):
151         """
152             输入参数 Inputs:
153                 - ROIboxes(RPN 结果): 该参数的形状为 [batch, num_boxes, 4], 其中, 最后一个维
154                 度 4 的内容为: (y1, x1, y2, x2)。
155                 - image_meta: [batch, (meta data)] 图片的附加信息 93
156                 - Feature maps: [P2, P3, P4, P5] 骨干网经过 FPN 后的特征数据, 形状依次为:
157                 [(1, 256, 256, 256), (1, 128, 128, 256), (1, 64, 64, 256), (1, 32, 32,
158                 256)]
159
160         #获取输入参数
161         ROIboxes = inputs[0]           # (1, 1000, 4)
162         image_meta = inputs[1]          # (1, 93)
163         feature_maps = inputs[2:]      # 将锚点坐标提出来
164         y1, x1, y2, x2 = tf.split(ROIboxes, 4, axis=2) #[batch, num_boxes, 4]
165         h = y2 - y1
166         w = x2 - x1
167         print("ROIboxes", ROIboxes.get_shape())
168         print("image_meta", image_meta.get_shape())
169         print("h", h.get_shape())       #(1, 1000, 1)
170         print("w", w.get_shape())
171
172         #在这 1000 个 ROI 里, 按固定算法匹配到不同 level 的特征
173         #获得图片形状
174         image_shape = parse_image_meta_graph(image_meta)['image_shape'][0]
175         print("image_shape", image_shape.get_shape())
176         image_area = tf.cast(image_shape[0] * image_shape[1], tf.float32)
177         #因为 h 与 w 是标准化坐标。其分母已经除以了 tf.sqrt(image_area)
178         #这里再除以 tf.sqrt(image_area) 分之 1, 是为了将 h 与 w 变为像素坐标
179         roi_level = self.log2_graph(tf.sqrt(h * w) / (224.0 /
180             tf.sqrt(image_area)))
181         roi_level = tf.minimum(5, tf.maximum(2, 4 +
182             tf.cast(tf.round(roi_level), tf.int32)))

```

```

181     roi_level = tf.squeeze(roi_level, 2)
182     print("roi", roi_level.get_shape()) # (1, 1000)

```

在代码第 179 行中, 计算 `roi_level` 时会发现分母比式(8.5)中多了一个“`/tf.sqrt(image_area)`”。原因是, 代码里的 `w` 与 `h` 是归一化后的值, 即“`像素值/tf.sqrt(image_area)`”后的结果。而公式里的 `w` 和 `h` 是像素值, 所以在分母上加了一个“`/tf.sqrt(image_area)`”将坐标统一成像素值。

代码第 180 行中, 对映射结果 `roi_level` 做了二次处理, 保证其变化后的值在 2~5 之间。

### 8.7.17 代码实现: 在金字塔网络的 ROIAlign 层中按区域边框提取内容

有了 `rpn_rois` 区域中的位置框与 `mrcnn_feature_maps` 列表中不同尺度特征的对应关系之后, 便可以按照 `rpn_rois` 区域中的 ROI 边框信息从特征数据中提取内容了。

可以将“按照 `rpn_rois` 区域框从特征数据中提取内容”过程理解为: 从图片中按照指定的区域框来提取内容。

- 图片: `mrcnn_feature_maps` 列表中的不同尺度的特征数据。从 P2 (第 2 特征层数据) 到 P5 (第 5 特征层数据) 所对应的尺寸依次为 [128,128]、[64,64]、[32,32]、[16,16]。
- 剪辑区域框: `rpn_rois` 区域中的每个 ROI 边框信息。
- 剪辑区域框与图的对应关系: 通过 `PyramidROIAlign` 类的算法规则进行匹配, 实现剪辑区域框与图片的一一对应。

#### 1. 了解提取内容关节中的边界值不匹配问题

在“从图片中按照剪辑区域框提取内容”环节中, 存在一个边界值不匹配的问题。这是由于图片中像素点的值是用整数表示的, 而 `rpn_rois` 区域中的 ROI 边框坐标是用浮点型的小数表示的, 二者存在数值不匹配的问题。例如, 无法从一个尺寸为 [16,16] 的图片上精确地提取出尺寸为 [10.5,10.5] 这样的区域内容。

#### 2. 边界值不匹配问题的解决方法

在 Mask-RCNN 模型的论文 (<https://arxiv.org/abs/1703.06870>) 中, 通过将浮点数坐标转化为对应像素点上的图像数值, 来解决边界值不匹配问题。

在具体实现中, 用双线性内插的算法来完成浮点数坐标到图像数值的转化。在本实例中, 直接使用 `tf.image.crop_and_resize` 函数即可实现从转化到提取的整套功能。

函数 `tf.image.crop_and_resize` 的作用是: 按照指定的区域去图片上进行截取, 并将截取的结果转化为指定的形状。该函数支持双线性内插和邻近值内插两种算法。在使用时, 可以通过参数进行控制。

#### 3. 按照区域边框提取内容的完整实现

在实际编码中, 除考虑提取内容的操作外, 还需要考虑在内容提取后的顺序问题。要保证提取后的内容顺序与 `rpn_rois` 区域中的边框顺序对应起来。整个步骤如下:

(1) 按照特征层数据的索引, 去 `rpn_rois` 区域框中找到对应的 ROI 边框, 得到 `level_boxes` 对象。

(2) 统一用 `tf.image.crop_and_resize` 函数在特征图中按照 `level_boxes` 对象中的各个剪辑框截取内容。

(3) 将所有截取后的结果合并起来，生成 `pooled` 对象（此时 `pooled` 对象的顺序是按照其所对应特征尺度的顺序来的）

(4) 按照原有 `rpn_rois` 顺序将 `pooled` 对象重新排列起来。

以上步骤形成了 `PyramidROIAlign` 类的后半部分。具体代码如下：

### 代码 8-27 othernet (续)

```

183 #每个 ROI 按照自己的区域去对应的特征里截取内容，并将尺寸改成 7×7 大小的特征数据
184     pooled = []
185     box_to_level = []
186     for i, level in enumerate(range(2, 6)):
187
188         #tf.equal 会返回一个 true false 的 (1, 1000)
189         #tf.where 返回其中为 true 的索引 [[0,1],[0,4] …[0,200]…]
190         ix = tf.where(tf.equal(roi_level, level), name="ix") #所得形状为 (828,
2)
191         print("ix", level, ix.get_shape(), ix.name)
192
193         #在多维上建立索引取值 [?, 4] (828, 4)
194         level_boxes = tf.gather_nd(ROIboxes, ix, name="level_boxes")
195
196         #形状为 (828, )
197         box_indices = tf.cast(ix[:, 0], tf.int32)
198         print("box_indices", box_indices.get_shape(), box_indices.name)
199         #跟踪索引值
200         box_to_level.append(ix)
201
202         #不希望下面两个值有变化，所以停止梯度
203         level_boxes = tf.stop_gradient(level_boxes) #ROIboxes 中按照不同尺
度划分好的索引
204         box_indices = tf.stop_gradient(box_indices)
205
206         #结果: [batch * num_boxes, pool_height, pool_width, channels]
207         #feature_maps [(1, 256, 256, 256), (1, 128, 128, 256), (1, 64, 64,
256), (1, 32, 32, 256)]
208         #box_indices 一共有 level_boxes 个。指定 level_boxes 中的第几个框作用于
feature_maps 中的第几个图片
209         pooled.append(tf.image.crop_and_resize(feature_maps[i],
level_boxes, box_indices, self.pool_shape, method="bilinear"))
210
211         #1000 个 roi 都取到了对应的内容，将它们组合起来。组合后的形状为 (1000, 7, 7, 256)
212         pooled = tf.concat(pooled, axis=0) #其中的顺序是按照 level 来的，需要重新
排列成原来 ROIboxes 顺序

```

```

214     #按照选取 level 的顺序重新排列成原来 ROIboxes 顺序
215     box_to_level = tf.concat(box_to_level, axis=0)
216     box_range = tf.expand_dims(tf.range(tf.shape(box_to_level)[0]), 1)
217     box_to_level = tf.concat([tf.cast(box_to_level, tf.int32),
218                               box_range], axis=1)           # [1000, 3] 3([xi] range)
219
220     #取出头两个“批次+序号”(1000个), 每个值代表原始 ROI 展开的索引
221     sorting_tensor = box_to_level[:, 0] * 100000 + box_to_level[:, 1] #
222     #保证一个批次在 100000 以内
223     #按照索引排序
224     ix = tf.nn.top_k(sorting_tensor,
225                        k=tf.shape(box_to_level)[0]).indices[::-1]
226     ix = tf.gather(box_to_level[:, 2], ix) #按照 ROI 中的顺序取出 pooled 中的
227     #加上批次维度, 并返回
228     pooled = tf.gather(pooled, ix) #将 pooled 按照原始顺序排列
229
230     return pooled
231
232     def compute_output_shape(self, input_shape):
233         return input_shape[0][:2] + self.pool_shape + (input_shape[2][-1], )

```

代码第 191 行和第 198 行将张量的名称 (name) 打印出来, 是为了调试时使用。运行后会看到该张量的名称。根据该名称编写代码将里面的值运行出来, 观察结果是否与预期的一致。

代码第 200 行将每次循环的索引值保存起来, 是为后面将结果重新排列做准备(见代码第 223 行)。

代码第 203 行, level\_boxes 对象是 ROIboxes 对象中按照不同尺度划分好的索引。代码第 204 行, box\_indices 对象是批次索引, 该批次索引与当前的尺度特征对应。这两个值是依赖 rpn\_rois 区域框的数值并根据常规算法得到的。在反向传递中, 不希望其值发生变化, 所以停止梯度。

代码第 210 行, 调用了 tf.image.crop\_and\_resize 函数的前 4 个重要参数。具体说明如下:

- feature\_maps[i]: 输入待裁剪的图。
- level\_boxes: 存放剪辑框的数组。按照该数组中的剪辑框尺寸去图中裁剪。
- box\_indices: 一个与 level\_boxes 对象长度一样的数组, 内容为特征数据 feature\_maps[i] 的索引值, 用于选取图片。该数组让 level\_boxes 对象中的每个 ROI 区域框去指定的图片上截取内容。
- self.pool\_shape: 将截取后的内容统一调整尺寸到指定尺寸。

### 8.7.18 代码实现：调试并输出 ROIAlign 层的内部运算值

用 utils.run\_graph 函数将相关节点打印出来。代码如下：

代码 8-23 Mask\_RCNN 应用（续）

```

115 roi_align_classifierlar = model.keras_model.get_layer
    ("roi_align_classifier").output #获得 ROI 节点，即 ProposalLayer 层
116
117 roi_align_classifier = utils.run_graph(model,[image], [
118     ("roi_align_classifierlar", model.keras_model.get_layer
        ("roi_align_classifier").output), #(1, 261888, 2)
119     ("ix", model.ancestor(roi_align_classifierlar,
        "roi_align_classifier/ix:0")),
120     ("level_boxes", model.ancestor(roi_align_classifierlar,
        "roi_align_classifier/level_boxes:0")),
121     ("box_indices", model.ancestor(roi_align_classifierlar,
        "roi_align_classifier/Cast_2:0")),
122
123 ],BATCH_SIZE)
124
125 print(roi_align_classifier ["ix"][:5])    #(828, 2)
126 print(roi_align_classifier ["level_boxes"][:5]) #(828, 4)
127 print(roi_align_classifier ["box_indices"][:5]) #(828, 4)

```

运行代码后，对输出的结果进行整理、解读。具体如下：

(1) 各张量的形状和数值信息。

```

roi_align_classifierlar shape: (1, 1000, 7, 7, 256) min: -39.27100 max:
44.52850 float32
ix shape: (421, 2) min: 0.00000 max: 999.00000 int64
level_boxes shape: (421, 4) min: 0.00000 max: 1.00000 float32
box_indices shape: (421,) min: 0.00000 max: 0.00000 int32

```

PyramidROIAlign 层的返回形状为(1, 1000, 7, 7, 256)。之所以将维度变成了 5，是为了在后面可以基于单个图片独立做卷积变化（见 8.7.19 小节）。

在 PyramidROIAlign 层中，每个尺度特征框索引的形状为(批次,2)。

level\_boxes 对象是从 rpn\_rois 区域框中根据索引 ix 拿出的区域框，形状为(批次,4)。box\_indices 数组的意义是，为 level\_boxes 对象中的每个框指定所要提取内容的图片索引，形状为(批次,)。注意，逗号之后是没有数字的。

(2) 索引 ix 的前 5 个元素的具体值（其中 0 代表批次的索引）：

```

[[ 0 22]
 [ 0 29]
 [ 0 48]
 [ 0 59]
 [ 0 64]]

```

214 (3) `level_boxes` 对象的前 5 个元素是被归一化处理的两个点坐标。

```
[ [0.7207245 0.7711525 0.7770287 0.8291931 ]
[0.70217687 0.51715106 0.7498454 0.60750276]
[0.7385003 0.6546944 0.79295075 0.7119865 ]
[0.75078577 0.70076877 0.81955194 0.77168196]
[0.8174721 0.6675705 0.8727931 0.7368731 ]]
```



### 提示：

一共 5 行，每行代表一个元素。每个元素有 4 个值，前 2 个值代表第 1 个点的  $x$ 、 $y$  值。后 2 个值代表第 2 个点的  $x$ 、 $y$  值。

220 (4) `box_indices` 数组的前 5 个元素的值全是 0。因为每个尺度特征的形状为(batch,N,N,256)，可以理解成深度为 256 通道的图片。本实例中 batch=1，所以 `rpn_rois` 区域框中所有的框都得去第 0 张图片上截取。

```
[0 0 0 0 0]
```

## 8.7.19 代码实现：对 ROI 内容进行分类

本小节将实现分类器，并通过代码验证其效果。

### 1. 实现分类器

完整的分类器是在 `fpn_classifier_graph` 函数中实现的。代码如下：

#### 代码 8-27 othernet (续)

```
233 def fpn_classifier_graph(rois, feature_maps, image_meta,
234                         pool_size, num_classes, batch_size, train_bn=True,
235                         fc_layers_size=1024):
236
237     #ROIAlign 层 Shape: [batch, num_boxes, pool_height, pool_width, channels]
238     x = PyramidROIAlign(batch_size,[pool_size, pool_size],
239                         name="roi_align_classifier")([rois, image_meta] +
feature_maps)
240
241     #用卷积替代两个 1024 全连接网络
242     x = KL.TimeDistributed(KL.Conv2D(fc_layers_size, (pool_size,
pool_size),
243                               padding="valid"),
name="mrcnn_class_conv1")(x)
244     x = KL.TimeDistributed(KL.BatchNormalization(),
name='mrcnn_class_bn1')(x,
245                           training=train_bn)
246     x = KL.Activation('relu')(x)
247     #1x1 卷积，代替第 2 个全连接
```

```

248     x = KL.TimeDistributed(KL.Conv2D(fc_layers_size, (1, 1)),
249         name="mrcnn_class_conv2") (x)
250     x = KL.TimeDistributed(KL.BatchNormalization(),
251         name='mrcnn_class_bn2') (x, training=train_bn)
252     x = KL.Activation('relu') (x)
253     # 共享特征，用于计算分类和边框
254     shared = KL.Lambda(lambda x: K.squeeze(K.squeeze(x, 3),
255         2), name="pool_squeeze") (x)
256     # (1) 计算分类
257     mrcnn_class_logits = KL.TimeDistributed(KL.Dense(num_classes),
258         name='mrcnn_class_logits') (shared)
259     mrcnn_probs = KL.TimeDistributed(KL.Activation("softmax"),
260         name="mrcnn_class") (mrcnn_class_logits)
261     # (2) 计算边框坐标 BBox (偏移和缩放量)
262     #[batch, boxes, num_classes * (dy, dx, log(dh), log(dw))]
263     x = KL.TimeDistributed(KL.Dense(num_classes * 4, activation='linear'),
264         name='mrcnn_bbox_fc') (shared)
265     # 将形状变成 [batch, boxes, num_classes, (dy, dx, log(dh), log(dw))]
266     mrcnn_bbox = KL.Reshape((-1, num_classes, 4), name="mrcnn_bbox") (x)
267
268     return mrcnn_class_logits, mrcnn_probs, mrcnn_bbox

```

代码第242、248行分别用两个卷积网络代替全连接生成共享特征shared(见代码第253行)。之后将共享特征用于分类(256)和边框(263)的计算。



### 提示：

本节代码中多次用到KL.TimeDistributed函数(例如代码第245、258行等)。该函数的作用是将输入特征按照时间维度应用到相同的层。其处理的数据第1维度是1,表示将整个数据当作一个样本。而批次和每一批次的数据个数被统一放在了第2维度。它与ProposalLayer类中的utils.batch\_slice函数(见8.7.12小节)并不一样。utils.batch\_slice是从不同的张量中收集与指定索引相对应的公共元素。

例如：在fpn\_classifier\_graph函数中，PyramidROIAlign层的输出形状为(batch,N,高度,宽度,通道)。这是一个5D张量。而tf.keras的卷积函数Conv2D仅接受4D张量。这时可以把batch看作tf.TimeDistributed中的图层，把第2维(N)当作Conv2D操作的批次。经过第一次ReLU操作后，输出形状为(batch, N, 1, 1, 1024)。

ProposalLayer类的输入是rpn\_class(batch,num\_anchors\_total,2)和rpn\_bbox(batch,num\_anchors\_total,4)。第1次调用utils.batch\_slice函数的输入是分值scores(batch,num\_anchors\_total)和索引ix(batch,pre\_nms\_limit)，目的是收集ix指定的所有批次的顶级pre\_nms\_limit锚点，并用tf.gather完成此操作。函数tf.gather是在第1个维度(batch)

上运行的。因此在函数 `utils.batch_slice` 中，通过一个 `for` 循环一次处理一个批次，在锚点总数 `num_anchors_total` 对象和 `pre_nms_limit` 维度之间进行函数 `tf.gather` 处理。

## 2. 可视化分类器结果

用 `utils.run_graph` 函数将分类器输出的分类和边框结果运行出来，并将得到的值可视化。代码如下：

### 代码 8-23 Mask\_RCNN 应用（续）

```

128 fpn_classifier = utils.run_graph(model, [image], [
129     ("probs", model.keras_model.get_layer("mrcnn_class").output), #shape:
130     (1, 1000, 81)
130     ("deltas", model.keras_model.get_layer("mrcnn_bbox").output), #(1, 1000,
131     81, 4)
131 ], BATCH_SIZE)
132 #因为 proposals 结果是相对于原始图片变形的框，所以要使用相对于原始图片变形后的图片
133 image2
133proposals=utils.denorm_boxes(rpn["proposals"][:], image2.shape[:2]) #(1000,
134 4)
134
135 #计算 81 类中的最大索引——class id(索引就是分类)
136 roi_class_ids = np.argmax(fpn_classifier["probs"][:], axis=1) #(1000,)
137 print(roi_class_ids.shape, roi_class_ids[:20])
138 roi_class_names = np.array(class_name)[roi_class_ids] #根据索引把名字取出来
139 print(roi_class_names[:20])
140 #去重类别个数
141 print(list(zip(*np.unique(roi_class_names, return_counts=True))))
142
143 roi_positive_ixs = np.where(roi_class_ids > 0)[:][0] #不是背景的类索引
144 print("{}中有{}个前景实例\n{}".format(len(proposals),
145
145 len(roi_positive_ixs), roi_positive_ixs))
146 #根据索引将最大的那个值取出来，当作分数
147 roi_scores = np.max(fpn_classifier["probs"][:], axis=1)
148 print(roi_scores.shape, roi_scores[:20])
149
150 #边框可视化
151 #通过两张图来完成：从第 1 张图中取出 50 个包含前景和背景的框，并显示出来；从第 2 张图中取出 5 个坐标调整后的前景框，并显示出来
152 limit = 50
153 ax = get_ax(1, 2)
154
155 ixs = np.random.randint(0, proposals.shape[0], limit)
156 captions = ["{} {:.3f}{}".format(class_name[c], s) if c > 0 else """
157         for c, s in zip(roi_class_ids[ixs], roi_scores[ixs])]
158
159 visib= np.where(roi_class_ids[ixs] > 0, 2, 1) #前景统一设为 2，背景统一设为 1

```

```

160
161 visualize.draw_boxes(image2, boxes=proposals[ixs], #原始的框放进去
162     visibilities=visib,#若为2，则突出显示；若为1，则一般显示
163     captions=captions, title="before fpn_classifier",
164     ax=ax[0])
164
165 #把指定类索引的坐标提取出来
166 #取出每个框对应分类的坐标偏差。fpn_classifier["deltas"] 的形状为(1,1000,81,4)
167 roi_bbox_specific = fpn_classifier["deltas"][0,
168     np.arange(proposals.shape[0]), roi_class_ids]
168 print("roi_bbox_specific", roi_bbox_specific) #形状为(1000,4)
169 name="rcnn_detections"
170 #根据偏移来调整 ROI, Shape: [N, (y1, x1, y2, x2)]
171 refined_proposals = utils.apply_box_deltas(
172     proposals, roi_bbox_specific *
173         mask_rcnn_model.BBOX_STD_DEV).astype(np.int32)
173 print("refined_proposals", refined_proposals)
174
175 limit =5
176 ids = np.random.randint(0, len(roi_positive_ixs), limit)#取出 5 个前景类
177
178 captions = ["{} {:.3f}{}".format(class_name[c], s) if c > 0 else ""
179             for c, s in zip(roi_class_ids[roi_positive_ixs][ids],
180                 roi_scores[roi_positive_ixs][ids])]
180
181 visualize.draw_boxes(image2, boxes=proposals[roi_positive_ixs][ids],
182     refined_boxes=refined_proposals[roi_positive_ixs][ids],
183     captions=captions, title="ROIs After
Refinement", ax=ax[1])

```

代码运行后，输出以下结果：

(1) ROI 分类结果的个数(1000个)，以及前20个ROI的分类结果。

```
(1000,) [33 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1]
```

(2) 对应的类名。

```
['sports ball' 'person' 'person' 'person' 'person' 'person' 'person' 'person'
'person' 'person' 'person' 'BG' 'person' 'BG' 'person' 'BG' 'person' 'person']
[('BG', 905), ('baseball glove', 11), ('handbag', 2), ('person', 76), ('sports ball',
6)]
```

1000 中有 95 个前景实例

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 14 16 18 19 20
.....822 850 879 972 992]
```

(3) 每个类的得分情况。

```
(1000,) [0.9996729 0.99934226 0.9994691 0.9995741 0.99940085 0.9998511
0.98432136 0.99704546 0.7715847 0.91766 0.7421977 0.941382]
```

```
0.6399888 0.95837957 0.5357658 0.7453531 0.99681807 0.7035237 0.9983895
0.6693075 ]
```

#### (4) 计算出的坐标。

```
roi_bbox_specific [[-0.08322562 -0.08801503 0.06453485 -0.07142121]
[-0.15577134 0.08577229 -0.08280858 0.2607019 ]
[ 0.18355349 0.30293933 0.12844186 -0.26919323]
...
[ 0.36996925 -0.24318382 0.2388043 0.13342915]
[ 0.67343915 -0.19701773 0.3982284 -0.13674134]
[ 0.45560795 -0.08497302 0.20064178 0.07753003]]
```

#### (5) 换算成像素点的坐标。

```
refined_proposals [[112 57 137 94]
[148 418 258 476]
[165 359 331 426]
...
[192 261 198 266]
[188 122 193 131]
[191 218 197 225]]
```

生成的图片如图 8-44 所示。



图 8-44 RPN 结果经过 NMS 算法处理后的锚点边框

在图 8-44 中，左图显示了前 50 个 ROI 区域框。其中虚线代表背景，实线代表前景的具体类别；右图显示了 5 个前景类中的 ROI 区域框。其中虚线表示 RPN 的位置框，实线表示调整后的位置框。每个虚线框和其调整后的实线框都通过左上角的直线相连。在右图中可以看到，最左边的那个人被画上了两个实线框。这表示：检测结果中出现了重复实例。所以，要对分类器 `fpn_classifier_graph` 处理后的结果再次去重，才可以得到最终的分类即边框。

### 8.7.20 代码实现：用检测器 DetectionLayer 检测 ROI 内容，得到最终的实物矩形

实物矩形检测的最后一个环节是通过 `DetectionLayer` 类来实现的。`DetectionLayer` 类的主要功能是对分类器 `fpn_classifier_graph` 输出结果的二次去重，该去重操作是根据分类的分数及边

框的位置来实现的。具体做法如下。

### 1. 在 MaskRCNN 类中调用检测器 DetectionLayer

在 MaskRCNN 类中添加代码，将使用 NMS 算法处理后的 RPN 结果（rpn\_rois）与分类器结果（mrcnn\_class 和 mrcnn\_bbox）组合起来，送入 DetectionLayer 类算出真实的 box 坐标。具体代码如下：

#### 代码 8-24 mask\_rcnn\_model (续)

```
288     #将 rpn_rois 与 mrcnn_class、mrcnn_bbox 组合起来，算出真实的 box 坐标
289     detections = DetectionLayer( batch_size= self.batch_size,
290         name="mrcnn_detection")(
291             [rpn_rois, mrcnn_class, mrcnn_bbox, input_image_meta])
```

### 2. 实现 DetectionLayer 类

下面用 tf.keras 接口定义检测器 DetectionLayer 类。DetectionLayer 类作为一个网络层用于输出 Mask R-CNN 模型最终的分类结果。在 DetectionLayer 类的 call 方法中实现了以下步骤：

(1) 取出图片的附加信息 m 字典（见代码 281 行）。

(2) 从 m 字典中取出 window 变量。Window 是经过 padding 处理后真实图片的像素坐标。该值是在对原始图片进行 resize（变换尺寸）操作时填入的。

(3) 将 window 变量所代表的图片像素坐标变成标准坐标（见代码 284 行）。

(4) 将分类信息统一放入 refine\_detections\_graph 函数中实现二次去重。

具体代码如下：

#### 代码 8-27 othernet (续)

```
269 # 实物边框检测，返回最终的标准化区域坐标[batch, num_detections, (y1, x1, y2, x2,
270   class_id, class_score)]
270 class DetectionLayer(tf.keras.layers.Layer):
271
272     def __init__(self, batch_size, **kwargs):
273         super(DetectionLayer, self).__init__(**kwargs)
274         self.batch_size = batch_size
275
276     def call(self, inputs):#输入：rpn_rois、mrcnn_class、mrcnn_bbox,
277         input_image_meta
278         #提取参数
279         rois,mrcnn_class,mrcnn_bbox,image_meta = inputs
280
281         #解析图片附加信息
282         m = parse_image_meta_graph(image_meta)
283         image_shape = m['image_shape'][0]
284         #window 是经过 padding 处理后真实图片的像素坐标，将其转化为标准坐标
285         window = norm_boxes_graph(m['window'], image_shape[:2])
286
287         #根据分类信息，对原始 ROI 进行再一次过滤，得到 DETECTION_MAX_INSTANCES 个 ROI。
```

```

287     detections_batch = utils.batch_slice(
288         [rois, mrcnn_class, mrcnn_bbox, window],
289         lambda x, y, w, z: refine_detections_graph(x, y, w, z),
290         self.batch_size)
291
292     #将标准化坐标及过滤后的结果变形后返回
293     return tf.reshape(
294         detections_batch,
295         [self.batch_size, mask_rcnn_model.DETECTION_MAX_INSTANCES, 6])
296
297     def compute_output_shape(self, input_shape):
298         return (None, mask_rcnn_model.DETECTION_MAX_INSTANCES, 6)
299
300 #将坐标按照图片大小转化为标准坐标
301 def norm_boxes_graph(boxes,                                     #像素坐标(y1, x1, y2, x2)
302                      shape):                                #像素边长(height, width)
303     h, w = tf.split(tf.cast(shape, tf.float32), 2)
304     scale = tf.concat([h, w, h, w], axis=-1) - tf.constant(1.0)
305     shift = tf.constant([0., 0., 1., 1.])
306     return tf.divide(boxes - shift, scale)  #标准化坐标[..., (y1, x1, y2,
x2)]

```

模型输出的 box 坐标是根据 resize (变换尺寸) 后的图片尺寸进行计算的。在返回最终结果之前，还需将该 box 坐标映射到真实图片的尺寸上去。

代码第 284 行得到标准坐标，用于还原边框在原始图片上的坐标。

代码第 301 行定义了函数 norm\_boxes\_graph，该函数将坐标按照图片大小转化为标准坐标。

### 3. 定义函数 refine\_detections\_graph，实现对结果去重

定义函数 refine\_detections\_graph，实现对结果去重，并对边框坐标进行简单的处理（根据偏移量 delta 修正出最终坐标，并进行合规剪辑）。具体代码如下：

#### 代码 8-27 othernet (续)

```

307 #定义分类器结果的最终处理函数，返回剪辑后的标准坐标与去重后的分类结果
308 def refine_detections_graph(rois, probs, deltas, window):
309
310     #从分类结果 probs 中取出分类分值最大的索引，probs 的形状是[1000, 81]
311     class_ids = tf.argmax(probs, axis=1, output_type=tf.int32)
312
313     #根据分类索引构造分类结果 probs 的切片索引，该切片索引用于以切片的方式从张量中取值
314     indices = tf.stack([tf.range(tf.shape(probs)[0]), class_ids], axis=1)
315     class_scores = tf.gather_nd(probs, indices) #根据索引获得分数
316
317     deltas_specific = tf.gather_nd(deltas, indices) #根据索引获得 box 区域坐标(待
修正的偏差)
318
319     #将偏差应用到 rois 框中

```

```

320     refined_rois = apply_box_deltas_graph(rois, deltas_specific *           E8L
    mask_rcnn_model.BBOX_STD_DEV)           E8L
321     #对出界的框进行剪辑           E8L
322     refined_rois = clip_boxes_graph(refined_rois, window)           E8L
323
324     #取出前景的类索引（将背景类过滤掉）           E8L
325     keep = tf.where(class_ids > 0)[:, 0]           E8L
326     #在前景类里，将小于DETECTION_MIN_CONFIDENCE的分数过滤掉           E8L
327     if mask_rcnn_model.DETECTION_MIN_CONFIDENCE:           E8L
328         conf_keep = tf.where(class_scores >=           E8L
    mask_rcnn_model.DETECTION_MIN_CONFIDENCE)[:, 0]           E8L
329         keep = tf.sets.set_intersection(tf.expand_dims(keep, 0),           E8L
330             tf.expand_dims(conf_keep, 0))           E8L
331         keep = tf.sparse_tensor_to_dense(keep)[0]           E8L
332
333     #根据剩下的keep索引取出对应的值           E8L
334     pre_nms_class_ids = tf.gather(class_ids, keep)           E8L
335     pre_nms_scores = tf.gather(class_scores, keep)           E8L
336     pre_nms_rois = tf.gather(refined_rois, keep)           E8L
337     unique_pre_nms_class_ids = tf.unique(pre_nms_class_ids)[0]           E8L
338
339     def nms_keep_map(class_id):#定义NMS算法处理函数，对每个类做去重           E8L
340
341         #找出类别为class_id的索引           E8L
342         ixs = tf.where(tf.equal(pre_nms_class_ids, class_id))[:, 0]           E8L
343
344         #对该类的roi按照阈值DETECTION_NMS_THRESHOLD进行区域去重，最多获得           E8L
    DETECTION_MAX_INSTANCES个结果           E8L
345         class_keep = tf.image.non_max_suppression(           E8L
346             tf.gather(pre_nms_rois, ixs),           E8L
347             tf.gather(pre_nms_scores, ixs),           E8L
348             max_output_size=mask_rcnn_model.DETECTION_MAX_INSTANCES,           E8L
349             iou_threshold=mask_rcnn_model.DETECTION_NMS_THRESHOLD)           E8L
350         #将去重后的索引转化为ROI中的索引           E8L
351         class_keep = tf.gather(keep, tf.gather(ixs, class_keep))           E8L
352         #数据对齐，当去重后的个数小于DETECTION_MAX_INSTANCES时，对其补-1           E8L
353         gap = mask_rcnn_model.DETECTION_MAX_INSTANCES -           E8L
    tf.shape(class_keep)[0]           E8L
354         class_keep = tf.pad(class_keep, [(0, gap)],           E8L
355             mode='CONSTANT', constant_values=-1)           E8L
356         #将形状统一变为[mask_rcnn_model.DETECTION_MAX_INSTANCES]，并返回           E8L
357         class_keep.set_shape([mask_rcnn_model.DETECTION_MAX_INSTANCES])           E8L
358         return class_keep           E8L
359
360     #对每个class IDs做去重操作           E8L
361     nms_keep = tf.map_fn(nms_keep_map, unique_pre_nms_class_ids,           E8L
362             dtype=tf.int64)           E8L

```

```

363     #将 list 结果中的元素合并到一个数组里，并删掉-1的值
364     nms_keep = tf.reshape(nms_keep, [-1])
365     nms_keep = tf.gather(nms_keep, tf.where(nms_keep > -1)[:, 0])
366     keep = nms_keep
367     #经过 NMS 处理后，根据剩下的 keep 索引取出对应的值，并将取值的个数控制在
368     #DETECTION_MAX_INSTANCES 之内
369     roi_count = mask_rcnn_model.DETECTION_MAX_INSTANCES
370     class_scores_keep = tf.gather(class_scores, keep)
371     num_keep = tf.minimum(tf.shape(class_scores_keep)[0], roi_count)
372     top_ids = tf.nn.top_k(class_scores_keep, k=num_keep, sorted=True)[1]
373     keep = tf.gather(keep, top_ids) #keep 个数小于 DETECTION_MAX_INSTANCES
374
375     #拼接输出结果，形状是[N, (y1, x1, y2, x2, class_id, score)]。其中，N 是结果
376     #的个数
377     detections = tf.concat([
378         tf.gather(refined_rois, keep),
379         tf.cast(tf.gather(class_ids, keep), tf.float32)[..., tf.newaxis],
380         tf.gather(class_scores, keep)[..., tf.newaxis]
381     ], axis=1)
382
383     #数据对齐，不足 DETECTION_MAX_INSTANCES 的补 0，并返回
384     gap = mask_rcnn_model.DETECTION_MAX_INSTANCES - tf.shape(detections)[0]
385     detections = tf.pad(detections, [(0, gap), (0, 0)], "CONSTANT")
386
387     return detections

```

代码第 315 行，用函数 `tf.gather_nd` 从分类结果 `probs` 中取值，得到分类分数。因为分类结果 `probs` 是张量，所以不能以 Python 切片的方式进行取值。

函数 `tf.gather_nd` 支持多维度取值，在调用函数 `tf.gather_nd` 时，将制作好的切片索引 `indices`（见代码第 314 行）传入即可实现 Python 中的切片效果。



### 提示：

代码第 311~315 行比较晦涩。为了方便理解，将该部分的逻辑用模拟数据实现出来。具体代码如下：

```

import tensorflow as tf
import numpy as np
tf.enable_eager_execution()          #启动动态图（在TensorFlow 2.x中可以去掉该句）
#假设 probs 中有 3 个锚点，每个锚点有 4 个分值
probs = np.array([[1,6,3,4], [2,2,3,4], [3,2,9,4]])
print(np.shape(probs))             #输出 probs 的形状：(3, 4)
class_ids = tf.argmax(probs, axis=1, output_type=tf.int32)
print(class_ids.numpy())           #输出 probs 中的最大索引：[1 3 2]
#构造切片索引
indices = tf.stack([tf.range(tf.shape(probs)[0]), class_ids], axis=1)

```

```

print(indices.numpy())          #输出切片索引：[[0 1] [1 3] [2 2]]
class_scores = tf.gather_nd(probs, indices) #根据切片索引获得分数
print(class_scores.numpy())      #输出所获得的分数：[6 4 9]
print(probs[tf.range(tf.shape(probs)[0]).numpy(), class_ids]) #用切片方式取数，输出：[6 4 9]
从输出的结果可以看出，函数 tf.gather_nd 的取值结果与 Python 语法中使用多维度切片
方式的取值结果相同。

```

在函数 refine\_detections\_graph 中，会对分类分数按照固定的阈值进行过滤（见代码第 328 行）。将剩下的部分再用 NMS 算法进行去重（见代码第 361 行）。最后取 DETECTION\_MAX\_INSTANCES 个 ROI(不足的补 0)，作为最终检测结果（见代码第 382 行）。

### 提示：

函数 refine\_detections\_graph 的输入、输出都是基于单个样本的，即该函数中的所有变量都没有批次维度。

## 4. 可视化检测器结果

用 utils.run\_graph 函数输出检测器结果，并通过坐标转化将其显示到原始图片上，并将得到的值可视化。代码如下：

### 代码 8-23 Mask\_RCNN 应用（续）

```

184 #定义函数按照窗口来调整坐标
185 def refineboxbywindow(window, coordinates):
186
187     wyl, wxl, wy2, wx2 = window
188     shift = np.array([wyl, wxl, wyl, wxl])
189     wh = wy2 - wyl # 计算 window height
190     ww = wx2 - wxl # 计算 window width
191     scale = np.array([wh, ww, wh, ww])
192     #按照窗口来调整坐标
193     refine_coordinates = np.divide(coordinates - shift, scale)
194     return refine_coordinates
195
196 #模型输出的最终检测结果
197 DetectionLayer = utils.run_graph(model, [image], [
198     #(1, 100, 6), 最后的 6 由 4 个位置、1 个分类、1 个分数组成
199     ("detections", model.keras_model.get_layer("mrcnn_detection").output),
200 ], BATCH_SIZE)
201
202 #获得分类的 ID
203 det_class_ids = DetectionLayer['detections'][0, :, 4].astype(np.int32)
204
205 det_ids = np.where(det_class_ids != 0)[0] #取出前景类不等于 0 的索引
206 det_class_ids = det_class_ids[det_ids] #预测的分类 ID

```

```

207 #将分类 ID 显示出来
208 print("{} detections: {}".format( len(det_ids),
209     np.array(class_name)[det_class_ids]))
210 roi_scores= DetectionLayer['detections'][0, :, -1] #获得分类分数
211
212 boxes_norm= DetectionLayer['detections'][0, :, :4] #获得边框坐标
213 window_norm = utils.norm_boxes(window, image2.shape[:2])
214 boxes = refineboxbywindow(window_norm,boxes_norm) #按照窗口缩放来调整坐标
215
216 #将坐标转化为像素坐标
217 refined_proposals=utils.denorm_boxes(boxes[det_ids],
218     image.shape[:2])#(1000, 4)
219 captions = ["{} {:.3f}{}".format(class_name[c], s) if c > 0 else ""
220         for c, s in zip(det_class_ids, roi_scores[det_ids])]
221
222 visualize.draw_boxes(                                #在原始图片上显示结果
223     image, boxes=refined_proposals[det_ids],
224     visibilities=[2] * len(det_ids),#统一设为 2, 表示用实线显示
225     captions=captions, title="Detections after NMS",
226     ax=get_ax())

```

代码运行后，输出以下结果：

```
5 detections: ['person' 'person' 'person' 'person' 'frisbee']
```

结果显示，检测出了 5 个类别，其中前 4 个是人物，最后一个 是飞盘。

同时又输出了最终可视化结果，如图 8-45 所示。



图 8-45 检测器的输出结果

至此，Mask R-CNN 模型已经完成了目标检测任务。从图 8-45 中可以看到，该网络可以精准定位实物坐标，并且对其进行分类。

### 8.7.21 代码实现：根据 ROI 内容进行实物像素分割

整个 Mask R-CNN 模型的最后一个环节就是实物像素分割。它可以让网络模型理解像素级别的语义。该环节通过函数 build\_fpn\_mask\_graph 来实现。build\_fpn\_mask\_graph 函数的功能主

要是：根据 DetectionLayer 返回的矩形框，用 ROIAlign 方法对特征进行池化提取（该特征来自骨干网经过 FPN 处理后的结果）；并将池化后的特征经过 4 个  $3 \times 3$  的卷积层，再进行一次上采样；最终通过全连接（用卷积代替）得到 81 个区域大小为  $28 \times 28$  的掩码。具体做法如下。

### 1. 在 MaskRCNN 类中添加 build\_fpn\_mask\_graph 实现

在 MaskRCNN 类中添加代码，将 DetectionLayer 层返回的矩形框提出来，输入函数 build\_fpn\_mask\_graph 进行像素分割，并完成 MaskRCNN 类中构建模型的全部功能。具体代码如下：

代码 8-24 mask\_rcnn\_model (续)

```

291         #像素分割
292         detection_boxes = KL.Lambda(lambda x: x[:, :, :4])(detections) # 取出 box 坐标
293         mrcnn_mask = build_fpn_mask_graph(detection_boxes,
294             mrcnn_feature_maps,
295             input_image_meta, MASK_POOL_SIZE, #14
296             self.num_class, self.batch_size, train_bn=False) #不用 bn
297         model = KM.Model([input_image, input_image_meta, input_anchors], #
    输入参数
298 [detections, mrcnn_class, mrcnn_bbox, mrcnn_mask, rpn_rois, rpn_class,
    rpn_bbox], #输出
299         name='mask_rcnn')
300     visualize.display_colored_segm( #可视化
301         rpn_bboxes, rpn_scores, rpn_class, rpn_bbox, #输入
302         mrcnn_class, mrcnn_bbox, mrcnn_mask, #输出
303         image, #输入
304         title='Mask R-CNN', #输出
305         figsize=(12, 12)) #输出
306     return model

```

代码 297 行，将前面所有的输入和输出传入 KM.Model 里，完成 MaskRCNN 类中模型 keras\_model 的构建。

### 2. 实现 build\_fpn\_mask\_graph

函数 build\_fpn\_mask\_graph 的处理过程与分类器函数 fpn\_classifier\_graph 的处理过程极为相似。步骤如下：

(1) 通过 ROIAlign 算法提取 FPN 处理后的特征。

(2) 对第(1)步的结果依次进行卷积操作、上采样操作、全连接（用卷积代替）操作。

(3) 得出与分类个数相同的特征数据 (feature map)。每个 feature map 为该区域内一个类别的掩码。

具体代码如下：

代码 8-27 othernet (续)

```

384 #语义分割
385 def build_fpn_mask_graph(rois, #目标实物检测结果，标准坐标[batch, num_rois, (y1,
    x1, y2, x2)]

```

```

386     feature_maps, #FPN 特征[P2, P3, P4, P5]
387     image_meta,
388     pool_size, num_classes, batch_size, train_bn=True):
389     """
390     返回值: Masks [batch, roi_count, height, width, num_classes]
391     """
392     #ROIAlign 最终统一池化的大小为 14
393     #形状为 [batch, boxes, pool_height, pool_width, channels]
394     x = PyramidROIAlign(batch_size, [pool_size, pool_size],
395                         name="roi_align_mask")([rois, image_meta] +
396                         feature_maps)
397     #卷积层
398     x = KL.TimeDistributed(KL.Conv2D(256, (3, 3), padding="same"),
399                             name="mrcnn_mask_conv1")(x)
400     x = KL.TimeDistributed(KL.BatchNormalization(),
401                             name='mrcnn_mask_bn1')(x, training=train_bn)
402     x = KL.Activation('relu')(x)
403     x = KL.TimeDistributed(KL.Conv2D(256, (3, 3), padding="same"),
404                             name="mrcnn_mask_conv2")(x)
405     x = KL.TimeDistributed(KL.BatchNormalization(),
406                             name='mrcnn_mask_bn2')(x, training=train_bn)
407     x = KL.Activation('relu')(x)
408     x = KL.TimeDistributed(KL.Conv2D(256, (3, 3), padding="same"),
409                             name="mrcnn_mask_conv3")(x)
410     x = KL.TimeDistributed(KL.BatchNormalization(),
411                             name='mrcnn_mask_bn3')(x, training=train_bn)
412     x = KL.Activation('relu')(x) #(1, ?, 14, 14, 256)
413
414     #用反卷积进行上采样
415     x = KL.TimeDistributed(KL.Conv2DTranspose(256, (2, 2), strides=2,
416                                             activation="relu"),
417                                             name="mrcnn_mask_deconv")(x) #(1, ?, 28, 28, 256)
418
419     #用卷积代替全连接
420     x = KL.TimeDistributed(KL.Conv2D(num_classes, (1, 1), strides=1,
421                                             activation="sigmoid"),
422                                             name="mrcnn_mask")(x)
423
424     return x

```