

### 3. 可视化检测器的结果

可视化步骤如下：

- (1) 用 `utils.run_graph` 函数将掩码结果输出。
  - (2) 将第 (1) 步的掩码结果转化为图片并显示出来。
  - (3) 将模型的最终检测结果转化为图片并显示出来。

具体代码如下：

代码 8-23 Mask RCNN 应用 (续)

代码运行后，生成如图 8-46、8-47、8-48 所示图片。

- 图 8-46 是模型输出的原始掩码结果，其大小为 28 pixel×28 pixel，里面的值是 0~1 之间的浮点数相对坐标。
  - 图 8-47 是将掩码结果换算到整个图片上的像素坐标，由代码第 241 行生成。
  - 图 8-48 是将模型最终的结果叠加到原始图片上的图像。



图 8-46 模型输出的掩码结果



图 8-47 结果坐标变化后的掩码结果

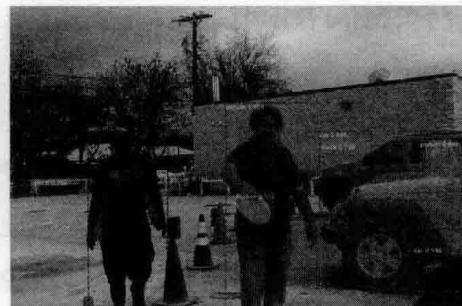


图 8-48 最终合成的结果

## 8.7.22 代码实现：用 Mask R-CNN 模型分析图片

在 MaskRCNN 类中实现 detect 方法，并通过 detect 方法用 Mask R-CNN 模型分析图片。

### 1. 实现 MaskRCNN 类的 detect 方法

实现 detect 方法的具体步骤如下：

- (1) 对输入图片做变形处理，得到变形后的图片 molded\_images 与附加信息 image\_metas。
- (2) 根据图片处理后的尺寸生成锚点框。
- (3) 调用 Mask R-CNN 模型的 predict 方法，将第（1）步的结果与锚点信息一起传入。
- (4) 调用 unmold\_detections 方法将模型的输出结果按照输入的真实图片尺寸进行还原。
- (5) 循环遍历每张输入图片，依次将其传入 unmold\_detections 方法中进行第（4）步的操作。
- (6) 将第（5）步返回的所有结果放到列表中返回。

具体代码如下：

### 代码 8-24 mask\_rcnn\_model (续)

```

302 def detect(self, images, verbose=0):#用模型进行检测
303     """用模型进行检测
304     输入: images
305     输出: 字典类型。包括如下内容
306         rois: 检测框[N, (y1, x1, y2, x2)]
307         class_ids: 类别[N]
308         scores: 分数[N]
309         masks: 掩码[H, W, N]
310     """
311     assert self.mode == "inference", "Create model in inference mode."
312     assert len(images) == self.batch_size, "len(images) must be equal to
313     BATCH_SIZE"
314     if verbose:#是否输出信息
315         print("Processing {} images".format(len(images)))
316
317

```

```

318     #图片预处理(统一大小，并返回图片附加信息)
319     molded_images, image_metas, windows = self.mold_inputs(images)
320
321     #验证尺寸
322     image_shape = molded_images[0].shape
323     for g in molded_images[1:]:
324         assert g.shape == image_shape,\n
325             "After resizing, all images must have the same size. Check\n"
326             IMAGE_RESIZE_MODE and image sizes."
327
328     #生成锚点
329     anchors = self.get_anchors(image_shape)
330     #复制锚点到批次
331     anchors = np.broadcast_to(anchors, (self.batch_size,) +
332     anchors.shape)
333
334     if verbose:
335         log("molded_images", molded_images)
336         log("image_metas", image_metas)
337         log("anchors", anchors)
338
339     #运行模型进行图片分析
340     detections, _, _, mrcnn_mask, _, _, _ = \
341         self.keras_model.predict([molded_images, image_metas, anchors],
342         verbose=0)
343
344     #处理分析结果
345     results = []
346     for i, image in enumerate(images):
347         final_rois, final_class_ids, final_scores, final_masks = \
348             self.unmold_detections(detections[i], mrcnn_mask[i],
349             image.shape, molded_images[i].shape,
350             windows[i])
351         results.append({
352             "rois": final_rois,
353             "class_ids": final_class_ids,
354             "scores": final_scores,
355             "masks": final_masks,
356         })
357
358     return results

```

在代码第328行，用get\_anchors方法生成锚点。该方法具体的实现见“2. 实现MaskRCNN类锚点生成”。

## 2. 实现MaskRCNN类锚点生成

在get\_anchors方法中加入缓存\_anchor\_cache对象，用于存放已经算好的锚点。在第1次获取锚点时，调用了下面就来介绍的get\_anchors方法与utils.generate\_pyramid\_anchors函数的实现过程来计算锚点。

(1) `get_anchors` 方法的具体实现见以下代码:

#### 代码 8-24 mask\_rcnn\_model (续)

```

354     def get_anchors(self, image_shape):
355         """根据指定图片大小生成锚点"""
356         backbone_shapes = compute_backbone_shapes( image_shape)
357         #缓存锚点
358         if not hasattr(self, "_anchor_cache"):
359             self._anchor_cache = {}
360         if tuple(image_shape) in self._anchor_cache:
361             #生成锚点
362             a =
363             utils.generate_pyramid_anchors(RPN_ANCHOR_SCALES,RPN_ANCHOR_RATIOS,
364                 backbone_shapes,BACKBONE_STRIDES,RPN_ANCHOR_STRIDE)
365             self.anchors = a
366             #设为标准坐标
367             self._anchor_cache[tuple(image_shape)] = utils.norm_boxes(a,
368             image_shape[:2])
367         return self._anchor_cache[tuple(image_shape)]

```

代码第 361 行, 对缓存对象`_anchor_cache`进行判断。如果该缓存对象中没有`image_shape`对象, 则调用`utils.generate_pyramid_anchors`函数生成锚点对象`a`, 并将`a`赋给成员变量`anchors`。

(2) `utils.generate_pyramid_anchors` 函数的具体实现见以下代码:

#### 代码 8-25 mask\_rcnn\_utils (续)

```

90     def generate_anchors(scales, ratios, shape, feature_stride, anchor_stride):
91         """
92             以 BACKBONE_STRIDES 个像素为单位, 在图片上划分网格。得到的网格按照 anchor_stride
93             进行计算, 并判断是否需要算作锚点
94             anchor_stride=1 表示都要被用作计算锚点, anchor_stride=2 表示隔一个取一个网格用
95             于计算锚点
96             每个网格第 1 个像素为中心点
97             边长由 scales 按照 ratios 种比例计算得到。每个中心点配上每种边长, 组成一个锚点
98             """
99             scales, ratios = np.meshgrid(np.array(scales), np.array(ratios))
100            scales = scales.flatten()#复制了 ratios 个 scales, 其形状为 [32,32,32]
101            ratios = ratios.flatten()#因为 scales 只有 1 个元素, 所以不变
102
103            #将比例开方再计算边长, 生成相对不规则一些的边框
104            heights = scales / np.sqrt(ratios)
105            widths = scales * np.sqrt(ratios)
106
107            #计算像素点为单位的网格位移
108            shifts_y = np.arange(0, shape[0], anchor_stride) * feature_stride
109            shifts_x = np.arange(0, shape[1], anchor_stride) * feature_stride

```

```

110     shifts_x, shifts_y = np.meshgrid(shifts_x, shifts_y) #得到x和y的位移
111
112     #将每个网格的第1点当作中心点，以3种边长为锚点大小
113     box_widths, box_centers_x = np.meshgrid(widths, shifts_x)
114     box_heights, box_centers_y = np.meshgrid(heights, shifts_y)
115
116     box_centers = np.stack(#Reshape并合并中心点坐标(y, x)
117                           [box_centers_y, box_centers_x], axis=2).reshape([-1, 2])
118     #合并边长(h, w)
119     box_sizes = np.stack([box_heights, box_widths], axis=2).reshape([-1, 2])
120
121     #将中心点边长转化为两个点的坐标(y1, x1, y2, x2)
122     boxes = np.concatenate([box_centers - 0.5 * box_sizes,
123                            box_centers + 0.5 * box_sizes], axis=1)
124     print(boxes[0]) #因为中心点从0开始，所以第1个锚点的x1、y1为负数
125     return boxes
126
127 def generate_pyramid_anchors(scales, ratios, feature_shapes,
128                               feature_strides,
129                               anchor_stride):
130     anchors = []
131     for i in range(len(scales)):#遍历不同的尺度，生成锚点
132         anchors.append(generate_anchors(scales[i], ratios,
133                                         feature_shapes[i],
134                                         feature_strides[i], anchor_stride))
135     return np.concatenate(anchors, axis=0) #[anchor_count, (y1, x1, y2, x2)]

```

代码第90行，函数generate\_anchors封装了基于在图片上划分锚点的算法。

代码第130行，在generate\_pyramid\_anchors函数内部遍历尺度列表scales，依次调用generate\_anchors函数在图片上划分不同的锚点。

### 3. 可视化检测器的结果

用模型进行图片分析的代码非常简单，只需要调用detect方法。具体代码如下：

#### 代码 8-23 Mask\_RCNN 应用（续）

```

247 results = model.detect([image], verbose=1) #用detect方法进行检测
248 r = results[0]
249 #可视化结果
250 visualize.display_instances(image, r['rois'], r['masks'], r['class_ids'],
251                             class_name, r['scores'])

```

代码运行后，可以看到与图8-33一样的效果。这里不再展示。

## 8.8 实例46：训练Mask R-CNN模型，进行形状的识别

由于Mask R-CNN模型过于庞大，本书将Mask R-CNN模型的知识点拆分成两部分：正向过程与训练部分。8.6节已经实现了Mask R-CNN模型的正向过程。本节将接着实现Mask R-CNN

的训练部分。

### 实例描述

用算法合成若干个图片，每个图片上都有不确定个数的形状。搭建 Mask R-CNN 模型，对合成图片进行训练，并用训练好的模型识别图片中的形状。

本实例需要借助 8.6 节中的代码，在其上面添加反向传播部分，使其具有可训练功能，然后训练并使用模型。

## 8.8.1 工程部署：准备代码文件及模型

将 8.7 节的代码全部复制到本地，并按照下列方式为其重命名。一共由 5 个文件组成，具体如下。

- “8-28 训练 Mask\_RCNN.py”：使用模型的全流程代码。包括训练及使用模型的代码。
- “8-29 mask\_rcnn\_model.py”：Mask-RCNN 模型的具体代码。
- “8-30 mask\_rcnn\_utils.py”：模型所需要的辅助工具代码。
- “8-31 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）。
- “8-32 mask\_rcnn\_visualize.py”：可视化部分的代码。

为了提高训练的速度，本实例同样需要使用预训练好的模型，所以需要将 8.6 节的模型 mask\_rcnn\_coco.h5 一起复制到本地路径下。

## 8.8.2 样本准备：生成随机形状图片

编写代码实现如下步骤：

- (1) 定义 ShapesDataset 类，用于生成随机形状图片。
- (2) 对 ShapesDataset 类进行实例化，得到训练数据集对象 dataset\_train 和验证数据集对象 dataset\_val。
- (3) 从数据集中取出部分样本，并显示出来。

具体代码如下：

### 代码 8-28 训练 Mask\_RCNN

```

01 import math
02 import random
03 import numpy as np
04 import cv2
05 import matplotlib.pyplot as plt #引入系统模块
06
07 mask_rcnn_model = __import__("8-29 mask_rcnn_model") #引入本地模块
08 MaskRCNN = mask_rcnn_model.MaskRCNN

```

```

09 utils = __import__("8-30 mask_rcnn_utils")
10 visualize = __import__("8-32 mask_rcnn_visualize")
11
12 #随机生成图片类
13 class ShapesDataset():
14
15     def __init__(self, class_map=None):
16         self.image_ids = []
17         .....
18
19     def get_ax(rows=1, cols=1, size=8):
20         _, ax = plt.subplots(rows, cols, figsize=(size*cols, size*rows))
21         return ax
22
23 #训练数据集 dataset
24 dataset_train = ShapesDataset()
25 dataset_train.load_shapes(500, mask_rcnn_model.IMAGE_DIM,
26                           mask_rcnn_model.IMAGE_DIM)
27 dataset_train.prepare()
28 #测试数据集 dataset
29 dataset_val = ShapesDataset()
30 dataset_val.load_shapes(50, mask_rcnn_model.IMAGE_DIM,
31                         mask_rcnn_model.IMAGE_DIM)
32 dataset_val.prepare()
33 #加载随机样本，并显示
34 image_ids = np.random.choice(dataset_train.image_ids, 4)
35 for image_id in image_ids:
36     image = dataset_train.load_image(image_id)
37     mask, class_ids = dataset_train.load_mask(image_id)
38     visualize.display_top_masks(image, mask, class_ids,
dataset_train.class_names)

```

代码运行后会生成 5 个图片，如图 8-49 所示。



图 8-49 模拟图片的部分显示

在图 8-49 中，左边第 1 个是边长 128 pixel 的图片。ShapesDataset 类会根据随机算法，向里面放置圆形、三角形、正方形。后面四个子图为该形状的标注，其中标注了具体形状的掩码信息及对应的分类。

### 8.8.3 代码实现：为 Mask R-CNN 模型添加损失函数

在 MaskRCNN 类的 build 方法中，添加代码实现损失值 loss 的处理。该代码需要添加在 RPN 之后的 mode 判断分支中（见代码第 11 行，将 loss 值处理添加到 if 语句的 else 分支中）。

具体代码如下：

代码 8-29 mask\_rcnn\_model

```

01 ..... #返回用 NMS 算法去重后前景概率值最大的 n 个 ROI (靠谱区域)
02 rpn_rois = ProposalLayer(proposal_count=proposal_count,
03   nms_threshold=RPN_NMS_THRESHOLD,batch_size=self.batch_size,
04   name="ROI")([rpn_class, rpn_bbox, anchors])
05 img_meta_size = 1 + 3 + 3 + 4 + 1 + self.num_class #定义图片的附加信息
06
07 input_image_meta = KL.Input(shape=[img_meta_size],
08   name="input_image_meta")#定义图片附加信息
09
10 if mode == "inference": #用模型预测时的代码
11   .....
12 else:
13   #获得输入数据的类
14   active_class_ids = KL.Lambda(lambda x:
15     parse_image_meta_graph(x)["active_class_ids"])
16   (input_image_meta)
17
18   if not USE_RPN_ROIS: #支持手动输入 ROI
19     input_rois = KL.Input(shape=[POST_NMS_ROIS_TRAINING, 4],
20       name="input_roi", dtype=np.int32)
21     #转为标准坐标
22     target_rois = KL.Lambda(lambda x: norm_boxes_graph(
23       x, K.shape(input_image)[1:3]))(input_rois)
24   else: #正常训练模式
25     target_rois = rpn_rois
26
27   #根据输入的样本制作 RPN 的标签
28   rois, target_class_ids, target_bbox, target_mask =
29   DetectionTargetLayer(self.batch_size,
30     name="proposal_targets")([target_rois,
31     input_gt_class_ids, gt_boxes, input_gt_masks])
32
33   #分类器
34   mrcnn_class_logits, mrcnn_class, mrcnn_bbox =
35   fpn_classifier_graph(rois, mrcnn_feature_maps, input_image_meta,
36   POOL_SIZE,
37   self.num_class, self.batch_size, train_bn=False,
38   fc_layers_size=1024) #全连接层 1024 个节点

```

```

31     #进行语义分割、掩码预测
32     mrcnn_mask = build_fpn_mask_graph(rois, mrcnn_feature_maps,
33     input_image_meta,
34             MASK_POOL_SIZE, self.num_class, self.batch_size,
35     train_bn=False)
36
37     #计算 Loss 值
38     rpn_class_loss = KL.Lambda(lambda x: rpn_class_loss_graph(*x),
39     name="rpn_class_loss")([input_rpn_match, rpn_class_logits])
40
41     rpn_bbox_loss = KL.Lambda(lambda x:
42         rpn_bbox_loss_graph(self.batch_size, *x),
43     name="rpn_bbox_loss")([input_rpn_bbox, input_rpn_match, rpn_bbox])
44
45     class_loss = KL.Lambda(lambda x:
46         mrcnn_class_loss_graph(self.num_class, self.batch_size, *x),
47     name="mrcnn_class_loss")(
48             [target_class_ids, mrcnn_class_logits, active_class_ids])
49
50     bbox_loss = KL.Lambda(lambda x: mrcnn_bbox_loss_graph(*x),
51     name="mrcnn_bbox_loss")([target_bbox, target_class_ids, mrcnn_bbox])
52
53     mask_loss = KL.Lambda(lambda x: mrcnn_mask_loss_graph(*x),
54     name="mrcnn_mask_loss")([target_mask, target_class_ids, mrcnn_mask])
55
56     #构建模型的输入节点
57     inputs = [input_image, input_image_meta, input_rpn_match,
58     input_rpn_bbox, input_gt_class_ids, input_gt_boxes, input_gt_masks]
59
60     if not USE_RPN_ROIS:
61         inputs.append(input_rois)
62
63     outputs = [rpn_class_logits, rpn_class, rpn_bbox, #构建模型的输出
64     节点
65         mrcnn_class_logits, mrcnn_class, mrcnn_bbox, mrcnn_mask, rpn_rois,
66     output_rois,
67         rpn_class_loss, rpn_bbox_loss, class_loss, bbox_loss,
68     mask_loss]
69
70     model = KM.Model(inputs, outputs, name='mask_rcnn')
71     .....

```

从代码第 10 行开始是训练模型的部分。

代码第 23 行, 用 DetectionTargetLayer 函数计算输入图片的锚点信息、分类信息与坐标框信息。这些数据将作为 RPN 的标签参与训练。

从代码第 38 行开始是计算损失值的部分。该模型的损失值包括 5 部分:

- RPN 的分类损失。
- RPN 的边框损失。
- 分类器的分类损失。
- 分类器的边框损失。
- 掩码的损失。

具体的 loss 函数可以参考本书的配套代码。这里不再展开。

## 8.8.4 代码实现：为 Mask R-CNN 模型添加训练函数，使其支持微调与全网训练

定义 MaskRCNN 类中的 train 方法，实现模型训练的全部过程。在 train 方法中，实现如下步骤：

- (1) 获得指定的训练规模，按照参数找到对应的层。见代码第 67 行。
- (2) 生成迭代器数据集，用于训练。见代码第 81 行。
- (3) 设置反向训练相关参数（优化器、正则化、学习率等）及指定层的训练开关。见代码第 100、101 行。
- (4) 训练模型。见代码第 103 行。

具体代码如下：

代码 8-29 mask\_rcnn\_model (续)

```

60     .....
61     def train(self, train_dataset, val_dataset, batch_size, learning_rate,
62     epochs, layers,
63         augmentation=None, custom_callbacks=None,
64         no_augmentation_sources=None):
65
66         #根据参数指定训练规模，用于微调
67         layer_regex = {
68             #训练除骨干网外的其他网络
69             "heads": r"(mrcnn\_.*)|(rpn\_.*)|(fpn\_.*)",
70             #选择指定的网络进行训练
71             "3+": r"(res3\_.*)|(bn3\_.*)|(res4\_.*)|(bn4\_.*)|(res5\_.*)|(bn5\_.*)|"
72             "(mrcnn\_.*)|(rpn\_.*)|(fpn\_.*)",
73             "4+": r"(res4\_.*)|(bn4\_.*)|(res5\_.*)|(bn5\_.*)|(mrcnn\_.*)|"
74             "(rpn\_.*)|(fpn\_.*)",
75             "5+": r"(res5\_.*)|(bn5\_.*)|(mrcnn\_.*)|(rpn\_.*)|(fpn\_.*)",
76             #全部训练
77             "all": "*",
78         }
79         if layers in layer_regex.keys():
80             layers = layer_regex[layers]

```

```

79
80      #生成数据
81      train_generator = data_generator(train_dataset,
82          shuffle=True, augmentation=augmentation,
83          batch_size=batch_size,
84          no_augmentation_sources=
85          no_augmentation_sources, num_class = self.num_class)
86
87      #添加日志存储的回调函数
88      callbacks = [
89          keras.callbacks.TensorBoard(log_dir=self.log_dir,
90              histogram_freq=0, write_graph=True,
91              write_images=False),
92          keras.callbacks.ModelCheckpoint(self.checkpoint_path,
93              verbose=0, save_weights_only=True),
94      ]
95      if custom_callbacks:
96          callbacks += custom_callbacks
97
98      #开始训练 Train
99      log("\nStarting at epoch {}. LR={}\n".format(self.epoch,
100         learning_rate))
101      log("Checkpoint Path: {}".format(self.checkpoint_path))
102      self.set_trainable(layers)           #根据指定的层设置训练开关
103      self.compile(learning_rate, LEARNING_MOMENTUM) #设置模型的优化器及学习
参数
104
105      self.keras_model.fit_generator(      #调用 fit_generator 进行训练
106          train_generator,
107          initial_epoch=self.epoch,
108          epochs=epochs,
109          steps_per_epoch=STEPS_PER_EPOCH,
110          callbacks=callbacks,
111          validation_data=val_generator,
112          validation_steps=VALIDATION_STEPS,
113          max_queue_size=100,
114          workers=0,
115          use_multiprocessing=False,
116          max(self.epoch, epochs)

```

代码 81 行，用函数 `data_generator` 生成训练使用的数据集。由于 Mask R-CNN 属于两阶段训练模型，在制作结果标签之外，还需要制作 RPN 标签。

在函数 `data_generator` 中，用 `build_rpn_targets` 函数实现了 RPN 标签的制作。

**提示：**

在训练 RPN 过程中，需要将锚点 anchors、样本、标注这三个信息合成 RPN 标签，这样才可以进行监督式训练。合成 RPN 标签的过程如下：

- (1) 根据样本和标注提取出图片的分类标签信息和矩形框标签信息。
- (2) 将锚点中的矩形框与矩形框标签信息按照区域重合度进行匹配。
- (3) 对每个锚点进行前景和背景的分类：将与矩形框标签信息匹配的锚点设为前景标签；将与矩形框标签信息不匹配的锚点设为背景标签。
- (4) 计算所有前景锚点与其矩形坐标框标签之间的坐标偏移（中心点偏移和边长的缩放比例）。
- (5) 将第(4)步所计算出的坐标偏移值除以 RPN\_BBOX\_STD\_DEV 进行归一化处理。

由于篇幅原因，这里不再将函数 data\_generator 与函数 build\_rpn\_targets 的代码一一列出，读者可以参考随书配套的代码资源自行查看。

## 8.8.5 代码实现：训练并使用模型

MaskRCNN 类中的代码准备好了之后，便开始搭建主体流程。

### 1. 创建 Mask R-CNN 模型，并加载权重

指定训练批次，用训练模式构建模型。具体代码如下：

**代码 8-28 训练 Mask\_RCNN (续)**

```

39 BATCH_SIZE = 3           # 批次
40 NUM_CLASSES = 1 + 3    # 1 个背景类和 3 个形状类
41 # 创建训练模式模型
42 MODEL_DIR = "./log"
43 model = MaskRCNN(mode="training", model_dir=MODEL_DIR,
44                   num_class=dataset_train.num_classes, batch_size = BATCH_SIZE)
45 # 模型权重文件路径
46 weights_path = "./mask_rcnn_coco.h5"
47 # 载入权重文件
48 print("Loading weights ", weights_path)
49 model.load_weights(weights_path,
50                     by_name=True, exclude=["mrcnn_class_logits", "mrcnn_bbox_fc",
51 "mrcnn_bbox", "mrcnn_mask"])

```

### 2. 训练并保存 Mask R-CNN 模型

训练模型分为两步：

(1) 固定骨干网的权重，训练其他层。

(2) 设置较低的学习率，对整个网络进行继续训练。

具体代码如下：

#### 代码 8-28 训练 Mask\_RCNN（续）

```

51 model.train(dataset_train, dataset_val, batch_size = BATCH_SIZE,
52             learning_rate=mask_rcnn_model.LEARNING_RATE,
53             epochs=1,
54             layers='heads')
55
56 model.train(dataset_train, dataset_val, batch_size = BATCH_SIZE,
57             learning_rate=mask_rcnn_model.LEARNING_RATE / 10,
58             epochs=2,
59             layers="all")
60 #保存模型
61 import os
62 MODEL_DIR = "mask_model"
63 model_path = os.path.join(MODEL_DIR, "mask_rcnn_shapes.h5")
64 model.keras_model.save_weights(model_path)

```

代码运行后，系统会在本地的 mask\_model 文件夹下生成模型文件 mask\_rcnn\_shapes.h5，并显示如下结果：

```

.....
Epoch 2/2
.....
99/100 [=====>.] - ETA: 9s - loss: 0.9817 - rpn_class_loss:
0.0166 -
.....
100/100 [=====] - 933s 9s/step - loss: 0.9780 -
rpn_class_loss: 0.0165 - rpn_bbox_loss: 0.4315 - mrcnn_class_loss: 0.2105 -
mrcnn_bbox_loss: 0.1693 - mrcnn_mask_loss: 0.1501 - val_loss: 0.9802 - val_rpn_class_loss:
0.0170 - val_rpn_bbox_loss: 0.5260 - val_mrcnn_class_loss: 0.1228 - val_mrcnn_bbox_loss:
0.1543 - val_mrcnn_mask_loss: 0.1601
.....

```

### 3. 用 Mask R-CNN 模型进行识别

编写代码使用模型，具体步骤如下：

- (1) 重新实例化一个模型 model2。
- (2) 载入训练好的模型权重。
- (3) 随机取出一张模拟图片。
- (4) 将取出的图片传入模型进行预测。
- (5) 将图片的标签信息与模型的预测结果分别叠加到模拟图片上，并显示出来。

具体代码如下：

#### 代码 8-28 训练 Mask\_RCNN（续）

```

65 MODEL_DIR = "mask_model"
66 model_path = os.path.join(MODEL_DIR, "mask_rcnn_shapes.h5")

```

```

67 #重新构建模型
68 model2 = MaskRCNN(mode="inference", model_dir=MODEL_DIR,
69                     num_class=dataset_train.num_classes,batch_size = 1)#加完背景后的 81 个类
70
71 #加载模型
72 print("Loading weights from ", model_path)
73 model2.load_weights(model_path, by_name=True)
74
75 #随机取出图片
76 image_id = random.choice(dataset_val.image_ids)
77 original_image, image_meta, gt_class_id, gt_bbox, gt_mask =\
78     mask_rcnn_model.load_image_gt(dataset_val, image_id,
79     use_mini_mask=False)
80
81 ax = get_ax(1, 2)
82 #显示原始图片及标注
83 visualize.display_instances(original_image, gt_bbox, gt_mask, gt_class_id,
84                             dataset_train.class_names, ax=ax[0])
85
86 #用模型进行预测，并显示结果
87 results = model2.detect([original_image], verbose=1)
88 r = results[0]
89 visualize.display_instances(original_image, r['rois'], r['masks'],
90                             r['class_ids'],
91                             dataset_val.class_names, r['scores'], ax=ax[1])

```

代码运行后，生成的结果如图 8-50 所示。

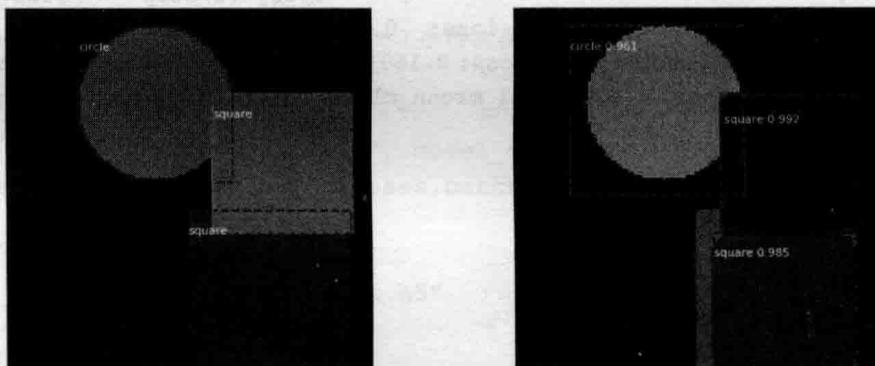


图 8-50 Mask R-CNN 模型训练后的识别结果

如图 8-50 所示，左边为样本的图片及标注，右边为模型生成的图片及标注。由于整个数据集只迭代了两次，所以误差还比较大。读者可以把迭代次数加大，以便训练出更精准的模型。

#### 4. 为模型评分

随机抽取 10 张照片，输入 Mask R-CNN 模型，并将模型生成的结果与原始图片的标注进行比较，得出模型的评分。具体代码如下：

**代码 8-28 训练 Mask\_RCNN (续)**

```

89 image_ids = np.random.choice(dataset_val.image_ids, 10)
90 APs = []
91 for image_id in image_ids:
92     #原始图片
93     image, image_meta, gt_class_id, gt_bbox, gt_mask =\
94         mask_rcnn_model.load_image_gt(dataset_val, image_id,
95         use_mini_mask=False)
95     molded_images = np.expand_dims(utils.mold_image(image), 0)
96     #运行结果
97     results = model2.detect([image], verbose=0)
98     r = results[0]
99     #计算模型分数
100    AP, precisions, recalls, overlaps =\
101        utils.compute_ap(gt_bbox, gt_class_id, gt_mask,
102                          r["rois"], r["class_ids"], r["scores"], r['masks'])
103    APs.append(AP)
104
105 print("mAP: ", np.mean(APs))

```

代码运行后，输出如下结果：

```
mAP: 0.9333333373069763
```

结果表示，模型的平均精度为 0.93。其中的 mAP (Mean Average Precision) 代表平均精度。

### 8.8.6 扩展：替换特征提取网络

在 YOLO V3 模型的论文 (<https://pjreddie.com/media/files/papers/YOLOv3.pdf>) 中，比较用 Darknet-53 模型提取的特征结果与 ResNet 模型提取的特征结果，得到的结论是：Darknet-53 模型提取的特征在 YOLO V3 模型中表现更优。如图 8-51 所示。

Backbone	Top-1	Top-5	Bn Ops	BFLOPs	FPS
Darknet-19 [13]	74.1	91.8	7.29	1246	<b>171</b>
ResNet-101[3]	77.1	93.7	19.7	1039	53
ResNet-152 [3]	<b>77.6</b>	<b>93.8</b>	29.4	1090	37
Darknet-53	77.2	<b>93.8</b>	18.7	<b>1457</b>	78

图 8-51 Darknet-53 模型的特征结果与 ResNet 模型的特征结果比较

读者可以尝试将 MaskRCNN 类中的骨干网 ResNet 模型替换成 Darknet-53 模型，并使用 8.8.5 小节“4. 为模型评分”的方法进行测试。观察 Darknet-53 模型在 Mask R-CNN 模型中是否也会表现出更好的效果。

# 第 9 章

## 循环神经网络（RNN）——处理序列样本的神经网络

循环神经网络（Recurrent Neural Networks，RNN）具有记忆功能，它可以发现样本之间的序列关系，是处理序列样本的首选模型。循环神经网络大量应用在数值、文本、声音、视频处理等领域。本章介绍循环神经网络中相关的计算单元及主流的网络架构。



### 提示：

本章内容偏重于讲解循环神经网络的搭建与具体应用，淡化了循环神经网络中的原理。例如词向量、词嵌入、各种 cell 结构等基础知识点及循环神经网络的底层原理，还需要读者额外学习。这里推荐读者参考《深度学习之 TensorFlow——入门、原理与进阶实战》一书的第 9 章内容。

### 9.1 快速导读

在学习本实例之前，读者有必要了解一下循环神经网络的基础知识。

#### 9.1.1 什么是循环神经网络

循环神经网络模型（以下简称 RNN 模型）是一个具有记忆功能的模型。它可以发现样本之间的相互关系，多用于处理带有序列特征的样本数据。

RNN 模型有很多种结构，其最基本的结构是将全连接网络的输出节点复制一份并传回到输入节点中，与输入数据一起进行下一次运算。这种神经网络将数据从输出层又传回到输入层，形成了循环结构，所以被叫作循环神经网络。

通过 RNN 模型，可以将上一个序列的样本输出结果与下一个序列样本一起输入模型中进行运算，使模型所处理的特征信息中，既含有该样本之前序列的信息，又含有该样本自身的数据信息，从而使网络具有记忆功能。

在实际开发中，所使用的 RNN 模型还会基于上述的原理做更多的结构改进，使网络的记忆功能更强。

在深层网络结构中，还会在 RNN 模型基础上结合全连接网络、卷积网络等组成拟合能力

更强的模型。

## 9.1.2 了解 RNN 模型的基础单元 LSTM 与 GRU

RNN 模型的基础结构是单元，其中比较常见的有 LSTM 单元、GRU 单元等，它们充当了 RNN 模型中的基础结构部分。使用单元搭建出来的 RNN 模型会有更好的拟合效果。

LSTM 单元与 GRU 单元是 RNN 模型中最常见的单元，其内部由输入门、忘记门和输出门三种结构组合而成。

LSTM 单元与 GRU 单元的作用几乎相同，唯一不同的是：

- LSTM 单元返回 cell 状态和计算结果。
- GRU 单元只返回计算结果，没有 cell 状态。

相比之下，使用 GRU 单元会更加简单。

## 9.1.3 认识 QRNN 单元

QRNN (Quasi-Recurrent Neural Networks) 单元是一种 RNN 模型的基础单元，它比 LSTM 单元的速度更快。

QRNN 单元被发表于 2016 年。它使用卷积操作替代传统的循环结构，其网络结构介于 RNN 与 CNN 之间。

QRNN 内部的卷积结构可以将序列数据以矩阵方式同时运算，不再像循环结构那样必须按照序列顺序依次计算。其以并行的运算方式取代了串行，提升了运算速度。在训练时，卷积结构也要比循环结构的效果更加稳定。

在实际应用中，QRNN 单元可以与 RNN 模型中的现有单元随意替换。

如果想更多了解 QRNN，可以参考以下论文：

<https://arxiv.org/abs/1611.01576>

## 9.1.4 认识 SRU 单元

SRU 单元是 RNN 模型的基础单元。它的作用与 QRNN 单元类似，也是对 LSTM 单元在速度方面进行了提升。

LSTM 单元必须要将样本按照序列顺序一个个地进行运算，才能够输出结果。这种运算方式使得该单元无法在多台机器并行计算的环境中发挥最大的作用。

SRU 单元被发表于 2017 年。它保留了 LSTM 单元的循环结构，通过调整运算先后顺序的方式（把矩阵乘法放在串行循环外，把相乘的再相加的运算放在串行循环内）提升了运算速度。

### 1. SRU 单元的结构

SRU 单元在本质上与 QRNN 单元很像。从网络构建上看，SKV 单元有点像 QRNN 单元中的一个特例，但是又比 QRNN 单元多了一个直连的设计。

若需要研究 SKV 单元更深层面的理论，可以参考如下论文：

<https://arxiv.org/abs/1709.02755>

## 2. SRU 单元的使用

在 TensorFlow 中，用函数 `tf.contrib.rnn.SRUCell` 可以使用 SRU 单元。该函数的用法与函数 `LSTMCell` 的用法完全一致（函数 `LSTMCell` 是 LSTM 单元的实现）。

关于函数 `tf.contrib.rnn.SRUCell` 的更多使用方法，可以参考官方帮助文档：

[https://www.tensorflow.org/api\\_docs/python/tf/contrib/rnn/SRUCell](https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/SRUCell)

## 9.1.5 认识 IndRNN 单元

IndRNN 单元是一种新型的循环神经网络单元结构，被发表于 2018 年，其效果和速度均优于 LSTM 单元。

IndRNN 单元不仅可以改善传统 RNN 模型所存在的梯度消失和梯度爆炸问题，还能够更好地学习样本中的长期依赖关系。

在搭建模型时：

- 以堆叠的方式使用 IndRNN 单元，可以搭建出更深的网络结构。
- 将 IndRNN 单元配合 ReLu 等非饱和激活函数一起使用，会使模型表现出更好的鲁棒性。

有关 IndRNN 单元的更多理论，可以参考论文：<https://arxiv.org/abs/1803.04831>。

### 1. IndRNN 单元与 RNN 模型其他单元的结构差异

与 LSTM 单元相比，IndRNN 单元的结构要简单得多。它更像一个原始的 RNN 模型结构（只将神经元的输出复制到输入节点中）。

与原始的 RNN 模型相比，IndRNN 单元主要在循环层部分做了特殊处理。下面通过公式来详细介绍。

### 2. 原始的 RNN 模型结构

原始的 RNN 模型结构见式（9.1）：

$$\mathbf{h}_t = \sigma(\mathbf{W}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + b) \quad (9.1)$$

在式（9.1）中， $\sigma$  代表激活函数， $\mathbf{W}$  代表权重， $\mathbf{x}$  代表输入， $\mathbf{U}$  代表循环层的权重， $\mathbf{h}$  代表前一个序列的输出， $b$  代表偏置。

在原始的 RNN 模型结构中，每个序列的输入数据乘以权重后，都要加上一个序列的输出与循环层的权重相乘的结果，再加上偏置，得到最终的结果。

### 3. IndRNN 单元的结构

IndRNN 单元的结构见式（9.2）：

$$\mathbf{h}_t = \sigma(\mathbf{W}\mathbf{x}_t + \mathbf{U} \odot \mathbf{h}_{t-1} + b) \quad (9.2)$$

式（9.2）与式（9.1）相比，不同之处在于  $\mathbf{U}$  与  $\mathbf{h}$  的运算。符号  $\odot$  代表两个矩阵的哈达玛积（Hadamard product），即两个矩阵的对应位置相乘。

在 IndRNN 单元中，要求  $\mathbf{U}$  和  $\mathbf{h}$  这两个矩阵的形状必须完全相同。

IndRNN 单元的核心就是将上一个序列的输出与循环层的权重进行哈达玛积操作。从某种

角度来讲，循环层的权重更像是卷积网络中的卷积核，该卷积核会对序列样本中的每个序列做卷积操作。

#### 4. TensorFlow 中的 IndRNN 单元

在 TensorFlow 1.10 之后的版本中提供了 IndRNNCell 类，它封装了 IndRNN 单元，并在 IndRNN 单元的基础上增加了与 GRU 单元和 LSTM 单元一样的门结构，生成 IndyGRUCell 类与 IndyLSTMCell 类。其用法与代码中 GRU 单元和 LSTM 单元的用法一样。具体用法见 9.4 节。

### 9.1.6 认识 JANET 单元

JANET 单元也是对 LSIM 单元的一种优化，被发表于 2018 年。该网络源于一个很胆的猜测——当 LSTM 单元只有忘记门会怎样？

实验表明，只有忘记门的网络，其性能居然优于标准 LSTM 单元。同样，该优化方式也可以被用在 GRU 单元中。

如想要了解更多关于 JANET 单元的内容，可以参考以下论文：

<https://arxiv.org/abs/1804.04849>

有关 JANET 单元在 RNN 模型中的实际应用，请参考本书的 9.5 节。

### 9.1.7 优化 RNN 模型的技巧

在优化 RNN 模型时，也需要使用例如批量正则化方法、dropout 方法等提升模型效果。

由于 RNN 模型具有独特的网络结构，在实现时，与常规优化技巧相比，基于 RNN 模型的优化技巧会略有不同。具体细节可以在本书的其他章节中找到详细内容，例如，基于 RNN 模型的 dropout 方法（见 9.4 实例）、基于 RNN 模型的批量正则化技术（见 10.1.6 小节）。

### 9.1.8 了解 RNN 模型中多项式分布的应用

自然语言一句话中的某个词并不是唯一固定的。例如“代码医生工作室真棒”这句话中的最后一个字“棒”，也可以换成“好”，不会影响整句话的语义。

在 RNN 模型中，将一个使用语言样本训练好的模型用于生成文本时，会发现模型总会将在下一时刻出现概率最大的那个词取出。这种生成文本的方式失去了语言本身的多样性。

为了解决这个问题，这里将 RNN 模型的最终结果当作一个多项式分布（Multinomial Distribution），以分布取样的方式预测出下一序列的词向量。用这种方法所生成的句子更符合语言的特性。

#### 1. 多项式分布

多项式分布是二项式分布的拓展。在学习多项式分布之前，先学习二项式分布比较容易。

二项式分布又被称为伯努利（Bernoulli）分布，其中典型的例子是“扔硬币”：硬币正面朝上的概率为  $p$ ，重复扔  $n$  次硬币，所得到  $k$  次正面朝上的概率，即为一个二项式分布概率。

把二项式分布公式拓展至多种状态，就得到了多项式分布。

## 2. RNN 模型中多项式分布的应用

在 RNN 模型中，预测的结果不再是下一个序列中出现的具体某一个词，而是这个词的分布情况。这便是在 RNN 模型中使用多项式分布的核心思想。

在获得该词的多项式分布之后，便可以在该分布中进行采样操作，获得具体的词。这种方式更符合 NLP 任务中语言本身的多样性（一个句子中的某个词并不是唯一的）。

在实际的 RNN 模型中，具体的实现步骤如下。

- (1) 将 RNN 模型预测的结果通过全连接或卷积，变换成与字典维度相同的数组。
- (2) 用该数组代表模型所预测结果的多项式分布。
- (3) 用 `tf.multinomial` 函数从预测结果中采样，得到真正的预测结果。

## 3. 函数 `tf.multinomial` 的使用方法

函数 `tf.multinomial` 可以按批次处理数据。该函数的使用细节如下。

- 在使用时：需要传入一个形状是`[batch_size, num_classes]`的分布数据。
- 在执行时：会按照分布数据中的 `num_classes` 概率抽取指定个数的样本并返回。

完整的示例代码如下：

```
import numpy as np
import tensorflow as tf
b = tf.constant(np.random.normal(size = (2, 4))) #生成一串随机数
with tf.Session() as sess:
    print(sess.run(b)) #输出: [[ 0.14730237  0.10002697 -0.3397995   0.08918727]
                           #[ 2.00974768 -1.30524175 -0.30822854  1.75512202]]
    print(sess.run(tf.multinomial(b, 1))) #按照 b 的分布进行 1 个数据的采样，输出: [[2] [0]]
    print(sess.run(tf.multinomial(b, 1))) #第二次采样，输出: [[0] [0]]
```

从上面的示例代码中可以看到，对于一个指定的多项式分布，多次采样可以得到不同的值。将多项式采样用于 RNN 模型的输出处理，更符合 NLP 的样本特性。

### 9.1.9 了解注意力机制的 Seq2Seq 框架

带注意力机制的 Seq2Seq (`attention_Seq2Seq`) 框架常用于解决 Seq2Seq 任务。为了防止读者对概念混淆，下面对 Seq2Seq 相关的任务、框架、接口、模型做出统一解释。

- **Seq2Seq(Sequence2Sequence)任务：**从一个序列(Sequence)映射到另一个序列(Sequence)的任务，例如：语音识别、机器翻译、词性标注、智能对话等。
- **Seq2Seq 框架：**也被叫作编解码框架（即 Encoder-Decoder 框架）是一种特殊的网络模型结构。这种结构适合于完成 Seq2Seq 任务。
- **Seq2Seq 接口：**是指用代码实现的 Seq2Seq 框架函数库。在 Python 中，以模块的方式提供给用户使用。用户可以使用 Seq2Seq 接口来进行模型的开发。
- **Seq2Seq 模型：**用 Seq2Seq 接口实现的模型被叫作 Seq2Seq 模型。

## 1. 了解 Seq2Seq 框架

Seq2Seq 任务的主流解决方法是使用 Seq2Seq 框架（即 Encoder-Decoder 框架）。Encoder-Decoder 框架的工作机制如下。

(1) 用编码器 (Encoder) 将输入编码映射到语义空间中，得到一个固定维数的向量，这个向量就表示输入的语义。

(2) 用解码器 (Decoder) 将语义向量解码，获得所需要的输出。如果输出的是文本，则解码器 (Decoder) 通常就是语言模型。

Encoder-Decoder 框架的结构如图 9-1 所示。

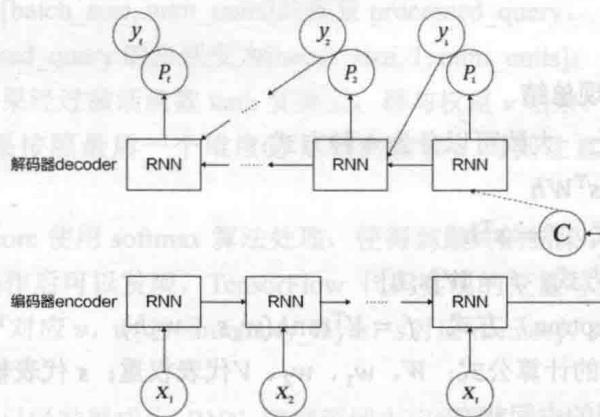


图 9-1 Encoder-Decoder 框架结构

该网络框架擅长解决：语音到文本、文本到文本、图像到文本、文本到图像等转换任务。

## 2. 了解带有注意力机制的 Seq2Seq 框架

注意力机制可用来计算输入与输出的相似度。一般将其应用在 Seq2Seq 框架中的编码器 (Encoder) 与解码器 (Decoder) 之间，通过给输入编码器的每个词赋予不同的关注权重，来影响其最终的生成结果。这种网络可以处理更长的序列任务。其具体结构如图 9-2 所示。

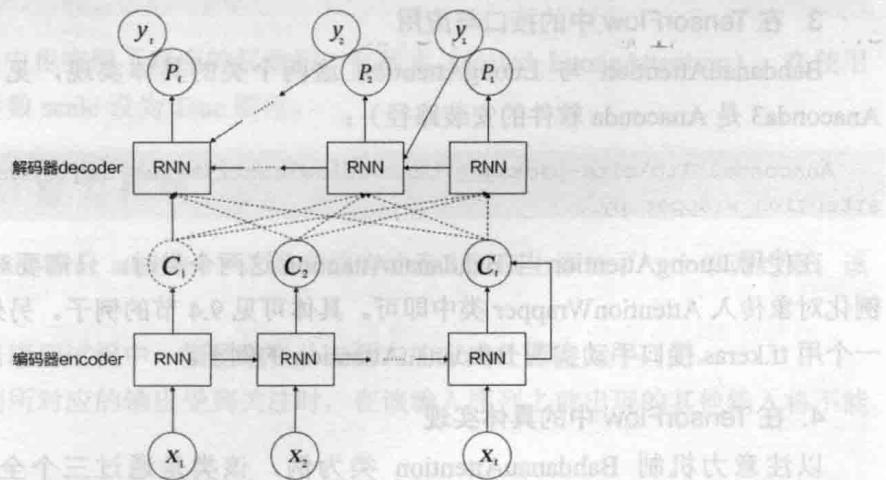


图 9-2 带有注意力机制的 Seq2Seq 框架

图 9-2 中的框架只是注意力机制中的一种。在实际应用中，注意力机制还有很多其他的变化。其中包括 LuongAttention、BahdanauAttention、LocationSensitiveAttention 等。更多关于注意力机制的内容还可以参考论文：

<https://arxiv.org/abs/1706.03762>

### 9.1.10 了解 BahdanauAttention 与 LuongAttention

在 TensorFlow 的 Seq2Seq 接口中实现了两种注意力机制的类接口：BahdanauAttention 与 LuongAttention。在介绍这两种注意力机制的区别之前，先系统地介绍一下注意力机制的几种实现方法。

#### 1. 注意力机制的实现总结

注意力机制在实现上，大致可以分为 4 种方式：

- 一般方式： $f = s^T Wh$  (9.3)

- 点积 (dot) 方式： $f = s^T h$  (9.4)

- 连接 (concat) 方式： $f = W[s; h]$  (9.5)

- 神经网络 (perceptron) 方式： $f = V^T \tanh(w_1 s + w_2 h)$  (9.6)

其中， $f$  代表注意力的计算公式； $W$ 、 $w_1$ 、 $w_2$ 、 $V$  代表权重； $s$  代表输入；上标的 T 代表矩阵转置； $h$  代表解码序列的中间状态。

#### 2. BahdanauAttention 与 LuongAttention 的区别

BahdanauAttention 与 LuongAttention 这两种注意力机制分别是由 Bahdanau 与 Luong 这两个作者实现的。前者是使用一般方式实现的，见式 (9.3)；后者使用的是使用神经网络方式实现的，见式 (9.6)。其对应的论文如下：

- BahdanauAttention: <https://arxiv.org/abs/1409.0473>
- LuongAttention: <https://arxiv.org/abs/1508.04025>

#### 3. 在 TensorFlow 中的接口与应用

BahdanauAttention 与 LuongAttention 这两个类的具体实现，见以下代码文件（其中的 Anaconda3 是 Anaconda 软件的安装路径）：

```
Anaconda3\lib\site-packages\tensorflow\contrib\seq2seq\python\ops\attention_wrapper.py
```

在使用 LuongAttention 与 BahdanauAttention 这两个类时，只需要对其进行实例化，并将实例化对象传入 AttentionWrapper 类中即可。具体可见 9.4 节的例子。另外，在 9.3 节中还介绍了一个用 tf.keras 接口手动实现 BahdanauAttention 的例子。

#### 4. 在 TensorFlow 中的具体实现

以注意力机制 BahdanauAttention 类为例，该类是通过三个全连接 (memory\_layer、query\_layer、v) 方式实现的，具体步骤如下。

(1) 在初始化时, 需要传入编码器的输出结果 `memory`(形状为`[batch_size, max_time, endim]`) 和注意力深度 `num_units`(全连接权重的神经元个数)。其中, 变量 `endim` 是编码器单元的个数。

(2) 在基类 `_BaseAttentionMechanism` 中, 会用全连接层 `memory_layer` 对编码器结果 `memory` 进行处理, 生成形状为`[batch_size, max_time, num_units]`的张量。该张量将作为 `keys`。

(3) 在解码过程中, 会将上一时刻的目标值  $y_{t-1}$  传入解码器中, 并将解码输出结果当作查询条件 `query`(形状为`[batch_size, dedim]`) 传入 `BahdanauAttention` 实例进行注意力计算。其中变量 `dedim` 是解码器的 cell 个数。

(4) 在 `BahdanauAttention` 类的 `__call__` 函数中, 用全连接层 `query_layer` 对解码结果 `query` 进行处理, 生成形状为`[batch_size, num_units]`的张量 `processed_query`。

(5) 将张量 `processed_query` 的形状变为`[batch_size, 1, num_units]`, 并与张量 `keys` 相加。

(6) 将相加后的结果经过激活函数 `tanh` 变换后, 再与权重 `v` 相乘。

(7) 将最后的结果按照最后一个维度进行规约加和, 得到注意力值 `score`。其形状是`[batch_size, max_time]`

(8) 对注意力值 `score` 使用 `softmax` 算法处理, 便得到最终的结果。

按照上面的步骤操作后可以发现, TensorFlow 代码实现的变量与式 (9.6)  $V^T \tanh(w_1 s + w_2 h)$  的对应关系是:  $V^T$  对应 `v`,  $w_1$  对应 `memory_layer`,  $s$  对应 `memory`,  $w_2$  对应 `query_layer`,  $h$  对应 `query`。

目前, 注意力机制已经发展成为 RNN 模型领域中应用最广的技术。建议读者结合上述过程说明和 TensorFlow 的源码仔细练习, 尽量达到熟练掌握的程度。该技术在序列任务处理中会大有用处。

### 5. normed\_BahdanauAttention 与 scaled\_LuongAttention

在 `BahdanauAttention` 类中有一个权重归一化的版本 (`normed_BahdanauAttention`), 它可以加快随机梯度下降的收敛速度。在使用时, 将初始化函数中的参数 `normalize` 设为 `True` 即可。

具体可以参考以下论文:

<https://arxiv.org/pdf/1602.07868.pdf>

在 `LuongAttention` 类中也实现了对应的权重归一化版本 (`scaled_LuongAttention`)。在使用时, 将初始化函数中的参数 `scale` 设为 `True` 即可。

## 9.1.11 了解单调注意力机制

单调注意力机制 (monotonic attention), 是在原有注意力机制上添加了一个单调约束。该单调约束的内容为:

(1) 假设在生成输出序列过程中, 模型是以从左到右的方式处理输入序列的。

(2) 当某个输入序列所对应的输出受到关注时, 在该输入序列之前出现的其他输入将不能在后面的输出中被关注。

即已经被关注过的输入序列, 其前面的序列中不再被关注。

更多描述可以参考以下论文:

<https://arxiv.org/pdf/1704.00784.pdf>

### 1. 在 TensorFlow 中的接口

在 TensorFlow 中，单调注意力机制有两个接口类：

- BahdanauMonotonicAttention 类。
- LuongMonotonicAttention 类。

在这两个类中，使用了同样的单调算法。在这两个类的实例化参数中，与单调注意力机制相关的参数有 3 个。

- sigmoid\_noise：用于调节注意力分数，默认值为 0.0。
- sigmoid\_noise\_seed：用于调节注意力分数，默认值为 None。
- mode：用于指定单调注意力机制的运算方式，默认值为 parallel。

### 2. 在 TensorFlow 中的具体实现

单调注意力机制（monotonic attention）的实现与原始的注意力机制仅有很小的变化。以 BahdanauMonotonicAttention 为例，在 9.1.10 小节中“4. 在 TensorFlow 中的具体实现”里的第（8）步，在对注意力分值 score 进行 softmax 算法处理时做了变化：将 softmax 算法换成了单调注意力算法（见源代码中的 \_monotonic\_probability\_fn 函数）。

在 \_monotonic\_probability\_fn 函数中会对传入的注意力分数做一次变化：用 sigmoid\_noise 与 sigmoid\_noise\_seed 两个参数进行调节。具体见以下代码：

```
if sigmoid_noise > 0:
    noise = random_ops.random_normal(array_ops.shape(score), dtype=score.dtype,
                                      seed=seed) #seed 的值为 sigmoid_noise_seed
    score += sigmoid_noise*noise
```

在调节注意力分数之后，可选择 3 种方式进行具体运算。这 3 种方式由参数 mode 来指定，具体介绍如下。

- 递归方式：取值为 recursive。用函数 tf.scan 递归计算分布。此方式虽然速度慢，但是精确。
- 并行方式：取值为 parallel。用并行的 cumulative-sum 函数和 cumulative-produce 函数计算注意力分布。此方式比递归方式效率高。如果输入序列 input\_sequence\_length 很长，或 p\_choose\_i（第 i 个输入序列元素的概率）非常接近 0 或 1，则计算出的注意力分布将会很不精确。为了避免这种情况，在使用该方式之前必须要对数字进行检查。
- 硬方式：取值为 hard。要求 p\_choose\_i 中的概率都是 0 或 1，此方式更有效、更精确。

如果 mode 值是 hard，则一般会将参数 sigmoid\_noise 的值设为大于 0。这样，模型会对现有的注意力分数进行放大，使注意力分数在 one-hot 编码转换时散列得更好。

如果在测试场景中，或是在 mode 值不是 hard 时，则建议将参数 sigmoid\_noise 的值设为 0。

## 9.1.12 了解混合注意力机制

混合注意力（hybrid attention）机制又被称作位置敏感注意力（location sensitive attention）机制，它主要是将上一时刻的注意力结果当作该序列的位置特征信息，并添加到原有注意力机

制基础上。这样得到的注意力中就会有内容和位置两种信息。

因为混合注意力中含有位置信息，所以它可以在输入序列中选择下一个编码的位置。这样的机制更适用于输出序列大于输入序列的 Seq2Seq 任务，例如语音合成任务（见 9.8 节）。

具体可以参考以下论文：

<https://arxiv.org/pdf/1506.07503.pdf>

### 1. 混合注意力机制的结构

在论文中，混合注意力机制的结构见式（9.7）。

$$a_i = \text{Attend}(s_{i-1}, a_{i-1}, h_i) \quad (9.7)$$

在式（9.7）中，符号的具体含义如下。

- $h$  代表编码后的中间状态（代表内容信息）。
- $a$  代表分配的注意力分数（代表位置信息）。
- $s$  代表解码后的输出序列。
- $i-1$  代表上一时刻。
- $i$  代表当前时刻。

可以将混合注意力分数  $a$  的计算描述为：上一时刻的  $s$  和  $a$ （位置信息）与当前时刻的  $h$ （内容信息）的点积计算结果。

$\text{Attend}$  代表注意力计算的整个流程。按照式（9.7）的方式，不带位置信息的注意力机制可以表述为：

$$a_i = \text{Attend}(s_{i-1}, h_i) \quad (9.8)$$

式（9.7）与式（9.8）的区别是：式（9.7）中多了一个  $a_{i-1}$ ，即在混合注意力机制中加入了上一时刻的注意力结果  $a_{i-1}$ 。

### 2. 混合注意力机制的具体实现

混合注意力机制的具体实现介绍如下。

(1) 对上一时刻的注意力结果做卷积操作，实现位置特征的提取。

(2) 对卷积操作的结果做全连接处理，实现维度的调整。

(3) 用可选的平滑归一化函数（smoothing normalization function）替换 9.1.10 小节中的 softmax 函数。平滑归一化函数的公式代码如下：

```
def _smoothing_normalization(e):
    return tf.nn.sigmoid(e) / tf.reduce_sum(tf.nn.sigmoid(e)), axis=-1,
keepdims=True)
```

具体代码实例见本书 9.8 节。



#### 提示：

在第（3）步中所提到的 softmax 函数，用在注意力分数 score 的最后处理环节。具体内容见 9.1.10 小节“4. 在 TensorFlow 中的具体实现”里的第（8）步。

### 9.1.13 了解 Seq2Seq 接口中的采样接口 (Helper)

TensorFlow 框架将解码器 (Decoder) 的采样过程抽象出来，单独封装到采样接口的 Helper 类中。基于 Helper 类实现的采样接口又派生了其他的 Helper 子类，如图 9-3 所示。

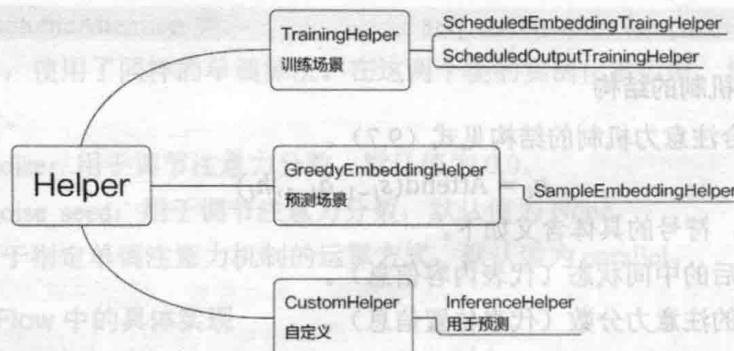


图 9-3 Helper 结构

如图 9-3 所示，每个采样接口类 (Helper) 的解释如下。

- **Helper:** 最基本的抽象类。
- **TrainingHelper:** 用于训练过程中。将上一序列的真实值传入，以计算下一序列的词嵌入分布情况，该结果用于计算 loss 值。
- **ScheduledEmbeddingTrainingHelper:** 用于训练过程中。其继承自 TrainingHelper 类，添加了广义伯努利分布（属于多项式分布，见 9.1.8 小节），对模型的输出结果进行采样。
- **ScheduledOutputTrainingHelper:** 用于训练过程中。其继承自 TrainingHelper 类，直接对输出进行采样。
- **GreedyEmbeddingHelper:** 用在模型使用过程中。从上一序列通过模型后的输出结果中找到概率最大的词，并将其从词嵌入转化成词向量。
- **SampleEmbeddingHelper:** 用于模型使用过程中。其继承自 GreedyEmbeddingHelper 类，将 GreedyEmbeddingHelper 中最大概率的采样规则改成了从生成的概率分布中采样。
- **CustomHelper:** 自定义的采样接口。
- **InferenceHelper:** 一个只用于预测的 helper。其属于 CustomHelper 类的特例，也由用户自定义来生成。

TrainingHelper 类与 GreedyEmbeddingHelper 类的使用实例，请参考本书 9.4 节。  
CustomHelper 类的使用实例，请参考本书 9.8 节。

### 9.1.14 了解 RNN 模型的 Wrapper 接口

TensorFlow 框架用一系列 Wrapper 类将 RNN 模型封装起来，它们形成了 RNN 模型特有的 Wrapper 接口，该接口包括以下几个。

- **InputProjectionWrapper:** 对输入的数据进行维度映射的 Wrapper 类。它对输入数据进行一次全连接转换，再将其输入网络。

- **OutputProjectionWrapper**: 对输出的数据进行维度映射的 `Wrapper` 类。它对网络的输出数据进行一次全连接转换。
- **DropoutWrapper**: 在调用单元的前后进行 dropout 操作，支持对输入层、cell 状态（state）和输出层进行 dropout 处理。
- **ResidualWrapper**: 基于 RNN 模型的残差包装类，相当于把输入用 `concat` 函数连接到输出上一起返回。
- **DeviceWrapper**: 为单元指定运行的设备。
- **MultiRNNCell**: 相当于一个 `wrapper` 类，将单元包装起来，实现多层 RNN 模型。
- **AttentionCellWrapper**: 注意力机制的包装类，参照 9.1.10 小节。

其中，`ResidualWrapper` 类的使用实例见本书 9.8.11 小节。其他 `Wrapper` 类的使用实例见本书 9.4 节。

### 9.1.15 什么是时间序列（TFTS）框架

时间序列（TFTS）框架是一个在估算器上集成好的、专用于序列处理的高级框架。

在使用时，可以直接调用 TFTS 框架中自带的模型（状态空间模型、自回归模型），也可以在 TFTS 框架中构建自定义的 RNN 模型。

该模型支持分块、批处理两种并行的计算方式。更多内容见以下链接：

<https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/timeseries>

在 TFTS 框架中有两个经典的建模工具，具体如下：

- 非线性自回归建模工具（参见源代码文件 `estimators.py` 中的 `ARRegressor` 类）。
- 线性状态空间建模的组件集合建模工具（参见源代码文件 `estimators.py` 中的 `StructuralEnsembleRegressor` 类）。

有关 TFTS 框架的具体实例，见本书 9.7 节。

### 9.1.16 什么是梅尔标度

梅尔标度（the mel scale）是一种符合人耳听觉特性的计算方法。它可以将声音转换为与人耳具有同样感受的数值关系。

例如，人耳对于声音由 1000Hz 变为 2000Hz 的感受并不是音量提高了两倍。而如果将梅尔标度数值提升两倍后，所生成的声音会让人耳感受到两倍的变化。

在本书的 9.8 节中介绍了一个语音合成的例子。其中使用音频数据的梅尔频谱特征作为样本标签，在模型得到梅尔频谱特征之后，再使用梅尔标度的逆向算法将其还原成音频数据。



#### 提示：

在《深度学习之 TensorFlow——入门、原理与进阶实战》一书中 9.5 节的语音识别实例中，用梅尔倒谱对音频进行特征提取。所谓的梅尔倒谱是指，在梅尔标度的频谱上做倒谱分析（取对数，做离散余弦变换）。在语音分析问题中，这样的特征常常用于表述音频数据。

### 9.1.17 什么是短时傅里叶变换

短时傅里叶变换（STFT）是最经典的时频域分析方法。其分为两步：

(1) 对长时信号进行分帧，将其转为短时信号。

(2) 对短时信号做傅里叶变换。

#### 1. 原理

短时傅里叶变换的原理是：将原始声音信号通过短时傅里叶变换展开，使其成为一个二维信号的声谱图。具体步骤如下。

(1) 把一段长时信号分帧、加窗。

(2) 对每一帧做快速傅里叶变换（Fast Fourier Transformation, FFT）。

(3) 把第(2)步的结果堆叠起来，得到二维的信号数据。

图 9-4 所示的是原始的声音信号，图 9-5 所示的是变换后的声谱图。

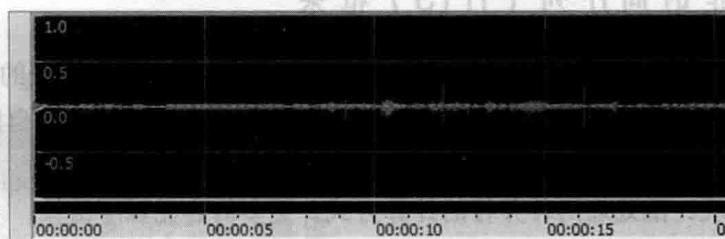


图 9-4 原始声音信号

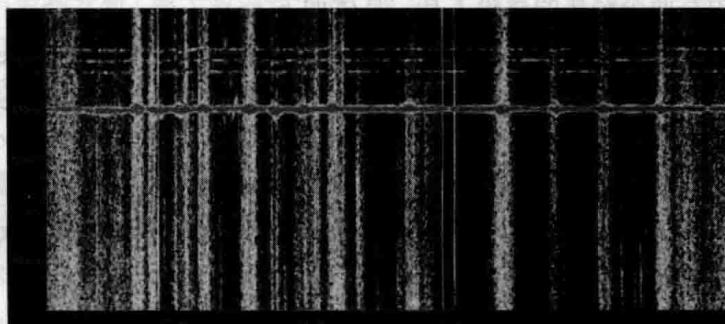


图 9-5 变换后的声谱图

#### 2. 用 librosa 库进行声音处理

librosa 库是一个语音处理的第三方库。用 librosa 库中的 stft 函数可以实现短时傅立叶变换（STFT）。该函数的定义如下：

```
def stft(y, n_fft=2048, hop_length=None, win_length=None, window='hann',
        center=True, dtype=np.complex64, pad_mode='reflect'):
```

其中主要参数有 4 个。

- y：输入的音频序列。
- n\_fft：快速傅里叶变换（FFT）窗口的大小。

- hop\_length：短时傅里叶变换（STFT）算法中的帧移步长。
- win\_length：短时傅里叶变换（STFT）算法中的相邻两个窗口的重叠长度（默认为参数 n\_fft）。

有关 librosa 库的更详细介绍请见 9.8.1 小节。

### 3. 用 TensorFlow 进行声音处理

在 TensorFlow 中，有关声音处理的接口如下。

- 函数 tf.contrib.signal.stft：可以实现短时傅里叶变换（STFT）。
- 函数 tf.contrib.signal.inverse\_stft：可以实现反向的短时傅里叶变换（STFT）。该函数可用于合成声音信号。
- 函数 tf.contrib.ffmpeg.decode\_audio：可以实现读取声音文件。

另外，再介绍一个很有参考价值的代码，链接如下：

[https://github.com/Kyubyong/tensorflow-exercises/blob/master/Audio\\_Processing.ipynb](https://github.com/Kyubyong/tensorflow-exercises/blob/master/Audio_Processing.ipynb)

在该代码文件中，用 TensorFlow 与 librosa 库实现了音频与数字的双向转换：

- (1) 将音频数据转换为梅尔频谱与梅尔倒谱。
- (2) 将梅尔频谱转换回音频数据。

## 9.2 实例 47：搭建 RNN 模型，为女孩生成英文名字

### 实例描述

有一批关于女孩的英文名字列表。让 RNN 模型学习已有的英文名字，并模拟出类似规则的字母序列，为女孩生成英文名字。

在动态图框架中，使用 tf.keras 接口搭建一个由 GRU 单元组成的 RNN 模型。具体做法如下。

### 9.2.1 代码实现：读取及处理样本

样本使用一个文件名为“女孩名字.txt”的数据集。数据集中的每个名字都带有它的寓意解释。例如：

Abby：意为娇小可爱的女人，令人喜爱，个性甜美。

Aimee：意为可爱的人。

Alisa：意为快乐的姑娘。

在随书的配套资源中找到该数据集，并将其放到本地代码的同级目录下。编写代码，实现以下步骤。

- (1) 用正则表达式将每一行的英文字母提取出来。
- (2) 将提取出来的英文字母转成向量。

(3) 对向量进行对齐操作。

具体代码如下。

### 代码 9-1 用 RNN 模型为女孩生成英文名字

```

01 from sklearn.model_selection import train_test_split
02 import numpy as np
03 import os
04 import time
05 from PIL import Image
06 import tensorflow as tf
07
08 import matplotlib.pyplot as plt
09 tf.enable_eager_execution()
10
11 def make_dictionary():
    # 定义函数生成字典
12     words_dic = [chr(i) for i in range(32, 127)]
13     words_dic.insert(0, 'None')                      # 补 0
14     words_dic.append("unknown")
15     words_redic = dict(zip(words_dic, range(len(words_dic))))  # 反向字典
16     print('字表大小:', len(words_dic))
17 return words_dic, words_redic
18 # 字符到向量
19 def ch_to_v(datalist, words_redic, normal = 1):
    # 字典里没有的就是 None
20     to_num = lambda word: words_redic[word] if word in words_redic else
        len(words_redic)-1
21     data_vector = []
22     for ii in datalist:
23         data_vector.append(list(map(to_num, list(ii))))
24
25     if normal == 1:                                # 归一化
26         return np.asarray(data_vector)/ (len(words_redic)/2) - 1
27     return np.array(data_vector)
28
29 # 对数据进行补 0 操作
30 def pad_sequences(sequences, maxlen=None, dtype=np.float32,
31                   padding='post', truncating='post', value=0.):
32
33     lengths = np.asarray([len(s) for s in sequences], dtype=np.int64)
34     nb_samples = len(sequences)
35     if maxlen is None:
36         maxlen = np.max(lengths)
37
38     sample_shape = tuple()
39     for s in sequences:
40         if len(s) > 0:
41             sample_shape = np.asarray(s).shape[1:]  # 宽度未知
42             break

```

```

43     BATCH_SIZE = 6
44     x = (np.ones((nb_samples, maxlen) + sample_shape) * value).astype(dtype)
45     for idx, s in enumerate(sequences):
46         if len(s) == 0:
47             continue #跳过空的列表
48         if truncating == 'pre':
49             trunc = s[-maxlen:]
50         elif truncating == 'post':
51             trunc = s[:maxlen]
52         else:
53             raise ValueError('Truncating type "%s" not understood' % truncating)
54
55     trunc = np.asarray(trunc, dtype=dtype)
56     if trunc.shape[1:] != sample_shape: #检查 trunc 形状
57         raise ValueError('Shape of sample %s of sequence at position %s is different from expected shape %s' % (trunc.shape[1:], idx, sample_shape))
58
59     if padding == 'post':
60         x[idx, :len(trunc)] = trunc
61     elif padding == 'pre':
62         x[idx, -len(trunc):] = trunc
63     else:
64         raise ValueError('Padding type "%s" not understood' % padding)
65     return x, lengths
66
67 def getbatchdata(batchx,charmap): #样本数据预处理（用于训练）
68     batchx = ch_to_v(batchx,charmap,0)
69     sampltpad,sampltlenghts = pad_sequences(batchx) #都填充为最大长度
70     zero = np.zeros([len(batchx),1])
71     tarsentence = np.concatenate((sampltpad[:,1:],zero),axis = 1)
72     return np.asarray(sampltpad,np.int32),np.asarray(tarsentence,np.int32),sampltlenghts
73
74 inv_charmap,charmap = make_dictionary() #生成字典
75 vocab_size = len(charmap)
76
77 DATA_DIR ='./女孩名字.txt' #定义载入样本的路径
78 input_text=[]
79 f = open(DATA_DIR)
80 import re
81 reforname=re.compile(r'[a-z]+', re.I) #用正则化，忽略大小写提取字母
82 for i in f:
83     t = re.match(reforname,i)
84     if t:
85         t=t.group()

```

```

86     input_text.append(t)
87     print(t)

```

在训练模型的过程中，需要将样本中的单个字符依次输入 RNN 模型中。让 RNN 模型根据已输入的字符预测出下一个字符。于是该数据集的标签不再是分类结果，而是样本中当前字符的下一个序列字符。

代码第 67 行定义了函数 `getbatchdata`，用来制作训练模型所使用的输入数据与标签数据。

代码第 71 行是在函数 `getbatchdata` 中制作标签的具体代码。该代码解读如下：

(1) 制作切片，取出输入数据第 1 个字符之后的数据。

(2) 在第(1)步的切片最后添加 0。

通过这两步操作，即可完成输入数据与标签数据的对应关系。例如，针对输入数据“ANNA”所制作的标签为“NNAnone”。

代码第 74 行用 `make_dictionary` 函数生成样本对应的字典。该字典是一个通用的字符字典，里面包含了如下内容：

- 数值在 32~127 的 ASCII 码字符。
- 对齐操作中的补 0 字符——None。
- 向量化操作中的未知字符——`unknown`（见代码第 14 行）。

代码第 77~87 行使用正则表达式提取样本文件中的内容，将每行的英文名字提取出来后放到列表里。其中代码第 81 行的 `re.compile` 函数有两个参数：

- `r[a-z]+'` 代表从头开始匹配属于 a~z 的字符，其中加号代表匹配多个这样的连续字符。
- `re.I` 代表忽略大小写。

用这两个参数来匹配所有以英文开头的字符串。

代码第 83 行是正则表达式的匹配操作。

代码第 85 行是将匹配到的内容取出来。

## 9.2.2 代码实现：构建 Dataset 数据集

下面用函数 `getbatchdata` 获得训练模型所需的样本数据，并将其转换为 `Dataset` 数据集。具体代码如下。

### 代码 9-1 用 RNN 模型为女孩生成英文名字（续）

```

88 input_text,target_text,samplelengths = getbatchdata(input_text,charmap) #
生成样本标签
89 print(input_text)
90 print(target_text)
91
92 max_length = len(input_text[0])
93 learning_rate = 0.001
94
95 embedding_dim = 256          # 定义词向量
96 units = 1024                 # 定义 GRU 单元个数

```

```

97 BATCH_SIZE = 6 # 定义批次
98
99 # 定义数据集
100 dataset = tf.data.Dataset.from_tensor_slices((input_text,
101     target_text)).shuffle(1000)
102 dataset = dataset.batch(BATCH_SIZE, drop_remainder=True)

```

### 9.2.3 代码实现：用 tf.keras 接口构建生成式 RNN 模型

编写代码，构建 RNN 模型，具体步骤如下。

- (1) 将词向量转换成词嵌入。
- (2) 将词嵌入输入 GRU 模型。
- (3) 将 GRU 模型的输出结果输入全连接网络。
- (4) 通过全连接网络，将最终结果收敛到与字典相同维度的特征。

可以将最终模型的输出结果理解为一个多项式分布，即在下一个序列中，每个词可能出现的概率。

模型的反向传播部分如下：

- 损失函数部分使用的是 `sparse_softmax_cross_entropy` 函数。
- 优化器使用的是 `AdamOptimizer` 函数。
- 其他都用默认值。



#### 提示：

`sparse_softmax_cross_entropy` 函数是一个计算交叉熵的函数，更多内容见《深度学习之 TensorFlow——入门、原理与进阶实战》一书的 6.4 节。

整个网络结构都是用 `tf.keras` 接口开发的，具体代码如下：

#### 代码 9-1 用 RNN 模型为女孩生成英文名字（续）

```

102 class Model(tf.keras.Model): # 构建模型
103     def __init__(self, vocab_size, embedding_dim, units, batch_size):
104         super(Model, self).__init__()
105         self.units = units
106         self.batch_sz = batch_size
107         # 定义词嵌入层
108         self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
109
110         if tf.test.is_gpu_available(): # 定义 GRU cell
111             self.gru = tf.keras.layers.CuDNNGRU(self.units,
112                                                 return_sequences=True,
113                                                 return_state=True,
114
115             recurrent_initializer='glorot_uniform')
116         else:

```

```

116     self.gru = tf.keras.layers.GRU(self.units,
117                                     return_sequences=True,
118                                     return_state=True,
119                                     recurrent_activation='sigmoid',
120                                     recurrent_initializer='glorot_uniform')
121
122     self.fc = tf.keras.layers.Dense(vocab_size) # 定义全连接层
123
124 def call(self, x, hidden):
125     x = self.embedding(x)
126
127     # 用 GRU 网络进行计算, output 的形状为 (batch_size, max_length, hidden_size)
128     # states 的形状为 (batch_size, hidden_size)
129     output, states = self.gru(x, initial_state=hidden)
130
131     # 变换维度, 用于后面的全连接, 输出形状为 (batch_size * max_length, hidden_size)
132     output = tf.reshape(output, (-1, output.shape[2]))
133
134     # 得到每个词的多项式分布
135     # 输出形状为 (max_length * batch_size, vocab_size)
136     x = self.fc(output)
137
138     return x, states
139 model = Model(vocab_size, embedding_dim, units, BATCH_SIZE)
140 optimizer = tf.train.AdamOptimizer()
141
142 # 损失函数
143 def loss_function(real, preds):
144     return tf.losses.sparse_softmax_cross_entropy(labels=real,
145                                                    logits=preds)

```

## 9.2.4 代码实现：在动态图中训练模型

编写代码，实现如下步骤：

- (1) 指定检查点文件的路径并建立循环。
- (2) 按照指定迭代次数 EPOCHS 进行迭代训练。
- (3) 在每次迭代训练中，用动态图的训练方式对模型权重进行优化。

具体代码如下。

### 代码 9-1 用 RNN 模型为女孩生成英文名字（续）

```

145 checkpoint_dir = './training_checkpoints'
146 checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt") # 定义检查点文件的路径
147 # 定义检查点文件
148 checkpoint = tf.train.Checkpoint(optimizer=optimizer, model=model)
149 latest_cpkt = tf.train.latest_checkpoint(checkpoint_dir)
150 if latest_cpkt:

```

```

151   print('Using latest checkpoint at ' + latest_cpkt)
152   checkpoint.restore(latest_cpkt)
153 else:
154     os.makedirs(checkpoint_dir, exist_ok=True)      #建立存放模型的文件夹
155
156 EPOCHS = 20                                     #定义迭代次数
157
158 for epoch in range(EPOCHS):                     #开始训练
159   start = time.time()
160
161   hidden = model.reset_states()                  #初始化 RNN 模型
162   totalloss = []
163   for (batch, (inp, target)) in enumerate(dataset):
164     hidden = model.reset_states()                #对于每个样本都需要重新初始化
165     with tf.GradientTape() as tape:              #应用梯度训练模型
166       predictions, hidden = model(inp, hidden)
167       target = tf.reshape(target, (-1,))
168       loss = loss_function(target, predictions)
169       totalloss.append(loss)                    #统计损失值
170   grads = tape.gradient(loss, model.variables)
171   optimizer.apply_gradients(zip(grads, model.variables))
172
173   if batch % 100 == 0:                          #显示结果
174     print ('Epoch {} Batch {} Loss {:.4f}'.format(epoch+1, batch,
175           loss))
176   #每迭代 5 次保存 1 次检查点
177   if (epoch + 1) % 5 == 0:
178     checkpoint.save(file_prefix = checkpoint_prefix)
179
180   print ('Epoch {} Loss {:.4f}'.format(epoch+1, np.mean(totalloss)))
181   print('Time taken for 1 epoch {} sec\n'.format(time.time() - start))

```

因为在样本中每个名字都是独立的，所以在对每个样本处理之前都需要对模型重新初始化（见代码第 164 行），让本次处理不受上一个样本信息的影响。

代码第 165~171 行是动态图中的反向传播实现，这部分内容可以参考本书 6.7 节。

## 9.2.5 代码实现：载入检查点文件并用模型生成名字

在使用模型时，需要对 RNN 模型的输出结果进行多项式采样，并将采样后的结果当作真正的目标结果。具体的实现步骤如下。

- (1) 用模型生成 20 个英文名字（见代码第 184 行）。
- (2) 在每次生成英文名字时，都会从样本集里面随机选出一个名字的首字符，作为本次的首字母（见代码第 185 行），并将首字符作为模型的输入。
- (3) 将输入的字符送入模型中进行预测，对模型的预测结果用基于多项式分布的采样操作

(从候选词分布中获得具体字母)，得到预测序列中的下一个字母（见代码第 196 行）。

(4) 将第(3)步的输出结果作为输入，再次送入第(3)步，继续预测下一个字符。

(5) 按照第(3)~(4)步骤进行循环，直到模型输出的字母向量为 0(0 表示生成结束)，见代码第 195 行。

(6) 如果一直没有 0 值，则第(5)步的循环会在执行 max\_length 次时结束，见代码第 192 行。

(7) 将第(2)步生成的首字符与第(3)步每次输出的字符连接，形成最终的结果。具体代码如下。

### 代码 9-1 用 RNN 模型为女孩生成英文名字（续）

```

182 checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir)) #载入模型
183
184 for iii in range(20):
185     input_eval = input_text[np.random.randint(len(input_text))][0] #获得一个随机数做开始
186     start_string = inv_charmap[input_eval]
187     input_eval = tf.expand_dims([input_eval], 0) #将其转成向量
188
189     text_generated = '' #定义空串，用于存放结果
190
191     hidden = model.reset_states() #初始化模型
192     for i in range(max_length):
193         predictions, hidden = model(input_eval, hidden) #输出模型结果
194         predicted_id = tf.multinomial(predictions,
195             num_samples=1)[0][0].numpy() #采样
196         if predicted_id==0: #出现 0 时表示结束
197             break
198         input_eval = tf.expand_dims([predicted_id], 0)
199         text_generated += inv_charmap[predicted_id] #保存单次结果
200
201     print (start_string + text_generated) #输出结果

```

代码运行之后，输出如下结果：

```

Epoch 14 Batch 0 Loss 0.2410
.....
Epoch 20 Batch 0 Loss 0.2626
Epoch 20 Loss 0.2943
Time taken for 1 epoch 2.3288302421569824 sec

```

```

Alisa Lena Moon Sellew Daisy Kotty Andrea Gloria Ann Amhndra Gladys Sweety Alisa Camille
Irene Angelina Alice Carol Eudora Dema

```

结果中的最后两行即为生成的名字。可以看到，这些名字与我们常见的英文名字很像（有的几乎是一样的），符合英文命名习惯。

## 9.2.6 扩展：用 RNN 模型编写文章

在《深度学习之 TensorFlow——入门、原理与进阶实战》中的 9.6 节有一个用 RNN 模型生成文章的实例，它与本实例非常相似，唯独不同的是它没有使用多项式分布进行采样。读者可以将多项式采样技术运用到那个例子中并观察效果。

## 9.3 实例 48：用带注意力机制的 Seq2Seq 模型为图片添加内容描述

在动态图上用 tf.keras 接口搭建带注意力机制的 Seq2Seq 模型，并用该模型为图片添加内容描述。

### 实例描述

用 COCO 数据集训练一个带有注意力机制的 Seq2Seq 模型，使模型能够识别图片内容，并根据内容生成描述。

本实例使用 COCO 数据集的文本描述标注内容来训练模型，即输入是具体的一张图片，输出是一段文字描述。



**提示：**

在 COCO 数据集中，关于文本描述标注内容见 8.7.2 小节“4. 加载并显示文本描述标注信息”。

### 9.3.1 设计基于图片的 Seq2Seq

本实例属于跨域任务，实现图片与文本之间的转换。在实现时，将 Seq2Seq 框架中编码器（Encoder）的输入部分改成能够提取图片特征的网络结构，使其支持对图片的处理，具体结构如图 9-6 所示。

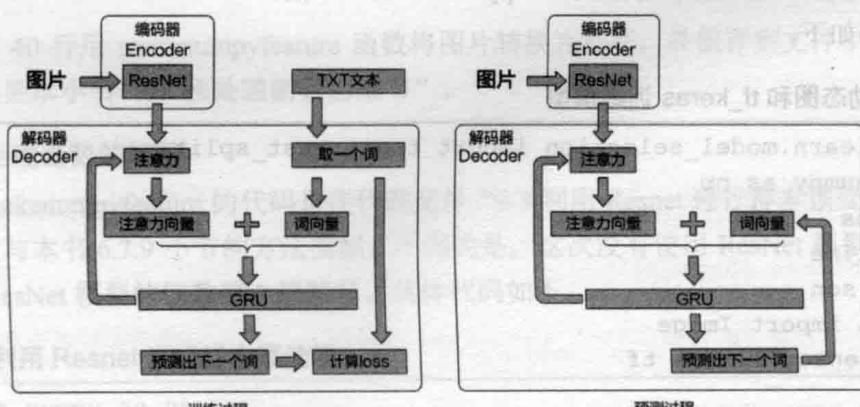


图 9-6 基于图片处理的 Seq2Seq 模型的结构

Seq2Seq 模型常用于处理纯文本类任务，其内部的数据转换可以理解为两步：

- (1) 将文本转换成特征。
- (2) 将特征转换成文本。

在图 9-6 所示的模型中，其内部的数据转换也可以理解为两步：

- (1) 将图片转换成特征。
- (2) 将特征转换成文本。

图 9-6 所示的模型将 Seq2Seq 模型的第(1)步(处理文本)换成了处理图片。使模型同样可以通过特征来与第(2)步进行对接。这种设计思想来源于神经网络模型的本质要素——特征。

在算法模型中，神经网络模型关注的是数据的特征，而某个特征是来自于文本序列还是来自于图片，并不是神经网络模型所关心的事情。

在设计模型时，建议读者要有特征的概念，不要将某个网络模型局限于处理某一方向的问题上。例如，本书 8.3 节介绍的那个实例，其中将擅长处理图片样本的卷积网络模型用在文本数据的分类任务之上，这也是考虑从特征角度来设计模型结构的。

### 9.3.2 代码实现：图片预处理——用 ResNet 提取图片特征并保存

在训练模型时，模型每次的迭代处理都需要先将图片转成特征向量再进行计算。这使得程序做了大量的重复工作。

可以在预处理环节中，提前将图片转换成特征向量并保存起来。这样，在迭代训练时模型直接读取转换后的特征向量即可，省去了将图片转换为特征向量的重复工作。

#### 1. 对样本进行预处理

由于 COCO 数据集 train2014 中的图片文件过多（见 8.7.2 小节“4. 加载并显示文本描述标注信息”中的数据集及标注），所以这里只取 300 张图片作为训练集来演示。

具体要实现如下步骤：

- (1) 将所有的文本标注读到内存中。
- (2) 将图片数据转换为特征数据。
- (3) 将转换后的特征数据存放到 numpyfeature 目录下面。

具体代码如下。

#### 代码 9-2 用动态图和 tf\_keras 训练模型

```

01 from sklearn.model_selection import train_test_split
02 import numpy as np
03 import os
04 import time
05 import json
06 from PIL import Image
07 import tensorflow as tf
08
09 import matplotlib.pyplot as plt
10 import tensorflow.contrib.eager as tfe

```

```

11 preimgdata = __import__("9-3 利用 Resnet 进行样本预处理")
12 makenumpyfeature = preimgdata.makenumpyfeature
13 # 在 Resnet 中调用
14 tf.enable_eager_execution() # 启动动态图
15 print("TensorFlow 版本: {}".format(tf.VERSION))
16 print("Eager execution: {}".format(tf.executing_eagerly()))
17 # 载入标注文件
18 annotation_file = r'cocos2014/annotations/captions_train2014.json'
19 PATH = r"cocos2014/train2014/"
20 numpyPATH = './numpyfeature/'
21
22 with open(annotation_file, 'r') as f:      # 读取标注文件
23     annotations = json.load(f)
24
25 num_examples = 300 # 加载指定个数的图片路径和对应的标题
26 train_captions = [] # 定义列表，用于保存所有的训练文本
27 img_filename = [] # 定义列表，用于保存所有的图片文件路径
28
29 for annot in annotations['annotations']: # 获取全部的文件名及对应的标注文本
30     caption = '<start> ' + annot['caption'] + ' <end>'
31     image_id = annot['image_id']
32     full_coco_image_path = 'COCO_train2014_{}%012d.jpg'.format(image_id)
33     img_filename.append(full_coco_image_path)
34     train_captions.append(caption)
35     if len(train_captions) >= num_examples:
36         break
37
38 # 如果本地没有生成特征文件，则进行数据预处理
39 if not os.path.exists(numpyPATH):
40     makenumpyfeature(numpyPATH, img_filename, PATH) # 生成特征文件，并保存

```

代码第 30 行将<start>与<end>标签分别添加到每行文本的开头和结尾处，旨在标志出每行文本的开始位置和结束位置。在句子中，标出开始位置和结束位置的方法是 Seq2Seq 接口的标准用法。

代码第 40 行用 makenumpyfeature 函数将图片转换为特征，并保存到文件中。该函数的具体实现过程见本小节“2. 预处理函数的细节”。

## 2. 预处理函数的细节

函数 makenumpyfeature 的代码是在代码文件“9-3 利用 Resnet 进行样本预处理.py”中实现的。该函数与本书 6.7.9 小节的方法类似。不同的是，这次没有使用 ResNet 模型的输出结果，而是提取 ResNet 模型的倒数第 2 层特征。具体代码如下。

### 代码 9-3 利用 Resnet 进行样本预处理

```

01 import numpy as np
02 import os
03 import shutil

```

```

04 import tensorflow as tf
05 from tensorflow.python.keras.applications.resnet50 import ResNet50
06
07 def makenumpyfeature(numpyPATH, img_filename, PATH):
08     if os.path.exists(numpyPATH):
09         shutil.rmtree(numpyPATH, ignore_errors=True) #去除已有的文件目录
10
11     os.mkdir(numpyPATH) #新建文件目录
12
13     size = [224, 224] #设置图片输出尺寸
14     batchsize = 10
15
16     def load_image(image_path): #输入图片的预处理
17         img = tf.read_file(PATH + image_path)
18         img = tf.image.decode_jpeg(img, channels=3)
19         img = tf.image.resize(img, size)
20         img = tf.keras.applications.resnet50.preprocess_input(img) #ResNet
    的统一预处理
21
22         return img, image_path
23
24     image_model =
25     ResNet50(weights='resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5',
26               include_top=False) #创建ResNet
27
28     new_input = image_model.input #定义输入节点
29     hidden_layer = image_model.layers[-2].output #获取ResNet的倒数第2层
30
31     image_features_extract_model = tf.keras.Model(new_input, hidden_layer) #图片数据集
32
33     encode_train = sorted(set(img_filename)) #对输入文件目录去重
34
35     image_dataset = tf.data.Dataset.from_tensor_slices(
36         encode_train).map(load_image).batch(batchsize) #按照批次进行转换
37
38     batch_features = image_features_extract_model(image_dataset) #输出形状(batch, 7,
39     2048) #输出形状(batch, 49, 2048)
40
41     batch_features = tf.reshape(batch_features, #输出形状(batch, 49, 2048)
42                                 (batch_features.shape[0], -1,
43                                batch_features.shape[3])) #将特征结果保存到文件中
44     for bf, p in zip(batch_features, path):
45         path_of_feature = p.numpy().decode("utf-8")
46         np.save(numpyPATH+path_of_feature, bf.numpy())

```

### 9.3 代码第27行将ResNet模型的倒数第2层当作输出节点。

#### 3. 在ResNet模型中找到输出节点

如果想要从ResNet模型中提取特征，则需要先了解ResNet模型的代码实现。

以作者的本地路径为例，在TensorFlow中，ResNet模型源代码文件的路径如下：

```
"C:\local\Anaconda3\lib\site-packages\tensorflow\python\keras\applications\resnet50.py"
```

在该源代码文件的第263行，可以找到ResNet模型的定义。具体代码如下。

#### 代码resnet50

```
263     x = identity_block(x, 3, [512, 512, 2048], stage=5, block='c')#最终的卷积结  
果  
264     x = AveragePooling2D((7, 7), name='avg_pool')(x)           #全局平均池化层  
265     if include_top:                                              #返回指定的顶层输出  
266         x = Flatten()(x)  
267         x = Dense(classes, activation='softmax', name='fc1000')(x)  
268     else:  
269         if pooling == 'avg':  
270             x = GlobalAveragePooling2D()(x)  
271         elif pooling == 'max':  
272             x = GlobalMaxPooling2D()(x)
```

可以看到，在代码第265行中返回了指定的顶层输出。

在第265行之前是全局平均池化层。在全局平均池化层之前（上一层）的内容（见代码第263行）便是需要提取的部分。

因为载入模型时使用的是去掉顶层的ResNet模型（见代码文件“9-3 利用Resnet进行样本预处理.py”代码第23行，`include_top=False`），即最后一层为平均池化层，所以这里取了输出节点的倒数第2层（见代码文件“9-3 利用Resnet进行样本预处理.py”第27行）。

另外，该代码还需要加载ResNet模型的预训练模型（在ImgNet数据集上训练好的权重文件）“`resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5`”。可以参考本书的6.7.9小节，将该预训练模型下载后直接放到本地路径下。

#### 4. 运行程序进行预处理

代码运行后，会在本地路径的`numpyfeature`文件夹下生成多个以“.np”结尾的文件。这些文件里面放置了形状为(49, 2048)的特征数据。

### 9.3.3 代码实现：文本预处理——过滤处理、字典建立、对齐与量化处理

在对COCO数据集中的图片预处理之后，还需要对每个图片的标注文本做预处理，具体步骤如下。

- (1) 过滤文本：去除无效符号。
- (2) 建立字典：生成正反向字典。
- (3) 向量化文本与对齐操作：将文本按照字典中的数字进行向量化处理，并按照指定长度进行对齐操作（多余的截掉，不足的补 0）。

最终将图片预处理的结果与文本预处理的结果结合，并按照一定比例拆分成训练集与评估数据集。

具体代码如下。

### 代码 9-2 用动态图和 tf\_keras 训练模型（续）

```

41 top_k = 5000                                #设置字典最大长度为 5000
42 tokenizer = tf.keras.preprocessing.text.Tokenizer(num_words=top_k,
43                                         oov_token=<unk>,
44                                         filters='!"#$%&()*+.,-/:;=?@[\]^_`{|}~ ')
45 tokenizer.fit_on_texts(train_captions)  #过滤处理
46
47 #建立字典
48 tokenizer.word_index = {key:value for key, value in
tokenizer.word_index.items() if value <= top_k}
49 tokenizer.word_index[tokenizer.oov_token] = top_k + 1#向字典中添加符号<unk>,
    用于处理未知单词
50 tokenizer.word_index['<pad>'] = 0
51 print(tokenizer.word_index)
52
53 index_word = {value:key for key, value in tokenizer.word_index.items()} #反向字典
54 train_seqs = tokenizer.texts_to_sequences(train_captions) #变为向量
55
56 #按照最长的句子对齐，不足的在其后面补 0
57 cap_vector = tf.keras.preprocessing.sequence.pad_sequences(train_seqs,
padding='post')
58 print("最大长度",len(cap_vector[0]))
59 max_length =len(cap_vector[0])
60
61 #将数据拆成训练集和测试集
62 img_name_train, img_name_val, cap_train, cap_val =
train_test_split(img_filename,
63                                         cap_vector,
64                                         test_size=0.2,
65                                         random_state=0)
66
67 np.save(numpy_path+'train_features', b)
68 np.save(numpy_path+'val_features', b)
69 np.save(numpy_path+'train_labels', b)
70 np.save(numpy_path+'val_labels', b)

```

### 9.3.4 代码实现：创建数据集

读入特征数据，用 `tf.data.Dataset` 接口将特征文件与文本向量组合到一起，生成数据集，为训练模型做准备。具体步骤如下。

代码 9-2 用动态图和 `tf_keras` 训练模型（续）

```

66 BATCH_SIZE = 20
67 embedding_dim = 256
68 units = 512
69 vocab_size = len(tokenizer.word_index)
70
71 #图片特征(47, 2048)
72 features_shape = 2048
73 attention_features_shape = 49
74
75 #加载 numpy 文件
76 def map_func(img_name, cap):
77     img_tensor = np.load(numpyPATH+img_name.decode('utf-8')+'.npy')
78     return img_tensor, cap
79
80 dataset = tf.data.Dataset.from_tensor_slices((img_name_train, cap_train))
81
82 #用 map 加载 numpy 特征文件
83 dataset = dataset.map(lambda item1, item2: tf.py_function(
84     map_func, [item1, item2], [tf.float32, tf.int32]),
85     num_parallel_calls=8)
86 dataset = dataset.shuffle(1000).batch(BATCH_SIZE).prefetch(1)

```

### 9.3.5 代码实现：用 `tf.keras` 接口构建 Seq2Seq 模型中的编码器

编码器模型比解码器模型简单，只有一个全连接网络。该全连接网络对原始图片的特征数据进行转换处理，使原始图片特征数据的维度与词嵌入的维度相同。具体代码如下。

代码 9-2 用动态图和 `tf_keras` 训练模型（续）

```

87 class DNN_Encoder(tf.keras.Model):#编码器模型
88     def __init__(self, embedding_dim):
89         super(DNN_Encoder, self).__init__()
90         #tf.keras 的全连接支持多维输入。仅对最后一维进行处理
91         self.fc = tf.keras.layers.Dense(embedding_dim)
92
93     def call(self, x):#最终输出特征的形状为(batch_size, 49, embedding_dim)
94         x = self.fc(x)
95         x = tf.nn.relu(x)
96         return x

```

在 tf.keras 接口中，全连接网络的输入既可以是二维数据，也可以是多维数据。如果输入的是多维数据，则按照最后一维进行全连接变换。在代码第 93 行的 call 方法中，DNN\_Encoder 模型最终会输出一个形状为(batch\_size, 49, embedding\_dim)的数据。

### 9.3.6 代码实现：用 tf.keras 接口构建 Bahdanau 类型的注意力机制

定义一个 BahdanauAttention 类，构建 Bahdanau 类型的注意力机制。具体代码如下。

代码 9-2 用动态图和 tf\_keras 训练模型（续）

```

97 class BahdanauAttention(tf.keras.Model):
98     def __init__(self, units):
99         super(BahdanauAttention, self).__init__()
100        self.W1 = tf.keras.layers.Dense(units)
101        self.W2 = tf.keras.layers.Dense(units)
102        self.V = tf.keras.layers.Dense(1)
103
104    def call(self, features,   #features 形状(batch_size, 49, embedding_dim)
105            hidden):          #hidden(batch_size, hidden_size)
106
107        hidden_with_time_axis = tf.expand_dims(hidden, 1)  #(batch_size, 1,
108        hidden_size)
109        #score 形状: (batch_size, 49, hidden_size)
110        score = tf.nn.tanh(self.W1(features) + self.W2(hidden_with_time_axis))
111
112        attention_weights = tf.nn.softmax(self.V(score), axis=1)
113        #(batch_size, 49, 1)
114
115        context_vector = attention_weights * features      #(batch_size, 49,
116        hidden_size)
117        context_vector = tf.reduce_sum(context_vector, axis=1) #(batch_size,
118        hidden_size)
119
120    return context_vector, attention_weights

```

### 9.3.7 代码实现：搭建 Seq2Seq 模型中的解码器 Decoder

定义类 RNN\_Decoder，构建 Seq2Seq 模型中的解码器。具体步骤如下。

- (1) 用注意力机制对编码器的特征进行处理。
- (2) 用 GRU 单元构建循环神经网络模型，进行解码工作。
- (3) 用两层全连接网络得出最终结果。

具体代码如下。

## 代码 9-2 用动态图和 tf\_keras 训练模型（续）

```

117 def gru(units):
118     if tf.test.is_gpu_available():
119         return tf.keras.layers.CuDNNGRU(units,
120                                         return_sequences=True,
121                                         return_state=True,
122                                         recurrent_initializer='glorot_uniform')
123     else:
124         return tf.keras.layers.GRU(units,
125                                     return_sequences=True,
126                                     return_state=True,
127                                     recurrent_activation='sigmoid',
128                                     recurrent_initializer='glorot_uniform')
129
130 class RNN_Decoder(tf.keras.Model):
131     def __init__(self, embedding_dim, units, vocab_size):
132         super(RNN_Decoder, self).__init__()
133         self.units = units
134
135         self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
136         self.gru = gru(self.units)
137         self.fc1 = tf.keras.layers.Dense(self.units)
138         self.fc2 = tf.keras.layers.Dense(vocab_size)
139
140         self.attention = BahdanauAttention(self.units)
141
142     def call(self, x, features, hidden):
143         # 返回注意力特征向量和注意力权重
144         context_vector, attention_weights = self.attention(features, hidden)
145
146         x = self.embedding(x) # 形状为 (batch_size, 1, embedding_dim)
147
148         x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1) # 形状为 (batch_size, 1, embedding_dim + hidden_size)
149
150         output, state = self.gru(x) # 用循环网络进行处理
151         # 全连接处理，形状为 (batch_size, max_length, hidden_size)
152         x = self.fc1(output)
153         # 将形状变化为 (batch_size * max_length, hidden_size)
154         x = tf.reshape(x, (-1, x.shape[2]))
155         # 第 2 层全连接得出最终结果，形状为 (batch_size * max_length, vocab)
156         x = self.fc2(x)
157
158         return x, state, attention_weights
159
160     def reset_state(self, batch_size):
161         return tf.zeros((batch_size, self.units))

```

### 9.3.8 代码实现：在动态图中计算 Seq2Seq 模型的梯度

在本书 6.3 节中，介绍过两种在动态图中计算梯度的方法。

- 用 `tfe.implicit_gradients` 函数计算梯度。
- 用 `tf.GradientTape` 函数计算梯度。

二者具有同样的效果。在本实例中是用 `tfe.implicit_gradients` 函数来计算梯度，具体代码如下。

#### 代码 9-2 用动态图和 `tf_keras` 训练模型（续）

```

162 def loss_function(real, pred):      #单个 loss 值的处理函数
163     mask = 1 - np.equal(real, 0)      #批次中被补 0 的序列不参与计算 loss 值
164     loss_ = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=real,
165     logits=pred) * mask
166
167 def all_loss(encoder, decoder, img_tensor, target):#定义函数,处理全部的 loss 值
168     loss = 0
169     hidden = decoder.reset_state(batch_size=target.shape[0])
170
171     dec_input = tf.expand_dims([tokenizer.word_index['<start>']] *
172         BATCH_SIZE, 1)
173     features = encoder(img_tensor) # (20, 49, 256)
174     for i in range(1, target.shape[1]):#通过 Decoder 网络生成预测结果
175         predictions, hidden, _ = decoder(dec_input, features, hidden)
176         loss += loss_function(target[:, i], predictions) #计算本次预测的 loss 值
177
178     #获得本次标签,用于下次序列的预测使用
179     dec_input = tf.expand_dims(target[:, i], 1)
180
181     return loss
182 grad = tfe.implicit_gradients(all_loss) #根据 all_loss 函数生成梯度

```

### 9.3.9 代码实现：在动态图中为 Seq2Seq 模型添加保存检查点功能

用 `tf.train.Checkpoint` 函数为模型添加保存检查点功能，具体代码如下。

#### 代码 9-2 用动态图和 `tf_keras` 训练模型（续）

```

183 model_objects = {
184     'encoder':DNN_Encoder(embedding_dim),
185     'decoder' :RNN_Decoder(embedding_dim, units, vocab_size),
186     'optimizer': tf.train.AdamOptimizer(),
187     'step_counter': tf.train.get_or_create_global_step(),
188 }
189

```

```

190 checkpoint_prefix = os.path.join("mytfmodel/", 'ckpt') 从这里开始到行 241 都是
191 checkpoint = tf.train.Checkpoint(**model_objects) 从这里开始到行 241 都是
192 latest_cpkt = tf.train.latest_checkpoint("mytfmodel/") 从这里开始到行 241 都是 #查找检查点文件
193 if latest_cpkt: 从这里开始到行 241 都是
194     print('Using latest checkpoint at ' + latest_cpkt) 从这里开始到行 241 都是
195     checkpoint.restore(latest_cpkt) 从这里开始到行 241 都是 #恢复权重
241

```

### 9.3.10 代码实现：在动态图中训练 Seq2Seq 模型

在动态图中训练 Seq2Seq 模型的步骤如下。

- (1) 定义单步训练函数 train\_one\_epoch。
  - (2) 用 for 循环按照指定迭代次数调用函数 train\_one\_epoch。
  - (3) 将在训练过程中用函数 train\_one\_epoch 返回的损失值 loss 数据保存起来，并将其输出。
- 具体代码如下。

代码 9-2 用动态图和 tf\_keras 训练模型（续）

```

196 #实现单步训练过程
197 def
198     train_one_epoch(encoder,decoder,optimizer,step_counter,dataset,epoch):
199         total_loss = 0
200         for (step, (img_tensor, target)) in enumerate(dataset):
201             loss = 0
202             #应用梯度
203             optimizer.apply_gradients(grad(encoder,decoder,img_tensor,
204             target),step_counter)
205             loss =all_loss(encoder,decoder,img_tensor, target)
206             total_loss += (loss / int(target.shape[1]))
207             if step % 5 == 0:
208                 print ('Epoch {} Batch {} Loss {:.4f}'.format(epoch + 1,
209                     step,
210                     loss.numpy() /
211                     int(target.shape[1])))
210             print("step",step)
211             return total_loss/(step+1)
212
213 loss_plot = []
214 EPOCHS = 50 #定义迭代次数
215
216 for epoch in range(EPOCHS): #训练模型
217     start = time.time()
218     total_loss=
219     train_one_epoch(dataset=dataset,epoch=epoch,**model_objects) #训练一次
220     loss_plot.append(total_loss) #保存 loss 值
221

```

```

222     print ('Epoch {} Loss {:.6f}'.format(epoch + 1, total_loss))
223     checkpoint.save(checkpoint_prefix)
224     print('Train time for epoch #{} (step {}): {}'.format(
225         (checkpoint.save_counter.numpy(), checkpoint.step_counter.numpy()),
226         time.time() - start))
226 plt.plot(loss_plot)
227 plt.xlabel('Epochs')
228 plt.ylabel('Loss')
229 plt.title('Loss Plot')
230 plt.show()

```

代码运行后，输出如下内容：

```

.....
Epoch 49 Loss 0.160428
Train time for epoch #109 (step 658): 8.619966
Epoch 50 Batch 0 Loss 0.1336
Epoch 50 Loss 0.170198
Train time for epoch #110 (step 670): 8.554722

```

显示的损失值 (loss) 结果如图 9-7 所示。

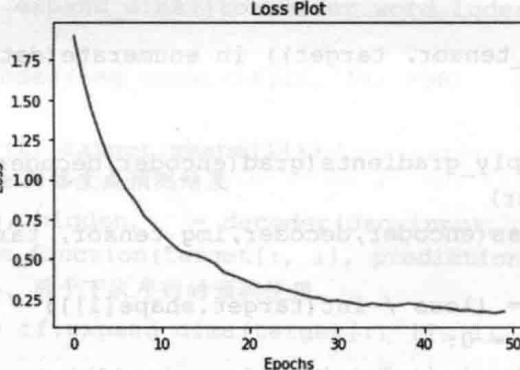


图 9-7 基于图片处理的 Seq2Seq 模型结构

### 9.3.11 代码实现：用多项式分布采样获取图片的内容描述

下面编写代码，实现两个函数。

- `evaluate` 函数用于为指定的图片生成内容描述，将整个网络连接起来。

- `plot_attention` 函数用于将模型中的注意力分值以图示化的方式显示出来。

具体代码如下。

#### 代码 9-2 用动态图和 tf\_keras 训练模型（续）

```

231 def evaluate(encoder, decoder, optimizer, step_counter, image):
232     attention_plot = np.zeros((max_length, attention_features_shape))
233
234     hidden = decoder.reset_state(batch_size=1)

```

```

235     size = [224, 224]
236     def load_image(image_path):
237         img = tf.read_file(PATH + image_path)
238         img = tf.image.decode_jpeg(img, channels=3)
239         img = tf.image.resize(img, size)
240         img = tf.keras.applications.resnet50.preprocess_input(img)
241         return img, image_path
242     from tensorflow.python.keras.applications import ResNet50
243
244     image_model =
245         ResNet50(weights='resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5',
246                  include_top=False) # 创建 ResNet
247
248     new_input = image_model.input
249     hidden_layer = image_model.layers[-2].output
250     image_features_extract_model = tf.keras.Model(new_input, hidden_layer)
251
252     temp_input = tf.expand_dims(load_image(image)[0], 0)
253     img_tensor_val = image_features_extract_model(temp_input) # 用 ResNet 生成
特征
254     img_tensor_val = tf.reshape(img_tensor_val, (img_tensor_val.shape[0],
255 -1, img_tensor_val.shape[3]))
256
257     features = encoder(img_tensor_val) # 用编码器对特征进行转换
258
259     dec_input = tf.expand_dims([tokenizer.word_index['<start>']], 0)
260     result = []
261
262     for i in range(max_length): # 用循环进行 Seq2Seq 框架的解码工作
263         predictions, hidden, attention_weights = decoder(dec_input, features,
hidden)
264         attention_plot[i] = tf.reshape(attention_weights, (-1, )).numpy()
265         # 将预测结果转化为词向量
266         predicted_id = tf.multinomial(predictions,
num_samples=1)[0][0].numpy()
267         result.append(index_word[predicted_id]) # 保存单次的预测结果
268         if index_word[predicted_id] == '<end>': # 如果出现结束标志，则停止循环
269             return result, attention_plot
270         dec_input = tf.expand_dims([predicted_id], 0) # 维度变化用于下一次的输入
271
272     attention_plot = attention_plot[:len(result), :]
273     return result, attention_plot
274
275     def plot_attention(image, result, attention_plot): # 图示化模型的注意力分值
276         temp_image = np.array(Image.open(PATH + image))
277
278         fig = plt.figure(figsize=(10, 10))

```

```

277 len_result = len(result)
278 for l in range(len_result):
279     temp_att = np.resize(attention_plot[l], (7, 7))
280     ax = fig.add_subplot(len_result//2, len_result//2+len_result%2, l+1)
281     ax.set_title(result[l])
282     img = ax.imshow(temp_image)
283     ax.imshow(temp_att, cmap='gray', alpha=0.4)
284     extent=img.get_extent()
285 plt.tight_layout()
286 plt.show()
287
288 rid = np.random.randint(0, len(img_name_val)) #随机选取一张图片
289 image = img_name_val[rid]
290 real_caption = ' '.join([index_word[i] for i in cap_val[rid] if i not in [0]])
291 result, attention_plot = evaluate(image=image, **model_objects) #生成结果
292 print ('Real Caption:', real_caption)
293 print ('Prediction Caption:', ' '.join(result)) #输出预测值与真实值的描述结果
294 plot_attention(image, result, attention_plot) #将注意力分值结果显示出来
295
296 img = Image.open(PATH +img_name_val[rid]) #打开输入文件并显示
297 plt.imshow(img)
298 plt.axis('off')
299 plt.show()

```

在函数 evaluate 中，实现以下步骤：

- (1) 构建 ResNet50 网络，作为图片的特征提取层。
- (2) 将提取图片的特征数据输入编码器模型中（见代码第 255 行）。
- (3) 使用循环进行 Seq2Seq 框架的解码工作，依次生成输出的文本内容（见代码第 260 行）。
- (4) 利用 Seq2Seq 框架的结构，将编码器模型的结果与中间的状态值一起放入解码器模型中进行解码（见代码第 261 行）。
- (5) 用多项式分布采样的方式从解码器模型的结果中取出当前序列的文本向量（见代码第 264 行）。
- (6) 按照步骤 (3) 所指定的循环次数，重复步骤 (4) 和 (5)，直到生成全部的文本。

代码运行后输出结果如下：

```

Real Caption: <start> a clean organized kitchen cabinet and countertop area <end>
Prediction Caption: a kitchen has red bricks lining the counter <end>

```

在输出结果中，第 1 行是数据集中的标注文本。第 2 行是模型生成的预测文本。图示化的结果如图 9-8 和图 9-9 所示。



图 9-8 原始图片

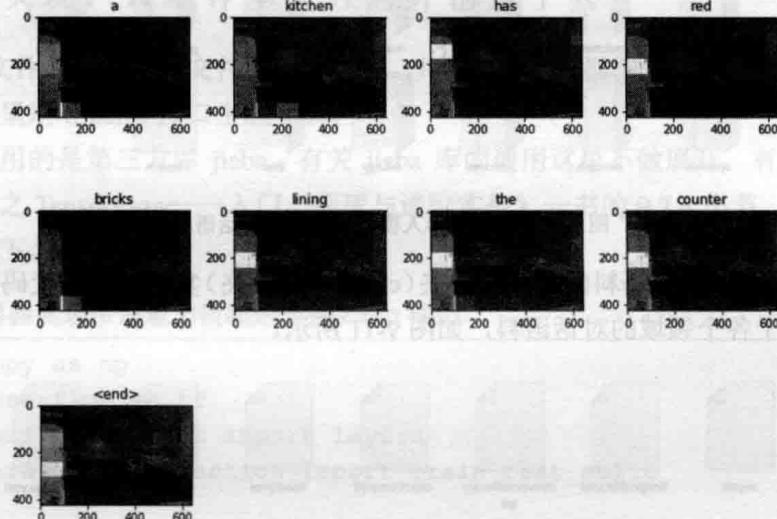


图 9-9 注意力结果的图示化显示

## 9.4 实例 49：用 IndRNN 与 IndyLSTM 单元制作聊天机器人

在《深度学习之 TensorFlow——入门、原理与进阶实战》一书的 9.8 节介绍了一个聊天机器人模型。那个模型是用 `tf.contrib.legacy_seq2seq` 接口实现的。`tf.contrib.legacy_seq2seq` 接口是一个比较老的接口。

TensorFlow 1.0 之后的版本对 Seq2Seq 框架进行了重新封装，推出新的 Seq2Seq 框架 API。本节使用新的 Seq2Seq 框架 API 实现一个聊天机器人。

### 实例描述

本实例用已有的对话语料（一问一答模式）进行训练，制作一个聊天机器人。当输入问题后，聊天机器人会计算出要回答的语句并显示。

新版本的 API 将注意力机制、解码器等几个主要的功能分别进行封装，直接用相应的 Wapper 类进行实例化。在编码与解码的过程中，用函数 `dynamic_rnn` 创建可以支持变长输入的动态 RNN 模型。这里不再需要用 `model_with_buckets` 等方法来实现桶机制。具体做法如下。

### 9.4.1 下载及处理样本

该例子的样本来源于 GitHub 网站上的一个开源项目，地址如下：

```
https://github.com/gunthercox/chatterbot-corpus
```

该项目中有多种语言的对话语料，如图 9-10 所示。

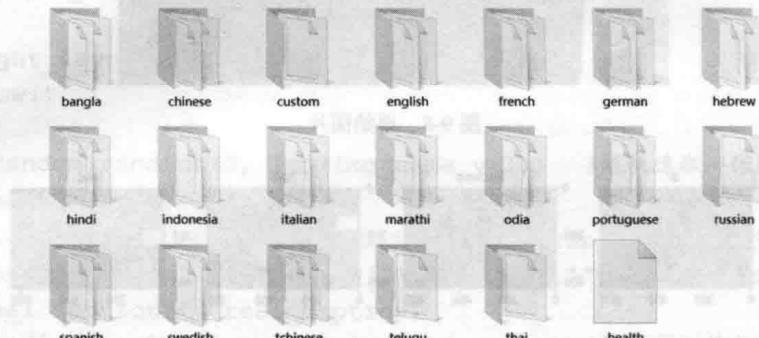


图 9-10 对话机器人模型多语言的对话语料

下载数据集，并将中文语料的部分文件夹（chinese 文件夹）复制到本地代码的同级目录下。该文件夹下有关于各个领域的对话语料，如图 9-11 所示。

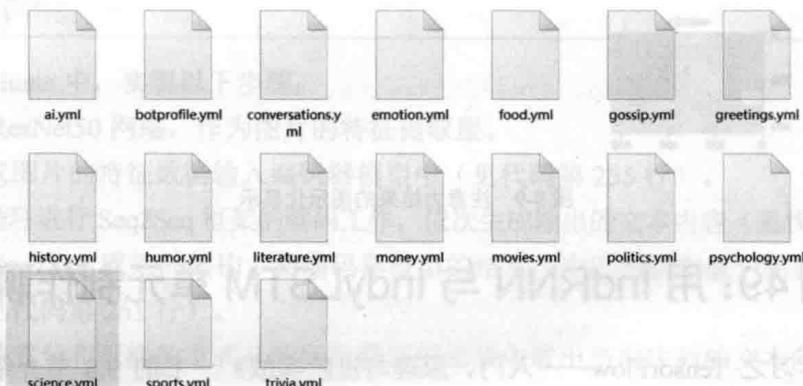


图 9-11 对话机器人的中文语料

每个文档都有相同的格式，例如“ai.yml”中的内容如下：

```
categories:
- AI
conversations:
- - 什么是人工智能
  - 人工智能是工程和科学的分支，致力于构建思维的机器
  - - 你写的是什么语言
    - python
```

其中代码第 1、2 行是该文档的分类，第 3 行及以下是对话内容。其中分为问题与回答两部分：

- 以“--”开头的是问题。
- 以“-”开头的是回答。



### 提示：

该实例只实现一问一答的对话模式，默认对话与对话之间没有上下文关系，每次对话都是独立的。

为了适应模型场景，还需要将有上下文联系的对话样本删除，即把样本文件“conversations.yml”与“literature.yml”直接从样本库里删除。

## 9.4.2 代码实现：读取样本，分词并创建字典

将chinese文件夹下的所有文件读入内存中，并对其进行分词，根据分词结果创建字典。为了使代码更简洁，这里用tf.keras接口生成字典。

分词操作使用的是第三方库jieba。有关jieba库的使用这里不做展开。有兴趣的读者可以参考《深度学习之TensorFlow——入门、原理与进阶实战》一书的9.7.4小节。

具体代码如下。

### 代码9-4 用估算器实现带注意力机制的Seq2Seq模型

```

01 import numpy as np
02 import tensorflow as tf
03 from tensorflow.contrib import layers
04 from sklearn.model_selection import train_test_split
05
06 START_TOKEN = 0
07 END_TOKEN = 1
08
09 import os
10 import jieba
11 path = "./chinese/"                                #指定数据的文件夹
12
13 alltext= []
14 for file in os.listdir(path):                      #获得所有文件
15     with open(path+file, 'r', encoding='UTF-8') as f: #依次打开文件
16         strtext = f.read().split('\n')                  #按行读取，变为列表
17         strtext=list(filter(lambda x:len(x)>0, strtext))
18         strtext = list(map(lambda x:".
".join(jieba.cut(x.replace('-', '').replace(' ', ''))), strtext[3:])) #用jieba库进行分词，并处理
19         print(file,strtext[:2])
20         alltext = alltext+strtext
21         print(len(alltext))
22
23 top_k = 5000                                     #过滤文本，选出5000个

```

```

24 #生成字典
25 tokenizer = tf.keras.preprocessing.text.Tokenizer(num_words=top_k,
26 oov_token=<unk>")
27 tokenizer.fit_on_texts(alltext)
28 #构造字典
29 tokenizer.word_index = {key:value for key, value in
30 tokenizer.word_index.items() if value <= top_k}
31 tokenizer.word_index[tokenizer.oov_token] = top_k + 1 #添加其他字符
32 tokenizer.word_index['<start>'] = START_TOKEN
33 tokenizer.word_index['<end>'] = END_TOKEN
34 #反向字典
35 index_word = {value:key for key, value in tokenizer.word_index.items()}
36 print(len(index_word))

```

代码第 17 行，将无效行过滤。

代码第 18 行，将每行字符中的“-”去掉，再进行分词，并将结果用 join 连接起来。



### 提示：

字符串的 join 语法属于 Python 基础语法。不熟悉的读者可以参考《Python 带我起飞——入门、进阶、商业实战》一书的 6.3 节、4.4.6 小节及 5.7 节。

在代码第 31、32 行，分别向字典中加入 START\_TOKEN 标签与 END\_TOKEN 标签，这两个标签是为了在 Seq2Seq 模型处理样本过程中，告诉模型输入样本的起始和结束位置信息。

代码运行后，输出如下结果：

```

ai.yml ['什么是 ai', '人工智能是工程和科学的分支，致力于构建思维的机器。']
110
.....
psychology.yml ['让我问你一个问题', '当然可以']
748
science.yml ['什么是热力学定律', '我不是一个物理学家，但我觉得这事做热，熵和节约能源，对不对？']
806
sports.yml ['每年 PRO 棒球', '金手套。']
846
1525

```

在输出结果中，以行的方式显示出所有数据集的文件名称，以及该文件中的第一个问答内容。

在每个文件名称的下一行，显示了该文件中的句子总数（例如：输出结果的第 1、2 行显示了在文件 ai.yml 中，一共有 110 个句子。）。

在输出结果的最后一行，输出了整个数据集的字典大小（1525 个词），该字典是数据集分词后的处理结果。

### 9.4.3 代码实现：对样本进行向量化、对齐、填充预处理

样本的预处理包括一些零碎的处理工作，同时它也是训练模型的必要工作。具体介绍如下。

(1) 将内存中的数据样本转化成向量。

(2) 将样本中的句子分解成问题与答案两个数组。其中，问题数据被当作输入数据，答案数据被当作标签数据。

(3) 对问题和答案的数据分别做对齐处理，不足的用 END\_TOKEN 标签填充。

(4) 对所有的句子添加结束标志。

(5) 将整个数据集按照一定比例拆分成训练集与测试集。

具体代码如下。

#### 代码 9-4 用估算器实现带注意力机制的 Seq2Seq 模型(续)

```

37 train_seqs = tokenizer.texts_to_sequences(alltext)      #变为向量
38
39 inputseq,outseq = train_seqs[0::2], train_seqs[1::2]; #拆分成问题与答案
40 print(len(inputseq), len(outseq))                      #输出二者的长度，便于对比
41
42 #按照最长的句子对齐。不足的在其后面补 END_TOKEN
43 input_vector = tf.keras.preprocessing.sequence.pad_sequences(inputseq,
   padding='post',value=END_TOKEN)
44 output_vector = tf.keras.preprocessing.sequence.pad_sequences(outseq,
   padding='post',value=END_TOKEN)
45
46 end = np.ones_like(input_vector[:,0])                  #对所有的句子添加结束标志
47 end = np.reshape(end, [-1,1])
48 print(np.shape(start),np.shape(input_vector),np.shape(end))
49 input_vector = np.concatenate((input_vector,end),axis= 1)
50 output_vector = np.concatenate((output_vector,end),axis= 1)
51
52 print("in 最大长度",len(input_vector[0]))
53 print("out 最大长度",len(output_vector[0]))
54 in_max_length =len(input_vector[0])
55 out_max_length =len(output_vector[0])                  #计算最大长度
56
57 input_vector_train, input_vector_val, output_vector_train,
   output_vector_val = train_test_split(input_vector,
   output_vector, test_size=0.2, random_state=0)          #拆分成训练集与测试集

```

### 9.4.4 代码实现：在 Seq2Seq 模型中加工样本

编写函数 seq2seq 实现 Seq2Seq 框架的主体结构。

在函数 seq2seq 中，首先要对输入样本进行统一化加工。具体的加工步骤如下。

(1) 为每个输入添加 START\_TOKEN 标志。

(2) 计算每个输入及标签的长度。

(3) 将输入和标签分别转换为各自的词嵌入形式。

具体代码如下。

#### 代码 9-4 用估算器实现带注意力机制的 Seq2Seq 模型（续）

```

58 useScheduled=True                                         # 设置训练过程中的采样方式
59 def seq2seq(mode, features, labels, params):
60     vocab_size = params['vocab_size']
61     embed_dim = params['embed_dim']
62     num_units = params['num_units']
63     output_max_length = params['output_max_length']
64
65     print("获得输入张量的名字", features.name, labels.name)
66     inp = tf.identity(features[0], 'input_0')                 # 用于钩子函数显示
67     output = tf.identity(labels[0], 'output_0')
68     batch_size = tf.shape(features)[0]
69     # 按照指定形状，复制 START_TOKEN
70     start_tokens = tf.tile([START_TOKEN], [batch_size])
71     train_output = tf.concat([tf.expand_dims(start_tokens, 1), labels], 1) # 添加开始标志
72     # 计算长度
73     input_lengths = tf.reduce_sum(tf.cast(tf.not_equal(features,
74 END_TOKEN), tf.int32), 1, name="len")
75     output_lengths = tf.reduce_sum(tf.cast(tf.not_equal(train_output,
76 END_TOKEN), tf.int32), 1, name="outlen")
77     # 生成问题与回答的词嵌入
78     input_embed = layers.embed_sequence( features, vocab_size=vocab_size,
79                                         embed_dim=embed_dim, scope='embed')
80     output_embed = layers.embed_sequence( train_output,
81                                         vocab_size=vocab_size, embed_dim=embed_dim, scope='embed', reuse=True)
82
83     with tf.variable_scope('embed', reuse=True):                # 用于模型的使用场景
84         embeddings = tf.get_variable('embeddings')
```

代码第 65 行，将张量 `features` 与张量 `labels` 的名字打印出来。张量 `features` 与张量 `labels` 的名字会在向模型注入数据时被使用（见 9.4.9 小节）。

代码第 66、67 行用 `tf.identity` 函数将张量复制一份。在运行代码时，新复制的张量会根据图中指定的名字显示其具体值（见 9.4.9 小节）。

代码第 70 行，用 `tf.tile` 函数将 `START_TOKEN` 标签按照形状 `[batch_size]` 进行复制，得到与输入数据第 0 维度（批次数量 `batch_size`）相同的张量。然后用 `tf.concat` 函数将 `START_TOKEN` 标签贴在每个输入数据的前面。



**提示：**

由于 `START_TOKEN` 的值为 0，所以代码第 70 行也可以被替换为 `tf.zeros([batch_size], dtype=tf.int32)`，直接生成 `[batch_size]` 个 0。

### 9.4.5 代码实现：在 Seq2Seq 模型中，实现基于 IndRNN 与 IndyLSTM 的动态多层 RNN 编码器

在 Seq2Seq 模型中，用 IndRNN 单元与 IndyLSTM 单元一起组成了动态的多层 RNN 编码器。具体代码如下。

代码 9-4 用估算器实现带注意力机制的 Seq2Seq 模型（续）

```

81     Indcell = tf.nn.rnn_cell.  
     DeviceWrapper(tf.contrib.rnn.IndRNNCell(num_units=num_units),  
     "/device:GPU:0")  
82     IndyLSTM_cell = tf.nn.rnn_cell.  
     DeviceWrapper(tf.contrib.rnn.IndyLSTMCell(num_units=num_units),  
     "/device:GPU:0")  
83     multi_cell = tf.nn.rnn_cell.MultiRNNCell([Indcell, IndyLSTM_cell])  
84     encoder_outputs, encoder_final_state = tf.nn.dynamic_rnn(multi_cell,  
     input_embed, sequence_length=input_lengths, dtype=tf.float32)

```

在代码第 82、83 行中，都用 DeviceWrapper 语法了指派了运行设备。这里也是为了演示 DeviceWrapper 语法的具体用法。

### 9.4.6 代码实现：为 Seq2Seq 模型中的解码器创建 Helper

在下面的代码中创建了两个采样器 train\_helper 与 pred\_helper。前者用于训练模型场景，后者用于使用模型场景。

- 在训练模型时，可以选择用 TrainingHelper 函数（一般方式）或是 ScheduledEmbeddingTrainingHelper 函数（采用多项式分布的采样方式，见 9.1.14 小节）来实现采样器 train\_helper。
- 在使用模型时，用 GreedyEmbeddingHelper 函数来实现采样器 pred\_helper。

具体代码如下。

代码 9-4 用估算器实现带注意力机制的 Seq2Seq 模型（续）

```

85     if useScheduled:                                     #根据配置选择 train_helper 的实现方式  
86         train_helper =  
87             tf.contrib.seq2seq.ScheduledEmbeddingTrainingHelper(output_embed,  
88             tf.tile([output_max_length], [batch_size]),  
89             embeddings, 0.3)  
90     else:  
91         train_helper = tf.contrib.seq2seq.TrainingHelper(output_embed,  
92             tf.tile([output_max_length],  
93             [batch_size]))  
94     #实现 pred_helper  
95     pred_helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(embeddings,  
96             start_tokens=tf.tile([START_TOKEN], [batch_size]),  
97             end_token=END_TOKEN)

```

在生成采样器 `train_helper` 对象时，用最大长度 `output_max_length`（见代码第 87、90 行）来指定采样器的长度。这很关键，只有按照最大长度采样才能得到与标签序列一样的长度（因为输入标签的序列长度也是最大长度 `output_max_length`）。这样在计算损失值 `loss` 时，才不会报错。否则在计算损失值 `loss` 值时，还需要对输出结果进行对齐才可以顺利进行。



**提示：**

代码第 89 行很容易会被写成以下这样：

```
train_helper = tf.contrib.seq2seq.TrainingHelper(output_embed, output_lengths)
```

这里是在 TensorFlow 的 Seq2Seq 接口中最容易犯错的地方。因为这种代码埋藏了一个隐含的 Bug。

在处理定长序列数据时，往往不会报错。一旦输入和输出都是变长序列，则程序将在计算 loss 值时，报出维度不匹配的错误。

除像代码第 86、89 行那样写外，还可以直接在计算 loss 值时将生成的结果填充成与标签序列相等的维度，再计算 loss 值，详见配套资源中的代码文件“9-5 估算器实现带注意力机制的 Seq2Seq 模型——手动对齐.py”。

#### 9.4.7 代码实现：实现带有 Bahdanau 注意力、dropout、OutputProjectionWrapper 的解码器

TensorFlow 中的新版 Seq2Seq 接口对解码器进行了调整。实现带有注意力机制的解码器主要有以下几个步骤。

- (1) 创建一个注意力机制对象和一个 RNN 单元。
  - (2) 用 `AttentionWrapper` 函数将注意力机制作用于 RNN 单元，生成 `attn_cell` 对象。
  - (3) 用 `OutputProjectionWrapper` 函数或全连接层对 `attn_cell` 对象进行维度变化，得到 `out_cell` 对象。
  - (4) 调用 `BasicDecoder` 函数并传入 `out_cell` 对象和 `helper` 对象，生成解码器模型。
  - (5) 将解码器模型放入 `dynamic_decode` 函数中进行解码，生成最终结果。

代码 6-4 用竹筒器实现带注意机制的 Gated-GRU 模型（续）

```
94     def decode(helper, scope, reuse=None):
95         with tf.variable_scope(scope, reuse=reuse):
96             attention_mechanism = tf.contrib.seq2seq.BahdanauAttention(
97                 num_units=num_units,
98                 memory=encoder_outputs,
99                 memory_sequence_length=input_lengths)
```

```

100     cell = tf.contrib.rnn.InRNNCell(num_units=num_units)
101     if reuse == None: #为模型添加 dropout 方法
102         keep_prob=0.8
103     else:
104         keep_prob=1
105     cell = tf.nn.rnn_cell.DropoutWrapper(cell,
106                                         output_keep_prob=keep_prob)
107     attn_cell = tf.contrib.seq2seq.AttentionWrapper(
108                                         cell, attention_mechanism, attention_layer_size=num_units /
109                                         2)
110     out_cell = tf.contrib.rnn.OutputProjectionWrapper(
111                                         attn_cell, vocab_size, reuse=reuse)
112     )
113     decoder = tf.contrib.seq2seq.BasicDecoder(cell=out_cell,
114                                         helper=helper,
115                                         initial_state=out_cell.zero_state(dtype=tf.float32,
116                                         batch_size=batch_size))
117     outputs = tf.contrib.seq2seq.dynamic_decode(
118                                         decoder=decoder, output_time_major=False,
119                                         impute_finished=True, maximum_iterations=output_max_length
120     )
121     return outputs[0]

```

代码第 105 行用 DropoutWrapper 函数为解码器添加 Dropout 层。DropoutWrapper 函数实现了 RNN 模型的 dropout 方法，该函数既可以通过 output\_keep\_prob 参数指定层与层之间的 dropout 操作，还可以通过 input\_keep\_prob 参数指定序列与序列之间的 dropout 操作。



### 提示：

更多有关 RNN 模型的 dropout 方法，请参考《深度学习之 TensorFlow——入门、原理与进阶实战》一书的 9.4.15 小节。

## 9.4.8 代码实现：在 Seq2Seq 模型中实现反向优化

下面是关于函数 seq2seq 的最后一部分内容：用 sequence\_loss 函数来计算损失值，并返回估算器模型。具体代码如下。

### 代码 9-4 用估算器实现带注意力机制的 Seq2Seq 模型（续）

```

121     train_outputs = decode(train_helper, 'decode') #训练场景的解码结果
122     pred_outputs = decode(pred_helper, 'decode', reuse=True) #使用场景的结果
123     #复制张量中的一个值，用于显示
124     tf.identity(train_outputs.sample_id[0], name='train_pred')
125

```

```

126     masks = tf.sequence_mask(output_lengths, output_max_length, #计算掩码
127                               dtype=tf.float32, name="masks")
128     #计算 loss 值
129     loss = tf.losses.seq2seq.sequence_loss(train_outputs.rnn_output,
130                                             labels, weights=masks)
131     #优化器
132     train_op = layers.optimize_loss(loss, tf.train.get_global_step(),
133                                     optimizer= params.get('optimizer', 'Adam'),
134                                     learning_rate= params.get('learning_rate', 0.001),
135                                     summaries=['loss', 'learning_rate'])
136     #用于钩子函数显示
137     tf.identity(pred_outputs.sample_id[0], name='predictions')
138     #返回估算器模型
139     return tf.estimator.EstimatorSpec(mode=mode,
predictions=pred_outputs.sample_id, loss=loss, train_op=train_op)

```

代码第 121、122 行，用共享变量的技术生成了训练场景和使用场景的解码结果。



### 提示：

共享变量属于静态图中的多模型权重共享技术。

在 TensorFlow 2.0 之后的版本中主推动态图的使用方式，共享变量技术将会被逐渐淡化，所以这里也不展开介绍。有兴趣的读者可以参考《深度学习之 TensorFlow——入门、原理与进阶实战》一书的 4.3 节）。

代码第 126 行调用 `tf.sequence_mask` 函数，根据输入标签数据的真实长度来创建掩码。该行代码也可以写成如下样子：

```
masks = tf.cast(tf.not_equal(train_output[:, :-1], 0), tf.float32)
```

生成的掩码会在计算 loss 值的过程中将标签数据中的填充部分忽略（见代码 129 行）。

## 9.4.9 代码实现：创建带有钩子函数的估算器，并进行训练

下面的代码实现了在估算器中注册钩子函数并打印日志。通过注册钩子函数可以将模型中的任意张量打印出来，这相当于在静态图中通过会话中的 `run` 方法打印模型中的任意张量节点的效果。

用钩子函数打印日志的具体步骤如下。

- (1) 创建一个入口函数 `feed_fn`。
- (2) 在 `feed_fn` 函数内部，用字典类型构造指定的输入数据。该字典对象会被注入模型中（见代码第 182 行）。
- (3) 在构造输入数据时，字典中的关键字（key）为输入张量的名称。该名称可通过输出函数进行查看（见代码第 65 行）。在本实例中，输入的张量是两个占位符，其名称是

“IteratorGetNext:0”与“IteratorGetNext:0”。

(4) 用函数 `tf.train.LoggingTensorHook` 定义钩子函数的打印过程。

(5) 在估算器模型的 `train` 方法中, 用 `tf.train.FeedFnHook` 函数对 `feed_fn` 进行封装, 并将封装后的结果与步骤(4)所生成的打印过程一起组成数组, 传入参数 `hooks` 中。

在本实例中定义了3个打印过程。其中:

- 代码第173、176行分别将指定张量内容按照注册好的函数 `get_formatter` 进行打印。
- 代码第179行定义钩子函数 `print_len`。该函数使用默认的输出功能, 可以将张量图中的节点内容打印出来。

具体代码如下。

#### 代码9-4 用估算器实现带注意力机制的Seq2Seq模型(续)

```

140 BATCH_SIZE = 10                                # 定义批次
141 params = {                                     # 定义模型参数
142     'vocab_size': len(index_word),
143     'batch_size': BATCH_SIZE,
144     'output_max_length': out_max_length,
145     'embed_dim': 100,
146     'num_units': 256
147 }
148
149 model_dir='./modelrnn'                          # 定义模型路径
150 est = tf.estimator.Estimator( model_fn=seq2seq, model_dir=model_dir,
151                               params=params)
152 # 定义训练集的输入函数
153 def train_input_fn(input_vector, output_vector, batch_size):
154     # 构造数据集的组成: 一个特征输入, 一个标签输入
155     dataset = tf.data.Dataset.from_tensor_slices( (input_vector,
156                                                       output_vector) )
157     # 将数据集乱序、重复、批次划分
158     dataset = dataset.shuffle(1000).repeat().batch(batch_size,
159     drop_remainder=True).
160     return dataset
161
162 def get_formatter(keys, rev_vocab): # 定义格式化函数, 用于钩子函数的格式化显示
163     def to_str(sequence):          # 定义内嵌函数, 根据词向量生成字符串
164         tokens = [
165             rev_vocab.get(x,
166                         "<UNK>") for x in filter(lambda x:x!=END_TOKEN and x!=
167             START_TOKEN, sequence)]
168         return ' '.join(tokens)
169     def format(values):            # 定义内嵌函数, 提取词向量
170         res = []
171         for key in keys:
172             res.append("%s = %s" % (key, to_str(values[key])))
173         return '\n'.join(res)

```

```

170     return format
171
172 #注册钩子函数，打印过程信息
173 print_inputs = tf.train.LoggingTensorHook(['input_0', 'output_0'],
174     every_n_iter=200, formatter=get_formatter(['input_0',
175     'output_0'], index_word))          #定义钩子函数，每200步输出一次
176 print_predictions = tf.train.LoggingTensorHook(['predictions',
177     'train_pred'], every_n_iter=200, formatter=get_formatter(['predictions',
178     'train_pred'], index_word))        #定义钩子函数，每200步输出一次
179 print_len = tf.train.LoggingTensorHook(['len',"outlen","input_0",
180     "train_pred"],every_n_iter=500)      #定义钩子函数，每500步输出一次
181
182 def feed_fn():                      #定义钩子函数的输入
183     index = np.random.randint(len(input_vector_val)-BATCH_SIZE)
184     return {'IteratorGetNext:0':input_vector_val[index:index+BATCH_SIZE],
185             'IteratorGetNext:1': output_vector_val[index:index+BATCH_SIZE]}
186
187 #训练模型
188 est.train(lambda: train_input_fn(input_vector_train, output_vector_train,
189             BATCH_SIZE),
190             hooks=[tf.train.FeedFnHook(feed_fn),
191                 print_inputs, print_predictions,print_len],steps=1000)

```

#### 9.4.10 代码实现：用估算器框架评估模型

评估模型的过程如下。

(1) 通过装饰器(见本书 6.4.8 小节)定义了估算器的输入函数 eval\_input\_fn(见代码第 198 行)。

(2) 调用估算器的 evaluate 方法，并将输入函数 eval\_input\_fn 传入参数 input\_fn 中，进行模型评估(见代码第 211 行)。

具体代码如下。

#### 代码 9-4 用估算器实现带注意力机制的 Seq2Seq 模型(续)

```

191 #模型评估
192 def wrapperFun(fn):                  #定义装饰器函数
193     def wrapper():                   #包装函数
194         return fn(input_vector_val, output_vector_val, BATCH_SIZE) #调用原
195     return wrapper
196
197 @wrapperFun  #定义测试或应用模型时，数据集的输入函数
198 def eval_input_fn(input_vector,labels, batch_size):
199     assert batch_size is not None, "batch_size must not be None"    #batch
不允许为空

```

```

200
201     if labels is None:          #如果预测，则没有标签
202         inputs = input_vector
203     else:
204         inputs = (input_vector,labels)
205
206     #构造数据集
207     dataset = tf.data.Dataset.from_tensor_slices(inputs)
208     dataset = dataset.batch(batch_size, drop_remainder=True) #按批次划分
209     return dataset
210
211 train_metrics = est.evaluate(input_fn=eval_input_fn)      #评估模型
212 print("train_metrics",train_metrics)

```

代码第 201 行，通过判断标签 labels 是否为空值，来获取当前函数的调用场景。

- 如果标签 labels 为空，则表示使用场景。
- 如果标签 labels 不为空，则表示评估场景。

代码运行后，输出结果如下。

(1) 训练部分的内容输出：

```

.....
INFO:tensorflow:input_0 = 是个骗子
output_0 = 我总觉得我被我自己的智慧生活。
INFO:tensorflow:predictions = 我是觉得我的自己。
train_pred = <UNK> 我 <UNK> 我 <UNK> <UNK> <UNK> <UNK> <UNK> <UNK> <UNK> <UNK> 。 .....
<UNK> <UNK> <UNK> <UNK> <UNK> 克隆 <UNK> 。 <UNK> <UNK> 很多
.....
INFO:tensorflow:loss = 1.0014467, step = 500 (97.611 sec)
INFO:tensorflow:global_step/sec: 1.04819
INFO:tensorflow:input_0 = 这会让伤心
output_0 = 我没有任何情绪所以我不能真正感到悲伤这样。
INFO:tensorflow:predictions = 我没有任何情绪所以我不能真正感到悲伤这样。
train_pred = <UNK> 没有任何 <UNK> <UNK> <UNK> <UNK> 这样 <UNK> 感到 <UNK> ..... 这样
感到 <UNK> 。 的这样 <UNK> 是。 <UNK> <UNK> 、 <UNK>
INFO:tensorflow:loss = 1.0113558, step = 600 (95.399 sec)
.....
INFO:tensorflow:input_0 = 什么是超声波
output_0 = 超声波在医学诊断和治疗中使用在手术等。
INFO:tensorflow:predictions = 超声波在医学诊断和治疗中使用在手术等。
train_pred = <UNK> <UNK> 医学 <UNK> 和治疗？ <UNK> <UNK> <UNK> 的 <UNK> <UNK> ..... <UNK>
<UNK> <UNK> <UNK> <UNK> <UNK> <UNK> 。 <UNK> <UNK> <UNK> <UNK> <UNK>
INFO:tensorflow:loss = 2.2775753, step = 800 (98.470 sec)
INFO:tensorflow:global_step/sec: 1.02077
INFO:tensorflow:loss = 0.35907432, step = 900 (98.259 sec)
.....

```

在输出结果中可以看到，模型每迭代训练 100 次，就会输出 6 行信息，具体内容如下。

- input\_0：输入模型的样本内容。

- `output_0`: 输入模型的标签内容。
- `predictions`: 模型模型输出的预测结果。
- `train_pred`: 模型输出的训练结果。
- `loss` 与 `step`: `loss` 是模型当前训练的损失结果, `step` 是模型当前的训练步数。
- `global_step/sec`: 平均每次迭代训练所用的时间。

## (2) 评估模型的内容输出:

```
获得输入张量的名字 IteratorGetNext:0 IteratorGetNext:1
input_0:0 output_0:0
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2019-01-06-09:47:49
.....
INFO:tensorflow:Saving dict for global step 10000: global_step = 10000, loss =
0.19673859
INFO:tensorflow:Saving 'checkpoint_path' summary for global step
10000: ./modelrnn\model.ckpt-10000
train_metrics {'loss': 0.19673859, 'global_step': 10000}
```

从输出结果的最后一行可以看到, 模型迭代 10000 次之后, 表示模型的损失值的是 0.19。还可以通过增加迭代次数、复杂化模型的方法继续提升模型的精度。

### 9.4.11 扩展: 用注意力机制的 Seq2Seq 模型实现中英翻译

在 GitHub 网站的 TensorFlow 项目中, 有一个用 `tf.keras` 接口实现的带有注意力机制的 Seq2Seq 模型。该模型可实现英文和法文语言互相翻译, 具体地址如下:

```
https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/eager/python/examples/nmt\_with\_attention/nmt\_with\_attention.ipynb
```

读者可以根据该实例, 配合所学的知识, 尝试将其改成中/英文翻译模型。

## 9.5 实例 50: 预测飞机发动机的剩余使用寿命

本节用 JANET 单元构建一个多层次的 RNN 模型, 来解决数值分析中的回归任务。

### 实例描述

本实例用已有的飞机发动机传感器数值训练模型, 并用模拟的飞机发动机传感器数值来预测飞机发动机在未来 15 个周期内是否可能发生故障和飞机发动机的 RUL( Remaining Useful Life, 剩余使用寿命)。

本实例属于一个深度学习在评估及监控资产状态领域中的应用实例, 其中将日常维护设备的日志与真实的飞机发动机寿命记录组合起来, 形成样本。用该样本训练模型, 让模型能够预测现有飞机发动机的剩余使用寿命。

传统的预测性维护任务, 是在特征工程基础上使用机器学习模型实现的。它需要使用该领

域的专业知识手动构建正确的特征。这种方式对专业人才的依赖性很大，而且做出来的模型与业务耦合性极强，缺少模型的通用性。

深度学习在解决这类问题时，可以自动从数据中提取正确的特征，大大降低了对特征工程的依赖性。

### 9.5.1 准备样本

该实例所使用样本的具体地址如下：

<https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/#turbofan>

该数据集共包括 3 个文件，里面记录着每个发动机的配置数据与该发动机上 21 个传感器的数据，这些数据可以反映出飞机发动机在生命周期中，各个时间点的详细情况，具体介绍如下。

- PM\_train.txt 文件：记录每个飞机发动机完整的生命周期数据。一共含有 100 个飞机发动机的周期性历史数据。具体内容见图 9-12 中的“Sample training data”部分。
- PM\_test.txt 文件：记录每个发动机的部分周期数据。一共含有 100 个飞机发动机的周期性历史数据。具体内容见图 9-12 中的“Sample testing data”部分。
- PM\_truth.txt 文件：记录 PM\_test.txt 文件中每个飞机发动机距离发生故障所剩的周期数。具体内容见图 9-12 中的“Sample ground truth data”部分。

Sample training data												
id	cycle	setting1	setting2	setting3	s1	s2	s3	...	s19	s20	s21	
1	1	-0.0007	-0.0004	100	518.67	641.82	1589.7		100	39.06	23.419	
1	2	0.0019	-0.0003	100	518.67	642.15	1591.82		100	39	23.4236	
1	3	-0.0043	0.0003	100	518.67	642.35	1587.99		100	38.95	23.3442	
...	...											
1	191	0	-0.0004	100	518.67	643.34	1602.36		100	38.45	23.1295	
1	192	0.0009	0	100	518.67	643.54	1601.41		100	38.48	22.9649	
2	1	-0.0018	0.0006	100	518.67	641.89	1583.84		100	38.94	23.4583	
2	2	0.0043	-0.0003	100	518.67	641.82	1587.05		100	39.06	23.4085	
2	3	0.0018	0.0003	100	518.67	641.55	1588.32		100	39.11	23.425	
...	...											
2	286	-0.001	-0.0003	100	518.67	643.44	1603.63		100	38.33	23.0169	
2	287	-0.0005	0.0006	100	518.67	643.85	1608.5		100	38.43	23.0848	

Sample testing data												
id	cycle	setting1	setting2	setting3	s1	s2	s3	...	s19	s20	s21	
1	1	0.0023	0.0003	100	518.67	643.02	1585.29		100	38.86	23.3735	
1	2	-0.0027	-0.0003	100	518.67	641.71	1588.45		100	39.02	23.3916	
1	3	0.0003	0.0001	100	518.67	642.46	1586.94		100	39.08	23.4166	
...	...											
1	30	-0.0025	0.0004	100	518.67	642.79	1585.72		100	39.09	23.4069	
1	31	-0.0006	0.0004	100	518.67	642.58	1581.22		100	38.81	23.3552	
2	1	-0.0009	0.0004	100	518.67	642.66	1589.3		100	39	23.3923	
2	2	-0.0011	0.0002	100	518.67	642.51	1588.43		100	38.84	23.2902	
2	3	0.0002	0.0003	100	518.67	642.58	1595.6		100	39.02	23.4064	
...	...											
2	48	0.0011	-0.0001	100	518.67	642.64	1587.71		100	38.99	23.2918	
2	49	0.0018	-0.0001	100	518.67	642.55	1586.59		100	38.81	23.2618	
3	1	-0.0001	0.0001	100	518.67	642.03	1589.92		100	38.99	23.296	
3	2	0.0039	-0.0003	100	518.67	642.23	1597.31		100	38.84	23.3191	
3	3	0.0006	0.0003	100	518.67	642.98	1586.77		100	38.69	23.3774	
...	...											
3	125	0.0014	0.0002	100	518.67	643.24	1588.64		100	38.56	23.227	
3	126	-0.0016	0.0004	100	518.67	642.88	1589.75		100	38.93	23.274	

Sample ground truth data												
RUL	112	98	69	82	91							
100 rows												

图 9-12 发动机记录样本

**提示：**

本实例只使用一个数据源（传感器值）进行预测。在实际的预测性维护任务中，还有许多其他数据源（例如历史维护记录、错误日志、机器和操作员功能等）。这些数据源都需要被处理成对应的特征数据，然后输入模型里进行计算，以便得到更准确的预测结果。

### 9.5.2 代码实现：预处理数据——制作数据集的输入样本与标签

本实例的任务有两个：

- 预测飞机发动机在未来 15 个周期内是否可能发生故障。
- 预测飞机发动机的剩余使用寿命（RUL）。

前者属于分类问题，后者属于回归问题。

在数据预处理环节，需要设置一个序列数据的时间窗口（在本实例中设为 50），并按照该时间窗口将数据加工成输入的样本数据与标签数据。在本实例中，根据分类任务与回归任务制作出两种标签。

- 分类标签：查找样本中的序列维护记录。以训练样本为例，在 PM\_train.txt 中，以每个发动机为单位，在其中截取 50 个连续的记录作为样本。如果该样本的最后一条记录在该飞机发动机的最后 15 条记录以内，则表明该样本在未来 15 个周期内会出现故障，否则为在未来 15 个周期内不出现故障。
- 回归标签：查找样本中的序列维护记录。以训练样本为例，在 PM\_train.txt 中，以每个飞机发动机为单位，在其中截取 50 个连续的记录作为样本。直接提取最后一条的 RUL 字段作为标签。

制作测试集时，还需要将 PM\_test.txt 文件与 PM\_truth.txt 文件中的内容关联起来，计算出 RUL（见代码第 59~62 行）。

制作好标签后，对数据进行归一化，并将其转换成数据集。具体代码如下。

#### 代码 9-6 预测飞机发动机的剩余使用寿命

```

01 import tensorflow as tf          # 导入模块
02 import pandas as pd
03 import numpy as np
04 import matplotlib.pyplot as plt
05 from sklearn import preprocessing
06
07 # 读入 PM_train 数据
08 train_df = pd.read_csv('./PM_train.txt', sep=" ", header=None)
09 train_df.drop(train_df.columns[[26, 27]], axis=1, inplace=True)
10 train_df.columns = ['id', 'cycle', 'setting1', 'setting2', 'setting3', 's1',
11                     's2', 's3',
12                     's4', 's5', 's6', 's7', 's8', 's9', 's10', 's11', 's12',
13                     's13', 's14',
14                     's15', 's16', 's17', 's18', 's19', 's20', 's21']
15 train_df = train_df.sort_values(['id', 'cycle'])

```

```

14     np_elements = data_matrix.shape[0]xbar_ib_mean = 'b1'ib_mean 88
15 #读入 PM_test 数据
16 test_df = pd.read_csv('./PM_test.txt', sep=" ", header=None)
17 test_df.drop(test_df.columns[[26, 27]], axis=1, inplace=True)
18 test_df.columns = ['id', 'cycle', 'setting1', 'setting2', 'setting3', 's1',
19   's2', 's3', 's4', 's5', 's6', 's7', 's8', 's9', 's10', 's11', 's12',
20   's13', 's14', 's15', 's16', 's17', 's18', 's19', 's20', 's21']
21
22 #读入 PM_truth 数据
23 truth_df = pd.read_csv('./PM_truth.txt', sep=" ", header=None)
24 truth_df.drop(truth_df.columns[[1]], axis=1, inplace=True)
25
26 #处理训练数据
27 rul = pd.DataFrame(train_df.groupby('id')['cycle'].max()).reset_index()
28 rul.columns = ['id', 'max']
29 train_df = train_df.merge(rul, on=['id'], how='left')
30 train_df['RUL'] = train_df['max'] - train_df['cycle']
31 train_df.drop('max', axis=1, inplace=True)
32
33 w0 = 15 #定义了两个分类参数——15 周期与 30 周期
34 w1 = 30
35
36 train_df['label1'] = np.where(train_df['RUL'] <= w1, 1, 0)
37 train_df['label2'] = train_df['label1']
38 train_df.loc[train_df['RUL'] <= w0, 'label2'] = 2
39
40 train_df['cycle_norm'] = train_df['cycle'] #训练数据归一化
41 train_df['RUL_norm'] = train_df['RUL']
42 cols_normalize = list(set(train_df.columns) - set(['id', 'cycle', 'RUL', 'label1', 'label2']))
43 min_max_scaler = preprocessing.MinMaxScaler()
44 norm_train_df =
45 pd.DataFrame(min_max_scaler.fit_transform(train_df[cols_normalize]),
46               columns=cols_normalize,
47               index=train_df.index)
48
49 #合成训练数据特征列
50 join_df =
51 train_df[train_df.columns.difference(cols_normalize)].join(norm_train_df)
52
53 train_df = join_df.reindex(columns=train_df.columns)
54
55 #处理测试数据
56 rul = pd.DataFrame(test_df.groupby('id')['cycle'].max()).reset_index()
57 rul.columns = ['id', 'max']
58
59 truth_df.columns = ['more']

```

```

55 truth_df['id'] = truth_df.index + 1
56 truth_df['max'] = rul['max'] + truth_df['more']
57 truth_df.drop('more', axis=1, inplace=True)
58
59 #生成测试数据的 RUL
60 test_df = test_df.merge(truth_df, on=['id'], how='left')
61 test_df['RUL'] = test_df['max'] - test_df['cycle']
62 test_df.drop('max', axis=1, inplace=True)
63
64 #生成测试标签
65 test_df['label1'] = np.where(test_df['RUL'] <= w1, 1, 0)
66 test_df['label2'] = test_df['label1']
67 test_df.loc[test_df['RUL'] <= w0, 'label2'] = 2
68
69 test_df['cycle_norm'] = test_df['cycle'] #对测试数据进行归一化处理
70 test_df['RUL_norm'] = test_df['RUL']
71 norm_test_df =
    pd.DataFrame(min_max_scaler.transform(test_df[cols_normalize]),
72                           columns=cols_normalize,
73                           index=test_df.index)
74 test_join_df =
    test_df[test_df.columns.difference(cols_normalize)].join(norm_test_df)
75 test_df = test_join_df.reindex(columns = test_df.columns)
76 test_df = test_df.reset_index(drop=True)
77
78 sequence_length = 50 #定义序列的长度
79 def gen_sequence(id_df, seq_length, seq_cols): #按照序列的长度获得序列数据
80     data_matrix = id_df[seq_cols].values
81     num_elements = data_matrix.shape[0]
82
83     for start, stop in zip(range(0, num_elements-seq_length),
84                           range(seq_length, num_elements)):
85         yield data_matrix[start:stop, :]
86
87 #合成特征列
88 sensor_cols = ['s' + str(i) for i in range(1,22)]
89 sequence_cols = ['setting1', 'setting2', 'setting3', 'cycle_norm']
90 sequence_cols.extend(sensor_cols)
91 seq_gen = (list(gen_sequence(train_df[train_df['id']==id], sequence_length,
92                         sequence_cols))
93             for id in train_df['id'].unique())
94 seq_array = np.concatenate(list(seq_gen)).astype(np.float32) #生成训练数据
95 print(seq_array.shape)
96 def gen_labels(id_df, seq_length, label): #生成标签
97     data_matrix = id_df[label].values

```

```

98     num_elements = data_matrix.shape[0]
99     return data_matrix[seq_length:num_elements,:]
100
101 #生成训练分类标签
102 label_gen = [gen_labels(train_df[train_df['id']==id], sequence_length,
103                         ['label1'])]
104                         for id in train_df['id'].unique()]
105 label_array = np.concatenate(label_gen).astype(np.float32)
106 label_array.shape
107
108 #生成训练回归标签
109 labelreg_gen = [gen_labels(train_df[train_df['id']==id], sequence_length,
110                           ['RUL_norm'])]
111                         for id in train_df['id'].unique()]
112 labelreg_array = np.concatenate(labelreg_gen).astype(np.float32)
113 print(labelreg_array.shape)
114 #从测试数据中找到序列长度大于 sequence_length 的数据，并取出其最后 sequence_length
115 seq_array_test_last =
116     [test_df[test_df['id']==id][sequence_cols].values[-sequence_length:]]
117     for id in test_df['id'].unique() if
118     len(test_df[test_df['id']==id]) >= sequence_length]
119 #生成测试数据
120 seq_array_test_last = np.asarray(seq_array_test_last).astype(np.float32)
121 y_mask = [len(test_df[test_df['id']==id]) >= sequence_length for id in
122 test_df['id'].unique()]
123 #生成分类回归标签
124 label_array_test_last =
125     test_df.groupby('id')['label1'].nth(-1)[y_mask].values
126 label_array_test_last = label_array_test_last.reshape(label_array_test_last.shape[0],1).astype(
127 np.float32)
128 #生成测试回归标签
129 labelreg_array_test_last =
130     test_df.groupby('id')['RUL_norm'].nth(-1)[y_mask].values
131 labelreg_array_test_last = labelreg_array_test_last.reshape(labelreg_array_test_last.shape[0],1).as-
132 type(np.float32)
133
134 BATCH_SIZE = 80                                #指定批次
135
136 dataset = tf.data.Dataset.from_tensor_slices((seq_array,
137                                                 (label_array, labelreg_array))).shuffle(1000)
138 dataset = dataset.repeat().batch(BATCH_SIZE)
139
140
141

```

```

132 # 测试集
133 testdataset = tf.data.Dataset.from_tensor_slices((seq_array_test_last,
    (label_array_test_last, labelreg_array_test_last)))
134 testdataset = testdataset.batch(BATCH_SIZE, drop_remainder=True)

```

代码第 43 行，用 sklearn 库中的 preprocessing 函数对数据进行归一化处理。



### 提示：

在第一次归一化处理后，需要将当时归一化的极值保存。在应用模型时，需要使用同样的极值来做归一化，这样才保证模型的数据分布统一。

## 9.5.3 代码实现：构建带有 JANET 单元的多层动态 RNN 模型

在随书配套资源中找到源代码文件“JANetLSTMCell.py”，该文件是 JANET 单元的具体代码实现（在 LSTM 单元结构上只保留了忘记门）。将其复制到本地代码的同级目录下。

编写代码，实现如下逻辑：

- (1) 导入实现 JANET 单元的代码模块。
- (2) 用 `tf.nn.dynamic_rnn` 接口创建包含 3 层 JANET 单元的 RNN 模型。
- (3) 在每层后面增加 dropout 功能。
- (4) 建立两个损失值：一个用于分类，另一个用于回归。
- (5) 对两个损失值取平均数，得到总的损失值。
- (6) 建立 Adam 优化器，用于反向传播。

具体代码如下。

### 代码 9-6 预测飞机发动机的剩余使用寿命（续）

```

135 import JANetLSTMCell
136 tf.reset_default_graph()
137 learning_rate = 0.001          # 定义学习率
138
139 # 构建网络节点
140 nb_features = seq_array.shape[2]
141 nb_out = label_array.shape[1]
142 reg_out = labelreg_array.shape[1]
143 n_classes = 2
144 x = tf.placeholder("float", [None, sequence_length, nb_features])
145 y = tf.placeholder(tf.int32, [None, nb_out])
146 yreg = tf.placeholder("float", [None, reg_out])
147
148 hidden = [100, 50, 36]          # 配置每层的 JANET 单元的个数
149 stacked_rnn = []
150 for i in range(3):
151     cell = JANetLSTMCell.JANetLSTMCell(hidden[i], t_max=sequence_length)

```

```

152     stacked_rnn.append(tf.nn.rnn_cell.DropoutWrapper(cell,
153         output_keep_prob=0.8))
154
155 mcell = tf.nn.rnn_cell.MultiRNNCell(stacked_rnn)
156 outputs, _ = tf.nn.dynamic_rnn(mcell, x, dtype=tf.float32)
157 outputs = tf.transpose(outputs, [1, 0, 2])
158 print(outputs.get_shape())
159 pred = tf.layers.conv2d(tf.reshape(outputs[-1], [-1, 6, 6, 1]), n_classes, 6, activation =
    tf.nn.relu)
160 pred = tf.reshape(pred, (-1, n_classes)) #分类模型
161 predreg = tf.layers.conv2d(tf.reshape(outputs[-1], [-1, 1, 1, 36]), 1, 1, activation =
    tf.nn.sigmoid)
162 predreg = tf.reshape(predreg, (-1, 1)) #回归模型
163
164 costreg = tf.reduce_mean(abs(predreg - yreg))
165 costclass = tf.reduce_mean(tf.losses.sparse_softmax_cross_entropy(logits=pred,
    labels=y))
166
167 cost = (costreg+costclass)/2 #总的损失值
168 optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

```

JANET 单元是一个只有忘记门的 GRU 单元或 LSTM 单元结构，更多介绍见 9.1.6 小节。

#### 9.5.4 代码实现：训练并测试模型

编写代码，完成如下步骤。

- (1) 生成数据集迭代器。
- (2) 在会话（session）中训练模型。
- (3) 待训练结束后，将模型测试的结果打印出来。

具体代码如下。

##### 代码 9-6 预测飞机发动机的剩余使用寿命（续）

```

169 iterator = dataset.make_one_shot_iterator() #生成一个训练集的迭代器
170 one_element = iterator.get_next()
171
172 iterator_test = testdataset.make_one_shot_iterator()#生成一个测试集的迭代器
173 one_element_test = iterator_test.get_next()
174
175 EPOCHS = 5000 #指定迭代次数
176 with tf.Session() as sess:
177     sess.run(tf.global_variables_initializer())

```

```

178     #训练模型
179     for epoch in range(EPOCHS):
180         alloss = []
181         inp, (target,targetreg) = sess.run(one_element)
182         if len(inp)!= BATCH_SIZE:
183             continue
184         predregv,_,loss =sess.run([predreg,optimizer,cost], feed_dict={x:
185             inp, y: target,yreg:targetreg})
186         alloss.append(loss)
187         if epoch%100==0: #每 100 次显示一次结果
188             print(np.mean(alloss))
189
190     #测试模型
191     alloss = [] #收集 loss 值
192     while True:
193         try:
194             inp, (target,targetreg) = sess.run(one_element_test)
195             predv,predregv,loss =sess.run([pred,predreg,cost], feed_dict={x:
196                 inp, y: target,yreg:targetreg})
197             alloss.append(loss)
198             print("分类结果: ",target[:20,0],np.argmax(predv[:20],axis = 1))
199             print("回归结果:
200                 ",np.asarray(targetreg[:20]*train_df['RUL'].max()+train_df['RUL'].min(),np.
201                 int32)[:,0],
202                 np.asarray(predregv[:20]*train_df['RUL'].max()+train_df['RUL'].min(),np.
203                 int32)[:,0])
204             print(loss)
205         except tf.errors.OutOfRangeError:
206             print("测试结束")
207             #可视化显示
208             y_true_test
209             =np.asarray(targetreg*train_df['RUL'].max()+train_df['RUL'].min(),np.int
210             32)[:,0]
211             y_pred_test =
212             np.asarray(predregv*train_df['RUL'].max()+train_df['RUL'].min(),np.int32
213             )[:,0]
214             fig_verify = plt.figure(figsize=(12, 8))
215             plt.plot(y_pred_test, color="blue")
216             plt.plot(y_true_test, color="green")
217             plt.title('prediction')
218             plt.ylabel('value')
219             plt.xlabel('row')
220             plt.legend(['predicted', 'actual data'], loc='upper left')

```

```
215     plt.show()
216     fig_verify.savefig("./model_regression_verify.png")
217     print(np.mean(alloss))
218     break
```

### 9.5.5 运行程序

代码运行后，输出如下结果。

(1) 训练结果：模型的损失值逐渐收敛到 0.05 左右。

0.65047395  
0.21954131  
0.15633471  
.....  
0.052825853  
0.054040894  
0.055623062

(2) 测试结果：分为分类结果、回归结果、测试模型的损失值，共3部分。

输出的可视化结果如图 9-13 所示。

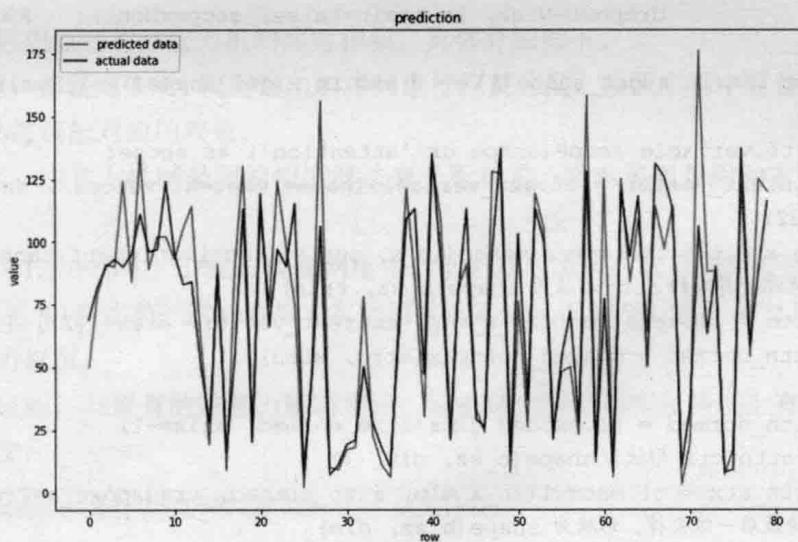


图 9-13 飞机发动机数据预测结果

在图 9-13 中有两条线：一条是真实值（相对峰值较低的线），另一条是预测值（相对峰值较高的线）。可以看出两条线的拟合程度还是很接近的。

## 9.5.6 扩展：为含有 JANET 单元的 RNN 模型添加注意力机制

在 9.5.3 小节代码第 158、161 行，只从 RNN 模型的输出结果中取出最后一个序列，作为预测结果。其实，该网络输出结果的其他序列也是有意义的。可以用注意力机制将其他的序列利用起来，以实现更好的拟合效果。

定义注意力函数 `task_specific_attention`，使用注意力机制为输出结果中的每个序列分配不同的权重。将 9.5.3 小节代码第 156~162 行替换成如下代码：

```

def mkMask(input_tensor, maxLen):                                # 支持变长序列
    shape_of_input = tf.shape(input_tensor)
    shape_of_output = tf.concat(axis=0, values=[shape_of_input, [maxLen]])

    oneDtensor = tf.reshape(input_tensor, shape=(-1,))
    flat_mask = tf.sequence_mask(oneDtensor, maxlen=maxLen)
    return tf.reshape(flat_mask, shape_of_output)

def masked_softmax(inp, seqLen):                                  # 变长序列掩码
    seqLen = tf.where(tf.equal(seqLen, 0), tf.ones_like(seqLen), seqLen)
    if len(inp.get_shape()) != len(seqLen.get_shape())+1:
        raise ValueError('rank of seqLen should be %d, but have the rank %d.\n' %
                           (len(inp.get_shape())-1, len(seqLen.get_shape())))
    mask = mkMask(seqLen, tf.shape(inp)[-1])
    masked_inp = tf.where(mask, inp, tf.ones_like(inp) * (-np.Inf))
    ret = tf.nn.softmax(masked_inp)
    return ret

def task_specific_attention(in_x, xLen, out_sz,
                           dropout=None, is_train=False, scope=None):  # 注意力机制

    assert len(in_x.get_shape()) == 3 and in_x.get_shape()[-1].value is not None

    with tf.variable_scope(scope or 'attention') as scope:
        context_vector = tf.get_variable(name='context_vector', shape=[out_sz],
                                         dtype=tf.float32)
        in_x_mlp = tf.layers.dense(in_x, out_sz, activation=tf.tanh, name='mlp')
        # 点积计算后的 attn 形状为 shape(b_sz, tstep)
        attn = tf.tensordot(in_x_mlp, context_vector, axes=[[2], [0]])
        attn_normed = masked_softmax(attn, xLen)

        attn_normed = tf.expand_dims(attn_normed, axis=-1)
        # 矩阵相乘后的 attn_ctx 形状为 shape(b_sz, dim, 1)
        attn_ctx = tf.matmul(in_x_mlp, attn_normed, transpose_a=True)
        # 将最后一维去掉，形状为 shape(b_sz, dim)
        attn_ctx = tf.squeeze(attn_ctx, axis=[2])
        if dropout is not None:
            attn_ctx = tf.layers.dropout(attn_ctx, rate=dropout,
                                         training=is_train)
    return attn_ctx

```

```

attention_outputs = task_specific_attention(outputs,
np.ones([BATCH_SIZE])*sequence_length, int(outputs.get_shape()[-1]))
pred
= tf.layers.conv2d(tf.reshape(attention_outputs, [-1, 6, 6, 1]), n_classes, 6, activation =
tf.nn.relu)
predreg
= tf.layers.conv2d(tf.reshape(attention_outputs, [-1, 1, 1, 36]), 1, 1, activation =
tf.nn.sigmoid)

pred = tf.reshape(pred, (-1, n_classes))      # 分类模型
predreg = tf.reshape(predreg, (-1, 1))        # 回归模型

```

注意力机制是 RNN 模型的升级版本。RNN 模型处理的序列越长，则注意力机制的效果越明显。在同样超参的情况下，用修改后的代码训练得到的 loss 值是 0.03077051，比不带注意力机制的 RNN 模型的损失值（0.038021535）更低。

## 9.6 实例 51：将动态路由用于 RNN 模型，对路透社新闻进行分类

本实例用带有动态路由算法的 RNN 模型，对序列编码进行信息聚合，实现基于文本的多分类任务。

### 实例描述

用新闻数据集训练模型，让模型能够将新闻按照 46 个类别进行分类。

本实例的思想原理与注意力机制非常相似，具体介绍如下。

- 相同点：都是对 RNN 模型输出的序列进行权重分配，按照序列中对整体语义的影响程度去动态调配对应的权重。
- 不同点：注意力机制是用相似度算法来分配权重，而本实例是用动态路由算法来分配权重。

在本书的 8.1.7 小节中，介绍过胶囊网络中的动态路由算法。其目的是要为  $\mathbf{a}$  分配对应的  $\mathbf{c}$ （ $\mathbf{a}$  与  $\mathbf{c}$  的意义见 8.1.7 小节）这恰恰与本实例的算法需求机制完全一致——为 RNN 模型的输出序列分配注意力权重。

而实践也证明，与原有的注意力机制相比，动态路由算法确实在精度上有所提升。具体介绍可见以下论文：

<https://arxiv.org/pdf/1806.01501.pdf>

### 9.6.1 准备样本

本实例使用的是用 `tf.keras` 接口集成的数据集。该数据集包含 11228 条新闻，共分成 46 个主题。具体接口如下。

```
tf.keras.datasets.reuters
```

该接口与 8.4 节的数据集 `tf.keras.datasets.imdb` 非常相似。不同的是，本实例是多分类任务，而 8.4 节是 2 分类任务。

## 9.6.2 代码实现：预处理数据——对齐序列数据并计算长度

编写代码，实现如下逻辑。

- (1) 用 `tf.keras.datasets.reuters.load_data` 函数加载数据。
- (2) 使用 `tf.keras.preprocessing.sequence.pad_sequences` 函数，对于长度不足 80 个词的句子，在后面补 0；对于长度超过 80 个词的句子，从前面截断，只保留后 80 个词。

具体代码如下。

**代码 9-7 用带有动态路由算法的 RNN 模型对新闻进行分类**

```
01 import tensorflow as tf
02 import numpy as np
03
04 # 定义参数
05 num_words = 20000
06 maxlen = 80
07
08 # 加载数据
09 print('Loading data...')
10 (x_train, y_train), (x_test, y_test) =
    tf.keras.datasets.reuters.load_data(path='./reuters.npz', num_words=num_words)
11
12 # 对齐数据
13 x_train = tf.keras.preprocessing.sequence.pad_sequences(x_train,
    maxlen=maxlen, padding = 'post')
14 x_test = tf.keras.preprocessing.sequence.pad_sequences(x_test,
    maxlen=maxlen, padding = 'post' )
15 print('Pad sequences x_train shape:', x_train.shape)
16
17 leng = np.count_nonzero(x_train, axis = 1) # 计算每个句子的真实长度
```

## 9.6.3 代码实现：定义数据集

将样本数据按照指定批次制作成 `tf.data.Dataset` 接口的数据集，并将不足一批次的剩余数据丢弃。具体代码如下。

**代码 9-7 用带有动态路由算法的 RNN 模型对新闻进行分类（续）**

```
18 tf.reset_default_graph()
19
20 BATCH_SIZE = 100 # 定义批次
```

```

21 # 定义数据集
22 dataset = tf.data.Dataset.from_tensor_slices(((x_train, leng),
23 y_train)).shuffle(1000)
23 dataset = dataset.batch(BATCH_SIZE, drop_remainder=True) # 丢弃剩余数据

```

## 9.6.4 代码实现：用动态路由算法聚合信息

将胶囊网络中的动态路由算法应用在 RNN 模型中还需要做一些改动，具体如下。

- (1) 定义函数 `shared_routing_uhat`。该函数使用全连接网络，将 RNN 模型的输出结果转换成动态路由中的  $\hat{U}$  ( $\hat{U}$  代表 `uhat`) 见代码第 33 行。
- (2) 定义函数 `masked_routing_iter` 进行动态路由计算。在该函数的开始部分（见代码第 50 行），对输入的序列长度进行掩码处理，使动态路由算法支持动态长度的序列数据输入，见代码第 45 行。
- (3) 定义函数 `routing_masked` 完成全部的动态路由计算过程。对 RNN 模型的输出结果进行信息聚合。在该函数的后部分（见代码 87 行），对动态路由计算后的结果进行 `dropout` 处理，使其具有更强的泛化能力（见代码第 78 行）。

具体代码如下。

### 代码 9-7 用带有动态路由算法的 RNN 模型对新闻进行分类（续）

```

24 def mkMask(input_tensor, maxlen): # 计算变长 RNN 模型的掩码
25     shape_of_input = tf.shape(input_tensor)
26     shape_of_output = tf.concat(axis=0, values=[shape_of_input, [maxlen]])
27
28     oneDtensor = tf.reshape(input_tensor, shape=(-1,))
29     flat_mask = tf.sequence_mask(oneDtensor, maxlen=maxlen)
30     return tf.reshape(flat_mask, shape_of_output)
31
32 # 定义函数，将输入转化成 uhat
33 def shared_routing_uhat(caps, # 输入的参数形状为 (batch_size, maxlen, caps_dim)
34                         out_caps_num, # 输出胶囊的个数
35                         out_caps_dim, scope=None): # 输出胶囊的维度
36
37     batch_size, maxlen = tf.shape(caps)[0], tf.shape(caps)[1] # 获取批次和长度
38
39     with tf.variable_scope(scope or 'shared_routing_uhat'): # 转成 uhat
40         caps_uhat = tf.layers.dense(caps, out_caps_num * out_caps_dim,
41                                     activation=tf.tanh)
42
43         caps_uhat = tf.reshape(caps_uhat, shape=[batch_size, maxlen,
44                                         out_caps_num, out_caps_dim])
45
46         # 输出的结果形状为 (batch_size, maxlen, out_caps_num, out_caps_dim)
47
48     return caps_uhat
49
50 def masked_routing_iter(caps_uhat, seqLen, iter_num): # 动态路由计算
51     assert iter_num > 0

```