

- (3) 将 c_l 经过权重矩阵 w_l^{vv} 进行线性变化 (c_l 与 w_l^{vv} 矩阵相乘)。
- (4) 将 c_l^T 经过权重矩阵 w_l^{ev} 进行线性变化。
- (5) 将 (3) 与 (4) 的结果相加, 再与偏置参数 b_l^v 相加, 得到 v_{l+1} 。 v_{l+1} 将用于推荐算法模型的后续计算。
- (6) 按照第 (3)、(4)、(5) 步的做法, 同理可以得到 e_{l+1} 。 e_{l+1} 将用于知识图谱嵌入模型的后续计算。

用 tf.layer 接口实现交叉压缩单元模型, 具体代码如下。

代码 7-14 MKR

```

01 import numpy as np
02 import tensorflow as tf
03 from sklearn.metrics import roc_auc_score
04 from tensorflow.python.layers import base
05
06 class CrossCompressUnit(base.Layer):
07     def __init__(self, dim, name=None):
08         super(CrossCompressUnit, self).__init__(name)
09         self.dim = dim
10         self.f_vv = tf.layers.Dense(1, use_bias = False) #构建权重矩阵
11         self.f_ev = tf.layers.Dense(1, use_bias = False)
12         self.f_ve = tf.layers.Dense(1, use_bias = False)
13         self.f_ee = tf.layers.Dense(1, use_bias = False)
14         self.bias_v = self.add_weight(name='bias_v', #构建偏置权重
15                                       shape=dim,
16                                       initializer=tf.zeros_initializer())
17         self.bias_e = self.add_weight(name='bias_e',
18                                       shape=dim,
19                                       initializer=tf.zeros_initializer())
20
21     def _call(self, inputs):
22         v, e = inputs #v 和 e 的形状为 [batch_size, dim]
23         v = tf.expand_dims(v, dim=2) #v 的形状为 [batch_size, dim, 1]
24         e = tf.expand_dims(e, dim=1) #e 的形状为 [batch_size, 1, dim]
25
26         c_matrix = tf.matmul(v, e) #c_matrix 的形状为 [batch_size, dim, dim]
27         c_matrix_transpose = tf.transpose(c_matrix, perm=[0, 2, 1])
28         #c_matrix 的形状为 [batch_size * dim, dim]
29         c_matrix = tf.reshape(c_matrix, [-1, self.dim])
30         c_matrix_transpose = tf.reshape(c_matrix_transpose, [-1, self.dim])
31
32         #v_output 的形状为 [batch_size, dim]
33         v_output = tf.reshape(
34             self.f_vv(c_matrix) + self.f_ev(c_matrix_transpose),
35             [-1, self.dim]
36             ) + self.bias_v
37         e_output = tf.reshape(
38             self.f_ve(c_matrix) + self.f_ee(c_matrix_transpose),
39             [-1, self.dim]
40             ) + self.bias_e
41
42         return tf.stack([v_output, e_output], axis=1)
43
44     def loss(self, labels, predictions):
45         labels = tf.reshape(labels, [-1])
46         predictions = tf.reshape(predictions, [-1])
47         return 1 - roc_auc_score(labels, predictions)

```

```

38         self.f_ve(c_matrix) + self.f_ee(c_matrix_transpose), (E)
39         [-1, self.dim]
40         ) + self.bias_e
41     #返回结果
42     return v_output, e_output

```

代码第 10 行, 用 `tf.layers.Dense` 方法定义了不带偏置的全连接层, 并在代码第 34 行, 将该全连接层作用于交叉后的特征向量, 实现压缩的过程。

2. 将交叉压缩单元模型集成到 MKR 框架中

在 MKR 框架中, 推荐算法模型和知识图谱词嵌入模型的处理流程几乎一样。可以进行同步处理。在实现时, 将整个处理过程横向拆开, 分为低层和高层两部分。

- 低层: 将所有的输入映射成词嵌入向量, 将需要融合的向量(图 7-15 中的 v 和 h) 输入交叉压缩单元, 不需要融合的向量(图 7-15 中的 u 和 r) 进行同步的全连接层处理。
- 高层: 推荐算法模型和知识图谱词嵌入模型分别将低层的传上来的特征连接在一起, 通过全连接层回归到各自的目标结果。

具体实现的代码如下。

代码 7-14 MKR (续)

```

43 class MKR(object):
44     def __init__(self, args, n_users, n_items, n_entities, n_relations):
45         self._parse_args(n_users, n_items, n_entities, n_relations)
46         self._build_inputs()
47         self._build_low_layers(args)      #构建低层模型
48         self._build_high_layers(args)    #构建高层模型
49         self._build_loss(args)
50         self._build_train(args)
51
52     def _parse_args(self, n_users, n_items, n_entities, n_relations):
53         self.n_user = n_users
54         self.n_item = n_items
55         self.n_entity = n_entities
56         self.n_relation = n_relations
57
58         #收集训练参数, 用于计算 L2 损失
59         self.vars_rs = []
60         self.vars_kge = []
61
62     def _build_inputs(self):
63         self.user_indices=tf.placeholder(tf.int32, [None], 'userInd')
64         self.item_indices=tf.placeholder(tf.int32, [None], 'itemInd')
65         self.labels = tf.placeholder(tf.float32, [None], 'labels')
66         self.head_indices =tf.placeholder(tf.int32, [None], 'headInd')
67         self.tail_indices =tf.placeholder(tf.int32, [None], 'tail_indices')
68         self.relation_indices=tf.placeholder(tf.int32, [None], 'relInd')
69     def _build_model(self, args):
70         self._build_low_layers(args)

```

```

71     self._build_high_layers(args)
72
73 def _build_low_layers(self, args):
74     #生成词嵌入向量
75     self.user_emb_matrix = tf.get_variable('user_emb_matrix',
76                                         [self.n_user, args.dim])
77     self.item_emb_matrix = tf.get_variable('item_emb_matrix',
78                                         [self.n_item, args.dim])
79     self.entity_emb_matrix = tf.get_variable('entity_emb_matrix',
80                                         [self.n_entity, args.dim])
81     self.relation_emb_matrix = tf.get_variable('relation_emb_matrix',
82                                         [self.n_relation, args.dim])
83
84     #获取指定输入对应的词嵌入向量, 形状为[batch_size, dim]
85     self.user_embeddings = tf.nn.embedding_lookup(
86         self.user_emb_matrix, self.user_indices)
87     self.item_embeddings = tf.nn.embedding_lookup(
88         self.item_emb_matrix, self.item_indices)
89     self.head_embeddings = tf.nn.embedding_lookup(
90         self.entity_emb_matrix, self.head_indices)
91     self.relation_embeddings = tf.nn.embedding_lookup(
92         self.relation_emb_matrix, self.relation_indices)
93     self.tail_embeddings = tf.nn.embedding_lookup(
94         self.entity_emb_matrix, self.tail_indices)
95
96     for _ in range(args.L):#按指定参数构建多层MKR结构
97         #定义全连接层
98         user_mlp = tf.layers.Dense(args.dim, activation=tf.nn.relu)
99         tail_mlp = tf.layers.Dense(args.dim, activation=tf.nn.relu)
100        cc_unit = CrossCompressUnit(args.dim)#定义CrossCompress单元
101
102        #实现MKR结构的正向处理
103        self.user_embeddings = user_mlp(self.user_embeddings)
104        self.tail_embeddings = tail_mlp(self.tail_embeddings)
105        self.item_embeddings, self.head_embeddings = cc_unit(
106            [self.item_embeddings, self.head_embeddings])
107
108        #收集训练参数
109        self.vars_rs.extend(user_mlp.variables)
110        self.vars_kge.extend(tail_mlp.variables)
111
112    def _build_high_layers(self, args):
113        #推荐算法模型
114        use_inner_product = True          #指定相似度分数计算的方式
115        if use_inner_product:           #内积方式
116            #self.scores 的形状为[batch_size]
117            self.scores = tf.reduce_sum(self.user_embeddings * self.item_embeddings,
118                                         axis=1)
118        else:
119            #self.user_item_concat 的形状为[batch_size, dim * 2]

```

```

120         self.user_item_concat = tf.concat([self.user_embeddings, self.item_embeddings], axis=1)
121
122         for _ in range(args.H - 1):
123             rs_mlp = tf.layers.Dense(args.dim * 2, activation=tf.nn.relu)
124             #self.user_item_concat 的形状为[batch_size, dim * 2]
125             self.user_item_concat = rs_mlp(self.user_item_concat)
126             self.vars_rs.extend(rs_mlp.variables)
127             #定义全连接层
128             rs_pred_mlp = tf.layers.Dense(1, activation=tf.nn.relu)
129             #self.scores 的形状为[batch_size]
130             self.scores = tf.squeeze(rs_pred_mlp(self.user_item_concat))
131             self.vars_rs.extend(rs_pred_mlp.variables) #收集参数
132             self.scores_normalized = tf.nn.sigmoid(self.scores)
133
134             #知识图谱词嵌入模型
135             self.head_relation_concat = tf.concat( #形状为[batch_size, dim * 2]
136                 [self.head_embeddings, self.relation_embeddings], axis=1)
137             for _ in range(args.H - 1):
138                 kge_mlp = tf.layers.Dense(args.dim * 2, activation=tf.nn.relu)
139                 #self.head_relation_concat 的形状为[batch_size, dim* 2]
140                 self.head_relation_concat = kge_mlp(self.head_relation_concat)
141                 self.vars_kge.extend(kge_mlp.variables)
142
143                 kge_pred_mlp = tf.layers.Dense(args.dim, activation=tf.nn.relu)
144                 #self.tail_pred 的形状为[batch_size, args.dim]
145                 self.tail_pred = kge_pred_mlp(self.head_relation_concat)
146                 self.vars_kge.extend(kge_pred_mlp.variables)
147                 self.tail_pred = tf.nn.sigmoid(self.tail_pred)
148
149                 self.scores_kge = tf.nn.sigmoid(tf.reduce_sum(self.tail_embeddings
150                     self.tail_pred, axis=1))
151                 self.rmse = tf.reduce_mean(
152                     tf.sqrt(tf.reduce_sum(tf.square(self.tail_embeddings - self.tail_pred),
153                         axis=1) / args.dim))

```

代码第 115~132 行是推荐算法模型的高层处理部分，该部分有两种处理方式：

- 使用内积的方式，计算用户向量和电影向量的相似度。有关相似度的更多知识，可以参考 8.1.10 小节的注意力机制。
- 将用户向量和电影向量连接起来，再通过全连接层处理计算出用户对电影的喜爱分值。

代码第 132 行，通过激活函数 sigmoid 对分值结果 scores 进行非线性变化，将模型的最终结果映射到标签的值域中。

代码第 136~152 行是知识图谱词嵌入模型的高层处理部分。具体步骤如下：

- (1) 将电影向量和知识图谱中的关系向量连接起来。
- (2) 将第 (1) 步的结果通过全连接层处理，得到知识图谱三元组中的目标实体向量。
- (3) 将生成的目标实体向量与真实的目标实体向量矩阵相乘，得到相似度分值。
- (4) 对第 (3) 步的结果进行激活函数 sigmoid 计算，将值域映射到 0~1 中。

3. 实现 MKR 框架的反向结构

MKR 框架的反向结构主要是 loss 值的计算，其 loss 值一共分为 3 部分：推荐算法模型模型的 loss 值、知识图谱词嵌入模型的 loss 值和参数权重的正则项。具体实现的代码如下。

代码 7-14 MKR（续）

```

152     def _build_loss(self, args):
153         #计算推荐算法模型的 loss 值
154         self.base_loss_rs = tf.reduce_mean(
155             tf.nn.sigmoid_cross_entropy_with_logits(labels=self.labels,
156             logits=self.scores))
156         self.l2_loss_rs = tf.nn.l2_loss(self.user_embeddings) + tf.nn.l2_loss
157         (self.item_embeddings)
157         for var in self.vars_rs:
158             self.l2_loss_rs += tf.nn.l2_loss(var)
159         self.loss_rs = self.base_loss_rs + self.l2_loss_rs * args.l2_weight
160
161         #计算知识图谱词嵌入模型的 loss 值
162         self.base_loss_kge = -self.scores_kge
163         self.l2_loss_kge = tf.nn.l2_loss(self.head_embeddings) + tf.nn.l2_loss
164         (self.tail_embeddings)
164         for var in self.vars_kge:    #计算 L2 正则
165             self.l2_loss_kge += tf.nn.l2_loss(var)
166         self.loss_kge = self.base_loss_kge + self.l2_loss_kge * args.l2_weight
167
168     def _build_train(self, args):  #定义优化器
169         self.optimizer_rs
170         tf.train.AdamOptimizer(args.lr_rs).minimize(self.loss_rs)
170         self.optimizer_kge = tf.train.AdamOptimizer(args.lr_kge). minimize(self.
171         loss_kge)
171
172     def train_rs(self, sess, feed_dict):      #训练推荐算法模型
173         return sess.run([self.optimizer_rs, self.loss_rs], feed_dict)
174
175     def train_kge(self, sess, feed_dict):      #训练知识图谱词嵌入模型
176         return sess.run([self.optimizer_kge, self.rmse], feed_dict)
177
178     def eval(self, sess, feed_dict):           #评估模型
179         labels, scores = sess.run([self.labels, self.scores_normalized], feed_dict)
180         auc = roc_auc_score(y_true=labels, y_score=scores)
181         predictions = [1 if i >= 0.5 else 0 for i in scores]
182         acc = np.mean(np.equal(predictions, labels))
183         return auc, acc
184
185     def get_scores(self, sess, feed_dict):
186         return sess.run([self.item_indices, self.scores_normalized], feed_dict)

```

代码第 173、176 行，分别是训练推荐算法模型和训练知识图谱词嵌入模型的方法。因为在训练的过程中，两个子模型需要交替的进行独立训练，所以将其分开定义。

- 全局平均池化层，将每张结果图的全局平均值
- 全局平均池化层，对生成的特征地图（feature map）取全局平均值

7.9.4 训练模型并输出结果

训练模型的代码在“7-15 train.py”文件中，读者可以自行参考。代码运行后输出以下结果：

epoch 9 train auc: 0.9540 acc: 0.8817 eval auc: 0.9158 acc: 0.8407 test auc: 0.9155 acc: 0.8399

在输出的结果中，分别显示了模型在训练、评估、测试环境下的分值。

7.10 可解释性算法的意义

本章中使用的算法都具有可解释性。在某些实际的应用场景中，为了保证算法的可控程度最大化，会使用具有可解释性的算法是一个非常硬性的要求。

这要求在处理问题的过程中，并不能只选择效果好的算法，而是需要在具有可解释性的算法中选择效果好的算法。这也是机器学习算法不可替代的价值。所以，建议读者对这类算法要适当关注，切不可全部忽略。

第 8 章

卷积神经网络（CNN）——在图像处理中应用最广泛的模型

卷积神经网络是深度学习中非常重要的一个模型，广泛应用在图像处理中。随着深度学习的发展，卷积神经网络也衍生出很多高级的网络结构及算法单元，其适用领域也由图像处理扩展到自然语言处理、数值分析、声音处理等。

本章就来具体学习卷积神经网络的相关知识。

8.1 快速导读

在学习实例之前，有必要了解一下卷积神经网络的基础知识。

8.1.1 认识卷积神经网络

卷积神经网络（CNN）是深度学习中的经典模型之一。在当今几乎所有的深度学习经典模型中，都能找到卷积神经网络的身影。它可以利用很少的权重，实现出色的拟合效果。

图 8-1 所示是一个卷积神经网络的结构，通常会包括以下 5 个部分。

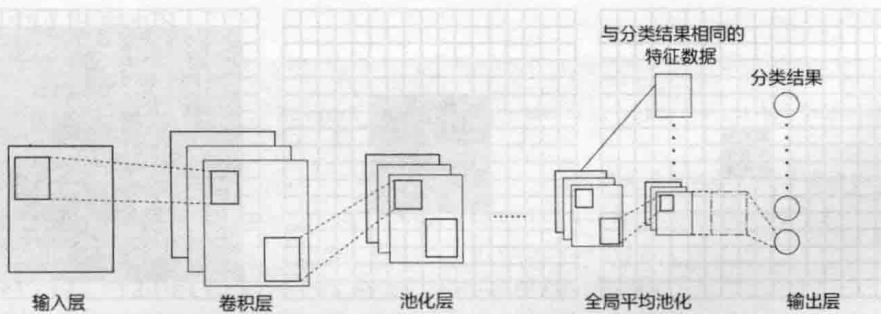


图 8-1 卷积神经网络完整结构

- 输入层：将每个像素代表一个特征节点输入进来。
- 卷积层：由多个滤波器组成。
- 池化层：将卷积结果降维。
- 全局平均池化层：对生成的特征数据（feature map）取全局平均值。

- 输出层：需要分成几类就有几个输出节点。输出节点的值代表预测概率。

卷积神经网络的主要组成部分是卷积层，它的作用是从图像的像素中分析出主要特征。在实际应用中，由多个卷积层通过深度和高度两个方向分析和提取图像的特征：

- 通过较深（多通道）的卷积网络结构，可以学习到图像边缘和颜色渐变等简单特征。
- 通过较高（多层）的卷积网络结构，可以学习到多个简单特征组合中的复杂的特征。

在实际应用中，卷积神经网络并不全是图 8-1 中的结构，而是存在很多特殊的变形。例如：在 ResNet 模型中引入了残差结构，在 Inception 系列模型中引入了多通道结构，在 NASNet 模型中引入了空洞卷积与深度可分离卷积等结构。

另外，卷积神经网络还常和循环神经网络一起应用在自编码网络、对抗神经网络等多模型的网络中。在多模型组合过程中，常用的卷积操作有反卷积、窄卷积、同卷积等。

有关卷积神经网络的原理及常用的操作，还可以参考《深度学习之 TensorFlow——入门、原理与进阶实战》一书的第 8 章。

8.1.2 什么是空洞卷积

空洞卷积（dilated convolutions），又叫扩展卷积或带孔卷积（atrous convolutions）。

这种卷积在图像语义分割相关任务（例如 DeepLab2 模型）中用处很大。它的功能与池化层类似，可以降低维度并能够提取主要特征。

相对于池化层，空洞卷积可以避免在卷积神经网络中进行池化操作时造成的信息丢失问题。

1. 空洞卷积的原理

空洞卷积的操作相对简单，只是在卷积操作之前对卷积核做了膨胀处理。而在卷积过程中，它与正常的卷积操作一样。

在使用时，空洞卷积会通过参数 `rate` 来控制卷积核的膨胀大小。参数 `rate` 与卷积核膨胀的关系如图 8-2 所示。

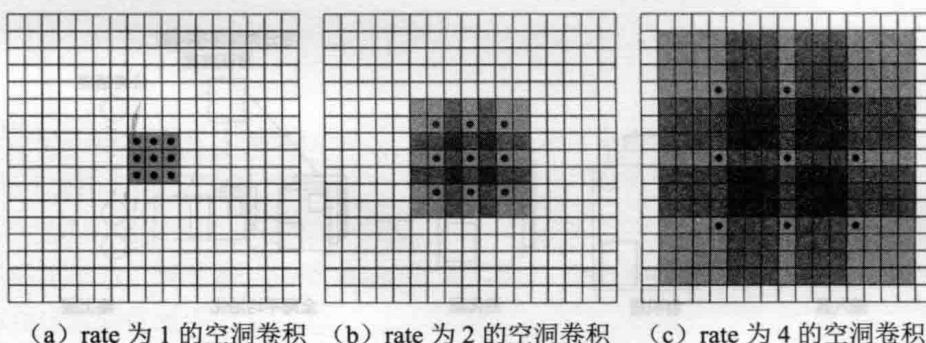


图 8-2 空洞卷积的操作

图 8-2 中的规则解读如下。

- 图 8-2 (a)：如果参数 `rate` 为 1，则表示卷积核不需要膨胀，值为 3×3 ，如图中的点那部分。此时的空洞卷积操作等效于普通的卷积操作。

- 图 8-2 (b)：如果 rate 为 2，则表示卷积核中的每个数字由 1 膨胀到 2。膨胀出来的卷积核值为 0，原有卷积核的值并没有变，如图中点那部分。值变成了 7×7 。
 - 图 8-2 (c)：如果 rate 为 4，则表示卷积核中的每个数字由 1 膨胀到 4。膨胀出来的卷积核值为 0，原有卷积核值并没有变，如图中点那部分。值变成了 15×15 。
- 另外，在卷积操作中，所有的空洞卷积的步长都是 1。

2. TensorFlow 中的空洞卷积函数

在 TensorFlow 中，空洞卷积的函数定义如下：

```
def atrous_conv2d(value, filters, rate, padding, name=None)
```

具体参数含义如下。

- value**: 需要做卷积的输入图像，要求是一个四维张量，形状为 [batch, height, width, channels]。
- filters**: 卷积核，要求是一个四维张量，形状为 [filter_height, filter_width, channels, out_channels]。这里的 channels 是输入通道，应与 value 中的 channels 相同。
- rate**: 卷积核膨胀的参数。要求是一个 int 型的正数。
- padding**: 字符串类型的常量，其值只能取“SAME”或“VALID”。它用于指定不同边缘的填充方式，与普通卷积操作中的补 0 (padding) 规则一致。
- name**: 该函数在张量图中的操作名字。

因为空洞卷积在膨胀时，只是向卷积核中插入了 0，所以仅仅增加了卷积核的大小，并没有增加参数的数量。

与池化的效果类似，使用膨胀后的卷积核在原有输入上做窄卷积 (padding 参数为“VALID”) 操作，可以把维度降下来，并且会保留比池化更丰富的数据。

3. 其他接口中的空洞卷积函数

在 tf.layers 接口中，也可以向 conv2d 函数内传入指定的 dilation_rate，用来实现空洞卷积功能。该函数的定义方法如下：

```
@tf_export('layers.conv2d')
def conv2d(inputs, filters, kernel_size, strides=(1, 1), padding='valid',
data_format='channels_last',
dilation_rate=(1, 1),                      #默认是(1,1)，即普通卷积
activation=None, use_bias=True, kernel_initializer=None,
bias_initializer=init_ops.zeros_initializer(),
kernel_regularizer=None, bias_regularizer=None,
activity_regularizer=None,
kernel_constraint=None, bias_constraint=None, trainable=True, name=None,
reuse=None):
```

另外，tf.keras 接口中的卷积函数 tf.keras.layers.Conv1D、tf.keras.layers.Conv2D 都支持设置参数 dilation_rate。该参数与 tf.layers 接口中 conv2d 函数的 dilation_rate 参数用法相同。

8.1.3 什么是深度卷积

深度卷积是指，将不同的卷积核独立地应用在输入数据的每个通道上。相比正常的卷积操作，深度卷积缺少了最后的“加和”处理。其最终的输出为“输入通道与卷积核个数的乘积”。

在 TensorFlow 中，深度卷积函数的定义方法如下：

```
def depthwise_conv2d(input, filter, strides, padding, rate=None, name=None,
data_format=None)
```

具体参数含义如下。

- **input**: 指需要做卷积的输入图像。
- **filter**: 卷积核。要求是一个 4 维张量，形状为 [filter_height, filter_width, in_channels, channel_multiplier]。这里的 `channel_multiplier` 是卷积核的个数。
- **strides**: 卷积的滑动步长。
- **padding**: 字符串类型的常量，其值只能取“SAME”或“VALID”。它用于指定不同边缘的填充方式，与普通卷积中的 `padding` 一样。
- **rate**: 卷积核膨胀的参数。要求是一个 `int` 型的正数。
- **name**: 该函数在张量图中的操作名字。
- **data_format**: 参数 `input` 的格式，默认为“NHWC”，也可以写成“NCHW”。

该函数会返回 `in_channels × channel_multiplier` 个通道的特征数据（feature map）。

8.1.4 什么是深度可分离卷积

从深度方向可以把不同 `channels` 独立开，先进行特征抽取，再进行特征融合。这样做可以用更少的参数取得更好的效果。



提示：

表示学习（representation learning）是指，基于深度模型的简单特征分析。

1. 深度可分离卷积的原理

在具体实现时，是将深度卷积的结果作为输入，然后进行一次正常的卷积操作。所以，该函数需要两个卷积核作为输入：深度卷积的卷积核 `depthwise_filter`、用于融合操作的普通卷积核 `pointwise_filter`。

例如：对一个输入 `input` 进行深度可分离卷积，具体步骤如下：

(1) 在模型内部会先对输入的数据进行深度卷积，得到 `in_channels × channel_multiplier` (`in_channels` 与 `channel_multiplier` 为 8.1.4 小节中函数 `depthwise_conv2d` 的参数 `filter` 输入通道数和卷积核个数) 个通道的特征数据（feature map）。

(2) 将特征数据（feature map）作为输入，再次用普通卷积核 `pointwise_filter` 进行一次卷积操作。

2. TensorFlow 中的深度可分离卷积函数

在 TensorFlow 中，深度可分离卷积的函数定义如下：

```
def separable_conv2d(input, depthwise_filter, pointwise_filter, strides, padding,
rate=None, name=None, data_format=None)
```

具体参数含义如下。

- **input**: 需要做卷积的输入图像。
- **depthwise_filter**: 用来做函数 `depthwise_conv2d` 的卷积核，即这个函数对输入首先做一次深度卷积。它的形状是`[filter_height, filter_width, in_channels, channel_multiplier]`。
- **pointwise_filter**: 用于融合操作的普通卷积核。例如：形状为`[1, 1, channel_multiplier × in_channels, out_channels]`的卷积核，代表在深度卷积之后的融合操作是采用卷积核为`1 × 1`、输入为`channel_multiplier × in_channels`、输出为`out_channels`的卷积层来实现的。
- **strides**: 卷积的滑动步长。
- **padding**: 字符串类型的常量，只能是“SAME”“VALID”其中之一。指定不同边缘的填充方式，与普通卷积中的 `padding`一样。
- **rate**: 卷积核膨胀的参数。要求是一个 `int` 型的正数。
- **name**: 该函数在张量图中的操作名字。
- **data_format**: 参数 `input` 的格式，默认为“NHWC”，也可以写成“NCHW”。

3. 其他接口中的深度可分离卷积函数

在 `tf.keras` 中，深度方向可分离的卷积函数有以下两个

- `tf.keras.layers.SeparableConv1D`: 支持一维卷积的深度方向可分离的卷积函数。
- `tf.keras.layers.SeparableConv2D`: 支持二维卷积的深度方向可分离的卷积函数。

参数 `depth_multiplier` 用于设置沿每个通道的深度方向进行卷积时输出的通道数量。

8.1.5 了解卷积网络的缺陷及补救方法

传统的卷积神经网络存在泛化性较差、过于依赖样本等缺陷。这是因为，在卷积神经网络中并不能发现组件之间的定向关系和相对空间关系。

一个训练好的卷积神经网络，只能处理比较接近训练数据集的图像。在处理异常的图像数据（例如处理颠倒、倾斜或其他朝向不同的图像）时，其表现会很差。

1. 卷积神经网络的缺陷举例

下面通过图 8-3 来说明卷积神经网络的缺陷：

- (1) 如图 8-3 (a) 所示，卷积神经网络会认为左图和右图同为一张正常人的人脸。
- (2) 如图 8-3 (b) 所示，将右图中人物的眼睛和嘴巴位置置换后，卷积网络错误地认为这是一个正常的人。当然，像图 8-3 (a)、(b) 中的情况比较少见。
- (3) 图 8-3 (c) 所示，如果将右图中的任务倒置，则卷积神经网络便错误地识别成这是一个背景颜色。



图 8-3 卷积神经网络的缺陷

2. 卷积神经网络存在缺陷的原因

图 8-3 中示范的反面例子，皆源于卷积神经网络对图像的理解粒度太粗。造成这种现象的原因是，卷积神经网络中的池化操作弄丢了一些隐含信息。

一般来讲，卷积神经网络的工作原理如下：

- (1) 第 1 层去理解细小的曲线和边缘。
- (2) 第 2 层去理解直线或小形状，例如上嘴唇、下嘴唇等。
- (3) 更高层便开始理解更复杂的形状，例如整个眼睛、整个嘴巴等。
- (4) 最后一层尝试总览全图（例如整个人脸）。

在上述过程中，每一层都会使用卷积核为 3×3 或 5×5 等卷积操作来理解图像，并获得基于像素级别的、非常细微的局部特征。每层的卷积操作完成之后，都会进行一次池化操作。池化本来是用来让特征更明显，但在提升局部特征的同时也弄丢了其内在的其他信息（比如位置信息）。这就造成了在第（4）步总览全图时，对每个局部特征的位置组合不敏感，从而产生了错误。

3. 补救卷积神经网络缺陷的方法

针对卷积神经网络的缺陷，可以用以下 3 种方式进行补救。

- 扩充数据集：训练时将图像进行各种变化，生成更全的多样数据集。通过提升样本的覆盖率，来尽量提升模型的泛化性（模型对一类数据的识别能力）。
- 在模型中，尽量少用或不用池化操作。
- 使用更复杂的模型，让模型在学习局部特征的同时，也关注局部特征间的位置信息（例如胶囊网络模型）。

8.1.6 了解胶囊神经网络与动态路由

胶囊网络（CapsNet）是一个优化过的卷积神经网络模型。它在常规的卷积神经网络模型的基础上做了特定的改进，能够发现组件之间的定向和空间关系。

它将原有的“卷积+池化”组合操作，换成了“主胶囊（PrimaryCaps）+数字胶囊（DigitCaps）”的结构，如图 8-4 所示。

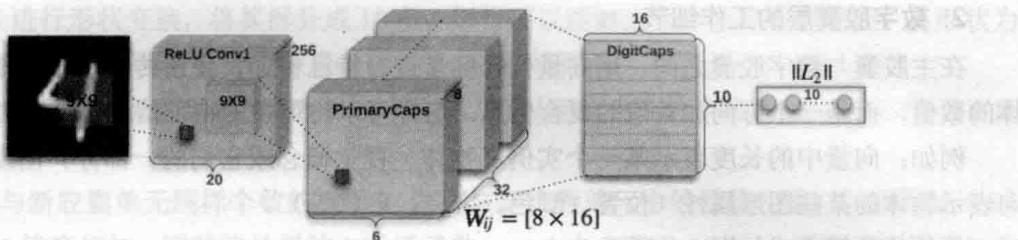


图 8-4 应用在 MNIST 数据集上的胶囊网络架构

图 8-4 是应用在 MNIST 数据集上的胶囊网络架构。以 MNIST 数据集为例，该模型处理数据的步骤如下：

(1) 将图像(形状为 $28 \times 28 \times 1$)输入一个带有 256 个 9×9 卷积核的卷积层 ReLU Conv1。采用步长为 1、无填充(VALID)的方式对其进行卷积操作。输出 256 个通道的特征数据(feature map)。每个特征数据(feature map)的形状为 $20 \times 20 \times 1$ (计算方法: $28 - 9 + 1 = 20$)。



提示:

更全的计算公式可以参考《深度学习之 TensorFlow——入门、原理与进阶实战》一书的 8.4.2 小节中 Padding 的规则介绍。

(2) 将第(1)步的特征输入胶囊网络的主胶囊层，输出带有向量信息的特征结果(具体维度变化见本小节下方的“1. 主胶囊层的工作细节”)。

(3) 将带有向量信息的特征结果输入胶囊网络的数字胶囊层，最终输出分类结果(具体的维度变化见本小节下方的“2. 数字胶囊层的工作细节”)。

胶囊网络中的主胶囊层与卷积神经网络中的卷积层功能类似，而胶囊网络中的数字胶囊层却与卷积神经网络中的池化层功能却有很大不同。具体的不同有以下几点。

1. 主胶囊层的工作细节

主胶囊层的操作沿用了标准的卷积方法。只是在输出时，把多个通道的特征数据(feature map)打包成一个个胶囊单元。将数据按胶囊单元进行后面的计算。

以 MNIST 数据集上的胶囊网络架构为例。在图 8-4 中，主胶囊层的具体处理步骤如下。

(1) 对形状为 $20 \times 20 \times 1$ 的特征图片做步长为 2、无填充(VALID)方式的卷积操作。用 32×8 个 9×9 大小的卷积核，输出 32×8 个通道的特征数据，每个特征数据的形状为 $6 \times 6 \times 1$ ，计算方法为: $(20 - 9 + 1) \div 2 = 6$ 。

(2) 将每个特征图片的形状变换为 $[32 \times 6 \times 6, 1, 8]$ ，该形状可以理解成 $32 \times 6 \times 6$ 个小胶囊，每个胶囊为八维向量，这便是主胶囊的最终输出结果。



提示:

主胶囊层中使用的卷积核大小为 9×9 ，比正常的卷积网络中常用的卷积核尺寸(常用的尺寸有: 1×1 、 3×3 、 5×5 、 7×7)略大，这是为了让生成的特征数据中包含有更多的局部信息。

2. 数字胶囊层的工作细节

在主胶囊与数字胶囊之间，用向量代替标量进行特征传递，使所传递的特征不再是一个具体的数值，而是一个方向加数值的复合信息。这样可以将更多的特征信息传递下去。

例如：向量中的长度表示某一个实例（物体、视觉概念或它们的一部分）出现的概率，方向表示物体的某些图形属性（位置、颜色、方向、形状等）。

具体的计算方式如图 8-5 所示。

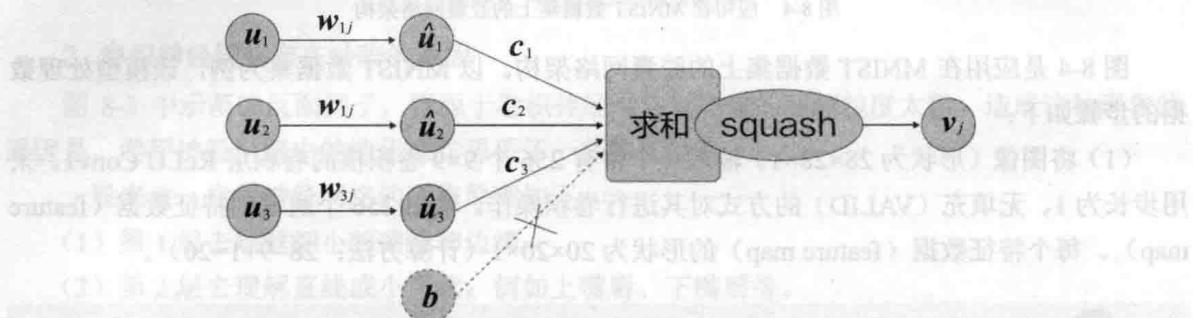


图 8-5 主胶囊与数字胶囊间的特征传递

在图 8-5 中，具体符号含义如下：

- (1) u 代表主胶囊层的输出 (u_1 、 u_2 、 u_3 等，每个代表一个胶囊单元)。
- (2) w 代表权重 (与神经网络中的 w 一致)。
- (3) \hat{u} 代表向量的大小。计算方法为：将 u 中的每个元素与对应的 w 相乘，并将相乘后的结果相加。
- (4) c 代表向量的方向，被称为耦合系数 (coupling coefficients)，也可以理解为权重。它表示每个胶囊数值的重要程度占比，即所有的 c (c_1 、 c_2 、 c_3 等) 相加后的值为 1。

将图 8-5 中的每个 \hat{u} 与其对应的 c 相乘，并将相乘后的结果相加，然后输入激活函数 squash (见本节的“3. 在数字胶囊层中使用全新的激活函数 (squash) ”) 中便得到了数字胶囊的最终输出结果 v_j (下标 j 代表输出的维度)，见式 (8.1)。

$$v_j = \text{squash}(\hat{u}_1 \times c_1 + \hat{u}_2 \times c_2 + \hat{u}_3 \times c_3 \dots) \quad (8.1)$$



提示：

在整个过程中，标准神经网络中的偏置权重 b 已经被去掉了。

还是以 MNIST 数据集上的胶囊网络架构为例。在图 8-3 中，主胶囊与数字胶囊之间的具体处理步骤如下：

- (1) 主胶囊的最终输出 u 为 $32 \times 6 \times 6$ 个胶囊单元。每个胶囊单元为一个 8 维向量。
- (2) 针对每个胶囊单元，定义 10×16 个权重 w 。让每个权重与胶囊单元中的 8 个数相乘，并将相乘后的结果相加。这样，每个胶囊单元由 8 维向量变成了 10×16 维向量。 \hat{u} 的形状变成了 $[32 \times 6 \times 6, 1, 10 \times 16]$ (这里做了优化，让胶囊单元中的 8 个数共享一个权重 w ，这样做可以减小权重 w 的个数。在该步骤如果不做优化，则需要 $8 \times 10 \times 16$ 个权重 w ，即每个胶囊单元中的 8 个数各需要一个权重 w)。

(3) 对 \hat{u} 进行形状变换，将其拆分成 10 份。每份可以理解为一个新的胶囊单元，其形状为 $[32 \times 6 \times 6, 1, 16]$ ，代表该图片在分类中属于标签 0~9 的可能。

(4) 每个新胶囊单元的形状为 $[32 \times 6 \times 6, 1, 16]$ ，可以理解成 \hat{u} 的个数为 $32 \times 6 \times 6$ ，每个 \hat{u} 是一个 16 维向量。

(5) 定义与新胶囊单元同样个数的权重 c ，依次与新胶囊单元中的数值相乘。并按照 $[32 \times 6 \times 6, 1, 16]$ 中的第 0 维度相加，同时将结果放入激活函数 squash 中，见式 (8.1)。得到了胶囊网络的最终输出 v_j （下标 j 代表输出的维度 10×16 ），其形状为 $[10, 16]$ 。

胶囊网络中巧妙地增加了向量的方向 c ，来控制神经元的激活权重。

在实际应用中， c 可以被解释成图像中某个特定实体的各种性质。这些性质可以包含很多种不同的参数，例如姿势（位置、大小、方向）、变形、速度、反射率、色彩、纹理等。而输入输出向量的大小表示某个实体出现的概率，所以它的值必须在 0~1 之间。

3. 在数字胶囊层中使用全新的激活函数 (squash)

因为原有的神经网络模型输出都是标量，显然用处理标量的激活函数来处理向量不太适合。所以有必要为胶囊网络设计一套全新的激活函数 squash，见式 (8.2)。

$$y = \frac{\|x\|^2}{1 + \|x\|^2} \frac{x}{\|x\|} \quad (8.2)$$

该激活函数由两部分组成：第 1 部分为 $\frac{\|x\|^2}{1 + \|x\|^2}$ ，作用是将数值转换成 0~1 之间的数；第 2 部分为 $\frac{x}{\|x\|}$ ，作用是保留原有向量的方向。

二者结合后，会使整个值域变为 -1~1 之间的小数。

该激活函数的图像如图 8-6 所示。

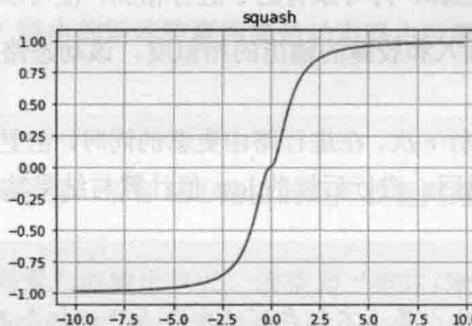


图 8-6 squash 激活函数

如果抛开理论，单纯从输入输出的数值上看，squash 激活函数确实与一般的激活函数没什么区别。而且如果将 squash 激活函数换成一般的激活函数也能够运行。只不过经过大量的实验证明，激活函数 squash 在胶囊网络中的表现确实胜于其他激活函数。这也再次验证了理论的正确性。

4. 利用动态路由选择算法，通过迭代的方式更新耦合系数

主胶囊与数字胶囊之间的耦合系数是通过训练得来的。在训练过程中，耦合系数的更新不是通过反向梯度传播实现的，而是采用动态路由选择算法完成的。该算法来自以下论文链接：

在论文中，列出了动态路由的具体计算方法，一共可分为 7 个步骤，每个步骤的解读如下。

(1) 假设该路由算法发生在胶囊网络的第 l 层，输入值 $\hat{u}_{j|i}$ 为主胶囊网络的输出特征 u_i （下标 i 代表胶囊单元的个数， j 代表每个胶囊单元向量的维数）与权重 w_{ij} 的乘积（见本小节“2. 数字胶囊层的工作细节”中 w 的介绍）。该路由算法需要迭代计算 r 次。

(2) 初始化变量 b_{ij} ，使其等于 0。变量 b_{ij} 与耦合系数 c 具有相同的长度。在迭代时， c 就是由 b 做 softmax 计算得来的。

(3) 让路由算法按照指定的迭代次数 r 进行迭代。

(4) 对变量 b 做 softmax 操作，得到耦合系数 c 。此时耦合系数 c 的值为总和为 1 的百分比小数，即每个权重的概率。 b 与 c 都带了一个下标 i ，表示 b 和 c 的数量各有 i 个，与胶囊单元的个数相同。

提示：

因为第 1 次迭代时， b 的值都为 0，所以第 1 次运行该句时，所有的 c 值也都相同。在后面的步骤中，还会通过计算 b 的值，来不断地修正 c ，从而达到更新耦合系数的作用。

(5) 将 c 与 $\hat{u}_{j|i}$ 相乘，并将乘积的结果相加，得到了数字胶囊 ($l+1$ 层) 的输出向量 s 。

(6) 通过激活函数 squash 对 s 做非线性变换，得到了最终的输出结果 V_j 。

提示：

第(5)、(6)两个步骤一起实现了公式 8-1 中的内容。

(7) 将 V_j 与 $\hat{u}_{j|i}$ 进行点积运算，再与原有的 b 进行相加，便可以求出新的 b 值。其中的点积运算的作用是：计算胶囊的输入和胶囊的输出的相似度。该动态路由协议的原理就是利用相似度来更新 b 值。

将第(4)~(7)步循环执行 r 次。在进行路由更新的同时，也更新了最终的输出结果 V_j 值，当迭代结束后，将最终的 V_j 返回，进行后续的 loss 值计算与结果输出。

提示：

通过该算法可以看出，路由算法不仅在训练中负责优化耦合系数，还在修改耦合系数的同时影响了最终的输出结果。

该模型在训练和测试场景中，都需要做动态路由更新计算。

5. 在胶囊网络中，用边距损失 (margin loss) 作为损失函数

边距损失 (margin loss) 是一种最大化正负样本到超平面距离的算法，见式 (8.3)。

$$L_k = T_k \max(0, m^+ - \|V_k\|)^2 + \lambda(1 - T_k) \max(0, \|V_k\| - m^-)^2 \quad (8.3)$$

其中， L_k 代表损失值， T_k 代表标签， m^+ 代表一个最大值的锚点， m^- 代表一个最小值的锚点， V_k 为模型输出的预测值， λ 为缩放参数。 $\|V_k\|$ 代表取 V_k 的范数，即 $\sqrt{v_1^2 + v_2^2 + v_3^2 + \dots}$ (其中 v_1, v_2, v_3, \dots 代表 V_k 中的元素)。

例如，在MNIST数据集上的胶囊网络架构中，设置了 m^+ 为0.9， m^- 为0.1， λ 为0.5。由于输出值的形状是[10,16]，所以得到的 L_k 形状也是[10,16]。还需要对每个类别的16维向量相加，使其形状变成[10,1]。再取平均值，得到最终的loss值。

由于在最终的输出结果中，每个类别都含有16维特征，所以还可以在其后面加入两层全连接网络，构成一个解码器。用该解码器对输入图片进行重建，并将重建后的损失值与边距损失放在一起进行训练，这样可以得到更好的效果，如图8-7所示。

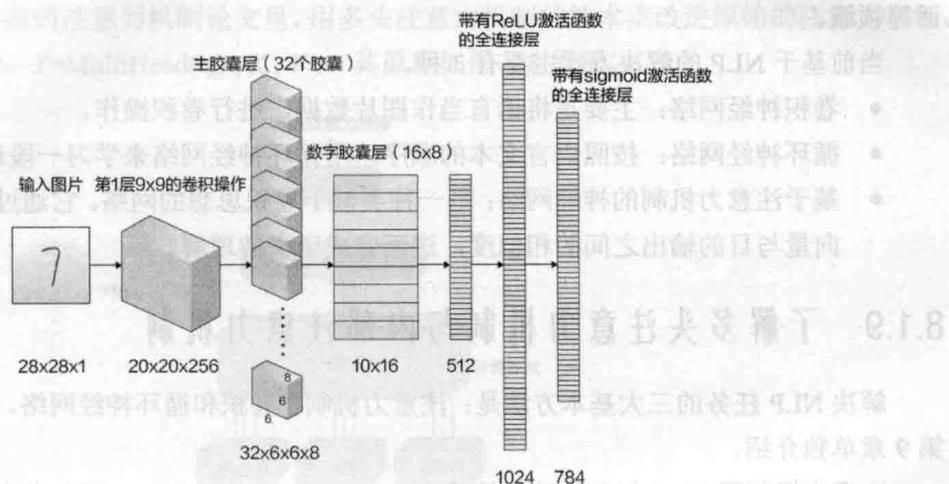


图8-7 带有解码器的胶囊网络结构

8.1.7 了解矩阵胶囊网络与EM路由算法

带有EM（期望最大化）路由的矩阵胶囊网络是动态路由胶囊网络的一个改进版本。论文链接如下：

<https://openreview.net/pdf?id=HJWLfGWRb>

针对动态路由胶囊网络的结构，带有EM路由的矩阵胶囊网络，在各个环节的细节实现上都做了调整。具体如下：

- 将主胶囊由“特征+向量”的输出形式，调整为“矩阵+激活值”的形式。其中矩阵代表图片的姿态矩阵（在某个角度下图片的特征），激活值代表分类结果。
- 用一个代表统一视角的权重矩阵与姿态矩阵相乘得到预测值（论文里叫作投票），即图片的真实特征。
- 在EM路由算法过程中，对预测值（投票）根据投票系数（为每个投票所分配的权重）进行加权计算，并使用加权计算的结果来计算新的投票系数，实现路由更新。
- 论文中的EM算法用修改后的高斯混合模型（简称GMM，是一个基于概率模型的聚类算法）对投票进行聚类。
- 用评估聚类后的信息熵来计算最终分类的激活值。信息熵越小，则表示该类的稳定性越好，该类的结果特征越明显。
- 用Spread损失函数来训练模型。

带有 EM 路由的胶囊网络涉及的算法理论比较多，由于篇幅原因，这里不做展开。在本书 8.2.10 小节提供了代码示例，读者可以自行研究。

8.1.8 什么是 NLP 任务

NLP (Natural Language Processing, 自然语言处理) 是人工智能 (AI) 研究的一个方向。其目标是通过算法让机器能够理解和辨识人类的语言。常用于文本分类、翻译、文本生成、对话等领域。

当前基于 NLP 的解决方式主要有 3 种。

- 卷积神经网络：主要是将语言当作图片数据，进行卷积操作。
- 循环神经网络：按照语言文本的顺序，用循环神经网络来学习一段连续文本中的语义。
- 基于注意力机制的神经网络：是一种类似于卷积思想的网络。它通过矩阵相乘计算输入向量与目的输出之间的相似度，进而完成语义的理解。

8.1.9 了解多头注意力机制与内部注意力机制

解决 NLP 任务的三大基本方法是：注意力机制、卷积和循环神经网络。循神经环网络会在第 9 章单独介绍。

注意力机制因 2017 年谷歌的一篇论文 *Attention is All You Need* 而名声大噪。下面就来介绍该技术的具体内容。如果想了解更多，还可以参考原论文，具体地址如下：

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

1. 注意力机制的基本思想

注意力机制的思想描述起来很简单：将具体的任务看作 query、key、value 三个角色（分别用 q 、 k 、 v 来简写）。其中 q 是要查询的任务，而 k 、 v 是个一一对应的键值对。其目的就是使用 q 在 k 中找到对应的 v 值。

在细节实现时，会比基本原理稍复杂一些，见式 (8.4)。

$$\mathbf{d}_v = \text{Attention}(q_t, k, v) = \text{softmax}\left(\frac{\langle q_t, k_s \rangle}{\sqrt{d_k}}\right) v_s = \sum_{s=1}^m \frac{1}{z} \exp\left(\frac{\langle q_t, k_s \rangle}{\sqrt{d_k}}\right) v_s \quad (8.4)$$

式 8.4 中的 z 是归一化因子。该公式可拆分成以下步骤：

- (1) 将 q_t 与各个 k_s 进行内积计算。
- (2) 将第 (1) 步的结果除以 $\sqrt{d_k}$ ，这里 $\sqrt{d_k}$ 起到调节数值的作用，使内积不至于太大。
- (3) 使用 softmax 函数对第 (2) 步的结果进行计算。
- (4) 使用第 (3) 步的结果与 v_s 相乘，来得到 q_t 与各个 v_s 的相似度。
- (5) 对第 (4) 步的结果加权求和，得到对应的向量 d_v 。

举例：

在中英翻译任务中，假设 K 代表中文，有 m 个词，每个词的词向量是 d_k 维度； V 代表英文，有 m 个词，每个词的词向量是 d_v 维度。

对一句由 n 个中文词组成的句子进行英文翻译时，抛开其他的数值及非线性变化运算，主

要的矩阵间运算可以理解为: $[n, d_k] \times [m, d_k] \times [m, d_v]$ 。将其变形之后得到 $[n, d_k] \times [d_k, m] \times [m, d_v]$, 根据线性代数的技巧, 两个矩阵相乘, 直接把相邻的维度约到剩下的就是结果矩阵的形状。具体做法是, (1) $[n, d_k] \times [d_k, m] = [n, m]$, (2) $[n, m] \times [m, d_v] = [n, d_v]$, 最终便得到了 n 个维度为 d_v 的英文词。

同样, 该模型还可以放在其他任务中, 例如: 在阅读理解任务中, 可以把文章当作 Q , 阅读理解的问题和答案当作 K 和 V 所形成的键值对。

2. 多头注意力机制

在谷歌公司发出的注意力机制论文里, 用多头注意力机制的技术点改进原始的注意力机制。该技术可以表示为: $Y = \text{MultiHead}(Q, K, V)$ 。其原理如图 8-8 所示。

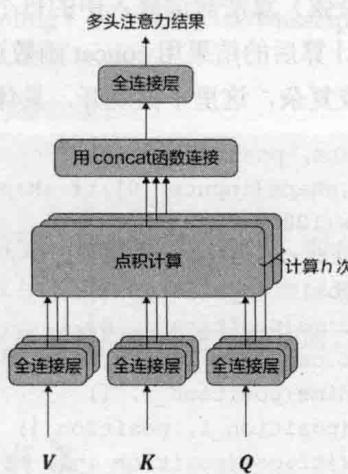


图 8-8 多头注意力机制

图 8-8 所示, 多头注意力机制的工作原理如下:

(1) 把 Q 、 K 、 V 通过参数矩阵进行全连接层的映射转化。

(2) 对第(1)步中所转化的三个结果做点积运算。

(3) 将第(1)步和第(2)步重复运行 h 次, 并且每次进行第(1)步操作时, 都使用全新的参数矩阵 (参数不共享)。

(4) 用 concat 函数把计算 h 次之后的最终结果拼接起来。

其中, 第(4)步的操作与多通道卷积 (见《深度学习之 TensorFlow——入门、原理与进阶实战》中的 8.9.2 小节) 非常相似, 其理论可以解释为:

(1) 每一次的 attention 运算, 都会使原数据中某个方面的特征发生注意力转化 (得到局部注意力特征)。

(2) 当发生多次 attention 运算之后, 会得到更多方向的局部注意力特征。

(3) 将所有的局部注意力特征合并起来, 再通过神经网络将其转化为整体的特征, 从而达到拟合效果。

3. 内部注意力机制

内部注意力机制用于发现序列数据的内部特征。具体做法是将 Q 、 K 、 V 都变成 X 。即 $\text{Attention}(X, X, X)$ 。

使用多头注意力机制训练出的内部注意力特征可以用于 Seq2Seq 模型（输入输出都是序列数据的模型）、分类模型等各种任务，并能够得到很好的效果，即 $Y = \text{MultiHead}(X, X, X)$ 。

8.1.10 什么是带有位置向量的词嵌入

由于注意力机制的本质是 key-value 的查找机制，不能体现出查询时 Q 的内部关系特征。于是，谷歌公司在实现注意力机制的模型中加入了位置向量技术。

带有位置向量的词嵌入是指，在已有的词嵌入技术中加入位置信息。在实现时，具体步骤如下：

- (1) 用 sin（正弦）和 cos（余弦）算法对词嵌入中的每个元素进行计算。
- (2) 将第(1)步中 sin 和 cos 计算后的结果用 concat 函数连接起来，作为最终的位置信息。

关于位置信息的转化公式比较复杂，这里不做展开，具体见以下代码：

```
def Position_Embedding(inputs, position_size):
    batch_size, seq_len = tf.shape(inputs)[0], tf.shape(inputs)[1]
    position_j = 1. / tf.pow(10000., \
                           2 * tf.range(position_size / 2, dtype=tf.float32 \
                                         ) / position_size)
    position_j = tf.expand_dims(position_j, 0)
    position_i = tf.range(tf.cast(seq_len, tf.float32), dtype=tf.float32)
    position_i = tf.expand_dims(position_i, 1)
    position_ij = tf.matmul(position_i, position_j)
    position_ij = tf.concat([tf.cos(position_ij), tf.sin(position_ij)], 1)
    position_embedding = tf.expand_dims(position_ij, 0) \
        + tf.zeros((batch_size, seq_len, position_size))
    return position_embedding
```

在示例代码中，函数 Position_Embedding 的输入和输出分别为：

- 输入参数 inputs 是形状为(batch_size, seq_len, word_size)的张量（可以理解成词向量）。
- 输出结果 position_embedding 是形状为(batch_size, seq_len, position_size)的位置向量。其中，最后一个维度 position_size 中的信息已经包含了位置。

通过函数 Position_Embedding 的输入和输出可以很明显地看到词嵌入中增加了位置向量信息。被转换后的结果，可以与正常的词嵌入一样在模型中被使用。

8.1.11 什么是目标检测任务

目标检测任务是视觉处理中的常见任务。该任务要求模型能检测出图片中特定的物体目标，并获得这一目标的类别信息和位置信息。

在目标检测任务中，模型的输出是一个列表，列表的每一项用一个数据组给出检出目标的类别和位置（常用矩形检测框的坐标表示）。

实现目标检测任务的模型，大概可以分为以下两类。

- 单阶段（1-stage）检测模型：直接从图片获得预测结果，也被称为 Region-free 方法。相关的模型有 YOLO、SSD、RetinaNet 等。

- 两阶段(2-stage)检测模型：先检测包含实物的区域，再对该区域内的实物进行分类识别。相关的模型有R-CNN、Faster R-CNN等。

在实际工作中，两阶段检测模型在位置框方面表现出的精度更高一些，而单阶段模型在分类方面表现出的精度更高一些。

8.5节中将通过一个YOLO V3模型实现目标检测任务。

8.1.12 什么是目标检测中的上采样与下采样

接触过视觉模型源码的读者会发现，在类似 NasNet、Inception Vx、ResNet 这种模型的代码中，会经常出现上采样(upsampling)与下采样(downsampling)这样的函数。它们的意义是什么呢？这里来解释一下。

上采样与下采样是指对图像的缩放操作：

- 上采样是将图像放大。
- 下采样是将图像缩小。

上采样与下采样操作并不能给图片带来更多的信息，而会对图像质量产生影响。在深度卷积网络模型的运算中，通过上采样与下采样操作可实现本层数据与上下层的维度匹配。

在模型以外，用上采样或下采样直接对图片进行操作时，常会使用一些特定的算法，以优化缩放后的图片质量。

8.1.13 什么是图片分割任务

图片分割是对图中的每个像素点进行分类，适用于对像素理解要求较高的场景（例如，在无人驾驶中对道路和非道路进行分割）。

图片分割包括语义分割(semantic segmentation)和实例分割(instance segmentation)，具体如下：

- 语义分割：能将图像中具有不同语义的部分分开。
- 实例分割：能描述出目标的轮廓（比检测框更为精细）。

目标检测、语义分割、实例分割三者的关系如图 8-9 所示。

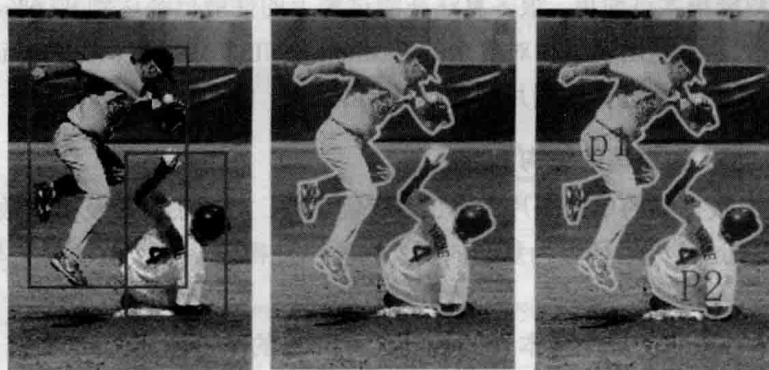


图 8-9 图片分割任务

在图 8-9 中，3 个子图的意义如下：

- 图 8-9 (a) 是目标检测的结果，该任务是在原图上找到目标物体的矩形框（见本章 8.5 节 YOLO V3 模型的实例）。
- 图 8-9 (b) 是语义分割的结果，该任务是在原图上找到目标物体所在的像素点（见本章 8.7 节 Mask R-CNN 模型的实例）。
- 图 8-9 (c) 是实例分割的结果，该任务在语义分割的基础上还要识别出单个的具体个体。

8.2 实例 39：用胶囊网络识别黑白图中服装的图案

实现一个带有路由算法的胶囊网络模型，并用该模型来解决实际问题。

实例描述

从 Fashion-MNIST 数据集中选择一幅图，这幅图上有 1 个服装图案。让机器模拟人眼来区分这个服装图案到底是什么。

实例中所用的图片来源于一个开源的训练数据集——Fashion-MNIST。

8.2.1 熟悉样本：了解 Fashion-MNIST 数据集

Fashion-MNIST 数据集常被用来测试模型。一般来讲，如果在 Fashion-MNIST 数据集上没有实现显著效果的模型，则在其他数据集上也不会有好的效果。

1. Fashion-MNIST 的起源

Fashion-MNIST 数据集是 MNIST 数据集的一个替代品。

MNIST 是一个入门级的计算机视觉数据集，是在 Fashion-MNIST 数据集出现之前人们最常使用的实验数据集。相当于学习编程过程中的打印“Hello World”操作。经典的 MNIST 数据集包含了大量手写数字。在《深度学习之 TensorFlow——入门、原理与进阶实战》一书中，大量使用该数据集来验证模型及阐述原理。

由于 MNIST 数据集太过简单，很多算法在测试集上的性能已经达到 99.6%，但是应用在真实图片上却相差很大。于是出现了相对复杂的 Fashion-MNIST 数据集。在 Fashion-MNIST 数据集上训练好的模型，会更接近真实图片的处理效果。

2. Fashion-MNIST 数据集的结构

Fashion-MNIST 数据集的单张图片大小、训练集个数、测试集个数及类别数，与 MNIST 数据集完全相同。只不过其采用了更为复杂的图片内容，使得做基础实验的模型与真实环境下的模型更加相近。

FashionMNIST 数据集的单个样本为 28 pixel×28 pixel 的灰度图片。训练集有 60000 张图片，测试集有 10000 张图片。样本内容为上衣、裤子、鞋子等服装，一共分为 10 类，如图 8-10 所示（每个类别占三行）。

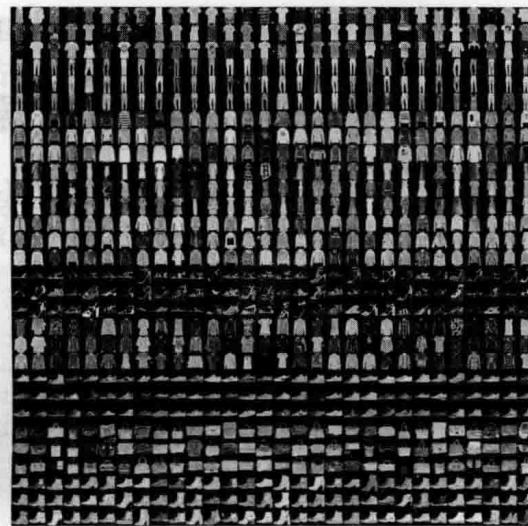


图 8-10 Fashion-MNIST 中的内容

FashionMNIST 数据集分类标签的标注编号仍然是 0~9，其代表的服装类别如图 8-11 所示。

标注编号	描述
0	T-shirt/top (T恤)
1	Trouser (裤子)
2	Pullover (套衫)
3	Dress (裙子)
4	Coat (外套)
5	Sandal (凉鞋)
6	Shirt (汗衫)
7	Sneaker (运动鞋)
8	Bag (包)
9	Ankle boot (踝靴)

图 8-11 Fashion-MNIST 中的标签

8.2.2 下载 Fashion-MNIST 数据集

Fashion-MNIST 数据集的官网下载链接如下：

<https://github.com/zalandoresearch/fashion-mnist>

打开官网，可以看到如图 8-12 所示的下载链接。

Name	Content	Examples	Size	Link	MD5 Checksum
train-images-idx3-ubyte.gz	training set images	60,000	26 MBytes	Download	8d4fb7e6c68d591d4c3dfe9ec88bf0d
train-labels-idx1-ubyte.gz	training set labels	60,000	29 Kbytes	Download	25c81989df183df01b3e8a0aad5dffbe
t10k-images-idx3-ubyte.gz	test set images	10,000	4.3 MBytes	Download	bef4ecab320f06d8554ea6380940ec79
t10k-labels-idx1-ubyte.gz	test set labels	10,000	5.1 Kbytes	Download	bb300cfdad3c16e7a12a480ee83cd310

图 8-12 Fashion-MNIST 数据集的下载链接

将数据集下载后，不需要解压缩，直接放到代码的同级目录下面即可。

8.2.3 代码实现：读取及显示 Fashion-MNIST 数据集中的数据

TensorFlow 提供了一个加载及读取 MNIST 数据集的库，可以直接使用该库来加载和读取 Fashion-MNIST 数据集。使用该库时，不需要修改任何代码，直接指定路径即可。

具体代码如下：

代码 8-1 读取 Fashion-MNIST 数据集

```
01 from tensorflow.examples.tutorials.mnist import input_data
02 mnist = input_data.read_data_sets("./fashion/", one_hot=False) #指定数据集
03 print ('输入数据:',mnist.train.images)
04 print ('输入数据的形状:',mnist.train.images.shape)
05 print ('输入数据的标签:',mnist.train.labels)
06
07 import pylab
08 im = mnist.train.images[1]
09 im = im.reshape(-1,28)
10 pylab.imshow(im)
11 pylab.show()
```

将数据集文件都放在本地同级目录下的 fashion 文件夹里，再在代码中指定路径（见代码第 2 行）。

运行代码，输出以下信息：

```
Extracting ./fashion/train-images-idx3-ubyte.gz
Extracting ./fashion/train-labels-idx1-ubyte.gz
Extracting ./fashion/t10k-images-idx3-ubyte.gz
Extracting ./fashion/t10k-labels-idx1-ubyte.gz
输入数据: [[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
输入数据的形状: (55000, 784)
输入数据的标签: [4 0 7 ... 3 0 5]
```

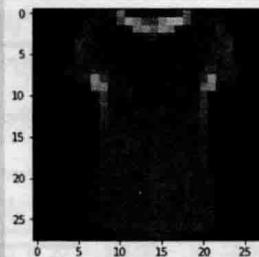


图 8-13 Fashion-MNIST 数据集中的一张图片

输出信息的前4行是解压缩数据集的操作。

从输出信息的第5行开始是训练集中的图片数据：一个55000行、784列的矩阵。

在矩阵中，每一行表示一张图片，即训练集里面有55000张图片。每一张图片是 28×28 的矩阵。

在输出信息的中括号里可以看到每个矩阵的值，每一个值代表一个像素值。



提示：

图片上的像素点与矩阵中的像素值之间的关系如下：

- 如果是1通道的黑白图片，则图片中黑色的地方像素值是0；有图案的地方像素值为1~255之间的数字，代表颜色的深度。
- 如果是3通道的彩色图片，则图片上的每一个像素点由3个像素值来表示，即R、G、B（红、黄、蓝）。这3个像素值分布在3个通道里。

1. 在tf.keras接口中读取Fashion_MNIST数据集

tf.keras接口中已经集成了Fashion_MNIST数据集，使用起来比原生的TensorFlow方式更为简单。代码如下：

```
import tensorflow as tf
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
```

上面代码运行后，便会得到Fashion-MNIST的训练集数据(X_{train} , y_{train})与测试集数据(X_{test} , y_{test})。

8.2.4 代码实现：定义胶囊网络模型类CapsuleNetModel

定义类CapsuleNetModel来实现胶囊网络模型，并在类CapsuleNetModel中定义模型相关的参数。

具体代码如下：

代码8-2 Capsulemodel

```
01 import tensorflow as tf
02 import tensorflow.contrib.slim as slim
03 import numpy as np
04
05 class CapsuleNetModel: # 定义胶囊网络模型类
06     def __init__(self, batch_size, n_classes, iter_routing): # 初始化
07         self.batch_size = batch_size
08         self.n_classes = n_classes
09         self.iter_routing = iter_routing
```

8.2.5 代码实现：实现胶囊网络的基本结构

在 CapsuleNetModel 类中定义 CapsuleNet 方法，并用 TF-slim 接口实现胶囊网络的基本结构。

该步骤与 8.1.7 小节的描述完全一致。其中， \hat{u} 的计算方法是通过卷积核为 [1,1] 的卷积操作来实现的。



提示：

代码第 74 行，在实现 squash 激活函数的过程中，分母部分加了一个常量 “1e-9”。这是与 8.1.7 小节 squash 公式的不同之处。

常量 “1e-9” 是一个很小的数，它接近于 0 却不等于 0。该值的意义是防止分母为 0 导致公式无意义。

具体代码如下：

代码 8-2 CapsuleModel（续）

```

10     def CapsuleNet(self, img):          # 定义模型结构
11         # 定义第 1 个正常卷积层
12         with tf.variable_scope('Conv1_layer') as scope:
13             output = slim.conv2d(img, num_outputs=256, kernel_size=[9, 9],
14 stride=1, padding='VALID', scope=scope)
15         assert output.get_shape() == [self.batch_size, 20, 20, 256]
16         # 定义主胶囊网络
17         with tf.variable_scope('PrimaryCaps_layer') as scope:
18             output = slim.conv2d(output, num_outputs=32*8, kernel_size=[9, 9],
19 stride=2, padding='VALID', scope=scope, activation_fn=None)
20             # 将结果变成 32×6×6 个胶囊单元，每个单元为 8 维向量
21             output = tf.reshape(output, [self.batch_size, -1, 1, 8])
22             assert output.get_shape() == [self.batch_size, 1152, 1, 8]
23         # 定义数字胶囊网络
24         with tf.variable_scope('DigitCaps_layer') as scope:
25             u_hats = []
26             # 将输入按照胶囊单元分开
27             input_groups = tf.split(axis=1, num_or_size_splits=1152,
28 value=output)
28             for i in range(1152): # 遍历每个胶囊单元
29                 # 利用卷积核为 [1,1] 的卷积操作，让  $u$  与  $w$  相乘，再相加得到  $\hat{u}$ 
30                 one_u_hat = slim.conv2d(input_groups[i], num_outputs=16*10,
31 kernel_size=[1, 1], stride=1, padding='VALID',
32 scope='DigitCaps_layer_w_'+str(i), activation_fn=None)
33                 # 每个胶囊单元变成了 16 维向量
34                 one_u_hat = tf.reshape(one_u_hat, [self.batch_size, 1, 10,
35 16])

```

```

32         u_hats.append(one_u_hat)
33     #将所有的胶囊单元中的one_u_hat合并起来
34     u_hat = tf.concat(u_hats, axis=1)
35     assert u_hat.get_shape() == [self.batch_size, 1152, 10, 16]
36
37     #初始化b值
38     b_ijs = tf.constant(np.zeros([1152, 10], dtype=np.float32))
39     v_js = []
40     for r_iter in range(self.iter_routing):#指定循环次数，计算动态路由
41         with tf.variable_scope('iter_'+str(r_iter)):
42             c_ij_groups = tf.split(axis=1, num_or_size_splits=10,
43                                     value=c_ijs)
44
45             b_ij_groups = tf.split(axis=1, num_or_size_splits=10,
46                                     value=b_ijs)
47             u_hat_groups = tf.split(axis=2, num_or_size_splits=10,
48                                     value=u_hat)
49
50             for i in range(10):
51                 #生成具有跟输入一样尺寸的卷积核[1152, 1]，输入为16通道，卷积核个数为1个
52                 c_ij = tf.reshape(tf.tile(c_ij_groups[i], [1, 16]),
53                                   [1152, 1, 16, 1])
54
55                 #利用深度卷积实现u_hat与c矩阵的对应位置相乘，输出的通道数为16×1个
56                 s_j = tf.nn.depthwise_conv2d(u_hat_groups[i], c_ij,
57                                               strides=[1, 1, 1, 1], padding='VALID')
58
59                 assert s_j.get_shape() == [self.batch_size, 1, 1, 16]
60
61                 s_j = tf.reshape(s_j, [self.batch_size, 16])
62                 v_j = self.squash(s_j)#调用激活函数squash生成最终结果v_j
63
64                 assert v_j.get_shape() == [self.batch_size, 16]
65                 #根据v_j来计算并更新b值
66                 b_ij_groups[i] =
67
68                 b_ij_groups[i]+tf.reduce_sum(tf.matmul(tf.reshape(u_hat_groups[i],
69                                           [self.batch_size, 1152, 16]), tf.reshape(v_j, [self.batch_size, 16, 1])), axis=0)
70
71
72             #迭代结束后，再生成一次v_j，得到数字胶囊真正的输出结果
73             if r_iter == self.iter_routing-1:
74                 v_js.append(tf.reshape(v_j, [self.batch_size, 1,
75                                              16]))
76
77             #将10类的b合并一起
78             b_ijs = tf.concat(b_ij_groups, axis=1)
79
80             #将10类的v_j合并到一起，生成的形状为[self.batch_size, 10, 16]的结果
81             output = tf.concat(v_js, axis=1)

```

```

70
71     return output
72 def squash(self, s_j): #定义激活函数
73     s_j_norm_square = tf.reduce_mean(tf.square(s_j), axis=1,
74                                     keepdims=True)
74     v_j =
75     s_j_norm_square*s_j/((1+s_j_norm_square)*tf.sqrt(s_j_norm_square+1e-9))
75     return v_j

```

在代码第 53 行，用深度卷积操作实现 \hat{a} 与 c 矩阵的对应位置相乘。

代码第 40 行是动态路由算法的实现，在路由计算的迭代最后一次时，还需要将最终的结果保存起来，作为整个网络的输出（见代码第 64 行）。函数 CapsuleNet 执行完，最终会把数字胶囊的结果返回（见代码第 69 行）。



提示：

代码第 51 行使用了 `tf.tile` 函数。该函数具有扩充矩阵的作用，即将张量内容按照指定维度进行复制。这是利用矩阵优化 `for` 循环计算速度的一种常用方法。为了更详细地解释，请看以下代码。

【代码 1】：用循环方式让 m_2 中的值与 m_1 中的值依次相乘。

```

m1 = tf.constant([[1],[2],[3]])      #被乘数
m2 = tf.constant([2])                #乘数（相当于代码第 51 行的权重 c_ij_groups[i]）
m1_sz = tf.unstack(a)               #将被乘数拆成列表
m_resurt = []                      #定义列表收集结果
for i in m1_sz :                   #循环相乘
    t = m2*i
    m_resurt.append(t)              #将结果加入列表中
resurt1 = tf.stack(m_resurt)        #在重新组合回张量
with tf.Session() as sess:
    print("resurt1",resurt1.eval())  #输出结果[[2][4][6]]

```

【代码 2】：用 `tile` 方式让 m_2 中的值与 m_1 中的值依次相乘。

```

m1 = tf.constant([[1],[2],[3]])      #被乘数
m2 = tf.constant([2])                #乘数（相当于代码第 51 行的权重 c_ij_groups[i]）
m2 = tf.expand_dims(m2,1)            #增加一个维度
u_tile = tf.tile(m2,[3,1])          #复制成与乘数相同的份数
resurt2 = u_tile*m1                #直接数组相乘
with tf.Session() as sess:
    print("resurt2",resurt2.eval())  #输出结果[[2][4][6]]

```

二者的结果是一样的。但是代码 2 使用 `tile` 方式省去了循环，提升了效率。

8.2.6 代码实现：构建胶囊网络模型

在 CapsuleNetModel 类中，定义 build_model 方法来构建胶囊网络模型。具体实现步骤如下：

- (1) 将张量图重置。
- (2) 用 CapsuleNet 方法构建网络节点。
- (3) 对 CapsuleNet 方法返回的结果进行范数计算，得到分类结果 self.v_len。
- (4) 在训练模式下，添加解码器网络，重建输入图片。
- (5) 实现 loss 方法，将边距损失与重建损失放在一起，生成总的损失值。
- (6) 将损失值放到优化器中，生成张量操作符 train_op，用于训练（代码第 110 行）。

完整代码如下：

代码 8-2 Capsulemodel (续)

```

76     def build_model(self, is_train=False, learning_rate = 1e-3):
77         tf.reset_default_graph()
78
79         # 定义占位符
80         self.y = tf.placeholder(tf.float32, [self.batch_size,
81             self.n_classes])
81         self.x = tf.placeholder(tf.float32, [self.batch_size, 28, 28, 1],
82             name='input')
83
84         # 定义计步器
85         self.global_step = tf.Variable(0, name='global_step',
86             trainable=False)
87         initializer = tf.truncated_normal_initializer(mean=0.0,
88             stddev=0.01)
89         biasInitializer = tf.constant_initializer(0.0)
90
91         with slim.arg_scope([slim.conv2d], trainable=is_train,
92             weights_initializer=initializer, biases_initializer=biasInitializer):
93             self.v_jsoutput = self.CapsuleNet(self.x) # 构建胶囊网络模型
94
95             with tf.variable_scope('Masking'):
96                 # 计算输出值的欧几里得范数 [self.batch_size, 10]
97                 self.v_len = tf.norm(self.v_jsoutput, axis=2)
98
99                 if is_train:                                # 如果是训练模式，则重建输入图片
100                     masked_v = tf.matmul(self.v_jsoutput, tf.reshape(self.y, [-1, 10,
101                         1]), transpose_a=True)
102                     masked_v = tf.reshape(masked_v, [-1, 16])
103
104                     with tf.variable_scope('Decoder'):
105                         output = slim.fully_connected(masked_v, 512,
106                             trainable=is_train)

```

```

102         output = slim.fully_connected(output, 1024,
103             trainable=is_train)
104         self.output = slim.fully_connected(output, 784,
105             trainable=is_train, activation_fn=tf.sigmoid)
106         # 使用退化学习率
107         learning_rate_decay = tf.train.exponential_decay(learning_rate,
108             global_step=self.global_step, decay_steps=1000, decay_rate=0.9)
109         # 定义优化器
110         self.train_op =
111             tf.train.AdamOptimizer(learning_rate_decay).minimize(self.total_loss,
112             global_step=self.global_step)
113         # 定义保存及恢复模型关键点要用的 saver
114         self.saver = tf.train.Saver(tf.global_variables(), max_to_keep=1)
115     def loss(self, v_len, output): # 定义 loss 值的计算函数
116         max_l = tf.square(tf.maximum(0., 0.9-v_len))
117         max_r = tf.square(tf.maximum(0., v_len - 0.1))
118
119         l_c = self.y * max_l + 0.5 * (1 - self.y) * max_r
120
121         margin_loss = tf.reduce_mean(tf.reduce_sum(l_c, axis=1))
122
123         origin = tf.reshape(self.x, shape=[self.batch_size, -1])
124         reconstruction_err = tf.reduce_mean(tf.square(output-origin))
125         # 将边距损失与重建损失一起构成 loss 值
126         total_loss = margin_loss + 0.0005 * reconstruction_err
127
128     return total_loss

```



提示：

在 build_model 方法中，一定要将 saver 的定义放在最后（见代码第 113 行），否则在 saver 后面的张量将无法保存到检查点文件中。因为 saver 的第 1 个参数为 tf.global_variables()，该函数只能载入之前定义的张量，后来定义的张量无法被载入。

8.2.7 代码实现：载入数据集，并训练胶囊网络模型

搭建胶囊网络模型，并载入 Fashion-MNIST 数据集，开始训练。

定义函数 save_images 与 mergeImgs，用于将模型的输出结果可视化。

完整代码如下：

代码 8-3 用胶囊网络识别黑白图中的服装图案

```

01 import tensorflow as tf
02 import time
03 import os
04 import numpy as np
05 import imageio
06
07 Capsulemodel = __import__("8-2_Capsulemodel")
08 CapsuleNetModel = Capsulemodel.CapsuleNetModel
09
10 #载入数据集
11 from tensorflow.examples.tutorials.mnist import input_data
12 mnist = input_data.read_data_sets("./fashion/", one_hot=True)
13
14 def save_images(imgs, size, path): #定义函数,保存图片
15     imgs = (imgs + 1.) / 2
16     return(imageio.imwrite(path, mergeImgs(imgs, size)))
17
18 def mergeImgs(images, size): #定义函数,合并图片
19     h, w = images.shape[1], images.shape[2]
20     imgs = np.zeros((h * size[0], w * size[1], 3))
21     for idx, image in enumerate(images):
22         i = idx % size[1]
23         j = idx // size[1]
24         imgs[j * h:j * h + h, i * w:i * w + w, :] = image
25         imgs[j * h:j * h + h, i * w:i * w + w, :] = image
26     return imgs
27
28 batch_size = 128 #定义批次
29 learning_rate = 1e-3 #定义学习率
30 training_epochs = 5 #数据集迭代次数
31 n_class = 10
32 iter_routing = 3 #定义胶囊网络中动态路由的训练次数

```

代码中的第 28~32 行是训练参数的定义。



提示:

胶囊网络中的权重参数比较多,会占用很大的 GPU 显存。如果在配置较低的机器上运行,则可以将批次变量 batch_size 改小一些(见代码第 28 行)。

8.2.8 代码实现: 建立会话训练模型

建立会话训练模型是在 main 函数中完成操作,具体步骤如下:

- (1) 实例化胶囊网络模型类 CapsuleNetModel。
- (2) 建立会话。

(3) 在会话中，用循环进行迭代训练。

完整具体代码如下：

代码 8-3 用胶囊网络识别黑白图中的服装图案（续）

```

33 def main(_):
34     #实例化模型
35     capsmodel = CapsuleNetModel(batch_size, n_class, iter_routing)
36     #构建网络节点
37     capsmodel.build_model(is_train=True, learning_rate=learning_rate)
38     os.makedirs('results', exist_ok=True)          #创建路径
39     os.makedirs('./model', exist_ok=True)
40
41     with tf.Session() as sess:                  #建立会话
42         sess.run(tf.global_variables_initializer())
43
44     #载入检查点文件
45     checkpoint_path = tf.train.latest_checkpoint('./model/')
46     print("checkpoint_path", checkpoint_path)
47     if checkpoint_path !=None:
48         capsmodel.saver.restore(sess, checkpoint_path)
49     history = []                                #收集 loss 值
50     for epoch in range(training_epochs):          #按照指定次数迭代数据集
51
52         total_batch = int(mnist.train.num_examples/batch_size)
53         lossvalue= 0                             #存放当前 loss 值
54         for i in range(total_batch):              #遍历数据集
55             batch_x, batch_y = mnist.train.next_batch(batch_size) #取数据
56             batch_x = np.reshape(batch_x,[batch_size, 28, 28, 1])
57
58             tic = time.time()                      #计算运行时间
59             _, loss_value = sess.run([capsmodel.train_op,
60                                         capsmodel.total_loss], feed_dict={capsmodel.x: batch_x, capsmodel.y:
61                                         batch_y})
62             lossvalue +=loss_value                 #累计 loss 值
63             if i % 20 == 0:                      #每训练 20 次，输出 1 次结果
64                 print(str(i)+'用时: '+str(time.time()-tic)+ ' loss:
65                 ', loss_value)
66                 cls_result, recon_imgs = sess.run([capsmodel.v_len,
67                                         capsmodel.output], feed_dict={capsmodel.x: batch_x, capsmodel.y:
68                                         batch_y})
69                 imgs = np.reshape(recon_imgs, (batch_size, 28, 28, 1))
70                 size = 6
71                 save_images(imgs[0:size * size, :], [size, size],
72 'results/test_%03d.png' % i)           #将结果保存为图片
73                 #获得分类结果，评估准确率
74                 argmax_idx = np.argmax(cls_result, axis= 1)
75                 batch_y_idx = np.argmax(batch_y, axis= 1)

```

```

70 6-4-capsnet.py    cls_acc = np.mean(np.equal(argmax_idx,
71                                batch_y_idx).astype(np.float32))
72 import tensorflow as tf
73         print('正确率：' + str(cls_acc * 100))
74         history.append(lossvalue/total_batch)      #保存本次迭代的 loss 值
75         if lossvalue/total_batch == min(history):#如果 loss 值变小，保存模型
76             ckpt_path = os.path.join('./model', 'model.ckpt')
77             capsmodel.saver.save(sess, ckpt_path,
78             global_step=capsmodel.global_step.eval())#生成检查点文件
79             print("save model", ckpt_path)
80         print(epoch, lossvalue/total_batch)
81 if __name__ == "__main__":
82     tf.app.run()

```

8.2.9 运行程序

直接运行代码文件“8-3 使用胶囊网络识别黑白图中的服装图案.py”。输出以下结果：

```

Extracting ./fashion/train-images-idx3-ubyte.gz
Extracting ./fashion/train-labels-idx1-ubyte.gz
Extracting ./fashion/t10k-images-idx3-ubyte.gz
Extracting ./fashion/t10k-labels-idx1-ubyte.gz
checkpoint_path None
0用时: 33.89296865463257 loss: 0.7986926
正确率: 11.71875
20用时: 0.5990476608276367 loss: 0.5816276
正确率: 9.375
.....
420用时: 2.351250648498535 loss: 0.16442175
正确率: 83.59375
1 0.15774308
.....

```

从输出结果中可以看出，整个数据集迭代训练1次后，模型的正确率是83.5%。

在程序运行时，本地的result文件夹下会生成一些结果图片，如图8-14所示。

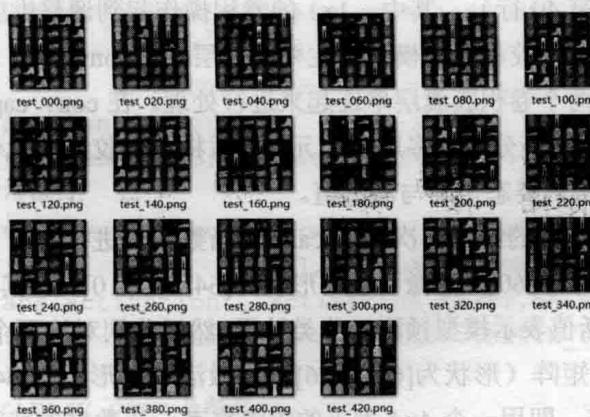


图8-14 胶囊网络模型重建后的输出结果

图 8-14 中的图片文件是胶囊网络重建后的输出结果。

可以看到，胶囊网络模型重建后的图片与原有的样本文件几乎相同。

8.2.10 实例 40：实现带有 EM 路由的胶囊网络

EM 胶囊网络模型的结构由以下部分组成：

- ReLU 卷积层。
- 主胶囊层（PrimaryCaps）。
- 若干个卷积胶囊层（ConvCaps）。
- 分类胶囊层（Class Capsules）。

在本实例中使用了两个卷积胶囊层，如图 8-15 所示。

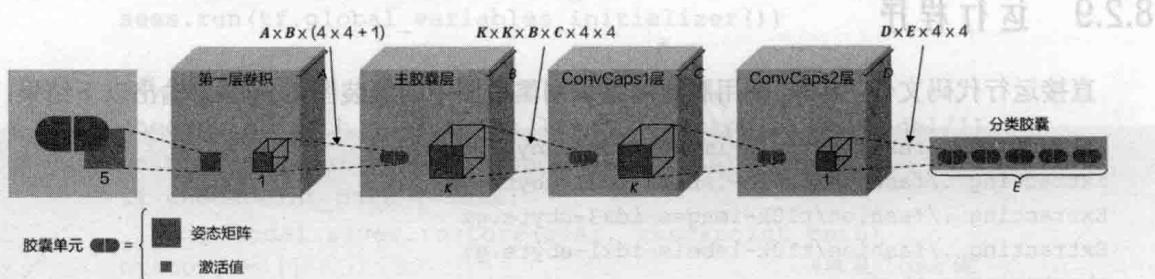


图 8-15 EM 路由胶囊网络模型的结构

下面介绍一下该实例中的主要代码。

1. 实现 EM 胶囊网络模型的主体结构

定义函数 build_em，按照图 8-15 所示的结构实现 EM 胶囊网络模型的主体结构。

在 build_em 中，网络的每一层将按照指定好的维度（见代码第 10 行）进行输出。每层具体的操作如下所示。

- ReLU 卷积层：使用一个 5×5 的卷积核进行卷积操作。
- 主胶囊层：用 1×1 的卷积核对上层的输出进行两次卷积操作，生成姿态矩阵与激活值（见代码第 27~第 40 行）。其中， 1×1 的卷积操作起到调整维度的作用。
- 卷积胶囊层：是 EM 胶囊网络模型的主要工作层，由 conv_caps 函数实现（见代码第 75 行）。本实例将两个卷积胶囊层串连起来进行处理。在 conv_caps 函数中，先调用函数 kernel_tile 将原始特征分成更多胶囊单元，然后将多个胶囊单元传入函数 calEM 中进行 EM 路由计算，得到姿态矩阵与激活值。
- 分类胶囊层：将上层的结果再次输入 calEM 函数中，进行 EM 路由计算，得到姿态矩阵（形状为 $[64, 3, 3, 160]$ ）与激活值（形状为 $[64, 3, 3, 10]$ ）。其中，姿态矩阵表示数据的特征向量，激活值表示模型预测的分类结果。然后分别对这两个结果做全局平均池化，得到最终的姿态矩阵（形状为 $[64, 10, 16]$ ）与激活值（形状为 $[64, 10]$ ）。姿态矩阵的最后一个维度是 16，即用一个 4×4 大小的矩阵来表示该数据的特征向量。

完整代码如下：

代码 8-4 capsnet_em

```

01 import tensorflow as tf
02 import tensorflow.contrib.slim as slim
03 import numpy as np
04
05 weight_reg = False          #是否使用参数正则化
06 epsilon=1e-9                #防止分母为 0 的最小数
07 iter_routing=2              #EM 算法的迭代次数
08
09 def build_em(input, batch_size, is_train: bool, num_classes: int):
10     A,B,C,D=32,8,16,16        #定义各层的输出维度
11
12     data_size = int(input.get_shape()[1])#输入尺寸为 28x28
13     bias_initializer = tf.truncated_normal_initializer( mean=0.0,
14     stddev=0.01)
15     #定义 12 正则层的参数
16     weights_regularizer = tf.contrib.layers.l2_regularizer(5e-04)
17     #输出形状: (?, 28, 28, 1)
18     tf.logging.info('input shape: {}'.format(input.get_shape()))
19     #为卷积权重统一初始化
20     with slim.arg_scope([slim.conv2d], trainable=is_train,
21         biases_initializer=bias_initializer,
22         weights_regularizer=weights_regularizer):
23         with tf.variable_scope('relu_conv1') as scope: #relu_conv1 层
24             output = slim.conv2d(input, num_outputs=A, kernel_size=[
25                 5, 5], stride=2, padding='VALID', scope=scope,
26             activation_fn=tf.nn.relu)
27             data_size = int(np.floor((data_size - 5+1) / 2))#计算卷积后的尺寸,
28             得到 12
29             tf.logging.info('conv1 output shape:
30             {}'.format(output.get_shape()))           #输出(?, 12, 12, 32)
31
32             with tf.variable_scope('primary_caps') as scope: #primary_caps 层
33                 pose = slim.conv2d(output, num_outputs=B * 16,   #计算姿态矩阵
34                     kernel_size=[1, 1], stride=1, padding='VALID',
35                     scope=scope, activation_fn=None)
36                 pose = tf.reshape(pose, [batch_size, data_size, data_size, B, 16])
37                 #计算激活值
38                 activation = slim.conv2d(output, num_outputs=B, kernel_size=[
39                     1, 1], stride=1, padding='VALID',
40                     scope='primary_caps/activation', activation_fn=tf.nn.sigmoid)
41                 activation = tf.reshape(activation, [batch_size, data_size,
42                     data_size, B, 1])
43                 #计算 primary_caps 层输出
44                 output = tf.concat([pose, activation], axis=4)
45

```

```

37     output = tf.reshape(output, shape=[batch_size, data_size,
38                             data_size, -1])
39     assert output.get_shape()[1:] == [data_size, data_size, B * 17]
40     tf.logging.info('primary capsule output shape: {}'.
41                      format(output.get_shape())) #形状为(batch_size, 12, 12, 136)
42
43     with tf.variable_scope('conv_caps1') as scope: #conv_caps1层
44         pose, activation =
45         conv_caps(output, 3, 2, C, weights_regularizer, "conv cap 1")
46         data_size = pose.get_shape()[1]
47         output = tf.reshape(tf.concat([pose, activation], axis=4), [
48             batch_size, data_size, data_size, C*17])
49         tf.logging.info('conv cap 1 output shape: {}'.
50                         format(output.get_shape()))
51
52     with tf.variable_scope('conv_caps2') as scope:
53         pose, activation =
54         conv_caps(output, 3, 1, D, weights_regularizer, "conv cap 2")
55         data_size = activation.get_shape()[1]
56
57     with tf.variable_scope('class_caps') as scope:
58         pose = tf.reshape(pose, [-1, D, 16]) #调整形状
59         activation = tf.reshape(activation, [-1, D, 1])
60         #计算EM, 获得姿态矩阵和激活值
61         miu, activation =
62         calEM(pose, activation, num_classes, weights_regularizer, "class cap")
63         #调整形状
64         activation = tf.reshape(activation, [batch_size, data_size,
65                                         data_size, num_classes])
66         miu = tf.reshape(miu, [batch_size, data_size, data_size, -1])
67
68         output = tf.nn.avg_pool(activation, ksize=[1, data_size, data_size,
69                                 1], strides=[1, 1, 1, 1], padding='VALID')
70         #最终分类结果
71         output = tf.reshape(output, [batch_size, num_classes])
72         tf.logging.info('class caps : {}'.format(output.get_shape()))
73
74     pose = tf.nn.avg_pool(miu, ksize=[1, data_size, data_size, 1],
75                           strides=[1, 1, 1, 1], padding='VALID') #获得每一类的最终特征
76     pose_out = tf.reshape(pose, shape=[batch_size, num_classes, 16])
77
78     return output, pose_out
79 #卷积胶囊层
80 def conv_caps(indata, kernel, stride, outputdim, weights_regularizer, name):

```

```

76     batch_size = int(indata.get_shape()[0])
77     data_size = int(indata.get_shape()[1])      #获得输入尺寸(默认h和w相等)
78     output = kernel_tile(indata, kernel, stride) #将主胶囊层的输出分成9个特征
79     data_size = int(np.floor((data_size - kernel+1) / stride))#计算卷积尺寸
80
81     newbatch = batch_size * data_size * data_size
82     #将output的形状变成[newbatch,kernel * kernel * 上层维度, 17]
83     output = tf.reshape(output, shape=[newbatch, -1, 17])
84     activation = tf.reshape(output[:, :, 16], [newbatch, -1, 1])
85
86     miu, activation =
87     calEM(output[:, :, :16], activation, outputdim, weights_regularizer, name)
88
89     #生成姿态矩阵
90     pose = tf.reshape(miu, [batch_size, data_size, data_size, outputdim, 16])
91     tf.logging.info('{} pose shape: {}'.format(name, pose.get_shape()))
92
93     #生成激活
94     activation = tf.reshape(activation, [batch_size, data_size,
95         data_size, outputdim, 1])
96     tf.logging.info('{} activation shape: {}'.format(name, activation.get_shape()))
97
98     return pose, activation

```

代码第78行，在conv_caps函数里，用kernel_tile函数将原始特征拆分成9个小特征。该操作可以得到更多的候选胶囊，用于输入calEM函数（在本节“3. 实现calEM函数”中会介绍）进行EM运算。

有关kernel_tile函数的实现，请参考下面的“2. 实现kernel_tile函数”。

2. 实现kernel_tile函数

函数kernel_tile的作用是，对原始特征按照指定的间隔和尺度来抽样提取。该函数可以将原始特征拆分成更多的候选胶囊。

在kernel_tile函数中用深度卷积（见8.1.4小节）操作对原始特征进行计算。具体步骤如下：

(1) 定义了一个深度卷积的卷积核。其size是[kernel, kernel]，输入的通道数是input_shape[3]，卷积核个数为kernel×kernel（见代码第99行）。

(2) 将kernel×kernel个卷积核进行值赋，使每个卷积核中只有一个元素的值是1，其他都为0（见代码第102～第104行）。

(3) 调用tf.nn.depthwise_conv2d函数，进行深度卷积操作（见代码第108行）。

进行深度卷积之后，会得到input_shape[3]×kernel×kernel个特征数据(feature map)。该特征数据(feature map)与输入数据相比，尺寸会缩小，通道数会增多。即：

- 在新的尺寸下，有3×3个特征数据。
- 每个特征中又包含input_shape[3]个最终特征。
- 每个最终特征中包含了指定的当前层输出维度个胶囊单元(姿态矩阵+激活，共17维)。

具体代码如下：

代码 8-4 capsnet_em (续)

```

95 def kernel_tile(input, kernel, stride):
96
97     input_shape = input.get_shape()
98     # 定义卷积核，输入 ch 是 input_shape[3]，卷积核个数 (ch) 是 kernel * kernel
99     tile_filter = np.zeros(shape=[kernel, kernel, input_shape[3],
100                           kernel * kernel], dtype=np.float32)
101    # 为这 9 个卷积核赋值，每个卷积核的 3×3 矩阵中有一个为 1
102    for i in range(kernel):
103        for j in range(kernel):
104            tile_filter[i, j, :, i * kernel + j] = 1.0  # kernel=3，步长为 2,
可以理解成分成 9 个一段。从中取样一个
105
106    tile_filter_op = tf.constant(tile_filter, dtype=tf.float32)
107    # 深度卷积，在 12×12 上按照 3×3 进行卷积。由于每个卷积核只有一个 1，相当于采样
108    output = tf.nn.depthwise_conv2d(input, tile_filter_op, strides=[1,
109                                  stride, stride, 1], padding='VALID')
110    output_shape = output.get_shape()
111    output = tf.reshape(output, shape=[int(output_shape[0]),
112                                int(output_shape[1]),
113                                int(output_shape[2]),
114                                int(output_shape[3]), kernel * kernel])
115    output = tf.transpose(output, perm=[0, 1, 2, 4, 3]) # (batch, 5, 5, 9,
    ch)
116
117    return output

```

代码第 113 行对输出结果做了变形处理，使输出结果的最后一维与输入数据的最后一维相同。

3. 实现 calEM 函数

函数 calEM 的实现分为两部分操作：

- 用 mat_transform 函数计算投票。
- 用 em_routing 函数计算 EM 路由。

具体代码如下：

代码 8-4 capsnet_em (续)

```

116 def calEM(pose, activation, votes_output, weights_regularizer, name):
117     with tf.variable_scope('v') as scope:          # 计算投票
118         votes = mat_transform(pose, votes_output, weights_regularizer)
119         tf.logging.info('{} votes shape:'.format(name))
120         tf.logging.info('{} votes shape()'.format(votes.get_shape())) # 形状为 (576, 16, 10, 16)
121     # 计算 EM 路由，得到最终的姿态矩阵和激活值
122     with tf.variable_scope('routing') as scope2:
123         miu, activation = em_routing(votes, activation, votes_output,
weights_regularizer)
124         tf.logging.info(

```

```

124     '() activation shape: {}'.format(name, activation.get_shape()))
125     return miu, activation
126
127 #输入为[batch, caps_num_i, 16], 输出为[batch, caps_num_i,caps_num_c, 16]
128 def mat_transform(input, caps_num_c, regularizer, tag=False):
129     batch_size = int(input.get_shape()[0])
130     caps_num_i = int(input.get_shape()[1])#caps_num_i 的值为 3 × 3 × B
131     output = tf.reshape(input, shape=[batch_size, caps_num_i, 1, 4, 4])
132
133     w = slim.variable('w', shape=[1, caps_num_i, caps_num_c, 4, 4],
134                         dtype=tf.float32,
135                         initializer=tf.truncated_normal_initializer(mean=0.0,
136                         stddev=1.0),
137                         regularizer=regularizer)
138
139     w = tf.tile(w, [batch_size, 1, 1, 1, 1])用 tile 代替循环相乘, 提升效率
140     output = tf.tile(output, [1, 1, caps_num_c, 1, 1])
141     votes = tf.reshape(output@w, [batch_size, caps_num_i, caps_num_c, 16])
142     return votes
143
144 ac_lambda0=0.01 #定义 softMax 的温度参数
145
146 def em_routing(votes, activation, caps_num_c, regularizer, tag=False):
147     batch_size = int(votes.get_shape()[0])
148     caps_num_i = int(activation.get_shape()[1])
149     n_channels = int(votes.get_shape()[-1])#姿态矩阵 16
150     print("n_channels",n_channels)
151
152     sigma_square = []
153     miu = []
154     activation_out = []
155     #定义 caps_num_c 个投票, 每个投票包括 n_channels 和激活值
156     beta_v = slim.variable('beta_v', shape=[caps_num_c, n_channels],
157     dtype=tf.float32,
158     initializer=tf.constant_initializer(0.0),
159     regularizer=regularizer)
160
161     beta_a = slim.variable('beta_a', shape=[caps_num_c], dtype=tf.float32,
162     initializer=tf.constant_initializer(0.0),
163     regularizer=regularizer)
164
165     votes_in = votes
166     activation_in = activation
167     for iters in range(iter_routing):
168         #E 步骤: 第 1 次, caps_num_c 中的每个概率都一样
169         if iters == 0:

```

```

168     r = tf.constant(np.ones([batch_size, caps_num_i, caps_num_c]),
169                      dtype=np.float32) / caps_num_c)
170     else:
171         log_p_c_h = -tf.log(tf.sqrt(sigma_square)) - (tf.square(votes_in -
172             miu) / (2 * sigma_square))
173         log_p_c_h = log_p_c_h - \
174             (tf.reduce_max(log_p_c_h, axis=[2, 3], keep_dims=True)
175             - tf.log(10.0))
176         p_c = tf.exp(tf.reduce_sum(log_p_c_h, axis=3))
177         ap = p_c * tf.reshape(activation_out, shape=[batch_size, 1,
178                                         caps_num_c])
179         r = ap / (tf.reduce_sum(ap, axis=2, keepdims=True) + epsilon)
180     #M 步骤
181     r = r * activation_in #更新概率值
182     r = r / (tf.reduce_sum(r, axis=2, keepdims=True)+epsilon) #将数值转化为总数的占比（总数为 1）
183     #所有胶囊的父胶囊连接概率收集起来
184     r_sum = tf.reduce_sum(r, axis=1, keepdims=True)
185     r1 = tf.reshape(r / (r_sum + epsilon),
186                     shape=[batch_size, caps_num_i, caps_num_c, 1])
187     miu = tf.reduce_sum(votes_in * r1, axis=1, keepdims=True)
188     sigma_square = tf.reduce_sum(tf.square(votes_in - miu) * r1,
189                                 axis=1, keepdims=True) + epsilon
190
191     if iters == iter_routing-1:
192         r_sum = tf.reshape(r_sum, [batch_size, caps_num_c, 1])
193         #计算信息熵
194         cost_h = (beta_v + tf.log(tf.sqrt(tf.reshape(sigma_square,
195                                         shape=[batch_size, caps_num_c, n_channels])))) * r_sum
196         activation_out = tf.nn.softmax(ac_lambda0 * (beta_a -
197             tf.reduce_sum(cost_h, axis=2)))
198     else:
199         activation_out = tf.nn.softmax(r_sum)
200
201     return miu, activation_out

```

代码第 128 行定义了 mat_transform 函数。

在函数 mat_transform 中定义了一组权重。用该权重依次与输入批次中的每个矩阵元素相乘，得到投票 votes。

代码第 137 行，用 tf.tile 函数将循环相乘的计算方式替换了矩阵相乘。这提高了预算效

率。`tf.tile` 函数的详细介绍见 8.2.5 小节。

代码第 144 行定义了函数 `em_routing`。该函数将输入的投票根据每个姿态矩阵所分配的系数进行加权计算。其中，每个姿态矩阵的分配系数是通过 EM 路由算法进行迭代更新的。

EM 路由算法属于非监督类学习算法。该算法使用聚类的方式由 E 步和 M 步两个环节组成：

- E 步负责在已有的激活值和投票分布上计算加权值（路由分配），见代码第 167~178 行。
- M 步负责更新加权值（路由值），并根据加权后的投票值算出其所在分布的均值与方差（见代码第 189~190 行）。

在具体执行时，通过 E 步和 M 步的循环交替迭代运算完成整个路由的更新。



提示：

在 EM 路由中，用于对投票进行聚类的算法类似于高斯混合模型（GMM）算法（GMM 本质上属于加权的 K 均值算法，其加权属性很符合当前场景），但在高斯混合模型（GMM）基础上又做了些改动。这里不做展开，有兴趣的读者请见 8.1.8 小节的论文链接。

代码第 192 行是 EM 路由计算之后的处理步骤，要对 EM 路由过程中的两个结果（投票均值 `miu` 和信息熵 `cost_h`）进行提取。

- 投票均值 `miu`：输出的姿态矩阵结果，表示样本中的不变特征。
- 信息熵 `cost_h`：代表分类结果（见代码第 195 行）。



提示：

代码第 198 行比较难懂，这里解释一下：

参数 `ac_lambda0` 是退火策略训练中的温度参数。

函数 `softmax` 负责找出特征值最大的索引，用于计算最终的分类结果。

`cost_h` 是信息熵，该值越小，则该类的稳定性越好，该类的结果特征越明显。但最终计算结果的 `softmax` 是按照最大值进行分类的，于是又对信息熵取负，取其反向的特征。

4. 运行程序

在 EM 路由中，用函数 `spread loss` 计算损失。训练部分的代码见本书配套资源中的代码文件“8-5_train_EM.py”。这里不再详述。

将代码运行后输入以下结果：

```
.....
38 epoch , 220 iteration finishes in 2.526243 second loss=0.024377 acc 1.0
38 epoch , 240 iteration finishes in 26437.966891 second loss=0.020554 acc 0.921875
38 epoch , 260 iteration finishes in 2.648914 second loss=0.030478 acc 0.9375
38 epoch , 280 iteration finishes in 2.760616 second loss=0.031670 acc 0.984375
38 epoch , 300 iteration finishes in 2.664881 second loss=0.014428 acc 0.984375
38 epoch , 320 iteration finishes in 2.717733 second loss=0.016623 acc 1.0
38 epoch , 340 iteration finishes in 2.732693 second loss=0.013332 acc 1.0
38 epoch , 360 iteration finishes in 2.604040 second loss=0.026992 acc 0.96875
```

```
38 epoch , 380 iteration finishes in 2.669863 second loss=0.028396 acc 0.953125
38 epoch , 400 iteration finishes in 2.536231 second loss=0.033339 acc 0.953125
```

由结果可见，EM 胶囊网络模型的识别率还是非常可观的。但由于该模型算法比较复杂，占用资源比较大，训练起来会相对慢一些。

8.3 实例 41：用 TextCNN 模型分析评论者是否满意

卷积神经网络不仅只用在处理图像视觉方面，在基于文本的 NLP 领域也会有很好的效果。TextCNN 模型是卷积神经网络用在文本处理方面的一个知名模型。在 TextCNN 模型中，通过多通道卷积技术实现了对文本的分类功能。下面就来了解一下。

实例描述

有一个记录评论语句的数据集，分为正面和负面两种情绪。通过训练，让模型能够理解正面与负面两种情绪的语义，并对评论文本进行分类。

对于 NLP 任务的处理，在模型中常用的技术是使用 RNN 模型。但如果把语言向量当作一副图像，CNN 模型也是可以对其分类的。

8.3.1 熟悉样本：了解电影评论数据集

本实例使用的数据集是康奈尔大学发布的电影评论数据集，具体的介绍见以下链接：

<http://www.cs.cornell.edu/people/pabo/movie-review-data/>

在其中找到数据集的下载地址，如下：

<http://www.cs.cornell.edu/people/pabo/movie-review-data/rt-polaritydata.tar.gz>

将压缩包“rt-polaritydata.tar.gz”下载后可以看到，里面包括 5331 个正面的评论和 5331 个负面的评论。

8.3.2 熟悉模型：了解 TextCNN 模型

TextCNN 模型是利用卷积神经网络对文本进行分类的算法，由 Yoon Kim 在 *Convolutional Neural Networks for Sentence Classification* 一文中提出。论文地址：

<https://arxiv.org/pdf/1408.5882.pdf>

该模型的结构可以分为以下 4 层。

- 词嵌入层：将每个词对应的向量转化成多维度的词嵌入向量。将每个句子当作一副图像来进行处理（词的个数×词嵌入向量维度）。
- 多通道卷积层：使用 2、3、4 等不同大小的卷积核对词嵌入转化后的句子做卷积操作。生成大小不同的特征数据。
- 多通道全局最大池化层：对多通道卷积层中输出的每个通道的特征数据做全局最大池化。

操作。

- 全连接分类输出层: 将池化后的结果输入全连接网络中, 输出分类个数, 得到最终结果。整个 TextCNN 模型的结构如图 8-16 所示。

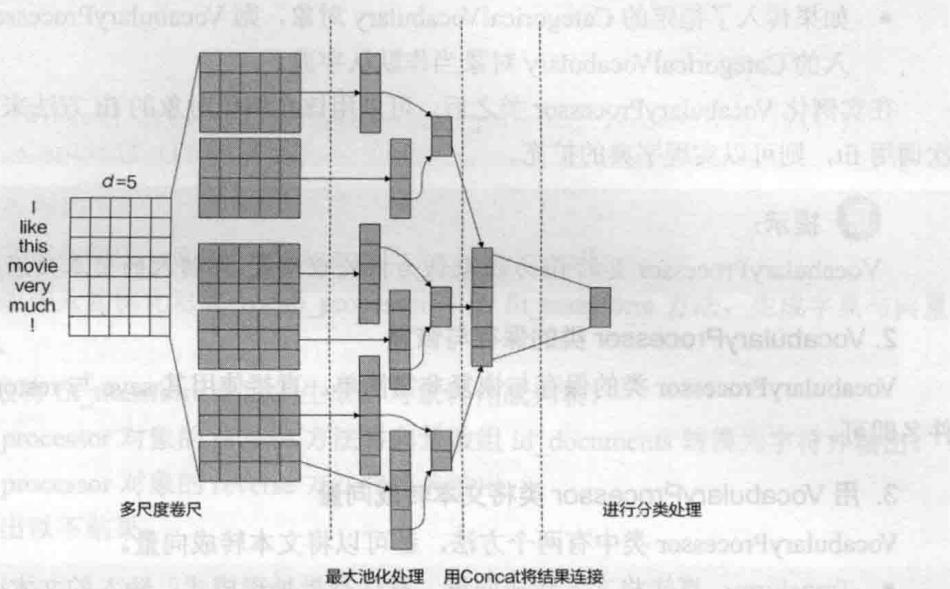


图 8-16 TextCNN 模型的结构

因为卷积神经网络具有提取局部特征的功能, 所以可用卷积神经网络提取句子中类似 n-gram 算法的关键信息。本实例的任务是可以理解为通过句子中的关键信息进行语义分类, 这与 TextCNN 模型的功能是相匹配的。



提示:

由于 TextCNN 模型中使用了池化操作, 在这个过程中丢失了一些信息, 导致该模型所表示的句子特征有限。如果要用处理相近语义的分类任务, 则还需要对其进行调整。

8.3.3 数据预处理: 用 preprocessing 接口制作字典

在 TensorFlow 的 contrib 模块中, 有个 learn 模块。该模块下的 preprocessing 接口可以用于 NLP 任务的数据预处理。其中包括一个 VocabularyProcessor 类, 该类可以实现文本与向量间的相互转化、字典的创建与保存、对词向量的对齐处理等操作。

1. VocabularyProcessor 类的定义

VocabularyProcessor 类的初始化函数如下:

```
VocabularyProcessor (
    max_document_length,      #语句预处理的长度。按照该长度对语句进行切断、补0处理
    min_frequency=0,          #词频的最小值。如果出现的次数小于最小词频，则不会被收录到词表中
    vocabulary=None,          #CategoricalVocabulary 对象。如果为 None，则重新创建一个
    tokenizer_fn=None)        #分词函数
```

在实例化 VocabularyProcessor 类时，其内部的字典与传入的参数 vocabulary 相关。

- 如果传入的参数 vocabulary 为 None，则 VocabularyProcessor 类会在内部重新生成一个 CategoricalVocabulary 对象用于存放字典。
- 如果传入了指定的 CategoricalVocabulary 对象，则 VocabularyProcessor 类会在内部将传入的 CategoricalVocabulary 对象当作默认字典。

在实例化 VocabularyProcessor 类之后，可以用该实例化对象的 fit 方法来生成字典。如果再次调用 fit，则可以实现字典的扩充。



提示：

VocabularyProcessor 类的 fit 方法默认为批处理模式，即传入的文本必须是可迭代的对象。

2. VocabularyProcessor 类的保存与恢复

VocabularyProcessor 类的保存与恢复非常简单。直接使用其 save 与 restore 方法，并传入文件名即可。

3. 用 VocabularyProcessor 类将文本转成向量

VocabularyProcessor 类中有两个方法，都可以将文本转成向量。

- Transform：直接将文本转成向量。默认是批处理模式，输入的文本必须是可迭代的对象（在使用时，需要确认 VocabularyProcessor 类的实例化对象中已经生成过字典）。
- fit_transform：将文本转成向量，同时也生成了字典。相当于先调用 fit 再调用 Transform。

4. 用 VocabularyProcessor 类将向量转成文本

直接用 VocabularyProcessor 类的 reverse 方法，可以将向量转成文本。默认是批处理模式，输入的文本必须是可迭代的对象。

5. 用简单代码演示

VocabularyProcessor 类的具体使用，见以下代码：

```
from tensorflow.contrib import learn          # 导入模块
import tensorflow as tf
import numpy as np

x_text = ['www.aiianaconda.com', 'xiangyuejiqiren']      # 定义待处理文本
max_document_length = max([len(x) for x in x_text])    # 计算最大长度

def e_tokenizer(documents):                      # 定义分词函数
    for document in documents:
        yield [i for i in document]              # 每个字母分一次

# 实例化 VocabularyProcessor
vocab_processor = learn.preprocessing.VocabularyProcessor(max_document_length, 1,
tokenizer_fn=e_tokenizer)

id_documents = list(vocab_processor.fit_transform(x_text)) # 生成字典并将文本转换成向量
for id_document in id_documents:
```

```

print(id_document)

for document in vocab_processor.reverse(id_documents):      #将向量转换为文本
    print(document.replace(' ', ''))

#输出字典
a=next
(vocab_processor.reverse([list(range(0,len(vocab_processor.vocabulary_)))]))
print("字典: ",a.split(' '))

```

该代码片段的流程如下：

- (1) 定义一个文本数组['www.ianaconda.com','xiangyuejiqiren']。
- (2) 将文本数组传入实例化对象 vocab_processor 中的 fit_transform 方法，生成字典与向量数组 id_documents。
- (3) 用 list 函数将 fit_transform 返回的生成器对象转化成列表。
- (4) 用 vocab_processor 对象的 reverse 方法将向量数组 id_documents 转换为字符并输出。
- (5) 用 vocab_processor 对象的 reverse 方法将字典输出。

程序运行后输出以下结果：

```

[4 4 4 5 1 2 1 3 1 6 8 3 0 1 5 6 8 0]
[0 2 1 3 0 0 0 7 0 2 0 2 0 7 3 0 0 0]
www.ianacon<UNK>a.co<UNK>
<UNK>ian<UNK><UNK><UNK>i<UNK>i<UNK>en<UNK><UNK><UNK>
字典: ['<UNK>', 'a', 'i', 'n', 'w', '.', 'c', 'e', 'o']

```

输出结果的第 2 行是一个列表。可以看到，该列表中最后 3 个元素的值是 0，表示在长度不足时系统会自动补 0。

从输出结果的最后一行可以看到，字典的第 0 个位置用 '<UNK>' 表示其他的低频字符。在实例化对象 vocab_processor 时，传入的参数 min_frequency 是 1，代表出现次数小于 1 的字符将被当作低频字符进行处理。在字符转化向量过程中，所有的低频字符将被统一用 '<UNK>' 的索引来替换。



提示：

由于 preprocessing 接口是完全用 Python 基本语法来实现的，与 TensorFlow 框架的关系不大。在 TensorFlow 2.x 中，preprocessing 接口被删掉了。

因为它可以独立于 TensorFlow，所以可以很容易通过手动的方式将 preprocessing 接口从 TensorFlow 框架中脱离出来。方法是：将整个 preprocessing 文件夹复制出来，放到本地代码同级路径下，使其从本地环境开始加载。

这样，在 TensorFlow 新的版本中，即使该代码被删掉也不会影响使用。

可以用 tf.keras 接口中的 preprocessing 模块来实现文本的预处理，这会使代码的开发更快捷（9.3 节有具体实例演示），具体用法可以参考以下链接：

<https://keras.io/zh/preprocessing/text/>

8.3.4 代码实现：生成 NLP 文本数据集

在编写代码之前，需要按照 8.3.3 小节中的最后一个提示部分，将 preprocessing 复制到本地代码的同级目录下。同时，也将样本数据复制到本地代码同级目录的 data 文件夹下。

将字符数据集的样本转换为字典和向量数据集。

具体代码如下：

代码 8-6 NLP 文本预处理

```

01 import tensorflow as tf
02 import preprocessing
03
04 positive_data_file = "./data/rt-polaritydata/rt-polarity.pos"
05 negative_data_file = "./data/rt-polaritydata/rt-polarity.neg"
06
07 def mydataset(positive_data_file,negative_data_file): # 定义函数，创建数据集
08     filelist = [positive_data_file,negative_data_file]
09
10    def gline(filelist): # 定义生成器函数，返回每一行的数据
11        for file in filelist:
12            with open(file, "r",encoding='utf-8') as f:
13                for line in f:
14                    yield line
15
16    x_text = gline(filelist)
17    lenlist = [len(x.split(" ")) for x in x_text]
18    max_document_length = max(lenlist)
19    vocab_processor =
preprocessing.VocabularyProcessor(max_document_length,5)
20
21    x_text = gline(filelist)
22    vocab_processor.fit(x_text)
23    a=list
    (vocab_processor.reverse( [list(range(0,len(vocab_processor.vocabulary_))
))]) )
24    print("字典: ",a)
25
26    def gen(): # 循环生成器（否则一次生成器结束就会没有了）
27        while True:
28            x_text2 = gline(filelist)
29            for i ,x in enumerate(vocab_processor.transform(x_text2)):
30                if i < int(len(lenlist)/2):
31                    onehot = [1,0]
32                else:
33                    onehot = [0,1]
34                yield (x,onehot)
35

```

```

36     data = tf.data.Dataset.from_generator( gen, (tf.int64,tf.int64) ) #ini 50
37     data = data.shuffle(len(lenlist)) as miledlinoo.wilcoonef rroqmi 60
38     data = data.batch(256) #nud,0
39     data = data.prefetch(1) # (doe)HNDixet seash 20
40     return data,vocab_processor,max_document_length #返回数据集、字典、最大长度
41 activation_fn=tanh,strengthen_fn=leaky_relu
42 if __name__ == '__main__':
43     data,_,_ =mydataset(positive_data_file,negative_data_file)
44     iterator = data.make_initializable_iterator()
45     next_element = iterator.get_next()
46
47     with tf.Session() as sess2:
48         sess2.run(iterator.initializer)
49         for i in range(80):
50             print("batched data 1:",i)
51             sess2.run(next_element)

```

代码第 26 行，定义了内置函数 gen。在内置函数 gen 中返回一个无穷循环的生成器对象。该生成器对象可以支持在迭代训练过程中对数据集的重复遍历。



提示：

代码第 26 行，在 gen 中设置的无穷循环的生成器对象非常重要。如果不循环，即使在外层数据集上做 repeat，也无法再次获取数据（因为如果没有循环，生成器迭代一次就结束了）。

代码第 36 行，将内置函数 gen 传入 tf.data.Dataset.from_generator 接口来制作数据集。

代码第 42 行是该数据集的测试实例。

生成字典的知识在 8.3.3 小节有介绍，这里不再详述。

整个代码运行后，输出以下内容：

```

字典： ["<UNK> the a and of to is in that it as but with film this for its an movie
it's be on you not by about more one like has are at from than all his -- have so if
or story i too just who into what
.....
wholesome wilco wisdom woo's ya youthful zhang"]
batched data 1: 0
....
batched data 1: 79

```

生成结果其中包括两部分内容：字典的内容（前 4 行）、数据集的循环输出（后 3 行）。

8.3.5 代码实现：定义 TextCNN 模型

下面按照 8.3.2 小节中介绍的 TextCNN 模型结构实现 TextCNN 模型。具体代码如下：

代码 8-7 TextCNN 模型

```

01 import tensorflow as tf

```

```

02 import numpy as np
03 import tensorflow.contrib.slim as slim
04
05 class TextCNN(object):
06     """
07     TextCNN 文本分类器
08     """
09     def __init__(self, sequence_length, num_classes, vocab_size, embedding_size, filter_sizes, num_filters, l2_reg_lambda=0.0):
10
11         # 定义占位符
12         self.input_x = tf.placeholder(tf.int32, [None, sequence_length], name="input_x")
13         self.input_y = tf.placeholder(tf.float32, [None, num_classes], name="input_y")
14         self.dropout_keep_prob = tf.placeholder(tf.float32, name="dropout_keep_prob")
15
16         # 定义词嵌入层
17         with tf.variable_scope('Embedding'):
18             embed = tf.contrib.layers.embed_sequence(self.input_x,
19                 vocab_size=vocab_size, embed_dim=embedding_size)
20             self.embedded_chars_expanded = tf.expand_dims(embed, -1)
21
22         # 定义多通道卷积与最大池化网络
23         pooled_outputs = []
24         for i, filter_size in enumerate(filter_sizes):
25             conv = slim.conv2d(self.embedded_chars_expanded, num_outputs =
26                 num_filters,
27                     kernel_size=[filter_size, embedding_size],
28                     stride=1, padding="VALID",
29                     activation_fn=tf.nn.leaky_relu, scope="conv%d" %
30                         filter_size)
31             pooled = slim.max_pool2d(conv, [sequence_length - filter_size +
32                 1, 1], padding='VALID',
33                         scope="pool%d" % filter_size)
34
35             pooled_outputs.append(pooled)      # 将各个通道结果合并起来
36
37         # 展开特征，并添加 dropout 方法
38         num_filters_total = num_filters * len(filter_sizes)
39         self.h_pool = tf.concat(pooled_outputs, 3)
40         self.h_pool_flat = tf.reshape(self.h_pool, [-1, num_filters_total])
41         with tf.name_scope("dropout"):
42             self.h_drop = tf.nn.dropout(self.h_pool_flat, self.dropout_keep_prob)

```

```

41      #计算 L2_loss 值
42      l2_loss = tf.constant(0.0)
43      with tf.name_scope("output"):
44          self.scores = slim.fully_connected(self.h_drop, num_classes,
45          activation_fn=None, scope="fully_connected")
46          for tf_var in tf.trainable_variables():
47              if ("fully_connected" in tf_var.name):
48                  l2_loss += tf.reduce_mean(tf.nn.l2_loss(tf_var))
49          print("tf_var", tf_var)
50
51      self.predictions = tf.argmax(self.scores, 1, name="predictions")
52
53      #计算交叉熵
54      with tf.name_scope("loss"):
55          losses =
56          tf.nn.softmax_cross_entropy_with_logits_v2(logits=self.scores,
57          labels=self.input_y)
58          self.loss = tf.reduce_mean(losses) + l2_reg_lambda * l2_loss
59
60      #计算准确率
61      correct_predictions = tf.equal(self.predictions,
62          tf.argmax(self.input_y, 1))
63      self.accuracy = tf.reduce_mean(tf.cast(correct_predictions,
64          "float"), name="accuracy")
65
66      def build_mode(self):#定义函数构建模型
67          self.global_step = tf.Variable(0, name="global_step",
68          trainable=False)
69          optimizer = tf.train.AdamOptimizer(1e-3)
70          grads_and_vars = optimizer.compute_gradients(self.loss)
71          self.train_op = optimizer.apply_gradients(grads_and_vars,
72          global_step=self.global_step)
73
74      #生成摘要
75      grad_summaries = []
76      for g, v in grads_and_vars:
77          if g is not None:
78              grad_hist_summary =
79              tf.summary.histogram("{} / grad / hist".format(v.name), g)
80              sparsity_summary =
81              tf.summary.scalar("{} / grad / sparsity".format(v.name),
82              tf.nn.zero_fraction(g))
83              grad_summaries.append(grad_hist_summary)
84              grad_summaries.append(sparsity_summary)
85      grad_summaries_merged = tf.summary.merge(grad_summaries)

```

```

78     #生成损失及准确率的摘要
79     loss_summary = tf.summary.scalar("loss", self.loss)
80     acc_summary = tf.summary.scalar("accuracy", self.accuracy)
81     class TextCNN:
82         def __init__(self, sequence_length, vocab_size, embedding_size,
83                      filter_sizes=[3, 4, 5], num_filters=64, l2_reg_lambda=0.1):
84             #合并摘要
85             self.train_summary_op = tf.summary.merge([loss_summary, acc_summary,
86             grad_summaries_merged])

```

在词嵌入部分使用了 `tf.layers` 接口，在多通道卷积部分使用了 `TF-slim` 接口。在模型中用到了 `dropout` 方法与正则化方法，这两个方法可以改善模型的过拟合问题。

8.3.6 代码实现：训练 TextCNN 模型

下面将 `TestCNN` 模型与数据集代码文件载入，在会话中训练模型。具体代码如下：

代码 8-8 用 TextCNN 模型进行文本分类

```

01 import tensorflow as tf
02 import os
03 import time
04 import datetime
05
06 predata = __import__("8-6 NLP 文本预处理")
07 mydataset = predata.mydataset
08 text_cnn = __import__("8-7 TextCNN 模型")
09 TextCNN = text_cnn.TextCNN
10
11 def train():
12     #指定样本文件
13     positive_data_file = "./data/rt-polaritydata/rt-polarity.pos"
14     negative_data_file = "./data/rt-polaritydata/rt-polarity.neg"
15     #设置训练参数
16     num_steps = 2000          #定义训练的次数
17     display_every=20          #定义训练中的显示间隔
18     checkpoint_every=100      #定义训练中保存模型的间隔
19     SaveFileName= "text_cnn_model" #定义保存模型文件夹名称
20     #设置模型参数
21     num_classes = 2           #设置模型分类
22     dropout_keep_prob = 0.8   #定义 dropout 系数
23     l2_reg_lambda=0.1          #定义正则化系数
24     filter_sizes = "3,4,5"    #定义多通道卷积核
25     num_filters = 64          #定义每通道的输出个数
26
27     tf.reset_default_graph()  #重置运算图
28
29     #预处理生成字典及数据集
30     data,vocab_processor,max_document_length

```

=mydataset(positive_data_file,negative_data_file)

```
31     iterator = data.make_one_shot_iterator()
32     next_element = iterator.get_next()
33
34     # 定义 TextCNN 模型
35     cnn = TextCNN(
36         sequence_length=max_document_length,
37         num_classes=num_classes,
38         vocab_size=len(vocab_processor.vocabulary_),
39         embedding_size=128,
40         filter_sizes=list(map(int, filter_sizes.split(","))),
41         num_filters=num_filters,
42         l2_reg_lambda=l2_reg_lambda)
43
44     # 构建网络
45     cnn.build_mode()
46
47     # 打开会话(session)，准备训练
48     session_conf =
49         tf.ConfigProto(allow_soft_placement=True, log_device_placement=False)
50
51     with tf.Session(config=session_conf) as sess:
52         sess.run(tf.global_variables_initializer())
53
54         # 准备输出模型路径
55         timestamp = str(int(time.time()))
56         out_dir = os.path.abspath(os.path.join(os.path.curdir, SaveFileName,
57                                             timestamp))
58         print("Writing to {}\n".format(out_dir))
59
60         # 设置输出摘要的路径
61         train_summary_dir = os.path.join(out_dir, "summaries", "train")
62         train_summary_writer = tf.summary.FileWriter(train_summary_dir,
63                                                       sess.graph)
64
65         # 设置检查点文件的名称
66         checkpoint_dir = os.path.abspath(os.path.join(out_dir,
67                                                 "checkpoints"))
68         checkpoint_prefix = os.path.join(checkpoint_dir, "model")
69         if not os.path.exists(checkpoint_dir):
70             os.makedirs(checkpoint_dir)
71
72         # 定义操作检查点的 saver
73         saver = tf.train.Saver(tf.global_variables(), max_to_keep=1)
74
75         # 保存字典
76         vocab_processor.save(os.path.join(out_dir, "vocab"))
77
78         def train_step(x_batch, y_batch):# 定义函数，完成训练步骤
79             feed_dict = {
80                 cnn.input_x: x_batch,
```

```

74     cnn.input_y: y_batch, sess.run(cnn.global_step) = 0
75     loss, cnn.dropout_keep_prob: dropout_keep_prob = 1
76     } summary = tf.summary.scalar("accuracy", self.accuracy)
77     _, step, summaries, loss, accuracy = sess.run(
78         [cnn.train_op, cnn.global_step, cnn.train_summary_op,
79         cnn.loss, cnn.accuracy],
80         feed_dict)
81     time_str = datetime.datetime.now().isoformat()
82     train_summary_writer.add_summary(summaries, step)
83     return (time_str, step, loss, accuracy)
84
85     i = 0
86     while tf.train.global_step(sess, cnn.global_step) < num_steps:
87         x_batch, y_batch = sess.run(next_element)
88         i = i+1
89         time_str, step, loss, accuracy = train_step(x_batch, y_batch)
90         current_step = tf.train.global_step(sess, cnn.global_step)
91         if current_step % display_every == 0:
92             print("{}: step {}, loss {:.3f}, acc {:.3f}".format(time_str, step,
93             loss, accuracy))
94             if current_step % checkpoint_every == 0:
95                 path = saver.save(sess, checkpoint_prefix,
96                 global_step=global_step)
97                 print("Saved model checkpoint to {} \n".format(path))
98
99     def main(argv=None):
100         train() # 启动训练
101     if __name__ == '__main__':
102         tf.app.run()

```

由于篇幅关系，本实例只演示了训练部分的代码文件。有关测试与应用的代码，读者可以参考本书其他实例自行实现。

8.3.7 运行程序

代码写好后，直接运行。输出以下结果：

```

2018-07-11T12:27:51.187195: step 20, loss 0.77673, acc 0.664062
2018-07-11T12:27:52.043903: step 40, loss 0.747624, acc 0.675781
...
2018-07-11T12:28:46.933766: step 1220, loss 0.0422899, acc 0.996094
2018-07-11T12:28:47.762518: step 1240, loss 0.0472618, acc 0.988281
2018-07-11T12:28:48.591300: step 1260, loss 0.0389083, acc 0.996094
2018-07-11T12:28:49.424072: step 1280, loss 0.039029, acc 0.992188
2018-07-11T12:28:50.249862: step 1300, loss 0.0413458, acc 0.988281

```

可以看到训练效果还是很显著的，在rt-polaritydata数据集上达到了0.9以上的准确率。

8.3.8 扩展：提升模型精度的其他方法

将视觉处理技术用在文本分类任务上，会产生很好的效果。读者可以尝试使用以下方法进一步提升TextCNN模型的精度。

- 用类似Inception系列模型的cell单元代替多通道卷积：TextCNN模型的结构与Inception系列模型的单元结构非常相似。所以，可以尝试用标准的Inception系列模型的cell（或是NASNet模型的cell）来处理多通道卷积部分。如果句子非常长，则可以在通道中尝试使用更大的卷积核。
- 将最大池化替换为空洞卷积：在8.1.6小节讲过，最大池化过程会丢失很多重要信息，所以，可以尝试用空洞卷积的方式让模型减小信息丢失。
- 更好地使用词嵌入：在模型中使用的词嵌入是从头开始训练的，在样本不足的情况下，模型的泛化能力会较差。可以在词嵌入的训练过程中，引入已经训练好的公开词向量对词嵌入层进行初始化；还可以直接用已经训练好的公开词向量将输入词转化为向量特征，并用转化后的向量特征来训练后面的模型。
- 通过一些小技巧来提升模型精度：例如，更换激活函数、更换优化器、调节学习率、调节dropout率、增加每通道的输出个数等。

8.4 实例42：用带注意力机制的模型分析评论者是否满意

用tf.keras接口搭建一个只带有注意力机制的模型，实现文本分类。

实例描述

有一个记录评论语句的数据集，分为正面和负面两种情绪。通过训练模型，让其学会正面与负面两种情绪对应的语义。

注意力机制是解决NLP任务的一种方法（见8.1.10小节）。其内部的实现方式与卷积操作非常类似。在脱离RNN结构的情况下，单独的注意力机制模型也可以很好地完成NLP任务。具体做法如下。

8.4.1 熟悉样本：了解tf.keras接口中的电影评论数据集

IMDB数据集中含有25000条电影评论，从情绪的角度分为正面、负面两类标签。该数据集相当于图片处理领域的MNIST数据集，在NLP任务中经常被使用。

在tf.keras接口中，集成了IMDB数据集的下载及使用接口。该接口中的每条样本内容都是以向量形式存在的。

调用tf.keras.datasets.imdb模块下的load_data函数即可获得数据，该函数的定义如下：

```
def load_data(path='imdb.npz', #默认的数据集文件
```

```

    num_words=None,           #单词数量，即文本转向量后的最大索引
    skip_top=0,               #跳过前面频度最高的几个词
    maxlen=None,              #只取小于该长度的样本
    seed=113,                 #乱序样本的随机种子
    start_char=1,             #每一组序列数据最开始的向量值。
    oov_char=2,                #在字典中，遇到不存在的字符用该索引来替换
    index_from=3,              #大于该数的向量将被认为是正常的单词
    **kwargs):                  #为了兼容性而设计的预留参数

```

该函数会返回两个元组类型的对象。

- (x_train, y_train): 训练数据集。如果指定了 num_words 参数，则最大索引值是 num_words-1。如果指定了 maxlen 参数，则序列长度大于 maxlen 的样本将被过滤掉。
- (x_test, y_test): 测试数据集。



提示：

由于 load_data 函数返回的样本数据没有进行对齐操作，所以还需要将其进行对齐处理（按照指定长度去整理数据集，多了的去掉，少了的补 0）后才可以使用。

8.4.2 代码实现：将 tf.keras 接口中的 IMDB 数据集还原成句子

本节代码共分为两部分，具体如下。

- 加载 IMDB 数据集及字典：用 load_data 函数下载数据集，并用 get_word_index 函数下载字典。
- 读取数据并还原句子：将数据集加载到内存，并将向量转换成字符。

1. 加载 IMDB 数据集及字典

在调用 tf.keras.datasets.imdb 模块下的 load_data 函数和 get_word_index 函数时，系统会默认去网上下载预处理后的 IMDB 数据集及字典。如果由于网络原因无法成功下载 IMDB 数据集与字典，则可以加载本书的配套资源：IMDB 数据集文件“imdb.npz”与字典“imdb_word_index.json”。

将 IMDB 数据集文件“imdb.npz”与字典文件“imdb_word_index.json”放到本地代码的同级目录下，并对 tf.keras.datasets.imdb 模块的源代码文件中的函数 load_data 进行修改，关闭该函数的下载功能。具体如下所示。

(1) 找到 tf.keras.datasets.imdb 模块的源代码文件。以作者本地路径为例，具体如下：

```
C:\local\Anaconda3\lib\site-packages\tensorflow\python\keras\datasets\imdb.py
```

(2) 打开该文件，在 load_data 函数中，将代码的第 80~84 行注释掉。具体代码如下：

```

# origin_folder = 'https://storage.googleapis.com/tensorflow/tf-keras-datasets/'
# path = get_file(
#     path,
#     origin=origin_folder + 'imdb.npz',
#     file_hash='599dadbb1135973df5b59232a0e9a887c')

```

(3) 在 get_word_index 函数中, 将代码第 144~148 行注释掉。具体代码如下:

```
# origin_folder = 'https://storage.googleapis.com/tensorflow/tf-keras-datasets/'
# path = get_file(
#     path,
#     origin=origin_folder + 'imdb_word_index.json',
#     file_hash='bfaf718b763782e994055a2d397834f')
```

2. 读取数据并还原其中的句子

从数据集中取出一条样本, 并用字典将该样本中的向量转成句子, 然后输出结果。具体代码如下:

代码 8-9 用 keras 注意力机制模型分析评论者的情绪

```
01 from __future__ import print_function
02 import tensorflow as tf
03 import numpy as np
04 attention_keras = __import__("8-10 keras 注意力机制模型")
05
06 # 定义参数
07 num_words = 20000
08 maxlen = 80
09 batch_size = 32
10
11 # 加载数据
12 print('Loading data...')
13 (x_train, y_train), (x_test, y_test) = tf.keras.datasets.imdb.load_data(path='./imdb.npz', num_words=num_words)
14 print(len(x_train), 'train sequences')
15 print(len(x_test), 'test sequences')
16 print(x_train[:2])
17 print(y_train[:10])
18 word_index = tf.keras.datasets.imdb.get_word_index('./imdb_word_index.json') # 生成字典: 单词与下标对应
19 reverse_word_index = dict([(value, key) for (key, value) in word_index.items()]) # 生成反向字典: 下标与单词对应
20
21 decoded_newswire = ' '.join([reverse_word_index.get(i - 3, '?') for i in x_train[0]])
22 print(decoded_newswire)
```

代码第 21 行, 将样本中的向量转化成单词。在转化过程中, 将每个向量向前偏移了 3 个位置。这是由于在调用 load_data 函数时使用了参数 index_from 的默认值 3 (见代码第 13 行), 表示数据集中的向量值, 从 3 以后才是字典中的内容。

在调用 load_data 函数时, 如果所有的参数都使用默认值, 则所生成的数据集会比字典中多 3 个字符 “padding” (代表填充值)、 “start of sequence” (代表起始位置) 和 “unknown” (代

表未知单词) 分别对应于数据集中的向量 0、1、2。

代码运行后, 输出以下结果。

(1) 数据集大小为 25000 条样本。具体内容如下:

```
25000 train sequences
25000 test sequences
```

(2) 数据集中第 1 条样本的内容。具体内容如下:

```
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256,
5, 25, 100, .....15, 297, 98, 32, 2071, 56, 26, 141, 6, 194, 7486, 18, 4, 226, 22, 21,
134, 476, 26, 480, 5, 144, 30, 5535, 18, 51, 36, 28, 224, 92, 25, 104, 4, 226, 65, 16,
38, 1334, 88, 12, 16, 283, 5, 16, 4472, 113, 103, 32, 15, 16, 5345, 19, 178, 32]
```

结果中第一个向量为 1, 代表句子的起始标志。可以看出, tf.keras 接口中的 IMDB 数据集为每个句子都添加了起始标志。这是因为调用函数 load_data 时用参数 start_char 的默认值 1 (见代码第 13 行)。

(3) 前 10 条样本的分类信息。具体内容如下:

```
[1 0 0 1 0 0 1 0 1 0]
```

(4) 第 1 条样本数据的还原语句。具体内容如下:

```
? this film was just brilliant casting location scenery story direction everyone's
really suited the part they played and you could just imagine being there robert ? is
an amazing actor and now the ..... someone's life after all that was shared with us all
```

结果中的第一个字符为“?”，表示该向量在字典中不存在。这是因为该向量值为 1，代表句子的起始信息。而字典中的内容是从向量 3 开始的。在将向量转换成单词的过程中，将字典中不存在的字符替换成了“?”(见代码第 21 行)。

8.4.3 代码实现：用 tf.keras 接口开发带有位置向量的词嵌入层

在 tf.keras 接口中实现自定义网络层，需要以下几个步骤。

- (1) 将自己的层定义成类，并继承 tf.keras.layers.Layer 类。
- (2) 在类中实现 __init__ 方法，用来对该层进行初始化。
- (3) 在类中实现 build 方法，用于定义该层所使用的权重。
- (4) 在类中实现 call 方法，用来相应调用事件。对输入的数据做自定义处理，同时还可以支持 masking (根据实际的长度进行运算)。
- (5) 在类中实现 compute_output_shape 方法，指定该层最终输出的 shape。

按照以上步骤，结合 8.1.11 小节中的描述，实现带有位置向量的词嵌入层。

具体代码如下：

代码 8-10 keras 注意力机制模型

```
01 import tensorflow as tf
02 from tensorflow import keras
```

```

03 from tensorflow.keras import backend as K #载入 keras 的后端实现
04
05 class Position_Embedding(keras.layers.Layer): #定义位置向量类
06     def __init__(self, size=None, mode='sum', **kwargs):
07         self.size = size #定义位置向量的大小, 必须为偶数, 一半是 cos, 一半是 sin
08         self.mode = mode
09         super(Position_Embedding, self).__init__(**kwargs)
10
11     def call(self, x):
12         if (self.size == None) or (self.mode == 'sum'):
13             self.size = int(x.shape[-1])
14             position_j = 1. / K.pow(10000., 2 * K.arange(self.size / 2, dtype='float32') / self.size)
15             position_j = K.expand_dims(position_j, 0)
16             #按照 x 的 1 维数值累计求和, 生成序列。
17             position_i = tf.cumsum(K.ones_like(x[:, :, 0]), 1)-1
18             position_i = K.expand_dims(position_i, 2)
19             position_ij = K.dot(position_i, position_j)
20             position_ij = K.concatenate([K.cos(position_ij), K.sin(position_ij)], 2)
21             if self.mode == 'sum':
22                 return position_ij + x
23             elif self.mode == 'concat':
24                 return K.concatenate([position_ij, x], 2)
25
26     def compute_output_shape(self, input_shape): #设置输出形状
27         if self.mode == 'sum':
28             return input_shape
29         elif self.mode == 'concat':
30             return (input_shape[0], input_shape[1], input_shape[2]+self.size)

```

代码第3行是原生 Keras 框架的内部语法。由于 Keras 框架是一个前端的代码框架，它通过 backend 接口来调用后端框架的实现，以保证后端框架的无关性。

代码第5行定义了类 Position_Embedding，用于实现带有位置向量的词嵌入层。该代码与 8.1.11 小节中代码的不同之处是：它是用 tf.keras 接口实现的，同时也提供了位置向量的两种合入方式。

- 加和方式：通过 sum 运算，直接把位置向量加到原有的词嵌入中。这种方式不会改变原有的维度。
- 连接方式：通过 concat 函数将位置向量与词嵌入连接到一起。这种方式会在原有的词嵌入维度之上扩展出位置向量的维度。

代码第11行是 Position_Embedding 类 call 方法的实现。当调用 Position_Embedding 类进行位置向量生成时，系统会调用该方法。

在 Position_Embedding 类的 call 方法中，先对位置向量的合入方式进行判断，如果是 sum 方式，则将生成的位置向量维度设置成输入的词嵌入向量维度。这样就保证了生成的结果与输入的结果维度统一，在最终的 sum 操作时不会出现错误。

8.4.4 代码实现：用 tf.keras 接口开发注意力层

下面按照 8.1.10 小节中的描述，用 `tf.keras` 接口开发基于内部注意力的多头注意力机制 `Attention` 类。

在 `Attention` 类中用比 8.1.10 小节更优化的方法来实现多头注意力机制的计算。该方法直接将多头注意力机制中最后的全连接网络中的权重提取出来，并将原有的输入 Q 、 K 、 V 按照指定的计算次数展开，使它们彼此以直接矩阵的方式进行计算。

这种方法采用了空间换时间的思想，省去了循环处理，提升了运算效率。

具体代码如下：

代码 8-10 keras 注意力机制模型（续）

```

31 class Attention(keras.layers.Layer):          # 定义注意力机制的模型类
32     def __init__(self, nb_head, size_per_head, **kwargs):
33         self.nb_head = nb_head           # 设置注意力的计算次数 nb_head
34         # 设置每次线性变化为 size_per_head 维度
35         self.size_per_head = size_per_head
36         self.output_dim = nb_head * size_per_head  # 计算输出的总维度
37         super(Attention, self).__init__(**kwargs)
38
39     def build(self, input_shape):          # 实现 build 方法，定义权重
40         self.WQ = self.add_weight(name='WQ',
41                         shape=(int(input_shape[0][-1]), self.output_dim),
42                         initializer='glorot_uniform',
43                         trainable=True)
44         self.WK = self.add_weight(name='WK',
45                         shape=(int(input_shape[1][-1]), self.output_dim),
46                         initializer='glorot_uniform',
47                         trainable=True)
48         self.WV = self.add_weight(name='WV',
49                         shape=(int(input_shape[2][-1]), self.output_dim),
50                         initializer='glorot_uniform',
51                         trainable=True)
52         super(Attention, self).build(input_shape)
53     # 定义 Mask 方法，按照 seq_len 的实际长度对 inputs 进行计算
54     def Mask(self, inputs, seq_len, mode='mul'):
55         if seq_len == None:
56             return inputs
57         else:
58             mask = K.one_hot(seq_len[:, 0], K.shape(inputs)[1])
59             mask = 1 - K.cumsum(mask, 1)
60             for _ in range(len(inputs.shape) - 2):
61                 mask = K.expand_dims(mask, 2)
62             if mode == 'mul':
63                 return inputs * mask
64             if mode == 'add':
65                 return inputs - (1 - mask) * 1e-12

```

```

67     def call(self, x):
68         if len(x) == 3:                                #解析传入的 Q_seq、K_seq、V_seq
69             Q_seq, K_seq, V_seq = x
70             Q_len, V_len = None, None                  #Q_len、V_len 是 mask 的长度
71         elif len(x) == 5:
72             Q_seq, K_seq, V_seq, Q_len, V_len = x
73
74         #对 Q、K、V 做线性变换，一共做 nb_head 次，每次都将维度转化成 size_per_head
75         Q_seq = K.dot(Q_seq, self.WQ)
76         Q_seq = K.reshape(Q_seq, (-1, K.shape(Q_seq)[1], self.nb_head,
77                                   self.size_per_head))
78         Q_seq = K.permute_dimensions(Q_seq, (0, 2, 1, 3)) #排列各维度的顺序。
79         K_seq = K.dot(K_seq, self.WK)
80         K_seq = K.reshape(K_seq, (-1, K.shape(K_seq)[1], self.nb_head,
81                                   self.size_per_head))
82         K_seq = K.permute_dimensions(K_seq, (0, 2, 1, 3))
83         V_seq = K.dot(V_seq, self.WV)
84         V_seq = K.reshape(V_seq, (-1, K.shape(V_seq)[1], self.nb_head,
85                                   self.size_per_head))
86         V_seq = K.permute_dimensions(V_seq, (0, 2, 1, 3))
87         #计算内积，然后计算 mask，再计算 softmax
88         A = K.batch_dot(Q_seq, K_seq, axes=[3, 3]) / self.size_per_head**0.5
89         A = K.permute_dimensions(A, (0, 3, 2, 1))
90         A = self.Mask(A, V_len, 'add')
91         A = K.permute_dimensions(A, (0, 3, 2, 1))
92         A = K.softmax(A)
93
94         #将 A 再与 V 进行内积计算
95         O_seq = K.batch_dot(A, V_seq, axes=[3, 2])
96         O_seq = K.permute_dimensions(O_seq, (0, 2, 1, 3))
97         O_seq = K.reshape(O_seq, (-1, K.shape(O_seq)[1], self.output_dim))
98         O_seq = self.Mask(O_seq, Q_len, 'mul')
99
100        return O_seq
101
102    def compute_output_shape(self, input_shape):
103        return (input_shape[0][0], input_shape[0][1], self.output_dim)

```

在代码第 39 行的 build 方法中，为注意力机制中的三个角色 Q 、 K 、 V 分别定义了对应的权重。该权重的形状为 [input_shape, output_dim]。其中：

- input_shape 是 Q 、 K 、 V 中对应角色的输入维度。
- output_dim 是输出的总维度，即注意力的运算次数与每次输出的维度乘积（见代码 36 行）。



提示：

多头注意力机制在多次计算时权重是不共享的，这相当于做了多少次注意力计算，就定义多少个全连接网络。所以在代码第 39~51 行，将权重的输出维度定义成注意力的运算次数与每次输出的维度乘积。

代码第 77 行调用了 `K.permute_dimensions` 函数，该函数实现对输入维度的顺序调整，相当于 `transpose` 函数的作用。

代码第 67 行是 `Attention` 类的 `call` 函数，其中实现了注意力机制的具体计算方式，步骤如下：

- (1) 对注意力机制中的三个角色的输入 Q 、 K 、 V 做线性变化（见代码第 75~83 行）。
- (2) 调用 `batch_dot` 函数，对第 (1) 步线性变化后的 Q 和 K 做基于矩阵的相乘计算（见代码第 85~89 行）。
- (3) 调用 `batch_dot` 函数，对第 (2) 步的结果与第 (1) 步线性变化后的 V 做基于矩阵的相乘计算（见代码第 85~89 行）。



提示：

这里的全连接网络是不带偏置权重 b 的。没有偏置权重的全连接网络在对数据处理时，本质上与矩阵相乘运算是一样的。

因为在整个计算过程中，需要将注意力中的三个角色 Q 、 K 、 V 进行矩阵相乘，并且在最后还要与全连接中的矩阵相乘，所以可以将这个过程理解为是 Q 、 K 、 V 与各自的全连接权重进行矩阵相乘。因为乘数与被乘数的顺序是与结果无关的，所以在代码第 67 行的 `call` 方法中，全连接权重最先参与了运算，并不会影响实际结果。

8.4.5 代码实现：用 `tf.keras` 接口训练模型

用定义好的词嵌入层与注意力层搭建模型，进行训练。具体步骤如下：

- (1) 用 `Model` 类定义一个模型，并设置好输入/输出的节点。
- (2) 用 `Model` 类中的 `compile` 方法设置反向优化的参数。
- (3) 用 `Model` 类的 `fit` 方法进行训练。

具体代码如下：

代码 8-9 用 keras 注意力机制模型分析评论者的情绪（续）

```

23 #数据对齐
24 x_train = tf.keras.preprocessing.sequence.pad_sequences(x_train,
maxlen=maxlen)
25 x_test = tf.keras.preprocessing.sequence.pad_sequences(x_test,
maxlen=maxlen)
26 print('Pad sequences x_train shape:', x_train.shape)
27
28 #定义输入节点
29 S_inputs = tf.keras.layers.Input(shape=(None,), dtype='int32')
30
31 #生成词向量
32 embeddings = tf.keras.layers.Embedding(num_words, 128)(S_inputs)
33 embeddings = attention_keras.Position_Embedding()(embeddings) #默认使用同等
维度的位置向量

```

```

34 #用内部注意力机制模型处理
35 O_seq =
36     attention_keras.Attention(8,16)([embeddings,embeddings,embeddings])
37
38 #将结果进行全局池化
39 O_seq = tf.keras.layers.GlobalAveragePooling1D()(O_seq)
40 #添加dropout
41 O_seq = tf.keras.layers.Dropout(0.5)(O_seq)
42 #输出最终节点
43 outputs = tf.keras.layers.Dense(1, activation='sigmoid')(O_seq)
44 print(outputs)
45 #将网络结构组合到一起
46 model = tf.keras.models.Model(inputs=S_inputs, outputs=outputs)
47
48 #添加反向传播节点
49 model.compile(loss='binary_crossentropy', optimizer='adam',
  metrics=['accuracy'])
50
51 #开始训练
52 print('Train...')
53 model.fit(x_train, y_train, batch_size=batch_size, epochs=5,
  validation_data=(x_test, y_test))

```

代码第 36 行构造了一个列表对象作为输入参数。该列表对象里含有 3 个同样的元素——embeddings，表示使用的是内部注意力机制。

代码第 39~44 行，将内部注意力机制的结果 O_seq 经过全局池化和一个全连接层处理得到了最终的输出节点 outputs。节点 outputs 是一个 1 维向量。

代码第 49 行，用 model.compile 方法，构建模型的反向传播部分，使用的损失函数是 binary_crossentropy，优化器是 adam。

8.4.6 运行程序

代码运行后，生成以下结果：

```

Epoch 1/5
25000/25000 [=====] - 42s 2ms/step - loss: 0.5357 - acc:
0.7160 - val_loss: 0.5096 - val_acc: 0.7533
Epoch 2/5
25000/25000 [=====] - 36s 1ms/step - loss: 0.3852 - acc:
0.8260 - val_loss: 0.3956 - val_acc: 0.8195
Epoch 3/5
25000/25000 [=====] - 36s 1ms/step - loss: 0.3087 - acc:
0.8710 - val_loss: 0.4135 - val_acc: 0.8184
Epoch 4/5
25000/25000 [=====] - 36s 1ms/step - loss: 0.2404 - acc:
0.9011 - val_loss: 0.4501 - val_acc: 0.8094
Epoch 5/5

```

```
25000/25000 [=====] - 35s 1ms/step - loss: 0.1838 - acc: 0.9289 - val_loss: 0.5303 - val_acc: 0.8007
```

可以看到，整个数据集迭代 5 次后，准确率达到了 80% 以上。



提示：

本节实例代码可以直接在 TensorFlow 1.x 与 2.x 两个版本中运行，不需要任何改动。

8.4.7 扩展：用 Targeted Dropout 技术进一步提升模型的性能

在 8.4.5 小节中的代码第 41 行，用 Dropout 增强了网络的泛化性。这里再介绍一种更优的 Dropout 技术——Targeted Dropout。

Targeted Dropout 不再像原有的 Dropout 那样按照设定的比例随机丢弃部分节点，而是对现有的神经元进行排序，按照神经元的权重重要性来丢弃节点。这种方式比随机丢弃的方式更智能，效果更好。更多理论见以下论文：

<https://openreview.net/pdf?id=HkgHWScuoQ>

1. 代码实现

Targeted Dropout 代码已经集成到代码文件“8-10 keras 注意力机制模型.py”中，这里不再展开介绍。使用时直接将 8.4.5 小节中的代码第 41 行改成 TargetedDropout 函数调用即可。具体请参考本书配套资源中的代码。

2. 运行效果

运行使用 Targeted Dropout 技术的代码，输出以下结果：

```
Epoch 1/5
25000/25000 [=====] - 32s 1ms/step - loss: 0.4388 - acc: 0.7950 - val_loss: 0.4041 - val_acc: 0.8234
Epoch 2/5
25000/25000 [=====] - 25s 1ms/step - loss: 0.3368 - acc: 0.8590 - val_loss: 0.3725 - val_acc: 0.8316
Epoch 3/5
25000/25000 [=====] - 25s 1ms/step - loss: 0.2491 - acc: 0.8947 - val_loss: 0.3758 - val_acc: 0.8334
Epoch 4/5
25000/25000 [=====] - 25s 1ms/step - loss: 0.1609 - acc: 0.9326 - val_loss: 0.4496 - val_acc: 0.8274
Epoch 5/5
25000/25000 [=====] - 25s 1ms/step - loss: 0.0961 - acc: 0.9609 - val_loss: 0.6461 - val_acc: 0.8194
```

从结果可以看出，最终的准确率为 0.8194，与 8.4.6 小节的结果（0.8007）相比，准确率得到了提升。

8.5 实例43：搭建YOLO V3模型，识别图片中的酒杯、水果等物体

YOLO模型是目标检测领域的经典模型，目前已经发展到V3版本。本实例将搭建一个YOLO V3模型的正向结构，让读者快速掌握目标检测算法。

实例描述

搭建YOLO V3模型，并加载现有的预训练权重。对任意一张图片进行目标检测，并在图上标出识别出来的物体名称及位置。

下面先介绍YOLO模型的原理，接着搭建网络的结构，然后加载COCO数据集（见8.7.1小节）上的预训练模型，最终完成对图片的检测。

8.5.1 YOLO V3模型的样本与结构

YOLO V3模型属于监督式训练模型。训练该模型所使用的样本需要包含两部分的标注信息：

- 物体的位置坐标（矩形框）。
- 物体的所属类别。

将样本中的图片作为输入，将图片上的物体类别及位置坐标作为标签，对模型进行训练。

最终得到的模型将会具有计算物体位置坐标及识别物体类别的能力。

在YOLO V3模型中，主要通过两部分结构来完成物体位置坐标计算和分类预测。

- 特征提取部分：用于提取图像特征。
- 检测部分：用于对提取的特征进行处理，预测出图像的边框坐标（bounding box）和标签（label）。

YOLO V3模型的更多信息可以参考以下链接中的论文：

<https://pjreddie.com/media/files/papers/YOLOv3.pdf>

1. 特征提取部分（Darknet-53模型）

在YOLO V3模型中用Darknet-53模型来提取特征。该模型包括52个卷积层和1个平均池化层，如图8-17所示。

在实际的使用中，没有用最后的全局平均池化层，只用了Darknet-53模型中的第52层。

2. 检测部分（YOLO V3模型）

YOLO V3模型的检测部分所完成的步骤如下。

- (1) 将Darknet-53模型提取到的特征输入检测块中进行处理。
- (2) 在检测块处理之后，生成具有bbox attrs单元的检测结果。
- (3) 根据bbox attrs单元检测到的结果在原有的图片上进行标注，完成检测任务。

	类型	卷积核个数	大小	输出
1x	Convolutional	32	3×3	256×256
1x	Convolutional	64	$3 \times 3 / 2$	128×128
1x	Convolutional	32	1×1	①
1x	Convolutional	64	3×3	
1x	Residual			128×128
2x	Convolutional	128	$3 \times 3 / 2$	64×64
2x	Convolutional	64	1×1	②
2x	Convolutional	128	3×3	
2x	Residual			64×64
8x	Convolutional	256	$3 \times 3 / 2$	32×32
8x	Convolutional	128	1×1	③
8x	Convolutional	256	3×3	
8x	Residual			32×32
8x	Convolutional	512	$3 \times 3 / 2$	16×16
8x	Convolutional	256	1×1	④
8x	Convolutional	512	3×3	
8x	Residual			16×16
4x	Convolutional	1024	$3 \times 3 / 2$	8×8
4x	Convolutional	512	1×1	⑤
4x	Convolutional	1024	3×3	
4x	Residual			8×8
	Avgpool		Global	
	Connected		1000	
	Softmax			

图 8-17 Darknet-53 模型的结构

bbox attrs 单元的维度为 “ $5+C$ ” 。其中：

- 5 代表边框坐标为 5 维，包括中心坐标 (x, y) 、长宽 (h, w) 、目标得分（置信度）。
 - C 代表具体分类的个数。
- 具体细节见下面的代码。

8.5.2 代码实现：Darknet-53 模型的 darknet 块

如图 8-17 所示，Darknet-53 模型由多个 darknet 块组成（见图 8-17 中的带有标注的方块），所有 darknet 块都具有一样的结构：由两个卷积（卷积核分别为 1 和 3）与一个残差链接组成。darknet 块的具体代码如下：

代码 8-11 yolo_v3

```

01 import numpy as np
02 import tensorflow as tf
03
04 slim = tf.contrib.slim
05
06 # 定义 darknet 块：一个短链接加一个同尺度卷积，再加一个下采样卷积
07 def _darknet53_block(inputs, filters):
08     shortcut = inputs
09     inputs = slim.conv2d(inputs, filters, 1, stride=1, padding='SAME') # 正常卷积
10    inputs = slim.conv2d(inputs, filters * 2, 3, stride=1, padding='SAME') # 正常卷积

```

```

11
12     inputs = inputs + shortcut
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

```

这里使用的是 SAME 卷积，并且步长为 1，表示每次卷积只改变通道数，并没有改变高和宽的尺寸。

8.5.3 代码实现：Darknet-53 模型的下采样卷积

如图 8-17 所示，每两个 darknet 块之间都有一个单独的卷积层。它们都是下采样卷积，是将原有的输入补 0，再通过步长为 2、卷积核为 3 的 VALID 卷积来实现。具体代码如下：

代码 8-11 yolo_v3 (续)

```

14 def _conv2d_fixed_padding(inputs, filters, kernel_size, strides=1):
15     assert strides>1
16
17     inputs = _fixed_padding(inputs, kernel_size) # 外围填充 0，支持 VALID 卷积
18     inputs = slim.conv2d(inputs, filters, kernel_size, stride=strides,
19                         padding='VALID')
20
21     return inputs
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

```

这里用 tf.pad 函数对输入进行补 0（见代码第 29 行）。

8.5.4 代码实现：搭建 Darknet-53 模型，并返回 3 种尺度特征值

按照图 8-17 所示的结构将网络堆叠起来。具体代码如下：

代码 8-11 yolo_v3 (续)

```

32 def darknet53(inputs): # 定义 Darknet-53 模型，返回 3 种不同尺度的特征
33     inputs = slim.conv2d(inputs, 32, 3, stride=1, padding='SAME') # 正常卷积
34
35     # 需要对输入数据进行补 0 操作，并使用了 VALID 卷积，卷积后的形状为 (-1, 208, 208, 64)
36     inputs = _conv2d_fixed_padding(inputs, 64, 3, strides=2)

```

```

37     inputs = _darknet53_block(inputs, 32)
#darknet 块
38     inputs = _conv2d_fixed_padding(inputs, 128, 3, strides=2)
39
40     for i in range(2):
41         inputs = _darknet53_block(inputs, 64)
42     inputs = _conv2d_fixed_padding(inputs, 256, 3, strides=2)
43
44     for i in range(8):
45         inputs = _darknet53_block(inputs, 128)
46     route_1 = inputs #特征1 (-1, 52, 52, 128)
47
48     inputs = _conv2d_fixed_padding(inputs, 512, 3, strides=2)
49     for i in range(8):
50         inputs = _darknet53_block(inputs, 256)
51     route_2 = inputs #特征2 (-1, 26, 26, 256)
52
53     inputs = _conv2d_fixed_padding(inputs, 1024, 3, strides=2)
54     for i in range(4):
55         inputs = _darknet53_block(inputs, 512) #特征3 (-1, 13, 13, 512)
56     #在原有的 darknet_53 模型中还会做一个全局池化操作，这里没有做，所以其实是只有 52 层
57     return route_1, route_2, inputs

```

Darknet-53 模型并没有只返回最后的特征结果，而是将倒数 3 个 darknet 块的结果返回（见图 8-17 中标注的 3、4、5 部分）。这三个返回值有不同的尺度（52、26、13），是为了给 YOLO 检测模块提供更丰富的视野特征。

8.5.5 代码实现：定义 YOLO 检测模块的参数及候选框

在 YOLO V3 模型中使用了候选框技术。该技术用于辅助 YOLO 检测模块对目标尺寸的计算，以提升 YOLO 检测模块的准确率。

候选框来自于训练模型时的数据集样本。即在模型训练时，对数据集的标注样本进行聚类分析，得到具体的尺寸（见 7.6 节的聚类 COCO 数据集实例）。

候选框可以代表目标样本中最常见的尺寸。在训练或测试模型时，将这些尺寸数据作为先验知识一起放到模型里，可以提高模型的准确率。具体代码如下：

代码 8-11 yolo_v3（续）

```

58 _BATCH_NORM_DECAY = 0.9
59 _BATCH_NORM_EPSILON = 1e-05
60 _LEAKY_RELU = 0.1
61
62 #定义候选框，来自 coco 数据集
63 _ANCHORS = [(10, 13), (16, 30), (33, 23), (30, 61), (62, 45), (59, 119), (116,
90), (156, 198), (373, 326)]

```

因为代码中使用的模型是通过 COCO 数据集训练出的, 所以要将 COCO 数据集的候选框数据放到代码里。

8.5.6 代码实现: 定义 YOLO 检测块, 进行多尺度特征融合

在 YOLO V3 模型中, 检测部分的模型是由一个 YOLO 检测块加一个检测层组成的。YOLO 检测块负责进一步提取特征; 检测层负责将最终的特征转化为 bbox attrs 单元(见 8.5.1 小节的“2. 检测部分”)。

其中 YOLO 检测块的代码如下:

代码 8-11 yolo_v3 (续)

```
64 #YOLO 检测块
65 def _yolo_block(inputs, filters):
66     inputs = slim.conv2d(inputs, filters, 1, stride=1, padding='SAME')
67     #正常卷积
68     inputs = slim.conv2d(inputs, filters * 2, 3, stride=1, padding='SAME')
69     #正常卷积
70     inputs = slim.conv2d(inputs, filters, 1, stride=1, padding='SAME')
71     #正常卷积
72     inputs = slim.conv2d(inputs, filters * 2, 3, stride=1, padding='SAME')
73     #正常卷积
74     route = inputs
75     inputs = slim.conv2d(inputs, filters * 2, 3, stride=1, padding='SAME')
76     #正常卷积
77     return route, inputs
```

在 YOLO V3 模型中, 函数 yolo_block 会被多次调用, 用于将 darknet 块返回的多个不同尺度的特征结果(见图 8-17 中标注的 3、4、5 部分)融合起来, 见 5.8.5 小节。

在函数 yolo_block 中, 有两个返回值: route 与 inputs。返回值 route 用于配合下一个尺度的特征一起进行计算; 返回值 inputs 用于输入检测层进行 bbox attrs 单元的计算(见 8.5.7 小节)。

8.5.7 代码实现: 将 YOLO 检测块的特征转化为 bbox attrs 单元

下面将定义函数 detection_layer, 以实现检测层的功能。函数 detection_layer 中的具体步骤如下:

- (1) 将每个尺度的像素展开, 当作预测结果的个数。
- (2) 按照候选框的个数, 为每个预测结果生成对应个数的 bbox attrs 单元。

在本实例中, 候选框 anchors 的个数为 3, 于是该函数会计算出 $3 \times w \times h$ 个 bbox attrs 单元(w 和 h 是输入特征的宽和高)。具体代码如下:

(1) 用函数 darknet3 得到 3 种尺度的特征, route 1 对应 route 2 对应 inputs 对象。

代码 8-11 yolo v3 (续)

```
74 def _detection_layer(inputs, num_classes, anchors, img_size, data_format):
#定义检测函数
75
76     print(inputs.get_shape())
77     num_anchors = len(anchors) #候选框的个数
78     predictions = slim.conv2d(inputs, num_anchors * (5 + num_classes), 1,
79     stride=1, normalizer_fn=None,
80                         activation_fn=None,
81     biases_initializer=tf.zeros_initializer())
82
83     shape = predictions.get_shape().as_list()
84     print("shape", shape) #3个尺度的形状分别为: [1, 13, 13, 3*(5+c)]、[1, 26, 26,
85     3*(5+c)]、[1, 52, 52, 3*(5+c)]
86
87     grid_size = shape[1:3] #取 NHWC 中的宽和高
88     dim = grid_size[0] * grid_size[1] #每个格子所包含的像素
89     bbox_attrs = 5 + num_classes
90
91     predictions = tf.reshape(predictions, [-1, num_anchors * dim,
92     bbox_attrs]) #把 h 和 w 展开成 dim
93
94     stride = (img_size[0] // grid_size[0], img_size[1] // grid_size[1]) #缩放参数 32 (416/13)
95
96     anchors = [(a[0] / stride[0], a[1] / stride[1]) for a in anchors] #将候选框的尺寸同比例缩小
97
98     #将包含边框的单元属性拆分
99     box_centers, box_sizes, confidence, classes = tf.split(predictions, [2,
100    1, num_classes], axis=-1)
101
102     box_centers = tf.nn.sigmoid(box_centers)
103     confidence = tf.nn.sigmoid(confidence)
104
105     grid_x = tf.range(grid_size[0], dtype=tf.float32) #定义网格索引 0,1,2,...n-1
106     grid_y = tf.range(grid_size[1], dtype=tf.float32) #定义网格索引 0,1,2,...m-1
107     a, b = tf.meshgrid(grid_x, grid_y) #生成网格矩阵 a0, a1,...an (共 M 行),
108     b0, b1,...bn (共 n 行), 第 2 行是 b1
109
110     x_offset = tf.reshape(a, (-1, 1)) #展开, 一共 dim 个
111     y_offset = tf.reshape(b, (-1, 1))
112
113     x_y_offset = tf.concat([x_offset, y_offset], axis=-1) #连接 x、y
114     x_y_offset = tf.reshape(tf.tile(x_y_offset, [1, num_anchors]), [1, -1,
115    2]) #按候选框的个数复制 x、y
```

```

109     box_centers = box_centers + x_y_offset #box_centers 是 0~1 之间的数,
110     x_y_offset 是具体网格的索引, 两者相加后就是真实位置(0.1+4=4.1, 第 4 个网格里 0.1 的偏
111     移)
112     anchors = tf.tile(anchors, [dim, 1])      #按第 0 维进行复制, 并复制 dim 份
113     box_sizes = tf.exp(box_sizes) * anchors      #计算边长: hw
114     box_sizes = box_sizes * stride            #真实边长
115
116     detections = tf.concat([box_centers, box_sizes, confidence], axis=-1)
117     classes = tf.nn.sigmoid(classes)
118     predictions = tf.concat([detections, classes], axis=-1) #将转化后的结果
119     合起来
120     print(predictions.get_shape()) #三个尺度的形状分别为:[1, 507(13×13×3), 5+c]、
121     [1, 2028, 5+c]、[1, 8112, 5+c]
122     return predictions           #返回预测值

```

代码第 99、100 行引入了网格的概念。主要用于将当前的坐标映射到图片真实坐标。例如：将 416 pixel×416 pixel 的原始图片转化矩阵形状为 13×13 大小的特征数据。可以理解为，将原始图片缩小了 32 倍，或是原始图片被等比例分成了 13 个网格。

同时，在代码第 99、100 行又用 range 函数生成了一个序列的数据。该代码可以理解成：为每个网格生成一个索引。

代码第 96~117 行是生成 bbox attrs 单元的具体操作。在该代码中用了以下小技巧。

- 中心点 box_centers 是使用 sigmoid 函数生成的，它代表在一个网格里的相对位移（即占有一个网格边长的百分比），见代码第 96 行。
- 边长 box_sizes 增加了指数变换，这是为了保证其值永远为正，并支持用 SGD 算法来反向求导。这里预测值的意义是对原始尺寸进行缩放的比例。所以，让其与候选框 anchors 的尺度相乘，来获得真实的边长。
- 置信度 confidence 是用 sigmoid 函数生成的，表示准确度的分数，值为 0~100% 之间的百分数，见代码第 97 行。
- 分类值 classes 也是用 sigmoid 函数生成的，表示被识别的物体分类不再互斥（即同一个物体可以被划分在多个类型中），见代码第 117 行。



提示：

在整个 YOLO V3 模型中，并没有去具体计算物体的矩形框坐标，而是采用预测中心点的偏移比例与边长相对于候选框的缩放比例来实现坐标定位。

8.5.8 代码实现：实现 YOLO V3 的检测部分

定义函数 yolo_v3，并实现以下步骤。

- (1) 用函数 darknet53 得到 3 种尺度的特征：route_1 对象、route_2 对象、inputs 对象（分

别代表特征 3、2、1)。

- (2) 将代表特征 1 的 inputs 对象传入 YOLO 检测块的 yolo_block 函数中进行处理。
 - (3) 将 YOLO 检测块的返回值放到检测层 detection_layer 函数中，进行 bbox attrs 单元的计算。
 - (4) 将第(2)步的结果经过一次卷积变化，然后进行上采样操作。
 - (5) 将第(4)步的结果与 route_2 对象连接起来，传入 YOLO 检测块的 yolo_block 函数中进行处理。
 - (6) 执行第(3)、(4)步实现特征 1 和特征 2 的融合并检测。
 - (7) 将第(6)步的结果与 route_3 对象连接起来，传入 YOLO 检测块的 yolo_block 函数中进行处理。
 - (8) 再次执行第(3)、(4)步，实现特征 2 和特征 3 的融合与检测。
 - (9) 将最终目标检测的结果合并起来返回。
- 其中，第(2)~(8)步的操作可以理解成为一个 FPN (特征金字塔网络)，见 8.7.9 小节的详细介绍。
- 第(4)步的上采样操作是为了让当前尺寸与下一个特征的尺寸保持一致（见代码第 161、170 行）。整体过程如图 8-18 所示。

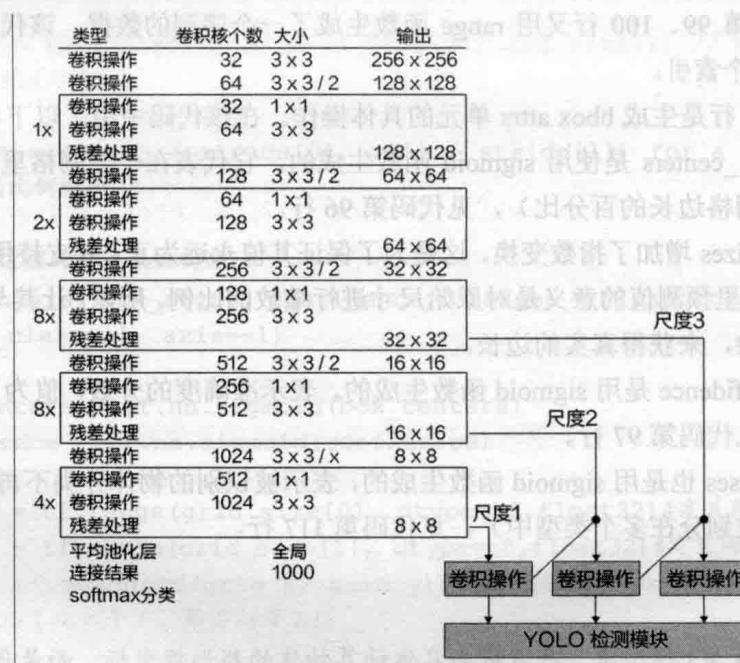


图 8-18 YOLO V3 的多尺度检测

具体代码如下：

代码 8-11 yolo_v3 (续)

```
121 # 定义上采样函数
122 def _upsample(inputs, out_shape):
123     # 由于上采样的填充方式不同, tf.image.resize_bilinear 会对结果影响很大
```

```

124     inputs = tf.image.resize_nearest_neighbor(inputs, (out_shape[1],
125                                                 out_shape[2]))
126     inputs = tf.identity(inputs, name='upsampled')
127     return inputs
128
129 # 定义函数，构建 YOLO V3 模型
130 def yolo_v3(inputs, num_classes, is_training=False, data_format='NHWC',
131             reuse=False):
132     assert data_format=='NHWC'
133     img_size = inputs.get_shape().as_list()[1:3] # 获得输入图片的大小
134
135     inputs = inputs / 255 # 归一化处理
136
137     # 定义批量归一化参数
138     batch_norm_params = {
139         'decay': _BATCH_NORM_DECAY,
140         'epsilon': _BATCH_NORM_EPSILON,
141         'scale': True,
142         'is_training': is_training,
143         'fused': None,
144     }
145
146     # 定义 YOLOV3 模型
147     with slim.arg_scope([slim.conv2d, slim.batch_norm],
148                         data_format=data_format, reuse=reuse):
149         with slim.arg_scope([slim.conv2d], normalizer_fn=slim.batch_norm,
150                             normalizer_params=batch_norm_params,
151                             biases_initializer=None, activation_fn=lambda x:
152                                 tf.nn.leaky_relu(x, alpha=_LEAKY_RELU)):
153             with tf.variable_scope('darknet-53'):
154                 route_1, route_2, inputs = darknet53(inputs)
155
156             with tf.variable_scope('yolo-v3'):
157                 route, inputs = _yolo_block(inputs, 512) # (-1, 13, 13, 1024)
158                 # 用候选框参数来辅助识别
159                 detect_1 = _detection_layer(inputs, num_classes,
160 _ANCHORS[6:9], img_size, data_format)
161                 detect_1 = tf.identity(detect_1, name='detect_1')
162
163             inputs = slim.conv2d(route, 256, 1, stride=1, padding='SAME') #
正常卷积
164             upsample_size = route_2.get_shape().as_list()
165             inputs = _upsample(inputs, upsample_size)
166             inputs = tf.concat([inputs, route_2], axis=3)

```

```

164         route, inputs = _yolo_block(inputs, 256) # (-1, 26, 26, 512)
165         detect_2 = _detection_layer(inputs, num_classes,
166             _ANCHORS[3:6], img_size, data_format)
167         detect_2 = tf.identity(detect_2, name='detect_2')
168
169         inputs = slim.conv2d(route, 128, 1, stride=1, padding='SAME') #
正常卷积
170         upsample_size = route_1.get_shape().as_list()
171         inputs = _upsample(inputs, upsample_size)
172         inputs = tf.concat([inputs, route_1], axis=3)
173
174         _, inputs = _yolo_block(inputs, 128) # (-1, 52, 52, 256)
175
176         detect_3 = _detection_layer(inputs, num_classes,
177             _ANCHORS[0:3], img_size, data_format)
178         detect_3 = tf.identity(detect_3, name='detect_3')
179
180         detections = tf.concat([detect_1, detect_2, detect_3], axis=1)
181         detections = tf.identity(detections, name='detections')
182         return detections#返回了3个尺度。每个尺度里又包含3个结果—— -1、
10647 ( 507 +2028 + 8112 ) 、5+c

```

代码第 124 行，在上采样时使用了 `nearest_neighbor` 方法。这是个很重要的点，如果改用二插值等其他方式，则会对模型的识别率影响很大。

函数 `yolo_V3` 的检测结果是一个 `(1,10647,5+c)` 形状的数据。其中 10647 代表了一副图片被检测出 10647 种结果。它是由 3 个尺度特征的检测结果 `(507, 2028, 8112)` 合并起来的。“`5+c`”是每一个结果的描述单元，即 `bbox attrs`。

代码第 138 行，定义了批量正则化的参数。由于本实例是直接运行 YOLOV3 的预训练模型，所以 `is_training` 的默认值是 `False`，在训练时还需将该值改为 `True`。



提示：

代码第 157、166、176 行，用到了一个函数 `tf.identity`。它的意义是恒等变换，即在图中增加一个节点。以 `detect_3` 为例，将张量 `detect_3` 转化为节点名为“`detect_3`”的操作符。这会使整个网络节点看上去更加规整，健壮性更好。

8.5.9 代码实现：用非极大值抑制算法对检测结果去重

YOLO 检测块从一张图片中检测出 10647 个结果。其中很有可能会出现重复物体（中心和大小略有不同）的情况。为了能够保留检测结果的唯一性，还要使用非极大值抑制（non-max suppression, Nms）的算法对 10647 个结果进行去重。

非极大值抑制算法的过程很简单：

(1) 从所有的检测框中找到置信度较大（置信度大于某个阈值）的那个框。

(2) 挨个计算其与剩余框的区域面积的重叠度(intersection over union, IOU)。

(3) 按照 IOU 阈值过滤。如果 IOU 大于一定阈值(重合度过高)，则将该框剔除。

(4) 对剩余的检测框重复上述过程，直到处理完所有的检测框。

整个过程中，用到的置信度阈值与 IOU 阈值需要提前给定。

另外，在去重之前还需要对坐标进行转换。

因为生成的坐标是中心点、高宽的形式，所以需要对其进行转化，变为左上角的坐标和右下角的坐标。这里用函数 detections_boxes 来实现。计算区域重叠度的算法用函数_iou 来实现。具体代码如下：

代码 8-12 用 YOLO V3 模型进行实物检测

```

01 import numpy as np
02 import tensorflow as tf
03 from PIL import Image, ImageDraw
04 yolo_model = __import__("8-11_yolo_v3")
05 yolo_v3 = yolo_model.yolo_v3
06
07 size = 416
08 input_img = 'timg.jpg'          # 输入文件名称
09 output_img = 'out.jpg'          # 输出文件名称
10 class_names = 'coco.names'      # 样本标签名称
11 weights_file = 'yolov3.weights' # 预训练模型文件名称
12 conf_threshold = 0.5           # 置信度阈值
13 iou_threshold = 0.4            # 重叠区域阈值
14
15 # 定义函数：将中心点、高、宽坐标转化为[x0, y0, x1, y1]形式
16 def detections_boxes(detections):
17     center_x, center_y, width, height, attrs = tf.split(detections, [1, 1,
18     1, 1, -1], axis=-1)
19     w2 = width / 2
20     h2 = height / 2
21     x0 = center_x - w2
22     y0 = center_y - h2
23     x1 = center_x + w2
24     y1 = center_y + h2
25
26     boxes = tf.concat([x0, y0, x1, y1], axis=-1)
27     detections = tf.concat([boxes, attrs], axis=-1)
28     return detections
29
30 # 定义函数计算两个框的内部重叠情况 (IOU), box1、box2 为左上、右下的坐标 [x0, y0, x1, x2]
31 def _iou(box1, box2):
32     b1_x0, b1_y0, b1_x1, b1_y1 = box1
33     b2_x0, b2_y0, b2_x1, b2_y1 = box2
34

```

```

35     int_x0 = max(b1_x0, b2_x0)
36     int_y0 = max(b1_y0, b2_y0)
37     int_x1 = min(b1_x1, b2_x1)
38     int_y1 = min(b1_y1, b2_y1)
39
40     int_area = (int_x1 - int_x0) * (int_y1 - int_y0)
41
42     b1_area = (b1_x1 - b1_x0) * (b1_y1 - b1_y0)
43     b2_area = (b2_x1 - b2_x0) * (b2_y1 - b2_y0)
44
45     # 分母加上 "1e-05"，避免除数为 0
46     iou = int_area / (b1_area + b2_area - int_area + 1e-05)
47     return iou
48
49
50 # 用 NMS 方法对结果去重
51 def non_max_suppression(predictions_with_boxes, confidence_threshold,
52                           iou_threshold=0.4):
53
54     conf_mask = np.expand_dims((predictions_with_boxes[:, :, 4] >
55                                 confidence_threshold), -1)
56     predictions = predictions_with_boxes * conf_mask
57
58     result = {}
59     for i, image_pred in enumerate(predictions):
60         shape = image_pred.shape
61         non_zero_idxs = np.nonzero(image_pred)
62         image_pred = image_pred[non_zero_idxs]
63         image_pred = image_pred.reshape(-1, shape[-1])
64
65         bbox_attrs = image_pred[:, :5]
66         classes = image_pred[:, 5:]
67         classes = np.argmax(classes, axis=-1)
68
69         unique_classes = list(set(classes.reshape(-1)))
70
71         for cls in unique_classes:
72             cls_mask = classes == cls
73             cls_boxes = bbox_attrs[np.nonzero(cls_mask)]
74             cls_boxes = cls_boxes[:, :-1].argsort()[:-1]
75             cls_scores = cls_boxes[:, -1]
76             cls_boxes = cls_boxes[:, :-1]
77
78             while len(cls_boxes) > 0:
79                 box = cls_boxes[0]
80                 score = cls_scores[0]
81                 if not cls in result:
82                     result[cls] = []

```

```

81     result[cls].append((box, score)) 201
82     cls_boxes = cls_boxes[1:] 801
83     ious = np.array([_iou(box, x) for x in cls_boxes]) 801
84     iou_mask = ious < iou_threshold 801
85     cls_boxes = cls_boxes[np.nonzero(iou_mask)] 801
86     cls_scores = cls_scores[np.nonzero(iou_mask)] 801
87
88 return result 801

```

代码第 51 行定义了 non_max_suppression 函数，用来实现 non_max_suppression 算法。其实也可以直接用库函数 tf.image.non_max_suppression 来实现。如果用库函数 tf.image.non_max_suppression，则必须保证当前的 TensorFlow 版本大于 1.8，否则会出现性能问题。

8.5.10 代码实现：载入预训练权重

通过以下网址下载预训练模型文件，并保存到本地。

```
https://pjreddie.com/media/files/yolov3.weights
```

该预训练模型文件是通过 COCO 数据集训练好的 YOLO V3 模型文件。该文件是二进制格式的。在文件中，前 5 个 int32 值是标题信息，包括以下 4 部分内容：

- 主要版本号（占 1 个 int32 空间）。
- 次要版本号（占 1 个 int32 空间）。
- 子版本号（占 1 个 int32 空间）。
- 训练图像个数（占 2 个 int32 空间）。

在标题信息之后，便是网络的权重。

该权重的存储格式以行为主。在使用时，需要先将其转成以列为主。具体代码如下：

代码 8-12 用 YOLO V3 模型进行实物检测（续）

```

89 #加载权重
90 def load_weights(var_list, weights_file):
91
92     with open(weights_file, "rb") as fp:
93         _ = np.fromfile(fp, dtype=np.int32, count=5) #跳过前 5 个 int32
94         weights = np.fromfile(fp, dtype=np.float32)
95
96     ptr = 0
97     i = 0
98     assign_ops = []
99     while i < len(var_list) - 1:
100         var1 = var_list[i]
101         var2 = var_list[i + 1]
102         #找到卷积项
103         if 'Conv' in var1.name.split('/')[-2]:
104             #找到 BN 参数项

```

```

105     if 'BatchNorm' in var2.name.split('/')[-2]: set
106         #加载批量归一化参数 [1] sexed_slo = sexed_slo
107         gamma, beta, mean, var = var_list[i+1:i+5]
108         batch_norm_vars = [beta, gamma, mean, var]
109         for var in batch_norm_vars:
110             shape = var.shape.as_list()
111             num_params = np.prod(shape)
112             var_weights = weights[ptr:ptr + num_params].reshape(shape)
113             ptr += num_params
114             assign_ops.append(tf.assign(var, var_weights,
115                                         validate_shape=True))
116             i += 4#已经加载了4个变量，指针位移加4
117         elif 'Conv' in var2.name.split('/')[-2]:
118             bias = var2
119             bias_shape = bias.shape.as_list()
120             bias_params = np.prod(bias_shape)
121             bias_weights = weights[ptr:ptr +
122                                   bias_params].reshape(bias_shape)
123             ptr += bias_params
124             assign_ops.append(tf.assign(bias, bias_weights,
125                                         validate_shape=True))
125             i += 1#移动指针
126             var1 = image_psd[i].image
127             shape = var1.shape.as_list()
128             num_params = np.prod(shape)
129             #加载权重
130             var_weights = weights[ptr:ptr + num_params].reshape((shape[3],
131             shape[2], shape[0], shape[1]))
132             var_weights = np.transpose(var_weights, (2, 3, 1, 0))
133             ptr += num_params
134             assign_ops.append(tf.assign(var1, var_weights,
135                                         validate_shape=True))
136             i += 1
137         if i > len(image_psd):
138             break
139     return assign_ops

```

8.5.11 代码实现：载入图片，进行目标实物的识别

用 main 函数完成整体的处理过程，需要先定义以下几个函数：

- 函数 draw_boxes，用于将结果显示在图片上。
- 函数 convert_to_original_size，用于将结果位置还原到真实图片上对应的位置。
- 函数 load_coco_names，用于加载 COCOS 数据集对应的标签名称。

具体的代码如下：

代码 8-12 用 YOLO V3 模型进行实物检测(续)

```

137 #将结果显示在图片上
138 def draw_boxes(bboxes, img, cls_names, detection_size):
139     draw = ImageDraw.Draw(img)
140
141     for cls, bboxes in boxes.items():
142         color = tuple(np.random.randint(0, 256, 3))
143         for box, score in bboxes:
144             box = convert_to_original_size(box, np.array(detection_size),
145                                             np.array(img.size))
146             draw.rectangle(box, outline=color)
147             draw.text(box[:2], '{} {:.2f}%'.format(cls_names[cls], score *
148                                         100), fill=color)
149             print('{} {:.2f}%'.format(cls_names[cls], score * 100), box[:2])
150
151     return draw
152
153
154 #加载数据集的标签名称
155 def load_coco_names(file_name):
156     names = {}
157     with open(file_name) as f:
158         for id, name in enumerate(f):
159             names[id] = name
160     return names
161
162 def main(argv=None):
163     tf.reset_default_graph()
164     img = Image.open(input_img)
165     img_resized = img.resize(size=(size, size))
166
167     classes = load_coco_names(class_names)
168
169     #定义输入占位符
170     inputs = tf.placeholder(tf.float32, [None, size, size, 3])
171
172     with tf.variable_scope('detector'):
173         detections = yolo_v3(inputs, len(classes), data_format='NHWC')#定义
174         #加载权重
175         load_ops = load_weights(tf.global_variables(scope='detector'),
176                                 weights_file)
177         boxes = detections_boxes(detections)
178

```