

```
stacked_rnn = []
stacked_bw_rnn = []
for i in range(3):
    stacked_rnn.append(tf.contrib.rnn.LSTMCell(n_hidden))
    stacked_bw_rnn.append(tf.contrib.rnn.LSTMCell(n_hidden))

outputs, _, _ = rnn.stack_bidirectional_rnn(stacked_rnn, stacked_bw_rnn,
                                            dtype=tf.float32)
```

还可以构建一个多层次cell放到
stack_bidirectional_rnn中，代码如下。

代码9-18 Multi双向RNN

```
stacked_rnn = []
stacked_bw_rnn = []
for i in range(3):
    stacked_rnn.append(tf.contrib.rnn.LSTMCell(n_hidden))
    stacked_bw_rnn.append(tf.contrib.rnn.LSTMCell(n_hidden))

mcell = tf.contrib.rnn.MultiRNNCell(stacked_rnn)
mcell_bw = tf.contrib.rnn.MultiRNNCell(stacked_bw_rnn)

outputs, _, _ = rnn.stack_bidirectional_rnn([mcell], [mcell_bw],
                                            dtype=tf.float32)
```

 **注意：** 使用MultiRNNCell时，虽然是多层次，但是从外表上看仍是一个输入，stack_bidirectional_rnn只关心输入的cell类是不是多个，而不会去识别输入的cell里面是否还包含多个。在这种情况下，就必须将输入用中括号括起来，让其变为list类型。

9.4.13 实例66：构建动态多层双向RNN对MNIST数据集分类

将前面代码中rnn.stack_bidirectional_rnn注释掉，换成stack_bidirectional_dynamic_rnn，输入变成x，同样将输出转置，然后送往下一层，代码如下。

实例描述

演示使用动态多层双向RNN网络对MNIST数据集分类。

代码9-19 动态Multi双向rnn

```
mcell = tf.contrib.rnn.MultiRNNCell(stacked_rnn)
mcell_bw = tf.contrib.rnn.MultiRNNCell(stacked_bw_rnn)

outputs, _, _ = rnn.stack_bidirectional_dynamic_rnn([mcell],
                                                    dtype=tf.float32)
outputs = tf.transpose(outputs, [1, 0, 2])

print(outputs[0].shape, outputs.shape)
pred = tf.contrib.layers.fully_connected(outputs[-1], n_classes,
                                         activation_fn = None)
```

运行代码后，输出的outputs类型如下：

```
(?, 256) (28, ?, 256)
```

前一个括号中是送往下一层的结果，仍然是
*****ebook converter DEMO Watermarks*****

256即正、反向的结果concat，后一个括号中是outputs的形状。

9.4.14 初始RNN

对应于9.4.2节中介绍的构建RNN的初始化cell状态参数，TensorFlow中也封装了对其初始化的方法，一起来看一下。

1. 初始化为0

对于正向或反向，第一个cell传入时没有之前的序列输出值，所以需要对其初始化。一般来讲，不用去刻意指定，系统会默认初始化0，当然也可以手动指定其初始化为0。代码如下。

```
initial_state = lstm_cell.zero_state(batch_size, dtype)
#在后续的cell实例化中，将initial_state传入即可
```

2. 初始化为指定值

在确保创建组成RNN的cell时，设置了输出为元组类型（见表9-1中，创建cell类的初始化参数state_is_tuple=True）的前提下，可以使用LSTMStateTuple函数。但有时想要给lstm_cell的initial state赋予我们想要的值，而不是简单的用0来初始化。

示例：

```
from tensorflow.contrib.rnn.python.ops.core_rnn_cell_impl import LSTMStateTuple
.....
c_state = .....
h_state = .....
# c_state , h_state 都为Tensor
initial_state = LSTMStateTuple(c_state, h_state)
```

9.4.15 优化RNN

RNN的优化技巧有很多，对于前面讲述的神经网络技巧大部分在RNN上都适用，但也有例外，下面就来介绍下RNN自己特有的两个优化方法的处理。

1. dropout功能

在RNN中，如果想使用dropout功能，不能用以前的CNN下的dropout，CNN中：

```
def dropout(x, keep_prob, noise_shape=None, seed=None, name=None)
```

因为RNN有自己的dropout，并且实现方式与RNN不一样：

```
def rnn_cell.DropoutWrapper(rnn_cell, input_keep_prob=1.0, output_keep_prob=1.0):
```

使用举例：

```
lstm_cell=tf.nn.rnn_cell.DropoutWrapper(lstm_cell,output_ke
```

从t-1时刻的状态传递到t时刻进行计算，这中间不进行memory的dropout，仅在同一个t时刻中，多层cell之间传递信息时进行dropout。所以，RNN的dropout方法会有两个设置参数input_keep_prob（传入cell的保留率）和output_keep_prob（输出cell的保留率）

- 如果希望是input传入cell时丢弃掉一部分input信息，就设置input_keep_prob，那么传入到cell的就是部分input。
- 如果希望cell的output只有一部分作为下一层cell的input，就定义为output_keep_prob。

示例代码如下：

```
lstm_cell=tf.nn.rnn_cell.BasicLSTMCell(size,forget_bias=0.0,  
tuple=True)  
lstm_cell=tf.nn.rnn_cell.DropoutWrapper(lstm_cell,output_ke
```

在上面代码中，一个RNN层后面跟一个DropoutWrapper，是一种常见的用法。

2. LN基于层的归一化

这部分内容是对应于批量归一化（BN）的。由于RNN的特殊结构，它的输入不同于前面所讲的全连接、卷积网络。

- 在BN中，每一层的输入只考虑当前批次样本（或批次样本的转化值）即可。

- 但是在RNN中，每一层的输入除了当前批次样本的转化值，还得考虑样本中上一个序列样本的输出值，所以对于RNN的归一化，BN算法不再适用，最小批次覆盖不了全部的输入数据，而是需要对于输入BN的某一层来做归一化，即layer-Normalization。

由于RNN的网络都被LSTM、GRU这样的结构给封装起来，所以想要实现LN并不像BN那样直接在外层添加一个BN层就可以，需要改写LSTM或GRU的cell，对其内部的输入进行归一化处理。

TensorFlow中目前还不支持这样的cell，所以需要开发者自己来改写原有cell的代码，具体的方法可以在下面例子中找到。

9.4.16 实例67：在GRUCell中实现LN

在本例中将改写GRUCell代码来实现LN，该例子是在前面的代码“9-3 LSTMMNist.py”文件中

改写的。

(1) 新加了一个函数LN用于做归一化处理。

(2) 定义一个LNGRU来代替原始的GRU。通过右击原有的代码tf.contrib.rnn.

GRUCell（n_hidden）中GRUCell部分，选择“go to definition”找到GRU实现的代码，全部复制过来，在其__call__函数里修改为如下代码，即完成了属于我们自己的LNGRU类。具体代码如下。

实例描述

手动构建GRUCell的 LN代码，并演示使用该cell对MNIST数据集分类。

代码9-20 lnGRUonMnist

```
01 from tensorflow.python.ops.rnn_cell_impl import _RNNCell
02 from tensorflow.python.ops.math_ops import sigmoid
03 from tensorflow.python.ops.math_ops import tanh
04 from tensorflow.python.ops import variable_scope as vs
05 from tensorflow.python.ops import array_ops
06 from tensorflow.contrib.rnn.python.ops.core_rnn_cell_implement
07 print(tf.__version__)
08 tf.reset_default_graph()
09
10 def ln(tensor, scope = None, epsilon = 1e-5):
11     """ Layer normalizes a 2D tensor along its second axis.
12     assert(len(tensor.get_shape()) == 2)
13     m, v = tf.nn.moments(tensor, [1], keep_dims=True)
14     if not isinstance(scope, str):
15         scope = ''
```

```

16     with tf.variable_scope(scope + 'layer_norm'):
17         scale = tf.get_variable('scale',
18                                shape=[tensor.get_shape(),
19                                       initializer=tf.constant_:
20         shift = tf.get_variable('shift',
21                                shape=[tensor.get_shape(),
22                                       initializer=tf.constant_:
23         LN_initial = (tensor - m) / tf.sqrt(v + epsilon)
24
25     return LN_initial * scale + shift
26
27 class LNGRUCell(RNNCell):
28     """Gated Recurrent Unit cell (cf. http://arxiv.org/abs/1406.1078)."""
29
30     def __init__(self, num_units, input_size=None, activation=tanh):
31         if input_size is not None:
32             print("%s: The input_size parameter is deprecated, "
33                   "use num_units instead." % self.__class__.__name__)
34         self._num_units = num_units
35         self._activation = activation
36
37     @property
38     def state_size(self):
39         return self._num_units
40
41     @property
42     def output_size(self):
43         return self._num_units
44
45     def __call__(self, inputs, state):
46         """Gated recurrent unit (GRU) with nunits cells."""
47         with vs.variable_scope("Gates"):
48             value = _linear([inputs, state], 2 * self._num_units)
49             r, u = array_ops.split(value=value, num_or_size_splits=2,
50                                   axis=1)
51             r = ln(r, scope='r/')
52             u = ln(u, scope='u/')
53             r, u = sigmoid(r), sigmoid(u)
54         with vs.variable_scope("Candidate"):
55             Cand = _linear([inputs, r * state], self._num_units)
56             c_pre = ln(Cand, scope='new_h/')
57             c = self._activation(c_pre)
58             new_h = u * state + (1 - u) * c
59         return new_h, new_h

```

LNGRU定义好之后，直接替换原有的GRU

*****ebook converter DEMO Watermarks*****

使用代码即可。

代码9-20 lnGRUonMnist（续）

```
58 .....
59 #3 gru
60 gru = LNGRUCell(n_hidden)
61 .....
```

其他代码均不用变化，运行后可以得出如下结果：

```
1.1.0-rc2
.....
Iter 93440, Minibatch Loss= 0.022772, Training Accuracy= 1.0
Iter 94720, Minibatch Loss= 0.060210, Training Accuracy= 0.9
Iter 96000, Minibatch Loss= 0.116144, Training Accuracy= 0.9
Iter 97280, Minibatch Loss= 0.057876, Training Accuracy= 0.9
Iter 98560, Minibatch Loss= 0.030294, Training Accuracy= 0.9
Iter 99840, Minibatch Loss= 0.158428, Training Accuracy= 0.9
Finished!
Testing Accuracy: 0.96875
```

生成的结果为0.96，比原来的效果有所提升（原来是0.945）。本例中只是使用了一个GRUcell。在多个cell中，LN的效果会更明显些（多cell的例子可以参考本书附带资源中的代码“9-21 LN多GRU1-1.py”文件），其实质只是在cell中调用了一次LN来处理。

读者可以仿照上面的形式来修改其他的cell。



注意：本例中已经将代码版本打印出来，这表明该例子是与代码强关联的。如果读者不是这个版本，很可能遇到错误。因为不同的版本有可能会修改原始的GRU实现代码，所以请读者记住修改该方法，千万不要直接复制代码。

另外，本书配套资源中还提供了关于1.2.0-rc0的代码修改，可以参考代码“9-22 LN多GRU1-2.py”文件。

9.4.17 CTC网络的loss——`ctc_loss`

CTC网络的loss就不能用平方差了，更不能用交叉熵，它有更为复杂的公式方法，在TensorFlow中已经有现成的封装函数`ctc_loss`。下面来一起学习一下。

1. `ctc_loss`函数介绍

配合前文的CTC，在TensorFlow中提供了一个`ctc_loss`函数，其作用就是按照序列来处理输出标签和标准标签之间的损失。因为也是成型的函数封装，对于初学者内部实现不用花太多时间关注，只要会用即。

```
tf.nn.ctc_loss(labels, inputs, sequence_length, preprocess_col:  
repeated=False, ctc_merge_repeated=True, time_major=True)
```

具体参数说明如下。

- labels：一个int32类型的稀疏矩阵张量(SparseTensor)。
- inputs：（常用变量logits表示）经过RNN后输出的标签预测值，三维的浮点型张量，当time_major为False时形状为[batch_size, max_time, num_classes]，否则为[max_time, batch_size, num_classes]。
- sequence_length：序列长度。
- preprocess_collapse_repeated：是否需要预处理，将重复的label合并成一个label，默认是false。
- ctc_merge_repeated：在计算时是否将每个non_blank（非空）重复的label当成单独的label来解释，默认是true。
- time_major：决定inputs的格式。

对于preprocess_collapse_repeated与ctc_merge_repeated，都是对于ctc_loss中重复标签处理的控制，各种情况组合后如表9-3所示。

表9-3 ctc_loss函数参数情况

参 数 情 况	说 明
preprocess_collapse_repeated=TRUE; ctc_merge_repeated=TRUE;	忽略全部重复标签，只计算不重复的标签
preprocess_collapse_repeated=False; ctc_merge_repeated=TRUE;	标准的CTC模式，也是默认模式，不做预处理，只在运算时重复标签将不再当成独立的标签来计算
preprocess_collapse_repeated=TRUE; ctc_merge_repeated=False;	忽略全部重复标签，只计算不重复的标签,因为预处理时已经把重复的标签去掉了
preprocess_collapse_repeated=False; ctc_merge_repeated=False;	所有重复标签都会参加计算

对于ctc_loss的返回值，仍然属于loss的计算模式，当取批次样本进行训练时，同样也需要对最终的ctc_loss求均值。



注意： 对于重复标签方面的ctc_loss计算，一般情况下默认即可。

另外这里有个隐含的规则，inputs中的classes是指需要输出多少类，在使用ctc_loss时，要将classes+1，即再多生成一个类，用于存放blank。因为输入的序列与label并不是一一对应的，所以需要通过添加blank类，当对应不上时，最后的softmax就会将其生成到blank。具体做法就是在最后的输出层多构建一个节点即可。

这个规则是ctc_loss内置的，否则当标准标签label中的类索引等于inputs中的size-1时会报错。

2. SparseTensor类型

前面提到了SparseTensor类型，这里主要介绍一下，本来应该将其放在前面章节介绍的，考虑到读者的接受程度，所以就放在这里介绍了。

首先介绍下稀疏矩阵，它是相对于密集矩阵而言的。

密集矩阵就是我们常见的矩阵。当密集矩阵中大部分的数都为0时，就可以使用一种更好的存储方式（只将矩阵中不为0的索引和值记录下来）来存储。这种方式就可以大大节省内存空间，它就是“稀疏矩阵”。

稀疏矩阵在TensorFlow中的结构类型如下：

```
SparseTensor(indices, values, dense_shape)
```

一个密集矩阵只需要3个参数即可，说明如下。

- indices：就是前面所说的不为0的位置信息。它是一个二维的int64 Tensor，shape为(N, ndims)，指定了sparse tensor中的索引，例如，`indices=[[1, 3], [2, 4]]`，表示dense tensor中对应索引为[1, 3], [2, 4]位置的元素的值不为0。

- values：一个list，存储密集矩阵中不为0位置所对应的值，它要与indices里的顺序对应。例

如，`indices=[[1, 3], [2, 4]]`, `values=[18, 3.6]`，表明[1, 3]的位置是18, [2, 4]的位置是3.6。

· `dense_shape`: 一个1D的int64 tensor, 代表原来密集矩阵的形状。

3. 生成SparseTensor

了解了SparseTensor类型之后，就可以按照参数来拼接出一个SparseTensor了。在实际应用中，常会用到需要将稠密矩阵dense转成稀疏矩阵SparseTensor。示例代码如下：

```
def sparse_from_dense(dense, dtype=np.int32):
    indices = []
    values = []

    for n, seq in enumerate(dense):
        indices.extend(zip([n] * len(seq), range(len(seq))))
        values.extend(seq)

    indices = np.asarray(indices, dtype=np.int64)
    values = np.asarray(values, dtype=dtype)
    shape = np.asarray([len(dense), indices.max(0)[1] + 1],
                      dtype=np.int64)

    return tf.SparseTensor(indices=indices, values=values, shape=shape)
```



注意：由于TensorFlow中没有现成的函数，可以自己封装好后保存下来，以后需要时随时拿来用。

4. SparseTensor转dense

在TensorFlow中，可以很方便地实现SparseTensor转dense：

```
tf.sparse_tensor_to_dense(sp_input, default_value=0, validate_
name=None)
```

参数说明如下。

- sp_input：一个SparseTensor。
- default_value：没有指定索引的对应的默认值，默认为0。
- validate_indices：布尔值。如果为True，该函数会检查sp_input的indices的lexicographic order是否有重复。
- name：返回tensor的名字前缀，可选。

5. levenshtein距离

前面讲到了ctc_loss是用来训练时间序列分类模型的。评估模型时，一般常使用计算得到的levenshtein距离值作为模型的评分（正确率或错误率）。

levenshtein距离又叫编辑距离（Edit

*****ebook converter DEMO Watermarks*****

Distance），是指两个字符串之间，由一个转成另一个所需的最少编辑操作次数。许可的编辑操作包括：将一个字符替换成另一个字符、插入一个字符、删除一个字符。一般来说，编辑距离越小，两个字符串的相似度越大。

这种方法应用非常广泛，在全序列对比、局部序列对比中都会用到，例如语音识别、拼写纠错、DAN比对等方面。

在TensorFlow中，levenshtein距离的处理被封装成对两个稀疏矩阵进行的操作，具体定义如下：

```
def edit_distance(hypothesis, truth, normalize=True, name="edit
```

参数说明如下。

- hypothesis: SparseTensor类型，输入预测的序列结果。
- truth: SparseTensor类型，输入真实的序列结果。
- normalize: 默认为True，求出来的Levenshtein距离除以真实序列的长度。
- name: operation的名字，可选。

- 返回值：R-1维的DenseTensor，包含着每个Sequence的Levenshtein距离。

9.4.18 CTCdecoder

CTC结构中还有一个重要的环节就是CTCdecoder，下面就来介绍一下。

1. CTCdecoer介绍

虽然在输入ctc_loss中的logits (inputs) 是我们的预测结果，但却是带有空标签 (blank) 的，而且是一个与时间序列强对应的输出。在实际情况下，我们需要一个转化好的类似于原始标准标签 (labels) 的输出。这时可以使用CTCdecoder，经过它对预测结果加工后，就可以与标准标签 (labels) 进行损失值 (loss) 的运算了。

2. CTCdecoder函数

在TensorFlow中，CTCdecoder有两个函数，如表9-4所示。

表9-4 CTCdecoder函数

函 数	说 明
tf.nn.ctc_greedy_decoder(inputs, sequence_length, merge_repeated=True)	<p>使用greedy策略的CTC解码:</p> <ul style="list-style-type: none"> • inputs: 模型的输出预测值logits, shape为(max_time × batch_size × num_classes) • sequence_length: 序列的长度。该sequence_length 和用在dynamic_rnn中的sequence_length是一致的 <p>返回值: tuple (decoded, log_probabilities)</p> <ul style="list-style-type: none"> • decoded: 是一个list。只有一个元素, 是一个SparseTensor, 保存着解码的结果 • log_probabilities: 一个浮点型矩阵(batch_size×1)包含着序列的log 概率
tf.nn.ctc_beam_search_decoder(inputs, sequence_length, beam_width=100, top_paths=1, merge_repeated=True)	另一种寻路策略, 参数同上

 **注意:** 在实际情况中, 解码完事的decoder是list, 不能直接用, 通常取decoder[0], 然后转成密集矩阵, 得到的是一个批次的结果, 然后再一条一条地取到每一个样本的结果。

9.5 实例68：利用BiRNN实现语音识别

在神经网络大势兴起之前，语音识别还是有一定门槛的。传统的语音识别方法，是基于语音学（Phonetics）的方法，它们通常包含拼写、声学和语言模型等单独组件。开发人员需要了解编程以外的很多语言学知识，语言学也会作为一门单独的专业学科存在。训练模型的语料中除了要标注具体的文字，还要标注按照时间对应的音素，需要大量的人工成本。

本节将通过一个例子来演示BiRNN在语音识别中的应用。

实例描述

准备一批带有文字标注的语音样本，构建BiRNN网络，通过该语料样本进行训练，最终实现一个能够识别语音的神经网络模型。

9.5.1 语音识别背景

使用神经网络技术可以将语音识别变得简单。通过能进行时序分类的连接时间分类（Connectionist Temporal Classification, CTC）目标函数，计算多个标签序列的概率，而序列是语音样本中所有可能的对应文字的集合。随后把预*****ebook converter DEMO Watermarks*****

测结果与实际进行比较，计算预测结果的误差，以在训练中不断更新网络权重。这样可以丢弃音素的概念，自然也不需要人工根据时序标注对应的音素了。由于是直接拿音频序列来对应文字，连语言模型都可以省去，这样就脱离了标准的语言模型与声学模型，将使语音识别技术与语言无关（也就是中文、英文、地方语言），只要样本足够多，就可以训练出来。

例子中使用了两个代码文件“9-24 yuyinutils.py”与“9-23 yuyinchall.py”。

- 代码文件 “9-24 yuyinutils.py”：放置语音识别相关的工具函数。
- 代码文件 “9-23 yuyinchall.py”：放置语音识别主体流程函数。

9.5.2 获取并整理样本

1. 样本下载

本例中使用了清华大学公开的语料库样本，下载地址如下：

- <http://data.csdl.org/thchs30/zip/wav.tgz>；
- <http://data.csdl.org/thchs30/zip/doc.tgz>。

第一个是音频WAV文件的压缩包。第二个是WAV文件中对应的文字。thchs30语料库本来有3部分，这里只列出了两部分，还有一部分是语言模型，暂时用不上，所以忽略。

省去了语言模型的语料库看起来简单多了，感兴趣的读者完全可以仿照thchs30语料库，自己录制音频，创建自己的语料库。这样你就可以学出一个识别自己口音的语音识别模型了。



注意：自己录制时，一定要将音频录制成单声道的，或者将双声道的音频转成单声道也可以。

文件下载好之后，解压并放到指定目录中即可，后面可以在代码中通过该目录进行读取。

2. 样本读取

下面通过代码将数据读入内存。指定训练语音的文件夹与对应的文档，调用get_wavs_labels函数即可。

代码9-23 yuyinchall

```
01 .....
02 yuyinutils = __import__("9-24_yuyinutils")
03 get_wavs_labels = yuyinutils.get_wavs_labels
04
```

```
05 wav_path='D:/ data_thchs30/data_thchs30/train'  
06 label_file='D: /data_thchs30/doc/trans/train.word.txt'  
07  
08 wav_files, labels = get_wavs_labels(wav_path, label_file)  
09 print(wav_files[0], labels[0])  
10 print("wav:", len(wav_files), "label", len(labels))  
11 .....
```

输出信息如下：

```
D:/ data_thchs30/data_thchs30/train/A11_0.WAV 绿 是 阳春 烟 霾  
wav: 8911 label 8911
```

可见，`wav_files`里面是一个个音频文件名称，其对应的文字都存放在`labels`数组里，一共是8911个文件。这里用到的`get_wavs_labels`函数是自己定义的函数，为了代码规整些，我们把它放到另一个py文件（代码“9-24 yuyinutils.py”）里。`get_wavs_labels`的定义如下。

代码9-24 yuyinutils

```
01 .....
```

```
02 import os  
03  
04 '''读取WAV文件对应的label'''  
05 def get_wavs_labels(wav_path=wav_path, label_file=label_file)  
06 #获得训练用的WAV文件路径列表  
07     wav_files = []  
08     for (dirpath, dirnames, filenames) in os.walk(wav_path)  
09         for filename in filenames:  
10             if filename.endswith('.wav') or filename.endswith('.WAV'):...  
11                 filename_path = os.sep.join([dirpath, filename])  
12                 if os.stat(filename_path).st_size < 24000:  
13                     continue
```

*****ebook converter DEMO Watermarks*****

```
14         wav_files.append(filename_path)
15
16     labels_dict = {}
17     with open(label_file, 'rb') as f:
18         for label in f:
19             label = label.strip(b'\n')
20             label_id = label.split(b' ', 1)[0]
21             label_text = label.split(b' ', 1)[1]
22             labels_dict[label_id.decode('ascii')] = label_text.decode('utf-8')
23
24     labels = []
25     new_wav_files = []
26     for wav_file in wav_files:
27         wav_id = os.path.basename(wav_file).split('.')[0]
28
29         if wav_id in labels_dict:
30             labels.append(labels_dict[wav_id])
31             new_wav_files.append(wav_file)
32
33     return new_wav_files, labels
```

首先是通过WAV文件路径读入文件。然后再将文本文件内容按照WAV文件名进行裁分放到labels里，最终将WAV与labels的对应顺序关联起来。

 **注意：** 在读取文本时使用的是UTF-8编码，如果在Windows下自建数据集，需要改成GB2312编码。

3. 建立批次获取样本函数

在代码“9-23 yuyinchall.py”文件中，读取完WAV文件和labels之后，添加如下代码，对labels的字数进行统计。接着定义一个next_batch函数，

*****ebook converter DEMO Watermarks*****

该函数的作用就是取一批次的样本数据进行训练。

代码9-23 yuyinchall (续)

```
12 from collections import Counter
13 ## 自定义
14 from yuyinutils import sparse_tuple_to_texts_ch,ndarray_
15 from yuyinutils import get_audio_and_transcriptch, pad_se
16 from yuyinutils import sparse_tuple_from
17 .....
18 # 字表
19 all_words = []
20 for label in labels:
21     all_words += [word for word in label]
22 counter = Counter(all_words)
23 words = sorted(counter)
24 words_size= len(words)
25 word_num_map = dict(zip(words, range(words_size)))
26
27 print('字表大小:', words_size)
28
29 n_input = 26          #计算美尔倒谱系数的个数
30 n_context = 9         #对于每个时间点，要包含上下文样本
31 sparse_tuple_to_texts_ch = yuyinutils.sparse_tuple_to_texts_
32 ndarray_to_text_ch      = yuyinutils.ndarray_to_text_ch
33 get_audio_and_transcriptch = yuyinutils.get_audio_and_transc
34 pad_sequences           = yuyinutils.pad_sequences
35 sparse_tuple_from        = yuyinutils.sparse_tuple_from
36 batch_size =8
37 def next_batch(labels, start_idx = 0,batch_size=1,wav_fi
38 files):
39     filesize = len(labels)
40     end_idx = min(filesize, start_idx + batch_size)
41     idx_list = range(start_idx, end_idx)
42     txt_labels = [labels[i] for i in idx_list]
43     wav_files = [wav_files[i] for i in idx_list]
44     (source, audio_len, target, transcript_len) = get_auc
45     transcriptch(None,
46                  wav_files,
47                  n_:
48                  n_c
49
50     txt_labels)
51
52
```

```
43     start_idx += batch_size
44     # 验证 start_idx
45     if start_idx >= filesize:
46         start_idx = -1
47
48     # 使用pad方式对其输入序列
49     source, source_lengths = pad_sequences(source) #如果有
50     sparse_labels = sparse_tuple_from(target)
51
52     return start_idx, source, source_lengths, sparse_label
```

将音频数据转成训练数据是在next_batch中的get_audio_and_transcriptch函数里完成的，然后使用pad_sequences函数将该批次的音频数据对齐。对于文本，使用sparse_tuple_from函数将其转成稀疏矩阵，这3个函数都放在代码“9-24yuyinutils.py”文件里面。

添加测试代码，取出批次数据并打印出来。

代码9-23 yuyinchall（续）

```
53 next_idx,source,source_len,sparse_lab = next_batch(label_s
size)
54 print(len(sparse_lab))
55 print(np.shape(source))
56 t = sparse_tuple_to_texts_ch(sparse_lab,words)
57 print(t[0])
58 #source为具体的样本，每条样本的内容为19个时间序列，包括：前9（不够
后9。每个时间序列有26个美尔倒谱系数。第一条的样本是从第10个时间序列开始
```

运行代码，结果如下：

词汇表大小： 2666

3

*****ebook converter DEMO Watermarks*****

(8, 1168, 494)

绿是阳春烟景大块文章的底色四月的林峦更是绿得鲜活秀媚诗意盎然

整个样本集里涉及的字数有2666个，
sparse_lab为文字转化成向量后并生成的稀疏矩阵，所以长度为3，补0对齐后的音频数据的shape为(8, 1168, 494)，8代表batchsize；1168代表时序的总个数。494是组合好的MFCC特征数：取前9个时序的MFCC，当前MFCC再加上后9个MFCC，每个MFCC由26个数字组成。最后一个输出是通过sparse_tuple_to_texts_ch函数将稀疏矩阵向量sparse_lab中的第一个内容还原成文字。函数sparse_tuple_to_texts_ch的定义同样在代码“9-24yuyinutils.py”文件里。

4. 安装python_speech_features工具

为了让机器识别音频数据，必须先将数据从时域转换为频域，需要将语音数据转换为需要计算的13位或26位不同倒谱特征的梅尔倒频谱系数(MFCC)。这一过程可以借助工具python_speech_features的代码包来实现，现在一起来安装该代码包。

在计算机联网的状态下，打开“开始”菜单，在“运行”框里输入cmd，调出控制台窗口，输入如下命令：

```
pip install python_speech_features
```

python_speech_features工具就会自动安装了。

5. 提取音频数据MFCC特征

对于WAV音频的样本，通过MFCC转换之后，在函数get_audio_and_transcriptch中将数据存储为时间（列）和频率特征系数（行）的矩阵，其代码如下。

代码9-24 yuyinutils（续）

```
34 import numpy as np
35
36 from python_speech_features import mfcc    #需要使用pip ins
37 import scipy.io.wavfile as wav
38 .....
39 def get_audio_and_transcriptch(txt_files, wav_files, n_ir
context,word_num_map,txt_labels=None):
40
41     audio = []
42     audio_len = []
43     transcript = []
44     transcript_len = []
45     if txt_files!=None:
46         txt_labels = txt_files
47
48     for txt_obj, wav_file in zip(txt_labels, wav_files):
49         # 载入音频数据并转化为特征值
50         audio_data = audiofile_to_input_vector(wav_file,
context)
51         audio_data = audio_data.astype('float32')
52
53         audio.append(audio_data)
54         audio_len.append(np.int32(len(audio_data)))
55
```

*****ebook converter DEMO Watermarks*****

```
56     # 载入音频对应的文本
57     target = []
58     if txt_files!=None:#txt_obj是文件
59         target = get_ch_lable_v(txt_obj,word_num_map)
60     else:
61         target = get_ch_lable_v(None,word_num_map,txt1
62 transcript.append(target)
63 transcript_len.append(len(target))
64
65     audio = np.asarray(audio)
66     audio_len = np.asarray(audio_len)
67     transcript = np.asarray(transcript)
68     transcript_len = np.asarray(transcript_len)
69     return audio, audio_len, transcript, transcript_len
```

这部分代码遍历所有音频文件及文本，将音频调用audiofile_to_input_vector转成MFCC，文本调用get_ch_lable_v函数将文本转成向量。所以接着看audiofile_to_input_vector的实现。

在audiofile_to_input_vector中先将其转化为MFCC特征码，例如第一个文件会被转成（277，26）数组，代表着277个时间序列，每个序列的特征值是26个。



注意： 这里有个小技巧，因为使用了双向循环神经网络，它的输出包含正、反向的结果，相当于每一个时间序列都扩大了一倍，所以为了保证总时序不变，使用orig_inputs = orig_inputs[:, : 2]对orig_inputs每隔一行进行一次取样。这样被忽略的那个序列可以用后文中反向RNN生成的输出来代替，维持了总的序列长度。

接着会扩展这26个特征值，将其扩展成：前9个时间序列MFCC+当前MFCC+后9个时间序列。比如第2个序列的前面只有一个序列不够9个，这时就要为其补0，将它凑够9个。同理对于取不到前9、后9时序的序列都做补0操作。这样数据就被扩成了（139， 494）。最后再将其进行标准化（减去均值然后再除以方差）处理，这是为了在训练中效果更好。代码如下。

代码9-24 yuyinutils（续）

```
70 .....
71 def audiofile_to_input_vector(audio_filename, numcep, numcontext):
72     # 加载wav文件
73     fs, audio = wav.read(audio_filename)
74
75     # 获得mfcc coefficients
76     orig_inputs = mfcc(audio, samplerate=fs, numcep=numcep)
77     orig_inputs = orig_inputs[:, ::2] # (139, 26)
78
79     train_inputs = np.array([], np.float32)
80     train_inputs.resize((orig_inputs.shape[0], numcep + 2 * numcontext))
81
82     empty_mfcc = np.array([])
83     empty_mfcc.resize((numcep))
84
85     # 准备输入数据。输入数据的格式由三部分安装顺序拼接而成，分为当
86     # 前9个序列样本，当前样本序列、后9个序列样本
87     time_slices = range(train_inputs.shape[0]) # 139个切片
88     context_past_min = time_slices[0] + numcontext
89     context_future_max = time_slices[-1] - numcontext # [9, 137, 129]
90     for time_slice in time_slices:
91         # 前9个补0, mfcc features
92         need_empty_past = max(0, (context_past_min - time_slice))
93         empty_source_past = list(empty_mfcc for empty_slice in
94         (need_empty_past)))
95         data_source_past = orig_inputs[max(0, time_slice - numcontext):time_slice + numcontext]
```

*****ebook converter DEMO Watermarks*****

```

        time_slice]
95      assert(len(empty_source_past) + len(data_source_) == numcontext)
96
97      # 后9个补0, mfcc features
98      need_empty_future = max(0, (time_slice - context_ - 9) * numcep)
99      empty_source_future = list(empty_mfcc for empty_mfcc in range(need_empty_future))
100     data_source_future = orig_inputs[time_slice + 1: time_slice + numcontext + 1]
101     assert(len(empty_source_future) + len(data_source_) == numcontext)
102
103     if need_empty_past:
104         past = np.concatenate((empty_source_past, data_source_[0:9]))
105     else:
106         past = data_source_past
107
108     if need_empty_future:
109         future = np.concatenate((data_source_future, empty_source_future))
110     else:
111         future = data_source_future
112
113     past = np.reshape(past, numcontext * numcep)
114     now = orig_inputs[time_slice]
115     future = np.reshape(future, numcontext * numcep)
116
117     train_inputs[time_slice] = np.concatenate((past, now, future))
118     assert(len(train_inputs[time_slice])) == numcep + numcontext
119
120     # 将数据使用正太分布标准化, 减去均值然后再除以方差
121     train_inputs = (train_inputs - np.mean(train_inputs)) / np.std(train_inputs)
122
123     return train_inputs

```

orig_inputs代表转化后的MFCC， train_inputs是将时间序列扩充后的数据，里面的for循环是做补0操作。最后两行是数据标准化。

6. 批次音频数据对齐

*****ebook converter DEMO Watermarks*****

前面是对单个文件里的特征补0，在训练环节中，文件是一批一批的获取并进行训练的，这要求每一批音频的时序数要统一，所以这里需要有一个对齐处理，`pad_sequences`的定义如下，可以支持补0和截断两个操作。对于补0和截断的方向都可以通过参数来控制，'post'代表后补0（截断），'pre'代表前补0（截断）。

代码9-24 yuyinutils（续）

```
123 .....
124 def pad_sequences(sequences, maxlen=None, dtype=np.float32,
125                  padding='post', truncating='post', value=0):
126     lengths = np.asarray([len(s) for s in sequences], dtype)
127     nb_samples = len(sequences)
128     if maxlen is None:
129         maxlen = np.max(lengths)
130
131     # 从第一个非空的序列中得到样本形状
132     sample_shape = tuple()
133     for s in sequences:
134         if len(s) > 0:
135             sample_shape = np.asarray(s).shape[1:]
136             break
137
138     x = (np.ones((nb_samples, maxlen)) + sample_shape) * value
139     x = x.astype(dtype)
140
141     for idx, s in enumerate(sequences):
142         if len(s) == 0:
143             continue    # 如果序列为空，则跳过
144         if truncating == 'pre':
145             trunc = s[-maxlen:]
146         elif truncating == 'post':
147             trunc = s[:maxlen]
148         else:
149             raise ValueError('Truncating type "%s" not '
150                             % truncating)
151
152     # 检查trunc
```

*****ebook converter DEMO Watermarks*****

```
152     trunc = np.asarray(trunc, dtype=dtype)
153     if trunc.shape[1:] != sample_shape:
154         raise ValueError('Shape of sample %s of seq\
155                         %s is different from expected shape %s' %
156                         (trunc.shape[1:], idx, samp
157     if padding == 'post':
158         x[idx, :len(trunc)] = trunc
159     elif padding == 'pre':
160         x[idx, -len(trunc):] = trunc
161     else:
162         raise ValueError('Padding type "%s" not unde\
padding)
163 return x, lengths
```

7. 文字样本的转化

对于文本方面的样本，需要将里面的文字转换成具体的向量。get_ch_label_v会按照传入的word_num_map将txt_label或是指定文件中的文字转化成向量。后面的get_ch_label是读取文件操作，本例中用不到。

代码9-24 yuyinutils（续）

```
164 .....
165 def get_ch_label_v(txt_file, word_num_map, txt_label=None):
166
167     words_size = len(word_num_map)
168
169     to_num = lambda word: word_num_map.get(word, words_s
170
171     if txt_file!= None:
172         txt_label = get_ch_label(txt_file)
173
174     labels_vector = list(map(to_num, txt_label))
175     return labels_vector
176
177 def get_ch_label(txt_file):
```

*****ebook converter DEMO Watermarks*****

```
178     labels= " "
179     with open(txt_file, 'rb') as f:
180         for label in f:
181             #labels =label.decode('utf-8')
182             labels = labels +label.decode('gb2312')
183
184     return  labels
```

8. 密集矩阵转成稀疏矩阵

TensorFlow中没有密集矩阵转稀疏矩阵函数，所以需要编写一个。该函数比较常用，可以当成工具来储备，具体代码如下。

代码9-24 yuyinutils（续）

```
185 .....
186 def sparse_tuple_from(sequences, dtype=np.int32):
187
188     indices = []
189     values = []
190
191     for n, seq in enumerate(sequences):
192         indices.extend(zip([n] * len(seq), range(len(seq))))
193         values.extend(seq)
194
195     indices = np.asarray(indices, dtype=np.int64)
196     values = np.asarray(values, dtype=dtype)
197     shape = np.asarray([len(sequences), indices.max(0)], dtype=np.int64)
198
199     return indices, values, shape
```

这里主要是算出indices、values、shape这3个值，得到之后可以使用tf.SparseTensor随时生成稀疏矩阵。

9. 将字向量转成文字

字向量转成文字主要有两个函数：

sparse_tuple_to_texts_ch函数，将稀疏矩阵的字向量转成文字； ndarray_to_text_ch函数，将密集矩阵的字向量转成文字。两个函数都需要传入字表，然后会按照字表对应的索引将字转化回来。

代码9-24 yuyinutils（续）

```
200 .....
201 # 常量
202 SPACE_TOKEN = '<space>'           # space符号
203 SPACE_INDEX = 0                     # 0 为space索引
204 FIRST_INDEX = ord('a') - 1
205
206 def sparse_tuple_to_texts_ch(tuple, words):
207     indices = tuple[0]
208     values = tuple[1]
209     results = [''] * tuple[2][0]
210     for i in range(len(indices)):
211         index = indices[i][0]
212         c = values[i]
213
214         c = ' ' if c == SPACE_INDEX else words[c]
215         results[index] = results[index] + c
216     # 返回strings的List
217     return results
218
219 def ndarray_to_text_ch(value, words):
220     results = ''
221     for i in range(len(value)):
222         results += words[value[i]]
223     return results.replace('`', ' ')
```

9.5.3 训练模型

样本准备好后，就开始模型的搭建了。

1. 定义占位符

定义3个占位符，具体说明如下。

- input_tensor：为输入的音频数据[none, none, Mfcc_features]，第一个是batch_size用none来表示；第二个是时序数也用none来表示，因为每一批次的时序都是不同的；第三个是MFCC的特征，是取当前特征n_input和前后n_context个特征的组合，即 $2 \times n_{context} + 1$ 个序列，每个序列特征数为n_input，于是得出 $n_{input} + (2 \times n_{input} * n_{context})$ 。
- targets：音频数据所对应的文本，是一个稀疏矩阵的占位符。
- seq_length：当前batch数据的序列长度。
- keep_dropout：dropout的参数。

代码9-23 yuyinchall（续）

```
59 ....
60 input_tensor = tf.placeholder(tf.float32, [None, None, r
   * n_input * n_context]), name='input') #语音MFCC features
61 # ctc_loss计算时需要使用sparse_placeholder来生成 SparseTens
62 targets = tf.sparse_placeholder(tf.int32, name='targets')
63 # 1d array of size [batch_size]
64 seq_length = tf.placeholder(tf.int32, [None], name='seq_
65 keep_dropout= tf.placeholder(tf.float32)
```

2. 构建网络模型

网络模型使用了双向RNN的结构，并将其封装在BiRNN_model函数里。调用的代码如下。

代码9-23 yuyinchall (续)

```
66 logits = BiRNN_model( input_tensor, tf.to_int64(seq_length_context, words_size +1, keep_dropout)
```

BiRNN_model的定义如下。

使用3个1024节点的全连接层，然后是一个双向RNN，最后接上2个全连接层，并且都带有dropout层。这里使用的激活函数是带截断的Relu，截断值设为20。学习参数的初始化使用标准差为0.046875的random_normal。keep_dropout_rate为0.95。

代码9-23 yuyinchall (续)

```
67 .....
```

```
68 b_stddev = 0.046875
```

```
69 h_stddev = 0.046875
```

```
70
```

```
71 n_hidden = 1024
```

```
72 n_hidden_1 = 1024
```

```
73 n_hidden_2 = 1024
```

```
74 n_hidden_5 = 1024
```

```
75 n_cell_dim = 1024
```

```
76 n_hidden_3 = 2 * 1024
```

```

77
78 keep_dropout_rate=0.95
79 relu_clip = 20
80
81 def BiRNN_model( batch_x, seq_length, n_input, n_context,
82   keep_dropout):
83 # batch_x_shape: [batch_size, n_steps, n_input + 2*n_input]
84   batch_x_shape = tf.shape(batch_x)
85
86   # 将输入转成时间序列优先
87   batch_x = tf.transpose(batch_x, [1, 0, 2])
88   # 再转成2维传入第一层
89   batch_x = tf.reshape(batch_x,
90     [-1, n_input + 2 * n_input * n_context])
91
92   # 使用clipped RELU activation and dropout.
93   # 第一层
94   with tf.name_scope('fc1'):
95     b1 = variable_on_cpu('b1', [n_hidden_1], tf.random_
96     initializer(stddev=b_stddev))
97     h1 = variable_on_cpu('h1', [n_input + 2 * n_input +
98       n_hidden_1],
99       tf.random_normal_initializer(stddev=h_stddev))
100    layer_1 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(b1,
101      layer_1), relu_clip))
102    layer_1 = tf.nn.dropout(layer_1, keep_dropout)
103
104   # 第二层
105   with tf.name_scope('fc2'):
106     b2 = variable_on_cpu('b2', [n_hidden_2], tf.random_
107     initializer(stddev=b_stddev))
108     h2 = variable_on_cpu('h2', [n_hidden_1, n_hidden_2],
109       tf.random_normal_initializer(stddev=h_stddev))
110    layer_2 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(b2,
111      layer_2), relu_clip))
112    layer_2 = tf.nn.dropout(layer_2, keep_dropout)
113
114   # 第三层
115   with tf.name_scope('fc3'):
116     b3 = variable_on_cpu('b3', [n_hidden_3], tf.random_
117     initializer(stddev=b_stddev))
118     h3 = variable_on_cpu('h3', [n_hidden_2, n_hidden_3],
119       tf.random_normal_initializer(stddev=h_stddev))
120    layer_3 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(b3,
121      layer_3), relu_clip))
122    layer_3 = tf.nn.dropout(layer_3, keep_dropout)
123
124   # 双向RNN
125   with tf.name_scope('lstm'):
```

*****ebook converter DEMO Watermarks*****

```

116      # 前向 cell:
117      lstm_fw_cell = tf.contrib.rnn.BasicLSTMCell(n_cell_dim,
118          bias=1.0, state_is_tuple=True)
119      lstm_fw_cell = tf.contrib.rnn.DropoutWrapper(lstm_fw_cell,
120          input_keep_prob=keep_prob)
121      # 反向 cell:
122      lstm_bw_cell = tf.contrib.rnn.BasicLSTMCell(n_cell_dim,
123          bias=1.0, state_is_tuple=True)
124      lstm_bw_cell = tf.contrib.rnn.DropoutWrapper(lstm_bw_cell,
125          input_keep_prob=keep_prob)
126
127      # 'layer_3' [n_steps, batch_size, 2*n_cell_dim]
128      layer_3 = tf.reshape(layer_3, [-1, batch_x_shape[0]])
129
130      outputs, output_states = tf.nn.bidirectional_dynamic_rnn(
131          cell_fw=lstm_fw_cell,
132          cell_bw=lstm_bw_cell,
133          inputs=batch_x,
134          sequence_length=sequence_length,
135          time_major=False)
136
137      # 连接正、反向结果[n_steps, batch_size, 2*n_cell_dim]
138      outputs = tf.concat(outputs, 2)
139
140      # 转化形状[n_steps*batch_size, 2*n_cell_dim]
141      outputs = tf.reshape(outputs, [-1, 2 * n_cell_dim])
142
143      with tf.name_scope('fc5'):
144          b5 = variable_on_cpu('b5', [n_hidden_5], tf.random_normal_initializer(stddev=b_stddev))
145          h5 = variable_on_cpu('h5', [(2 * n_cell_dim), n_hidden_5], tf.random_normal_initializer(stddev=h_stddev))
146          layer_5 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(outputs, h5), b5)), relu_clip)
147          layer_5 = tf.nn.dropout(layer_5, keep_dropout)
148
149      with tf.name_scope('fc6'):
150          # 全连接层用于softmax分类
151          b6 = variable_on_cpu('b6', [n_character], tf.random_normal_initializer(stddev=b_stddev))
152          h6 = variable_on_cpu('h6', [n_hidden_5, n_character], tf.random_normal_initializer(stddev=h_stddev))
153          layer_6 = tf.add(tf.matmul(layer_5, h6), b6)
154
155
156      # 将二维[n_steps*batch_size, n_character]转成三维 time-steps, batch_size, n_character].
157      layer_6 = tf.reshape(layer_6, [-1, batch_x_shape[0], n_character])

```

*****ebook converter DEMO Watermarks*****

```
155 # Output shape: [n_steps, batch_size, n_character]
156 return layer_6
157
158 """
159 使用 CPU memory.
160 """
161 def variable_on_cpu(name, shape, initializer):
162     # 使用/cpu:0 device
163     with tf.device('/cpu:0'):
164         var = tf.get_variable(name=name, shape=shape, in:
165                               initializer)
166     return var
```

这里的shape变化比较复杂，需要先将输入变为二维的Tensor，才可以传入全连接层。全连接层进入BIRNN时也需要形状转换成三维的Tensor，BIRNN输出的结果是 $2 \times n_hidden$ ，所以后面的全连接层输入是 $2 \times n_hidden$ ，最终输出时还要再转回三维的Tensor。

与图片分类不同的是，RNN输出的outputs没有取outputs[-1]，而是全部进入了后面的全连接层。语音识别是对输入的每个时序对应的结果进行转换，所以要将RNN的全部结果送入后面的全连接层；而RNN中的图片识别只是把行当成时序，只需要知道最后一行输入后的结果，所以只取了最后一个时序的输出。



注意： 这里使用了一个小技巧。通过函数variable_on_cpu来声明学习参数变量，将所有的学习参数定义在CPU的内存中，可以让GPU的内

存充分地用于运算。

3. 定义损失函数即优化器

语音识别是属于非常典型的时间序列分类问题，前面讲过，对于这样的问题要使用ctc_loss的方法来计算损失值。优化器还是使用AdamOptimizer，学习率为0.001。代码如下。

代码9-23 yuyinchall（续）

```
167 .....  
168 #调用ctc_loss  
169 avg_loss = tf.reduce_mean(ctc_ops.ctc_loss(targets, lengths))  
170 #####  
171 #####  
172 #优化器  
173  
174 learning_rate = 0.001  
175 optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)  
176 minimize(avg_loss)
```

4. 定义解码并评估模型节点

使用ctc_beam_search_decoder 函数以CTC的方式对预测结果logits进行解码，生成了decoded。前面说过，decoded是一个只有一个元素的数组，所以将其decoded[0]传入edit_distance函数，计算与正确标签targets之间的levenshtein距离。下列代码第182行中的targets与decoded[0]都是稀疏矩阵张量（SparseTensor）类型。对得到的

distance取reduce_mean，可以得出该模型对于当前batch的平均错误率。

代码9-23 yuyinchall（续）

```
176     .....
177     with tf.name_scope("decode"):
178         decoded, log_prob = ctc_ops.ctc_beam_search_decoder(
179             length, merge_repeated=False)
180     with tf.name_scope("accuracy"):
181         distance = tf.edit_distance(tf.cast(decoded[0], tf.
182             targets))
183         # 计算label error rate (accuracy)
184         ler = tf.reduce_mean(distance, name='label_error_rat
```

5. 建立session并添加检查点处理

到此模型已经建立好了，剩下的就是训练部分的搭建了。由于样本比较大，运算时间比较长，所以很有必要为模型添加检查点功能。如下代码在session建立之前，定义一个类（名为saver），用于保存检查点的相关操作，并指定检查点文件夹为当前路径下的log\yuyinchalltest\，然后启动session，进行初始，同时在指定路径下查找最后一次检查点。如果有文件就载入到模型，同时更新迭代次数epoch。

代码9-23 yuyinchall（续）

```
184     .....
185     epochs = 100
```

*****ebook converter DEMO Watermarks*****

```
186 savedir = "log/yuyinchalltest/"  
187 saver = tf.train.Saver(max_to_keep=1) # 生成模型  
188 # 创建session  
189 sess = tf.Session()  
190 # 没有模型，就重新初始化  
191 sess.run(tf.global_variables_initializer())  
192  
193 kpt = tf.train.latest_checkpoint(savedir)  
194 print("kpt:", kpt)  
195 startepo = 0  
196 if kpt!=None:  
197     saver.restore(sess, kpt)  
198     ind = kpt.find("-")  
199     startepo = int(kpt[ind+1:])  
200     print(startepo)
```

6. 通过循环来迭代训练模型

记录下开始时间，启用循环，进行迭代训练，每次循环通过next_batch函数取一批次样本数据，并设置keep_dropout参数，通过sess.run来运行模型的优化器，同时输出loss的值。总样本迭代100次，每次迭代中，一批次取8条数据。

代码9-23 yuyinchall（续）

```
201 .....  
202 # 准备运行训练步骤  
203 section = '\n{0:=^40}\n'  
204 print(section.format('Run training epoch'))  
205  
206 train_start = time.time()  
207 for epoch in range(epochs): #样本集  
208     epoch_start = time.time()  
209     if epoch<startepo:  
210         continue  
211  
212     print("epoch start:", epoch, "total epochs= ", epochs)  
213 #####运行batch####
```

```
214     n_batches_per_epoch = int(np.ceil(len(labels) / batch_size))
215     print("total loop ",n_batches_per_epoch,"in one epoch")
216
217     train_cost = 0
218     train_ler = 0
219     next_idx = 0
220
221     for batch in range(n_batches_per_epoch): #每次取batch_size个数据
222         #取数据
223         next_idx,source,source_lengths,sparse_labels = \
224             next_batch(labels,next_idx,batch_size)
225         feed = {input_tensor: source, targets: sparse_labels,
226                 seq_length: source_lengths,keep_dropout:keep_dropout_rate}
227         #计算 avg_loss optimizer
228         batch_cost, _ = sess.run([avg_loss, optimizer],feed )
229         train_cost += batch_cost
```

7. 定期评估模型，输出模型解码结果

每取20次batch数据，就将过程信息打印出来，将样本数据送入模型进行语音识别，并输出预测结果。为防止打印信息过多，每次只打印一条信息，并将其文件名、原始的文本和解码文本打印出来。

代码9-23 yuyinchall（续）

```
230     .....
231     if (batch +1)%20 == 0:
232         print('loop:',batch, 'Train cost: ', train_cost)
233         feed2 = {input_tensor: source, targets: sparse_labels,
234                 seq_length: source_lengths,keep_dropout:1.0}
235         d,train_ler = sess.run([decoded[0],ler], feed2)
236         dense_decoded = tf.sparse_tensor_to_dense( (d
237             value=-1).eval(session=sess)
238         dense_labels = sparse_tuple_to_texts_ch(spar
```

```
words)
238
239     counter =0
240     print('Label err rate: ', trainлер)
241     for orig, decoded_arr in zip(dense_labels, (
242         # 转成strings
243         decoded_str = ndarray_to_text_ch(decoded_arr)
244         print(' file {}'.format(counter))
245         print('Original: {}'.format(orig))
246         print('Decoded:  {}'.format(decoded_str))
247         counter=counter+1
248         break
249
250     epoch_duration = time.time() - epoch_start
251
252     log = 'Epoch {}/{}, train_cost: {:.3f}, train_ler: {:.2f} sec'
253     print(log.format(epoch ,epochs, train_cost,train_ler,duration))
254     saver.save(sess, savedir+"yuyinch.cpkt", global_step
255
256 train_duration = time.time() - train_start
257 print('Training complete, total duration: {:.2f} min'.format(
duration / 60))
258
259 sess.close()
```

通过sess.run计算decoded[0]的值只是个SparseTensor类型，需要用tf.sparse_tensor_to_dense将其转成dense矩阵（记住Tensor Flow里的类型必须用eval或session.run才能得到真实值），然后再调用sparse_tuple_to_texts_ch将其转成文本dense_labels。

在每次迭代的最后加入检查点保存代码，以便中断可以恢复。

运行以上代码，经过一段时间之后（十几小时或几十个小时），会得到如下输出：

*****ebook converter DEMO Watermarks*****

```
.....  
file 0  
Original: 另外 加工 修理 和 修配 业务 不 属于 营业税 的 应 税 劳务  
Decoded: 另外 加工 理 和 修配 务 不 属于 营业税 的 应 税 劳务 不  
loop: 79 Train cost: 10.3595850527  
Label err rate: 0.0189385  
    file 0  
Original: 这 碗 离 娘 饭 姑娘 再有 离 娘 痛楚 也 要 每样 都 吃 一点  
Decoded: 这 碗 离 娘 饭 姑 有 离 娘 痛楚 也 要 每样 都 吃 一点 才  
loop: 99 Train cost: 10.3084330273  
Label err rate: 0.0270463  
    file 0  
Epoch 99/100, train_cost: 1176.815, train_ler: 0.047, time:  
WARNING:tensorflow:Error encountered when serializing LAYER_  
Type is unsupported, or the types of the items don't match 1  
'dict' object has no attribute 'name'  
Training complete, total duration: 1182.50 min
```

由此可见程序基本可以将样本库中的语音全部识别出来，错误率在0.02左右。最后打印的警告是出至TensorFlow中保存模型节点时的，不影响整体功能，可以不用管。

一般来讲，将训练好的模型作为识别后端，通过编写程序录音采集，将WAV文件传入进行解码，即可实现在线实时的语音识别了。

9.5.4 练习题

(1) 试着按照前面例子中的样本形式，自己录制一些语音样本，并做出对应的文本文件，用该代码训练出适应自己声音的语音模型。

(2) 实例68中的模型里，fc5层使用的是全

*****ebook converter DEMO Watermarks*****

连接的方法，还可以使用全局平均池化层的方式代替，读者可以试着改写一下代码，看看效果。

(3) 尝试在BIRNN部分加入更深的RNN层，来获得更好的识别率。

9.6 实例69：利用RNN训练语言模型

下面来做一个实验，用RNN预测语言模型，并让它输出一句话，具体业务描述如下。

先让RNN学习一段文字，之后模型可以根据我们的输入再自动预测后面的文字。同时将模型预测出来的文字当成输入，再放到模型里，模型就会预测出下一个文字，这样循环下去，可以看到RNN能够输出一句话。

那么RNN是怎么样来学习这段文字呢？这里将整段文字都看成一个个的序列。在模型里预设值只关注连续的4个序列，这样在整段文字中，每次随意拿出4个连续的文字放到模型里进行训练，然后把第5个连续的值当成标签，与输出的预测值进行loss的计算，形成一个可训练的模型，通过优化器来迭代训练。

实例描述

通过让RNN网络对一段文字的训练学习来生成模型，最终可以使用机器生成的模型来表达自己的意思。

下面看看具体实现过程。

9.6.1 准备样本

这个环节很简单，随便复制一段话放到txt里即可。在例子中使用的样本如下：

在尘世的纷扰中，只要心头悬挂着远方的灯光，我们就会坚持不懈地走，理想为我们灌注了精神的蕴藉。所以，生活再平凡、再普通、再琐碎，我们都要坚持一种信念，默守一种精神，为自己积淀站立的信心，前行的气力。

这是笔者随意下载的一段文字，把该段文字放到代码同级目录下，起名为wordstest.txt。

1. 定义基本工具函数

具体的基本工具函数与语音识别例子差不多，都是与文本处理相关的，首先引入头文件，然后定义相关函数，其中get_ch_lable函数从文件里获取文本，get_ch_lable_v函数将文本数组转换成向量。具体如下。

代码9-25 rnnwordtest

```
01 import numpy as np
02 import tensorflow as tf
03 from tensorflow.contrib import rnn
04 import random
05 import time
06 from collections import Counter
07 start_time = time.time()
```

*****ebook converter DEMO Watermarks*****

```

08 def elapsed(sec):
09     if sec<60:
10         return str(sec) + " sec"
11     elif sec<(60*60):
12         return str(sec/60) + " min"
13     else:
14         return str(sec/(60*60)) + " hr"
15
16 tf.reset_default_graph()
17 training_file = 'wordstest.txt'
18
19 #处理多个中文文件
20 def readalltxt(txt_files):
21     labels = []
22     for txt_file in txt_files:
23
24         target = get_ch_label(txt_file)
25         labels.append(target)
26     return labels
27
28 #处理汉字
29 def get_ch_label(txt_file):
30     labels= ""
31     with open(txt_file, 'rb') as f:
32         for label in f:
33
34             labels = labels +label.decode('gb2312')
35
36     return labels
37
38 #优先转文件里的字符到向量
39 def get_ch_label_v(txt_file,word_num_map,txt_label=None):
40
41     words_size = len(word_num_map)
42     to_num = lambda word: word_num_map.get(word, words_s:
43     if txt_file!= None:
44         txt_label = get_ch_label(txt_file)
45
46     labels_vector = list(map(to_num, txt_label))
47     return labels_vector

```

2. 样本预处理

样本预处理工作主要是读取整体样本，并存

*****ebook converter DEMO Watermarks*****

放到training_data里，获取全部的字表words，并生成样本向量wordlabel和与向量对应关系的word_num_map。具体代码如下。

代码9-25 rnnwordtest（续）

```
48 training_data = get_ch_label(training_file)
49 print("Loaded training data...")
50
51 counter = Counter(training_data)
52 words = sorted(counter)
53 words_size= len(words)
54 word_num_map = dict(zip(words, range(words_size)))
55
56 print('字表大小:', words_size)
57 wordlabel = get_ch_label_v(training_file,word_num_map)
```

9.6.2 构建模型

本例中使用多层RNN模型，后面接入一个softmax分类，对下一个字属于哪个向量进行分类，这里认为一个字就是一类。整个例子步骤如下。

1. 设置参数定义占位符

学习率为0.001，迭代10000次，每1000次输出一次中间状态。每次输入4个字，来预测第5个字。

网络模型使用了3层的LSTM RNN，第一层

为256个cell，第二层和第三层都是512个cell。

代码9-25 rnnwordtest（续）

```
58 # 定义参数
59 learning_rate = 0.001
60 training_iters = 10000
61 display_step = 1000
62 n_input = 4
63
64 n_hidden1 = 256
65 n_hidden2 = 512
66 n_hidden3 = 512
67 #定义占位符
68 x = tf.placeholder("float", [None, n_input,1])
69 wordy = tf.placeholder("float", [None, words_size])
```

代码中定义了两个占位符x和wordy，其中，x代表输入的4个连续文字，wordy则代表一个字，由于用的是字索引向量的one_hot编码，所以其大小为words_size，代表总共的字数。

2. 定义网络结构

将x形状变换并按找时间序列裁分，然后放入3层LSTM网络，最终通过一个全连接生成words_size个节点，为后面的softmax做准备。具体代码如下。

代码9-25 rnnwordtest（续）

```
70 x1 = tf.reshape(x, [-1, n_input])
71 x2 = tf.split(x1,n_input,1)
```

*****ebook converter DEMO Watermarks*****

```
72 # 2-layer LSTM, 每层有 n_hidden 个units
73 rnn_cell = rnn.MultiRNNCell([rnn.LSTMCell(n_hidden1), rnn.
(n_hidden2), rnn.LSTMCell(n_hidden3)])
74
75 # 通过RNN得到输出
76 outputs, states = rnn.static_rnn(rnn_cell, x2, dtype=tf.t
77
78 # 通过全连接输出指定维度
79 pred = tf.contrib.layers.fully_connected(outputs[-1], word
activation_fn = None)
```

3. 定义优化器

优化器同样使用AdamOptimizer，loss使用的是softmax的交叉熵，正确率是统计one_hot中索引对应的位置相同的个数。

代码9-25 rnnwordtest（续）

```
80 # 定义loss与优化器
81 loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_l
(logits=pred, labels=wordy))
82 optimizer = tf.train.AdamOptimizer(learning_rate=learning_
minimize(loss)
83
84 # 模型评估
85 correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(size
86 accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float
```

4. 训练模型

在训练过程中同样添加保存检查点功能。在session中每次随机取一个偏移量，然后取后面4个文字向量当作输入，第5个文字向量当作标签用来计算loss。

*****ebook converter DEMO Watermarks*****

代码9-25 rnnwordtest (续)

```
87 savedir = "log/rnnword/"
88 saver = tf.train.Saver(max_to_keep=1) # 生成s
89
90 # 启动session
91 with tf.Session() as session:
92     session.run(tf.global_variables_initializer())
93     step = 0
94     offset = random.randint(0,n_input+1)
95     end_offset = n_input + 1
96     acc_total = 0
97     loss_total = 0
98
99     kpt = tf.train.latest_checkpoint(savedir)
100    print("kpt:",kpt)
101    startepo= 0
102    if kpt!=None:
103        saver.restore(session, kpt)
104        ind = kpt.find("-")
105        startepo = int(kpt[ind+1:])
106        print(startepo)
107        step = startepo
108
109    while step < training_iters:
110
111        # 随机取一个位置偏移
112        if offset > (len(training_data)-end_offset):
113            offset = random.randint(0, n_input+1)
114
115        inwords = [ [wordlabel[ i]] for i in range(offset, offset+n_input) ] # 按照指定的位置偏移获得后4个文字向量, 当作输入
116
117        inwords = np.reshape(np.array(inwords), [-1, n_input])
118
119        out_onehot= np.zeros([words_size], dtype=float)
120        out_onehot[wordlabel[offset+n_input]] = 1.0
121        out_onehot = np.reshape(out_onehot,[1, -1])#所有的
122
123        _, acc, lossval, onehot_pred = session.run([opt: loss, pred],feed_dict={x: inwords, wordy: out_onehot})
124        loss_total += lossval
125        acc_total += acc
126        if (step+1) % display_step == 0:
127            print("Iter= " + str(step+1) + ", Average Loss= " + str(lossval))
128            "%.6f".format(loss_total/display_step))
```

```
129         Accuracy= " + \
130             "{:.2f}%".format(100*acc_total/display)
131         acc_total = 0
132         loss_total = 0
133         in2 = [words[wordlabel[i]] for i in range(0,n_input)]
134         out2 = words[wordlabel[offset + n_input]]
135         out_pred=words[int(tf.argmax(onehot_pred, 1))]
136         print("%s - [%s] vs [%s]" % (in2,out2,out_pred))
137         saver.save(session, savedir+"rnnwordtest.cpkt", global_step=step)
138         step += 1
139         offset += (n_input+1) #调整下一次迭代使用的偏移量
140     print("Finished!")
141     saver.save(session, savedir+"rnnwordtest.cpkt", global_step=step)
142     print("Elapsed time: ", elapsed(time.time() - start_
```

由于检查点文件是建立在log/rnnword/目录下的，所以在运行程序之前需要先在代码文件的当前目录下依次建立log/rnnword/文件夹（有兴趣的读者可以改成自动创建）。

运行代码，训练模型得到如下输出：

```
.....  
Type is unsupported, or the types of the items don't match 1  
CollectionDef.  
'dict' object has no attribute 'name'  
Iter= 9000, Average Loss=0.585445, Average Accuracy=79.10%  
['注','了','精','神'] - [的]vs[了]  
WARNING:tensorflow:Error encountered when serializing LAYER  
Type is unsupported, or the types of the items don't match 1  
CollectionDef.  
'dict' object has no attribute 'name'  
Iter= 10000, Average Loss= 0.409709, Average Accuracy= 85.60%  
['平','凡','、','再'] - [普]vs[普]  
WARNING:tensorflow:Error encountered when serializing LAYER  
Type is unsupported, or the types of the items don't match 1  
CollectionDef.  
'dict' object has no attribute 'name'
```

*****ebook converter DEMO Watermarks*****

```
Finished!
WARNING:tensorflow:Error encountered when serializing LAYER_
Type is unsupported, or the types of the items don't match 1
CollectionDef.
'dict' object has no attribute 'name'
Elapsed time: 1.3609554409980773 min
```

迭代10 000次的正确率达到了85%。达到了模型基本可用的状态。当然这只是个例子，读者可以尝试在模型中添加全连接及更多节点的LSTM或是更深层的LSTM来优化识别率，并且当学习的字数变多时，还会有更强大的拟合功能。

5. 运行模型生成句子

启用一个循环，等待输入文字，当收到输入的文本后，通过eval计算onehot_pred节点，并进行文字的转义，得到预测文字。接下来将预测文字再循环输入模型中，预测下一个文字。代码中设定循环32次，输出32个文字。

代码9-25 rnnwordtest（续）

```
143 while True:
144     prompt = "请输入%s个字: " % n_input
145     sentence = input(prompt)
146     inputword = sentence.strip()
147
148     if len(inputword) != n_input:
149         print("您输入的字符长度为: ", len(inputword), "请")
150         continue
151     try:
152         inputword = get_ch_table_v(None, word_num_map)
153
154         for i in range(32):
```

*****ebook converter DEMO Watermarks*****

```
155     keys = np.reshape(np.array(inputword), |  
156         onehot_pred = session.run(pred, feed_dict=  
157             onehot_pred_index = int(tf.argmax(onehot  
158                 sentence = "%s%s" % (sentence, words[onehot  
159                     inputword = inputword[1:]  
160                         inputword.append(onehot_pred_index)  
161                         print(sentence)  
162             except:  
163                 print("该字我还没学会")
```

运行代码，输出如下：

请输入4个字：生活平凡
生活平凡，要坚持一种信念，默守一种精神，为自己积淀站立的信心，默守一种精

在本例中，输入了“生活平凡”4个字，可以看到神经网络自动按照这个开头开始往下输出句子，看起来语句还算通顺。

9.7 语言模型的系统学习

语言模型包括文法语言模型和统计语言模型。一般我们指的是统计语言模型。

9.7.1 统计语言模型

统计语言模型是指：把语言（词的序列）看作一个随机事件，并赋予相应的概率来描述其属于某种语言集合的可能性。

统计语言模型的作用是，为一个长度为 m 的字符串确定一个概率分布 $P(w_1; w_2; \dots; w_m)$ ，表示其存在的可能性。其中， $w_1 \sim w_m$ 依次表示这段文本中的各个词。用一句话简单地说就是计算一个句子的概率大小。

用这种模型来衡量一个句子的合理性，概率越高，说明越符合人们说出来的自然句子，另一个用处是通过这些方法均可以保留住一定的词序信息，获得一个词的上下文信息。

9.7.2 词向量

前面9.6节的例子可以看作是一个统计语言模型，所使用的词向量是one-hot编码，由于one-hot

编码中所有的字都是独立的，所以该语言模型学到的词与词的上下文信息只能存放在网络节点中。

而现实生活中，我们人类对字词的理解却并非如此，例如“手”和“脚”，会自然让人联想到人体的器官，而“墙”则与人体器官相差甚远。这表明本身的词与词之间是有远近关系的。如果让机器学习这种关系并能加以利用，那么便可以使机器像人一样理解语言的意义。

1. 词向量解释

在神经网络中是通过一个描述词分布关系的方法来实现语义的理解，这种方法描述的词与 one_hot 描述的词都可以叫做词向量，但它还有个另外的名字叫 word embedding（词嵌入）。如何理解呢？将 one_hot 词向量中的每一个元素由整型改为浮点型，变为整个实数范围的表示；然后将原来稀疏的巨大维度压缩嵌入到一个更小维度的空间内，如图9-22所示。

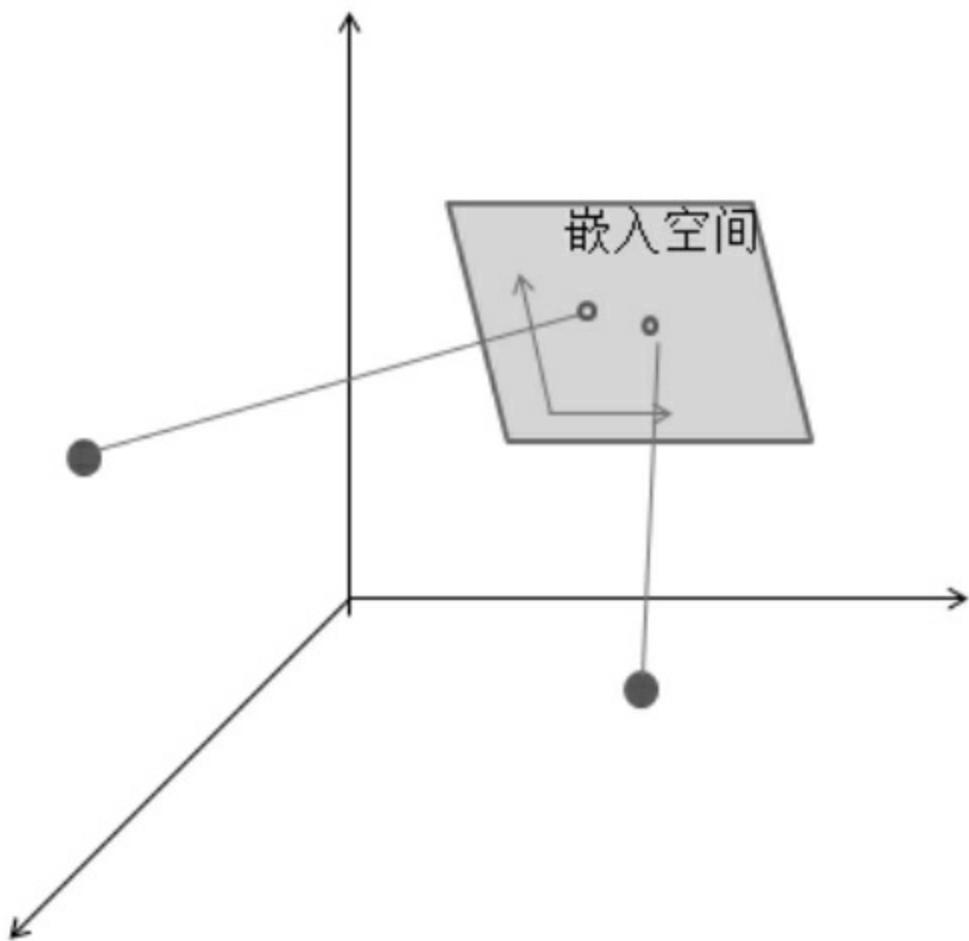


图9-22 词嵌入

图9-22中只举了个例子，将三维的向量映射到二维平面里。实际在语言模型中，常常是将二维的张量[batch, 字的index]映射到多维空间[batch, embedding的index]。即，embedding中的元素将不再是一个字，而变成了字所转化的多维向量，所有向量之间是有距离远近关系的。

其实one_hot的映射也是这种方法，把每个字表示为一个很长的向量。这个向量的维度是词表大小，其中绝大多数元素为0，只有一个维度的值为1，这个维度就代表了当前的字。one_hot映射

*****ebook converter DEMO Watermarks*****

与词嵌入的唯一区别就是仅仅将字符符号化，不包含任何语义信息而已。

word embedding的映射方法是建立在分布假说（distributional hypothesis）基础上的，即假设词的语义由其上下文决定，上下文相似的词，其语义也相似。

词向量的核心步骤由两部分组成：

- (1) 选择一种方式描述上下文。
- (2) 选择一种模型刻画某个词（下文称“目标词”）与其上下文之间的关系。

一般来讲就是使用前面介绍的语言模型来完成这种任务。这类方法的最大优势在于可以表示复杂的上下文。

2. 词向量训练

在神经网络训练的词嵌入（word embedding）中，一般会将所有的embedding随机初始化，然后在训练过程中不断更新embedding矩阵的值。对于每一个词与它对应向量的映射值，在TensorFlow中使用了一个叫tf.nn.embedding_lookup的方法来完成。

举例如下：

```
with tf.device("/cpu:0"):
    embedding = tf.get_variable("embedding", [vocab_size, size])
    inputs = tf.nn.embedding_lookup(embedding, input_data)
```

上面的代码先定义的embedding表示有vocab_size个词，每个词的向量个数为size个。最终得到的inputs就是输入向量input_data映射好的词向量了。比如input_data的形状为[batch_size, ndim]，那么inputs就为[batch_size, ndim, size]。



注意： · 由于该词向量定义好之后是需要在训练中优化的，所以embedding类型必须是tf.Variable，并且trainable=True（default）。

· embedding_lookup这个函数目前只支持在CPU上运行。

3. 候选采样技术

对于语言模型相关问题，本质上还是属于多分类问题。对于多分类问题，一般的做法是在最后一层生成与类别相等维度的节点，然后根据输入样本对应的标签来计算损失值，最终反向传播优化参数。但是由于词汇量的庞大，导致要分类的基数也会非常巨大，这会使得最后一层要有海量的节点来对应词汇的个数（如上亿的词汇量），并且还要对其逐个计算概率值，判断其是该词汇的可能性。这种做法会使训练过程变得非

*****ebook converter DEMO Watermarks*****

常缓慢，进而无法完成任务。

为了解决这个问题，可使用一种候选采样的技巧，每次只评估所有类别的一个很小的子集，让网络的最后一层只在这个子集中做每个类别的评估计算。因为是监督学习，所以能够知道对应的正确标签（即正样本），额外挑选的子集（对应标签为0）被称为负样本。这样来训练网络，可在保证效率的同时同样会有很好的效果。

4. 词向量的应用

在自然语言处理中，一般都会将该任务中涉及的词训练成词向量。然后让每个词以词向量的形式作为神经网络模型的输入，进行一些指定任务的训练。对于一个完整的训练任务，词向量的训练更多的情况是发生在预训练环节。

词向量也可以理解成为onehot的升级版特征映射。从这个角度来看，只要样本序列彼此间有着某种联系，即使不是词，也可以用这种方法处理。例如，在做恶意域名分析检测任务中，可以把某一个域名字符当作一个词，进行词向量的训练。然后再将每个字符用训练好的一组特定的向量进行映射，作为后面模型真实的输入。这样的输入就会比单纯的onehot编码映射效果好很多。

9.7.3 word2vec

*****ebook converter DEMO Watermarks*****

word2vec是谷歌提出的一种词嵌入的工具或者算法集合，采用了两种模型（CBOW与Skip-Gram模型）与两种方法（负采样与层次softmax方法）的组合，比较常见的组合为Skip-Gram和负采样方法。因为其速度快、效果好而广为人知，在任何场合可直接使用。

CBOW模型（Continous Bag of Words Model, CBOW）和Skip-Gram模型都是可以训练出词向量的方法，在具体代码操作中可以只选择其一，但CBOW要比Skip-Gram更快一些。

1. CBOW&Skip-Gram

前文说过统计语言模型就是给出几个词，在这几个词出现的前提下计算某个词出现的概率（事后概率）。

CBOW也是统计语言模型的一种，顾名思义就是根据某个词前面的n个词或者前后n个连续的词，来计算某个词出现的概率。

Skip-Gram模型与之相反，是根据某个词，然后分别计算它前后出现某几个词的各个概率。

如例，“我爱人工智能”对于CBOW模型来讲，首先会将所有的字转成one_hot，然后取出其中的一个字当作输入，将其前面和后面的字分别当作标签，拆分成如下样子：

“我” “爱”

“爱” “我”

“爱” “人”

“人” “爱”

“人” “工”

每一行代表一个样本，第一列代表输入，第二列代表标签。将输入数据送进神经网络（如“我”），同时将输出的预测值与标签（“爱”）计算loss（如输入“我”对应的标签为“爱”，模型的预测输出值为“好”，则计算“爱”和“好”之间的损失偏差，用来优化网络），进行迭代优化，在整个词库中如果字数特别多，会产生很大的矩阵，影响softmax速度。

word2vec使用基于Huffman编码的Hierarchical softmax筛选掉了一部分不可能的词，然后又用negative sampling再去掉了一些负样本的词，所以时间复杂度就从 $O(V)$ 变成了 $O(\log V)$ 。

2. TensorFlow的word2vec

在TensorFlow中提供了几个候选采样函数，用来处理loss计算中候选采样的工作，它们按不

同的采样规则被封装成了不同的函数，说明如下。

- `tf.nn.uniform_candidate_sampler`: 均匀地采样出类别子集。
- `tf.nn.log_uniform_candidate_sampler`: 按照 log-uniform (Zipfian) 分布采样。zipfian叫齐夫分布，指只有少数词经常被使用，大部分词很少被使用。
- `tf.nn.learned_unigram_candidate_sampler`: 按照训练数据中出现的类别分布进行采样。
- `tf.nn.fixed_unigram_candidate_sampler`: 按照用户提供的概率分布进行采样。

在实际使用中一般先通过统计或者其他渠道知道待处理的类别满足哪些分布，接着就可以指定函数（或是在 `nn.fixed_unigram_candidate_sampler` 中指定对应的分布）来进行候选采样。如果实在不知道类别分布，还可以用 `tf.nn.learned_unigram_candidate_sampler`。
`learned_unigram_candidate_sampler` 的做法是先初始化一个 `[0, range_max]` 的数组，数组元素初始为 1，在训练过程中碰到一个类别，就将相应数组元素加 1，每次按照数组归一化得到的概率进行

采样来实现的。



注意： 在语言相关的任务中，词按照出现频率从大到小排序之后，服从Zipfian分布。一般会先对类别按照出现频率从大到小排序，然后使用`log_uniform_candidate_sampler`函数。

TensorFlow的word2vec实现里，比对目标样本的损失值、计算softmax、负采样等过程统统封装到了`nce_loss`函数中，其默认使用的是`log_uniform_candidate_sampler`采样函数，在不指定特殊的采样器时，在该函数实现中会把词频越大的词，其类别编号也定义得越大，即优先采用词频高的词作为负样本，词频越高越有可能成为负样本。`nce_loss`函数配合优化器可以对最后一层的权重进行调优，更重要的是其还会以同样的方式调节`word embedding`（词嵌入）中的向量，让它们具有更合理的空间关系。

下面先来看看`nce_loss`函数的定义：

```
def nce_loss(weights, biases, inputs, labels, num_sampled, r
             num_true=1,
             sampled_values=None,
             remove_accidental_hits=False,
             partition_strategy="mod",
             name="nce_loss")
```

假设输入数据是K维的，一共有N个类，其参

*****ebook converter DEMO Watermarks*****

数说明如下。

- weight: shape为 (N, K) 的权重。
- biases: shape为 (N) 的偏执。
- inputs: 输入数据, shape为 (batch_size, K) 。
- labels: 标签数据, shape为 (batch_size, num_true) 。
- num_true: 实际的正样本个数。
- num_sampled: 采样出多少个负样本。
- num_classes: 类的个数N。
- sampled_values: 采样出的负样本, 如果是None, 就会用默认的sampler去采样, 优先采用词频高的词作为负样本。
- remove_accidental_hits: 如果采样时采样到的负样本刚好是正样本, 是否要去掉。
- partition_strategy: 对weights进行embedding_lookup时并行查表时的策略。TensorFlow的embedding_lookup是在CPU里实现的, 这里需要考虑多线程查表时的锁的问题。

*****ebook converter DEMO Watermarks*****



注意： 在TensorFlow中还有一个类似于 nce_loss 的函数 sampled_softmax_loss，其用法与 nce_loss 函数完全一样。不同的是内部实现， nce_loss 函数可以进行多标签分类问题，即标签之前不互斥，原因在于其对每一个输出的类都连接一个 logistic 二分类。而 sampled_softmax_loss 只能对单个标签分类，即输出的类别是互斥的，原因是其对每个类的输出放在一起统一做了一个多分类操作。

9.7.4 实例70：用CBOW模型训练自己的 word2vec

本例将使用CBOW模型来训练word2vec，最终将所学到的词向量分布关系可视化出来，同时通过该例子练习使用nce_loss函数与word embedding技术，实现自己的word2vec。

实例描述

准备一段文字作为训练的样本，对其使用 CBOW 模型计算 word2vec，并将各个词的向量关系用图展示出来。

1. 引入头文件

本例的最后需要将词向量可视化出来。第7

~12行代码是可视化相关的引入，即初始化，通过设置mpl的值让plot能够显示中文信息。Scikit-Learn的t-SNE算法模块的作用是非对称降维，是结合了t分布将高维空间的数据点映射到低维空间的距离，主要用于可视化和理解高维数据。

代码9-26 word2vect

```
01 import numpy as np
02 import tensorflow as tf
03 import random
04 import collections
05 from collections import Counter
06 import jieba
07
08 from sklearn.manifold import TSNE
09 import matplotlib as mpl
10 import matplotlib.pyplot as plt
11 mpl.rcParams['font.sans-serif']=['SimHei'] #用来正常显示中文
12 mpl.rcParams['font.family'] = 'STSong'
13 mpl.rcParams['font.size'] = 20
```

这次重点关注的是词，不再对字进行one_hot处理，所以需要借助分词工具将文本进行分词处理。本例中使用的是jieba分词库，需要使用之前先安装该分词库。

在“运行”中，输入cmd，进入命令行模式。
保证计算机联网状态下在命令行里输入：

```
Pip install jieba
```

安装完毕后可以新建一个py文件， 使用如下代码简单测试一下：

```
import jieba  
seg_list = jieba.cut("我爱人工智能")      # 默认是精确模式  
print(" ".join(seg_list))
```

如果能够正常运行并且可以分词， 就表明jieba分词库安装成功了。

2. 准备样本创建数据集

这个环节使用一篇笔者在另一个领域发表的比较有深度的文章“阴阳人体与电能.txt”来做样本， 将该文件放到代码的同级目录下。

代码中使用get_ch_label函数将所有文字读入training_data， 然后在fenci函数里使用jieba分词库对training_data分词生成training_ci， 将training_ci放入build_dataset里并生成指定长度（350）的字典。

代码9-26 word2vect（续）

```
14 training_file = '人体阴阳与电能.txt'  
15  
16 #中文字  
17 def get_ch_label(txt_file):  
18     labels= ""  
19     with open(txt_file, 'rb') as f:  
20         for label in f:  
21             labels+=label
```

*****ebook converter DEMO Watermarks*****

```

22         labels = labels+label.decode('gb2312')
23
24     return labels
25
26 #分词
27 def fenci(training_data):
28     seg_list = jieba.cut(training_data)    # 默认是精确模式
29     training_ci = " ".join(seg_list)
30     training_ci = training_ci.split()
31     #用空格将字符串分开
32     training_ci = np.array(training_ci)
33     training_ci = np.reshape(training_ci, [-1, ])
34     return training_ci
35
36 def build_dataset(words, n_words):
37
38     """Process raw inputs into a dataset."""
39     count = [['UNK', -1]]
40     count.extend(collections.Counter(words).most_common(n_words))
41     dictionary = dict()
42     for word, _ in count:
43         dictionary[word] = len(dictionary)
44     data = list()
45     unk_count = 0
46     for word in words:
47         if word in dictionary:
48             index = dictionary[word]
49         else:
50             index = 0    # dictionary['UNK']
51             unk_count += 1
52         data.append(index)
53     count[0][1] = unk_count
54     reversed_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
55     return data, count, dictionary, reversed_dictionary
56
57 training_data = get_ch_label(training_file)
58 print("总字数", len(training_data))
59 training_ci = fenci(training_data)
60 print("总词数", len(training_ci))
61 training_label, count, dictionary, words = build_dataset(
62     training_ci, n_words=350)
63 words_size = len(dictionary)
64 print("字典词数", words_size)
65
66 print('Sample data', training_label[:10], [words[i] for i in
       label[:10]])

```

build_dataset中的实现方式是将统计词频0号位置给unknown（用UNK表示），其余按照频次由高到低排列。unknown的获取按照预设词典大小，比如350，则频次排序靠后于350的都视为unknown。

运行代码，生成结果如下：

```
总字数 1567
总词数 961
字典词数 350
Sample data [263, 31, 38, 30, 27, 0, 10, 9, 104, 197] ['阴阳
```

程序显示整个文章的总字数为1567个，总词数为961个，建立好的字典词数为350。接下来是将文字里前10个词即对应的索引显示出来。

3. 获取批次数据

定义generate_batch函数，取一定批次的样本数据。

代码9-26 word2vect（续）

```
67 data_index = 0
68 def generate_batch(data, batch_size, num_skips, skip_window):
69
70     global data_index
71     assert batch_size % num_skips == 0
72     assert num_skips <= 2 * skip_window
73
```

*****ebook converter DEMO Watermarks*****

```

74     batch = np.ndarray(shape=(batch_size), dtype=np.int32)
75     labels = np.ndarray(shape=(batch_size, 1), dtype=np.int32)
76     span = 2 * skip_window + 1    #每一个样本由前skip_window + 
77     buffer = collections.deque(maxlen=span)
78
79     if data_index + span > len(data):
80         data_index = 0
81
82     buffer.extend(data[data_index:data_index + span])
83     data_index += span
84
85     for i in range(batch_size // num_skips):
86         target = skip_window # target 在buffer中的索引为skip_
87         targets_to_avoid = [skip_window]
88         for j in range(num_skips):
89             while target in targets_to_avoid:
90                 target = random.randint(0, span - 1)
91
92             targets_to_avoid.append(target)
93             batch[i * num_skips + j] = buffer[skip_window]
94             labels[i * num_skips + j, 0] = buffer[target]
95
96         if data_index == len(data):
97             buffer = data[:span]
98             data_index = span
99         else:
100            buffer.append(data[data_index])
101            data_index += 1
102
103     # 注意防止越界
104     data_index = (data_index + len(data) - span) % len(data)
105     return batch, labels
106
107 batch, labels = generate_batch(training_label, batch_size,
108                                skips=2, skip_window=1)
109 for i in range(8):# 先循环8次，然后将组合好的样本与标签打印出来
110     print(batch[i], words[batch[i]], '->', labels[i, 0], '\n')

```

generate_batch函数中使用CBOW模型来构建样本，是从开始位置的一个一个字作为输入，然后将其前面和后面的字作为标签，再分别组合在一起变成2组数据。运行当前代码，输出如下：

*****ebook converter DEMO Watermarks*****

```
31 人体 -> 38 与
31 人体 -> 263 阴阳
38 与 -> 31 人体
38 与 -> 30 电能
30 电能 -> 27 阴
30 电能 -> 38 与
27 阴 -> 0 UNK
27 阴 -> 30 电能
```

如果是Skip-Gram方法，根据字取标签的方法正好相反，输出会变成以下这样：

```
263 阴阳 -> 31 人体
38 与 -> 31 人体
31 人体 -> 38 与
31 人体 -> 263 阴阳
30 电能 -> 38 与
.....
```

4. 定义取样参数

下面代码中每批次取128个，每个词向量的维度为128，前后取词窗口为1，num_skips表示一个input生成2个标签，nce中负采样的个数为num_sampled。接下来是验证模型的相关参数，valid_size表示在0- words_size/2中的数取随机不能重复的16个字来验证模型。

代码9-26 word2vect（续）

```
111 batch_size = 128
112 embedding_size = 128      # embedding vector的维度
113 skip_window = 1           # 左右的词数量
```

*****ebook converter DEMO Watermarks*****

```
114 num_skips = 2          # 一个input生成2个标签  
115  
116 valid_size = 16  
117 valid_window = words_size/2    # 取样数据的分布范围  
118 valid_examples = np.random.choice(valid_window, valid_size, replace=False)  
#0- words_size/2中的数取16个。不能重复  
119 num_sampled = 64        # 负采样个数
```

5. 定义模型变量

初始化图，为输入、标签、验证数据定义占位符，定义词嵌入变量embeddings为每个字定义128维的向量，并初始化为-1~1之间的均匀分布随机数。tf.nn.embedding_lookup是将输入的train_inputs转成对应的128维向量embed，定义nce_loss要使用的nce_weights和nce_biases。

代码9-26 word2vect（续）

```
120 tf.reset_default_graph()  
121  
122 train_inputs = tf.placeholder(tf.int32, shape=[batch_size])  
123 train_labels = tf.placeholder(tf.int32, shape=[batch_size])  
124 valid_dataset = tf.constant(valid_examples, dtype=tf.int32)  
125  
126 # CPU上执行  
127 with tf.device('/cpu:0'):  
128     # 查找embeddings  
129     embeddings = tf.Variable(tf.random_uniform([words_size, embedding_size], -1.0, 1.0)) #94个字，每个128个向量  
130  
131     embed = tf.nn.embedding_lookup(embeddings, train_inputs)  
132  
133     # 计算NCE的loss的值  
134     nce_weights = tf.Variable(tf.truncated_normal([words_size, embedding_size],  
135                                         stddev=1.0 / tf.sqrt(np.float32(words_size))))  
136
```

*****ebook converter DEMO Watermarks*****

```
137     nce_biases = tf.Variable(tf.zeros([words_size]))
```

在反向传播中，embeddings会与权重一起被nce_loss代表的loss值所优化更新。

6. 定义损失函数和优化器

使用nce_loss计算loss来保证softmax时的运算速度不被words_size过大问题所影响，在nce中每次会产生num_sampled（64）个负样本来参与概率运算。优化器使用学习率为1的GradientDescentOptimizer。

代码9-26 word2vect（续）

```
138 loss = tf.reduce_mean(  
139 tf.nn.nce_loss(weights=nce_weights, biases=nce_biases,  
140                 labels=train_labels, inputs=embed,  
141                 num_sampled=num_sampled, num_classes=words_size)  
142  
143 # 梯度下降优化器  
144 optimizer = tf.train.GradientDescentOptimizer(1.0).minimize(loss)  
145  
146 # 计算minibatch examples 和所有 embeddings 的 cosine 相似度  
147 norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1, True))  
148 normalized_embeddings = embeddings / norm  
149 valid_embeddings = tf.nn.embedding_lookup(normalized_embeddings,  
    valid_dataset)  
150 similarity = tf.matmul(valid_embeddings, normalized_embeddings,  
    transpose_b=True)
```

验证数据取值时做了些特殊处理，将

embeddings中每个词对应的向量进行平方和再开方得到norm，然后将embeddings与norm相除得到normalized_embeddings。当使用embedding_lookup获得自己对应normalized_embeddings中的向量valid_embeddings时，将该向量与转置后的normalized_embeddings相乘得到每个词的similarity。这个过程实现了一个向量间夹角余弦（Cosine）的计算。

7. 夹角余弦介绍

为了能够读懂代码，有必要介绍下夹角余弦的概念。

余弦定理：给定三角形的三条边 a 、 b 、 c ，对应三个角为 A 、 B 、 C ，则角 A 的余弦见式（9-1）：

$$\cos A = \frac{b^2 + c^2 - a^2}{2bc} \quad \text{式 (9-1)}$$

如果将 b 和 c 看成两个向量，则上述式子等价于（见式9-2）：

$$\cos A = \frac{\langle \mathbf{b}, \mathbf{c} \rangle}{\|\mathbf{b}\| \|\mathbf{c}\|} \quad \text{式 (9-2)}$$

分母表示两个向量的长度，分子表示两个向

量的内积。引申到二维空间中，向量A（x1, y1）与向量B（x2, y2）的夹角余弦公式见式（9-3）：

$$\cos \theta = \frac{x_1 x_2 + y_1 y_2}{\sqrt{x_1^2 + y_1^2} \sqrt{x_2^2 + y_2^2}} \quad \text{式 (9-3)}$$

再扩展到两个n维样本点，a（x11, x12, ..., x1n）和b（x21, x22, ..., x2n）的夹角余弦的公式见式（9-4）：

$$\cos \theta = \frac{x_{11} x_{21} + x_{12} x_{22} + \dots}{\sqrt{x_{11}^2 + x_{12}^2 + \dots} \sqrt{x_{21}^2 + x_{22}^2 + \dots}} \quad \text{式 (9-4)}$$

这回可以理解前面的代码了，norm代表每一个词对应向量的长度矩阵，见式（9-5）：

$$norm = \begin{Bmatrix} \sqrt{x_{11}^2 + x_{12}^2 + \dots} \\ \sqrt{x_{21}^2 + x_{22}^2 + \dots} \\ \sqrt{x_{31}^2 + x_{32}^2 + \dots} \\ \dots \end{Bmatrix} \quad \text{式 (9-5)}$$

normalized_embeddings表示的意思是向量除以自己的模，即单位向量，它可以确定向量的方向。

很显然，similarity就是valid_dataset中对应的单位向量valid_embeddings与整个词嵌入字典中单位向量的夹角余弦。

如图9-23所示，算了这么多夹角余弦的目的就是为了衡量两个n维向量间的相似程度。当 $\cos\theta$ 为1时，表明夹角为0，即两个向量的方向完全一样。所以当 $\cos\theta$ 的值越小，表明两个向量的方向越不一样，相似度越低。

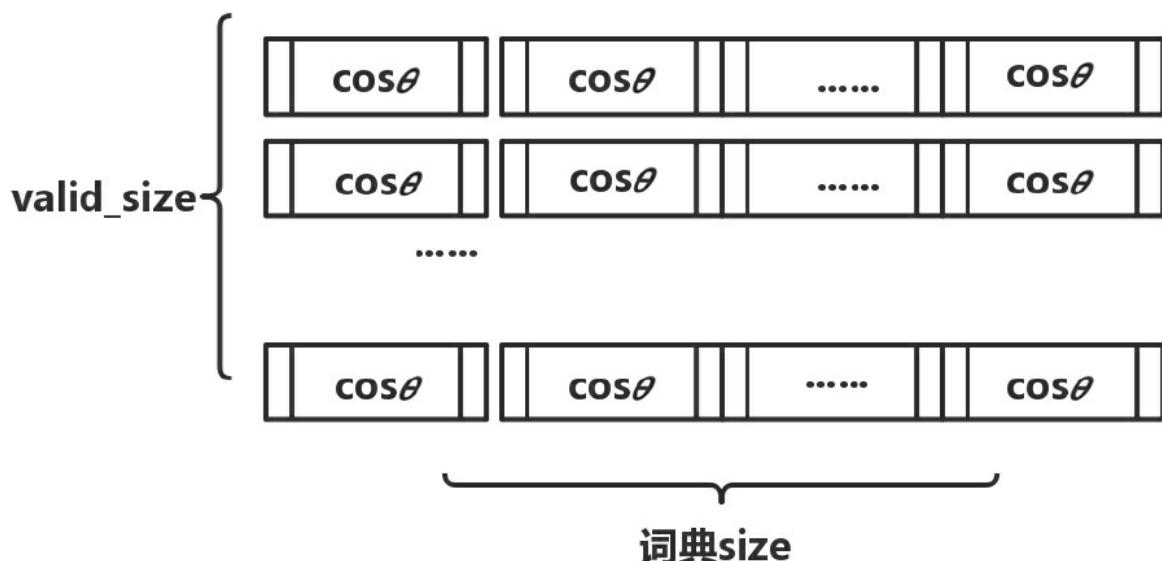


图9-23 词嵌入夹角余弦结构

8. 启动session，训练模型

有了理论基础之后，对代码的模型应该好理解了，接下来启动session将模型训练出来。

代码9-26 word2vect（续）

```
151 num_steps = 100001
152 with tf.Session() as sess:
153     sess.run( tf.global_variables_initializer() )
154     print('Initialized')
155
156     average_loss = 0
157     for step in range(num_steps):
158         batch_inputs, batch_labels = generate_batch(trai:
batch_size, num_skips, skip_window)
159         feed_dict = {train_inputs: batch_inputs, train_:
labels}
160
161         _, loss_val = sess.run([optimizer, loss], feed_(
162             average_loss += loss_val
163
164 #通过打印测试可以看到, embed的值在逐渐被调节
165         emv = sess.run(embed, feed_dict = {train_inputs:
166             print("emv-----", emv[0])
167
168         if step % 2000 == 0:
169             if step > 0:
170                 average_loss /= 2000
171             # 平均loss
172             print('Average loss at step ', step, ': ', ave
173             average_loss = 0
```

这里设置的迭代次数为10 0001次，每迭代2000次就输出一次loss值。

9. 输入验证数据，显示效果

为了能够看到词向量的效果，添加如下代码，将验证数据输入模型中，找出与其相近的词。这里使用了一个argsort函数，是将数组中的值从小到大排列后，返回每个值对应的索引。在使用argsort函数之前，将sim取负，得到的就是从大到小排列的结果了。sim就是当前词与整个词典中每个词的夹角余弦，9.7.4节中讲过夹角余弦值

最大则代表相似度越高。

代码9-26 word2vect（续）

```
174         if step % 10000 == 0:  
175             sim = similarity.eval(session=sess)  
176  
177             for i in range(valid_size):  
178                 valid_word = words[valid_examples[i]]  
179  
180                 top_k = 8    # 取排名最靠前的8个词  
181                 nearest = (-sim[i, :]).argsort()[1:top_k + 1]  
182  
183                 log_str = 'Nearest to %s:' % valid_word  
184  
185                 for k in range(top_k):  
186                     close_word = words[nearest[k]]  
187                     log_str = '%s,%s' % (log_str, close_word)  
188             print(log_str)
```

运行代码，结果如下：

```
.....  
Average loss at step 92000 : 2.52053320134  
Average loss at step 94000 : 2.51920971239  
Average loss at step 96000 : 2.51831436144  
Average loss at step 98000 : 2.54135515364  
Average loss at step 100000 : 2.51433812357  
Nearest to 或:,与,和,提升,比,大小,觉得,每次,为阳来  
Nearest to , :,起来,保养,过程,分裂细胞,为什么,新,就是,感到  
Nearest to 相当于:,也,会,寿命,刺激,了,低电量,加速,在  
Nearest to 桩:,训练,来源,修道,糖,马步,睡觉,放空,第一  
Nearest to 衰退:,也,短时间,累,下来,觉得,假如,排量,的  
Nearest to 加速:,快速,跑,病变,也,输出,走,相当于,加大  
.....
```

由于样本量不大，所以结果并不精确。但是也可以看出，模型基本上按照近义词被归类了一

些，如第一个的“或”，与其最近的词有“与”“和”，基本上与人类的理解差不多。

10. 词向量可视化

接着继续编写代码，将词向量可视化。在可视化之前，将词典中的词嵌入向量转成单位向量（只有方向），然后将它们通过t-SNE降维映射到二维平面中显示。

代码9-26 word2vect（续）

```
189     final_embeddings = normalized_embeddings.eval()
190
191 def plot_with_labels(low_dim_embs, labels, filename='tsne.png'):
192     assert low_dim_embs.shape[0] >= len(labels), 'More labels than embeddings'
193     plt.figure(figsize=(18, 18)) # in inches
194     for i, label in enumerate(labels):
195         x, y = low_dim_embs[i, :]
196         plt.scatter(x, y)
197         plt.annotate(label, xy=(x, y), xytext=(5, 2), textcoords='points',
198                      ha='right', va='bottom')
199     plt.savefig(filename)
200
201 try:
202
203     tsne = TSNE(perplexity=30, n_components=2, init='pca',
204                  plot_only = 80#输出100个词
205     low_dim_embs = tsne.fit_transform(final_embeddings[plot_only, :])
206     labels = [words[i] for i in range(plot_only)]
207
208     plot_with_labels(low_dim_embs, labels)
```

上面代码运行后，输出图片如图9-24所示。

从输出图片中可以看出模型对词意义的理解。距离越近的词，他们的意义越相似，如图中的“但”与“不同”。

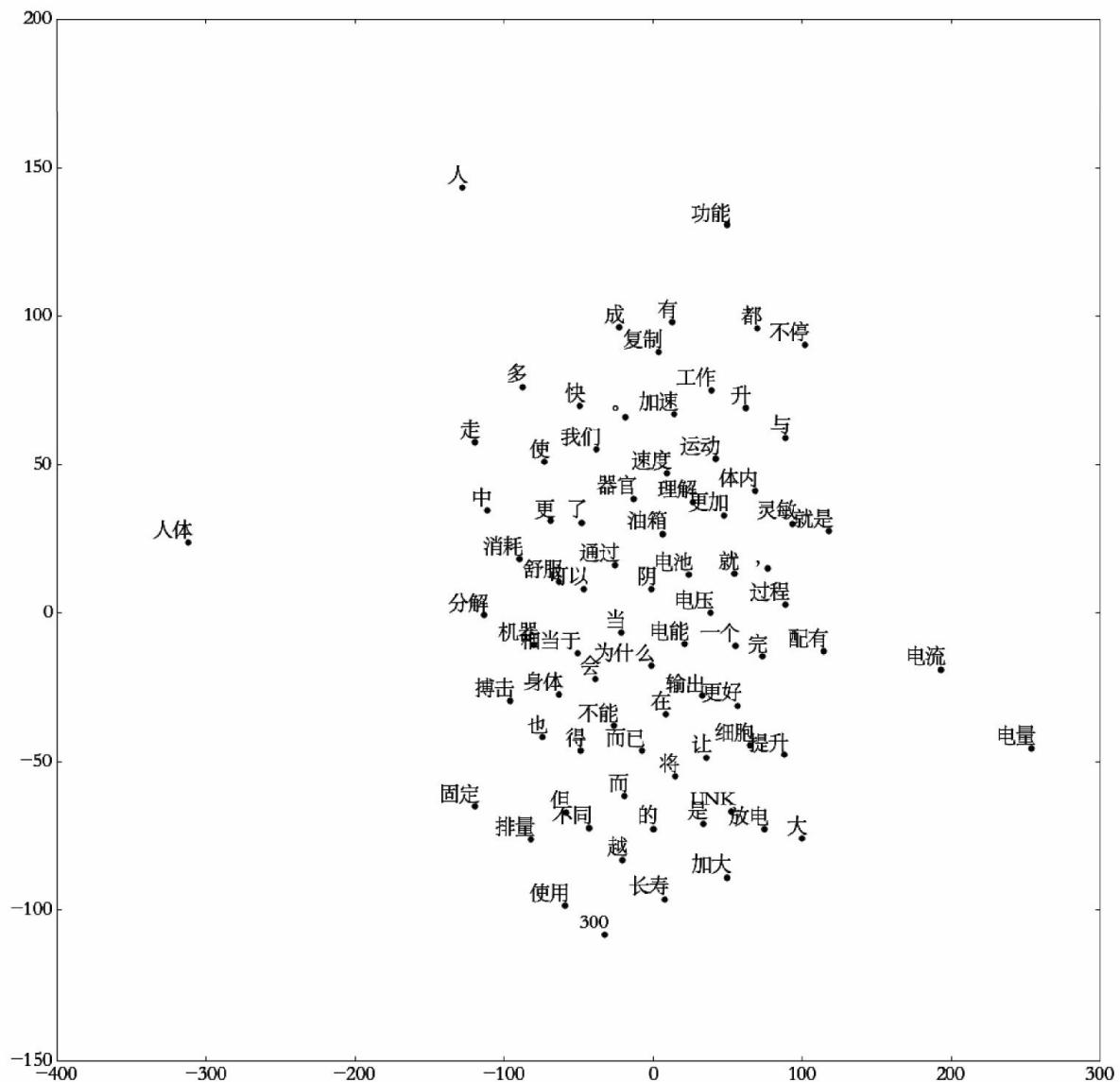


图9-24 词向量结果

9.7.5 实例71：使用指定候选采样样本训练 word2vec

上面的例子使用了`nce_loss`中默认的候选采样方法，本例将其扩展成可以手动指定候选样本来计算loss。例子中，通过手动指定词频数据生成样本，然后再根据生成的样本计算loss。这么做

虽然只是将前面的步骤换为手动执行，但该方法具有更强的通用性，使模型不仅适用于满足Zipfian分布的样本，对于其他分布的样本，只需要按照本方法配置指定分布的样本即可。具体步骤如下。

实例描述

准备一段文字作为训练的样本，对其使用CBOW模型计算得到word2vec，并将各个词的向量关系用图表示出来。其中，通过使用手动指定词频样本生成候选词的方法，来代替nce_loss中的默认选词方法。

1. 修改字典处理部分，生成词频数据

词频数据是指，对应于字典里的顺序，每个词所出现的频率统计。该数据作为候选择本采样的依据，在代码里是list类型。修改代码“9-26 word2vect.py”文件，在build_dataset函数中添加生成词频数据vocab_freqs。

代码9-27 word2vect自定义候选择样

```
01 .....
02 def build_dataset(words, n_words):
03
04     """建立数据集."""
05     count = [['UNK', -1]]
06     count.extend(collections.Counter(words).most_common(n_w
07
```

```
08     dictionary = dict()
09     vocab_freqs = [] #定义词频数据list
10     for word, nvocab in count:
11         dictionary[word] = len(dictionary)
12         vocab_freqs.append(nvocab) # 加入字典里的每个词频
13     data = list()
14     unk_count = 0
15     for word in words:
16         if word in dictionary:
17             index = dictionary[word]
18         else:
19             index = 0 # dictionary['UNK']
20             unk_count += 1
21         data.append(index)
22     count[0][1] = unk_count
23     reversed_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
24
25     return data, count, dictionary, reversed_dictionary, vocab_freqs
26 .....
27 #使用vocab_freqs接收词频数据的返回值
28 training_label, count, dictionary, words, vocab_freqs = build_dataset(
    training_file, 350)
```

2. 通过词频数据进行候选项本采样

拿到词频数据后，将其放到自定义采样的函数fixed_unigram_candidate_sampler里生成指定数量的采样数据。这里需要注意的是，原有字典中的第一个词并不是词频最高的词，而是手动添加的一个UNK词（见代码05行），并且当时设置的出现次数为-1。由于词频序数需要从大到小排列，所以需要手动将其改为最大值。通过打印vocab_freqs的数据能够看到，最大的值是89，所以设置一个比89大的数即可，这里设置的是90。

代码9-27 word2vect自定义候选项本采样（续）

```
29 .....
30     nce_weights = tf.Variable(tf.truncated_normal([words_size,
31                                         embedding_size],
32                                         stddev=1.0 / tf.sqrt(np.float32(embedding_size))))
33     nce_biases = tf.Variable(tf.zeros([words_size]))
34 vocab_freqs[0] = 90 #将手动添加的第一个词UNK的词频改为最大
35
36 sampled = tf.nn.fixed_unigram_candidate_sampler(
37             true_classes=tf.cast(train_labels, tf.int64),
38             num_true=1,
39             num_sampled=num_sampled,
40             unique=True,
41             range_max=words_size,
42             unigrams=vocab_freqs)
```



注意： 这里有个小技巧，如何知道 fixed_unigram_candidate_sampler 的定义？需要为其填写哪些参数？这里有个方法，用鼠标双击该函数，使其为选中状态，然后右击该函数，在弹出的快捷菜单中选择 Go to definition 命令（如图9-25所示），即可跳转到该函数的定义。

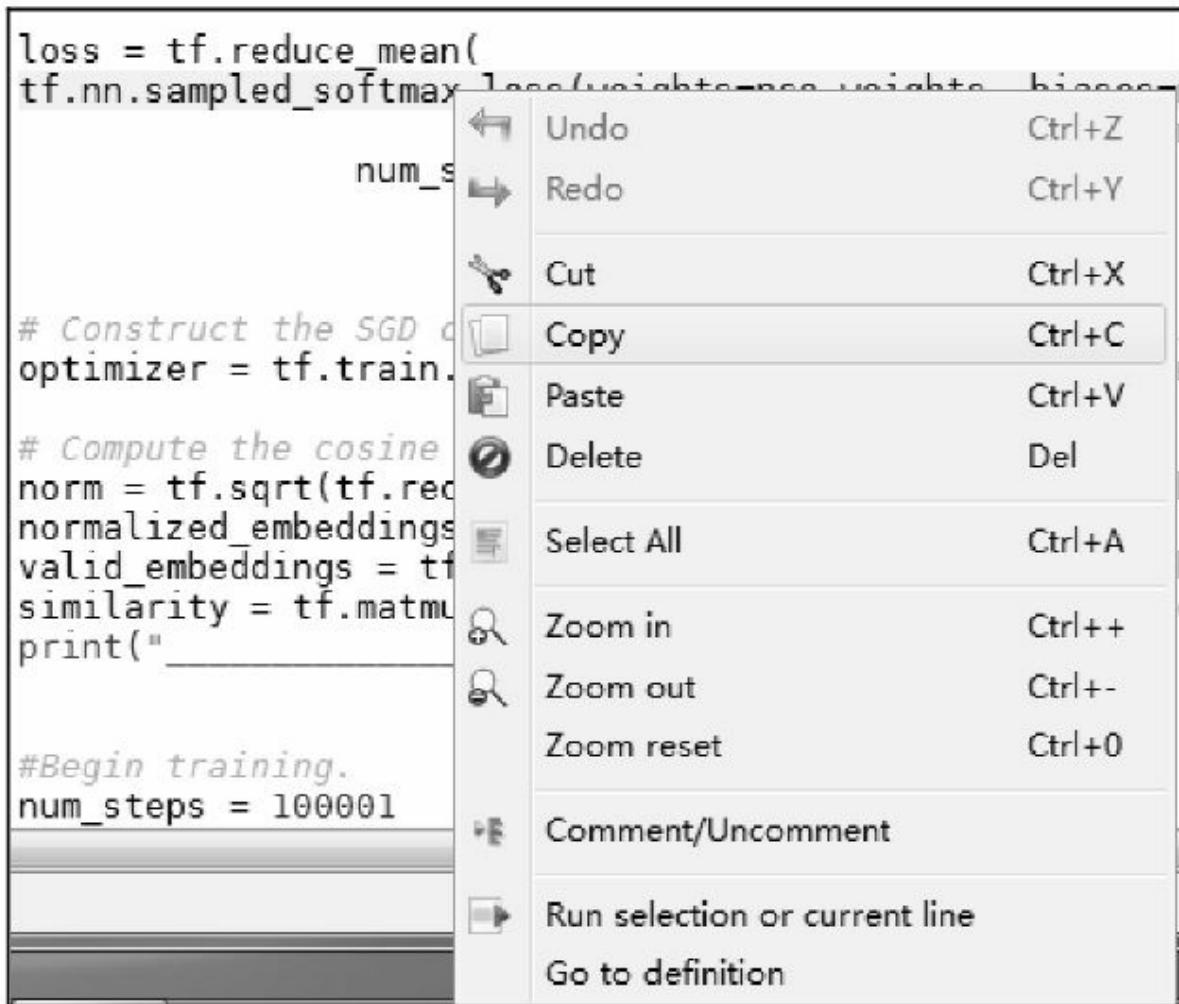


图9-25 查找函数定义

3. 使用自己的采样计算softmax的loss

使用loss生成函数sampled_softmax_loss（当然也可以用nce_loss函数）来计算loss，不同的是，在设置最后一个参数时会传入上一步生成的样本。

代码9-27 word2vect自定义候选采样（续）

43

44 loss = tf.reduce_mean(

*****ebook converter DEMO Watermarks*****

```
45 tf.nn.sampled_softmax_loss(weights=nce_weights, biases=nce_b  
46                                     labels=train_labels, inputs=embed  
47                                     num_sampled=num_sampled, num_classes=words_size  
                                     sampled_values=sampled))
```

4. 运行生成结果

其他代码都不用改动，直接运行即可，得到的效果与实例70一样，这里不再赘述。

9.7.6 练习题

(1) 想一想：9.7.4节的例子，如果将 nce_loss 改写成sampled_softmax_loss会不会有效？为什么？

答案：有效，因为对于语言模型的每个结果的输出是唯一的，也就是只会有一个词，所以也符合单标签分类。

将如下代码替换nce_loss的调用（参考配套代码“9-28 word2vect -2.py”文件）：

```
loss = tf.reduce_mean(  
    tf.nn.sampled_softmax_loss(weights=nce_weights, biases=nce_b  
                                labels=train_labels, inputs=embed,  
                                num_sampled=num_sampled, num_classes=words_size))
```

(2) 试着将9.7.5节中的例子改成按照训练数据中类别出现分布方法（9.7.3节有介绍）进行

*****ebook converter DEMO Watermarks*****

采样，看看有什么效果。

答案可参考随书代码“9-29 word2vect学习样本候选采样.py”文件。

9.8 处理Seq2Seq任务

本节继续介绍RNN的使用场景，处理Seq2Seq任务。Seq2Seq任务，即从一个序列映射到另一个序列的任务。在生活中会有很多符合这样特性的例子：前面的语言模型、语音识别例子，都可以理解成一个Seq2Seq的例子，类似的应用还有机器翻译、词性标注、智能对话等。下面就来学一下Seq2Seq任务的处理方法。

9.8.1 Seq2Seq任务介绍

Seq2Seq（Sequence 2 Sequence）任务可以理解为，从一个Sequence做某些工作映射到（to）另外一个Sequence的任务，泛指一些Sequence到Sequence的映射问题。

Sequence可以理解为一个字符串序列，在给定一个字符串序列后，希望得到与之对应的另一个字符串序列（如翻译后的、语义上对应的）。Seq2Seq不关心输入和输出的序列是否长度对应。

Seq2Seq如果再细分，可以分成输入、输出序列不一一对应和一一对应两种。前面的语言模型就是一一对应的，类似的还有词性标注，可以用图9-26所示的网络结构来理解。如果给定的每个输入都会有对应的输出，这种情况使用简单的*****ebook converter DEMO Watermarks*****

RNN模型就可以解决。而输入输出序列不对应时会比较复杂一些，除了像前面语音识别模型中双向RNN+TensorFlow中的ctc_loss组合的方式之外，还有一种相对比较主流的解决方法——Encoder-Decoder框架。

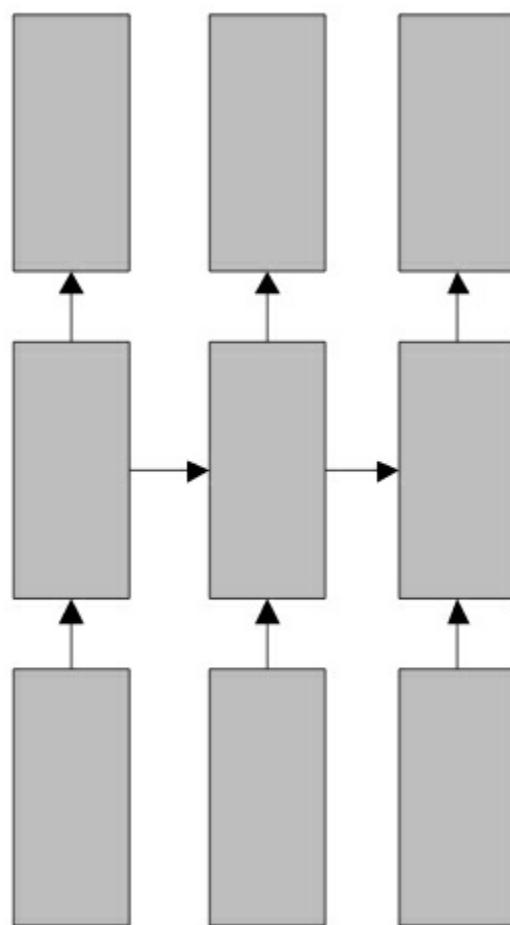


图9-26 多对多RNN

9.8.2 Encoder-Decoder框架

1. Encoder-Decoder框架介绍

Encoder-Decoder框架的工作机制是：先使用*****ebook converter DEMO Watermarks*****

Encoder将输入编码映射到语义空间（通过Encoder网络生成的特征向量），得到一个固定维数的向量，这个向量就表示输入的语义；然后再使用Decoder将这个语义向量解码，获得所需要的输出。如果输出是文本，则Decoder通常就是语言模型。其内部结构如图9-27所示。

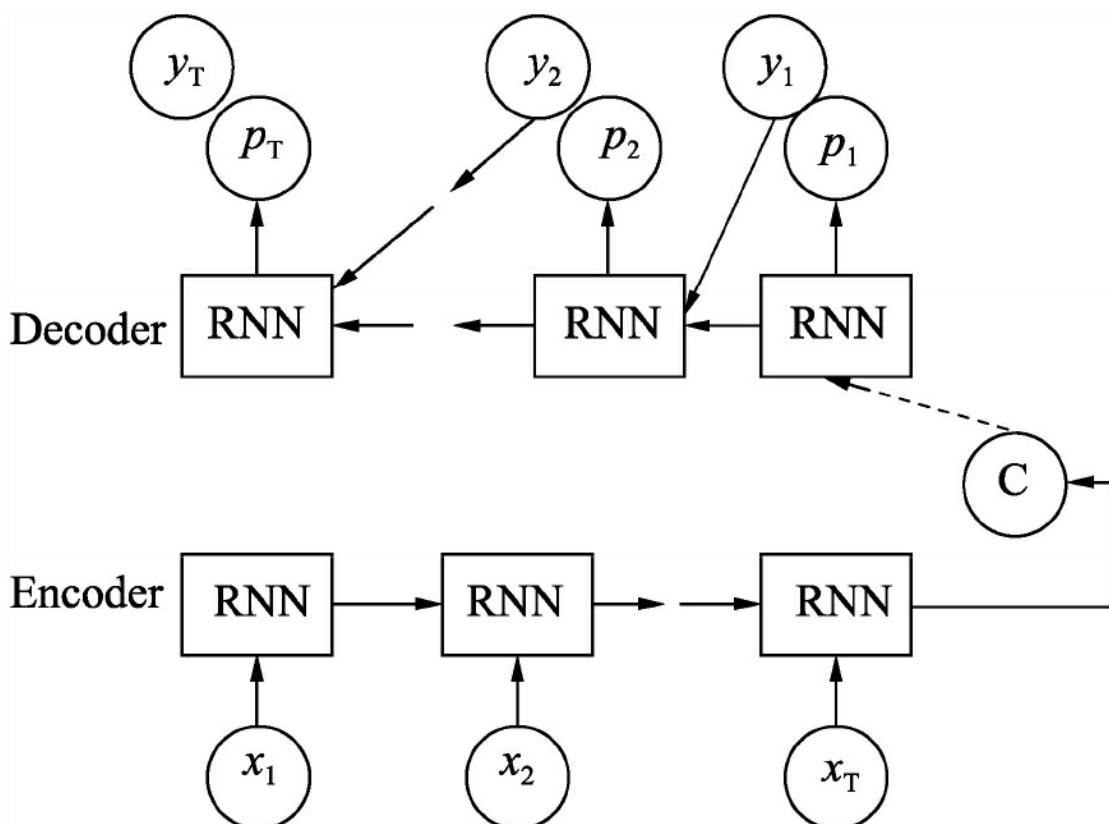


图9-27 Encoder-Decoder结构

图9-27中Encoder-Decoder框架有两个输入：一个是 x 输入作为Encoder的输入，另一个是 y 输入作为Decoder输入， x 和 y 依次按照各自的顺序传入网络。

可以看出在Seq2Seq的训练中，标签 y 既参与

计算loss，又参与节点运算，而不是像前面学习的其他网络只用来做loss监督。在Encoder与Decoder之间的C节点就是码器Encoder输出的解码向量，将它作为解码Decoder中cell的初始状态，进行对输出的解码。

这种机制的优点如下：

- 非常灵活，并不限制Encoder、Decoder使用何种神经网络，也不限制输入和输出的内容（例如image caption任务，输入是图像，输出是文本）。
- 这是一个端到端（end-to-end）的过程，将语义理解和语言生成合在了一起，而不是分开处理。

2. TensorFlow中的Seq2Seq

在TensorFlow中有两套Seq2Seq的接口。一套是TensorFlow 1.0版本之前的旧接口。在tf.contrib.legacy_seq2seq下；另一套为TensorFlow 1.0版本之后推出的新接口，在tf.contrib.seq2seq下。

旧接口的功能相对简单，是静态展开的网络模型。而新接口的功能更加强大，使用的是动态展开的网络模型，并提供了训练和应用两种场景的Helper类封装。从使用角度来看，旧接口同样

*****ebook converter DEMO Watermarks*****

也是比较简单。而新接口会更加灵活，需要自己组建Encoder和Decoder并通过函数把它们手动连接起来。

为了便于理解，本书主要以旧接口中Seq2Seq框架来举例介绍。关于Seq2Seq的更多例子，及新接口的应用演示，可以参考如下网址中的实例：

<https://github.com/ematvey/tensorflow-seq2seq-tutorials>

旧接口中基本Seq2Seq函数的定义如下。

```
tf.contrib.legacy_seq2seq.basic_rnn_seq2seq(encoder_inputs,  
                                              decoder_inputs,  
                                              cell,  
                                              dtype=dtypes.float32,  
                                              scope=None)
```

参数说明如下。

- encoder_inputs：一个形状为[batch_size x input_size]的list。
 - decoder_inputs：同encoder_inputs。
 - cell：定义的cell网络。
 - dtype：encoder_inputs和decoder_inputs中的
- *****ebook converter DEMO Watermarks*****

类型（默认是tf.float32）。

- 返回值：outputs和state。outputs为[batch_size, output_size]的张量；state为[batch_size, cell.state_size]；cell.state_size可以表示一个或者多个子cell的状态，视输入参数cell而定。

其函数的实现只有如下几行代码：

```
with variable_scope.variable_scope(scope or "basic_rnn_seq2seq"):
    enc_cell = copy.deepcopy(cell)
    _, enc_state = core_rnn.static_rnn(enc_cell, encoder_inputs)
    return rnn_decoder(decoder_inputs, enc_state, cell)
```

现将传入的cell做一次深拷贝（deepcopy），用来当做Encoder的网络，将生成的结果和原来的cell再加上输入的decoder_inputs一起放到Decoder中，并输出生成结果。



注意：在使用过程中，由于需要通过输入x来预测y，没有标签，这种情况就需要手动填充Decoder来代替训练时的标签。

9.8.3 实例72：使用basic_rnn_seq2seq拟合曲线

TensorFlow虽然对Seq2Seq的框架的封装只用一个函数就完成了。但是，Seq2Seq的这个函数用*****ebook converter DEMO Watermarks*****

起来并不友好，跟我们以前使用的TensorFlow中的函数并不是一样，所以有必要通过例子来演示一下。本例中使用2层的GRU循环网络，每层有12个节点。编码器与解码器中使用同样的网络结构。

实例描述

通过sin与cos进行叠加变形生成无规律的模拟曲线，使用Seq2Seq模式对其进行学习，拟合特征，从而达到可以预测下一时刻数据的效果。

该例子共分为以下几步。

1. 定义模拟样本函数

本例中通过函数制作规则的曲线来验证网络模型；定义两个曲线sin和cos，通过随机值将其变形偏移，将两个曲线叠加。具体代码如下。

代码9-30 基本Seq2Seq

```
01 import random
02 import math
03
04 import tensorflow as tf
05 import numpy as np
06 import matplotlib.pyplot as plt
07
08 def do_generate_x_y(isTrain, batch_size, seqlen):
09     batch_x = []
10     batch_y = []
11     for _ in range(batch_size):
```

```

12     offset_rand = random.random() * 2 * math.pi
13     freq_rand = (random.random() - 0.5) / 1.5 * 15 +
14     amp_rand = random.random() + 0.1
15
16     sin_data = amp_rand * np.sin(np.linspace(
17         seqlen / 15.0 * freq_rand * 0.0 * math.pi + (
18             seqlen / 15.0 * freq_rand * 3.0 * math.pi + (
19                 * 2) ) )
20     offset_rand = random.random() * 2 * math.pi
21     freq_rand = (random.random() - 0.5) / 1.5 * 15 +
22     amp_rand = random.random() * 1.2
23
24     sig_data = amp_rand * np.cos(np.linspace(
25         seqlen / 15.0 * freq_rand * 0.0 * math.pi + (
26             seqlen / 15.0 * freq_rand * 3.0 * math.pi + (
27                 * 2)) + sin_data
28     batch_x.append(np.array([ sig_data[:seqlen] ]).T)
29     batch_y.append(np.array([ sig_data[seqlen:] ]).T)
30
31     # 当前shape: (batch_size, seq_length, output_dim)
32     batch_x = np.array(batch_x).transpose((1, 0, 2))
33     batch_y = np.array(batch_y).transpose((1, 0, 2))
34     # 转换后shape: (seq_length, batch_size, output_dim)
35
36     return batch_x, batch_y
37
38 #生成15个连续序列，将con和sin随机偏移变化后的值叠加起来
39 def generate_data(isTrain, batch_size):
40     seq_length = 15
41     if isTrain :
42         return do_generate_x_y(isTrain, batch_size, seq_
43         length)
44     else:
45         return do_generate_x_y(isTrain, batch_size, seq_
46         length*2)

```

将该曲线按照30个序列一组的样式组成训练用的样本。30个序列分成了两部分：一部分当成现在的序列batch_x，一部分当成将来的序列batch_y。

2. 定义参数及网络结构

前面介绍过basic_rnn_seq2seq的输入是一个list，这与我们平时遇到过的模型不太一样，所以需要构建一个list，以方便传入basic_rnn_seq2seq中。

在代码中，定义3个list（encoder_input、expected_output、decode_input），按照时间序列的数量来循环创建占位符，并使用append方法放入到list中。

网络模型定义为2层的循环网络，每层12个GRUcell。用MultiRNNCell将cell定义好后与前面的list一起传入basic_rnn_seq2seq中。

生成的结果为dec_outputs，dec_outputs中为每个时刻有12个GRUcell的输出，所以还需要通过循环在每个时刻下加一个全连接层，将其转为输出维度output_dim（output_dim=1）的节点。

代码9-30 基本Seq2Seq（续）

```
45 sample_now, sample_f = generate_data(isTrain=True, batch_
46 print("training examples : ")
47 print(sample_now.shape)
48 print("(seq_length, batch_size, output_dim)")
49
50 seq_length = sample_now.shape[0]
51 batch_size = 10
52
53 output_dim = input_dim = sample_now.shape[-1]
```

```

54 hidden_dim = 12
55 layers_stacked_count = 2
56
57 # 学习率
58 learning_rate = 0.04
59 nb_iters = 100
60
61 lambda_l2_reg = 0.003 # L2 正则参数
62
63 tf.reset_default_graph()
64
65 encoder_input = []
66 expected_output = []
67 decode_input = []
68 for i in range(seq_length):
69     encoder_input.append( tf.placeholder(tf.float32, shape=[None, input_dim]) )
70     expected_output.append( tf.placeholder(tf.float32, shape=[None, output_dim]) )
71     decode_input.append( tf.placeholder(tf.float32, shape=[None, input_dim]) )
72
73 tcells = []
74 for i in range(layers_stacked_count):
75     tcells.append(tf.contrib.rnn.GRUCell(hidden_dim))
76 Mcell = tf.contrib.rnn.MultiRNNCell(tcells)
77
78 dec_outputs, dec_memory
tf.contrib.legacy_seq2seq.basic_rnn_seq2seq
(encoder_input, decode_input, Mcell)
79
80 reshaped_outputs = []
81 for ii in dec_outputs :
82     reshaped_outputs.append( tf.contrib.layers.fully_connected(
        output_
dim, activation_fn=None))

```

3. 定义loss函数及优化器

为了防止过拟合，对basic_rnn_seq2seq循环网络中的参数使用了l2_loss正则，由于最后一个全连接只是起到转化作用，就忽略不做l2_loss正则了（也可以加上，效果没有影响）。L2的调节

*****ebook converter DEMO Watermarks*****

因子设为0.003，学习率设为0.04。

代码9-30 基本Seq2Seq（续）

```
83 #计算L2的loss值
84 output_loss = 0
85 for _y, _Y in zip(reshaped_outputs, expected_output):
86     output_loss += tf.reduce_mean(tf.pow(_y - _Y, 2))
87
88 # 求正则化loss值
89 reg_loss = 0
90 for tf_var in tf.trainable_variables():
91     if not ("fully_connected" in tf_var.name):
92
93         reg_loss += tf.reduce_mean(tf.nn.l2_loss(tf_var))
94
95 loss = output_loss + lambda_l2_reg * reg_loss
96 train_op = tf.train.AdamOptimizer(learning_rate).minimize(loss)
```

预测结果与真实结果的平方差再加上l2的loss值，作为输出的loss值。优化器同样使用AdamOptimizer。

4. 启用session开始训练

在session中将训练和测试单独封装成了两个函数。在train_batch函数里先取指定批次的数据，通过循环来填充到encoder_input和expected_output列表里。

代码9-30 基本Seq2Seq（续）

```
97 sess = tf.InteractiveSession()
98
```

*****ebook converter DEMO Watermarks*****

```

99 def train_batch(batch_size):
100
101     X, Y = generate_data(isTrain=True, batch_size=batch_
102         feed_dict = {encoder_input[t]: X[t] for t in range(:_
103             input)})
104         feed_dict.update({expected_output[t]: Y[t] for t in
105             (expected_output)}))
106
107     c = np.concatenate(([np.zeros_like(Y[0])], Y[:-1]), axis=0)
108
109     feed_dict.update({decode_input[t]: c[t] for t in range(1, len(c))})
110
111     _, loss_t = sess.run([train_op, loss], feed_dict)
112     return loss_t
113
114
115 def test_batch(batch_size):
116     X, Y = generate_data(isTrain=False, batch_size=batch_
117         feed_dict = {encoder_input[t]: X[t] for t in range(:_
118             input)})
119         feed_dict.update({expected_output[t]: Y[t] for t in
120             (expected_output)}))
121
122     c = np.concatenate(([np.zeros_like(Y[0])], Y[:-1]), axis=0)
123     feed_dict.update({decode_input[t]: c[t] for t in range(1, len(c))})
124
125     output_lossv, reg_lossv, loss_t = sess.run([output_loss,
126         reg_loss, loss], feed_dict)
127
128     print("-----")
129     print(output_lossv, reg_lossv)
130     return loss_t
131
132
133 # 训练
134 train_losses = []
135 test_losses = []
136
137 sess.run(tf.global_variables_initializer())
138 for t in range(nb_iters + 1):
139     train_loss = train_batch(batch_size)
140     train_losses.append(train_loss)
141     if t % 50 == 0:
142         test_loss = test_batch(batch_size)
143         test_losses.append(test_loss)
144         print("Step {}/{}, train loss: {}, \tTEST loss: {}".
145             (t, nb_iters, train_loss, test_loss))
146
147 print("Fin. train loss: {}, \tTEST loss: {}".format(
148     train_losses[-1], test_losses[-1]))
149
150
151 # 输出loss图例
152 plt.figure(figsize=(12, 6))
153 plt.plot(np.array(range(0, len(test_losses))) /
154         float(len(test_losses) - 1) * (len(train_losses) - 1),
155         test_losses)

```

*****ebook converter DEMO Watermarks*****

```
141     np.log(test_losses),label="Test loss")
142
143 plt.plot(np.log(train_losses),label="Train loss")
144 plt.title("Training errors over time (on a logarithmic scale")
145 plt.xlabel('Iteration')
146 plt.ylabel('log(Loss)')
147 plt.legend(loc='best')
148 plt.show()
```

对于decode_input的输入要重点说明一下，将其第一个序列的输入变为0，作为起始输入的标记，接上后续的Y数据（未来序列）作为解码器部分的Decoder来输入。由于第一个序列被占用了，保证总长度不变的情况下，Y的最后一个序列没有作为Decoder的输入。但是输出时会有关于未来序列预测的全部序列值，并在计算loss时与真实值Y进行平方差。

最终将loss值通过plot打印出来，生成结果如下，loss结果曲线如图9-28所示。

```
training examples :
(15, 3, 1)
(seq_length, batch_size, output_dim)
-----
7.66522 113.373
Step 0/100, train loss: 8.341724395751953,      TEST loss:8.0051
-----
1.11881 99.788
Step 50/100, train loss:2.0858113765716553,      TEST loss:1.418
-----
0.618375 83.6507
Step 100/100, train loss:0.9577032327651978,      TEST loss:0.869327306
Fin. train loss:0.9577032327651978,      TEST loss:0.869327306
```

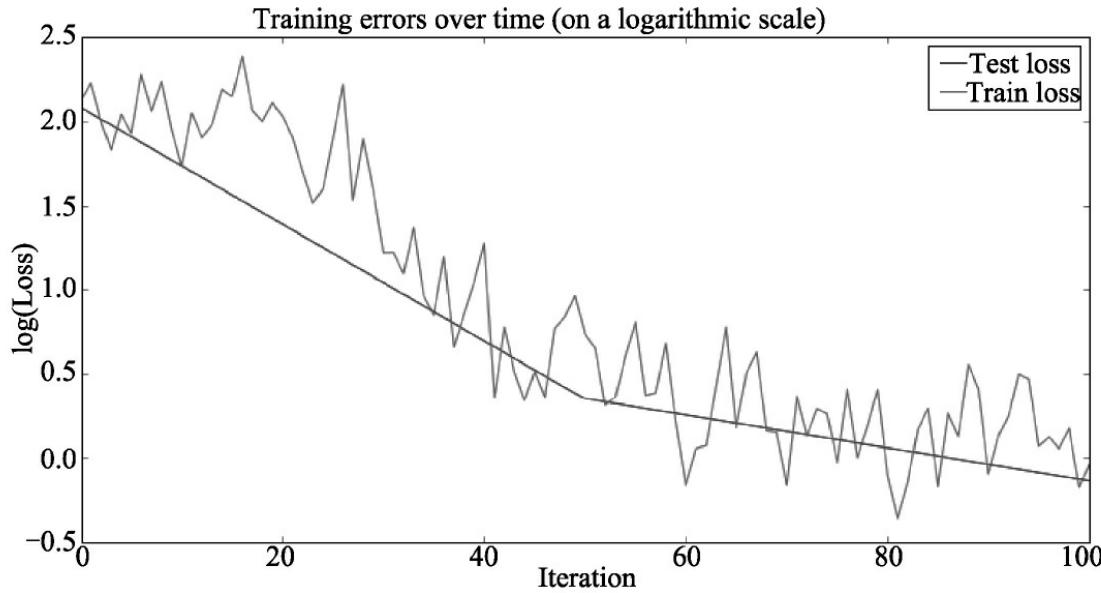


图9-28 loss结果曲线

5. 准备可视化数据

一般情况下，将整个输出值进行显示即可。但这里考虑到要配合使用时的演示，因此我们需要模型来预测未来序列，即没有decode_input的输入。前面说了，这种情况可以将decode_input全设为0，但其识别效果不客观。为了模型可用，可以将预测值范围稍加改变，只预测之后一次时间序列的值。例如，知道前面的所有序列，预测当天股票的收盘价格、开盘价格等。这也是非常实际的应用。

于是在可视化部分，取时间序列2倍的样本，前一倍用于输入模型，会产生最后一天的预测值，同时也将后一倍的数据显示出来，用于比对每个序列的预测值。

代码9-30 基本Seq2Seq（续）

```
149 # 测试
150 nb_predictions = 4
151 print("visualize {} predictions data:".format(nb_predictions))
152
153 preout = []
154 X, Y = generate_data(isTrain=False, batch_size=nb_predictions)
155 print(np.shape(X), np.shape(Y))
156 for tt in range(seq_length):
157     feed_dict = {encoder_input[t]: X[t+tt] for t in range(tt, seq_length)}
158     feed_dict.update({expected_output[t]: Y[t+tt] for t in range(tt, seq_length)})
159     c = np.concatenate(([np.zeros_like(Y[0])], Y[tt:seq_length]), axis=0) #从前15个序列的最后一个开始预测
160
161     feed_dict.update({decode_input[t]: c[t] for t in range(tt, seq_length)})
162     outputs = np.array(sess.run([reshaped_outputs], feed_dict))
163     preout.append(outputs[-1])
164
165 print(np.shape(preout)) #将每个未知预测值收集起来准备显示出来
166 preout = np.reshape(preout, [seq_length, nb_predictions, output_size])
```

前15次时间序列用于输入，后15次循环来使用模型预测，每次都将输出的最后一个时间序列收集起来，最终得到15个时间序列批次的预测结果preout。

6. 画图显示数据

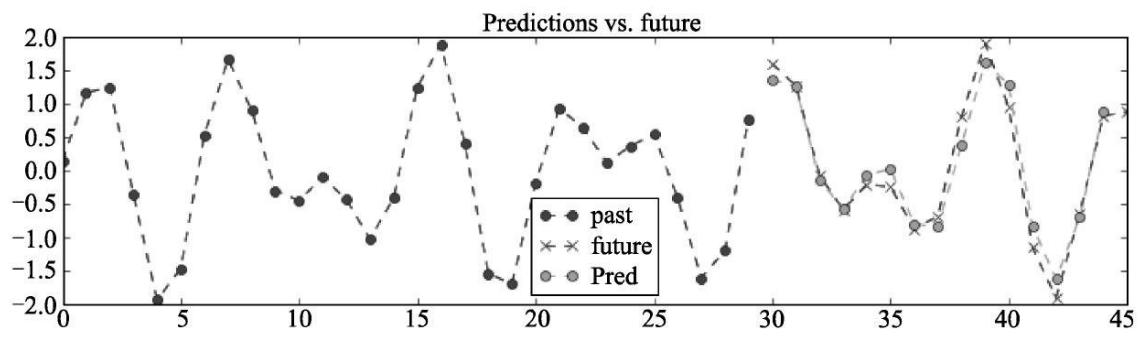
将批次设为4，随机取4个序列片段，每个片段的15个序列预测以图像形式显示出来。

代码9-30 基本Seq2Seq（续）

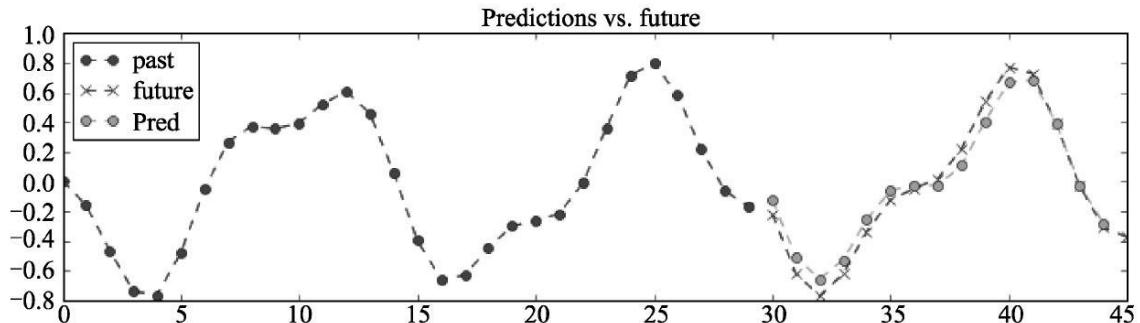
```
167 for j in range(nb_predictions):
*****ebook converter DEMO Watermarks*****
```

```
168     plt.figure(figsize=(12, 3))
169
170     for k in range(output_dim):
171         past = X[:, j, k]
172         expected = Y[seq_length-1:, j, k] #对应预测值的打印
173
174         pred = preout[:, j, k]
175
176         label1 = "past" if k == 0 else "_nolegend_"
177         label2 = "future" if k == 0 else "_nolegend_"
178         label3 = "Pred" if k == 0 else "_nolegend_"
179         plt.plot(range(len(past)), past, "o--b", label=label1)
180         plt.plot(range(len(past)), len(expected) + len(past),
181                  expected, "x--b", label=label2)
182         plt.plot(range(len(past)), len(pred) + len(past))
183                  pred, "o--y", label=label3)
184
185     plt.legend(loc='best')
186     plt.title("Predictions vs. future")
187     plt.show()
```

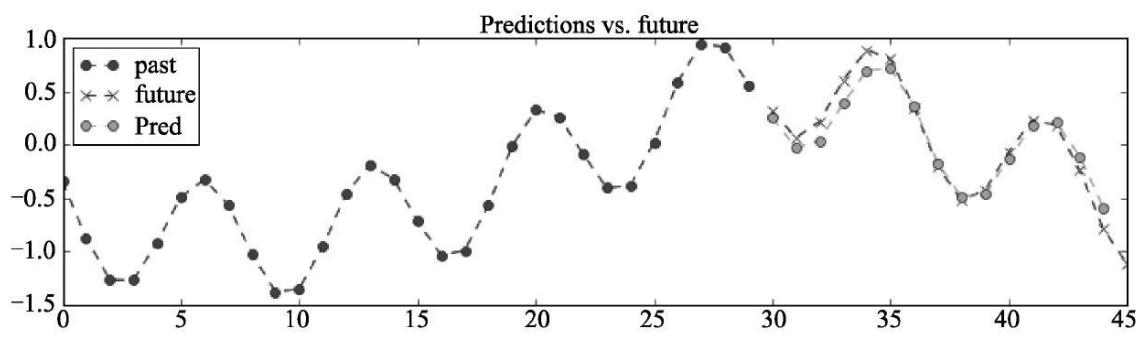
为了跟真实的序列值比较，这里将真实的序列值也从15个序列开始打印出来，index=14的值即为预测的第一个值。运行上面的代码，结果如图9-29所示。



a) 序列片段 1

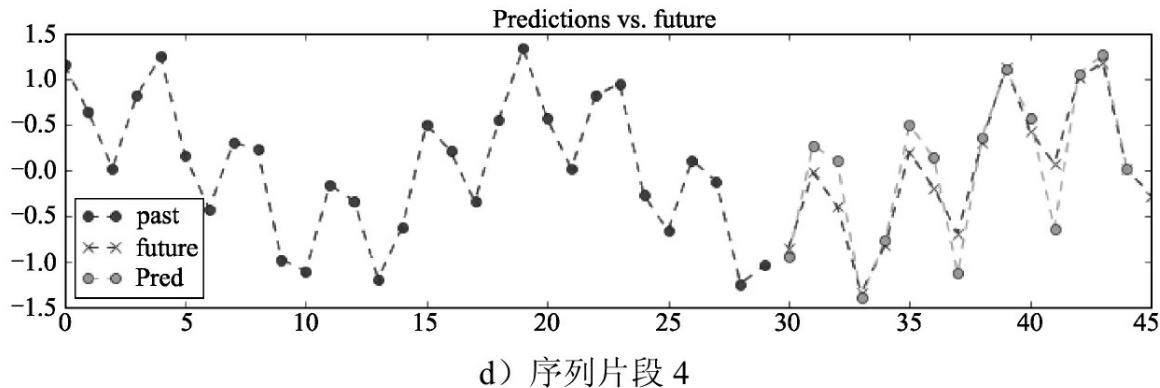


b) 序列片段 2



c) 序列片段 3

图9-29 基于Seg2Seg实例结果



d) 序列片段 4

图9-29 基于Seg2Seg实例结果（续）

可以看到，生成的预测数据与真实数据相差并不大。



注意： 这里使用了feed_dict的update方法来处理复杂的feed_dict的情况，通过Update可以在原有的feed_dict中加入新的feed数据，将一行语句变为多行输入。

9.8.4 实例73：预测当天的股票价格

既然前面我们用预测股票来打比方，那么这里就演示一个预测股票的例子。直接修改实例72中的数据源即可。

实例描述

使用Seq2Seq模式对某个股票数据的训练学习，拟合特征，从而达到可以预测第二天股票价格的效果。

1. 准备数据

需要准备一个股票的数据，本例中的格式是CSV，也可使用本书的配套例子中的数据“600000.csv”（笔者只是随意爬取了A股中的第一个股票，没有其他特殊意义），本书配套代码中提供了一个爬虫代码文件，见代码“9-31”

STOCKDATA.py”文件。

2. 导入股票数据

直接在“9-30：基本seq2seq.py”文件基础上修改代码，添加载入股票函数loadstock，里面使用了pandas，所以要将该库导入进去。实例中将close收盘价格载入内存用于做样本生成。当然读者也可以自行修改字段，可以将开盘价、最高价格和最低价格等都载入内存作为样本数据，只需将对应的列名放入predictor_names数组中即可。

代码9-32 seq2seqstock

```
01 import pandas as pd
02 pd.options.mode.chained_assignment = None # default='warn'
03 def loadstock(window_size):
04     names = ['date',
05             'code',
06             'name',
07             'Close',
08             'top_price',
09             'low_price',
10             'opening_price',
11             'bef_price',
12             'floor_price',
13             'floor',
14             'exchange',
15             'Volume',
16             'amount',
17             '总市值',
18             '流通市值']
19     data = pd.read_csv('600000.csv', names=names, header=
= "gbk")
20
21     predictor_names = ["Close"]
22     training_features = np.asarray(data[predictor_names],
"float32")
```

*****ebook converter DEMO Watermarks*****

```
23     kept_values = training_features[1000:]
24
25     X = []
26     Y = []
27     for i in range(len(kept_values) - window_size * 2):
# x为前window_size个序列, y为后window_size一个序列
28         X.append(kept_values[i:i + window_size])
29         Y.append(kept_values[i + window_size:i + window_size + 1])
30
31     X = np.reshape(X, [-1, window_size, len(predictor_names)])
32     Y = np.reshape(Y, [-1, window_size, len(predictor_names)])
33     print(np.shape(X))
34
35     return X, Y
```

3. 生成样本

直接修改代码中生成样本的函数
`generate_data`, 和其对应的内部调用的
`do_generate_x_y`函数, 代码如下。

代码9-32 seq2seqstock (续)

```
36 def generate_data(isTrain, batch_size):
37     # 用前40个样本来预测后40个样本
38
39     seq_length = 40
40     seq_length_test = 80
41
42     global Y_train
43     global X_train
44     global X_test
45     global Y_test
46     # 载入内存
47     if len(Y_train) == 0:
48         X, Y = loadstock(window_size=seq_length)
49
50     # Split 80-20:
51     X_train = X[:int(len(X) * 0.8)]
52     Y_train = Y[:int(len(Y) * 0.8)]
53
```

```
54     if len(Y_test) == 0:
55         X, Y = loadstock( window_size=seq_length_test)
56
57         # Split 80-20:
58         X_test = X[int(len(X) * 0.8):]
59         Y_test = Y[int(len(Y) * 0.8):]
60
61     if isTrain:
62         return do_generate_x_y(X_train, Y_train, batch_s:
63     else:
64         return do_generate_x_y(X_test, Y_test, batch_s:
65
66 def do_generate_x_y(X, Y, batch_size):
67     assert X.shape == Y.shape, (X.shape, Y.shape)
68     idxes = np.random.randint(X.shape[0], size=batch_size)
69     X_out = np.array(X[idxes]).transpose((1, 0, 2))
70     Y_out = np.array(Y[idxes]).transpose((1, 0, 2))
71     return X_out, Y_out
```

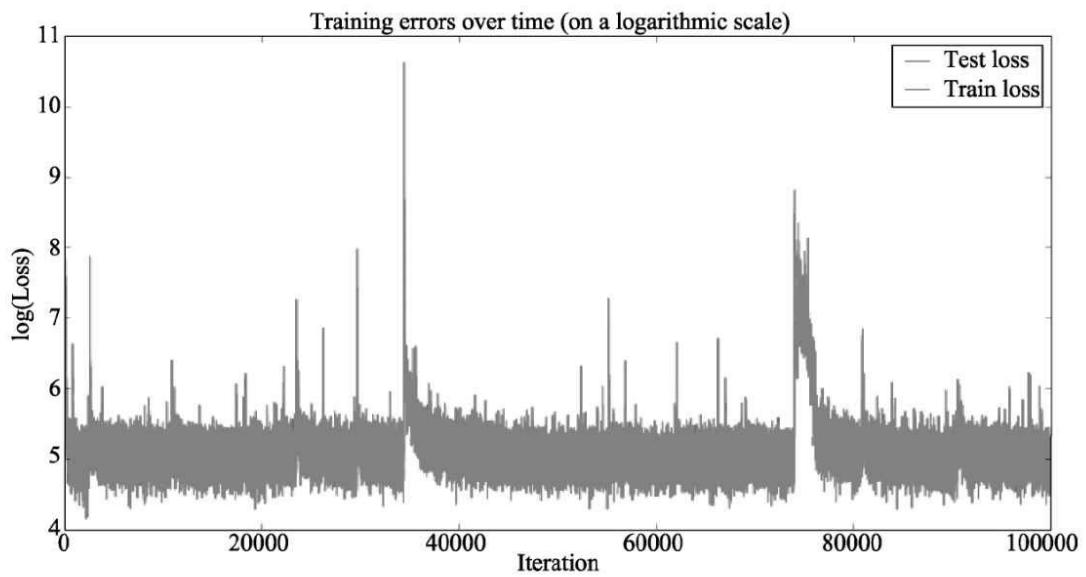
4. 运行程序查看效果

由于股票数据没有固定的规则而言，并且数据量又较大，所以加大batch到100，加大迭代次数到100000，代码片段如下。

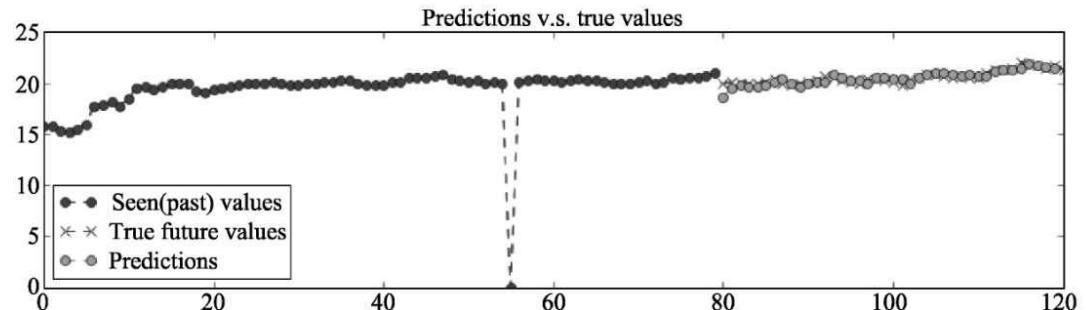
代码9-32 seq2seqstock（续）

```
72 ....
73 seq_length = sample_now.shape[0]
74 batch_size = 100
75
76 output_dim = input_dim = sample_now.shape[-1]
77 hidden_dim = 12
78 layers_num = 2
79
80 # 学习率
81 learning_rate = 0.04
82 nb_iters = 100000
83 lambda_l2_reg = 0.003 # L2 regularization of weights - a\y
84 .....
```

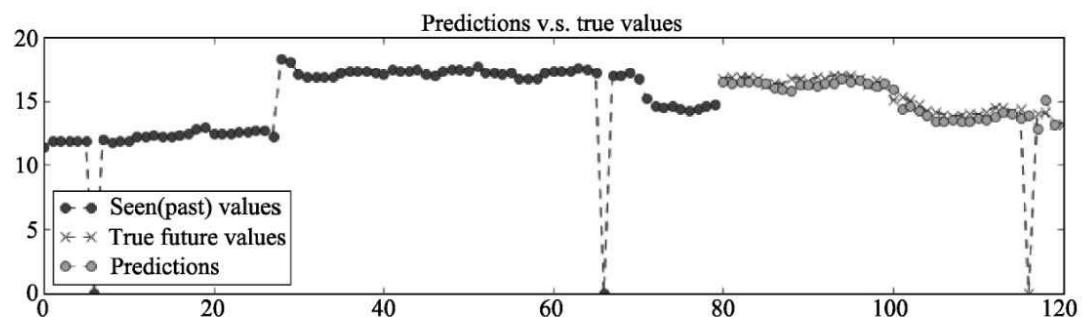
其他地方均不用变，直接运行代码即可，输出如图9-30所示。



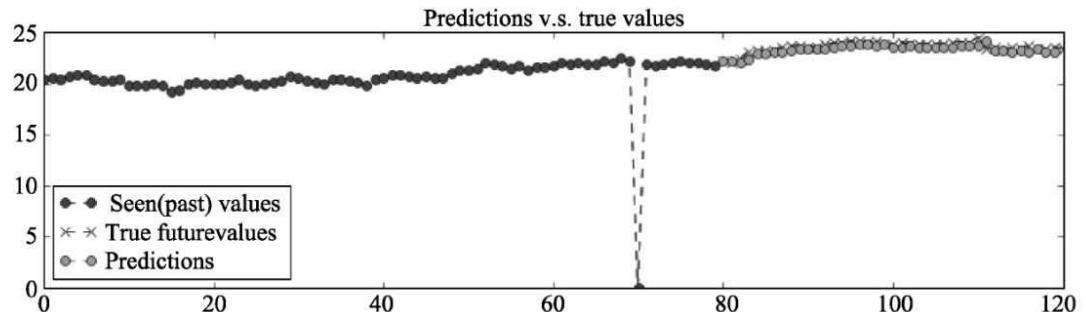
a) loss 值



b) 序列 1



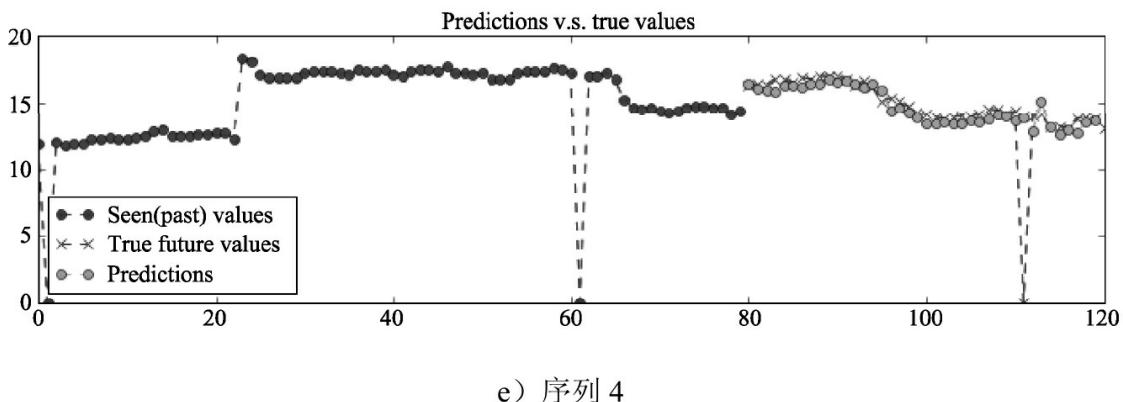
c) 序列 2



d) 序列 3

*****ebook converter DEMO Watermarks*****

图9-30 股票示例结果



e) 序列 4

图9-30 股票示例结果（续）

可见损失值还是比较高的，中间有两次还出现了飙升，由于没有对数据进行清洗和修正，所以会看到序列中有突然变为0的情况，这是由于或许当天是停牌或者数据缺失等情况照成的。恰好我们可以把它当成噪声数据来泛化网络。图9-30中序列80~120之间的点（即图中灰色的点）代表预测的结果，X代表真实的结果，可以看到，虽然不是很精确，但是总体还是与真实数据很接近的。在真实使用场景中，可以修改显示部分的测试代码，不用随机取样本数据，而是把最后一段时间序列拿出来并放到模型里，输出的最后一个预测值即是当天的收盘价预测。



提示：股市有风险，用机器炒股也要谨慎。这里演示的只是一个模型，其精确度和拟合度还有待提高，并不能当作炒股指导工具。

通过两个例子的练习，希望读者可以掌握Seq2Seq的基本使用。其实，这种简单的Seq2Seq框架在实际应用中对超长序列数据的学习效果并不是很好。这是因为无论输入端有多大变化，Encoder给出的都是一个固定维数的向量，存在信息损失，所以输入的序列越长，Encoder的输出丢失的原始信息就越多，传入Decoder后，很难在Decoder中有太多的特征表现。

对于这个问题，引出了下面的基于注意力的Seq2Seq。

9.8.5 基于注意力的Seq2Seq

本节就来介绍一下这个基于注意力的Seq2Seq网络。

1. attention_seq2seq介绍

注意力机制，即在生成每个词时，对不同的输入词给予不同的关注权重。如图9-31所示，右侧序列是输入序列，上方序列是输出序列。在注意力机制下，对于一个输出网络会自动学习与其对应的输入关系的权重。如How下面一列。

How old are you ?

5	5	5	80	5
80	5	10	5	0
10	80	10	0	0
5	10	65	15	5
0	0	10	0	90

你
多
大
了
?

图9-31 注意力表现

在训练过程中，模型会通过注意力机制把某个输出对应的所有输入列出来，学习其关系并更新到权重上。如图9-31所示，“you”下面那一列（80、5、0、15、0），就是模型在生成you这个词时的概率分布，对应列的表格中值最大的地方对应的是输入的“你”（对应图中第1行第4列，值为80），说明模型在生成you这个词时最为关注的输入词是“你”。这样在预测时，该机制就会根据输入及其权重反向推出更有可能的预测值了。

注意力机制是在原有Seq2Seq中的Encoder与Decoder框架中修改而来，具体结构如图9-32所示。

修改后的模型特点是序列中每个时刻Encoder

生成的 c , 都将要参与Decoder中解码的各个时刻, 而不是只参与初始时刻。当然对于生成的结果节点 c , 参与到Decoder的每个序列运算都会经过权重 w , 那么这个 w 就可以以loss的方式通过优化器来调节了, 最终会逐渐逼近与它紧密的那个词, 这就是注意力的原理。添加入了Attention注意力分配机制后, 使得Decoder在生成新的Target Sequence时, 能得到之前Encoder编码阶段每个字符的隐藏层的信息向量Hidden State, 使得新生成序列的准确度提高。

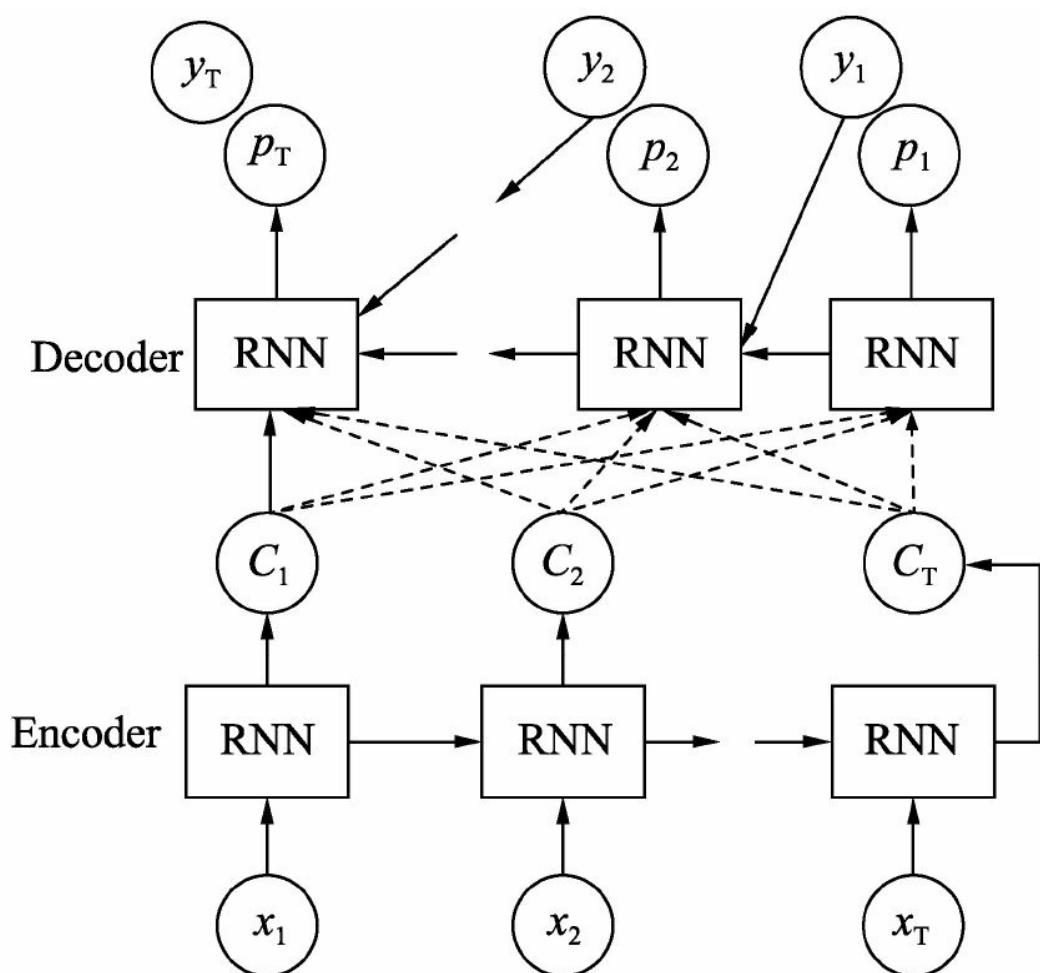


图9-32 Seq2Seq attention

2. TensorFlow中的attention_seq2seq

在TensorFlow中也有关于带有注意力机制的Seq2Seq定义，封装后的Seq2Seq与前面basic_rnn_seq2seq差不多，具体函数如下：

```
tf.contrib.legacy_seq2seq.embedding_attention_seq2seq (encoder_inputs,
                                                       decoder_inputs,
                                                       cell,
                                                       num_encoder_symbols,
                                                       num_decoder_symbols,
                                                       embedding_size,
                                                       num_heads=1,
                                                       output_projection=None,
                                                       feed_previous=False,
                                                       dtype=None,
                                                       scope=None,
                                                       initial_state_attention=False)
```

参数说明如下。

- encoder_inputs：一个形状为[batch_size]的list。
- decoder_inputs：同encoder_inputs。
- cell：定义的cell网络。
 - num_encoder_symbols：输入数据对应的词总个数。
 - num_decoder_symbols：输出数据对应的词的总个数。

- embedding_size: 每个输入对应的词向量编码大小。
- num_heads: 从注意力状态里读取的个数。
- output_projection: 对输出结果是否进行全连接的维度转化，如果需要转化，则传入全连接对应的w和b。
- feed_previous: 为True时，表明只有第一个Decoder输入以Go开始，其他都使用前面的状态。如果为False时，每个Decoder的输入都会以Go开始。Go为自己定义模型时定义的一个起始符，一般用0或1来指定。

3. Seq2Seq中桶（bucket）的实现机制

在Seq2Seq模型中，由于输入、输出都是可变长的，这就给计算带来了很大的效率影响。在TensorFlow中使用了一个“桶”（bucket）的观念来权衡这个问题，思想就是初始化几个bucket，对数据预处理，按照每个序列的长短，将其放到不同的bucket中，小于bucket size部分统一补0来完成对齐的工作，之后就可以进行不同bucket的批处理计算了。

由于该问题与Seq2Seq模型关联比较紧密，在TensorFlow中就将其封装成整体的框架模式，开发者只需要将输入、输出、网络模型传入函数*****ebook converter DEMO Watermarks*****

中，其他的都交给函数自己来处理，大大简化了开发过程，其定义如下：

```
model_with_buckets(encoder_inputs,  
                    decoder_inputs,  
                    targets,  
                    weights,  
                    buckets,  
                    seq2seq,  
                    softmax_loss_function=None,  
                    per_example_loss=False,  
                    name=None):
```

参数说明如下。

- encoder_inputs：一个形状为[batch_size]的list。
- decoder_inputs：同encoder_inputs，作为解码器部分的输入。
- targets：最终输出结果的label。
- weights：传入的权重值，必须与decoder_inputs的size相同。
- buckets：传入的桶，描述为[(xx, xx) ,
(xx, xx) …]每一对有两个数，第一个数为输入的size，第二个数为输出的size。
- seq2seq：带有Seq2Seq结构的网络，以函数

名的方式传入。在Seq2Seq里可以载入定义的cell网络。

- softmax_loss_function：是否使用自己指定的loss函数。
- per_example_loss：是否对每个样本求loss。

这里面有疑问的部分就是weights，为什么会多了个weights？它是做什么的呢？跟进代码里可以看到，它会调用sequence_loss_by_example函数，在sequence_loss_by_example函数中weights是用来做loss计算的。具体见

tensorflow\contrib\legacy_seq2seq\python\ops\seq2seq.py文件中第1048行函数sequence_loss_by_example的实现，代码如下：

```
.....
with ops.name_scope(name, "sequence_loss_by_example",
                     logits + targets + weights):
    log_perp_list = []
    for logit, target, weight in zip(logits, targets, weights):
        if softmax_loss_function is None:
            # TODO(irving,ebrevdo):为了符合调用sequence_loss_by_example
            target = array_ops.reshape(target, [-1])
            crossent = nn_ops.sparse_softmax_cross_entropy_with_
                       labels=target, logits=logit)
        else:
            crossent = softmax_loss_function(labels=target, logits=logit)
        log_perp_list.append(crossent * weight)
    log_perps = math_ops.add_n(log_perp_list)
    if average_across_timesteps:
        total_size = math_ops.add_n(weights)
        total_size += 1e-12  # 避免除数为0
        log_perps /= total_size
    return log_perps
```

*****ebook converter DEMO Watermarks*****

可见在求每个样本loss时对softmax_loss的结果乘了weight，同时又将乘完weight后的总和结果除以weights的总和（ $\log_{perps} / total_size$ ）。这种做法就是叫做基于权重的交叉熵计算（weighted cross_entropy loss）（具体细节不再展开，读者简单了解即可）。

9.8.6 实例74：基于Seq2Seq注意力模型实现中英文机器翻译

本例中将使用前面介绍的函数，将它们组合在一起实现一个具有机器翻译功能的例。该例中共涉及4个代码文件，各文件说明如下。

- 文件“9-33 datautil.py”：样本预处理文件。
- 文件“9-34 seq2seq_model.py”：模型文件，该文件是在GitHub上TensorFlow的例子基础上修改而来。
- 文件“9-35 train.py”：模型的训练文件。
- 文件“9-36 test.py”：模型的测试使用文件。

本例同样也是先从样本入手，然后搭建模型、训练、测试，具体步骤如下。

实例描述

准备一部分中英对照的翻译语料，使用Seq2Seq模式对其进行学习，拟合特征，从而实现机器翻译。

1. 样本准备

对于样本的准备，本书配套资源中提供了一个“中英文平行语料库.rar”文件，如果读者需要更多、更全的样本，需要自己准备。解压后有两个文件，一个是英文文件，一个是对应的中文文件。

如果想与本书同步路径，可以将英文文件放到代码同级文件夹fanyichina\yuliao\from下；中文文件放在代码同级文件夹fanyichina\yuliao\to下。

2. 生成中、英文字典

编写代码，分别载入两个文件，并生成正反向字典。



注意： 样本的编码是UTF-8，如果读者使用自己定义的样本，不是UTF-8编码，则在读

*****ebook converter DEMO Watermarks*****

取文件时会报错，需要改成正确的编码。如果是Windows编辑的样本，编码为GB2312。

代码9-33 datautil

```
01 data_dir = "fanyichina/"
02 raw_data_dir = "fanyichina/yuliao/from"
03 raw_data_dir_to = "fanyichina/yuliao/to"
04 vocabulary_fileen ="dicten.txt"
05 vocabulary_filech = "dictch.txt"
06
07 plot_histograms = plot_scatter =True
08 vocab_size =40000
09
10 max_num_lines =1
11 max_target_size = 200
12 max_source_size = 200
13
14 def main():
15     vocabulary_filenameen = os.path.join(data_dir, vocabulary_fileen)
16     vocabulary_filenamech = os.path.join(data_dir, vocabulary_filech)
17 ##########
18 # 创建英文字典
19     training_dataen, counten, dictionaryen, reverse_dicten:
20         textsszen =create_vocabulary(vocabulary_filenameen
21                                     , raw_
22                                     , size, Isch=False, normalize_digits = True)
23     print("training_data",len(training_dataen))
24     print("dictionary",len(dictionaryen))
25 ##########
26 #创建中文字典
27     training_datach, countch, dictionarych, reverse_dictch:
28         textsszch =create_vocabulary(vocabulary_filenamech
29                                     , raw_
30                                     , size, Isch=True, normalize_digits = True)
31     print("training_datach",len(training_datach))
32     print("dictionarych",len(dictionarych))
```

执行完上面的代码后，会在当前目录下的fanyichina文件夹里找到dicten.txt与dictch.txt两个字典文件。

*****ebook converter DEMO Watermarks*****

其中所调用的部分代码定义如下，严格来讲本例中生成的应该是词点，因为在中文处理中用了jieba分词库将文字分开了，是以词为单位存储对应索引的。

代码9-33 datautil（续）

```
29 import jieba
30 jieba.load_userdict("myjiebadict.txt")
31
32 def fenci(training_data):
33     seg_list = jieba.cut(training_data)          # 默认是精
34     training_ci = " ".join(seg_list)
35     training_ci = training_ci.split()
36     return training_ci
37
38 import collections
39 #系统字符，创建字典时需要加入
40 _PAD = "_PAD"
41 _GO = "_GO"
42 _EOS = "_EOS"
43 _UNK = "_UNK"
44
45 PAD_ID = 0
46 GO_ID = 1
47 EOS_ID = 2
48 UNK_ID = 3
49
50 #文字字符替换，不属于系统字符
51 _NUM = "_NUM"
52 #Isch=true 中文,
53 #false 英文
54 #创建词典，max_vocabulary_size=500代表字典中有500个词
55 def create_vocabulary(vocabulary_file, raw_data_dir, max_
size, Isch=True, normalize_digits=True):
56     texts, textssz = get_ch_path_text(raw_data_dir, Isch, nc
digits)
57     print( texts[0],len(texts))
58     print("行数",len(textssz),textssz)
59 # 处理多行文本texts
60     all_words = []
61     for label in texts:
```

*****ebook converter DEMO Watermarks*****

```

61         print("词数",len(label))
62         all_words += [word for word in label]
63         print("词数",len(all_words))
64
65     training_label, count, dictionary, reverse_dictionary
66     dataset(all_words,max_vocabulary_size)
67     print("reverse_dictionary",reverse_dictionary,len(rev
68     dictionary))
69     if not gfile.Exists(vocabulary_file):
70         print("Creating vocabulary %s from data %s" % (vo
71         data_dir))
72         if len(reverse_dictionary) > max_vocabulary_size:
73             reverse_dictionary = reverse_dictionary[:max_
74             size]
75         with gfile.GFile(vocabulary_file, mode="w") as vo
76             for w in reverse_dictionary:
77                 print(reverse_dictionary[w])
78                 vocab_file.write(reverse_dictionary[w] + "
79             else:
80                 print("already have vocabulary!  do nothing !!!!!
81                 !!!!!!!!!!!!!")
82             return training_label, count, dictionary, reverse_dic
83             textssz
84
85 def build_dataset(words, n_words):
86     """Process raw inputs into a dataset."""
87     count = [[_PAD, -1], [_GO, -1], [_EOS, -1], [_UNK, -1]]
88     count.extend(collections.Counter(words).most_common(n_w
89     dictionary = dict()
90     for word, _ in count:
91         dictionary[word] = len(dictionary)
92     data = list()
93     unk_count = 0
94     for word in words:
95         if word in dictionary:
96             index = dictionary[word]
97         else:
98             index = 0 # dictionary['UNK']
99             unk_count += 1
100            data.append(index)
101            count[0][1] = unk_count
102            reversed_dictionary = dict(zip(dictionary.values(), di
103                keys()))
104            return data, count, dictionary, reversed_dictionary

```

在字典中添加额外的字符标记PAD、_GO、

*****ebook converter DEMO Watermarks*****

`_EOS`、`_UNK`是为了在训练模型时起到辅助标记的作用。

- `PAD`用于在桶机制中为了对齐填充占位。
- `_GO`是解码输入时的开头标志位。
- `_EOS`是用来标记输出结果的结尾。
- `_UNK`用来代替处理样本时出现字典中没有的字符。

另外还有`_NUM`，用来代替文件中的数字（`_NUM`是根据处理的内容可选项，如果内容与数字高度相关，就不能用`NUM`来代替）。

在jieba的分词库中，附加一个字典文件`myjiebadict.txt`，以免自定义的字符标记被分开。`myjiebadict.txt`里的内容如下：

```
_NUM nz
_PAD nz
_GO nz
EOS nz
_UNK nz
```

每一行有两项，用空格分开，第一项为指定的字符，第二项`nz`代表不能被分开的意思。

3. 将数据转成索引格式

原始的中英文是无法让机器认知的，所以要根据字典中对应词的索引对原始文件进行相应的转化，方便读取。在本地建立两个文件夹fanyichina\fromids和fanyichina\toids，用于存放生成的ids文件。在main函数中编写以下代码，先通过initialize_vocabulary将前面生成的字典读入内存中，然后使用textdir_to_idsdir函数将文本转成ids文件。

textdir_to_idsdir函数中最后的两个参数说明如下。

- normalize_digits：代表是否将数字替换掉。
- Isch：表示是否是按中文方式处理。

中文方式会在处理过程中对读入的文本进行一次jieba分词。

代码9-33 datautil（续）

```
98     def main():
99         .....
100         vocaben, rev_vocaben = initialize_vocabulary(vocab
filenameen)
101         vocabch, rev_vocabch = initialize_vocabulary(vocab
filenamech)
102
103         print(len(rev_vocaben))
104         textdir_to_idsdir(raw_data_dir, data_dir+"fromids",
normalize_digits=True, Isch=False)
105         textdir_to_idsdir(raw_data_dir_to, data_dir+"toids",
normalize_digits=True, Isch=True)
```

所使用的函数定义如下：

代码9-33 datautil (续)

```
106 def initialize_vocabulary(vocabulary_path):  
107     if gfile.Exists(vocabulary_path):  
108         rev_vocab = []  
109         with gfile.GFile(vocabulary_path, mode="r") as f:  
110             rev_vocab.extend(f.readlines())  
111         rev_vocab = [line.strip() for line in rev_vocab]  
112         vocab = dict([(x, y) for (y, x) in enumerate(rev_vocab)])  
113         return vocab, rev_vocab  
114     else:  
115         raise ValueError("Vocabulary file %s not found.", vocabulary_path)  
116 #将文件批量转成ids文件  
117 def textdir_to_idsdir(textdir, idsdir, vocab, normalize_digits=True, Isch=True):  
118     text_files, filenames = getRawFileList(textdir)  
119  
120     if len(text_files)== 0:  
121         raise ValueError("err: no files in ", raw_data_dir)  
122  
123     print(len(text_files),"files, one is",text_files[0])  
124  
125     for text_file,name in zip(text_files,filenames):  
126         print(text_file,idsdir+name)  
127         textfile_to_idsfile(text_file,idsdir+name,vocab,  
normalize_digits,Isch)
```

其他用到的底层函数代码如下：

代码9-33 datautil (续)

```
128 #获取文件列表  
129 def getRawFileList( path):  
130     files = []  
131     names = []  
132     for f in os.listdir(path):
```

*****ebook converter DEMO Watermarks*****

```

133         if not f.endswith("~") or not f == "":
134             files.append(os.path.join(path, f))
135             names.append(f)
136     return files,names
137 #读取分词后的中文词
138 def get_ch_lable(txt_file,Isch=True,normalize_digits=False):
139     labels= list()#""
140     labelssz = []
141     with open(txt_file, 'rb') as f:
142         for label in f:
143             linstr1 =label.decode('utf-8')
144             if normalize_digits :
145                 linstr1=re.sub('\d+','_NUM',linstr1)
146                 notoken = basic_tokenizer(linstr1 )
147                 if Isch:
148                     notoken = fenci(notoken)
149                 else:
150                     notoken = notoken.split()
151
152             labels.extend(notoken)
153             labelssz.append(len(labels))
154     return labels,labelssz
155
156 #获取文件中的文本
157 def get_ch_path_text(raw_data_dir,Isch=True,normalize_digits=False):
158     text_files,_ = getRawFileList(raw_data_dir)
159     labels = []
160
161     training_dataszs = list([0])
162
163     if len(text_files)== 0:
164         print("err: no files in ",raw_data_dir)
165         return labels
166     print(len(text_files),"files, one is",text_files[0])
167     shuffle(text_files)
168
169     for text_file in text_files:
170         training_data,training_datasz =get_ch_lable(text_file,Isch,normalize_digits)
171
172         training_ci = np.array(training_data)
173         training_ci = np.reshape(training_ci, [-1, 1])
174         labels.append(training_ci)
175
176         training_datasz =np.array( training_datasz)+trainig_datasz
177         training_dataszs.extend(list(training_datasz))
178         print("here",training_dataszs)
179     return labels,training_dataszs
180

```

*****ebook converter DEMO Watermarks*****

```

181 def basic_tokenizer(sentence):
182     _WORD_SPLIT = "[.,!?'':;)(]"
183     _CHWORD_SPLIT = '、。।ঁ।।'
184     str1 = ""
185     for i in re.split(_CHWORD_SPLIT, sentence):
186         str1 = str1 +i
187     str2 = ""
188     for i in re.split(_WORD_SPLIT , str1):
189         str2 = str2 +i
190     return str2
191 #将句子转成索引ids
192 def sentence_to_ids(sentence, vocabulary,
193                      normalize_digits=True, Isch=True):
194
195     if normalize_digits :
196         sentence=re.sub('\'\d+',_NUM,sentence)
197     notoken = basic_tokenizer(sentence )
198     if Isch:
199         notoken = fenci(notoken)
200     else:
201         notoken = notoken.split()
202
203     idsdata = [vocabulary.get( w, UNK_ID) for w in notoken]
204
205     return idsdata
206
207 #将文件中的内容转成ids, 不是Windows下的文件要使用utf8编码格式
208 def textfile_to_idsfile(data_file_name, target_file_name,
209                         normalize_digits=True, Isch=True):
210
211     if not gfile.Exists(target_file_name):
212         print("Tokenizing data in %s" % data_file_name)
213         with gfile.GFile(data_file_name, mode="rb") as data_file:
214             with gfile.GFile(target_file_name, mode="w") as ids_file:
215                 counter = 0
216                 for line in data_file:
217                     counter += 1
218                     if counter % 100000 == 0:
219                         print(" tokenizing line %d" % counter)
220                     token_ids = sentence_to_ids(line.decode('utf8',
221                                     normalize_digits, Isch))
222                     ids_file.write(" ".join([str(tok) for tok in token_ids])
223                                     +"\n")
223 def ids2texts( indices,rev_vocab):
224     texts = []
225     for index in indices:
226         texts.append(rev_vocab[index])

```

*****ebook converter DEMO Watermarks*****

```
227     return texts
```

运行上述代码后，可以在本地路径fanyichina\fromids、fanyichina\toids文件夹下面找到同名的txt文件，打开后能够看到里面全是索引值。

4. 对样本文件进行分析图示

为了使bucket的设置机制较合理，我们把样本的数据用图示方式显示出来，直观地看一下每个样本的各个行长度分布情况，在main函数中接着添加以下代码：

代码9-33 datautil（续）

```
228 def main():
229     .....
230 #分析样本分布
231     filesfrom,_=getRawFileList(data_dir+"fromids/")
232     filesto,_=getRawFileList(data_dir+"toids/")
233     source_train_file_path = filesfrom[0]
234     target_train_file_path= filesto[0]
235     analysisfile(source_train_file_path,target_train_file_path)
236
237 if __name__=="__main__":
238     main()
```

最后两行为启动main函数。analysisfile为文件的分析函数，实现如下：

代码9-33 datautil（续）

*****ebook converter DEMO Watermarks*****

```
239 def analysisfile(source_file,target_file):
240 #分析文本
241     source_lengths = []
242     target_lengths = []
243
244     with gfile.GFile(source_file, mode="r") as s_file:
245         with gfile.GFile(target_file, mode="r") as t_file:
246             source= s_file.readline()
247             target = t_file.readline()
248             counter = 0
249
250             while source and target:
251                 counter += 1
252                 if counter % 100000 == 0:
253                     print(" reading data line %d" % counter)
254                     sys.stdout.flush()
255                 num_source_ids = len(source.split())
256                 source_lengths.append(num_source_ids)
257                 num_target_ids = len(target.split()) + 1
258                 target_lengths.append(num_target_ids)
259                 source, target = s_file.readline(), t_file.readline()
260             print(target_lengths,source_lengths)
261             if plot_histograms:
262                 plot_histo_lengths("target lengths", target_lengths)
263                 plot_histo_lengths("source_lengths", source_lengths)
264             if plot_scatter:
265                 plot_scatter_lengths("target vs source length",
266                                     "target length", source_lengths, target_lengths)
266 def plot_scatter_lengths(title, x_title, y_title, x_lengths,
267                           y_lengths):
268     plt.scatter(x_lengths, y_lengths)
269     plt.title(title)
270     plt.xlabel(x_title)
271     plt.ylabel(y_title)
272     plt.ylim(0, max(y_lengths))
273     plt.xlim(0,max(x_lengths))
274     plt.show()
275
275 def plot_histo_lengths(title, lengths):
276     mu = np.std(lengths)
277     sigma = np.mean(lengths)
278     x = np.array(lengths)
279     n, bins, patches = plt.hist(x, 50, facecolor='green', edgecolor='black')
280     y = mlab.normpdf(bins, mu, sigma)
281     plt.plot(bins, y, 'r--')
282     plt.title(title)
```

*****ebook converter DEMO Watermarks*****

```
283 plt.xlabel("Length")
284 plt.ylabel("Number of Sequences")
285 plt.xlim(0,max(lengths))
286 plt.show()
```

运行代码，得到如图9-33所示结果。

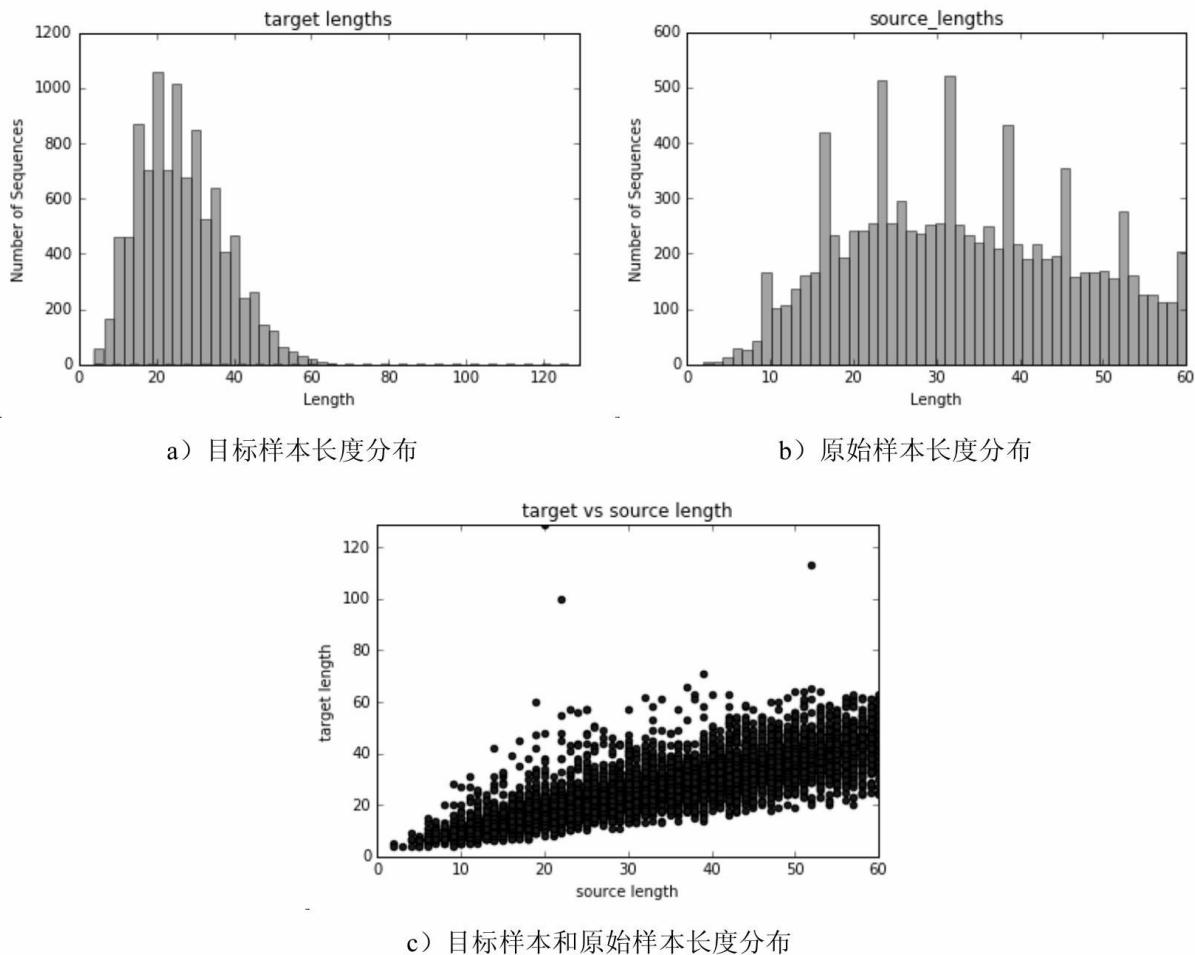


图9-33 样本分析

从图9-33可知，样本的长度都在60之间，可以将bucket分为4个区间，即`_buckets = [(20, 20), (40, 40), (50, 50), (60, 60)]`。由于输入和输出的长度差别不大，所以令它们的bucket相等。这部分还有更好的方法：可以使用聚类方式处理，然后自动化生成bucket，这样会更加方便，有兴趣的读者可以自己尝试一下。



说明： 网络模型初始化的部分，放到了后面讲解（见代码“9-34 seg2seg_model.py”文

*****ebook converter DEMO Watermarks*****

件），是想让读者先对整个流程有个大致了解。

5. 载入字典准备训练

预处理结束后，就可以开始编写训练代码了，在代码“9-35 train.py”文件里将刚才生成的字典载入，在getfanyiInfo中通过datautil.initialize_vocabulary将字典读入本地。同时引入库，设置初始参数，网络结构为两层，每层100个GRUcell组成的网络，在Seq2Seq模型中解码器与编码器同为相同的这种结构。

代码9-35 train

```
01 import os
02 import math
03 import sys
04 import time
05 import numpy as np
06 from six.moves import xrange
07 import tensorflow as tf
08 datautil = __import__("9-33 datautil")
09 seq2seq_model = __import__("9-34 seq2seq_model")
10 import datautil
11 import seq2seq_model
12
13 tf.reset_default_graph()
14
15 steps_per_checkpoint=200
16
17 max_train_data_size= 0 #(0代表输入数据的长度没有限制)
18
19 dropout = 0.9
20 grad_clip = 5.0
21 batch_size = 60
22
23 num_layers =2
24 learning_rate =0.5
```

```
25 lr_decay_factor =0.99
26
27 #设置翻译模型相关参数
28 hidden_size = 100
29 checkpoint_dir= "fanyichina/checkpoints/"
30 _buckets =[(20, 20), (40, 40), (50, 50), (60, 60)]
31 def getfanyiInfo():
32     vocaben, rev_vocaben=datautil.initialize_vocabulary(
33         (datautil.data_dir, datautil.vocabulary_fileen))
34     vocab_sizeen= len(vocaben)
35     print("vocab_size",vocab_sizeen)
36
37     vocabch, rev_vocabch=datautil.initialize_vocabulary(
38         (datautil.data_dir, datautil.vocabulary_filech))
39     vocab_sizech= len(vocabch)
40     print("vocab_sizech",vocab_sizech)
41
42     filesfrom,_=datautil.getRawFileList(datautil.data_dir)
43     filesto,_=datautil.getRawFileList(datautil.data_dir+'source_train_file_path = filesfrom[0]
44     target_train_file_path= filesto[0]
45     return vocab_sizeen,vocab_sizech,rev_vocaben,rev_vocabch,
46 train_file_path,target_train_file_path
47
48 def main():
49     vocab_sizeen,vocab_sizech,rev_vocaben,rev_vocabch,sol
file_path,target_train_file_path = getfanyiInfo()
```

通过getfanyiInfo函数得到中英词的数量、反向的中英字典、输入样本文件的路径以及目标样本的路径。

6. 启动session，创建模型并读取样本数据

代码9-35 train（续）

```
48     if not os.path.exists(checkpoint_dir):
49         os.mkdir(checkpoint_dir)
50     print ("checkpoint_dir is {0}".format(checkpoint_dir))
51
52     with tf.Session() as sess:
```

*****ebook converter DEMO Watermarks*****

```
53     model = createModel(sess, False, vocab_sizeen, vocal
54     print ("Using bucket sizes:")
55     print (_buckets)
56
57     source_test_file_path = source_train_file_path
58     target_test_file_path = target_train_file_path
59
60     print (source_train_file_path)
61     print (target_train_file_path)
62
63     train_set = readData(source_train_file_path, tar
file_path,max_train_data_size)
64     test_set = readData(source_test_file_path, target
path,max_train_data_size)
65
66     train_bucket_sizes = [len(train_set[b]) for b in
(_buckets))]
67     print( "bucket sizes = {0}".format(train_bucket_s
68     train_total_size = float(sum(train_bucket_sizes))
69
70     train_buckets_scale = [sum(train_bucket_sizes[:i]
total_size for i in xrange(len(train_bucket_sizes)))]
71     step_time, loss = 0.0, 0.0
72     current_step = 0
73     previous_losses = []
```

由于样本不足，这里直接在测试与训练中使用相同的样本，仅仅是为了演示。通过createModel创建模型，并查找检查点文件是否存在，如果存在，则将检测点载入。在createModel中通过调用Seq2SeqModel类生成模型，并指定模型中的具体初始参数。

代码9-35 train（续）

```
74 def createModel(session, forward_only, from_vocab_size,to_
size):
75     model = seq2seq_model.Seq2SeqModel(
76         from_vocab_size,#from
77         to_vocab_size,#to
```

*****ebook converter DEMO Watermarks*****

```
78     _buckets,
79     hidden_size,
80     num_layers,
81     dropout,
82     grad_clip,
83     batch_size,
84     learning_rate,
85     lr_decay_factor,
86     forward_only=forward_only,
87     dtype=tf.float32)
88
89     print("model is ok")
90
91     ckpt = tf.train.latest_checkpoint(checkpoint_dir)
92     if ckpt!=None:
93         model.saver.restore(session, ckpt)
94         print ("Reading model parameters from {0}".format(
95     else:
96         print ("Created model with fresh parameters.")
97         session.run(tf.global_variables_initializer())
98
99     return model
```

通过latest_checkpoint发现检查点文件。如果有检查点文件，就将其恢复到session中。

读取文件的函数定义如下：为了适用带有bucket机制的网络模型，按照bucket的大小序列读取数据，先按照bucket的个数定义好数据集data_set，然后在读取每一对输入、输出时，都会比较其适合哪个bucket，并将其放入对应的bucket中，最后返回data_set。

代码9-35 train（续）

```
100 def readData(source_path, target_path, max_size=None):
101     """
102     这个方法来自于tensorflow 中的translation 例子
```

*****ebook converter DEMO Watermarks*****

```
103 ''
104 data_set = [[] for _ in _buckets]
105 with tf.gfile.GFile(source_path, mode="r") as source_file:
106     with tf.gfile.GFile(target_path, mode="r") as target_file:
107         source, target = source_file.readline(), target_file.readline()
108         counter = 0
109         while source and target and (not max_size or counter <
110             max_size):
110             counter += 1
111             if counter % 100000 == 0:
112                 print("  reading data line %d" % counter)
113                 sys.stdout.flush()
114             source_ids = [int(x) for x in source.split()]
115             target_ids = [int(x) for x in target.split()]
116             target_ids.append(vocab_utils.EOS_ID)
117             for bucket_id, (source_size, target_size) in enumerate(
118                 _buckets):
119                 if len(source_ids) < source_size and len(target_ids) <
120                     target_size:
121                     data_set[bucket_id].append([source_ids,
122                         target_ids])
120             break
121             source, target = source_file.readline(), target_file.readline()
122         return data_set
```

对于输出的每一句话都会加上EOS_ID，这么做的目的是为了让网络学习到结束的标记，可以控制输出的长短。

7. 通过循环进行训练

在main函数中接着添加代码：通过循环来调用model.step进行迭代训练，每执行steps_per_checkpoint次，就保存检查点；测试结果，并将结果输出。

代码9-35 train（续）

*****ebook converter DEMO Watermarks*****

```
123 def main():
124     .....
125     while True:
126         # 根据数据样本的分布情况来选择bucket
127
128         random_number_01 = np.random.random_sample()
129         bucket_id = min([i for i in xrange(len(train_buckets_scale))
130                         if train_buckets_scale[i] > random_number_01])
131
132         # 开始训练
133         start_time = time.time()
134         encoder_inputs, decoder_inputs, target_weights =
135             get_batch(train_set, bucket_id)
136         _, step_loss, _ = model.step(sess, encoder_
137             inputs, target_weights, bucket_id, False)
138         step_time += (time.time() - start_time) / steps_per_checkpoint
139         loss += step_loss / steps_per_checkpoint
140         current_step += 1
141
142         # 保存检查点，测试数据
143         if current_step % steps_per_checkpoint == 0:
144             # Print statistics for the previous epoch
145             perplexity = math.exp(loss) if loss < 30
146             perplexity = 'inf' if loss >= 30 else perplexity
147             print ("global step %d learning rate %.2f perplexity %.
148                 "%.2f" % (model.global_step.eval(),
149                             rate.eval(), step_time, perplexity))
150             # 退化学习率
151             if len(previous_losses) > 2 and loss > previous_
152                 losses[-3:]):
153                 sess.run(model.learning_rate_decay_op)
154                 previous_losses.append(loss)
155             # 保存checkpoint
156             checkpoint_path = os.path.join(checkpoint_dir,
157                 "seq2seqtest.ckpt")
158             print(checkpoint_path)
159             model.saver.save(sess, checkpoint_path,
160                 model.global_step)
161             step_time, loss = 0.0, 0.0 # 初始化为0
162             # 输出test_set中 empty bucket的bucket_id
163             if len(test_set[bucket_id]) == 0:
164                 print(" eval: empty bucket %d"
165                     continue
166             encoder_inputs, decoder_inputs, target_weights =
167             model.get_batch(test_set, bucket_id)
```

*****ebook converter DEMO Watermarks*****

```
159
160             _, eval_loss, output_logits = model.s
161     encoder_inputs, decoder_inputs, target_weights, bucket
162             eval_ppx = math.exp(eval_loss) if ev
163             float('inf'))
164             print(" eval: bucket %d perplexity
165     inputstr = datautil.ids2texts(revers
166     in encoder_inputs] ,rev_vocaben)
167             print("输入",inputstr)
168             print("输出",datautil.ids2texts([en[
169     decoder_inputs] ,rev_vocabch))
170
171             outputs = [np.argmax(logit, axis=1)]
172             output_logits]
173
174             if datautil.EOS_ID in outputs:
175                 outputs = outputs[:outputs.index(
176                     EOS_ID)]
177                 print("结果",datautil.ids2texts(
178                     vocabch))
179
180             sys.stdout.flush()
181
182 if __name__ == '__main__':
183 main()
```

这里使用的是一个死循环，默认会一直训练下去。因为有检查点文件，所以可以不用关注迭代次数，通过输出测试的打印结果与loss值，可以看出模型的好坏。训练到一定程度后直接退出即可。

8. 网络模型Seq2SeqModel的初始化

这里为了先让读者对整体流程有个了解，所以将网络模型放在了最后单独介绍。这部分的代

码在“9-34 seq2seq_model.py”文件中，该代码为GitHub中的一个例子代码，我们在其上面做了修改，增加了dropout功能，在初始化函数中增加了dropout_keep_prob参数。

在原有代码中，由于指定了输出的 target_vocab_size，表明要求在模型结束后输出的应该是 target_vocab_size 其中的一类（one_hot），所以先定义了 output_projection 参数，里面由 w 和 b 构成，作为最后输出的权重。

代码9-34 seq2seq_model

```
01 """带有注意力机制的Sequence-to-sequence 模型."""
02
03 from __future__ import absolute_import
04 from __future__ import division
05 from __future__ import print_function
06
07 import random
08
09 import numpy as np
10 from six.moves import xrange # pylint: disable=redefined
11 import tensorflow as tf
12 datautil = __import__("9-33_datautil")
13 import datautil as data_utils
14
15 class Seq2SeqModel(object):
16     """带有注意力机制并且具有multiple buckets的Sequence-to-sequence
17     这个类实现了一个多层循环网络组成的编码器和一个具有注意力机制的解码器。
18     照论文:http://arxiv.org/abs/1412.7449 - 中所描述的机制实现。更多细节
19     论文内容
20     这个class 除了使用LSTM cells还可以使用GRU cells, 还使用了softmax 来
21     处理大词汇量的输出. 在论文http://arxiv.org/abs/1412.2007中
22     sampled softmax。在论文http://arxiv.org/abs/1409.0473里面
         这个模型的一个单层的使用双向RNN编码器的版本
22
```

```

23     """
24
25     def __init__(self,
26                 source_vocab_size,
27                 target_vocab_size,
28                 buckets,
29                 size,
30                 num_layers,
31                 dropout_keep_prob,
32                 max_gradient_norm,
33                 batch_size,
34                 learning_rate,
35                 learning_rate_decay_factor,
36                 use_lstm=False,
37                 num_samples=512,
38                 forward_only=False,
39                 dtype=tf.float32):
40         """创建模型
41
42         Args:
43             source_vocab_size: 原词汇的大小.
44             target_vocab_size: 目标词汇的大小.
45             buckets: 一个 (I, O) 的 list, I 代表输入的最大长度, O 代表输出的最大长度.
46             [(2, 4), (8, 16)].
47             size: 模型中每层的units个数.
48             num_layers: 模型的层数.
49             max_gradient_norm: 截断梯度的阀值.
50             batch_size: 训练中的批次数据大小;
51             learning_rate: 开始学习率.
52             learning_rate_decay_factor: 退化学习率的衰减参数.
53             use_lstm: 如果true, 使用 LSTM cells 替代GRU cells.
54             num_samples: sampled softmax的样本个数.
55             forward_only: 如果设置了, 模型只有正向传播.
56             dtype: internal variables的类型.
57         """
58         self.source_vocab_size = source_vocab_size
59         self.target_vocab_size = target_vocab_size
60         self.buckets = buckets
61         self.batch_size = batch_size
62         self.dropout_keep_prob_output = dropout_keep_prob
63         self.dropout_keep_prob_input = dropout_keep_prob
64         self.learning_rate = tf.Variable(
65             float(learning_rate), trainable=False, dtype=dtype)
66         self.learning_rate_decay_op = self.learning_rate.assign(
67             self.learning_rate * learning_rate_decay_factor)
68         self.global_step = tf.Variable(0, trainable=False)
69
70         # 如果使用 sampled softmax, 需要一个输出的映射.
71         output_projection = None

```

*****ebook converter DEMO Watermarks*****

```
72     softmax_loss_function = None
73     # 当采样数小于vocabulary size 时Sampled softmax 才有意义
74     if num_samples > 0 and num_samples < self.target_vocab_size:
75         w_t = tf.get_variable("proj_w", [self.target_vocab_size,
76                                     self._vocab_size], dtype=dtype)
77         w = tf.transpose(w_t)
78         b = tf.get_variable("proj_b", [self.target_vocab_size],
79                             dtype=dtype)
80         output_projection = (w, b)
```

lobal_step变量的作用是同步检查点文件对应的迭代步数。

9. 自定义损失函数

sampled_loss为自定义损失函数，计算在分类target_vocab_size里模型输出的logits与标签labels（seq2seq框架中的输出）之间的交叉熵，并将该函数指针赋值给softmax_loss_function。softmax_loss_function会在后面使用model_with_buckets时，作为参数传入。

代码9-34 seq2seq_model（续）

```
79     def sampled_loss(labels, logits):
80         labels = tf.reshape(labels, [-1, 1])
81         #需要使用 32bit的浮点数类型来计算sampled_softmax_loss
82         #的不稳定性
83         local_w_t = tf.cast(w_t, tf.float32)
84         local_b = tf.cast(b, tf.float32)
85         local_inputs = tf.cast(logits, tf.float32)
86         return tf.cast(
87             tf.nn.sampled_softmax_loss(
88                 weights=local_w_t,
89                 biases=local_b,
90                 labels=labels,
91                 inputs=local_inputs,
```

*****ebook converter DEMO Watermarks*****

```
91             num_sampled=num_samples,
92             num_classes=self.target_vocab_size),
93             dtype)
94 softmax_loss_function = sampled_loss
```

10. 定义Seq2Seq框架结构

seq2seq_f函数的作用是定义Seq2Seq框架结构，该函数也是为了使用model_with_buckets时，作为参数传入。前面介绍model_with_buckets函数时说该函数更像一个封装好的框架，原因就在于此。

读者也要适应这种方式：将损失函数、网络结构、buckets统统定义完，然后将它们作为参数放入model_with_buckets函数中，之后一切交给TensorFlow来实现即可。

代码9-34 seq2seq_model（续）

```
95     # 使用词嵌入量（embedding）作为输入
96     def seq2seq_f(encoder_inputs, decoder_inputs, do_decode):
97
98         with tf.variable_scope("GRU") as scope:
99             cell = tf.contrib.rnn.DropoutWrapper(
100                 tf.contrib.rnn.GRUCell(size),
101                 input_keep_prob=self.dropout_keep_prob_:
102                 output_keep_prob=self.dropout_keep_prob_
103             if num_layers > 1:
104                 cell = tf.contrib.rnn.MultiRNNCell([cell])
105
106             print("new a cell")
107             return tf.contrib.legacy_seq2seq.embedding_attention_
108                 encoder_inputs,
109                 decoder_inputs,
110                 cell,
```

*****ebook converter DEMO Watermarks*****

```
111     num_encoder_symbols=source_vocab_size,  
112     num_decoder_symbols=target_vocab_size,  
113     embedding_size=size,  
114     output_projection=output_projection,  
115     feed_previous=do_decode,  
116     dtype=dtype)
```

上面代码中，额外加了一个打印信息
print ("new a cell")，是为了测试seq2seq_model
函数是什么时被调用的，在实验中可以得出结
论。在构建网络模型时，会由model_with_buckets
函数来调用，而model_with_buckets函数调用的次
数取决于bucket的个数，即在model_with_buckets
函数中，会为每个bucket使用seq2seq_f函数构建
出一套网络Seq2Seq的网络模型，但是不用担心，
它们的权重是共享的。具体可以参见
model_with_buckets函数的实现，就是使用了共享
变量的机制。

11. 定义Seq2seq模型的输入占位符

下面定义Seq2Seq模型的输入占位符，这些
占位符都是为了传入model_with_buckets函数中做
准备的。

首先是Seq2Seq模型自己的两个list占位符：
一个是输入encoder_inputs，一个是输出
decoder_inputs。另外，model_with_buckets还需要
一个额外的输入，在前面已经提过，因为其在做
loss时使用的是带权重的交叉熵，所以还要输入

大小等同于decoder_inputs的权重target_weights。

另外还有一个输入就是做交叉熵时的标签targets，因为它与decoder_inputs一样，所以可以直接由decoder_inputs变换而来，把decoder_inputs的第一个“_GO”去掉，在放到targets中。

代码9-34 seq2seq_model（续）

```
117      # 注入数据
118      self.encoder_inputs = []
119      self.decoder_inputs = []
120      self.target_weights = []
121      for i in xrange(buckets[-1][0]):    # 最后的bucket 是最
122          self.encoder_inputs.append(tf.placeholder(tf.int32,
123                                              shape=[None],
124                                              name="encoder_inputs"))
125          self.decoder_inputs.append(tf.placeholder(tf.int32,
126                                              shape=[None],
127                                              name="decoder_inputs"))
128          self.target_weights.append(tf.placeholder(dtype,
129                                              name="weights"))
130      #将解码器移动一位得到targets
131      targets = [self.decoder_inputs[i + 1]
132                  for i in xrange(len(self.decoder_inputs) - 1)]
```

占位符的list大小是取buckets中的最大数。targets的长度与buckets 的长度一致，decoder_inputs与target_weights的长度会比buckets 的长度大1，因为前面有“_GO”占位。

12. 定义正向的输出与loss

当一切参数准备好后，就可以使用model_with_buckets将整个网络贯穿起来了。

在测试时会只进行正向传播，这时seq2seq_f里面的最后一个参数为True，该参数最终会在seq2seq_f里的embedding_attention_seq2seq中的feed_previous中生效。前面介绍过，如果为True时，表明只有第一个decoder输入是“_GO”开头，这样可以保证测试时，模型可以一直记着前面的cell状态。

代码9-34 seq2seq_model（续）

```
133     # 训练的输出和loss定义
134     if forward_only:
135         self.outputs, self.losses = tf.contrib.legacy_seq2seq.
136             with_buckets(
137                 self.encoder_inputs, self.decoder_inputs, target_
138                 weights, buckets, lambda x, y: seq2seq_f(x, y,
139                 True),
140                 softmax_loss_function=softmax_loss_function)
141             # 如果使用了输出映射，
```

需要为解码器映射输出处理

```
140             if output_projection is not None:
141                 for b in xrange(len(buckets)):
142                     self.outputs[b] = [
143                         tf.matmul(output, output_projection[0]) +
144                         projection[1]
145                         for output in self.outputs[b]
146                     ]
147             else:
148                 self.outputs, self.losses = tf.contrib.legacy_seq2seq.
149             with_buckets(
150                 self.encoder_inputs, self.decoder_inputs, target_
151                 weights, buckets,
152                 lambda x, y: seq2seq_f(x, y, False),
153                 softmax_loss_function=softmax_loss_function)
```

在测试过程中，还需要将model_with_buckets的输出结果转化成outputs维度的one_hot。因为model_with_buckets是多个桶的输出，所以需要对每个桶都进行转换。

13. 反向传播计算梯度并通过优化器更新

在前面已经通过model_with_buckets得到了loss。

下面的代码先通过tf.trainable_variables函数获得可训练的参数params，然后用tf.gradients计算loss对应参数params的梯度，并通过tf.clip_by_global_norm将过大的梯度按照max_gradient_norm来截断，将截断后的梯度通过优化器opt来迭代更新。同样，还要针对每个桶(bucket)进行这样的操作。

代码9-34 seq2seq_model (续)

```
152     # 梯度下降更新操作
153     params = tf.trainable_variables()
154     if not forward_only:
155         self.gradient_norms = []
156         self.updates = []
157         opt = tf.train.GradientDescentOptimizer(self.lear
158         for b in xrange(len(buckets)):
159             gradients = tf.gradients(self.losses[b], params)
160             clipped_gradients, norm = tf.clip_by_global_norm(
161             gradients, max_gradient_norm)
162             self.gradient_norms.append(norm)
163             self.updates.append(opt.apply_gradients(
164                 zip(clipped_gradients, params), global_step=step))
```

```
165  
166     self.saver = tf.train.Saver(tf.global_variables())
```

最后更新saver，在代码“9-35 train.py”中会调用这部分代码来保存训练中的学习参数及相关变量。

14. 按批次获取样本数据

在模型中，按批次获取的样本数据并不能直接使用，还需要在get_batch函数中进行相应转化，首先根据指定bucket_id所对应的大小确定输入和输出的size，根据size进行pad的填充，并且针对输出数据进行第一位为“_Go”的重整作为解码的input。这里用了个小技巧将输入的数据进行了倒序排列。而对于输入weight则将其全部初始化为0，对应的size为每一批次中decoder每个序列一个权重weight，即与decoder相等。

代码9-34 seq2seq_model（续）

```
167 def get_batch(self, data, bucket_id):  
168     """在迭代训练过程中，从指定 bucket 中获得一个随机批次数据  
169  
170     Args:  
171         data: 一个大小为 len(self.buckets) 的 tuple，包含了创建一  
入输出的  
172             lists.  
173         bucket_id: 整型，指定从哪个 bucket 中取数据。  
174  
175     Returns:  
176         方便以后调用的 triple (encoder_inputs, decoder_input  
weights)  
177     .
```

*****ebook converter DEMO Watermarks*****

```

178     """
179     encoder_size, decoder_size = self.buckets[bucket_id]
180     encoder_inputs, decoder_inputs = [], []
181
182     # 获得一个随机批次的数据作为编码器与解码器的输入
183     # 如果需要时会有pad操作，同时反转encoder的输入顺序，并且为decoder的输入添加一个额外的“GO”，
184     for _ in xrange(self.batch_size):
185         encoder_input, decoder_input = random.choice(data)
186
187         # pad和反转Encoder 的输入数据
188         encoder_pad = [data_utils.PAD_ID] * (encoder_size - len(encoder_input))
189         encoder_inputs.append(list(reversed(encoder_input + encoder_pad)))
190
191     # 为Decoder输入数据添加一个额外的“GO”，

```

并且进行pad

```

192     decoder_pad_size = decoder_size - len(decoder_inputs)
193     decoder_inputs.append([data_utils.GO_ID] + decoder_pad)
194     decoder_inputs += [data_utils.PAD_ID] * decoder_pad_size
195
196     # 从上面选择好的数据中创建 batch-major vectors
197     batch_encoder_inputs, batch_decoder_inputs, batch_weights = [], [], []
198
199     for length_idx in xrange(encoder_size):
200         batch_encoder_inputs.append(
201             np.array([encoder_inputs[batch_idx][length_idx]
202                      for batch_idx in xrange(self.batch_size)], dtype=np.int32))
203
204     for length_idx in xrange(decoder_size):
205         batch_decoder_inputs.append(
206             np.array([decoder_inputs[batch_idx][length_idx]
207                      for batch_idx in xrange(self.batch_size)], dtype=np.int32))
208
209     # 定义target_weights 变量， 默认是1， 如果对应的targets是PAD符号，则target_weights就为0
210     batch_weight = np.ones(self.batch_size, dtype=np.float32)
211     for batch_idx in xrange(self.batch_size):
212         # 如果对应的输出target 是一个 PAD符号， 就将weight设为0
213         # 将decoder_input向前移动1位得到对应的target
214         if length_idx < decoder_size - 1:
215             target = decoder_inputs[batch_idx][length_idx + 1]
216             if length_idx == decoder_size - 1 or target == data_utils.PAD_ID:
217                 batch_weight[batch_idx] = 0.0
218             batch_weights.append(batch_weight)
219     return batch_encoder_inputs, batch_decoder_inputs, batch_weights

```

*****ebook converter DEMO Watermarks*****

15. Seq2Seq框架的迭代更新处理

这部分代码主要是构建输入feed数据，即输出的OP。在输入时，根据传入的bucket_id构建相应大小的输入输出list，通过循环传入list中对应的操作符里。由于decoder_inputs的长度比bucket中的长度大1，所以需要再多放一位到decoder_inputs的list中，在前面构建targets时，需要将所有的decoder_inputs向后移一位，targets作为标签要与bucket中的长度相等。确切地说target_weights是与targets相等的，所以不需要再输入值。

代码9-34 seq2seq_model (续)

```
220     def step(self, session, encoder_inputs, decoder_inputs,
221               weights,
222               bucket_id, forward_only):
223         """注入给定输入数据步骤
224
225         Args:
226             session: tensorflow 所使用的session
227             encoder_inputs:用来注入encoder输入数据的numpy int vec
228             decoder_inputs:用来注入decoder输入数据的numpy int vec
229             target_weights:用来注入target weights的numpy float
230             bucket_id: which bucket of the model to use
231             forward_only: 只进行正向传播
232
233         Returns:
234             一个由gradient norm (不做反向时为none), average perplexity
235             outputs组成的tuple
236
237         Raises:
238             ValueError:如果 encoder_inputs, decoder_inputs, 或
239             weights 的长度与指定bucket_id 的bucket size不符合
```

*****ebook converter DEMO Watermarks*****

```

237     """
238     # 检查长度
239     encoder_size, decoder_size = self.buckets[bucket_id]
240     if len(encoder_inputs) != encoder_size:
241         raise ValueError("Encoder length must be equal to
bucket, "
242                         " %d != %d." % (len(encoder_input
243                             if len(decoder_inputs) != decoder_size:
244                                 raise ValueError("Decoder length must be equal to
bucket, "
245                                     " %d != %d." % (len(decoder_input
246                                         if len(target_weights) != decoder_size:
247                                             raise ValueError("Weights length must be equal to
bucket, "
248                                                 " %d != %d." % (len(target_weight
249
250     # 定义Input feed
251     input_feed = {}
252     for l in xrange(encoder_size):
253         input_feed[self.encoder_inputs[l].name] = encoder_
254     for l in xrange(decoder_size):
255         input_feed[self.decoder_inputs[l].name] = decoder_
256         input_feed[self.target_weights[l].name] = target_v
257
258     last_target = self.decoder_inputs[decoder_size].name
259     input_feed[last_target] = np.zeros([self.batch_size]
int32)
260
261     # 定义Output feed
262     if not forward_only:
263         output_feed = [self.updates[bucket_id],
264                         self.gradient_norms[bucket_id],
265                         self.losses[bucket_id]]
266     else:
267         output_feed = [self.losses[bucket_id]]
268         for l in xrange(decoder_size):
269             output_feed.append(self.outputs[bucket_id][l])
270
271     outputs = session.run(output_feed, input_feed)
272     if not forward_only:
273         return outputs[1], outputs[2], None
274     else:
275         return None, outputs[0], outputs[1:]

```

对于输出，也要区分是测试还是训练。如果

*****ebook converter DEMO Watermarks*****

是测试，需要将loss与logit输出，结果在outputs中，outputs[0]为loss，outputs[1:]为输出的decoder_size大小序列。如果是训练，输出需要更新的梯度与loss。这里多输出一个None是为了统一输出，保证第二位输出的都是loss。

整个代码进展到这里就可以进行训练操作了，运行train.py文件，将模型运行起来进行迭代训练。输出结果如下：

```
Building prefix dict from the default dictionary ...
Loading model from cache C:\Users\LIJINH~1\AppData\Local\Temp\tmpfjwv\...
Loading model cost 0.672 seconds.
Prefix dict has been built successfully.
vocab_size 11963
vocab_sizech 15165
checkpoint_dir is fanyichina/checkpoints/
new a cell
new a cell
new a cell
new a cell
model is ok
Using bucket sizes:
[(20, 20), (40, 40), (50, 50), (60, 60)]
fanyichina/fromids/english1w.txt
fanyichina/toids/chinese1w.txt
bucket sizes = [1649, 4933, 1904, 1383]
fanyichina/checkpoints/seq2seqtest.ckpt
WARNING:tensorflow:Error encountered when serializing LAYER_
Type is unsupported, or the types of the items don't match 1
CollectionDef.
'dict' object has no attribute 'name'
eval: bucket 0 perplexity 1.71
```

可以看到输出了词典的大小vocab_size 11963、vocab_sizech 15165，与定义的buckets，4个bucket分别需要调用4次seq2seq_f，于是打印了*****ebook converter DEMO Watermarks*****

4次new a cell。接着会显示每一批次中每个bucket的输入（因为是反转的，这里已经给反过来 了），并且能够看到对输入的pad进行了填充。对于每个输出由'_GO'字符开始，结束时都会有'_EOS'字符。对于模型预测的输出结果，也是将'_EOS'字符前面的内容打印出来，没有'_EOS'字符的预测结果将视为没有翻译成功，因此没有打印出来。

16. 测试模型

测试模型代码在代码“9-36 test.py”文件中，与前面实例中的代码基本相似，需要考虑的是，在创建模型时要使用测试模式（最后一个参数为True），并且dropout设为1.0。在main函数里，先等待用户输入，然后对用户输入的字符进行处理并传入模型，最终输出结果并显示出来。完整代码如下。

代码9-36 test

```
01 import tensorflow as tf
02 import numpy as np
03 import os
04 from six.moves import xrange
05
06 _buckets = []
07 convo_hist_limit = 1
08 max_source_length = 0
09 max_target_length = 0
10
11 flags = tf.app.flags
12 FLAGS = flags.FLAGS
```

```

13 datautil = __import__("9-33  datautil")
14 seq2seq_model = __import__("9-34  seq2seq_model")
15 import datautil
16 import seq2seq_model
17
18 tf.reset_default_graph()
19
20 max_train_data_size= 0 #0表示训练数据的输入长度没有限制
21
22 data_dir = "datacn/"
23
24 dropout = 1.0
25 grad_clip = 5.0
26 batch_size = 60
27 hidden_size = 14
28 num_layers =2
29 learning_rate =0.5
30 lr_decay_factor =0.99
31
32 checkpoint_dir= "data/checkpoints/"
33
34 #####翻译
35 hidden_size = 100
36 checkpoint_dir= "fanyichina/checkpoints/"
37 data_dir = "fanyichina/"
38 _buckets =[ (20, 20), (40, 40), (50, 50), (60, 60) ]
39
40 def getfanyiInfo():
41     vocaben, rev_vocaben=datautil.initialize_vocabulary((datautil.data_dir, datautil.vocabulary_fileen))
42     vocab_sizeen= len(vocaben)
43     print("vocab_size",vocab_sizeen)
44
45     vocabch, rev_vocabch=datautil.initialize_vocabulary((datautil.data_dir, datautil.vocabulary_filech))
46     vocab_sizech= len(vocabch)
47     print("vocab_sizech",vocab_sizech)
48
49     return vocab_sizeen,vocab_sizech,vocaben,rev_vocabch
50
51 def main():
52
53     vocab_sizeen,vocab_sizech,vocaben,rev_vocabch= getfar
54
55     if not os.path.exists(checkpoint_dir):
56         os.mkdir(checkpoint_dir)
57     print ("checkpoint_dir is {0}".format(checkpoint_dir))
58
59     with tf.Session() as sess:

```

*****ebook converter DEMO Watermarks*****

```

60     model = createModel(sess, True, vocab_sizeen, vocab_
61
62     print (_buckets)
63     model.batch_size = 1
64
65     conversation_history = []
66     while True:
67         prompt = "请输入："
68         sentence = input(prompt)
69         conversation_history.append(sentence.strip())
70         conversation_history = conversation_history[·
71             limit:]
72
73         token_ids = list(reversed( datautil.sentence_
74             join(conversation_history) ,vocaben,normalize_d:
75             Isch=False) ) )
76         print(token_ids)
77         bucket_id = min([b for b in xrange(len(_buckets))
78             [b][0] > len(token_ids)])
79
80         encoder_inputs, decoder_inputs, target_weights =
81             get_batch({bucket_id: [(token_ids, [])]}), bucket_
82
83         _, _, output_logits = model.step(sess, encoder_
84             inputs, decoder_inputs, target_weights, bucket_id, True)
85
86         #使用 beam search策略
87         outputs = [int(np.argmax(logit, axis=1)) for
88             output_logits]
89         print("outputs",outputs,datautil.EOS_ID)
90         if datautil.EOS_ID in outputs:
91             outputs = outputs[:outputs.index(datauti·
92
93             convo_output = " ".join(datautil.ids2te·
94             rev_vocabch))
95             conversation_history.append(convo_output)
96             print (convo_output)
97         else:
98             print("can not translation! ")
99
92 def createModel(session, forward_only, from_vocab_size,to_
93     """Create translation model and initialize or load pa·
94     session."""
95     model = seq2seq_model.Seq2SeqModel(
96         from_vocab_size,#from
97         to_vocab_size,#to
98         _buckets,
99         hidden_size,
99         num_layers,

```

*****ebook converter DEMO Watermarks*****

```
100     dropout,
101     grad_clip,
102     batch_size,
103     learning_rate,
104     lr_decay_factor,
105     forward_only=forward_only,
106     dtype=tf.float32)
107
108     print("model is ok")
109
110     ckpt = tf.train.latest_checkpoint(checkpoint_dir)
111     if ckpt!=None:
112         model.saver.restore(session, ckpt)
113         print ("Reading model parameters from {0}".format(ckpt))
114     else:
115         print ("Created model with fresh parameters.")
116         session.run(tf.global_variables_initializer())
117
118     return model
119
120 if __name__=="__main__":
121 main()
```

运行代码，结果如下：

```
Building prefix dict from the default dictionary ...
Loading model from cache C:\Users\LIJINH~1\AppData\Local\Temp\tmpfjwv\seq2seqtest.ckpt-99600
Loading model cost 0.719 seconds.
Prefix dict has been built successfully.
vocab_size 11963
vocab_sizech 15165
checkpoint_dir is fanyichina/checkpoints/
new a cell
new a cell
new a cell
new a cell
model is ok
INFO:tensorflow:Restoring parameters from fanyichina/checkpoints/seq2seqtest.ckpt-99600
Reading model parameters from fanyichina/checkpoints/seq2seqtest.ckpt-99600
[(20, 20), (40, 40), (50, 50), (60, 60)]
```

请输入： will reap good results and the large
[149, 4, 6, 341, 169, 4980, 22]

*****ebook converter DEMO Watermarks*****

```
not use
outputs[838, 838, 26, 105, 643, 8, 1595, 1089, 5, 968, 8, 968, 6, 2, 5, 136
最终 最终 也会 对此 和 坚强 有力 的 指导 和 指导 .
```

当前的例子是“跑了”约半天时间的模型效果，通过载入检查点打印信息可以看到当前迭代了99 600次，从原有的样本中简单复制几句话输入系统中，则系统可以大致翻译出一些汉语。可以看到它并没有按照词顺序逐个翻译，而是用学到原有样本的意思来表达，尽管语句还不通畅。这里只是做个演示，如果需要训练更好的模型，可以增加样本数量，并增加训练时间。

9.9 实例75：制作一个简单的聊天机器人

实例74中的Seq2Seq模型的代码可以作为很好的框架来扩展使用，简单地改变一下数据样本，即可扩展到许多更有意思的应用中。例如，让机器人对对联、讲故事、生成文章摘要、汉语翻译成英语、聊天机器人等都可以实现。这些扩展应用基本上不需要改动太多的代码就可以完成，本节以聊天机器人来举例演示。

实例描述

准备一部分聊天对话的语料，使用Seq2Seq模式对其进行学习，拟合特征，从而实现聊天机器人的功能。

基于9.8.6节例子中的代码文件，本例中需要变化的代码主要在处理样本方面，包括“9-33 datautil.py”“9-35 train.py”“9-36 test.py”。“9-34 seq2seq_model.py”文件为模型文件，可以不做变化，如需要修改网络结构，可以在其seq2seq_f函数中改变cell的组成即可。这样新生成的文件就是“9-37 datautil.py”“9-38 seq2seq_model.py”“9-39 train.py”“9-40 test.py”，具体步骤如下。

9.9.1 构建项目框架

新建一个文件夹（本例为“实例75 dialog”），将“实例74”原有代码全部复制进去，然后建立一个子文件夹datacn用于放样本，同时在datacn文件夹里建立checkpoints、dialog、fromids和toids这4个文件夹。

9.9.2 准备聊天样本

因本例只是演示作用，因此并没有用正规样本，只是随意写了几句对话放到了两个文件里，然后将文件放到dialog下。

9.9.3 预处理样本

修改代码“9-33 datautil.py”文件，将main函数修改如下，更新data_dir、raw_data_dir_to路径，将英文字典相关的代码全部注释掉见代码第15~21行。

代码9-37 datautil

```
01 .....
02 data_dir = "datacn/"
03 raw_data_dir_to = "datacn/dialog/"
04 vocabulary_filech = "dictch.txt"
05
06 plot_histograms = plot_scatter =True
07 vocab_size =40000
08
09 max_num_lines =1
10 max_target_size = 200
```

```
11 max_source_size = 200
12
13 def main():
14     vocabulary_filenamech = os.path.join(data_dir, vocabu
15 ##########
16     #创建英文字典
17     #     training_dataen, counten, dictionaryen, reverse_dict
18     textsszen =create_vocabulary(vocabulary_filenameen
19     #                                     ,raw_data_dir,vocab_s:
20     #                                     normalize_digits = True)
21     #     print("training_data",len(training_dataen))
22     #     print("dictionary",len(dictionaryen))
23 ##########
24     #创建中文字典
25     training_datach, countch, dictionarych, reverse_dict:
26     textsszch =create_vocabulary(vocabulary_filenamech
27                                     ,raw_data_dir_to,vocab
28                                     normalize_digits = True)
29     source_file,target_file =splitFileOneline(training_da
30     textsszch)
31     print("training_datach",len(training_datach))
32     print("dictionarych",len(dictionarych))
33     analysisfile(source_file,target_file)
```

创建中文字典之后，通过splitFileOneline函数将原有样本分为from和to，即把对话中的两个角色分到两个文档里。splitFileOneline的定义如下：

代码9-37 datautil（续）

```
29 #将读好的对话文本按行分开，一行问，一行答。存为两个文件。training_
textssz为每行的索引
30 def splitFileOneline(training_data ,textssz):
31     source_file = os.path.join(data_dir+'fromids/ ', "data_
test.txt")
32     target_file = os.path.join(data_dir+'toids/ ', "data_1
test.txt")
33     create_seq2seqfile(training_data,source_file ,target_
textssz)
34     return source_file,target_file
```

运行之后可以看到，在datacn下生成了中文字典，并且在datacn\fromids与datacn\toids下生成了两个ids文件。

9.9.4 训练样本

训练样本步骤只修改代码“9-35 train.py”中的样本部分即可，代码如下，将原来在main函数之前的翻译相关的信息代码全部去掉，换成dialog的相关信息，更新checkpoint_dir与buckets，定义getdialogInfo。为了返回值不变，返回的英文词典和英文词典中的英文词数量替换成返回中文词典和中文词典中的中文词数量。

代码9-39 train

```
.....  
checkpoint_dir= "datacn/checkpoints/"  
  
_buckets =[(5, 5), (10, 10), (20, 20)]  
def getdialogInfo():  
    vocabch, rev_vocabch=datautil.initialize_vocabulary(os.p  
dir, datautil.vocabulary_filech))  
    vocab_sizech= len(vocabch)  
    print("vocab_sizech",vocab_sizech)  
    filesfrom,_=datautil.getRawFileList(datautil.data_dir+"1  
filesto,_=datautil.getRawFileList(datautil.data_dir+"to:  
source_train_file_path = filesfrom[0]  
target_train_file_path= filesto[0]  
return vocab_sizech,vocab_sizech,rev_vocabch,rev_vocabch,sol  
path,target_train_file_path  
def main():  
    vocab_sizeen,vocab_sizech,rev_vocaben,rev_vocabch,source_tr  
path,target_train_file_path = getdialogInfo()  
.....
```

在main函数的第一句中，修改调用的函数为getdialogInfo以获得dialog的信息，修改完样本后，就可以运行该文件进行模型训练了。由于样本量非常小（仅仅是演示而已），因此模型训练的时间也很短，几分钟即可。

9.9.5 测试模型

与上一步类似，修改代码“9-36 test.py”中的样本部分即可，代码如下，将原来在main函数之前的翻译相关的信息代码全部去掉，换成dialog的相关信息，更新checkpoint_dir与buckets，定义getdialogInfo。为了返回值不变，将原来返回的英文词典中的英文词数量变为返回中文词典中的中文词数量用中文词典代替英文词典，并且在转换成ids的地方需要将isch改为True。

代码9-40 test

```
hidden_size = 100
checkpoint_dir= "datacn/checkpoints/"
_buckets =[(5, 5), (10, 10), (20, 20)]
def getdialogInfo():
    vocabch,rev_vocabch=datautil.initialize_vocabulary(os.path.join(
        datautil.data_dir, datautil.vocabulary_filech))
    vocab_sizech= len(vocabch)
    print("vocab_sizech",vocab_sizech)
    filesfrom,_=datautil.getRawFileList(datautil.data_dir+"1")
    filesto,_=datautil.getRawFileList(datautil.data_dir+"to")
    source_train_file_path = filesfrom[0]
    target_train_file_path= filesto[0]
    return vocab_sizech,vocab_sizech,vocabch,rev_vocabch
```

```
def main():
    vocab_sizeen,vocab_sizech,vocaben,rev_vocabch= getdialogInfo()
    .....
    while True:
        prompt = "请输入："
        sentence = input(prompt)
        conversation_history.append(sentence.strip())
        conversation_history = conversation_history[-conv_max_size:]
        token_ids=list(reversed(datautil.sentence_to_ids(
            conversation_history) ,vocaben,normalize_digits=True,Isch=True) )
    .....
```

同样在main函数的第一行中修改代码，调用getdialogInfo获得信息。

整个代码完成后运行程序，并输入类似样本中简单的对话，可以看到如下结果：

```
请输入：你好
[20]
outputs [30, 2, 2, 2, 2] 2
您好
```

```
请输入：你吃了吗
[12, 7, 4, 6]
outputs [5, 4, 7, 2, 2] 2
我吃了
```

```
请输入：吃的啥
[3, 10, 4]
outputs [5, 4, 10, 2, 2] 2
我吃的
```

```
请输入：你吃啥
[3, 4, 6]
outputs [5, 4, 17, 2, 2] 2
我吃三文鱼
```

```
请输入：还有吗
[12, 11]
```

*****ebook converter DEMO Watermarks*****

```
outputs [5, 29, 13, 16, 2] 2  
我 没吃够 呢 不能
```

可以看到，在样本里最后一句的回答完全不一样，但是神经网络仿佛学到了里面的语义。在简单的问话：“还有吗？”可以读懂说话人的意思是想要，于是输出：“我 没吃够 呢 不能”。从聊天机器人的例子可以看出，通过学习某个专业方面的对话样本（如某个业务的客服对话），会在该业务下产生很好的语义，并有很好的专业交流。

该例只是演示，主要目的是为了让读者学会如何应用框架代码，有兴趣的读者可以找些样本，自己动手试试，将前面举例的几种场景应用到模型中。当然，前面列出的场景只是一部分，只要符合序列对序列的模式都可以用Seq2Seq的框架来学习。希望读者可以举一反三，在此基础上做出更多出色的应用。

9.10 时间序列的高级接口TFTS

TFTS（TensorFlow Time Series）是一个专门处理时间序列数据的高级接口，从TensorFlow的1.3版本开始，陆续改进迭代，直到1.5版本得到了最终的完善。

TFTS属于估算器框架下的一个具体应用。估算器是TensorFlow 1.3版本的新功能，是对机器学习全流程的代码封装。使得开发者不需要再编写流程代码，按照估算器的框架专心实现某一部分具有独立功能（如模型结构、样本输入处理）的代码即可。

TFTS的具体接口在tf.contrib.timeseries下。它支持非线性自动回归模型（估算器：ARRegressor）、基于线性状态空间建模的组件集合模型（包括趋势、预测、向量自回归、移动平均值等，估算器：StructuralEnsembleRegressor）、自定义LSTM模型。

开发者可以按照估算器的框架对时间序列数据进行训练、预测。该接口不仅具有处理单变量和多变量的时间序列数据的功能，还具有按照标注忽略具体序列数据的功能。

为了让开发者方便使用，该接口特意给出了4个例子。统一放在如下地址中：

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

上述网址打开后，可以找到如下4个代码文件，分别对应4个例子，具体介绍如下。

- known_anomaly.py：单变量时间序列训练及模型评估，并带有按照标注值忽略的功能；
- multivariate.py：多变量时间序列训练及模型评估；
- predict.py：对时间序列进行训练及预测的例子；
- lstm.py：自定义lstm模型例子。

这4个例子中的知识点几乎介绍了TFTS的全部应用。但对模型的导出、载入并未有代码演示。TFTS模型部分使用了saved_model接口。读者可以在如下网站查看关于TFTS的导出、载入说明。

https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/timeseries/python/timeseries/saved_model.py



注意：TFTS的预测功能并不是一个独

立的计算，它是依赖估算器的evaluate方法的返回值进行的。这就要求在使用predict之前必须进行一次evaluate的调用。但是在TFTS的代码中，对evaluate的封装是默认在开发环境下运行的，每次调用都会生成支持TensorBoard的Summary日志文件。在生产情况下，这个功能会消耗不必要的性能。可以通过注释掉源码库中的对应代码将其关闭。具体做法如下：

- (1) 打开文件Anaconda3\lib\site-packages\tensorflow\python\estimator\estimator.py；
- (2) 在_evaluate_model函数中，找到如下代码：

```
_write_dict_to_summary(  
    output_dir=eval_dir,  
    dictionary=eval_results,  
    current_global_step=eval_results[ops.GraphKeys.GLOBAL_STEP])
```

- (3) 将其全部注释掉即可。

第10章 自编码网络——能够自学习样本特征的网络

深度学习领域主要有两种训练模式：一种是监督学习，即不仅有样本，还有对应的标签；另一种是非监督学习，即只有样本没有标签。此外还有半监督学习，但也属于非监督领域，这里不展开讲解了。

对于监督学习的训练任务，为已有样本准备对应的标签是项很繁重的工作，所以相对来讲，非监督学习就显得简单得多。如果能让网络直接使用样本进行训练，不再需要再准备标签，则是更高效的事情。

本章来学习一个非监督模型的网络——自编码网络。

本章含有教学视频共11分15秒。

作者按照本章的内容结构，对主要内容进行了概括性讲解，对自编码网络的结构、用途及类型部分依次做了简要介绍（掌握自编码网络的分步训练方法，以及条件变分自编码网络是本章内容的重点）。



*****ebook converter DEMO Watermarks*****

10.1 自编码网络介绍及应用

人们平时看一幅图时，并不是像计算机那个逐个像素去读，一般是扫一眼物体，大致能得到需要的信息，如形状、颜色和特征等。那么怎样让机器也有这项能力呢？这里就为大家介绍一下自编码网络。

自编码网络是非监督学习领域中的一种，可以自动从无标注的数据中学习特征，是一种以重构输入信号为目标的神经网络，它可以给出比原始数据更好的特征描述，具有较强的特征学习能力，在深度学习中常用自编码网络生成的特征来取代原始数据，以得到更好的结果。

10.2 最简单的自编码网络

自编码（Auto-Encoder，AE）网络是输入等于输出的网络，最基本的模型可以视为三层的神经网络，即输入层、隐藏层、输出层。其中，输入层的样本也会充当输出层的标签角色。换句话说，这个神经网络就是一种尽可能复现输入信号的神经网络。具体的网络结构如图10-1所示。

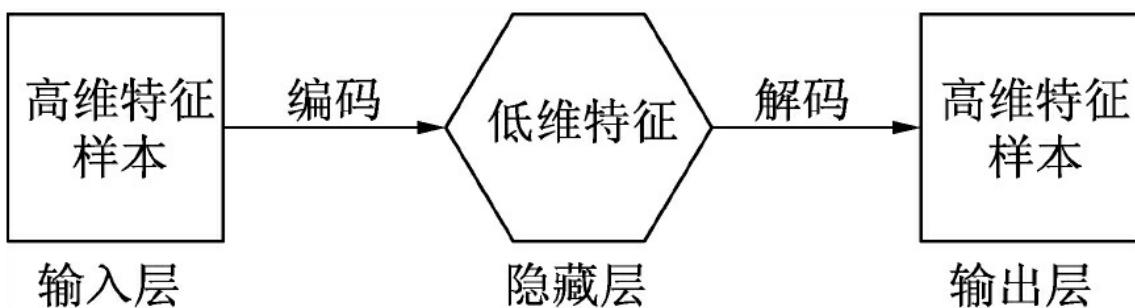


图10-1 让输出的信号等于输入

其中，从输入到中间状态的过程叫做编码，从中间状态再回到输出的过程叫做解码。这样构成的自动编码器可以捕捉代表输入数据的最重要的因素，类似PCA算法（主成份分析），找到可以代表原信息的主要成分。

自编码器要求输出尽可能等于输入，并且其隐藏层必须满足一定的稀疏性，是通过将隐藏层中的后一层个数比前一层神经元个数少的方式来实现稀疏效果的。相当于隐藏层对输入进行了压缩，并在输出层中解压缩。整个过程中肯定会丢

失信息，但训练能够使丢失的信息尽量减少，最大化地保留其主要特征。

如果激活函数不使用Sigmoid函数，而使用线性函数，那么便是PCA模型了。

10.3 自编码网络的代码实现

本节通过实例演示自编码网络的实现。

10.3.1 实例76：提取图片的特征，并利用特征还原图片

实例描述

通过构建一个两层降维的自编码网络，将MNIST数据集的数据特征提取出来，并通过这些特征再重建一个MNIST数据集。

本例分为如下几个步骤。

1. 引入头文件，并加载MNIST数据

假设MNIST数据放在代码文件同级目录的data下，将其以one-hot的形式载入。

代码10-1 自编码

```
01 import tensorflow as tf
02 import numpy as np
03 import matplotlib.pyplot as plt
04
05 # 导入MINST 数据集
06 from tensorflow.examples.tutorials.mnist import input_data
07 mnist = input_data.read_data_sets("/data/", one_hot=True)
```

2. 定义网络模型

下面输入MNIST数据集的图片，将其像素点组成的数据（ $28 \times 28 = 784$ ）从784维降维到256，然后再降到128，最后再以同样的方式经过128再经过256，最终还原到原来的图片，其过程如图10-2所示。

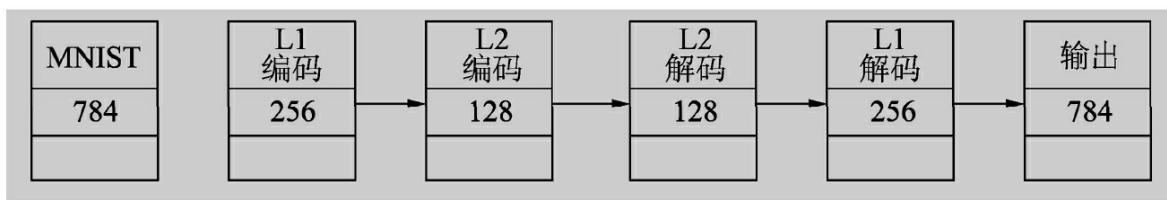


图10-2 自编码实例代码的维度变化过程

定义网络模型的具体代码如下。

代码10-1 自编码（续）

```
08 learning_rate = 0.01
09 n_hidden_1 = 256
10 n_hidden_2 = 128
11 n_input = 784
12
13 # 占位符
14 x = tf.placeholder("float", [None, n_input])      #输入
15 y = x
16
17 #学习参数
18 weights = {
19     'encoder_h1': tf.Variable(tf.random_normal([n_input,
20         1])),
20     'encoder_h2': tf.Variable(tf.random_normal([n_hidden_
21         2])),
21     'decoder_h1': tf.Variable(tf.random_normal([n_hidden_
22         1])),
22     'decoder_h2': tf.Variable(tf.random_normal([n_hidden_
```

```
    input])),  
23 }  
24 biases = {  
25     'encoder_b1': tf.Variable(tf.zeros([n_hidden_1])),  
26     'encoder_b2': tf.Variable(tf.zeros([n_hidden_2])),  
27     'decoder_b1': tf.Variable(tf.zeros([n_hidden_1])),  
28     'decoder_b2': tf.Variable(tf.zeros([n_input])),  
29 }  
30  
31 # 编码  
32 def encoder(x):  
33     layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['  
h1']),biases['encoder_b1']))  
34     layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, we:  
['encoder_h2']), biases['encoder_b2']))  
35     return layer_2  
36  
37 # 解码  
38 def decoder(x):  
39     layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['  
h1']),biases['decoder_b1']))  
40     layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, we:  
['decoder_h2']),biases['decoder_b2']))  
41     return layer_2  
42  
43 #输出的节点  
44 encoder_out = encoder(x)  
45 pred = decoder(encoder_out)  
46  
47 # cost为y与pred的平方差  
48 cost = tf.reduce_mean(tf.pow(y - pred, 2))  
49 optimizer = tf.train.RMSPropOptimizer(learning_rate).min:
```

上面代码里先定义了学习率为0.01，这个值可以动态调节，会直接影响到收敛速度和学习的准确性，由于输出标签也是输入标签，所以后面直接定义 $y=x$ 。

3. 开始训练

接下来设置训练参数，一次取256条数据，

*****ebook converter DEMO Watermarks*****

将所有的训练数据集进行20次的迭代训练。

代码10-1 自编码（续）

```
50 # 训练参数
51 training_epochs = 20          #一共迭代20次
52 batch_size = 256             #每次取256个样本
53 display_step = 5             #迭代5次输出一次信息
54
55 # 启动会话
56 with tf.Session() as sess:
57     sess.run(tf.global_variables_initializer())
58     total_batch = int(mnist.train.num_examples/batch_size)
59     # 开始训练
60     for epoch in range(training_epochs): #迭代
61
62         for i in range(total_batch):
63             batch_xs, batch_ys = mnist.train.next_batch(batch_size)
64             _, c = sess.run([optimizer, cost], feed_dict={x: batch_xs,
65                         y: batch_ys})
66             if epoch % display_step == 0: # 现实日志信息
67                 print("Epoch:", '%04d' % (epoch+1), "cost=", \
68                     format(c))
69     print("完成!")
```

4. 测试模型

接下来通过MNIST数据集中的test集来测试一下模型的准确度。

代码10-1 自编码（续）

```
68     correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
69     # 计算错误率
70     accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
71     print ("Accuracy:", 1-accuracy.eval({x: mnist.test.images,
72                                         y: mnist.test.labels}))
```

执行代码，信息输出如下：

```
Epoch: 0001 cost= 0.216481194
Epoch: 0006 cost= 0.144190893
Epoch: 0011 cost= 0.128914982
Epoch: 0016 cost= 0.120772459
完成!
Accuracy: 0.885104
```

上面的输出信息中，前面打印的是每一次迭代的错误率，最终输出的Accuracy指的是整个模型的正确率。

5. 双比输入和输出

随意取出10张图片，比对一下输入与输出，可以看到自编码网络还原的图片与真实图片几乎一样。

代码10-1 自编码（续）

```
72 # 可视化结果
73     show_num = 10
74     reconstruction = sess.run(
75         pred, feed_dict={x: mnist.test.images[:show_num]}
76     f, a = plt.subplots(2, 10, figsize=(10, 2))
77     for i in range(show_num):
78         a[0][i].imshow(np.reshape(mnist.test.images[i], (
79             a[1][i].imshow(np.reshape(reconstruction[i], (28,
80             plt.draw()
```

执行上面的代码，会生成如图10-3所示图片。图片分为上下两行，第一行显示的内容为输
*****ebook converter DEMO Watermarks*****

入图片，第二行显示的内容为输出图片。

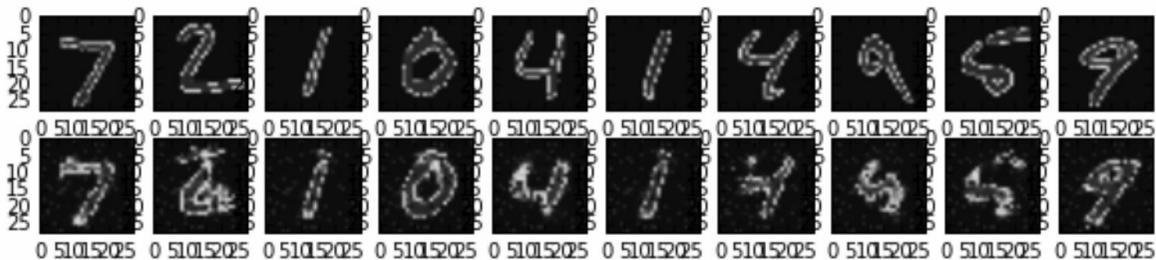


图10-3 自编码实例的输出结果

10.3.2 线性解码器

在实例76中使用的激活函数为S型激活函数，输出范围是[0, 1]，当我们对最终提取的特征节点采用该激励函数时，就相当于对输入限制或缩放，使其位于[0, 1]范围内。有一些数据集，比如MNIST，能方便地将输出缩放到[0, 1]中，但是很难满足对输入值的要求。例如，PCA白化处理的输入并不满足[0, 1]范围要求，也不清楚是否有最好的办法可以将数据缩放到特定范围内。

如果利用一个恒等式来作为激励函数，就可以很好地解决这个问题，即将 $f(z) = z$ 作为激励函数（即，没有激励函数）。



注意：这个方法只是对最后的输出层而言，对于神经网络中隐含层的神经元依然还要使用S型（或者tanh）激励函数。

由多个带有S型激活函数的隐含层及一个线性输出层构成的自编码器，称为线性解码器。下面来看一个线性解码器的例子。

10.3.3 实例77：提取图片的二维特征，并利用二维特征还原图片

本节用一个更为极致的例子来展示自编码网络的“威力”。将MNIST图片压缩成二维数据，这样也可以在直角坐标系上将其显示出来，让读者更形象地了解自编码网络在特征提取方面的功能。

实例描述

在自编码网络中使用线性解码器对MNIST数据特征进行再压缩，并将其映射到直角坐标系上。

这里使用4层逐渐压缩将784维度分别压缩成256、64、16、2这4个特征向量。编码部分的具体结构如图10-4所示。

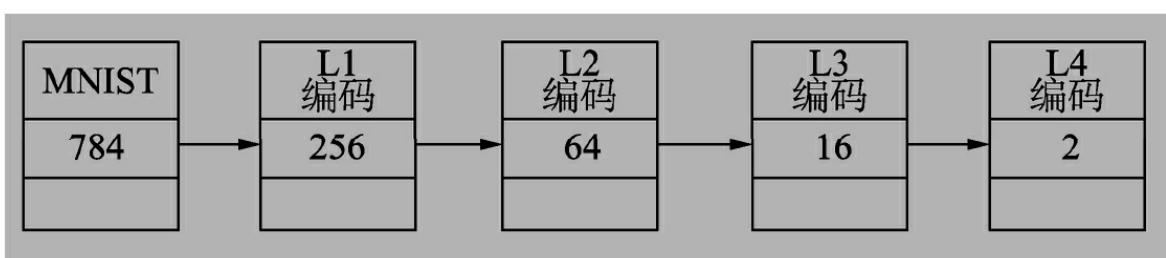


图10-4 线性解码器实例编码部分网络结构

然后以直角坐标系的形式将数据点显示出来，这样可以更直观地看到自编码器对于同一类图片的聚类效果。



说明：如果读者想得到更好的特征提取效果，可以将压缩的层数变得更多，每层压缩一点点（如：512、256、128、64、32、16、2），由于Sigmoid函数的“天生”缺陷，无法使用更深的层，所以这里只能做成4层压缩。但不用担心，这个问题可以在学完本章内容之后得到一个满意的解决办法（使用栈式自编码器）。

在这个例子中分为如下几步来编写代码。

1. 引入头文件，定义学习参数变量

由于要建立4层网络，所以要为每一层分配节点个数，学习参数。

代码10-2 自编码进阶

```
01 import tensorflow as tf
02 import numpy as np
03 import matplotlib.pyplot as plt
04
05 # 导入MNIST数据集
06 from tensorflow.examples.tutorials.mnist import input_data
07 mnist = input_data.read_data_sets("/data/", one_hot=True)
08
```

```
09 # 定义学习率
10 learning_rate = 0.01
11 # 隐藏层设置
12 n_hidden_1 = 256
13 n_hidden_2 = 64
14 n_hidden_3 = 16
15 n_hidden_4 = 2
16 n_input = 784 # MNIST data 输入(img shape: 28*28)
17
18 #定义输入占位符
19 x = tf.placeholder("float", [None, n_input])
20 y=x
21 weights = {
22     'encoder_h1': tf.Variable(tf.random_normal([n_input,
23     'encoder_h2': tf.Variable(tf.random_normal([n_hidden_
24     'encoder_h3': tf.Variable(tf.random_normal([n_hidden_
25     'encoder_h4': tf.Variable(tf.random_normal([n_hidden_
26
27     'decoder_h1': tf.Variable(tf.random_normal([n_hidden_
28     'decoder_h2': tf.Variable(tf.random_normal([n_hidden_
29     'decoder_h3': tf.Variable(tf.random_normal([n_hidden_
30     'decoder_h4': tf.Variable(tf.random_normal([n_hidden_
31 }
32
33 biases = {
34     'encoder_b1': tf.Variable(tf.zeros([n_hidden_1])),
35     'encoder_b2': tf.Variable(tf.zeros([n_hidden_2])),
36     'encoder_b3': tf.Variable(tf.zeros([n_hidden_3])),
37     'encoder_b4': tf.Variable(tf.zeros([n_hidden_4])),
38
39     'decoder_b1': tf.Variable(tf.zeros([n_hidden_3])),
40     'decoder_b2': tf.Variable(tf.zeros([n_hidden_2])),
41     'decoder_b3': tf.Variable(tf.zeros([n_hidden_1])),
42     'decoder_b4': tf.Variable(tf.zeros([n_input])),
43 }
```

2. 定义网络模型

下面的代码是定义编码和解码的网络结构，这里使用了线性解码器。在编码的最后一层，没有进行Sigmoid变换，这是因为生成的二维数据其数据特征已经变得极为重要，所以我们希望让它

透传到解码器中，少一些变换可以最大化地保存原有的主要特征。当然，这一切也是通过分析之后实际测试得来的结果。

代码10-2 自编码进阶（续）

```
44 def encoder(x):
45     layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['encoder_w1']),
46                                 biases['encoder_b1']))
47     layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights['encoder_h2']),
48                                 biases['encoder_b2']))
49     layer_3 = tf.nn.sigmoid(tf.add(tf.matmul(layer_2, weights['encoder_h3']),
50                                 biases['encoder_b3']))
51     layer_4 = tf.add(tf.matmul(layer_3, weights['encoder_h4']),
52                     biases['encoder_b4'])
53     return layer_4
54
55 def decoder(x):
56     layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['decoder_w1']),
57                                 biases['decoder_b1']))
58     layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights['decoder_h2']),
59                                 biases['decoder_b2']))
60     layer_3 = tf.nn.sigmoid(tf.add(tf.matmul(layer_2, weights['decoder_h3']),
61                                 biases['decoder_b3']))
62     layer_4 = tf.nn.sigmoid(tf.add(tf.matmul(layer_3, weights['decoder_h4']),
63                                 biases['decoder_b4']))
64     return layer_4
65 # 构建模型
66 encoder_op = encoder(x)
67 y_pred = decoder(encoder_op) # 784 维度
68
69 cost = tf.reduce_mean(tf.pow(y - y_pred, 2))
70 optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
```

3. 开始训练

这一步中还是一次取256条数据，将全部数据集迭代20次。

代码10-2 自编码进阶（续）

```
71 #训练
72 training_epochs = 20 # 迭代训练20一次
73 batch_size = 256
74 display_step = 1
75
76 with tf.Session() as sess:
77     sess.run(tf.global_variables_initializer())
78     total_batch = int(mnist.train.num_examples/batch_size)
79     # 启动循环开始训练
80     for epoch in range(training_epochs):
81         # 遍历全部数据集
82         for i in range(total_batch):
83             batch_xs, batch_ys = mnist.train.next_batch(
84                 _, c = sess.run([optimizer, cost], feed_dict=_
85                 # 显示训练中的详细信息
86                 if epoch % display_step == 0:
87                     print("Epoch:", '%04d' % (epoch+1),
88                           "cost=", "{:.9f}".format(c))
89         print("完成!")
```

输出结果如下：

```
Epoch: 0001 cost= 0.106694221
Epoch: 0002 cost= 0.096146211
Epoch: 0003 cost= 0.089687020
Epoch: 0004 cost= 0.085342437
Epoch: 0005 cost= 0.076942392
Epoch: 0006 cost= 0.077152036
Epoch: 0007 cost= 0.074504733
Epoch: 0008 cost= 0.071438089
Epoch: 0009 cost= 0.070937753
Epoch: 0010 cost= 0.067885153
Epoch: 0011 cost= 0.068935215
Epoch: 0012 cost= 0.067724347
Epoch: 0013 cost= 0.065405361
```

```
Epoch: 0014 cost= 0.069433592
Epoch: 0015 cost= 0.068582796
Epoch: 0016 cost= 0.067146875
Epoch: 0017 cost= 0.065363437
Epoch: 0018 cost= 0.066899218
Epoch: 0019 cost= 0.065677144
Epoch: 0020 cost= 0.064701408
完成!
```

可以看出，通过自编码网络将784维的数据压缩成了二维，用二维数据来代替784维，这就是自编码网络的神奇之处！

4. 对比输入和输出

同样我们再添加一些代码将效果显示出来。随意取出10张图片，并将图片输入模型中，得到输出图片。同时比对一下输入与输出的图片。

代码10-2 自编码进阶（续）

```
90      # 可视化结果
91      show_num = 10
92      encode_decode = sess.run(
93          y_pred, feed_dict={x: mnist.test.images[:show_num]})
94      # 将自编码输出结果和原始样本显示出来
95      f, a = plt.subplots(2, 10, figsize=(10, 2))
96      for i in range(show_num):
97          a[0][i].imshow(np.reshape(mnist.test.images[i], (28,
98              28)))
99          a[1][i].imshow(np.reshape(encode_decode[i], (28,
```

执行上面的代码，生成如图10-5所示图片。

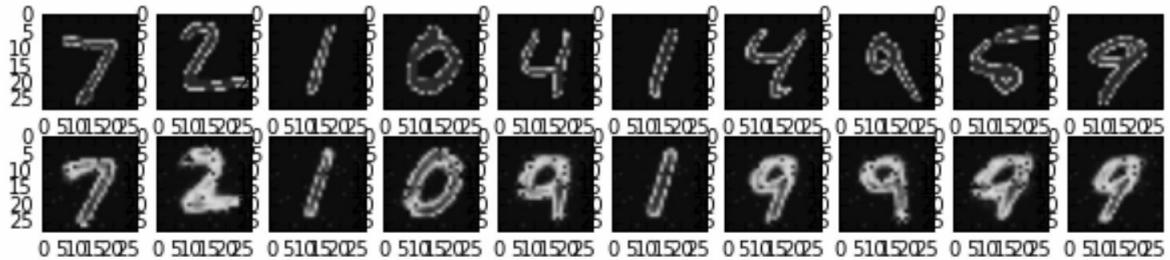


图10-5 自编码进阶实例结果1

5. 显示数据的二维特征

接着就是比较好玩的事情了。我们要把数据压缩后的二维特征显示出来。

代码10-2 自编码进阶（续）

```
100      aa = [np.argmax(l)for l in mnist.test.labels]#将onehot
101      encoder_result = sess.run(encoder_op, feed_dict={x:
102          images})
103      plt.scatter(encoder_result[:, 0], encoder_result[:, 1],
104      plt.colorbar()
105      plt.show()
```

执行上面的代码，生成如图10-6所示图片。

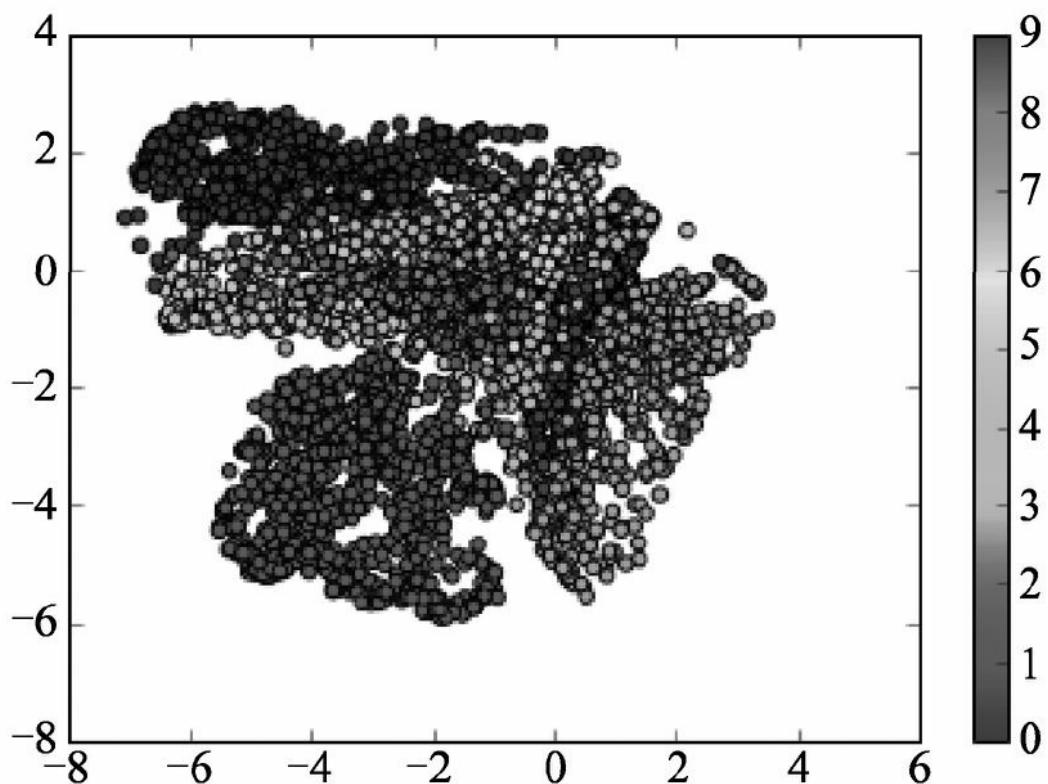


图10-6 自编码进阶实例结果2

这样看是不是直观多了！看了这个图你会有什么感觉，聚类？K均值？softmax？没错，一般来讲用自编码网络将数据降维之后的数据更有利进行分类处理。



注意： 上面代码中的aa是将mnist.test.labels里面的one_hot转换成一般的数字，然后进行图像显示。这个代码建议读者把它收集起来，因为在深度学习过程中one_hot的互转会经常用到。另外，one_hot转码有多种方法，细心的读者可以在前面章节中找到更简洁的转换代码。

当然也可以不用这句代码，那么在最前面引
*****ebook converter DEMO Watermarks*****

入MNIST时就必须把onehot关掉，将

```
mnist = input_data.read_data_sets("/data/", one_hot=True)
```

改成：

```
mnist = input_data.read_data_sets("/data/", one_hot=False)
```

同时将倒数第三句改成使用mnist的测试标签：

```
plt.scatter(encoder_result[:,0],encoder_result[:,1],c=mnist
```

10.3.4 实例78：实现卷积网络的自编码

自编码结构不仅只用在全连接网络上，还可用在卷积网络上。下面举例实现一个卷积网络的自编码。代码变化不大，是在原有的基础上将全连接改成卷积，具体改动步骤如下。

实例描述

在自编码网络中使用卷积网络完成MNIST的自编码功能。

1. 修改网络权重定义

保持b不变， 将原来的w改成卷积核的定义如下。

代码10-3 卷积网络自编码

```
01 .....
02 #学习参数
03 weights = {
04     'encoder_conv1': tf.Variable(tf.truncated_normal([5,
05     1], stddev=0.1)),
05     'encoder_conv2': tf.Variable(tf.random_normal([3, 3,
06     conv_2], stddev=0.1)),
06     'decoder_conv1': tf.Variable(tf.random_normal([5, 5,
07     1], stddev=0.1)),
07     'decoder_conv2': tf.Variable(tf.random_normal([3, 3,
08     conv_2], stddev=0.1))
08 }
09 .....
```

2. 改变编码和解码结构

原来的编码器和解码器分别由全连接改成卷积和反卷积操作，通过外层的池化与反池化将整个网络贯穿起来。在网络入口处还要将输入的维度改成[-1, 28, 28, 1]。

代码10-3 卷积网络自编码（续）

```
10 x_image = tf.reshape(x, [-1, 28, 28, 1])
11 # 编码
12 def encoder(x):
13     h_conv1 = tf.nn.relu(conv2d(x, weights['encoder_conv1'],
14     ['encoder_conv1']))
14     h_conv2 = tf.nn.relu(conv2d(h_conv1, weights['encoder_conv2'],
15     biases['encoder_conv2']))
15     return h_conv2, h_conv1
```

*****ebook converter DEMO Watermarks*****

```
16
17 # 解码
18 def decoder(x, conv1):
19     t_conv1 = tf.nn.conv2d_transpose(x-biases['decoder_conv1'],
20                                     weights['decoder_conv2'], conv1.shape, [1,1,1,1])
21     t_x_image = tf.nn.conv2d_transpose(t_conv1-biases['decoder_conv1'],
22                                         weights['decoder_conv1'], x_image.shape, [1,1,1,1])
23     return t_x_image
24
25 #输出的节点
26 encoder_out, conv1 = encoder(x_image)
27 h_pool2, mask = max_pool_with_argmax(encoder_out, 2)
28 h_upool = unpool(h_pool2, mask, 2)
29 pred = decoder(h_upool, conv1)
```

上面代码中用到的卷积函数和反池化函数，可以在8.4节和8.6节中查看具体实现过程。

3. 测试及可视化部分改动

因为反池化的函数要求输入图片的一个维度不能为None，所以，需要把评估部分也改一下。

代码10-3 卷积网络自编码（续）

```
29     # 测试
30     batch_xs, batch_ys = mnist.train.next_batch(batchsize)
31     print ("Error:", cost.eval({x: batch_xs}))
32
33     # 可视化结果
34     show_num = 10
35     reconstruction = sess.run(
36
37         pred, feed_dict={x: batch_xs})
38
39     f, a = plt.subplots(2, 10, figsize=(10, 2))
40     for i in range(show_num):
41
42         a[0][i].imshow(np.reshape(batch_xs[i], (28, 28)))
*****ebook converter DEMO Watermarks*****
```

```
43         a[1][i].imshow(np.reshape(reconstruction[i], (28,
44             plt.draw()
```

运行上面的代码，可以看到如下信息，生成的图片如图10-7所示。

```
Epoch: 0001 cost= 0.026543943
Epoch: 0006 cost= 0.538754463
Epoch: 0011 cost= 0.006631755
Epoch: 0016 cost= 0.003391982
完成!
error: 0.0214768
```

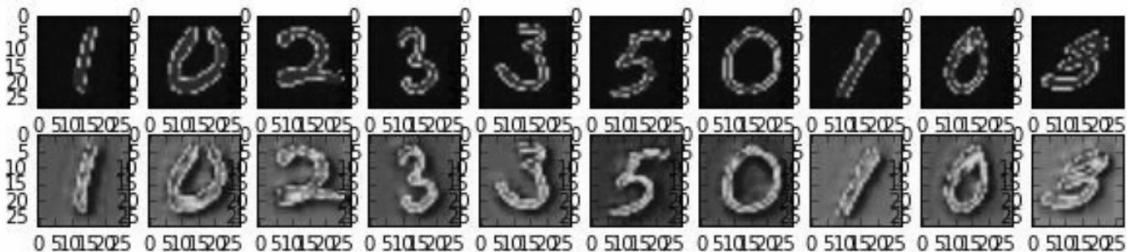


图10-7 卷积自编码网络实例

10.3.5 练习题

仿照10.3.1节的例子，试着建立一个更深层的自编码，分为3层将图片压缩成512、256、128维度，然后再将其还原（可以参考本书配套代码的代码“10-4 自编码练习题.py”文件）。

10.4 去噪自编码

要想取得好的特征只靠重构输入数据是不够的，在实际应用中，还需要让这些特征具有抗干扰的能力，即当输入数据发生一定程度的扰动时，生成的特征仍然保持不变。这时需要添加噪声来为模型增加更大的困难。在这种情况下训练出来的模型才会有更好的鲁棒性，于是就有了本节所介绍的去噪自动编码器。

去噪自动编码器（Denoising Autoencoder, DA），是在自动编码的基础上，训练数据加入噪声，输出的标签仍是原始的样本（没有加过噪声的），这样自动编码器必须学习去除噪声而获得真正的没有被噪声污染过的输入特征。因此，这就迫使编码器去学习输入信号的更加鲁棒的特征表达，即具有更加强悍的泛化能力。

在实际训练中，人为加入的噪声有两种途径：

- (1) 在选择训练数据集时，额外选择一些样本集以外的数据。
- (2) 改变已有的样本数据集中的数据（使样本个体不完整，或通过噪声与样本进行的加减乘除之类的运算，使样本数据发生变化）。

10.5 去噪自编码网络的代码实现

下面进入实例环节，通过例子来构建一个去噪自编码网络。

10.5.1 实例79：使用去噪自编码网络提取MNIST特征

本节做一个更简单的自编码模型，让784维只通过一层压缩成256维。与前面例子唯一不同的是，将原始的数据进行一些变换，每个像素点都乘以一个高斯噪声，然后在输出的位置仍然使用原始的输入样本，这样迫使网络在提取特征的同时将噪声去掉。为了防止其过拟合，还需要在其中加入Dropout层。

在这个例子中分为如下几个步骤来编写代码。

实例描述

对MNIST集原始输入图片加入噪声，在自编码网络中进行训练，以得到抗干扰更强的特征提取模型。

1. 引入头文件，创建网络模型及定义学习参数变量

*****ebook converter DEMO Watermarks*****

代码10-5 去噪声自编码

```
01 import numpy as np
02 import tensorflow as tf
03 import matplotlib.pyplot as plt
04 from tensorflow.examples.tutorials.mnist import input_data
05
06 mnist = input_data.read_data_sets("/data/", one_hot=True)
07
08 train_X = mnist.train.images
09 train_Y = mnist.train.labels
10 test_X = mnist.test.images
11 test_Y = mnist.test.labels
12
13 n_input = 784
14 n_hidden_1 = 256
15
16 # 占位符
17 x = tf.placeholder("float", [None, n_input])
18 y = tf.placeholder("float", [None, n_input])
19 dropout_keep_prob = tf.placeholder("float")
20
21 # 学习参数
22 weights = {
23     'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
24     'h2': tf.Variable(tf.random_normal([n_hidden_1, n_input])),
25     'out': tf.Variable(tf.random_normal([n_hidden_1, n_input]))
26 }
27 biases = {
28     'b1': tf.Variable(tf.zeros([n_hidden_1])),
29     'b2': tf.Variable(tf.zeros([n_hidden_1])),
30     'out': tf.Variable(tf.zeros([n_input]))
31 }
32
33 # 网络模型
34 def denoise_auto_encoder(_X, _weights, _biases, _keep_prob):
35     layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(_X, _weights['h1']),
36         _biases['b1']))
37     layer_1out = tf.nn.dropout(layer_1, _keep_prob)
38     layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1out,
39         [_weights['h2']]), _biases['b2']))
40     layer_2out = tf.nn.dropout(layer_2, _keep_prob)
41     return tf.nn.sigmoid(tf.matmul(layer_2out, _weights['out']))
42
43 reconstruction = denoise_auto_encoder(x, weights, biases,
```

*****ebook converter DEMO Watermarks*****

```
keep_prob)
42
43 # COST计算
44 cost = tf.reduce_mean(tf.pow(reconstruction-y, 2))
45 # 优化器
46 optm = tf.train.AdamOptimizer(0.01).minimize(cost)
```

可以看到，在定义学习参数时，加了dropout的学习参数，因为后面要为网络添加dropout层。

2. 设置训练参数，开始训练

这一步还和前面例子一样，重点看下一步的变化。

代码10-5 去噪声自编码（续）

```
47 #训练参数
48 epochs      = 20
49 batch_size  = 256
50 disp_step   = 2
51
52 with tf.Session() as sess:
53     sess.run(tf.global_variables_initializer())
54     print ("开始训练")
55     for epoch in range(epochs):
56         num_batch = int(mnist.train.num_examples/batch_size)
57         total_cost = 0.
58         for i in range(num_batch):
```

3. 生成噪声数据

在这里做了添加噪声的操作，每次取出一批次的数据，将输入数据的每一个像素都加上0.3倍的高斯噪声。

*****ebook converter DEMO Watermarks*****

代码10-5 去噪声自编码（续）

```
59 batch_xs, batch_ys = mnist.train.next_batch(batch_size)
60         batch_xs_noisy = batch_xs + 0.3*np.random.ran
61         feeds = {x: batch_xs_noisy, y: batch_xs, drop
62             sess.run(optm, feed_dict=feeds)
63             total_cost += sess.run(cost, feed_dict=feeds)
64
65     # 显示训练日志
66     if epoch % disp_step == 0:
67         print ("Epoch %02d/%02d average cost: %.6f"
68                 % (epoch, epochs, total_cost/num_batch
69     print ("完成")
```

执行上面的代码，生成如下信息：

```
开始训练
Epoch 00/20 average cost: 0.097613
Epoch 02/20 average cost: 0.073714
Epoch 04/20 average cost: 0.068687
Epoch 06/20 average cost: 0.065391
Epoch 08/20 average cost: 0.063086
Epoch 10/20 average cost: 0.062062
Epoch 12/20 average cost: 0.061144
Epoch 14/20 average cost: 0.060415
Epoch 16/20 average cost: 0.060192
Epoch 18/20 average cost: 0.059686
完成
```

4. 数据可视化

接下来是数据可视化部分，接着添加以下代码。

代码10-5 去噪声自编码（续）

*****ebook converter DEMO Watermarks*****

```
70     show_num = 10
71     test_noisy = mnist.test.images[:show_num] + 0.3*np.random.randn(
72         show_num, 784)
73     encode_decode = sess.run(
74         reconstruction, feed_dict={x: test_noisy, dropout: 0.5})
75     f, a = plt.subplots(3, 10, figsize=(10, 3))
76     for i in range(show_num):
77         a[0][i].imshow(np.reshape(test_noisy[i], (28, 28)))
78         a[1][i].imshow(np.reshape(mnist.test.images[i], (28, 28)))
79         a[2][i].matshow(np.reshape(encode_decode[i], (28, 28)),
80                         cmap='gray')
81     plt.show()
```

执行上面的代码，生成如图10-8所示图片。

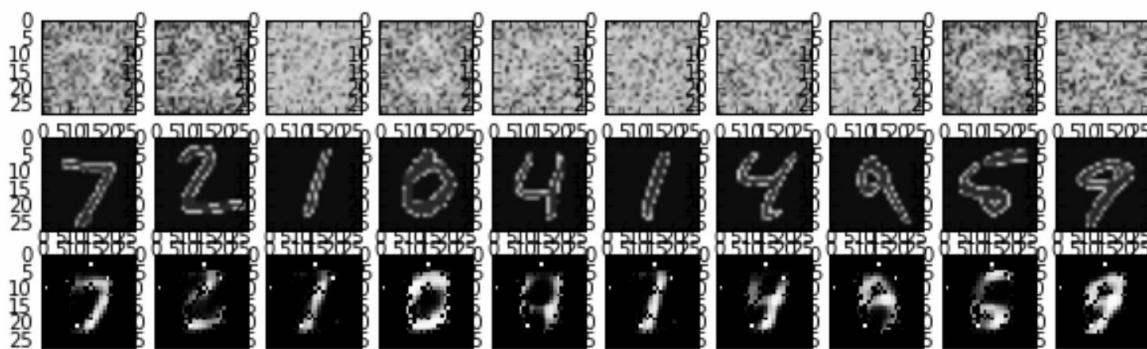


图10-8 去噪自编码结果1

第一行图片是加入噪声后的输入，第二行图片是原始的样本（在这里作为标签），最后一行是输出。这里为了让结果看起来明显一些，将输出以灰色的图来显示。可以看出，输出的图片还能看出原来的样子，而且基本上将前面的噪声大部分都过滤掉了。

5. 测试鲁棒性

为了测试模型的鲁棒性，我们换一种噪声方

*****ebook converter DEMO Watermarks*****

式，然后再生成一个样本测试效果（接着上面的 sess）。

代码10-5 去噪声自编码（续）

```
80     randidx = np.random.randint(test_X.shape[0], size=1)
81     orgvec = test_X[randidx, :]
82     testvec = test_X[randidx, :]
83     label = np.argmax(test_Y[randidx, :], 1)
84
85     print ("label is %d" % (label))
86     # 噪音类型
87     print ("Salt and Pepper Noise")
88     noisyvec = testvec
89     rate = 0.15
90     noiseidx = np.random.randint(test_X.shape[1]
91                                     , size=int(test_X.shape[1]))
92     noisyvec[0, noiseidx] = 1-noisyvec[0, noiseidx]
93     outvec = sess.run(reconstruction, feed_dict={x: noisyvec,
94                                                   dropout_keep_prob: 1})
94     outimg = np.reshape(outvec, (28, 28))
95
96     # 可视化
97     plt.matshow(np.reshape(orgvec, (28, 28)), cmap=plt.cm.gray)
98     plt.title("Original Image")
99     plt.colorbar()
100
101    plt.matshow(np.reshape(noisyvec, (28, 28)), cmap=plt.cm.gray)
102    plt.title("Input Image")
103    plt.colorbar()
104
105    plt.matshow(outimg, cmap=plt.get_cmap('gray'))
106    plt.title("Reconstructed Image")
107    plt.colorbar()
108    plt.show()
```

执行上面的代码，会生成如下信息，生成的图片如图10-9所示。

label is 9
Salt and Pepper Noise

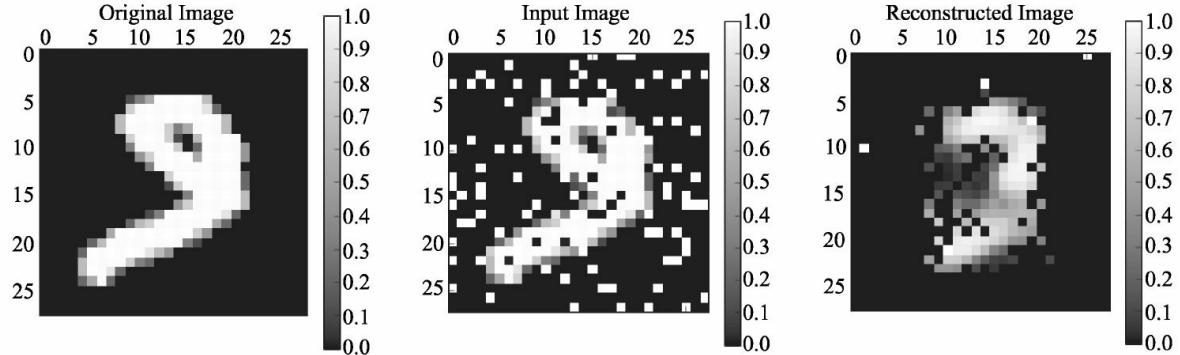


图10-9 去噪自编码结果2

可以看出，使用Salt and Pepper噪声对原始图片进行干扰后，仍然可以得到很好的效果。

10.5.2 练习题

试着将dropout的值改成0.5，看一下模型训练出来的样子，然后再将Salt and Pepper噪声变换后的图片放到模型里，观察输出的图片。想想这是为什么？为什么要在去噪声自编码网络里加入dropout。

10.6 栈式自编码

接下来一起看看什么是栈式自编码。

10.6.1 栈式自编码介绍

栈式自编码神经网络（Stacked Autoencoder，SA），是对自编码网络的一种使用方法，是一个由多层训练好的自编码器组成的神经网络。由于网络中的每一层都是单独训练而来，相当于都初始化了一个合理的数值。所以，这样的网络会更容易训练，并且有更快的收敛性及更高的准确度。

栈式自编码常常被用于预训练（初始化）深度神经网络之前的权重预训练步骤。例如，在一个分类问题上，可以按照从前向后的顺序执行每一层通过自编码器来训练，最终将网络中最深层的输出作为softmax分类器的输入特征，通过softmax层将其分开。

为了使这个过程容易理解，下面以训练一个包含两个隐含层的栈式自编码网络为例，一步一步为大家介绍具体操作。

(1) 训练一个自编码器，得到原始输入的一阶特征表示 $h^{(1)}$ （如图10-10中的features1所
*****ebook converter DEMO Watermarks*****

示)。

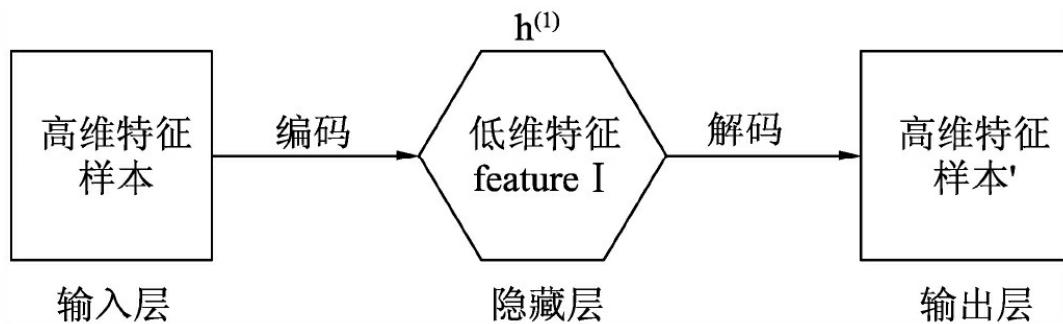


图10-10 栈式自编码一层结构

(2) 将上一步输出的特征 $h^{(1)}$ 作为输入，对其进行再一次的自编码，并同时获取特征 $h^{(2)}$ （如图10-11中的featuresII所示）。

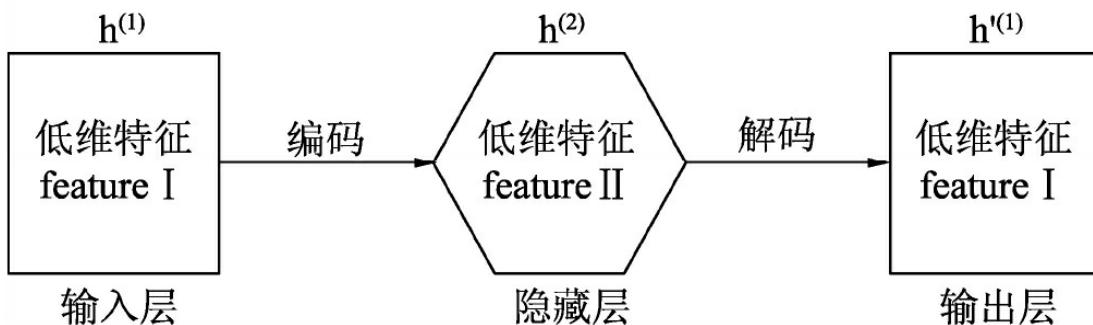


图10-11 栈式自编码二层结构

(3) 把上一步的特征 $h^{(2)}$ 连上softmax分类器，得到了一个图片数字标签分类的模型，具体网络结构如图10-12所示。

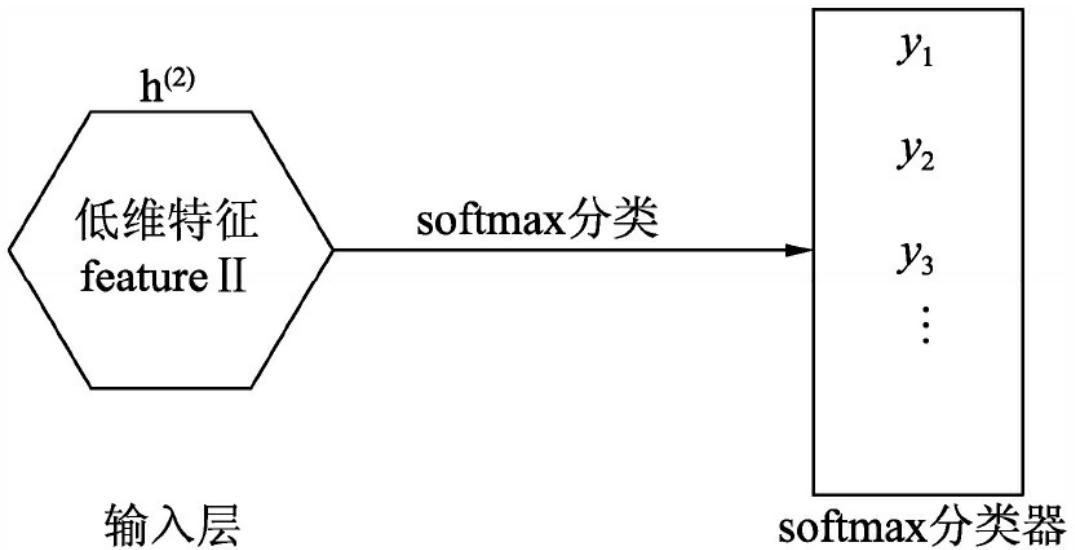


图10-12 栈式自编码三层结构

(4) 把这3层结合起来，就构成了一个包含两个隐藏层加一个softmax的栈式自编码网络，它可以对数字图片分类。具体网络结构如图10-13所示。

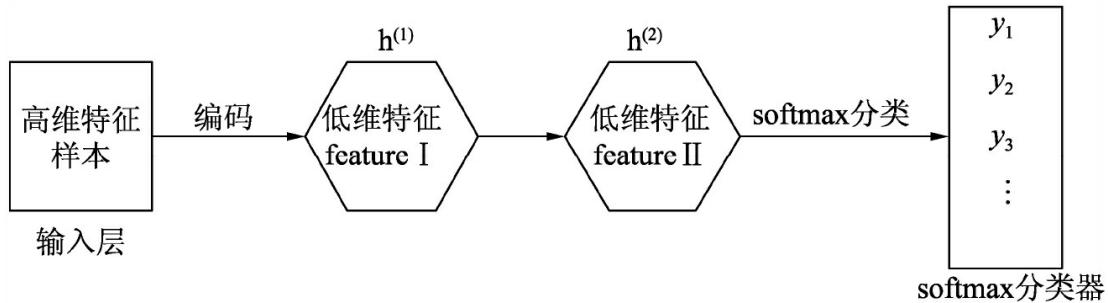


图10-13 栈式自编码级联结构

10.6.2 栈式自编码在深度学习中的意义

看到这里或许有读者会有疑问，为什么要这么麻烦，直接使用多层神经网络来训练不是也可

以吗？在这里是为大家介绍一种训练方法，而这个训练更像是手动训练，之所以我们愿意这么麻烦，主要是因为其有以下几个优点：

- 每一层都可以单独训练，保证降维特征的可控性。
- 对于高维度的分类问题，一下拿出一套完整可用的模型相对来讲并不是容易的事，因为节点太多，参数太多，一味地增加深度只会使结果越来越不可控，成为测底的黑盒，而使用栈式自编码逐层降维，可以将复杂问题简单化，更容易完成任务。
- 任意深层，理论上是越深层的神经网络对现实的拟合度越高，但是传统的多层神经网络，由于使用的是误差反向传播方式，导致层越深，传播的误差越小。栈式自编码巧妙地绕过这个问题，直接使用降维后的特征值进行二次训练，可以任意层数的加深。

栈式自编码神经网络具有强大的表达能力和深度神经网络的所有优点，它通常能够获取到输入的“层次型分组”或者“部分-整体分解”结构，自编码器倾向于学习得到与样本相对应的低维向量，该向量可以更好地表示高维样本的数据特征。

如果网络输入的是图像，第一层会学习去识别边，二层会学习组合边、构成轮廓角等，更高层会学习组合更形象的特征。例如，人脸图像，学习如何识别眼睛、鼻子、嘴等。

10.7 深度学习中自编码的常用方法

下面介绍深度学习中关于自编码的常用方法。

10.7.1 代替和级联

栈式自编码会将网络中的中间层作为下一个网络的输入进行训练。我们可以得到网络中每一个中间层的原始值，为了能有更好的效果，还可以使用级联的方式进一步优化网络参数。

在已有的模型上接着优化参数的步骤习惯上称为“微调”。该方法不仅在自编码网络，在整个深度学习里都是常见的技术。

在什么时候应用微调呢？通常仅在有大量已标注训练数据的情况下使用。在这样的情况下，微调能显著提升分类器性能。但如果有大量未标注数据集（用于非监督特征学习/预训练），却只有相对较少的已标注训练集，则微调的作用非常有限。

10.7.2 自编码的应用场景

本章使用MNIST举例，主要是为了得到一个

很好的可视化效果。但在实际应用中，全连接的自编码网络并不适合处理图片类问题，原因在第8章的开头部分已经讲过了，这里不再赘述。

自编码更像是一种技巧，任何一种网络及方法不可能不变化就可以满足所有的问题，现实环境中，需要使用具体的模型配合各种技巧来解决问题。明白其原理，知道它的优、劣势才是核心。在任何一个多维数据的分类中也可以用自编码，或在大型图片文类任务中，卷积池化后的特征数据进行自编码降维也是一个好方法。

10.8 去噪自编码与栈式自编码的综合实现

本节将前面的知识综合一下，实现一个把去噪自编码加入栈式自编码网络中的例子。

10.8.1 实例80：实现去噪自编码

这次我们把前面所学的知识全部用上一起做一个综合的实例。首先建立一个去噪自编码，然后再对第一层的输出做一次简单的自编码压缩，然后再将第二层的输出做一个softmax的分类，最后，把这3个网络里的中间层拿出来，组成一个新的网络进行微调。下面就来一一操作。

1. 引入头文件，创建网络模型及定义学习参数变量

实例描述

对MNIST集中的原始输入图片加入噪声，在自编码网络中进行训练，得到抗干扰更强的特征提取模型。

引入头文件，创建MINST数据集。

代码10-6 自编码综合

*****ebook converter DEMO Watermarks*****

```
01 import numpy as np
02 import tensorflow as tf
03 import matplotlib.pyplot as plt
04 from tensorflow.examples.tutorials.mnist import input_data
05 mnist = input_data.read_data_sets("/data/", one_hot=True)
06
07 train_X = mnist.train.images
08 train_Y = mnist.train.labels
09 test_X = mnist.test.images
10 test_Y = mnist.test.labels
```

2. 定义占位符

最终训练的网络为一个输入、一个输出和两个隐藏层，结构如图10-14所示。

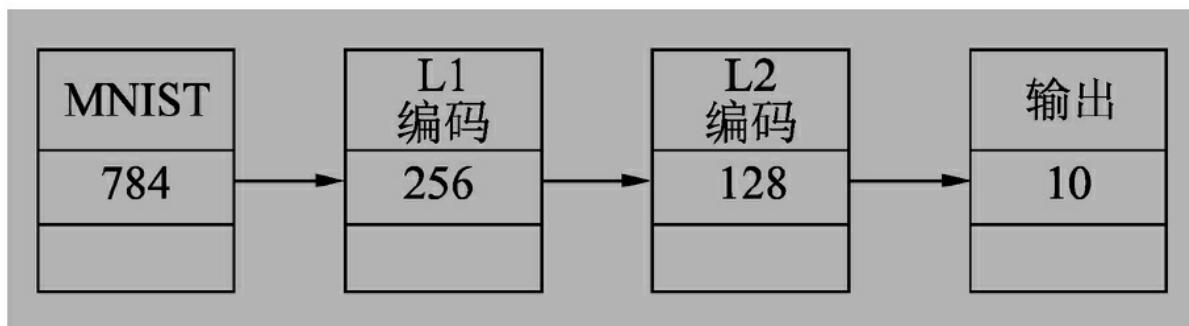


图10-14 自编码综合实例结构

在这个例子中要建立4个网络：每一层都用一个网络来训练，于是我们需要训练3个网络，最后再把训练好的各个层组合到一起，形成第4个网络。

代码10-6 自编码综合（续）

```
11 # NETWORK PARAMETERS
```

*****ebook converter DEMO Watermarks*****

```
12 n_input      = 784
13 n_hidden_1   = 256          #第一层自编码
14 n_hidden_2   = 128          #第二层自编码
15 n_classes    = 10
16
17 # 占位符
18 # 第一层输入
19 x = tf.placeholder("float", [None, n_input])
20 y = tf.placeholder("float", [None, n_input])
21 dropout_keep_prob = tf.placeholder("float")
22 #第二层输入
23 l2x = tf.placeholder("float", [None, n_hidden_1])
24 l2y = tf.placeholder("float", [None, n_hidden_1])
25 #第三层输入
26 l3x = tf.placeholder("float", [None, n_hidden_2])
27 l3y = tf.placeholder("float", [None, n_classes])
```

3. 定义学习参数

除了输入层，后面的其他三层（256、128、10）每一层都需要单独使用一个自编码网络来训练，所以要为这3个网络创建3套学习参数。

代码10-6 自编码综合（续）

```
28 # WEIGHTS
29 weights = {
30     #网络1 784-256-784
31     'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
32     'l1_h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
33     'l1_out': tf.Variable(tf.random_normal([n_hidden_1, n_classes])),
34     #网络2 256-128-256
35     'l2_h1': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
36     'l2_h2': tf.Variable(tf.random_normal([n_hidden_2, n_hidden_1])),
37     'l2_out': tf.Variable(tf.random_normal([n_hidden_2, n_classes])),
38     #网络3 128-10
39     'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
40 }
41 biases = {
42     'b1': tf.Variable(tf.zeros([n_hidden_1])),
43     'l1_b2': tf.Variable(tf.zeros([n_hidden_1])),
```

*****ebook converter DEMO Watermarks*****

```
44     'l1_out': tf.Variable(tf.zeros([n_input])),  
45  
46     'l2_b1': tf.Variable(tf.zeros([n_hidden_2])),  
47     'l2_b2': tf.Variable(tf.zeros([n_hidden_2])),  
48     'l2_out': tf.Variable(tf.zeros([n_hidden_1])),  
49  
50     'out': tf.Variable(tf.zeros([n_classes]))  
51 }
```

4. 第1层网络结构

为第1层建立一个自编码网络，并定义其网络结构。这里注意，由于要往第1层里加入噪声，所以第1层需要有dropout层。

代码10-6 自编码综合（续）

```
52 #第1层的编码输出  
53 l1_out = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['h1'])  
['b1']))  
54  
55 #l1 解码器MODEL  
56 def noise_l1_autodecoder(layer_1, _weights, _biases, _keep_prob)  
57     layer_1out = tf.nn.dropout(layer_1, _keep_prob)  
58     layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1out,  
['l1_h2']), _biases['l1_b2']))  
59     layer_2out = tf.nn.dropout(layer_2, _keep_prob)  
60     return tf.nn.sigmoid(tf.matmul(layer_2out, _weights['  
_biases['l1_out']])))  
61  
62 # 第一层的解码输出  
63 l1_reconstruction = noise_l1_autodecoder(l1_out, weights,  
dropout_keep_prob)  
64  
65 # 计算COST  
66 l1_cost = tf.reduce_mean(tf.pow(l1_reconstruction-y, 2))  
67 # OPTIMIZER  
68 l1_optm = tf.train.AdamOptimizer(0.01).minimize(l1_cost)
```

5. 第2层网络结构

为第2层建立一个自编码网络，并定义其网络结构。

代码10-6 自编码综合（续）

```
69 #12 解码器MODEL
70 def l2_autodecoder(layer1_2, _weights, _biases):
71     layer1_2out = tf.nn.sigmoid(tf.add(tf.matmul(layer1_2,
72         ['l2_h2']), _biases['l2_b2']))
73     return tf.nn.sigmoid(tf.matmul(layer1_2out, _weights|
74         _biases['l2_out']))
75
76 #第二层的编码输出
77 l2_out = tf.nn.sigmoid(tf.add(tf.matmul(l2x, weights['l2_'
78     'biases['l2_b1']])))
79 # 第二层的解码输出
80 l2_reconstruction = l2_autodecoder(l2_out, weights, biases)
81 # COST计算
82 l2_cost = tf.reduce_mean(tf.pow(l2_reconstruction-l2y, 2))
83 # 优化器
84 optm2 = tf.train.AdamOptimizer(0.01).minimize(l2_cost)
```

6. 第3层网络结构

为第3层建立一个自编码网络，并定义其网络结构。

代码10-6 自编码综合（续）

```
83 l3_out = tf.matmul(l2x, weights['out']) + biases['out']
84 l3_cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_
85 (logits=l3_out, labels=l3y))
86 l3_optm = tf.train.AdamOptimizer(0.01).minimize(l3_cost)
```

*****ebook converter DEMO Watermarks*****

7. 定义级联网络结构

将3层网络级联在一起，建立第4个网络，并定义其网络结构。这里复用了l1_out的节点，因为它是第1层的输出，其输入数据也是原始样本，与级联网络结构里的第一层一样，其他几层则需要重新定义。

代码10-6 自编码综合（续）

```
86 #3层 级联
87 #1联2
88 l1_l2out = tf.nn.sigmoid(tf.add(tf.matmul(l1_out, weights['l2_w1']),
89 # 2联3
90 pred = tf.matmul(l1_l2out, weights['out']) + biases['out']
91 # 定义loss和优化器
92 cost3 = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_
93 (logits=pred, labels=l3y))
94 optm3 = tf.train.AdamOptimizer(0.001).minimize(cost3)
```

8. 第1层网络训练

网络结构定义好之后，下面开始第1层网络的训练。

代码10-6 自编码综合（续）

```
94 epochs = 50
95 batch_size = 100
96 disp_step = 10
97
```

*****ebook converter DEMO Watermarks*****

```
98 with tf.Session() as sess:  
99     sess.run(tf.global_variables_initializer())  
100  
101    print ("开始训练")  
102    for epoch in range(epochs):  
103        num_batch = int(mnist.train.num_examples/batch_  
104        total_cost = 0.  
105        for i in range(num_batch):  
106            batch_xs, batch_ys = mnist.train.next_batch(t  
107            batch_xs_noisy = batch_xs + 0.3*np.random.rar  
108            feeds = {x: batch_xs_noisy, y: batch_xs, drop  
109            sess.run(l1_optm, feed_dict=feeds)  
110            total_cost += sess.run(l1_cost, feed_dict=fee  
111            # DISPLAY  
112            if epoch % disp_step == 0:  
113                print ("Epoch %02d/%02d average cost: %.6f"  
114                                % (epoch, epochs, total_cost/num_batc  
115    print ("完成")
```

执行上面的代码，生成如下信息：

```
开始训练  
Epoch 00/50 average cost: 0.113718  
Epoch 10/50 average cost: 0.035614  
Epoch 20/50 average cost: 0.033097  
Epoch 30/50 average cost: 0.032123  
Epoch 40/50 average cost: 0.031541  
完成
```

从测试数据集中拿出10个样本放到模型里，将生成的结果可视化。

代码10-6 自编码综合（续）

```
116 show_num = 10  
117     test_noisy = mnist.test.images[:show_num] + 0.3*np.i  
     (show_num, 784)  
118     encode_decode = sess.run(  
119         l1_reconstruction, feed_dict={x: test_noisy, dr
```

*****ebook converter DEMO Watermarks*****

```
    prob: 1.})
120     f, a = plt.subplots(3, 10, figsize=(10, 3))
121     for i in range(show_num):
122         a[0][i].imshow(np.reshape(test_noisy[i], (28, 28)))
123         a[1][i].imshow(np.reshape(mnist.test.images[i],
124                               (28, 28)))
125         a[2][i].matshow(np.reshape(encode_decode[i], (28, 28)),
cmap=plt.get_cmap('gray'))
126     plt.show()
```

执行上面的代码，生成如图10-15所示信息。

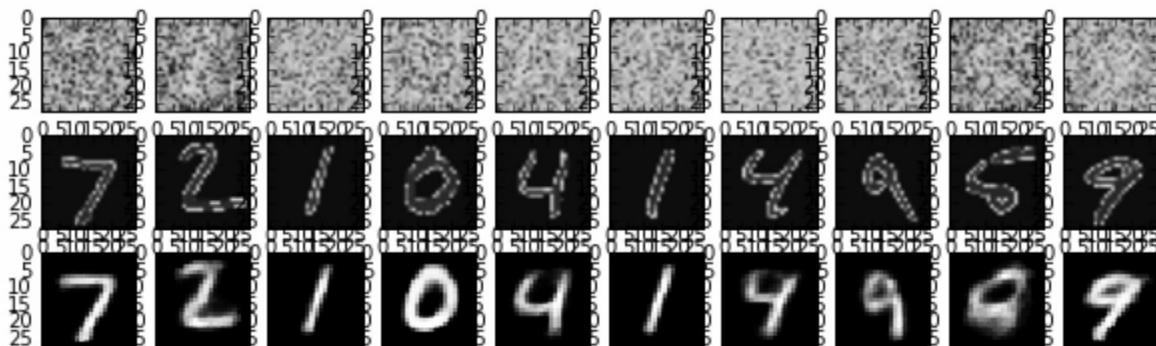


图10-15 自编码综合实例第1层结果

为什么这次的还原数据几乎将噪声全部过滤了呢？还记得去噪自编码那一章的练习题吗？这就是dropout的效果。仔细看，这次dropout的值设成了0.5，意味着有一半的节点是丢弃的，所以才会得到更好的拟合效果。

9. 第2层网络训练

下面开始训练第2层网络。需要注意的地方是，这个网络模型的输入已经不再是我们的MNIST图片了，而是上一层的输出，所以在准备输入数据时，要让输入的数据在上一层的模型中

运算一次才可以作为本次的输入。

代码10-6 自编码综合（续）

```
126 with tf.Session() as sess:  
127     sess.run(tf.global_variables_initializer())  
128     print ("开始训练")  
129     for epoch in range(epochs):  
130         num_batch = int(mnist.train.num_examples/batch_  
131         total_cost = 0.  
132         for i in range(num_batch):  
133             batch_xs, batch_ys = mnist.train.next_batch()  
134  
135             l1_h = sess.run(l1_out, feed_dict={x: batch_  
136             dropout_keep_prob: 1.})  
137             _,l2cost = sess.run([optm2,l2_cost], feed_d:  
138             12y: l1_h })  
139             total_cost += l2cost  
140  
141             # log输出  
142             if epoch % disp_step == 0:  
143                 print ("Epoch %02d/%02d average cost: %.6f"  
144                         % (epoch, epochs, total_cost/num_batch))  
145             print ("完成 layer_2 训练")  
146
```

执行上面的代码，生成如下信息：

```
开始训练  
Epoch 00/50 average cost: 0.126013  
Epoch 10/50 average cost: 0.019285  
Epoch 20/50 average cost: 0.014477  
Epoch 30/50 average cost: 0.012606  
Epoch 40/50 average cost: 0.012108  
完成 layer_2 训练
```

同理，可视化部分也是这样，所有准备输入的点都要在模型1中生成一次，见以下代码。

*****ebook converter DEMO Watermarks*****

代码10-6 自编码综合（续）

```
145     show_num = 10
146     testvec = mnist.test.images[:show_num]
147     out1vec = sess.run(l1_out, feed_dict={x: testvec,y:
148         dropout_keep_prob: 1.})
149     out2vec = sess.run(l2_reconstruction, feed_dict={l2>
150         f, a = plt.subplots(3, 10, figsize=(10, 3))
151         for i in range(show_num):
152             a[0][i].imshow(np.reshape(testvec[i], (28, 28)))
153             a[1][i].imshow(np.reshape(out1vec[i], (16, 16)))
154             a[2][i].matshow(np.reshape(out2vec[i], (16, 16))
155             get_cmap('gray'))
156         plt.show()
```

执行上面的代码，生成如图10-16所示信息。

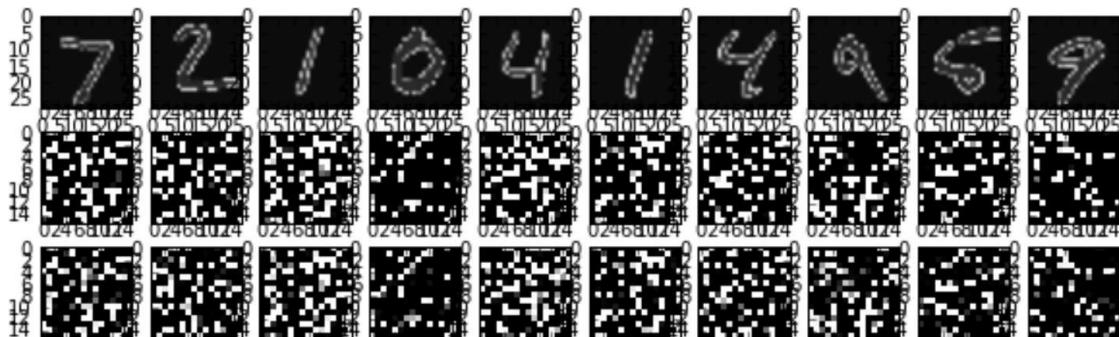


图10-16 自编码综合实例第2层结果

10. 第3层网络训练

现在开始训练第3层的网络，同理，这次的输入数据要经过前面两层网络的运算才可以生成。

代码10-6 自编码综合（续）

*****ebook converter DEMO Watermarks*****

```
156 with tf.Session() as sess:  
157     sess.run(tf.global_variables_initializer())  
158  
159     print ("开始训练")  
160     for epoch in range(epochs):  
161         num_batch = int(mnist.train.num_examples/batch_  
162         total_cost = 0.  
163         for i in range(num_batch):  
164             batch_xs, batch_ys = mnist.train.next_batch()  
165             l1_h = sess.run(l1_out, feed_dict={x: batch_  
166             dropout_keep_prob: 1.})  
167             l2_h = sess.run(l2_out, feed_dict={l2x: l1_h})  
168             l3cost = sess.run([l3_optm, l3_cost], feed_  
169             h, l3y: batch_ys})  
170             total_cost += l3cost  
171             # 输出cost  
172             if epoch % disp_step == 0:  
173                 print ("Epoch %02d/%02d average cost: %.6f"  
174                         % (epoch, epochs, total_cost/num_batch))  
175     print ("完成 layer_3 训练")
```

执行上面的代码，生成如下信息：

```
开始训练  
Epoch 00/50 average cost: 1.379495  
Epoch 10/50 average cost: 0.277589  
Epoch 20/50 average cost: 0.271069  
Epoch 30/50 average cost: 0.269604  
Epoch 40/50 average cost: 0.269904  
完成 layer_3 训练
```

11. 栈式自编码网络验证

这次我们暂时略过对第3层网络模型的单独验证，直接去验证整个分类模型，将MNIST数据输入进去，看看栈式自编码器的分类效果如何。

代码10-6 自编码综合（续）

```
175 # 测试 model  
176     correct_prediction = tf.equal(tf.argmax(pred, 1), t1  
177     # 计算准确率  
178     accuracy = tf.reduce_mean(tf.cast(correct_prediction,  
179     print ("Accuracy:", accuracy.eval({x: mnist.test.images,  
mnist.test.labels}))
```

执行上面的代码，生成如下信息：

```
Accuracy: 0.9213
```

可以看出，直接将每层的训练参数堆起来，网络会有很好的表现。为了进一步优化，来看看下面的步骤。

12. 级联微调

下面进入微调阶段，将网络模型联起来进行分类训练，这部分的测试代码与前面一样，所以这里只把训练部分的代码贴出来。

代码10-6 自编码综合（续）

```
180 with tf.Session() as sess:  
181     sess.run(tf.global_variables_initializer())  
182  
183     print ("开始训练")  
184     for epoch in range(epochs):  
185         num_batch  = int(mnist.train.num_examples/batch_  
186         total_cost = 0.
```

*****ebook converter DEMO Watermarks*****

```
187     for i in range(num_batch):
188         batch_xs, batch_ys = mnist.train.next_batch()
189
190         feeds = {x: batch_xs, l3y: batch_ys}
191         sess.run(optm3, feed_dict=feeds)
192         total_cost += sess.run(cost3, feed_dict=feeds)
193
194         # 输出cost
195         if epoch % disp_step == 0:
196             print ("Epoch %02d/%02d average cost: %.6f"
197                   % (epoch, epochs, total_cost/num_batch))
198
199     print ("完成 级联 训练")
```

执行上面的代码，生成如下信息：

```
开始训练
Epoch 00/50 average cost: 1.003439
Epoch 10/50 average cost: 0.035012
Epoch 20/50 average cost: 0.001034
Epoch 30/50 average cost: 0.000112
Epoch 40/50 average cost: 0.000039
完成 级联训练
```

可以看到，由于网络模型中各层的初始值已经训练好了，所以开始就是很低的错误率，错误率接着每次的迭代都有很大幅度的下降。到此这个例子就算是完成了，该例已经非常接近真实工作中的场景了，读者学习时在跟着做例子的同时更要理解所使用的方法，它可以将任何复杂的任务化简。



提示： 搭建网络模型时，层数过多，参数也会随之增加，所以逻辑一定要清楚，一个好的命名规范可以让你在这个问题上轻松许多。

10.8.2 实例81：添加模型存储支持分布训练

栈式自编码的另一个优点就是可以将神经网络模块化，便于分工，非常适合团队合作。在实际中，为了得到更好的模型，需要将上述的每个环节分别单独训练，由不同的小组或人员来分步工作。

小组或人员之间的对接则需要通过模型文件来完成，这就要求每个环节的参数都必须保存下来，在自己的步骤开始之前先把上个步骤的环境加载进去。

实例描述

对自编码模型进行分布式模型存储与载入，使每一层都可以单个环节逐一训练。

可以修改上述代码，在训练开始前定义模型保存参数。

代码10-7 分布自编码综合

```
01 savedir = "d:/python/log/"  
02 saver    = tf.train.Saver(max_to_keep=1) #保存一个模型  
03 load_epoch = 49  
04 print (savedir)  
05 with tf.Session() as sess:  
06     ....
```

在每个训练迭代之后都将模型保存起来。

代码10-7 分布自编码综合（续）

```
07 .....
08 if epoch % disp_step == 0:
09         print ("Epoch %02d/%02d average cost: %.6f"
10             % (epoch, epochs, total_cost/num_batch)
11 saver.save(sess, savedir + 'stacked_auto_encoder.ckpt',
12 step=epoch)
12     print ("完成 级联 训练")
```

在每次训练的开始将模型读入（第一个网络不需要读入）。

代码10-7 分布自编码综合（续）

```
13 with tf.Session() as sess:
14     sess.run(tf.global_variables_initializer())
15     saver.restore(sess, savedir +"stacked_auto_encoder.cl
16 str(load_epoch))
16 .....
```

 **注意：** 这里给出的代码是按照迭代保存的。这是一种好习惯，尤其在训练海量数据时，由于某种意外导致训练终止，使所有的训练结果丢失，还得需要花大量的时间重新训练。这种方法可以恢复到上一次迭代的结果。

加完这些代码之后，就可以将上面的代码放

在一个文件里，通过注释的方式，按照步骤一步一步往下进行了（每次只打开一个模型训练的代码）。

10.8.3 小心分布训练中的“坑”

如果读者是使用Anaconda在Windows下运行，那么这里有个“坑”需要填一下。当加上保存模型的功能之后，在一步一步训练过程中，如果运行两次读取模型则会报错。更奇怪的事情是，在保存模型过程中，如果你第一次运行一半停止了，第二次运行成功并且也生成了模型，但是当你从模型里载入时，会是错误的参数。

原因是，在Anaconda中的py程序默认都是在同一个图中运行的，即除非关掉Anaconda，否则运行两次时，这两次的代码是在一个图中，那么会有什么影响呢？

在定义变量时，在同一个图中，同一句代码tf.Variable所生成的变量是不同的名字，例如在代码中添加打印信息，输出变量的名字：

```
# 权重
weights = {
    #网络1 784-256-784
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1]),
.....)
    print (weights['h1'].name)
```

执行上面的代码，生成如下信息：

```
Variable:0  
Variable:14
```

这时会发现，运行两次weights[h1]有了两个不同的名字，在内存里会有两套变量。所以当运行两次后保存模型时，其实是保存了两套，但是读取时只是读取了第一套。同理。当运行两次读取时，第一次的模型可以与变量对应上，第二次会新生成另外一套变量，并且模型里找对应关系，但因为找不到所以就报错了。

解决方法：只需要在变量定义之前加上下面一行代码。让所有的图环境重置，即可解决问题。

```
tf.reset_default_graph()
```

10.8.4 练习题

使用栈式自编码对MNIST数据集逐层训练压缩特征，直到压缩成二维数据（如512、256、128、64、32、16、2），再将其图示出来。

10.9 变分自编码

前面所描述的自编码可以降维重构样本，在这个基础上我们来学习一个更强大的自编码网络。

10.9.1 什么是变分自编码

变分自编码学习的不再是样本的个体，而是学习样本的规律。这样训练出来的自编码不单具有重构样本的功能，还具有仿照样本的功能。

听起来这么强大的功能，到底是怎么做到的呢？下面我们来讲讲它的原理。

变分自编码，其实就是在编码过程中改变了样本的分布（“变分”可以理解为改变分布）。前面所说的“学习样本的规律”，具体指的就是样本的分布，假设我们知道样本的分布函数，就可以从这个函数中随便取一个样本，然后进行网络解码层前向传导，这样就可以生成一个新的样本。

为了得到这个样本的分布函数，模型训练的目的将不再是样本本身，而是通过加一个约束项，将网络生成一个服从于高斯分布的数据集，这样按照高斯分布里的均值和方差规则就可以任意取相关的数据，然后通过解码层还原成样本。

10.9.2 实例82：使用变分自编码模拟生成MNIST数据

对于变分自编码，好多文献都给出了一堆晦涩难懂的公式，其实里面真正的公式只有一个——KL离散度的计算。而它也属于成熟的式子，就跟交叉熵一样，直接拿来用就可以。

公式本来是语言的高度概括，而如果一篇文章全是公式没有语言就会令人难以理解，本节会通过代码加上语言描述，让这部分知识学习起来不会感觉晦涩难懂。

本例共分如下几个步骤，下面就来一一操作。

实例描述

使用变分自编码模型进行模拟MNIST数据的生成。

1. 引入库，定义占位符

本例建立的网络与之前略有不同，编码器为两个全连接层，第一个全连接层由784个维度的输入变化256个维度的输出；第二个全连接层并列连接了两个输出网络（mean与lg_var），每个网络都输出了两个维度的输出。然后将两个输出通过

一个公式的计算，输入到以一个2节点为开始的解码部分，接着后面为两个全连接层的解码器，第一层由两个维度的输入到256个维度的输出，第二层由256个维度的输入到784个维度的输出，如图10-17所示。

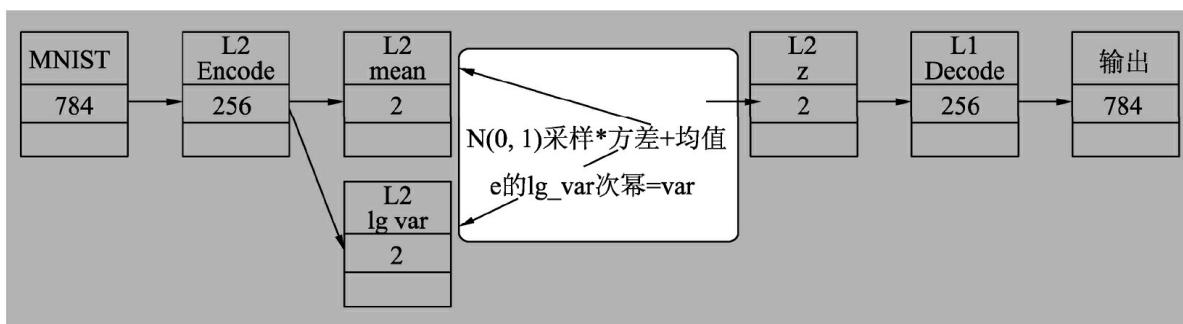


图10-17 变分自编码器层次

具体的计算公式，后面会有详细介绍。

在下面的代码中与前面代码不同，引入了一个scipy库，在后面可视化时会用到。头文件引入之后，定义操作符x和z。x用于原始的图片输入，z用于中间节点解码器的输入。

代码10-8 变分自编码

```

01 import numpy as np
02 import tensorflow as tf
03 import matplotlib.pyplot as plt
04 from scipy.stats import norm
05
06 from tensorflow.examples.tutorials.mnist import input_data
07 mnist = input_data.read_data_sets("/data/") #, one_hot=True
08
09 n_input = 784
10 n_hidden_1 = 256

```

*****ebook converter DEMO Watermarks*****

```
11 n_hidden_2 = 2
12
13 x = tf.placeholder(tf.float32, [None, n_input])
14
15 zinput = tf.placeholder(tf.float32, [None, n_hidden_2])
```

zinput是个占位符，在后面要通过它输入分布数据，用来生成模拟样本数据。

2. 定义学习参数

由于这次的网络结构不同，所以定义的参数也有变化，mean_w1与mean_b1是生成mean的权重，log_sigma_w1与log_sigma_b1是生成log_sigma的权重。

代码10-8 变分自编码（续）

```
16 weights = {
17
18     'w1': tf.Variable(tf.truncated_normal([n_input, n_hi
19                             stddev=0.001)),
20     'b1': tf.Variable(tf.zeros([n_hidden_1])), 
21
22     'mean_w1': tf.Variable(tf.truncated_normal([n_hidden_
23                             stddev=0.001)),
24     'log_sigma_w1': tf.Variable(tf.truncated_normal([n_h
hidden_2],
25                             stddev=0.001)),
26
27     'w2': tf.Variable(tf.truncated_normal([n_hidden_2, n_
28                             stddev=0.001)),
29
30     'b2': tf.Variable(tf.zeros([n_hidden_1])),
31     'w3': tf.Variable(tf.truncated_normal([n_hidden_1, n_
32                             stddev=0.001)),
33
34     'b3': tf.Variable(tf.zeros([n_input])),
```

*****ebook converter DEMO Watermarks*****

```
35  
36     'mean_b1': tf.Variable(tf.zeros([n_hidden_2])),  
37  
38     'log_sigma_b1': tf.Variable(tf.zeros([n_hidden_2]))  
39 }
```



注意： 这里初始化w的权重与以往不同，使用了很小的值（方差为0.001的truncated_normal）。这里设置得非常小心，由于在算KL离散度时计算的是与标准高斯分布的距离，如果网络初始生成的模型均值方差都很大，那么与标准高斯分布的距离就会非常大，这样会导致模型训练不出来，生成NAN的情况。

3. 定义网络结构

按照图10-17的描述，网络节点可以按照以下代码来定义，在变分解码器中为训练的中间节点赋予了特殊的意义，让它们代表均值和方差，并将它们所代表的数据集向着标准高斯分布数据集靠近（也就是原始数据是样本，高斯分布数据是标签），然后可以使用KL离散度公式，来计算它所代表的集合与标准的高斯分布集合（均值是0，方差为1的正态分布）间的距离，将这个距离当成误差，让它最小化从而优化网络参数。

这里的方差节点不是真正意义的方差，是取了 \log 之后的，所以会有 $\text{tf.exp}(\text{z_log_sigma_sq})$ 的变换，是取得方差的值，再通过 tf.sqrt 将其开平

方得到标准差。用符合标准正太分布的一个数乘以标准差加上均值，就使这个数成为符合(z_mean, sigma)数据分布集合里的一个点(z_mean是指网络生成均值, sigma是指网络生成的z_log_sigma_sq变换后的值)。



注意：输出的值当成任意一个意义，并通过训练得到对应的关系。具体做法为：将具有代表该意义的值代入相应的公式（该公式要求必须能够支持反向传播），计算公式输出值与目标值的误差，并将误差放到优化器里，然后通过多次迭代的方式进行训练即可。

到此，完成了编码阶段。将原始数据编码输出3个值：

- 一个是表示该数据分布的均值。
- 一个是表示该数据分布的方差。
- 还有一个是得到了该数据分布中的一个实际的点z。



注意：这里的变换对应的知识点是：假如一个符合高斯分布的数据集均值、标准差为(m, sigma)，其里面的某个点，可以通过一个符合标准高斯分布(0, 1)中的点x，通过

$m + x \times \sigma$ 的方式转化而成。

但这是在一个假设背景下完成的，假设数据分布属于高斯分布。我们现在无法保证转换后的数据分布符合高斯分布，则可以通过测量输出值代表的数据集与标准高斯分布数据集之间的差距，利用神经网络来将其训练成符合高斯分布的数据集。

代码10-8 变分自编码（续）

```
40 h1=tf.nn.relu(tf.add(tf.matmul(x, weights['w1']), weights['b1']))
41 z_mean = tf.add(tf.matmul(h1, weights['mean_w1']), weights['mean_b1'])
42 z_log_sigma_sq = tf.add(tf.matmul(h1, weights['log_sigma_w1']), weights['log_sigma_b1'])
43
44 # 高斯分布样本
45 eps = tf.random_normal(tf.stack([tf.shape(h1)[0], n_hidden]), 0, 1, dtype = tf.float32)
46 z =tf.add(z_mean, tf.multiply(tf.sqrt(tf.exp(z_log_sigma_sq)), eps))
47 h2=tf.nn.relu( tf.matmul(z, weights['w2'])+ weights['b2'])
48 reconstruction = tf.matmul(h2, weights['w3'])+ weights['b3']
49
50 h2out=tf.nn.relu( tf.matmul(zinput, weights['w2'])+ weights['b2'])
51 reconstructionout = tf.matmul(h2out, weights['w3'])+ weights['b3']
```

得到了符合原数据集上的一个具体点 z 之后，就可以通过神经网络这个点 z 还原成原始数据 $reconstruction$ 了。解码部分和前面一样，参照编码的网络逐层还原回去即可。

$h2out$ 和 $reconstructionout$ 两个节点不属于训练中的结构，是为了生成指定数据时用的。

4. 构建模型的反向传播

这一步和前面一样，需要定义损失函数的节点和优化算法的OP，代码如下。

代码10-8 变分自编码（续）

```
52 # 计算重建loss
53 reconstr_loss = 0.5 * tf.reduce_sum(tf.pow(tf.subtract(reconstr, x), 2.0))
54 latent_loss = -0.5 * tf.reduce_sum(1 + z_log_sigma_sq
55                                     - tf.square(z_mean)
56                                     - tf.exp(z_log_sigma_sq))
57 cost = tf.reduce_mean(reconstr_loss + latent_loss)
58
59 optimizer = tf.train.AdamOptimizer(learning_rate = 0.001).minimize(cost)
```

上面代码描述了网络的两个优化方向：

- 一个是比较生成的数据分布与标准高斯分布的距离，这里使用KL离散度的公式（见latent_loss）。

- 另一个是计算生成数据与原始数据间的损失，这里用的是平方差，也可以用交叉熵。

最后将两种损失值放在一起，通过Adam的随机梯度下降算法实现在训练中的优化参数。

5. 设置参数，进行训练

这一步与前面类似，设置训练参数，迭代50次，在session中每次循环取指定批次数据进行训练。

代码10-8 变分自编码（续）

```
60 training_epochs = 50
61 batch_size = 128
62 display_step = 3
63
64 with tf.Session() as sess:
65     sess.run(tf.global_variables_initializer())
66
67     for epoch in range(training_epochs):
68         avg_cost = 0.
69         total_batch = int(mnist.train.num_examples/batch_
70
71         # 遍历全部数据集
72         for i in range(total_batch):
73             batch_xs, batch_ys = mnist.train.next_batch(1
74
75             # 输入数据，运行优化器
76             _,c = sess.run([optimizer,cost], feed_dict={}
77
78             # 显示训练中的详细信息
79             if epoch % display_step == 0:
80                 print("Epoch:", '%04d' % (epoch + 1), "cost="
format(c))
81
82             print("完成!")
83
84             # 测试
85             print ("Result:", cost.eval({x: mnist.test.images}))
```

可视化部分这里不再详述，读者可以参考本书的配套代码，最终程序运行的结果输出如下，输出图片如图10-18所示。

```
Epoch: 0001 cost= 2766.966308594
Epoch: 0004 cost= 2503.895507812
Epoch: 0007 cost= 2177.547363281
Epoch: 0010 cost= 2221.667724609
Epoch: 0013 cost= 2110.643798828
Epoch: 0016 cost= 2103.255859375
Epoch: 0019 cost= 2258.502685547
Epoch: 0022 cost= 2231.131347656
Epoch: 0025 cost= 2092.596191406
Epoch: 0028 cost= 2018.563964844
Epoch: 0031 cost= 1993.950439453
Epoch: 0034 cost= 2091.635253906
Epoch: 0037 cost= 1992.461059570
Epoch: 0040 cost= 2018.574462891
Epoch: 0043 cost= 1992.727661133
Epoch: 0046 cost= 2056.166503906
Epoch: 0049 cost= 1939.133544922
完成!
Result: 156414.0
```

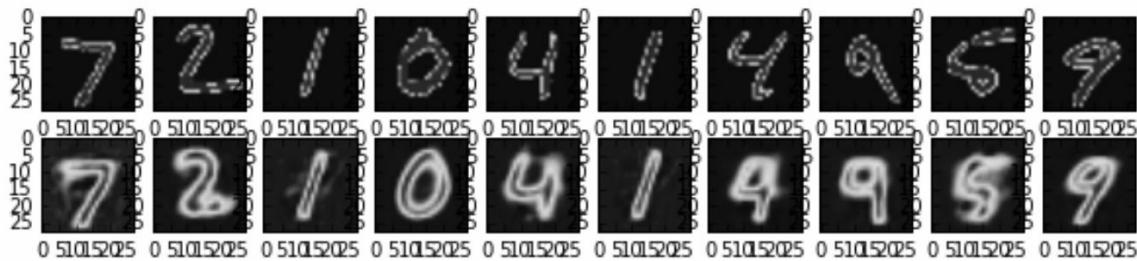


图10-18 变分自编码结果

图10-18中，第一行代表原始的样本图片，第二行代表变分自编码重建后生成的图片。可以看到生成的数字中不再一味单纯地学习形状，而是通过数据分布的方式学习规则，对原有图片具有更清晰的修正功能。

仿照前面的可视化代码，将均值和方差代表的二维数据在直角坐标系中展现如下，如图10-19所示。

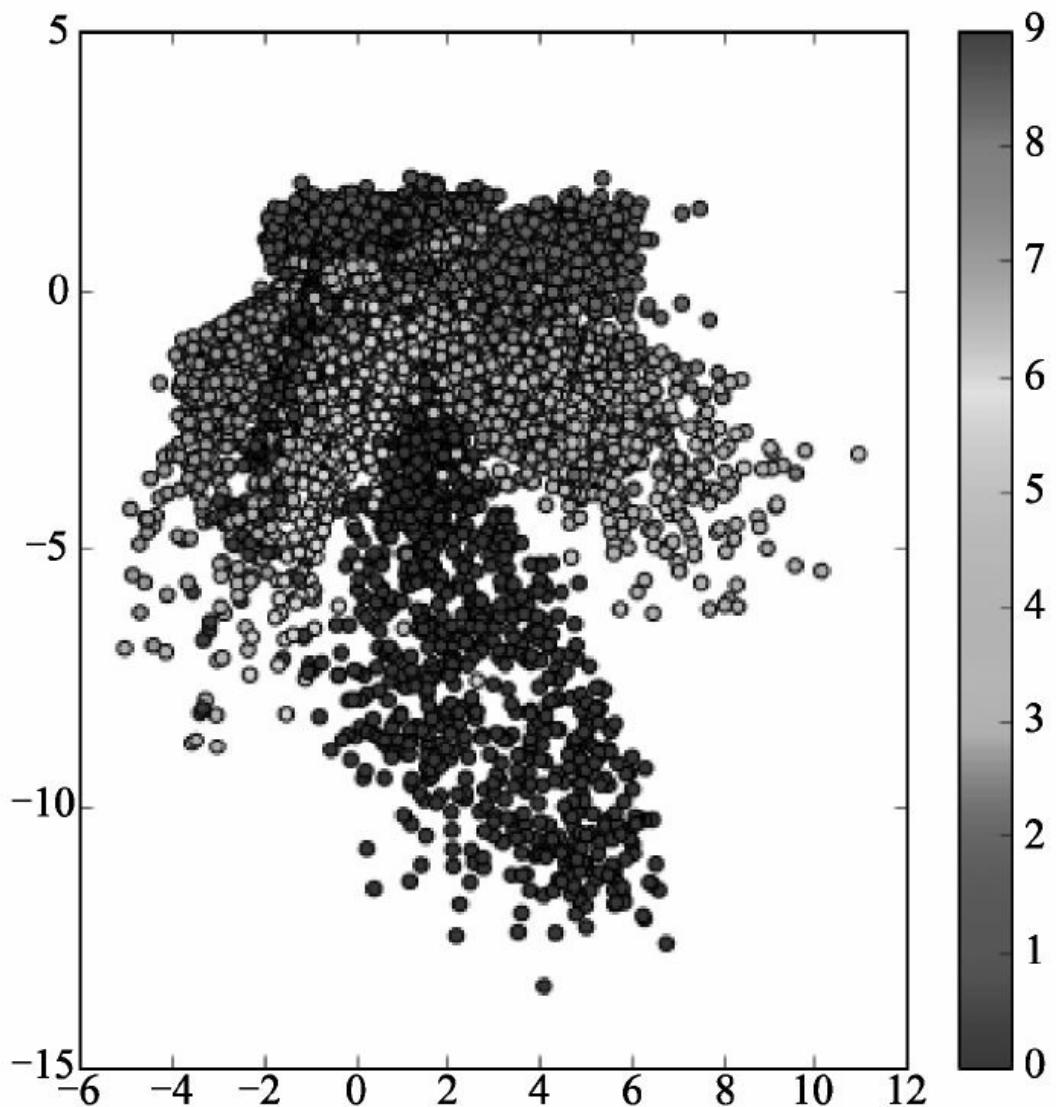


图10-19 变分自编码二维可视化

从图10-19中可以看出，MNIST数据集中同一类样本的特征分布还是比较集中的，说明变分自解码也具有降维功能，也可以用它进行分类任务的数据降维预处理。

6. 高斯分布取样，生成模拟数据

为了进一步证实模型学到了数据分布的情况，这次在高斯分布抽样中取一些点，将其映射

*****ebook converter DEMO Watermarks*****

到模型中的z，然后通过解码部分还原成真实图片看看效果，代码如下。



注意： 代码中norm.ppf函数的作用是从按照百分比由大到小排列后的标准高斯分布中取值。np.linspace (0.05, 0.95, n) 的意思是，将整个高斯分布数据集从大到小排列，取出前0.05%到0.95%区间，并且分成n份，每份对应的点的具体数值。

norm代表标准高斯分布，ppf代表累积分布函数的反函数。累积分布的意思是，在一个结合里所有小于a的值出现的概率的和。例如， $x=ppf(0.05)$ 就代表在集合里有个x，集合中每个小于x的数在集合里出现的概率的总和等于0.05。

代码10-8 变分自编码（续）

```
86     n = 15 # 15×15 的figure
87     digit_size = 28
88     figure = np.zeros((digit_size * n, digit_size * n))
89     grid_x = norm.ppf(np.linspace(0.05, 0.95, n))
90     grid_y = norm.ppf(np.linspace(0.05, 0.95, n))
91
92     for i, yi in enumerate(grid_x):
93         for j, xi in enumerate(grid_y):
94             z_sample = np.array([[xi, yi]])
95             x_decoded = sess.run(reconstructionout, feed_
96             sample})
97
98             digit = x_decoded[0].reshape(digit_size, digi:
99                                         figure[i * digit_size:(i + 1) * digit_size,
```

*****ebook converter DEMO Watermarks*****

```
99                     j * digit_size: (j + 1) * digit_size]
100
101     plt.figure(figsize=(10, 10))
102     plt.imshow(figure, cmap='Greys_r')
103     plt.show()
```

运行以上代码，生成图片如图10-20所示。

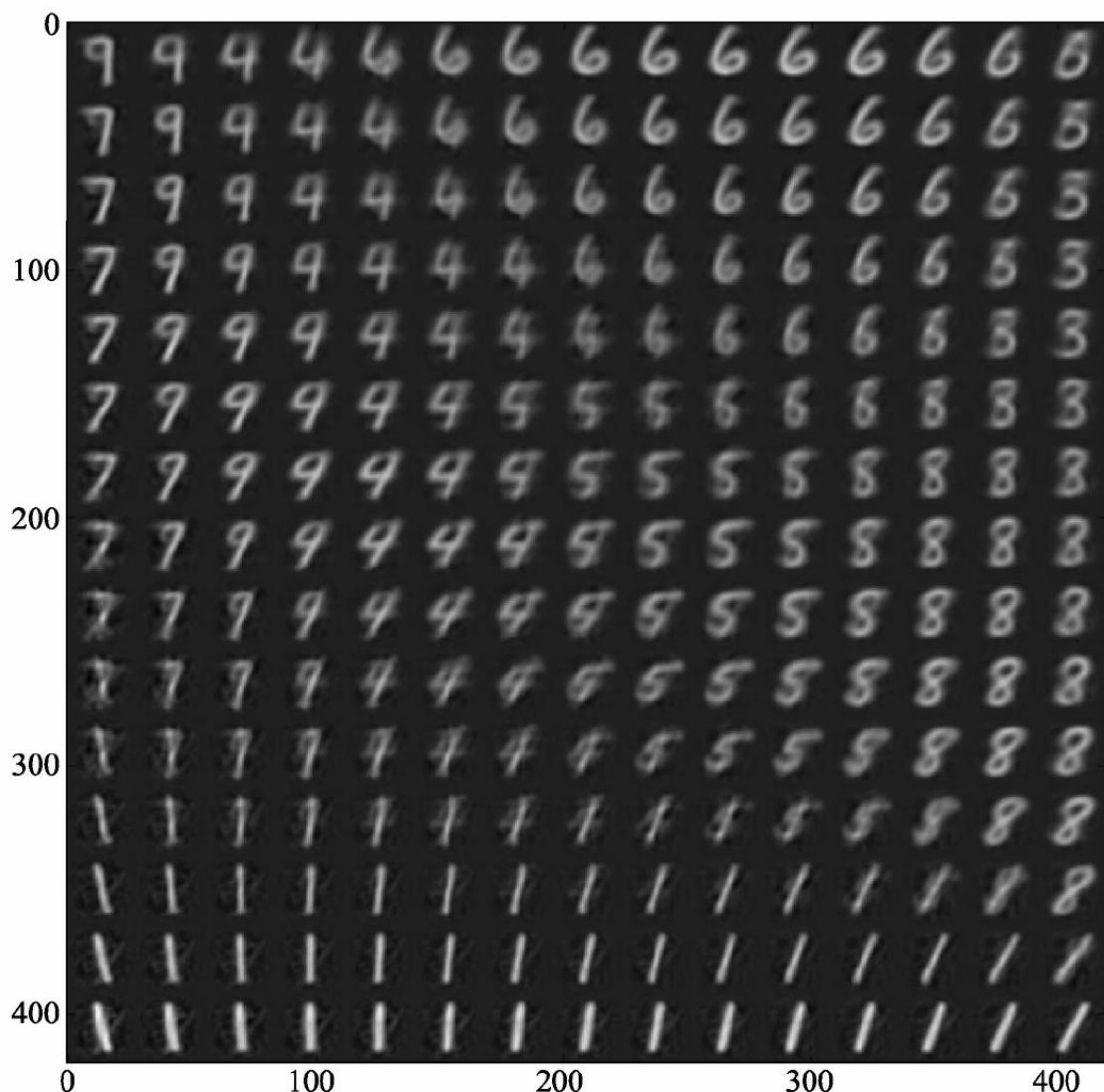


图10-20 变分自编码生成模拟数据

可以看到，在神经网络的世界里，从左下角到右上角显示了网络是按照图片的形状变化而排列的，并不像人类一样，把数字按照1到9的顺序排列，因为机器学的只是图片，而人类对数字的理解更多的是其代表的意思。

10.9.3 练习题

读者可以自己试试在初始化时将所有的w设为0和设为truncated_normal，不指定stddev的效果，想想为什么？

10.10 条件变分自编码

前面的变分自编码是为了本节条件变分自编码器做铺垫的，在实际中条件变分自编码的应用更广泛一些，下面来介绍条件变分自编码器。

10.10.1 什么是条件变分自编码

变分自编码存在一个问题——虽然可以生成一个样本，但只能输出与输入图片相同类别的样本。虽然也可以随机从符合模型生成的高斯分布中取数据来还原成样本，但是这样的话我们并不知道生成的样本属于哪个类别。条件变分解码则可以解决这个问题，让网络按指定的类别生成样本。

在变分自编码的基础上，再去理解条件变分自编码会很容易。主要的改动是，在训练、测试时加入一个one-hot向量，用于表示标签向量。其实，就是给变分自编码网络加入了一个条件，让网络学习图片分布时加入了标签因素，这样可以按照标签的数值生成指定的图片。

10.10.2 实例83：使用标签指导变分自编码网络生成MNIST数据

了解完原理，再来介绍下具体做法。在编码阶段需要在输入端添加标签对应的特征，在解码阶段同样也需要将标签加入输入，这样，再解码的结果向原始的输入样本不断逼近，最终得到的模型将会把输入的标签特征当成MNIST数据的一部分，从而实现通过标签生成MNIST数据。

在输入端添加标签时，一般是通过一个全连接层的变换将得到的结果使用contact函数连接到原始输入的地方，在解码阶段也将标签作为样本输入，与高斯分布的随机值一并运算，生成模拟样本。条件变为自编码结构如图10-21所示。

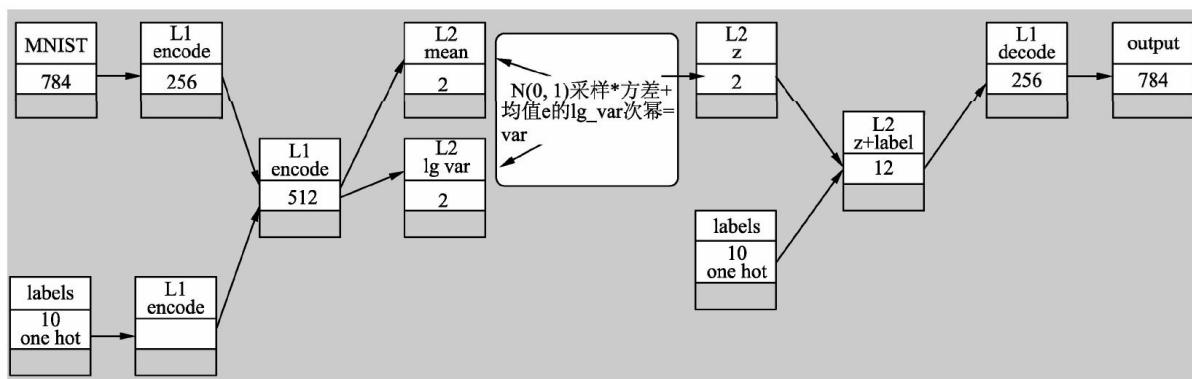


图10-21 条件变分自编码结构

具体代码步骤如下。

实例描述

使用条件变分自编码模型，通过指定标签输入生成对应类别的MNIST模拟数据。

1. 添加标签占位符

在“10-8变分自编码.py”代码基础上修改，添加占位符y。

代码10-9 条件变分自编码

```
01 .....
02 y = tf.placeholder(tf.float32, [None, n_labels])
03 .....
```

2. 添加输入全连接权重

添加全连接层的权重'wlab1'与'blab1'，作为输入标签的特征转换。这里输入的标签也转换成256个输出，因为最终要连接到原始的图片全连接输出，所以到第二层全连接时，输入就变成了 256×2 ，因此也需要将mean_w1和log_sigma_w1的输入修改成 $n_hidden_1 \times 2$ 。

代码10-9 条件变分自编码（续）

```
04 weights = {
05
06     'w1': tf.Variable(tf.truncated_normal([n_input, n_hi
07                                         stddev=0.001)),
08     'b1': tf.Variable(tf.zeros([n_hidden_1])),
09
10     'wlab1': tf.Variable(tf.truncated_normal([n_labels, r
11                                         stddev=0.001]),
12     'blab1': tf.Variable(tf.zeros([n_hidden_1])),
13     'mean_w1': tf.Variable(tf.truncated_normal([n_hidden_
hidden_2]),
```

```
14                                         stddev=0.001)),  
15     'log_sigma_w1': tf.Variable(tf.truncated_normal([n_h:  
hidden_2],  
16                                         stddev=0.001)),  
17     'w2': tf.Variable(tf.truncated_normal([n_hidden_2+n_  
hidden_1],  
18                                         stddev=0.001)),  
19     ....
```

同样，对于生成的z也要与label连接后输入加
码器，所以w2的输入维度需要被改成
`n_hidden_2+n_labels`。

3. 修改模型，将标签输出接入编码

定义新节点`hlab1`为输入标签的输出，接着使
用`concat`函数将它与原来的`h1`合并到一起，变成
`hall1`。此时，`hall1`的`shape`为`[batch_size,
n_hidden_1×2]`。接着，将合成好的`hall1`代替原
来的`h1`输入`z_mean`与`z_log_sigma_sq`中。

代码10-9 条件变分自编码（续）

```
20 h1=tf.nn.relu(tf.add(tf.matmul(x, weights['w1']), weights[  
21 'mean_b1']))  
22 hlab1=tf.nn.relu(tf.add(tf.matmul(y, weights['wlab1']), w  
['blab1']))  
23  
24 hall1= tf.concat([h1,hlab1],1)#256*2  
25  
26 z_mean = tf.add(tf.matmul(hall1, weights['mean_w1']), we  
['mean_b1'])  
27 z_log_sigma_sq = tf.add(tf.matmul(hall1, weights['log_si  
weights['log_sigma_b1']])
```

4. 修改模型将标签接入解码

这一步里中间的z不用变化，在z之后同样连接上y的特征，一起输入到解码器中。这里需要同时修改reconstruction和reconstructionout节点，一个用来训练，一个用来生成。

代码10-9 条件变分自编码（续）

```
28 .....
29 zall=tf.concat([z,y],1)
30 h2=tf.nn.relu( tf.matmul(zall, weights['w2'])+ weights['t
31 reconstruction = tf.matmul(h2, weights['w3'])+ weights['t
32
33 zinputall = tf.concat([zinput,y],1)
34 h2out=tf.nn.relu( tf.matmul(zinputall, weights['w2'])+ we
35 reconstructionout = tf.matmul(h2out, weights['w3'])+ wei
```

5. 修改session中的feed部分

优化器不用变化，直接修改session的feed部分即可，在feed中加入标签占位符及对应的数据。

代码10-9 条件变分自编码（续）

```
36 with tf.Session() as sess:
37     sess.run(tf.global_variables_initializer())
38
39     for epoch in range(training_epochs):
40         avg_cost = 0.
41         total_batch = int(mnist.train.num_examples/batch_
42
43         # 遍历全部数据集
```

*****ebook converter DEMO Watermarks*****

```
44         for i in range(total_batch):
45             batch_xs, batch_ys = mnist.train.next_batch(t
46
47             # 输入数据，运行优化器
48             _,c = sess.run([optimizer,cost], feed_dict={y:batch_ys})
49
50             # 显示训练中的详细信息
51             if epoch % display_step == 0:
52                 print("Epoch:", '%04d' % (epoch + 1), "cost='
53                 format(c))
54
55             print("完成!")
56
57             # 测试
58             print ("Result:", cost.eval({x: mnist.test.images,y:r
59             test.labels}))
```

6. 运行模型生成模拟数据

这一步是最有意思的部分了。随意生成一个高斯分布随机数，并通过指定的one_hot输入标签，就可以命令模型生成指定的MNIST图片数据了。

代码10-9 条件变分自编码（续）

```
58 .....
59     # 根据label模拟生产图片可视化结果
60     show_num = 10
61     z_sample = np.random.randn(10,2)
62
63     pred = sess.run(
64         reconstructionout, feed_dict={zinput:z_sample,y:
65         labels[:show_num]})
```

```
66     f, a = plt.subplots(2, 10, figsize=(10, 2))
67     for i in range(show_num):
68         a[0][i].imshow(np.reshape(mnist.test.images[i], (
69             a[1][i].imshow(np.reshape(pred[i], (28, 28))))
```

*****ebook converter DEMO Watermarks*****

上面代码取了10个测试样本数据，将样本数据的label随高斯分布值z_sample一起生成了模拟的MNIST数据。运行代码，生成如图10-22和图10-23所示的数据。



图10-22 根据原数据生成模拟数据

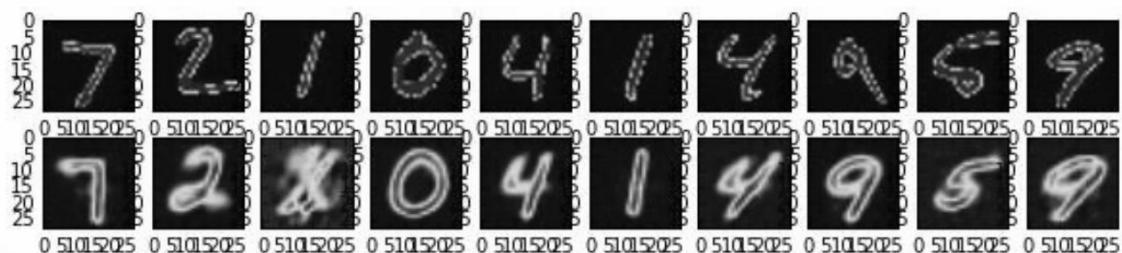


图10-23 根据标签生成模拟数据

图10-22为根据原始图片生成的自编码数据，第一行为原始数据，第二行为自编码数据。

图10-23为根据label生成的模拟数据，第一行为label对应的原始数据，第二行为解码器生成的模拟数据。

比较两幅图片可以看出，使用原图生成的自编码数据还会带有一些原来的样子，而以标签生
*****ebook converter DEMO Watermarks*****

成的解码数据，已经彻底地学会了数据的分布，并生成截然不同却带有相同意义的数据。

*****ebook converter DEMO Watermarks*****

第3篇 深度学习进阶

本篇是对基础网络模型的灵活运用与自由组合，是对前面知识的综合及拔高。本篇内容包括：

第11章 深度神经网络

第12章 对抗神经网络 (GAN)

第11章 深度神经网络

本章开始学习深度神经网络的知识。作为深度学习的代表，深度神经网络可以算是深度学习中最主要的知识。前面讲了许多网络形态，都会在各自的领域中有一定的效果，但是要体现出真正的人工智能能力，就必须将这些网络形态组合起来，利用各种网络的优势，使整体效果达到最优，实现可以匹配人工智能的要求。

本章含有教学视频共10分25秒。

作者按照本章的内容结构，对主要内容进行了讲解，包括深度神经网络、实物检测相关模型、slim、Object Detection API及预训练模型等相关知识（重点为掌握深度神经网络模型的使用及训练方法）。



*****ebook converter DEMO Watermarks*****

11.1 深度神经网络介绍

本节主要介绍深度神经网络。先从深度神经网络的起源说起，接着介绍在深度神经网络中都有哪些知名的经典模型及各自的特点。

11.1.1 深度神经网络起源

深度学习的兴起源于深度神经网络的崛起。2012年，由 Alex Krizhevsky开发的一个深度学习模型AlexNet，赢得了视觉领域竞赛ILSVRC 2012的冠军，并且效果大幅度超过传统的方法。在百万量级的ImageNet数据集合上，识别率从传统的70%多提升到80%多，将深度学习正式推上了舞台。之后ILSVRC每年都不断被深度学习刷榜，并且模型变得越来越深，错误率也越来越低，目前已经降到了3.5%左右，而在同样的ImageNet数据集合上，人眼的辨识错误率大概在5.1%，也就是说目前的深度学习模型的识别能力已经超过了人眼。

自从2012年之后，在ILSVRC竞赛中获得冠军的模型如下。

- 2012年： AlexNet；
- 2013年： VGG；

- 2014年： GoogLeNet；
- 2015年： ResNet；
- 2016年： Inception-ResNet-v2。

随着深度神经网络学科的进步，使用神经网络征服ImageNet的门槛已经越来越低，于是在2017年，ILSVRC竞赛举办完最后一届，宣布了停办。与此同时，在2017年的ICCV竞赛中，在物体检测、物体分割等细分领域的冠军中出现了多家中国企业的名字，这表明中国的人工智能技术正在逐步地引领全球。

11.1.2 经典模型的特点介绍

下面具体介绍各界冠军模型的特点。

1. VGG模型

VGG又分为VGG16和VGG19，分别在AlexNet的基础上将层数增加到16和19层，它除了在识别方面很优秀之外，对图像的目标检测也有很好的识别效果，是目标检测领域的较早期模型。

2. GoogLeNet模型

GoogLeNet除了层数加深到22层以外，主要的创新在于它的Inception，这是一种网中网（Network In Network）的结构，即原来的节点也是一个网络。用了Inception之后整个网络结构的宽度和深度都可扩大，能够带来2到3倍的性能提升。

3. ResNet模型

ResNet直接将深度拉到了152层，其主要的创新在于残差网络，其实这个网络的提出本质上是要解决层次比较深时无法训练的问题。这种借鉴了Highway Network思想的网络，相当于旁边专门开个通道使得输入可以直达输出，而优化的目标由原来的拟合输出 $H(x)$ 变成输出和输入的差 $H(x) - x$ ，其中 $H(x)$ 是某一层原始的期望映射输出， x 是输入。

4. Inception-ResNet-v2模型

Inception-ResNet-v2：是目前比较新的经典模型，将深度和宽带融合到一起，在当下ILSVRC图像分类基准测试中实现了最好的成绩，是将Inception v3与ResNet结合而成的。

接下来主要对当前比较前沿的GoogLeNet、ResNet、Inception-ResNet-v2几种网络结构进行详细介绍。

11.2 GoogLeNet模型介绍

前面已经介绍过GoogLeNet，其中最核心的亮点就是它的Inception，GoogLeNet网络最大的特点就是去除了最后的全连接层，用全局平均池化层（即使用与图片尺寸相同的过滤器来做平均池化）来取代它。

这么做的原因是：在以往的AlexNet和VGGNet网络中，全连接层几乎占据90%的参数量，占用了过多的运算量内存使用率，而且还会引起过拟合。

GoogLeNet的做法是去除全连接层，使得模型训练更快并且减轻了过拟合。

之后GoogLeNet的Inception还在继续发展，目前已经有v2、v3和v4版本，主要针对解决深层网络的以下3个问题产生的。

- 参数太多，容易过拟合，训练数据集有限。
- 网络越大计算复杂度越大，难以应用。
- 网络越深，梯度越往后传越容易消失（梯度弥散），难以优化模型。

Inception的核心思想是通过增加网络深度和宽度的同时减少参数的方法来解决问题。

Inception v1有22层深，比AlexNet的8层或者VGGNet的19层更深。但其计算量只有15亿次浮点运算，同时只有500万的参数量，仅为AlexNet参数量（6000万）的1/12，却有着更高的准确率。

下面沿着Inception的进化来一步步了解Inception网络。Inception是在一些突破性的研究成果之上推出的，所以有必要从Inception的前身理论开始介绍。下面先介绍MLP卷积层。

11.2.1 MLP卷积层

MLP卷积层（Mlpconv）源于2014年ICLR的一篇论文《Network In Network》。它改进了传统的CNN网络，在效果等同的情况下，参数只是原有的Alexnet网络参数的1/10。

卷积层要提升表达能力，主要依靠增加输出通道数，每一个输出通道对应一个滤波器，同一个滤波器共享参数只能提取一类特征，因此一个输出通道只能做一种特征处理。所以在传统的CNN中会使用尽量多的滤波器，把原样本中尽可能多的潜在的特征提取出来，然后再通过池化和大量的线性变化在其中筛选出需要的特征。这样的代价就是参数太多，运算太慢，而且很容易引

起过拟合。

MLP卷积层的思想是将CNN高维度特征转成低维度特征，将神经网络的思想融合在具体的卷积操作当中。直白的理解就是在网络中再放一个网络，即，使每个卷积的通道中包含一个微型的多层网络，用一个网络来代替原来具体的卷积运算过程（卷积核的每个值与样本对应的像素点相乘，再将相乘后的所有结果加在一起生成新的像素点的过程）。其结构如图11-1所示。

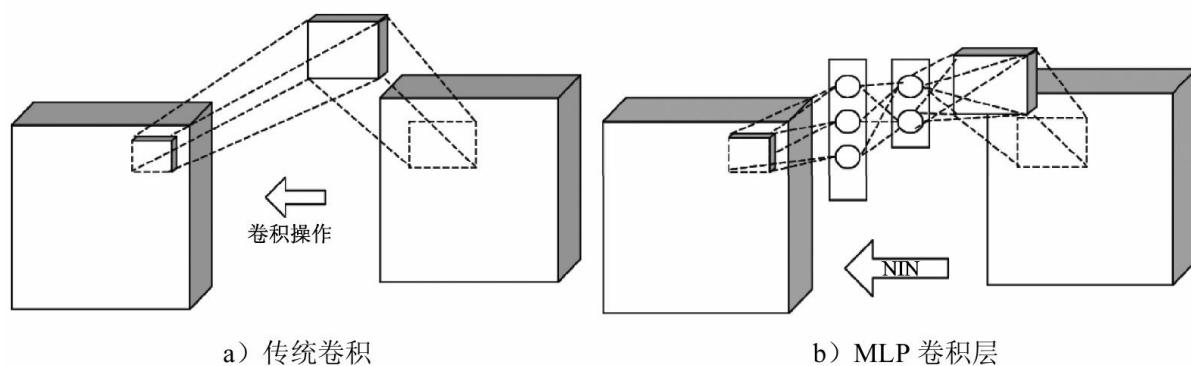


图11-1 MLP结构

图11-1中a为传统的卷积结构，图11-1b为MLP结构。相比较而言，利用多层次MLP的微型网络，对每个局部感受野的神经元进行更加复杂的运算，而以前的卷积层，局部感受野的运算仅仅只是一个单层的神经网络。在MLP网络中比较常见的是使用一个三层的全连接网络结构，这等效于普通卷积层后再连接1：1的卷积和ReLU激活函数。

11.2.2 全局均值池化

在11.2节开始时已经提到过全局均值池化的方法，就是在平均池化层中使用同等大小的过滤器将其特征保存下来。这种结构用来代替深层网络结构最后的全连接输出层。这个方法也是《Network In Network》论文中所论述的。

全局均值池化的具体用法是在卷积处理之后，对每个特征图的整张图片进行全局均值池化，生成一个值，即每张特征图相当于一个输出特征，这个特征就表示了我们输出类的特征。例如，在做1000个分类任务时，最后一层的特征图个数就要选择1000，就可以直接得出分类了。

在《Network In Network》论文中作者利用其进行1000物体分类问题，最后设计了一个4层的NIN+全局均值池化网络，如图11-2所示。

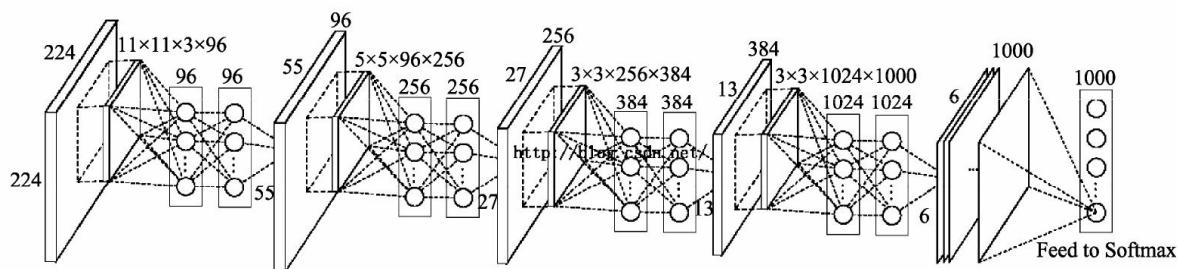


图11-2 NIN+全局均值池化

11.2.3 Inception 原始模型

Inception的原始模型是相对于MLP卷积层更为稀疏，它采用了MLP卷积层的思想，将中间的全连接层换成了多通道卷积层。Inception与MLP卷积在网络中的作用一样，把封装好的Inception作为一个卷积单元，堆积起来形成了原始的GoogLeNet网络。

Inception的结构是将 1×1 、 3×3 、 5×5 的卷积核对应的卷积操作和 3×3 的滤波器对应的池化操作堆叠在一起，一方面增加了网络的宽度，另一方面增加了网络对尺度的适应性，如图11-3所示。

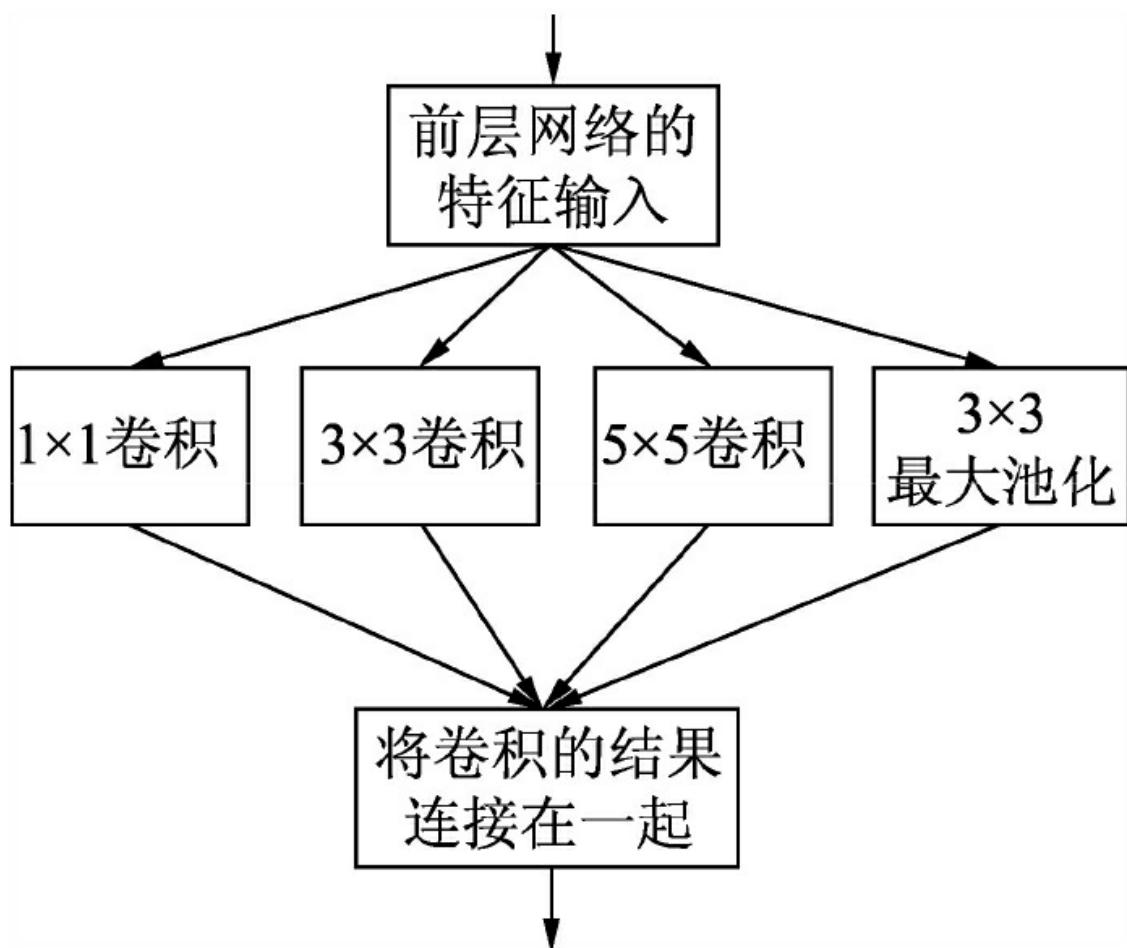


图11-3 Inception模型

Inception模型中包含了3种不同尺寸的卷积和一个最大池化，增加了网络对不同尺度的适应性，这和Multi-Scale的思想类似。早期计算机视觉的研究中，受灵长类神经视觉系统的启发，Serre使用不同尺寸的Gabor滤波器处理不同尺寸的图片，Inception v1借鉴了这种思想。Inception v1的论文中指出，Inception模型可以让网络的深度和宽度高效率地扩充，提升了准确率且不致于过拟合。

形象的解释就是Inception模型本身如同大网络中的一个小网络，其结构可以反复堆叠在一起形成更大网络。

11.2.4 Inception v1模型

Inception v1模型在原有的Inception模型基础上做了一些改进，原因是由于Inception的原始模型是将所有的卷积核都在上一层的所有输出上来做，那么 5×5 的卷积核所需的计算量就比较大，造成了特征图厚度很大。

为了避免这一现象，Inception v1模型在 3×3 前、 5×5 前、最大池化层后分别加上了 1×1 的卷积核，起到了降低特征图厚度的作用（其中 1×1 卷积主要用来降维），网络结构如图11-4所示。

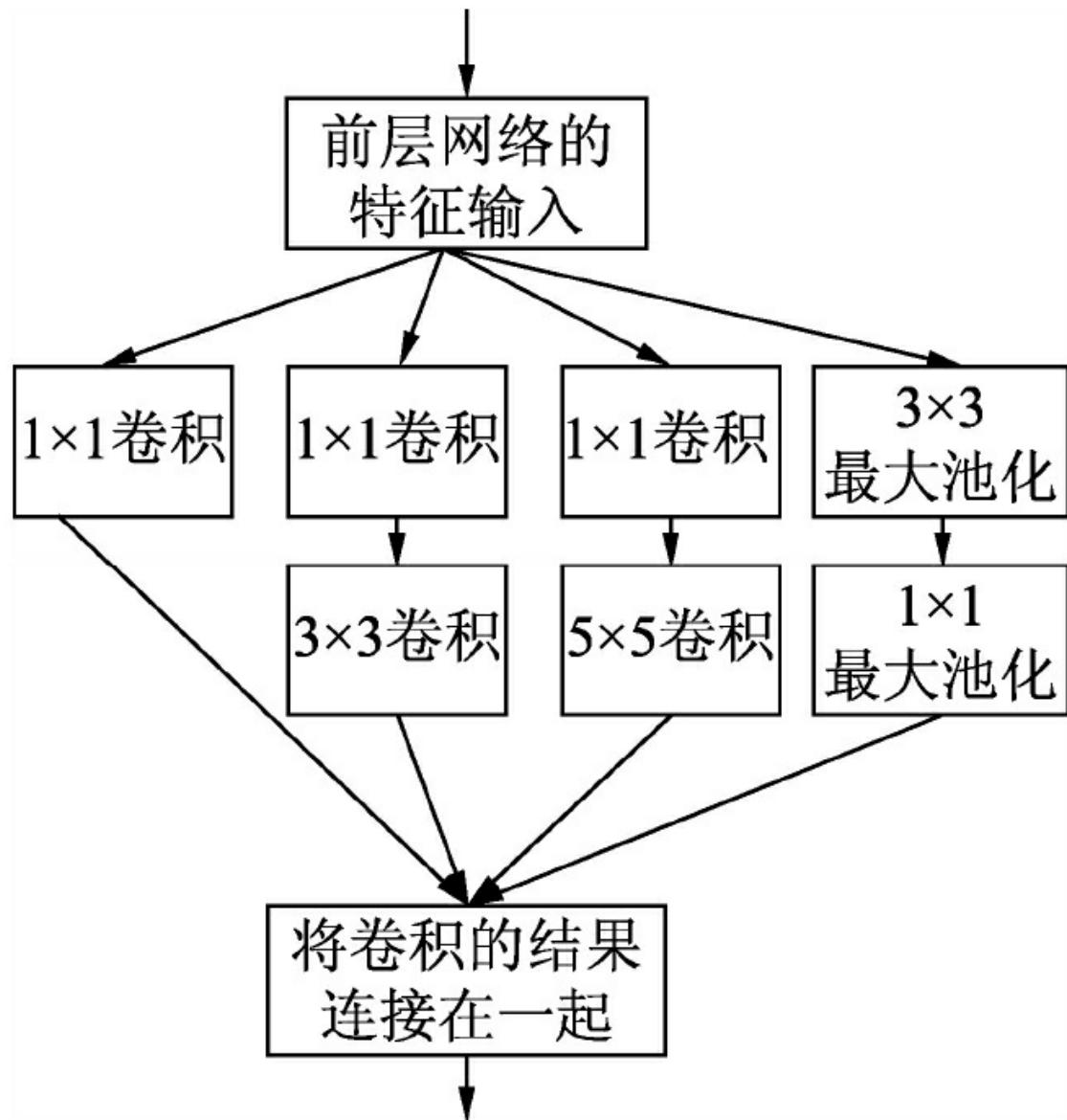


图11-4 Inception v1模型

Inception v1模型中有以下4个分支。

- 第1个分支对输入进行 1×1 的卷积，这其实也是NIN中提出的一个重要结构。 1×1 的卷积可以跨通道组织信息，提高网络的表达能力，同时可以对输出通道升维和降维。

- 第2个分支先使用了 1×1 卷积，然后连接

3×3 卷积，相当于进行了两次特征变换。

- 第3个分支与第2个分支类似，先是 1×1 的卷积，然后连接 5×5 卷积。
- 第4个分支则是 3×3 最大池化后直接使用 1×1 卷积。

可以发现4个分支都用到了 1×1 卷积，有的分支只使用 1×1 卷积，有的分支在使用了其他尺寸的卷积的同时会再使用 1×1 卷积，这是因为 1×1 卷积的性价比很高，增加一层特征变换和非线性转化所需的计算量更小。

Inception v1模型的4个分支在最后再通过一个聚合操作合并（使用tf.concat函数在输出通道数的维度上聚合）。

11.2.5 Inception v2模型

Inception v2模型在Inception v1模型基础上应用当时的主流技术，在卷积之后加入了BN层，使每一层的输出都归一化处理，减少了内变协变量的移动问题；同时还使用了梯度截断技术，增加了训练的稳定性。

另外，Inception学习了VGG，用2个 3×3 的conv替代inception模块中的 5×5 ，这既降低了参数

数量，也提升了计算速度。其结构如图11-5所示。

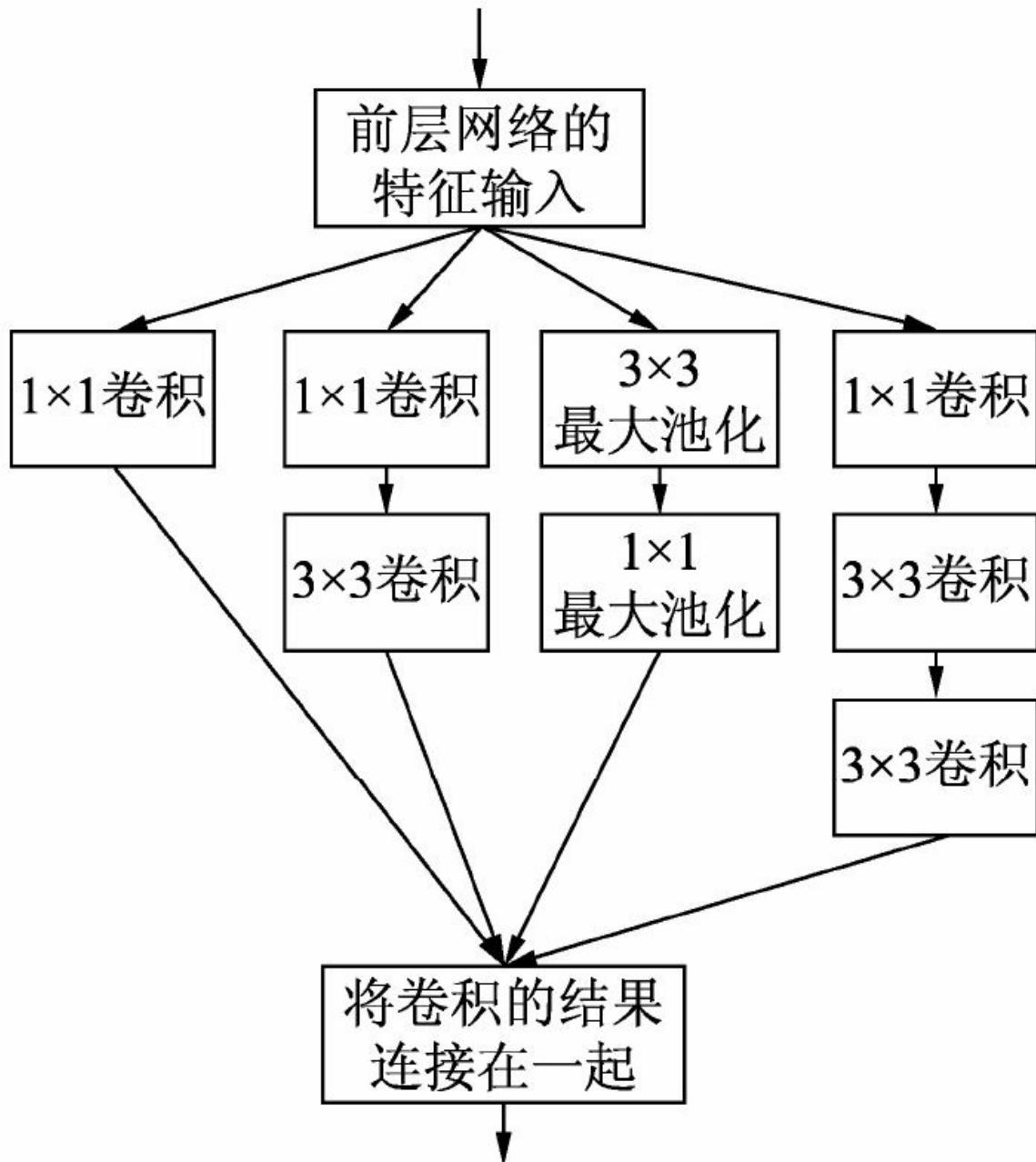


图11-5 Inception v2模型

11.2.6 Inception v3模型

Inception v3模型没有再加入其他的技术，只
*****ebook converter DEMO Watermarks*****

是将原有的结构进行了调整，其最重要的一个改进是分解，将图11-5中的卷积核变得更小。

具体的计算方法是：将 7×7 分解成两个一维的卷积 $(1 \times 7, 7 \times 1)$ ， 3×3 的卷积操作也一样 $(1 \times 3, 3 \times 1)$ 。这种做法是基于线性代数的原理，即一个 $[n, n]$ 的矩阵，可以分解成矩阵 $[n, 1] \times$ 矩阵 $[1, n]$ ，得出的结构如图11-6所示。

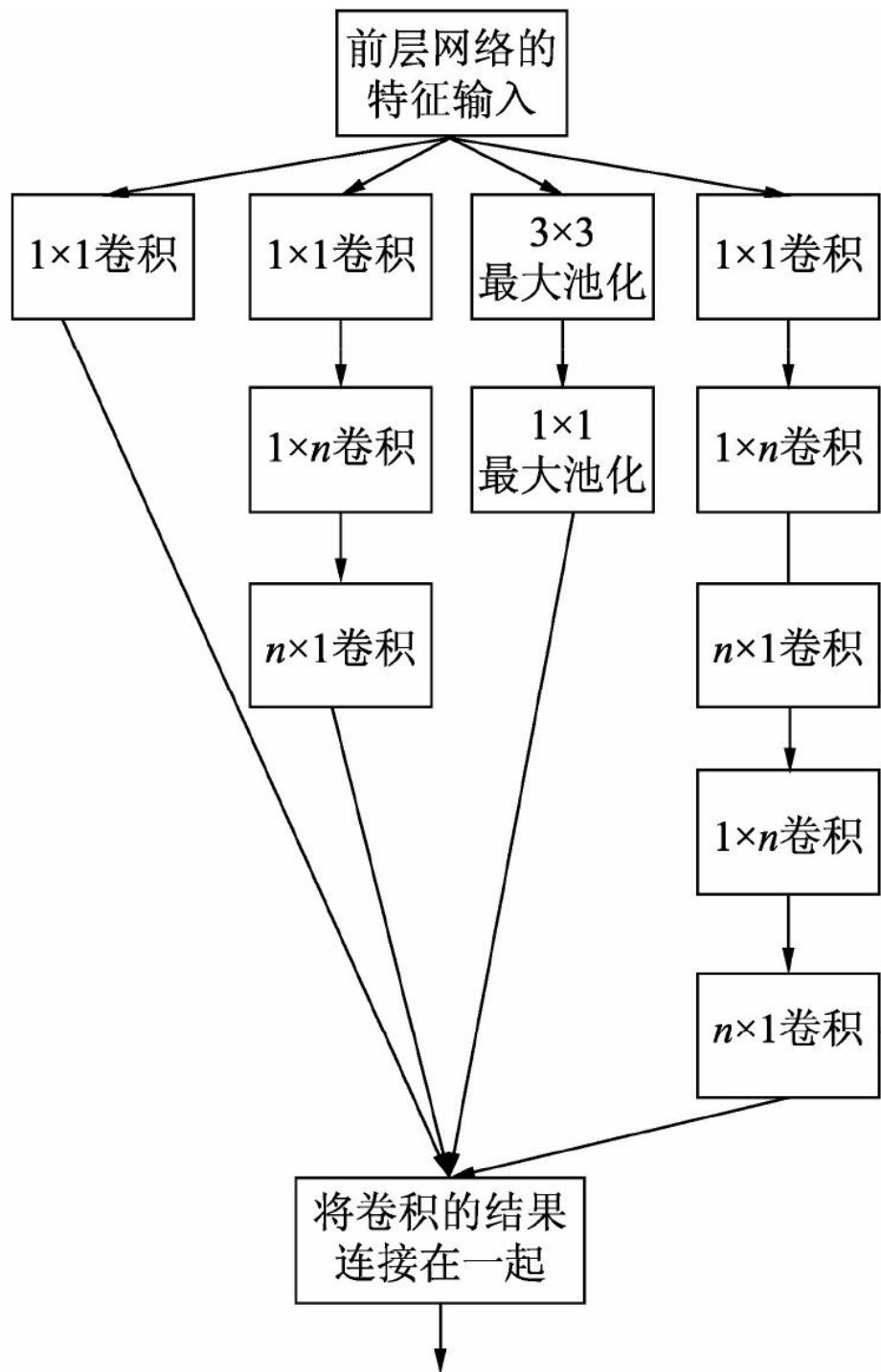


图11-6 Inception v3模型

这么做会有什么效果呢？我们举一个例子。

假设有256个特征输入，256个特征输出，如果Inception层只能执行 3×3 的卷积，即总共要完

成 $256 \times 256 \times 3 \times 3$ 的卷积（将近589 000次乘积累加运算）。

如果需要减少卷积运算的特征数量，将其变为64（即 $256/4$ ）个。则需要先进行 $256 \rightarrow 64$ 1×1 的卷积，然后在所有Inception的分支上进行64次卷积，接着再使用一个来自 $64 \rightarrow 256$ 的特征的 1×1 卷积，运算的公式如下：

$$\begin{aligned} 256 \times 64 \times 1 \times 1 &= 16\ 000s \\ 64 \times 64 \times 3 \times 3 &= 36\ 000s \\ 64 \times 256 \times 1 \times 1 &= 16\ 000s \end{aligned}$$

相比于之前的60万，现在共有7万的计算量，几乎原有的 $1/10$ 。

在实际测试中，这种结构在前几层处理较大特征数据时的效果并不太好，但在处理中间状态生成的大小在 $12 \sim 20$ 之间的特征数据时效果会非常明显，也可以大大提升运算速度。另外，Inception v3还做了其他的变化，将网络的输入尺寸由 224×224 变为了 299×299 ，并增加了卷积核为 $35 \times 35 / 17 \times 17 / 8 \times 8$ 的卷积模块。

11.2.7 Inception v4模型

Inception v4是在Inception模块基础上，结合残差连接（Residual Connection）技术的特点进行

了结构的优化调整。Inception-ResNet v2网络与Inception v4网络，二者性能差别不大，结构上的区别在于Inception v4中仅仅是在Inception v3基础上做了更复杂的结构变化（从Inception v3的4个卷积模型变为6个卷积模块等），但没有使用残差连接。

这里提到了一个残差连接（Residual Connection），它属于ResNet网络模型里的核心技术，通过对下面ResNet的学习，读者将会了解残差连接的含义。

11.3 残差网络（ResNet）

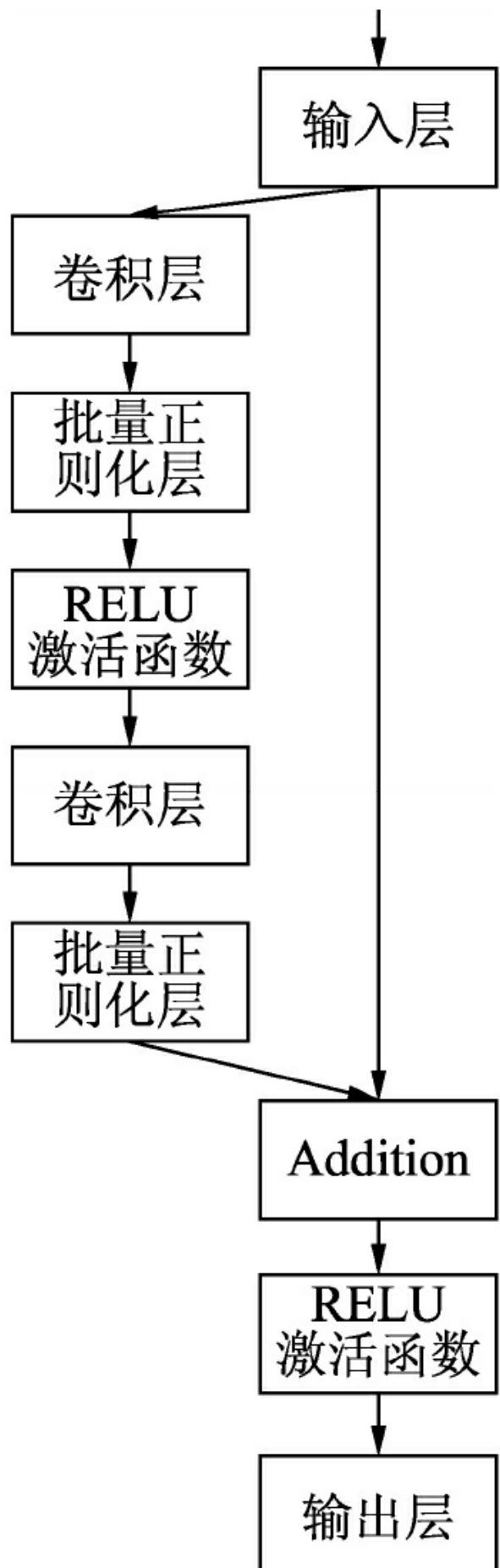
残差网络（ResNet），在ILSVRC 2015中取得了冠军，该框架能够大大简化模型网络的训练时间，使得在可接受时间内，模型能够更深。

在深度学习领域中，网络越深意味着拟合越强，出现过拟合问题是正常的，训练误差越来越大却是不正常的。但是，网络逐渐加深会对网络的反向传播能力提出挑战，在反向传播中每一层的梯度都是在上一层的基础上计算的，层数多会导致梯度在多层传播时越来越小，直到梯度消失，于是表现的结果就是随着层数变多，训练的误差会越来越大。

残差网络通过一个叫残差连接的技术解决了这个问题。所谓的残差连接就是在标准的前馈卷积网络上加一个跳跃，从而绕过一些层的连接方式。

11.3.1 残差网络结构

残差网络的结构，如图11-7所示。



*****ebook converter DEMO Watermarks*****

图11-7 残差网络结构

假设，经过两个神经层之后输出的 $H(x)$ 如下所示：

$$f(x) = \text{relu}(xw+b)$$
$$H(x) = \text{relu}(f(x)w+b)$$

$H(x)$ 和 x 之间存在一个函数的关系，如果这两层神经网络构成的是 $H(x) = 2x$ 的关系，则残差网络的定义如下：

$$H(x) = \text{relu}(f(x)w+b) + x$$

11.3.2 残差网络原理

如图11-7所示，ResNet中，输入层与Addition之间存在着两个连接，左侧的连接是输入层通过若干神经层之后连接到Addition，右侧的连接是输入层直接传给Addition，在反向传播的过程中误差传到Input时会得到两个误差的相加和，一个是左侧一堆网络的误差，一个是右侧直接的原始误差。左侧的误差会随着层数变深而梯度越来越小，右侧则是由Addition直接连到Input，所以还会保留着Addition的梯度。这样Input得到的相加和后的梯度就没有那么小了，可以保证接着将误差往下传。

这种方式看似解决了梯度越传越小的问题，但是残差连接在正向同样也发挥了作用。由于正向的作用，导致网络结构已经不再是深层了，而是一个并行的模型，即残差连接的作用是将网络串行改成了并行。这也可以理解为什么Inception v4结合了残差网络的原理后，却没有使用残差连接，反而做出了与Inception-ResNet v2等同的效果。

介绍Resnet主要是为下面的Inception-ResNet v2做铺垫，下面就来看看ILSVRC 2016年的冠军Inception-ResNet v2网络。

11.4 Inception-ResNet-v2结构

Inception-ResNet v2网络主要是在Inception v3的基础上，加入了ResNet的残差连接而来的。其原理与Inception v4一样，都是进行了细微的结构调整，并且二者的结构复杂度也不相上下。

通过有关论文实验表明：在网络复杂度相近的情况下，Inception-ResNet-v2略优于Inception v4。并且实验出，残差连接在Inception结构中具有提高网络准确率且不会提升计算量的作用，通过将3个带有残差连接的Inception模型和一个Inception v4的组合，就可以在ImageNet上得到3.08%的错误率。

关于二者的具体结构在11.5.2节会介绍相关代码位置，有兴趣的读者可以自行研究，这里不再展开介绍。

11.5 TensorFlow中的图片分类模型库 ——slim

TensorFlow 1.0之后推出了一个叫slim的库，TF-slim是TensorFlow的一个新的轻量级高级API接口。它类似前面所介绍的TensorFlow.contrib.layers模块，将很多常见的TensorFlow函数进行了二次封装，使代码变得更加简洁，特别适用于构建复杂结构的深度神经网络，它可以用来定义、训练和评估复杂的模型。

同时，在TensorFlow的models模块里又提供了大量用slim写好的网络模型结构代码，以及用该代码训练出的模型检查点文件，可以作为我们的预训练模型来使用。这些模型主要是与图片分类相关，包括ResNet、VGG、Inception-ResNet-v2等。下面就来详细了解下slim。

11.5.1 获取models中的 slim模块代码

为了能够使用models中的代码，需要先验证下我们的TensorFlow版本是否集成了slim模块，接着再从GitHub上将models代码下载下来，具体操作如下。

1. 验证slim库

在使用slim前，要测试本地的tf.contrib.slim模块是否有效。在命令行中输入如下命令：

```
python -c "import tensorflow.contrib.slim as slim;
eval = slim.evaluation.evaluate_once"
```

如果没有生成任何错误，则表明TF-Slim是可以工作的。

2. 下载models模块

接下来需要安装TF-slim image models library。来到以下网址<https://github.com/tensorflow/models/>，可以通过Git将代码复制下来，也可以手动下载下来（具体操作见8.5.2的详细介绍）。然后解压到本地workspace路径下（就是你自己建立的用来放个人TF代码的路径），通过下面的代码来验证它是否工作。

```
cd $workspace/models/research/slim
python -c "from nets import cifarnet; mynet = cifarnet.cifar10_v1()
```

将上面的\$workspace替换成你的工作路径（如笔者的是d:\python）。运行时如果没有发生任何错误，则表明一切正常。

11.5.2 models中的slim目录结构

*****ebook converter DEMO Watermarks*****

在models下的slim中一共有5个文件夹。

- Datasets：处理数据集相关的代码。
- Deployment：部署。通过创建clone方式实现跨机器的分布训练，可以在多CPU和多GPU上实现运算的同步或异步。
- Nets：该文件夹里放着各种网络模型。
- Preprocessing：适用于各个网络的图片处理函数。
- Scripts：运行网络模型的一些案例脚本，这些脚本只能在支持shell的系统下使用。

在这里重点介绍Datasets、nets和Preprocessing这3个文件夹。

1. Dataset——数据集处理模块

Dataset里放着常用的图片训练数据集相关的代码。主要支持的数据集主要有cifar10、flowers、mnist、imagenet。

代码文件的名字与数据集相对应，可以使用这些代码下载或获取数据集中的数据。以imagenet为例，可以使用如下函数从网上获取imagenet标签。

```
imagenet.create_readable_names_for_imagenet_labels()
```

上面代码返回的是imagenet中1000个类的分类标签名字（与样本序列对应）。

2. nets模块

nets文件夹下包含前面介绍的各种网络模型，如图11-8所示。

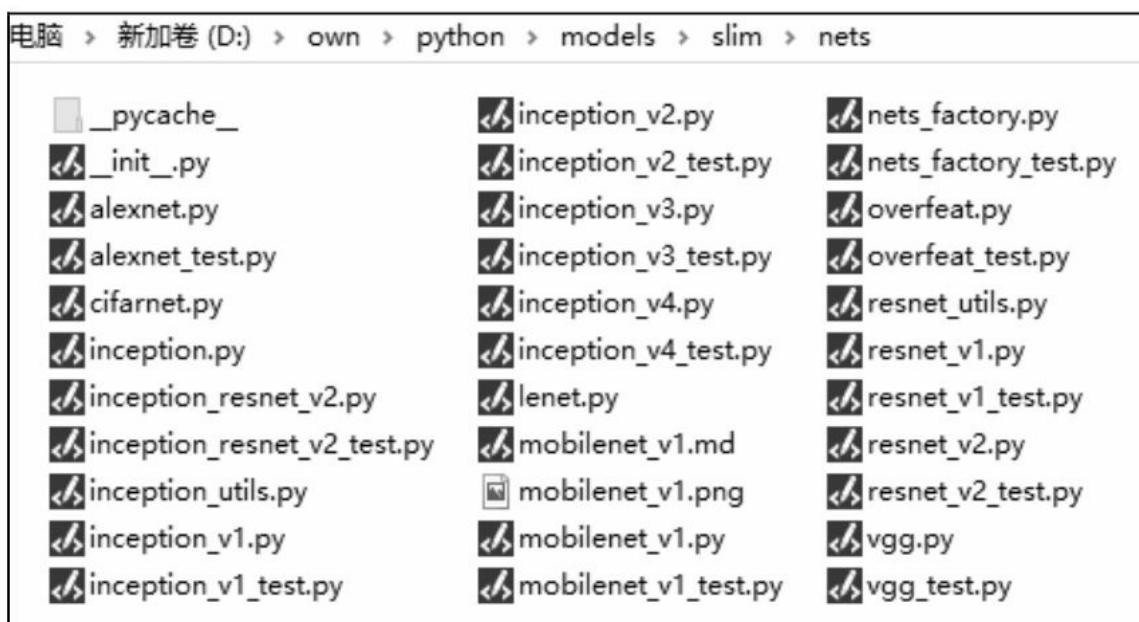


图11-8 nets文件结构

每个网络模型文件都是以自己的名字命名的，而且里面的代码结构框架也大致相同，以inception_resnet_v2为例，如表11-1中列出了比较常用的函数接口。

表11-1 slim中nets的代码框架接口

*****ebook converter DEMO Watermarks*****

操作	说明
inception_resnet_v2.default_image_size	默认图片大小
inception_resnet_v2.inception_resnet_v2	同名的网络结构函数，这个函数有两个输出，一个是预测结果logits，另一个是辅助信息AuxLogits。辅助信息为了显示或分析使用，主要包括summaries或losses
inception_resnet_v2_arg_scope	命名空间的名字。在外层修改或使用模型时，可以使用与模型相同的命名空间
inception_resnet_v2_base	为inception_resnet_v2的基本结构实现，输出inception_resnet_v2网络中最原始的数据，默认是传到inception_resnet_v2.inception_resnet_v2函数中，一般不会改动内部。当要使用自定义的输出层时，会将传入自己的函数来替换inception_resnet_v2.inception_resnet_v2



注意： 表11-1中的框架全部是使用slim库代码来实现的，由于与tensorflow.contrib.layers模块的使用方式很相似，这里不再展开介绍，但是建议读者配合前面讲的各个模型的结构再看看其具体在代码中的真实实现，对自己构建高效的模型会有很大帮助。

3. Preprocessing模块

该模块代码里包含几个图片预处理文件，命名也是按照模型的名字来命名的。slim会把某一类模型常用的预处理函数放到一个文件里，并命名为该类模型相关的名字，而且每个代码文件函数结构也大致相似。例如，调用inception_preprocessing函数中的代码如下：

```
inception_preprocessing.preprocess_image
```

该函数是将传入的图片转化成模型尺寸并归

*****ebook converter DEMO Watermarks*****

一化处理。

11.5.3 slim中的数据集处理

slim模块包自带了函数，可以用来下载数据集，也可以对数据集进行转换操作。它可以下载标准的数据集并转换为TensorFlow自带的TFRecord格式，还可以使用TF-slim的data reading和queueing utilities来读取TFRecord格式的数据集。slim所支持的数据集如表11-2所示。

表11-2 slim中集成的数据集

数 据 集	训练数据集大小	测试数据集大小	分 类 个 数	备 注
Flowers	2500	2500	5	尺寸可变
Cifar10	60×1000	10×1000	10	32×32 彩色图
MNIST	60×1000	10×1000	10	28×28 灰度图
ImageNet	$1.2 \times 1000 \times 1000$	50×1000	1000	尺寸可变

1. 将数据转为TFRecord格式

TFRecord是TensorFlow推荐的数据集格式，与TensorFlow框架结合紧密。在TensorFlow中提供了一系列接口可以访问TFRecord格式。该结构存在的意义主要是为了满足在处理海量样本集时，需要边执行训练边从硬盘上读取数据的需求。将原始文件转化成TFRecord的格式，然后在运行中通过多线程的方式来读取，这样可以减小主线程训练的负担，使整个训练过程变得更高效。

只需要在命令行里输入下列命令，即可将下载数据集并将其转成TFRecord格式：

```
D:\python\research\models\slim>python download_and_convert_
```

这里需要指定两个关键点：一个是数据集（例子中的flowers），另一个是下载路径（笔者的Python文件是在D盘，所以会下载到D:\tmp\data\flowers下）。执行完后会看到下载的数据文件和生成的TFRecord文件，如图11-9所示。

这里包含5个训练数据文件、5个验证数据文件及一个标签文件。标签文件定义了整数标签和分类名称。

如果想将其他数据集转成TFRecord格式，可以参考上面的代码实现，这里不再展开介绍。

同样，也可以按照这个方法下载MNIST和cifar10数据集。如果需要下载imageNet数据集，则需要在image-net.org中注册一个账号，然后再运行下载脚本，大概有500GB，因此需要留出足够的硬盘空间，并且下载时间会很长。

2. 处理slim数据集时的常见错误

由于有时网络有时会不稳定，因此使用上面讲的方法下载数据时往往回遇到如下错误，主要

*****ebook converter DEMO Watermarks*****

是由于没有下载完成的原因。

```
urllib.error.ContentTooShortError: <urlopen error retrieval  
got only  
64456280 out of 228813984 bytes>
```

这表明由于网络原因数据包没有下载完整，有两种方法可以解决。

- 如果你的网速较快的话，可以多运行几次，总有一次可以成功。
- 可以

将http://download.tensorflow.org/example_images/flower_photos.tgz 网址放到下载工具（如迅雷等）里自行下载。

解压后的路径如图11-10所示。

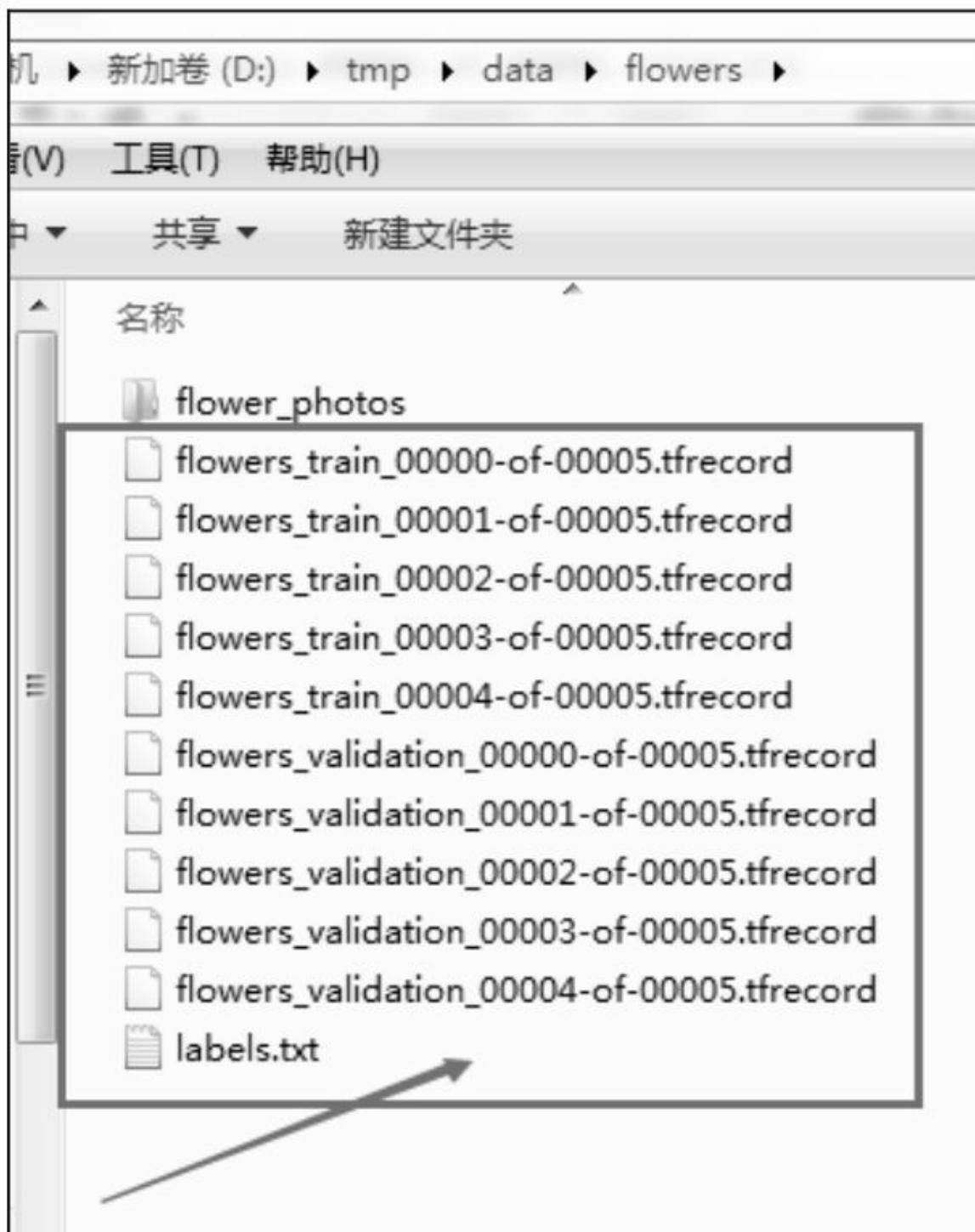


图11-9 flowers文件夹的TFRecord数据集

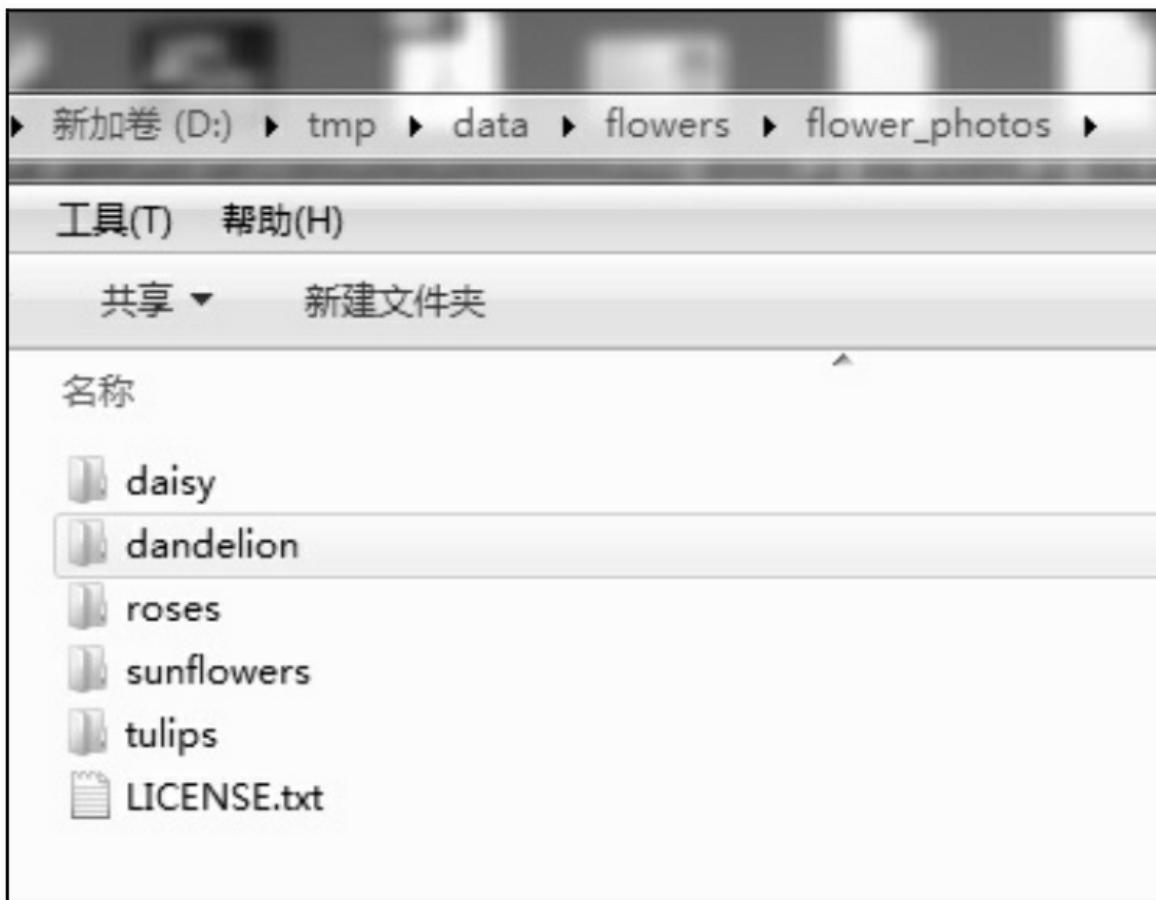


图11-10 flowers数据集

然后来到download_and_convert_flowers.py第191行，注释掉下列代码即可。

```
#dataset_utils.download_and_uncompress_tarball(_DATA_URL, dat
```

本书配套的代码中会有实例文件（见“11.5.3 代码参考”文件夹）。

下载完成后，运行如下命令将刚下载的数据集转成TFRecord格式（以图11-10中的路径“/tmp/data/flowers”为例）：

*****ebook converter DEMO Watermarks*****

```
D:\python\research\models\slim>python download_and_convert_c
```

11.5.4 实例84：利用slim读取TFRecord中的数据

TFRecord文件创建好后，就可以读取文件中的数据了，本例将演示如何读取TFRecord中的数据，步骤如下。

实例描述

利用slim代码库里的函数读取TFRecord格式的数据并显示出来。

1. 定义slim数据集，创建provider

在图11-9中可以看到，有两个数据集train与validation。在读取时，需要指定一个数据集然后创建provider对象，接着就可以从provider里读取数据了，代码如下：

代码11-1 tfrecodertest

```
01 import tensorflow as tf
02 from datasets import flowers
03 import pylab
04
05 slim = tf.contrib.slim
06
07 DATA_DIR="D:/own/python/flower_photosos" #指定flower数据集
08
09 #选择数据集validation
```

*****ebook converter DEMO Watermarks*****

```
10 dataset = flowers.get_split('validation', DATA_DIR)
11
12 #创建一个provider
13 provider = slim.dataset_data_provider.DatasetDataProvider(
14 #通过provider的get获得一条样本数据
15 [image, label] = provider.get(['image', 'label'])
16 print(image.shape)
```

上述代码中，先引入头文件，然后创建provider，通过get来获得image与label两个张量。这时并没有真的读到数据，只是一个构建图的过程。具体取数据则要通过session中启动队列线程后才可以。

provider是使用DatasetDataProvider类的实例化实现的，在DatasetDataProvider类中还可以有更多的设置：

```
class DatasetDataProvider(data_provider.DataProvider):

    def __init__(self,
                 dataset,
                 num_readers=1,
                 reader_kwargs=None,
                 shuffle=True,
                 num_epochs=None,
                 common_queue_capacity=256,
                 common_queue_min=128,
                 record_key='record_key',
                 seed=None,
                 scope=None):
```

必选参数是传入指定的数据集dataset，其他还包括指定几个并行读取器来读取数据num_readers、是否打乱顺序shuffle、指定数据源

*****ebook converter DEMO Watermarks*****

读取的循环次数num_epochs（None表示无限循环）、队列大小common_queue_capacity等。没有特殊要求的情况下，直接默认即可。



注意：本例演示的是只读取一条样本，在训练中需要按批次读取指定数量的样本，这时会需要配合tf.train.batch一起使用，tf.train.batch有个条件就是必须指定样本的固定大小，所以在传入时需要将变长的图片按固定大小调整。在slim的训练模型代码里有使用的例子，读者可以自己参考。另外，在第12章对抗神经网络里，超分辨率部分也有实例供读者学习、参考。

2. 启用session读取数据

在session中初始化变量之后，需要通过tf.train.start_queue_runners来启动队列线程。这时会有一个线程专门负责从磁盘里读图片数据，接着通过run来运行图节点image与label得到真实的数据。

代码11-1 tfrecordtest（续）

```
17 sess = tf.InteractiveSession()
18 tf.global_variables_initializer().run()
19 #启动队列
20 tf.train.start_queue_runners()
21 #获取数据
22 image_batch, label_batch = sess.run([image, label])
23 #显示
```

```
24 print(label_batch)
25 pylab.imshow(image_batch)
26 pylab.show()
```

运行上述代码，输出的图片如图11-11所示。

```
(?, ?, 3)
1
```

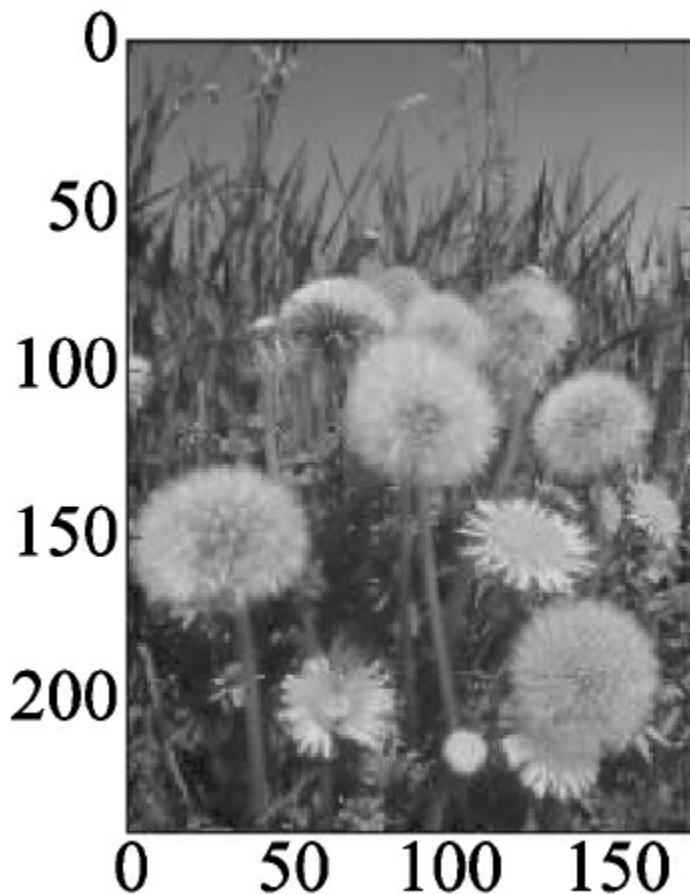


图11-11 TFRecord例子

多运行几次发现，每次的结果都不一样，再次证实了默认是随机读取的。



注意： 在处理大数据样本时，将数据转成TFRecord后使用线程来读取，是一个较常规的方式。千万不能像MNIST数据集读取那样一次都读入内存，内存会被样本耗尽，系统就无法处理其他的数据了。另外，除了使用TFRecord方式以外，还可以从filenames中读取，通过异步读取文件，然后按批次的随机抽取指定样本数量，再输入到模型中来做模型参数的更新。

11.5.5 在slim中训练模型

slim提供了很多便捷的方式，前面提到的全部模型在slim中都可以找到对应的代码实现。不仅如此，slim还共享了模型的训练代码，使用者不再需要关注模型代码，只需通过命令行方式即可完成训练、微调、测试等任务，大大方便了模型的产出。

对于linux用户，在slim的Scripts文件夹下还提供了模型下载、训练、预训练、微调、测试等一条龙的完整shell脚本。如果你用的是Windows，也可以在命令行下一条一条地复制命令并执行命令。

```
D:\python\research\models\slim>python train_image_classifier.py log/in3flower --dataset_name=flowers --dataset_split_name=train --train_dir=/tmp/data/flowers/flower_photos --model_name=inception_v3
```

关于shell脚本代码，不再逐条解释，下面举例演示使用命令行来训练模型的相关操作。

1. 从头训练

训练模型的代码被放在slim下的train_image_classifier.py文件里，这里用flower数据集来训练Inception_v3网络结构的深度神经网络模型。在命令行中执行如下命令：

参数说明如下。

- train_dir：是要生成模型的路径。
- dataset_name：数据集名字。
- dataset_split_name：数据集中的哪一部分是validation还是train。
- dataset_dir：数据集路径。
- model_name：模型名字。

这里只列出了主要的参数，其他的参数可以仿照shell脚本中的例子，如果读者想知道全部的参数，可参看train_image_classifier.py文件。也可以修改train_image_classifier.py文件，添加自己喜欢的参数。



注意： dataset_name、
dataset_split_name、 model_name的名字不是随意命名的，必须与代码中的名字对应。如果使用自己的数据集，则需要在slim中的dataset文件夹下仿照其他的数据集加一个.py文件，然后也可以用train_image_classifier.py来运行。当然读者也可以不使用train_image_classifier.py，直接自己编写代码载入数据集。

2. 预训练模型

预训练就是在别人训练好的模型基础上进行二次训练，以得到自己需要的模型。可以帮你省去大量的时间。一些高质量的模型都是通过了大量的数据样本训练而来的。GitHub上提供了很多训练好的模型，可用于预训练，可以在<https://github.com/TensorFlow/models/tree/master/research/slim#Pretrained> 中下载。

该链接是TensorFlow里slim模块在GitHub中的页面，页面中的表的部分内容如图11-12所示。

Model	TF-Slim File	Checkpoint	Top-1 Accuracy	Top-5 Accuracy
Inception V1	Code	inception_v1_2016_08_28.tar.gz	69.8	89.6
Inception V2	Code	inception_v2_2016_08_28.tar.gz	73.9	91.8
Inception V3	Code	inception_v3_2016_08_28.tar.gz	78.0	93.9
Inception V4	Code	inception_v4_2016_09_09.tar.gz	80.2	95.2
Inception-ResNet-v2	Code	inception_resnet_v2_2016_08_30.tar.gz	80.4	95.3
ResNet V1 50	Code	resnet_v1_50_2016_08_28.tar.gz	75.2	92.2
ResNet V1 101	Code	resnet_v1_101_2016_08_28.tar.gz	76.4	92.9
ResNet V1 152	Code	resnet_v1_152_2016_08_28.tar.gz	76.8	93.2
ResNet V2 50^	Code	resnet_v2_50_2017_04_14.tar.gz	75.6	92.8
ResNet V2 101^	Code	resnet_v2_101_2017_04_14.tar.gz	77.0	93.7
ResNet V2 152^	Code	resnet_v2_152_2017_04_14.tar.gz	77.8	94.1
ResNet V2 200	Code	TBA	79.9*	95.2*

图11-12 模型下载截图

图11-12中的表格内，Checkpoint列是模型下载的链接。这些模型都是在ILSVRC-2012-CLS (ImageNet) 数据集上训练而来的，这个数据集共500GB，共分为1000个类的图片。想要了解更多关于ImageNet的信息，可以看网站
<http://www.image-net.org/challenges/LSVRC/2012/>

◦

下载完预训练模型后，只需要在11.5.5节“从头训练”的命令中添加一个参数——checkpoint_path即可。

--checkpoint_path =模型的路径

--checkpoint_path里的模型用于预训练模型的
*****ebook converter DEMO Watermarks*****

参数初始化。在训练过程中不会改变，新产生的模型会被保存在--train_dir指定的路径下面。



注意： 预训练时使用的样本必须与原来的输入尺寸和输出的分类个数一致。这些可下载的模型都是要分成1000类的，如果你不想分这么多类，可以使用下面微调的方法。

3. 微调fine-tuning

上述的预训练模型都是在imagenet上训练的，最终输出的是1000个分类，如果我们想使用预训练模型训练自己的数据集时，就要微调了。

在微调过程中，需要将原有模型中的最后一层去掉，换成自己的数据集对应的分类层。例如我们要训练flowers数据集，就需要将1000个输出换成10个输出。

具体做法如下。

(1) 通过参数--checkpoint_exclude_scopes指定载入预训练模型时哪一层的权重不被载入。

(2) 再通过--trainable_scopes参数指定对哪一层的参数进行训练。当--trainable_scopes出现时，没有被指定训练的参数将在训练中被冻结。

举例：使用inception_v3的模型进行微调，使其可以训练flowers数据集。将下载好的模型inception_v3.ckpt解压后放在当前目录文件夹inception_v3下，通过cmd进入命令行来到models\slim文件夹下，运行如下命令：

```
D:\own\python\research\models\slim>python train_image_classifier.py inception_v3/inception_v3.ckpt --checkpoint_exclude_scopes=
```

在例子中，--checkpoint_path里的模型会被载入，将权重初始化成模型里的值，同时--checkpoint_exclude_scopes限制了最后一层没有被初始化成模型里的参数。--trainable_scopes指定了只训练最后新加的一层，这样在训练过程中被冻结的其他参数具有原来训练好的合适值，而新加的一层则通过迭代在不断地优化自己的参数。

在微调的过程中，还可以通过在上面命令中加入：

```
--max_number_of_steps=500
```

来指定训练步数。如果没有指定训练步数，默认会一直训练下去。更多的参数，可参看train_image_classifier.py的源码。另外，slim的Scripts中还有使用模型来识别图片的例子，读者可以一起配合着学习。



注意： 有时会报初始化失败的错误：

```
E c:\tf_jenkins\home\workspace\release-win\device\gpu\os\wir
NOT_INITIALIZED
2017-05-02 17:48:48.334466: E c:\tf_jenkins\home\workspace\i
2017-05-02 17:48:48.343454: E c:\tf_jenkins\home\workspace\i
BAD_PARAM
```

这种问题表明显卡没有启动，重启计算机即可

4. 评估模型

eval_image_classifier.py文件是已经封装好用来评估模型的。下面还是以上面的flower集合微调Inception_v3的模型为例，评估模型的命令如下：

```
python eval_image_classifier.py --checkpoint_path=log/in3/mo
-3416059 --eval_dir=log/in3/model.ckpt-3416059 --dataset_name=
--dataset_split_name=validation --dataset_dir=D:\own\python\1
photosos --model_name=inception_v3
```

其中，指定路径的文件为log/in3/model.ckpt-3416059，即在微调中训练出来的模型文件。

5. 打包模型

训练好的模型可以被打包到各个平台上使

用，无论是iOS、Android还是Linux系列。具体是通过一个bazel开源工具来实现的。这部分内容不在本书的范围之内，有兴趣的读者可以自行研究。

更多的内容可以参考链接

<https://github.com/tensorflow/models/tree/master/research/slim#Export>。

11.6 使用slim中的深度网络模型进行图像的识别与检测

前面模型训练的知识点可以覆盖模型方面的大部分情况。如果读者刚好有图片分类的任务，或是想进行图片的识别，用slim中已有的网络结构来训练出自己的模型，比自己重新写一个模型的可行性更高一些。智者必须要学会借力而行。

有了模型之后就是使用模型了。下面通过几个实例来演示如何使用模型。

11.6.1 实例85：调用Inception_ResNet_v2模型进行图像识别

本例是使用在ImageNet上训练好的Inception_ResNet_v2模型来识别图片内容，练习通过编写代码来调用slim中的inception_resnet_v2函数。具体步骤如下。

实例描述

使用基于ImageNet上训练的Inception_ResNet_v2模型对任意图片进行识别。

1. 准备工作

需要准备好Inception_ResNet_v2的模型文件（上文有下载方法介绍），以及两张用于识别的图片。

2. 引入头文件，指定模型

在slim文件夹下创建代码文件。代码中通过导入nets中的Inception模块，即可包含slim中的所有网络结构代码，导入Datasets中的imagenet是为了使用imagenet的label标签，方便识别后的显示。为了让代码简洁一些，令slim = tf.contrib.slim。

代码11-2 inception_resnet_v2使用

```
01 import tensorflow as tf
02
03 from PIL import Image
04 from matplotlib import pyplot as plt
05 from nets import inception
06 import numpy as np
07 from datasets import imagenet
08
09 tf.reset_default_graph()
10 image_size = inception.inception_resnet_v2.default_image_
11 names = imagenet.create_readable_names_for_imagenet_label_
12
13 slim = tf.contrib.slim
14
15 checkpoint_file = 'inception_resnet_v2/inception_resnet_\
08_30.ckpt'
16 sample_images = ['img.jpg', 'ps.jpg']
```

将inception_resnet_v2中的默认尺寸取到，给

出模型文件和图片的路径即文件名。

3. 载入模型

获取模型参数的命名空间arg_scope，定义相同命名空间下的输出节点。Logits是刚从网络结构里运算出来的输出。end_points为一个全集，里面包含logits和将logits经过softmax之后的预测结果及其他信息。具体可以参考nets下Inception_ResNet_v2里的inception_resnet_v2函数。

代码11-2 inception_resnet_v2使用（续）

```
17 input_imgs = tf.placeholder("float", [None, image_size, image_size, 3])
18
19 #载入 model
20 sess = tf.Session()
21 arg_scope = inception.inception_resnet_v2_arg_scope()
22
23 with slim.arg_scope(arg_scope):
24     logits, end_points = inception.inception_resnet_v2(input_imgs,
25     is_training=False)
26 saver = tf.train.Saver()
27 saver.restore(sess, checkpoint_file)
28
```

在session里通过saver载入模型，这部分内容前面讲过，这里不再赘述。

4. 输入图片进行识别

通过循环读入sample_images中指定的图片，然后使用resize函数将其重新调整尺寸到指定大小，再使用reshape函数重新将形状调整成[-1, image_size, image_size, 3]矩阵，并将其除以255再乘上2，然后减去1，归一化成[-1, 1]之间的值，输入模型生成结果。

代码11-2 inception_resnet_v2使用（续）

```
29 for image in sample_images:
30     reimg = Image.open(image).resize((image_size,image_s:
31     reimg = np.array(reimg)
32     reimg = reimg.reshape(-1,image_size,image_size,3)
33
34     plt.figure()
35     p1 = plt.subplot(121)
36     p2 = plt.subplot(122)
37
38     p1.imshow(reimg[0])# 显示图片
39     p1.axis('off')
40     p1.set_title("organization image")
41
42     reimg_norm = 2 *(reimg / 255.0)-1.0
43
44     p2.imshow(reimg_norm[0])                      # 显示图片
45     p2.axis('off')
46     p2.set_title("input image")
47
48     plt.show()
49
50     predict_values, logit_values = sess.run([end_points['
logits], feed_dict={input_imgs: reimg_norm})
51
52     print (np.max(predict_values), np.max(logit_values))
53     print (np.argmax(predict_values), np.argmax(logit_va:
[np.argmax(logit_values)]))
```



注意： 在数值变换中，本来应该使用slim自带的inception_preprocessing.preprocess_image函数将图片直接处理好，但是该代码似乎有点bug，模型不能识别出处理完的图片。于是改为手动来转化。GitHub中的代码还在不断更新中，或许当读者看这本书的时候已经没有bug了，那么就可以用inception_preprocessing.preprocess_image函数来替代。

运行代码，得到输出如下，输出图片如图11-13和图11-14所示。

```
INFO:tensorflow:Restoring parameters from inception_resnet_\resnet_v2_2016_08_30.ckpt
0.61667 9.0568
621 621 laptop, laptop computer
```

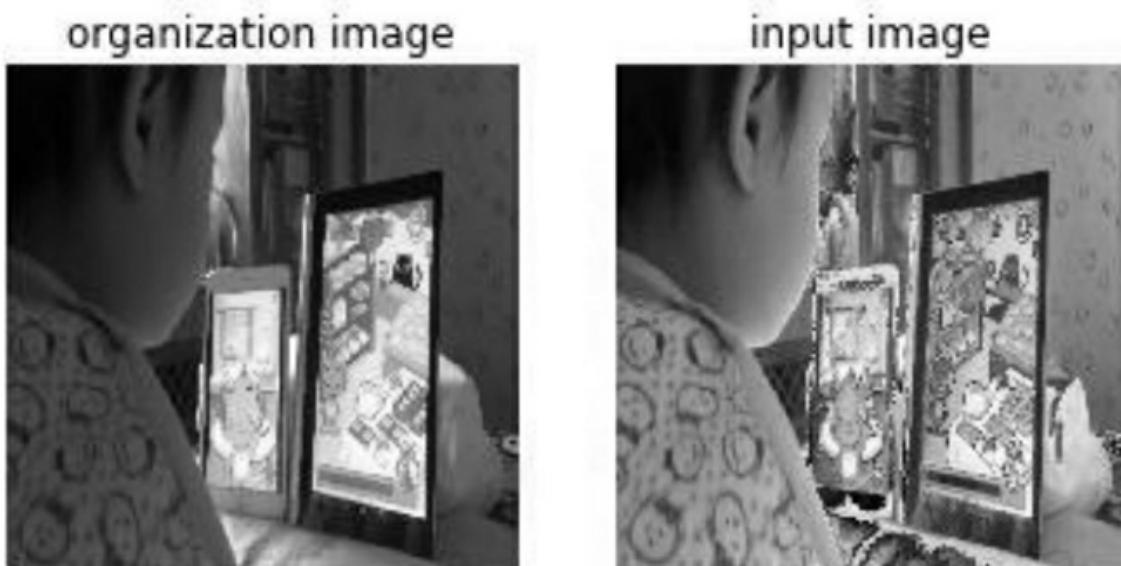


图11-13 inception_resnet_v2例子结果1

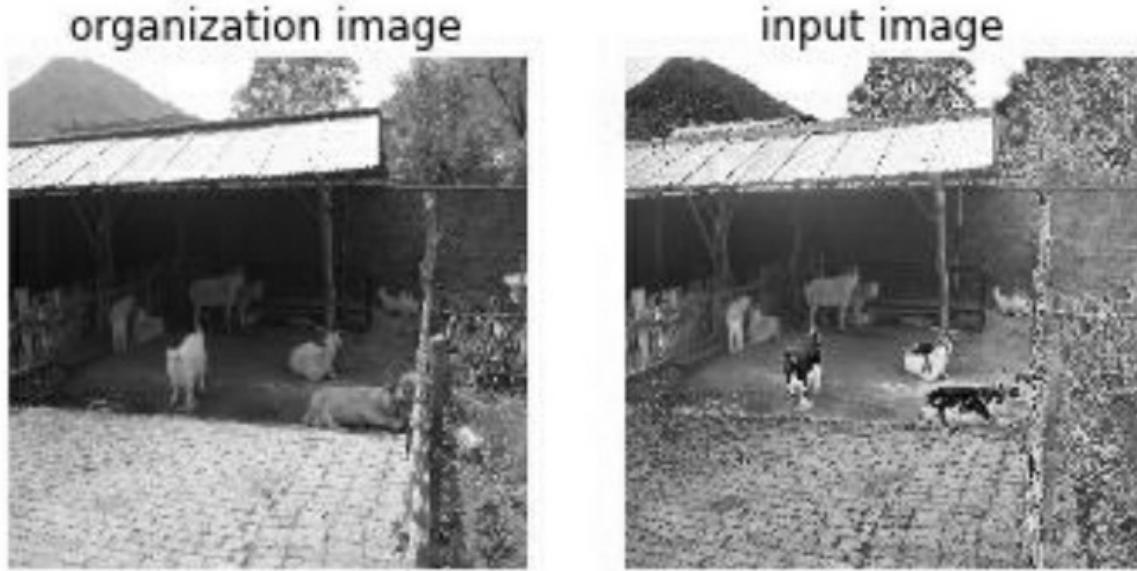


图11-14 inception_resnet_v2例子结果2

0.242343 8.80805
223 223 kuvasz

可以看到模型成功地识别了平板电脑。对于第二幅图，本来是只羊，却识别成了库瓦兹犬，这可能是由于训练样本中关于狗的样本比较多的原因，整个ImageNet数据集对狗的分类比较细致。不过看一下库瓦兹犬的图片（如图11-5所示），它跟羊真的有点相像。



图11-15 库瓦兹犬

从图11-5中看，库瓦兹犬就像是一只“披着羊皮”的狗，可见Inception_ResNet_v2惊人的识别力。

slim中的所有的模型的使用方法几乎一样，这里使用的Inception_ResNet_v2模型只是一个例子。若读者想使用其他模型，可以仿照该例子直接将模型名字替换Inception_ResNet_v2即可。

11.6.2 实例86：调用VGG模型进行图像检测

VGG作为深度学习模型，本来是为了识别图像而产生的，但其在图像检测方面的效果很好，

于是就成为图像检测方面的标杆模型。下面通过一个实例来使用VGG19模型对图片进行检测，看看VGG模型能从图片中识别哪些东西。具体步骤如下。

实例描述

使用基于ImageNet上训练的VGG19模型对任意图片进行检测。

1. 准备工作

准备好解压后的vgg_19.ckpt模型文件，放到当前vgg_19_2016_08_28目录下，这里还使用实例85中的两张图片进行检测。

2. 引入头文件，指定模型

类似实例85，在slim文件夹下创建代码文件。导入nets中的VGG模块，同时导入像素均值处理函数mean_image_subtraction，导入Datasets中imagenet的label标签，令slim = tf.contrib.slim。

代码11-3 vgg19图片检测使用

```
01 import numpy as np
02 import os
03 import tensorflow as tf
04
05 from PIL import Image
06 from datasets import imagenet
```

*****ebook converter DEMO Watermarks*****

```
07 from nets import vgg
08 # 加载像素均值及相关函数
09 from preprocessing.vgg_preprocessing import (_mean_image_
10 _R_MEAN, _G_MEAN, _B_MEAN)
11 from matplotlib import pyplot as plt
12 import matplotlib as mpl
13 mpl.rcParams['font.sans-serif']=['SimHei']      #用来正常显示
14 mpl.rcParams['font.family'] = 'STSong'
15 mpl.rcParams['font.size'] = 12
16
17 tf.reset_default_graph()
18
19 slim = tf.contrib.slim
20
21 # 网络模型的输入图像有默认的尺寸
22 # 先调整输入图片的尺寸
23
24 names = imagenet.create_readable_names_for_imagenet_labels
25 checkpoints_dir = 'vgg_19_2016_08_28'
26 sample_images = ['hy.jpg', 'ps.jpg']
```

3. 定义节点，载入模型

定义输入占位符，在这里不需要使用VGG的默认尺寸，所以使用[None, None, 3]的shape对输入节点进行均值处理，并使用reshape函数更新，将形状调整成1, None, None, 3]。

获取模型参数的命名空间arg_scope，定义相同命名空间下的输出节点。Logits是刚从网络结构里运算出来的输出。手动将logits的最大索引放入pred里，即代表分类。

代码11-3 vgg19图片检测使用（续）

```
27 input_imgs = tf.placeholder("float", [None, None, 3])
28 # 每个像素减去像素的均值
```

*****ebook converter DEMO Watermarks*****

```
29 processed_image = _mean_image_subtraction(input_imgs,
30                                     [_R_MEAN, _G_MEAN, _B_MEAN])
31
32 input_image = tf.expand_dims(processed_image, 0)
33 with slim.arg_scope(vgg.vgg_arg_scope()): #spatial_squeeze=False):
34
35     logits, _ = vgg.vgg_19(input_image,
36                             num_classes=1000,
37                             is_training=False,
38                             spatial_squeeze=False)
39
40 pred = tf.argmax(logits, dimension=3)
41
42 init_fn = slim.assign_from_checkpoint_fn(
43     os.path.join(checkpoints_dir, 'vgg_19.ckpt'),
44     slim.get_model_variables('vgg_19'))
45
46 with tf.Session() as sess:
47     init_fn(sess)
```

指定模型文件vgg_19.ckpt，并在session中载入。

4. 输入图片进行检测

通过循环读入sample_images中指定的图片，传入模型，生成结果obj。VGG的输出与其他模型不一样，它会返回识别出来的所有类别，并且顺序是与像素位置关系相对应的。使用np.unique函数会返回两个值，第一个值为对应的类别，第二个值为该类在obj中的起始位置。

代码11-3 vgg19图片检测使用（续）

```
48 for image in sample_images:
49     reimg = Image.open(image)
50     plt.suptitle("原始图片", fontsize=14, fontweight='bold')
*****ebook converter DEMO Watermarks*****
```

```
51     plt.imshow(reimg)          # 显示图片
52     plt.axis('off')           # 不显示坐标轴
53     plt.show()
54
55     reimg = np.asarray(reimg, dtype='float')
56
57     obj, inpt= sess.run([pred, input_image], feed_dict={reimg})
58
59     obj = np.squeeze(obj)
60
61     unique_classes, relabeled_image = np.unique(obj,
62                                               return_inverse=True)
63
64     obj_size = obj.shape
65     relabeled_image = relabeled_image.reshape(obj_size)
66     labels_names = []
67
68     for index, current_class_number in enumerate(unique_classes):
69         labels_names.append(str(index) + ' ' + names[current_class_number+1])
```

5. 输出结果

接着添加如下代码，将结果显示出来。

代码11-3 vgg19图片检测使用（续）

```
70     showobjlab(img=relabeled_image, labels_str=labels_names,
71                  title="画面识别")
72     plt.show()
```

这里用到了一个显示函数showobjlab，需要先定义一下。它将img位置obj中的类以不同的颜色在图像中显示出来。具体实现如下。

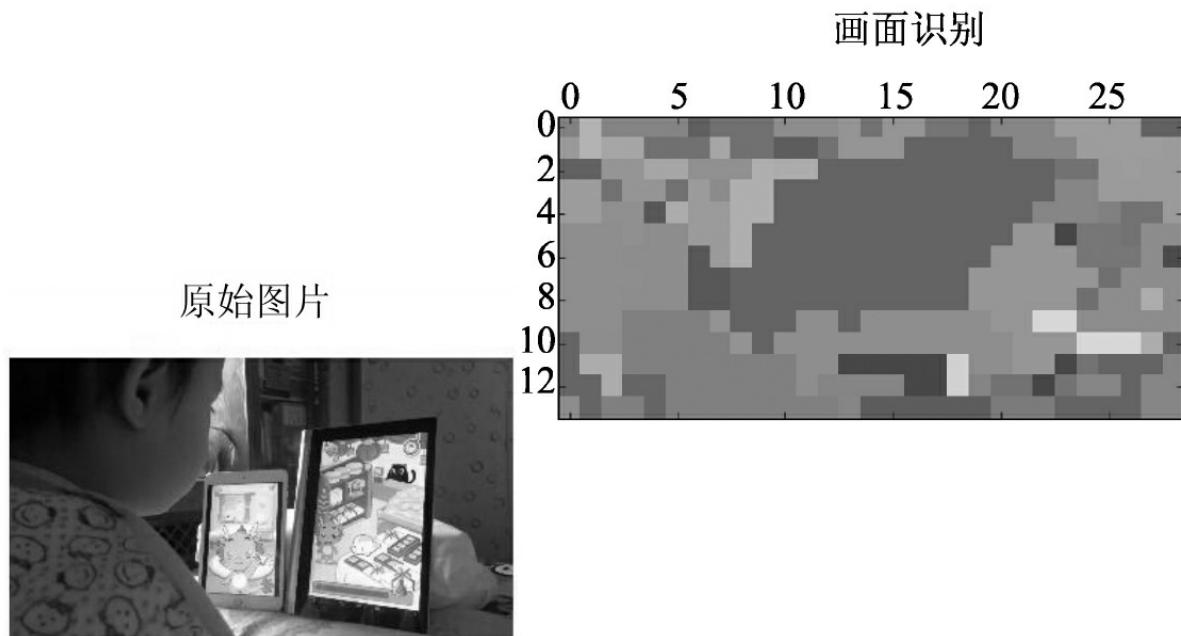
代码11-3 vgg19图片检测使用（续）

*****ebook converter DEMO Watermarks*****

```
72 def showobjlab(img, labels_str=[], title=""):  
73     minval = np.min(img)  
74     maxval = np.max(img)  
75     # 获取离散化的色彩表  
76     plt.figure(figsize=(3,3))  
77     cmap = plt.get_cmap('Paired', np.max(img)-np.min(img))  
78     mat = plt.matshow(img, cmap=cmap,vmin = minval-0.5,vr  
79  
80     # 定义colorbar  
81     cax = plt.colorbar(mat,ticks=np.arange(minval,maxval-  
82  
83     # 添加类别名称  
84     if labels_str:  
85         cax.ax.set_yticklabels(labels_str)  
86  
87     if title:  
88         plt.suptitle(title, fontsize=14, fontweight='bold')
```

运行代码，得到如下结果，输出图片如图11-16~图11-19所示。

```
INFO:tensorflow:Restoring parameters from vgg_19_2016_08_28\
```



*****ebook converter DEMO Watermarks*****

图11-16 Vgg例子1的原始图片

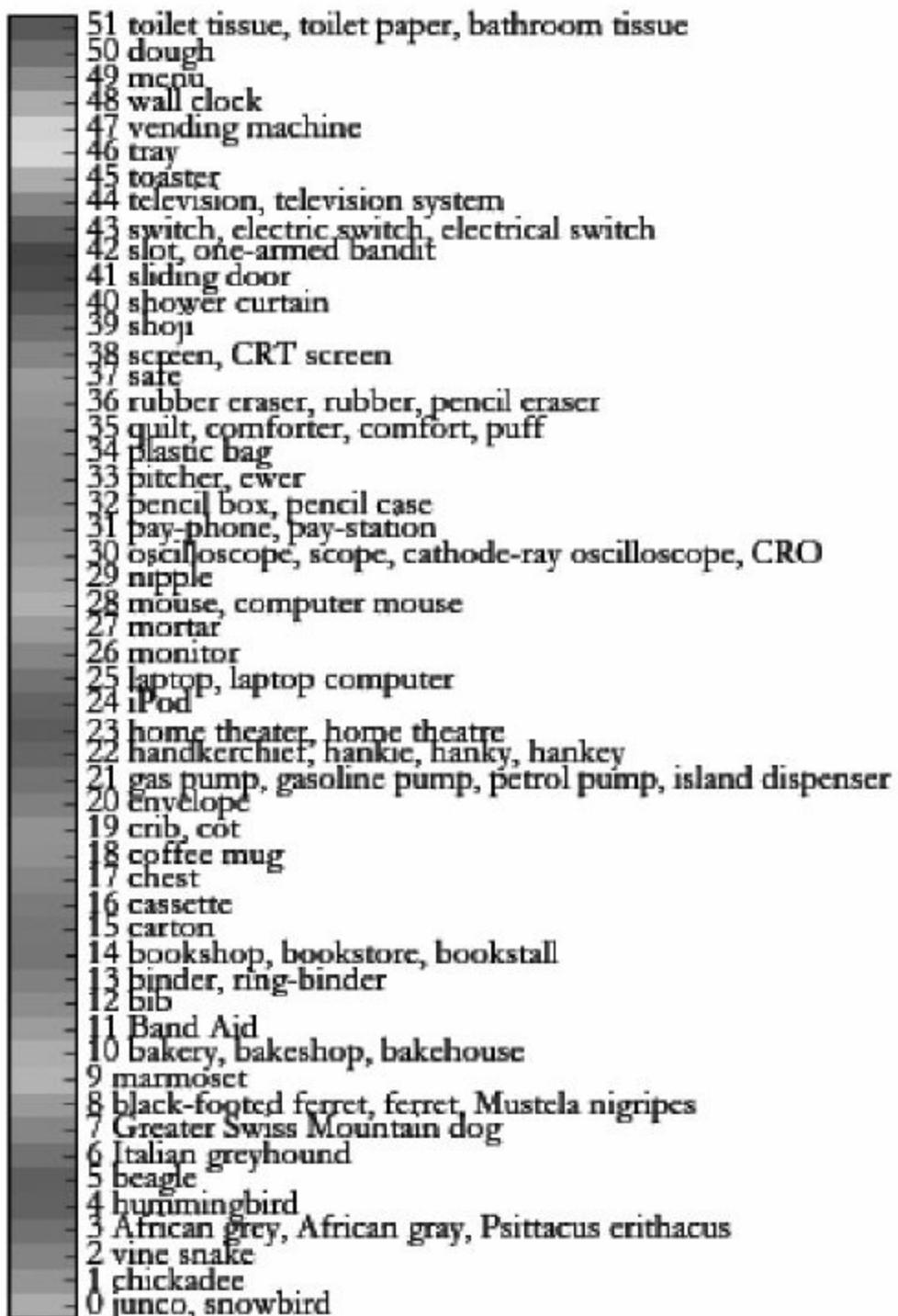


图11-17 Vgg例子1的识别结果

画面识别

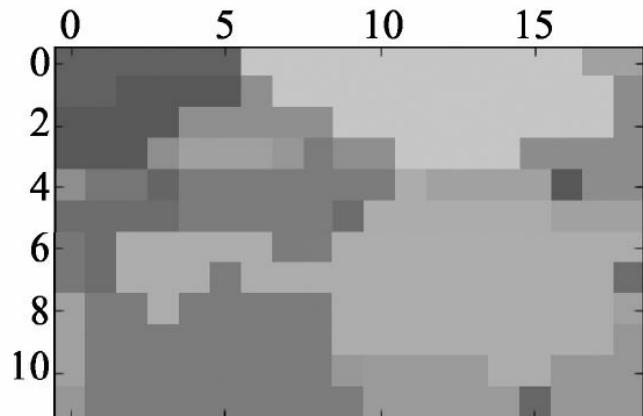


图11-18 Vgg例子2的原始图片

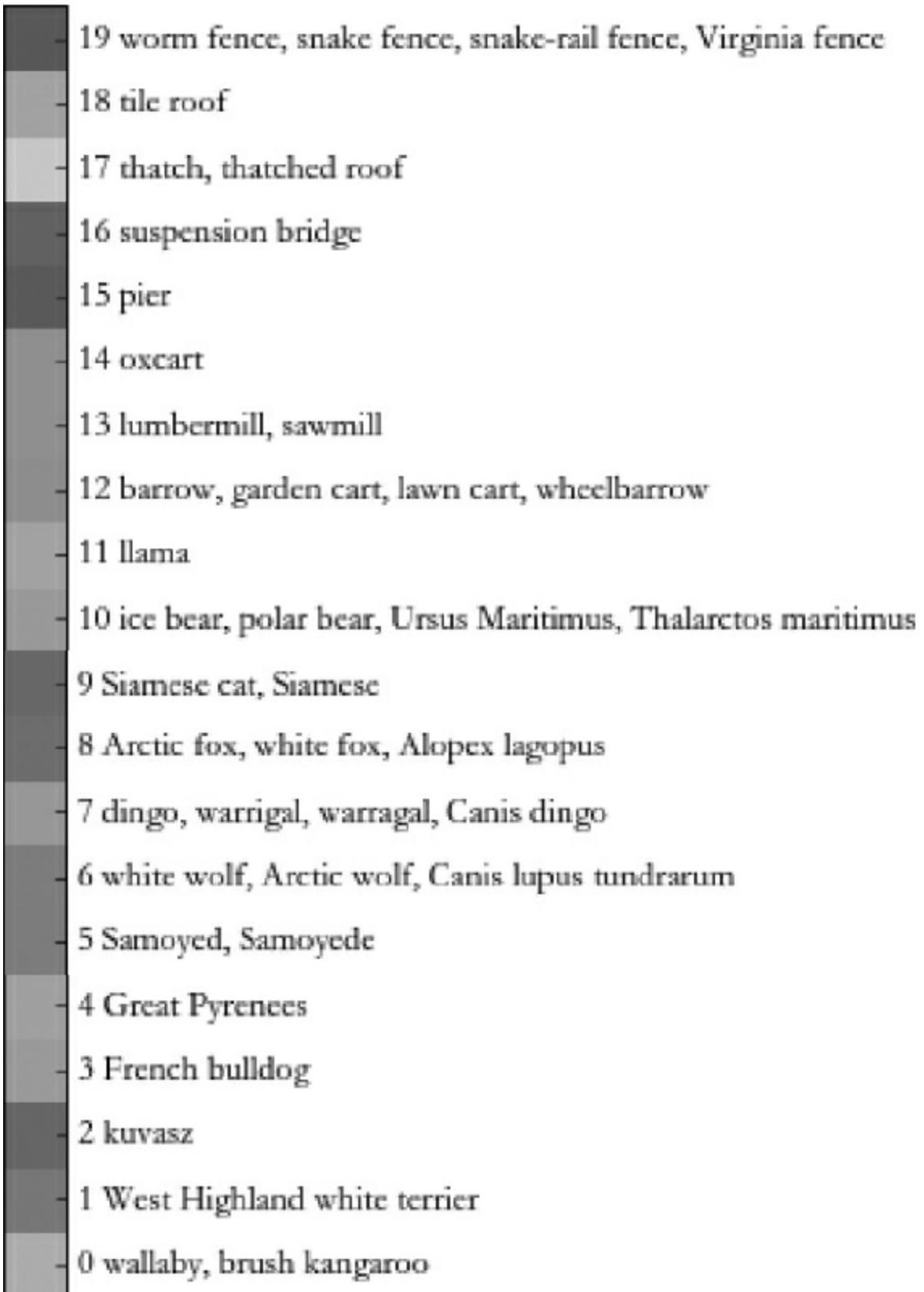


图11-19 Vgg例子2的识别结果

代码中将检测到的物体类别分别用不同的颜色来显示，并在图片上的对应位置做了标记。可以看到，对于元素较多的第一幅图片，VGG会识别出来更多的类型。

11.7 实物检测模型库——Object Detection API

Object Detection API是谷歌开放的一个内部使用的物体识别系统。2016年10月，该系统在COCO识别挑战中名列第一。它支持当前最佳的实物检测模型，能够在单个图像中定位和识别多个对象。该系统不仅用于谷歌于自身的产品和服务，还被推广至整个研究社区。

1. 代码位置与内置的模型

Object Detection模块的位置与slim的位置相近，同在github.com中TensorFlow的models\research目录下。类似slim，Object Detection也囊括了各种关于物体检测的各种先进模型：

- 带有MobileNets的SSD（Single Shot Multibox Detector）。
- 带有Inception V2的SSD。
- 带有Resnet 101的R-FCN（Region-Based Fully Convolutional Networks）。
- 带有Resnet 101的Faster RCNN。

- 带有Inception-Resnet v2的Faster RCNN。

上述每一个模型的冻结权重（在COCO数据集上训练）可被直接加载使用。

SSD模型使用了轻量化的 MobileNet，这意味着它们可以轻而易举地在移动设备中实时使用。谷歌使用了Fast RCNN模型需要更多计算资源，但结果更为准确。

2. COCO数据集介绍

在实物检测领域，训练模型的最权威数据集就是COCO数据集。

COCO数据集是微软发布的一个可以用来进行图像识别训练的数据集，官方网址为<http://mscoco.org/>。其图像主要从复杂的日常场景中截取，图像中的目标通过精确的segmentation进行位置的标定。

COCO数据集包括91类目标，分两部分发布，前部分于2014年发布，后部分于2015年发布。

- 2014年版本：训练集有82783个样本，验证集有40504个样本，测试集有40775个样本，有270KB的人物标注和886KB的物体标注。

- 2015年版本：训练集有165482个样本，验证集有81208个样本，测试集有81434个样本。

11.7.1 准备工作

1. 获取protobuf

Object Detection API使用protobufs来配置模型和训练参数，这些文件以“.proto”的扩展名放在models\research\object_detection\protos下。在使用框架之前，必须使用protobuf库将其编译成py文件才可以正常运行。protobuf库使用的是2.6版本，下载地址为<https://github.com/google/protobuf/releases/tag/v2.6.1>。

进入网址后会看到如图11-20所示，单击相应链接即可下载。

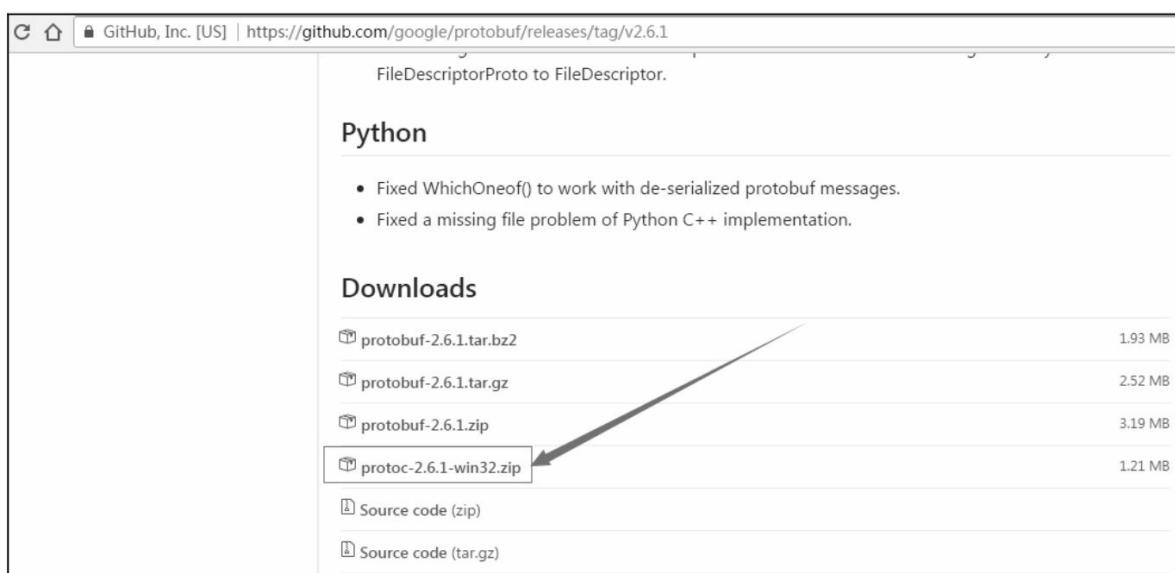


图11-20 protobuf下载包

protoc-2.6.1-win32.zip文件是个绿色程序，可以直接在命令行里运行。下载并解压后将其放到models\research路径下（假设你已经完成了在11.5.1中下载models的步骤）。

2. 编译proto配置文件

来到命令行里，进入models\research目录（如笔者的目录是D:\own\python\models\research）下，执行如下命令：

```
D:\own\python\models\research>protoc.exe object_detection\p
```

如果不显示任何信息，则表明运行成功了。为了检验成功效果，可以来到D:\own\python\models\research\object_detection\protos下，如图11-21所示，可以看到生成了很多py文件。

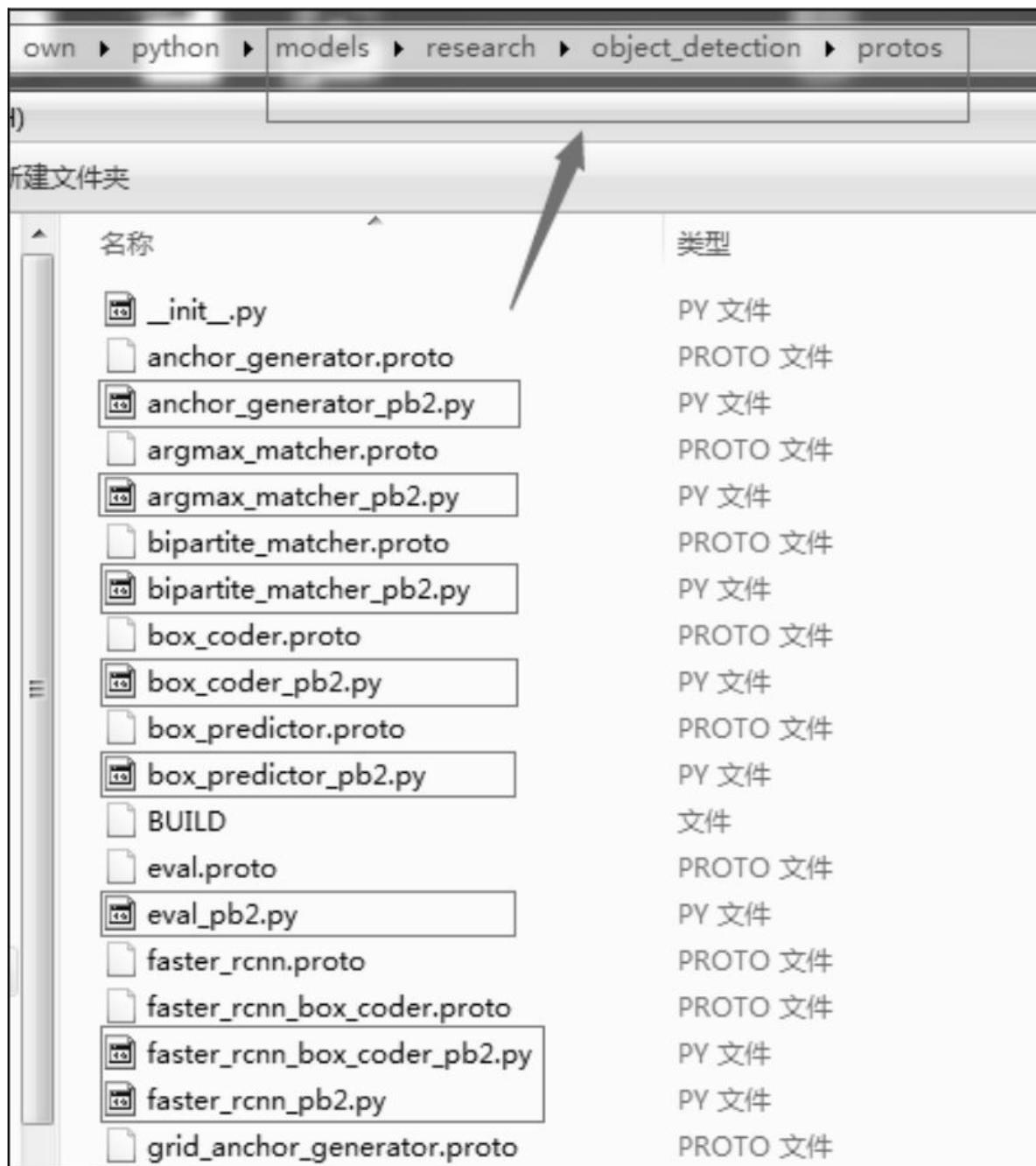


图11-21 编译protos

3. 检测API是否正常

如果前面两步都完成了，下面可以测试一下 Object Detection API 是否可以正常使用了，还需要两步操作：

(1) 将models\research\slim中的nets文件夹复制出来放到models\research下。

(2) 将models\research\object_detection\builders下的model_builder_test.py复制到models\research下。

变成如图11-22所示的文件夹结构。

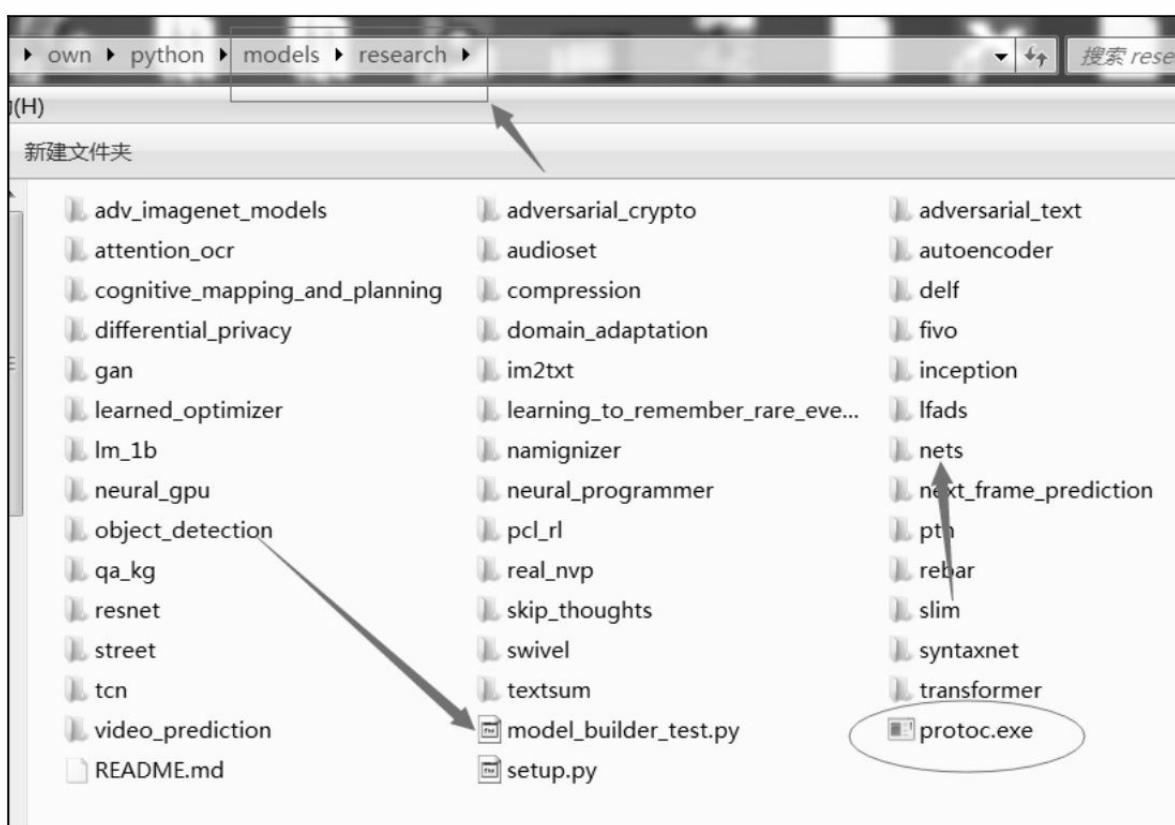


图11-22 Object Detection配置的文件结构

用spyder将model_builder_test.py/research文件打开，运行之后会看到如下信息：

```
runfile('D:/own/python/models/research/model_builder_test.py',  
       own/python/models/research')
```

*****ebook converter DEMO Watermarks*****

```
Reloaded modules: object_detection.protos.box_predictor_pb2,
detection.core, object_detection.anchor_generators.multiple_
object_detection.core.box_list_ops, object_detection.protos.
detection.builders.box_predictor_builder, object_detection.u1
utils, object_detection.core.box_list, object_detection.anchor_
object_detection.box_coders, object_detection.box_coders.mean_
coder, object_detection.protos, object_detection.protos.hyper_
object_detection.core.box_coder, object_detection.protos.ssd_
generator_pb2, object_detection.meta_architectures, object_
core.model, object_detection.protos.square_box_coder_pb2, obje
core.post_processing, object_detection.box_coders.faster_rcnn_
object_detection.protos.grid_anchor_generator_pb2, object_de
matcher_pb2, object_detection.models, object_detection.protos
generator_pb2, object_detection.matchers.bipartite_matcher, obj
core.preprocessor, object_detection.meta_architectures.ssd_me
object_detection.protos.post_processing_pb2, object_detection.
argmax_matcher_pb2, object_detection.core.anchor_generator, obj
utils.ops, object_detection.matchers, object_detection.matchers_
matcher, object_detection.protos.faster_rcnn_box_coder_pb2, obj
builders.region_similarity_calculator_builder, object_detecti
builders.image_resizer_builder, object_detection.core.matcher_
detection.meta_architectures.faster_rcnn_meta_arch, object_d
core.minibatch_sampler, object_detection.core.balanced_posit
sampler, object_detection.protos.bipartite_matcher_pb2, object_
core.keypoint_ops, object_detection.protos.region_similarity_
pb2, object_detection.builders.matcher_builder, object_detecti
losses_builder, object_detection.meta_architectures.rfcn_meta_
detection.core.box_predictor, object_detection.builders.box_
object_detection.box_coders.square_box_coder, object_detectio
mean_stddev_box_coder_pb2, object_detection.utils.variables_
detection, object_detection.core.region_similarity_calculator_
detection.builders.post_processing_builder
.....
-----
Ran 7 tests in 0.047s

OK
To exit: use 'exit', 'quit', or Ctrl-D.
An exception has occurred, use %tb to see the full traceback.

SystemExit: <sitecustomize.IPyTesProgram object at 0x0000002E
```

表明Object Detection API一切正常， 可以使
用了。

*****ebook converter DEMO Watermarks*****

4. 将Object Detection API加入Python库默认搜索路径

为了不用每次都将文件复制到Object Detection文件夹外，可以将Object Detection加到Python引入库的默认搜索路径中，将Object Detection文件夹整个复制到anaconda3安装文件目录下的lib\site-packages下，如图11-23所示。



图11-23 Object Detection安装

这样无论文件在哪里，只要搜索import Object Detection xxx，系统都会找到Object Detection了。

11.7.2 实例87：调用Object Detection API进行实物检测

下面用一个例子来测试下Object Detection API中的检测效果。该例子改编于Object Detection API的自带程序，使用的图片也是Object Detection API中的图片。具体步骤如下。

实例描述

使用Object Detection API基于COCO上训练的ssd_mobilenet_v1模型，对任意图片进行分类识别。

1. 下载模型

上面介绍的已有模型，在以下网址都可以下载https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md。

下载模型如图11-24所示。

Model name	Speed	COCO mAP	Outputs
ssd_mobilenet_v1_coco	fast	21	Boxes
ssd_inception_v2_coco	fast	24	Boxes
rfcn_resnet101_coco	medium	30	Boxes
faster_rcnn_resnet101_coco	medium	32	Boxes
faster_rcnn_inception_resnet_v2_atrous_coco	slow	37	Boxes

图11-24 下载Detection模型

每一个压缩文件里都包含如下3种文件：

- 放置权重的检查点文件。

- 描述网络变量的txt文件。

· 可用于变量载入内存的图frozen文件。该文件与检查点结合可以实现“开箱即用”的使用理念，即不需要如前面例子中再引入一次网络模型源码文件。

2. 载入模型及数据集样本标签

在Object Detection文件夹下新建一个py文件，编写如下代码。

代码中首先加载引入库。然后指定检测点文件及相关路径，将*.pb文件读入serialized_graph中，重新定义一个图od_graph_def，使用其ParseFromString方法将serialized_graph的内容恢复到图中，接着再使用tf.import_graph_def将od_graph_def的内容导入到当前的默认图中。

代码11-4 Object Detection使用

```
01 import numpy as np
02 import os
03
04 import tensorflow as tf
05 from matplotlib import pyplot as plt
06 from PIL import Image
07 from object_detection.utils import label_map_util
08
09 from object_detection.utils import visualization_utils as
10
11 # 指定要使用模型的名字
12 MODEL_NAME = 'ssd_mobilenet_v1_coco_11_06_2017'
13
14 # 指定模型的路径
15 PATH_TO_CKPT = MODEL_NAME + '/frozen_inference_graph.pb'
16
```

```
17 # 数据集对应的label
18 PATH_TO_LABELS = os.path.join('data', 'mscoco_label_map.pbtxt')
19
20 NUM_CLASSES = 90
21
22 tf.reset_default_graph()
23
24 od_graph_def = tf.GraphDef()
25 with tf.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
26     serialized_graph = fid.read()
27     od_graph_def.ParseFromString(serialized_graph)
28     tf.import_graph_def(od_graph_def, name='')
29 #载入coco数据集标签文件
30 label_map = label_map_util.load_labelmap(PATH_TO_LABELS)
31 categories = label_map_util.convert_label_map_to_categories(
    label_map, max_num_classes=NUM_CLASSES, use_display_name=True)
32 category_index = label_map_util.create_category_index(categories)
```

在Object Detection模块中有一个data文件夹，里面为放置好的coco数据集对应的标签txt文件和其他的数据集标签文件（pascal与pet数据集）。使用Object Detection自带的label_map_util类可以将其以index的方式读入内存中。

3. 定义session加载待测试的图片文件

本例也使用Object Detection自带的测试图片来演示。该图片存放在Object Detection\test_images中，一共有两张。当然读者也可以自己再添加图片进行测试，但要修改对应的名字和代码。

代码11-4 Object Detection使用（续）

```
33 def load_image_into_numpy_array(image):
```

*****ebook converter DEMO Watermarks*****

```
34     (im_width, im_height) = image.size
35     return np.array(image.getdata()).reshape(
36         (im_height, im_width, 3)).astype(np.uint8)
37
38 PATH_TO_TEST_IMAGES_DIR = 'test_images'
39 TEST_IMAGE_PATHS = [ os.path.join(PATH_TO_TEST_IMAGES_DIR,
40                         '{}.jpg'.format(i)) for i in range(1, 3) ] #将要测试的图片路径放到数组
41 # 设置输出图片的大小
42 IMAGE_SIZE = (12, 8)
43
44 detection_graph = tf.get_default_graph()
45 with tf.Session(graph=detection_graph) as sess:
46     for image_path in TEST_IMAGE_PATHS:
47         image = Image.open(image_path)
48
49         image_np = load_image_into_numpy_array(image)
```

本例中新建立了一个图，为了不易混淆，可通过get_default_graph获得当前的默认图，接下来在默认的图上建立session并进行测试。

4. 定义节点，运行结果并可视化

下面可以体验一下Object Detection中的“开箱即用”概念。因为在前面已经将变量导入图中了，所以这里不需要再定义一套变量，直接通过get_tensor_by_name拿到变量并使用即可。这种方式将模型与应用很好的解耦，做应用的人不再需要了解模型的结构，只需关心输入输出和模型文件，做模型的人也不用担心模型代码被误改导致功能失效。

代码11-4 Object Detection使用（续）

```
50 # 扩充维度 shape, 变成: [1, None, None, 3]
51     image_np_expanded = np.expand_dims(image_np, axis=0)
52     image_tensor = detection_graph.get_tensor_by_name('
53         tensor:0')
54         # boxes用来显示识别结果
55         boxes = detection_graph.get_tensor_by_name('detecti
56         # Each score代表识别出的物体与标签匹配的相似程度, 在类型标
57         scores = detection_graph.get_tensor_by_name('detecti
58         scores:0')
59         classes = detection_graph.get_tensor_by_name('detec
60         classes:0')
61         num_detections = detection_graph.get_tensor_by_name(
62             'detections:0')
63         # 开始检测
64         (boxes, scores, classes, num_detections) = sess.run(
65             [boxes, scores, classes, num_detections],
66             feed_dict={image_tensor: image_np_expanded})
67         # 可视化结果
68         vis_util.visualize_boxes_and_labels_on_image_array(
69             image_np,
70             np.squeeze(boxes),
71             np.squeeze(classes).astype(np.int32),
72             np.squeeze(scores),
73             category_index,
74             use_normalized_coordinates=True,
75             line_thickness=8)
76         plt.figure(figsize=IMAGE_SIZE)
77         plt.imshow(image_np)
```

模型的检测结果有3个输出，一个是位置
boxes、一个是类型，另一个是分数。得到这3个
输出后调用Object Detection中的
visualize_boxes_and_labels_on_image_array函数，
将图片显示出来。运行代码，输出如图11-25所
示。

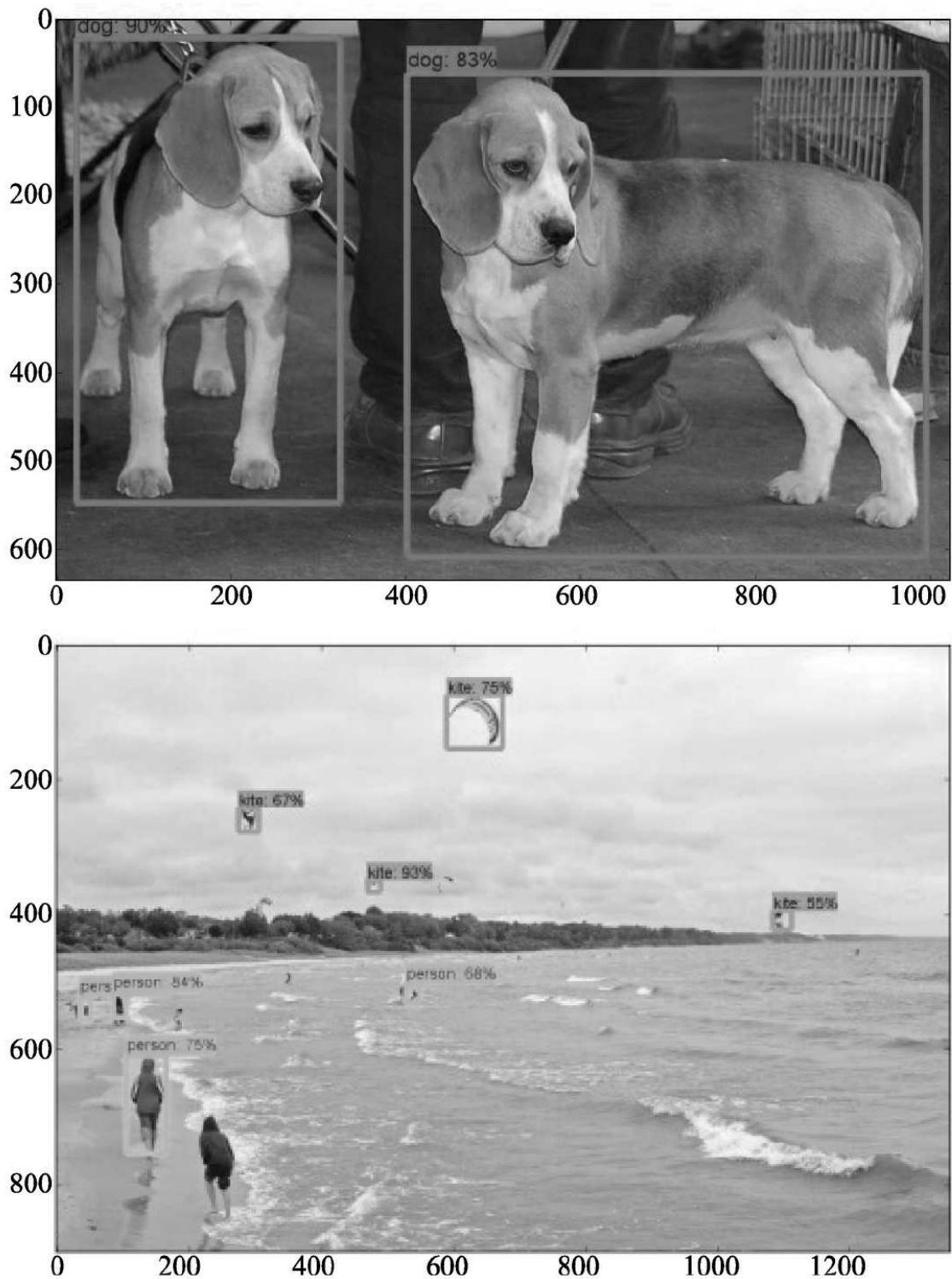


图11-25 实物检测例子运行结果

 注意：如何得到里面的变量名？一般会在提供模型时会给出对应例子；如果没有，则可*****ebook converter DEMO Watermarks*****

以在代码中找到相应的模型定义；也可以通过在代码中添加`print (detection_graph.get_operations ())` 将图中所有的变量打印出来，第一行就可以找到占位符，最后一行也可以找到输出结果。

11.8 实物检测领域的相关模型

前面的预置模型都属于实物检测领域的优秀模型，也是比较成熟的模型。本节来介绍一下该领域的其他相关模型知识。

11.8.1 RCNN基于卷积神经网络特征的区域方法

实物检测领域的基础模型需要从RCNN（regions with CNN）说起。RCNN模型可以理解为，增加特征的穷举范围，然后在其中发现有价值的特征。大概步骤如下。

(1) 对于一幅输入的图片，通过选择性搜索，找出2000个候选窗口。

(2) 利用CNN对它们提取特征向量，即将这2000个子图片统一缩放到 227×227 ，然后进行卷积操作。

(3) 利用SVM算法对特征向量进行分类识别。

RCNN中对每一类都进行SVM训练，根据输出的特征类为每一个区打分，最终决定保留或拒绝该区域特征。

11.8.2 SPP-Net：基于空间金字塔池化的优化RCNN方法

RCNN这种海量的穷举方法显然会带来巨大的计算量，有一种优化办法是使用空间金字塔池化方法。

空间金字塔池化（Spatial Pyramid Pooling, SPP）最大的特点是，不再关心输入图片的尺寸，而是根据最后的输出类别个数，通过算法来生成多个不同范围的池化层，由它们对输入进行并行池化处理，使最终的输出特征个数与生成类别个数相等，接着再进行类别的比较和判定。

由这样的技术产生的网络叫做SPP-Net。该网络只需要计算完整图像的特征图（feature maps）一次，然后通过池化子窗口的特征，来保持固定长度的输出，比RCNN先划分窗口再对每个窗口进行卷积的效率要快30~170倍，并且有更好的准确率。

11.8.3 Fast-R-CNN快速的RCNN模型

Fast-R-CNN在SPP-Net基础上进行了改进，并将它嫁接到VGG16上所形成的网络，将SPP改成RoI Layer pooling层，并且不再使用SVM分类器，而是通过Softmax Classifier和Bounding-Box Regressors联合训练的方式来更新所有参数，实现*****ebook converter DEMO Watermarks*****

了整个网络端到端的训练。

RoI Pooling Layer可以理解为SPP-Layer的简化形式。SPP-Layer中会包含不同尺度的池化层；而RoI Layer只包含一种尺度，它是先将图片进行相同尺度的裁分，每个子块就成为RoI，然后对所有的RoI进行单独的Max-Pool，得到每个Block的最大值。

Fast-R-CNN保留了VGG16中的第5个池化层之前的网络，后面接上自己的RoI Pooling Layer，然后通过全连接层进行softmax分类，最终形成了整个网络。其结构可以简单描述为：“13个卷积层+4个Pooling层+RoI层+2个FC层+两个平级层”（即SoftmaxLoss层和SmoothL1Loss层）。

后来人们习惯在其前面加上一个RPN网络，用来对图片进行一次候选框的筛选，所以整个网络结构会变成“RPN+Fast-R-CNN”的形式。

所谓的RPN（Region Proposal Network）是指，先使用 $n \times n$ 的滑块窗口在原图像上扫描，生成M个特征值，将这M个特征值接到两个卷积网络reglayer与classlayer中输出。Reglayer里面包含图像坐标的x、y与长宽，classlayer里面有判断这部分是前景还是背景的标志值。在训练时，一个Mini-batch是由一幅图像中任意选取的256个候选框组成的，其中正、负样本的比例为1：1。如果

正样本不足128，则多用一些负样本，以满足有256个Proposal可以用于训练。对于正、负样本的标注是，reglayer范围内对应的classlayer的重合度大于0.7（即为正样本），如果都不大于0.7，则其中的最大值为正样本。最终通过softmax loss和regression loss按照一定权重比例计算loss。

11.8.4 YOLO：能够一次性预测多个位置和类别的模型

使用滑窗（即前景背景）时，RPN常常把背景区域误检为特定目标。所以YOLO（You Only Look Once）使用了全新的训练方式筛选候选框的筛选——采用整图的方式来训练模型，并且可以一次性预测多个Box的位置和类别。

YOLO的方式是，先将图片分为 $S \times S$ 个网格，每个网格相当于一个任务，负责检测内部是否有物体的中心点落入该区域，一旦有的话，则启动该任务来检测n个bounding boxes对象。

bounding boxes由中心点坐标（x, y）、宽高（w, h）和置信度评分这5部分组成。置信度评分可以理解为当前网格内物体属于该类别的概率与真实和预测区域的重叠度的乘积。

例如，如果一共有4类物体，那么每个网格里面就会有该物体对应的这4个类的概率（p0,

p_1, p_2, p_3 ），同时通过bounding boxes的位置信息（ x, y, w, h ）可以知道其预测区域，并算出与对于类别真实区域的重叠度（Iou1、Iou2、Iou3、Iou4），二者相乘就可以得到置信度。这样，如果有9个网格（ 7×7 ），每个网格负责找到2个bounding boxes，每个bounding boxes内部由5个元素组成，而且每个网格还需要有对应10个类别的概率，如式子 $49 \times (2 \times 5 + 10) = 1470$ 个特征值。YOLO网络通过预测该特征值的训练，来实现实物的识别检测。

对于这1470个特征值的loss计算，并没有用常用的平方差等方法，原因是大多数网格实际不包含物体（即很多网格的分类概率为0），这会出现位置误差正常、分类误差稀疏的情况。



提示： 106维度的数据内部存在着某部分维度分布不均的情况，如直接用平方差会使整体的loss很不稳定，所以这部分也采用了更复杂的算法，这里不再展开。

YOLO网络结构分为两种：一个是正常的网络结构，用到了Inception的结构；另一个是其简化版，会有更好的速度，但是准确度会降低。

11.8.5 SSD：比YOLO更快更准的模型

前面讲的YOLO也有缺陷：

- 每个网格预测的物体个数是指定的，容易造成遗漏（如指定检测2个，但是实际有3个）。
- 对物体的尺度相对比较敏感，对尺度变化较大的物体泛化能力较差。

而SSD（Single Shot MultiBox Detector）的方法在YOLO的基础上融合了RPN的思想，在不同卷积层所输出的不同尺度的卷积结果（Feature Map）上面划格子，在多种尺度的格子上提取目标中心点，从而大大改善了这两个问题。

类似于Fast-R-CNN，SSD网络使用的是基于VGG 16改进的模型结构。

11.8.6 YOLO2：YOLO的升级版模型

YOLO2，在YOLO的基础上也改掉了很多缺陷，去掉了网格与类别的预测绑定在一起，也使用了anchor box模式。另外，在一些结构细节上做了一些优化：更多地使用了卷积来代替全连接网络，并增加了BN算法，同时提升了网络的入口分辨率，去掉最后池化层，保证有更好的分辨率等。同样，YOLO 2沿用了基于GoogLeNet的自定制网络，也使用了Inception（见11.2.3节至11.2.7节）中的很多最新技术，算是目前最好的实物检

测模型了。

*****ebook converter DEMO Watermarks*****

11.9 机器自己设计的模型（NASNet）

NASNet是谷歌公司AutoML项目产出的模型。AutoML项目是一种实现机器学习模型设计自动化的项目，致力于让计算机设计出性能可与人类专家设计的神经网络相媲美的神经网络。而NASNet就是该项目的产出成果。NASNet架构在CIFAR-10、ImageNet分类和COCO实物检测上都优于现有的开源模型。

NASNet架构由两种类型的层组成：正常层和还原层。下面引用在谷歌的博客（Google Research Blog）上公开NASNet的结构，如图11-26所示。

根据初始化NASNet结构的不同规模，TensorFlow中提供了两种版本的NASNet，即large NASNet model与mobile NASNet model。large NASNet model可实现最高的准确率，适用于在后端服务器上应用；mobile NASNet model是一个小规模模型。在保留了原有74%的准确率基础上，将计算开销控制在非常低的水平，适用于在移动平台上应用。

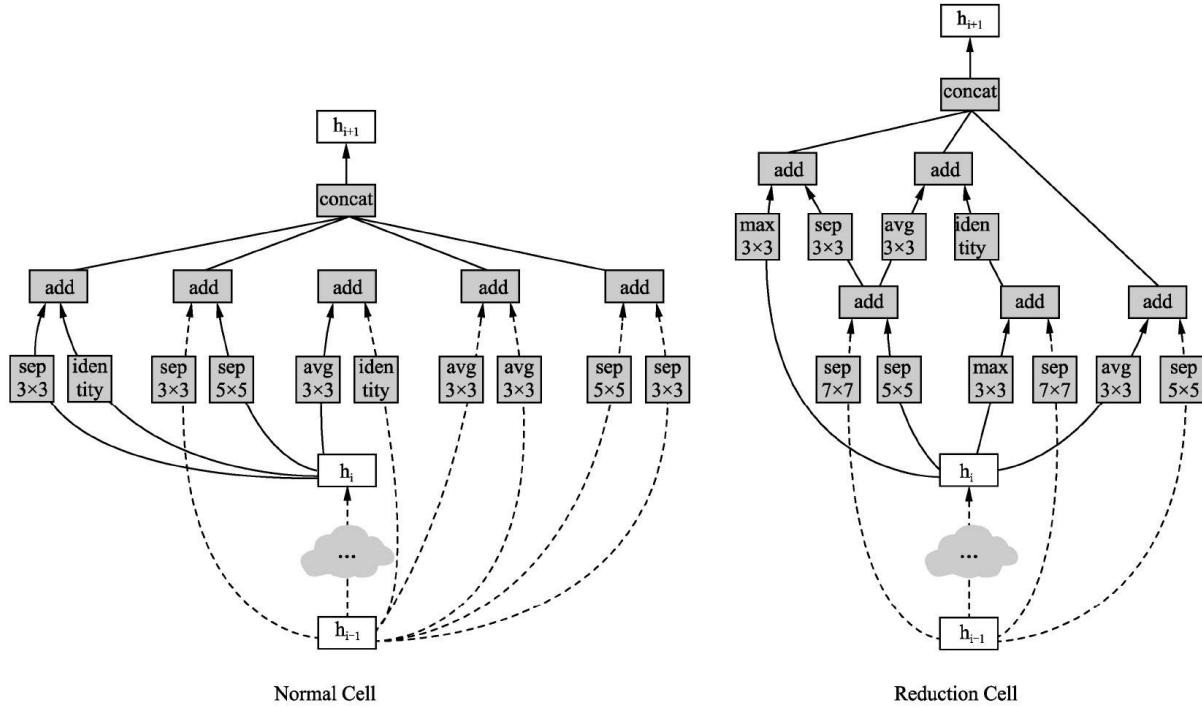


图11-26 NASNet结构：正常层（左）、还原层（右）

更多信息可以参考如下链接：

<https://github.com/tensorflow/models/blob/master/research/nasnet/>

- 该链接中提供了NASNet两种版本的预编模型，并且NASNet结构的实现代码在以下链接里也可以找到：

<https://github.com/tensorflow/models/blob/master/research/nasnet/>

- 该模型代码中提供了两个初始化规模的函数 `_large_imagenet_config` 和 `_mobile_`

`imagenet_config`, 分别对应于large NASNet model与mobile NASNet model两种模型。读者可以将本章介绍的模型使用例子套用到NASNet模型的使用上。在实际工作中，遇到图片分类问题时，建议优先考虑NASNet模型。

第12章 对抗神经网络（GAN）

对抗神经网络其实是两个网络的组合，可以理解为一个网络生成模拟数据，另一个网络判断生成的数据是真实的还是模拟的。生成模拟数据的网络要不断优化自己让别的网络判断不出来，别的网络也要优化自己让自己判断得更准确。二者关系形成对抗，因此叫对抗神经网络。

实验证明，利用这种网络间的对抗关系所形成的网络，在无监督及半监督领域取得了很好的效果，可以算是用网络来监督网络的一个自学习过程。

下面我们就来系统地学习对抗神经网络的相关知识。

本章含有教学视频共22分11秒。

作者按照本章的内容结构，对主要内容进行了讲解，包括基本神经网络的概念和结构，以及在此之上的其他几种GAN网络模型的结构部分等（重点是最后一个实例，以及对SRGAN的掌握）。

深度学习之TensorFlow

入门、原理与进阶实战

第12章 对抗神经网络（GAN）

配套视频



代码医生

qq群: 40016981

Blog: <http://blog.csdn.net/ljlin6249>



字 幕



*****ebook converter DEMO Watermarks*****

12.1 GAN的理论知识

GAN由generator（生成式模型）和discriminator（判别式模型）两部分构成。

- generator：主要是从训练数据中产生相同分布的samples，对于输入 x ，类别标签 y ，在生成式模型中估计其联合概率分布（两个及以上随机变量组成的随机向量的概率分布）。
- discriminator：判断输入是真实数据还是generator生成的数据，即估计样本属于某类的条件概率分布。它采用传统的监督学习的方法。

二者结合后，经过大量次数的迭代训练会使generator尽可能模拟出以假乱真的样本，而discriminator会有更精确的鉴别真伪数据的能力，最终整个GAN会达到所谓的纳什均衡，即discriminator对于generator的数据鉴别结果为正确率和错误率各占50%。

GAN的网络结构如图12-1所示。

- 生成式模型又叫生成器。它先用一个随机编码向量来输出一个模拟样本（如图12-1左侧所示）。

· 判别式模型又叫判别器。它的输入是一个样本（可以是真实样本也可以是模拟样本），输出一个判断该样本是真样本还是模拟样本（假样本）的结果，如图12-1右侧所示。

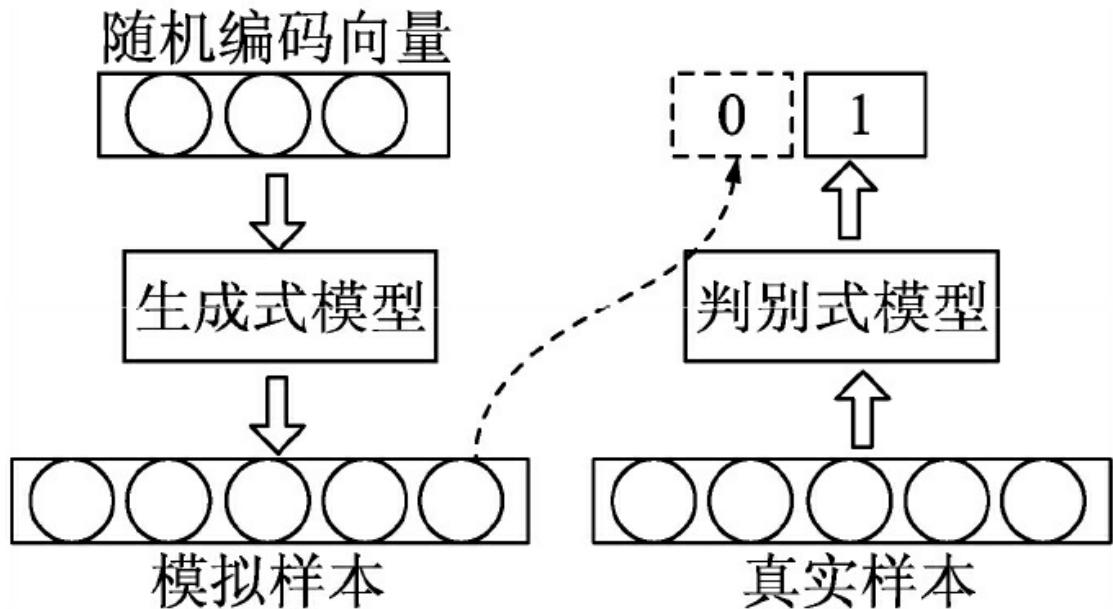


图12-1 GAN网络

判别器的目标是区分真假样本，生成器的目标是让判别器区分不出真假样本，两者目标相反，存在对抗。

我们前面学习的监督学习神经网络就属于discriminator。下面介绍generator。

12.1.1 生成式模型的应用

generator的特性主要包括以下几方面：

- 在应用数学和工程方面，能够有效地表征高维数据分布。

- 在强化学习方面，作为一种技术手段，有效表征强化学习模型中的state状态。

- 在半监督学习方面，能够在数据缺失下训练模型，并给出相应的输出。

generator还适用于一个输入伴随多个输出的场景下，如在视频中通过场景预测下一帧的场景，而discriminator通过最小化模型输出和期望输出的某个预测值，无法训练单输入多输出的模型。前面学习的自编码部分就属于一个generator。

12.1.2 GAN的训练方法

根据GAN的结构不同，会有不同的对应训练方法。无论什么方法，其原理是一样的，即在迭代训练的优化过程中进行两个网络的优化。有的会在一个优化步骤中对两个网络优化，有的会对两个网络采取不同的优化步骤。

12.2 DCGAN——基于深度卷积的GAN

DCGAN即使用卷积网络的对抗网络，其原理和GAN一样，只是把CNN卷积技术用于GAN模式的网络里，G（生成器）网在生成数据时，使用反卷积的重构技术来重构原始图片。D（判别器）网用卷积技术来识别图片特征，进而作出判别。

同时，DCGAN中的卷积神经网络也做了一些结构的改变，以提高样本的质量和收敛速度：

- G网中取消所有池化层，使用转置卷积(transposed convolutional layer) 并且步长大于等于2进行上采样。
- D网中也用加入stride的卷积代替pooling。
- 在D网和G网中均使用批量化归一(batch normalization)，而在最后一层时通常不会使用batch normalization，这是为了保证模型能够学习到数据的正确均值和方差。
- 去掉了FC层，使网络变为全卷积网络。
- G网中使用ReLU作为激活函数，最后一层使用Tanh作为激活函数。

- D网中使用LeakyReLU作为激活函数。

DCGAN中换成了两个卷积神经网络(CNN)的G和D，可以更好地学到对输入图像层次化的表示，尤其在生成器部分会有更好的模拟效果。DCGAN在训练中会使用Adam优化算法。

12.3 InfoGAN和ACGAN：指定类别生成模拟样本的GAN

InfoGAN是一种把信息论与GAN相融合的神经网络，能够使网络具有信息解读功能。下面来一起看看它的介绍。

12.3.1 InfoGAN：带有隐含信息的GAN

GAN的生成器在构建样本时使用了任意的噪声向量 z ，并从低维的噪声数据 z 中还原出来高维的样本数据。这说明数据 z 中含有具有与样本相同的特征。

由于随意使用的噪声都能还原出高维样本数据，表明噪声中的特征数据部分是与无用的数据部分高度地纠缠在一起的，即我们能够知道噪声中含有有用特征，但无法知道哪些是有用特征。

InfoGAN是GAN模型的一种改进，是一种能够学习样本中的关键维度信息的GAN，即对生成样本的噪音进行了细化。先来看它的结构，相比对抗自编码，InfoGAN的思路正好相反，InfoGAN是先固定标准高斯分布作为网络输入，再慢慢调整网络输出去匹配复杂样本分布。

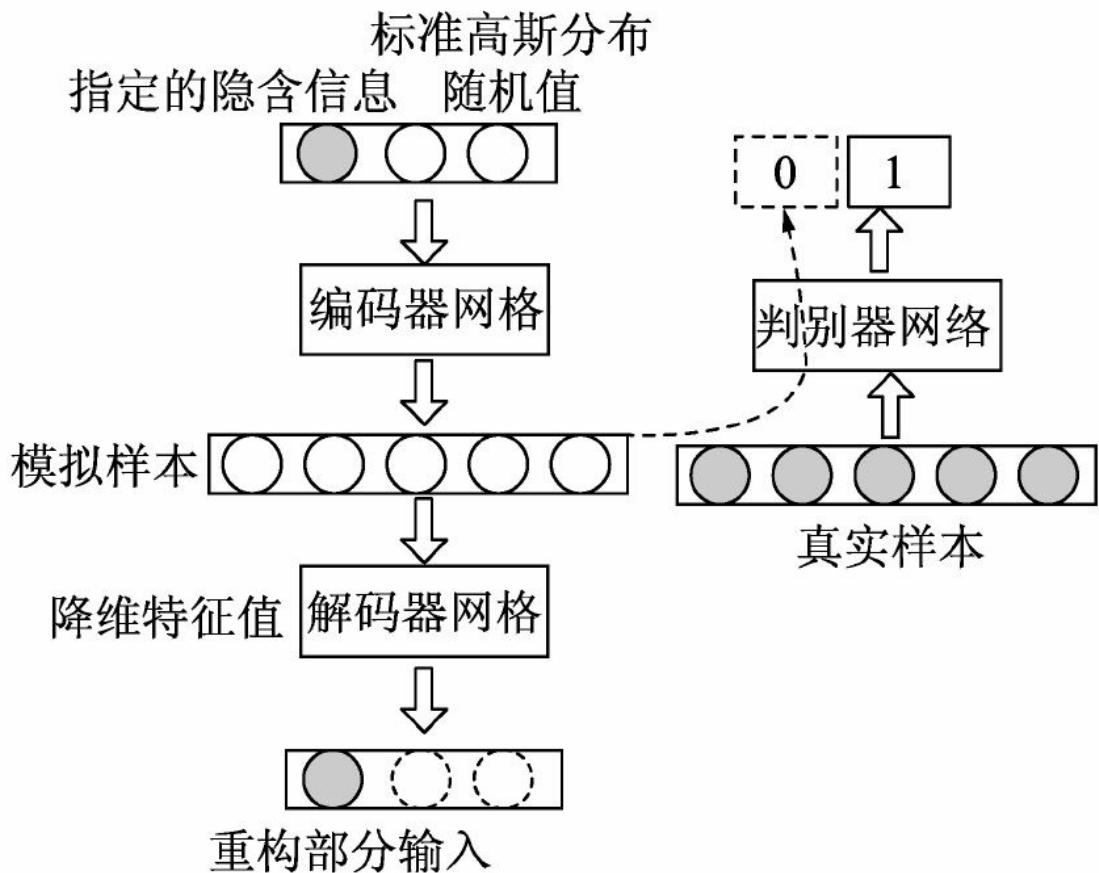


图12-2 InfoGAN模型

如图12-2所示，InfoGAN生成器是从标准高斯分布中随机采样来作为输入，生成模拟样本，解码器是将生成器输出的模拟样本还原回生成器输入的随机数中的一部分，判别器是将样本作为输入来区分真假样本。

InfoGAN的理论思想是将输入的随机标准高斯分布当成噪音数据，并将噪音分为两类，第一类是不可压缩的噪音 Z ，第二类是可解释性的信息 C 。假设在一个样本中，决定其本身的只有少量重要的维度，那么大多数的维度是可以忽略的。而这里的解码器可以更形象地叫成重构器，

即通过重构一部分输入的特征来确定与样本互信息的那些维度。最终被找到的维度可以代替原始样本的特征（类似PCA算法中的主成份），实现降维、解耦的效果。

12.3.2 AC-GAN：带有辅助分类信息的GAN

AC-GAN（Auxiliary Classifier GAN），即在判别器discriminator中再输出相应的分类概率，然后增加输出的分类与真实分类的损失计算，使生成的模拟数据与其所属的class一一对应。

一般来讲，AC-GAN可以属于InfoGAN的一部分，class信息可以作为InfoGAN中的潜在信息，只不过这部分信息可以使用半监督方式来学习。

12.3.3 实例88：构建InfoGAN生成MNIST模拟数据

本例演示在MNISTT数据集上使用InfoGAN网络模型生成模拟数据，并且加入标签信息的loss函数同时实现AC-GAN网络。其中的D和G都是用卷积网络来实现的，相当于DCGAN基础上的InfoGAN例子。

实例描述

通过使用InfoGAN网络学习MNIST数据特征，生成以假乱真的MNIST模拟样本，并发现内部潜在的特征信息。

具体实现可以分为如下几个步骤。

1. 引入头文件并加载MNIST数据

假设MNIST数据放在本地磁盘根目录的data下。本例中将使用前面介绍的slim模块构建网络结构，所以需要引入slim。当然也可以不用slim，引入slim的目的是为了编写代码比较方便，不用考虑输入维度即相关权重的定义，最主要的是slim还对反卷积有封装，后面会用到。

代码12-1 Mnistinfogan

```
01 import numpy as np
02 import tensorflow as tf
03 import matplotlib.pyplot as plt
04 from scipy.stats import norm
05 import tensorflow.contrib.slim as slim
06
07 from tensorflow.examples.tutorials.mnist import input_data
08 mnist = input_data.read_data_sets("/data/")#, one_hot=True
```

2. 网络结构介绍

建立两个噪声数据（一般噪声和隐含信息）与label结合放到生成器中，生成模拟样本，然后将模拟样本和真实样本分别输入到判别器中，生

成判别结果、重构构造的隐含信息，以及样本标签。

在优化时，让判别器对真实的样本判别结果为1、对模拟数据的判别结果为0来做损失值计算（loss）；对生成器让判别结果为1来做损失值计算（loss）。

3. 定义生成器与判别器

由于是先从模拟噪声数据来恢复样本，所以在生成器中要使用反卷积函数。这里通过“两个全连接+两个反卷积”模拟样本的生成，并且每一层都有BN（批量归一化）处理。

代码12-1 Mnistinfogan（续）

```
09 def generator(x):#生成器函数
10     reuse = len([t for t in tf.global_variables() if t.name == 'generator']) > 0
11
12     with tf.variable_scope('generator', reuse=reuse):
13         x = slim.fully_connected(x, 1024)
14
15         x = slim.batch_norm(x, activation_fn=tf.nn.relu)
16         x = slim.fully_connected(x, 7*7*128)
17         x = slim.batch_norm(x, activation_fn=tf.nn.relu)
18         x = tf.reshape(x, [-1, 7, 7, 128])
19
20         x = slim.conv2d_transpose(x, 64, kernel_size=[4, 4],
21                                   activation_fn=None)
22
23         x = slim.batch_norm(x, activation_fn=tf.nn.relu)
24         z = slim.conv2d_transpose(x, 1, kernel_size=[4, 4],
25                                   activation_fn=tf.nn.sigmoid)
```

```

25     return z
26
27 def leaky_relu(x):
28     return tf.where(tf.greater(x, 0), x, 0.01 * x)
29 #判别器函数
30 def discriminator(x, num_classes=10, num_cont=2):
31     reuse = len([t for t in tf.global_variables() if t.name == ('discriminator')]) > 0
32
33     with tf.variable_scope('discriminator', reuse=reuse):
34         x = tf.reshape(x, shape=[-1, 28, 28, 1])
35         x = slim.conv2d(x, num_outputs=64, kernel_size=[5, 5], stride=2, activation_fn=leaky_relu)
36         x = slim.conv2d(x, num_outputs=128, kernel_size=[5, 5], stride=2, activation_fn=leaky_relu)
37
38         x = slim.flatten(x)
39         shared_tensor = slim.fully_connected(x, num_outputs=1024, activation_fn=leaky_relu)
40         recog_shared = slim.fully_connected(shared_tensor, num_outputs=128, activation_fn=leaky_relu)
41         disc = slim.fully_connected(shared_tensor, num_outputs=1, activation_fn=None)
42         disc = tf.squeeze(disc, -1)
43
44         recog_cat = slim.fully_connected(recog_shared, num_classes, activation_fn=None) #判别类型
45         recog_cont = slim.fully_connected(recog_shared, num_cont, activation_fn=tf.nn.sigmoid) #判别info
46     return disc, recog_cat, recog_cont

```

如果判别器输入的是真正的样本，同样也要经过两次卷积，再接两次全连接，生成的数据可以分别连接不同的输出层产生不同的结果，其中1维的输出层产生判别结果1或是0，10维的输出层产生分类结果，2维输出层产生隐含维度信息。



注意： 在生成器与判别器中都会使用各自的命名空间，这是在多网络模型里定义变量的

一个好习惯。在指定训练参数、获取及显示训练参数时，都可以通过指定的命名空间来拿到对应的变量，不至于混乱。

4. 定义网络模型

令一般噪声的维度为38，应节点为z_rand；隐含信息维度为2，应节点为z_con，二者都是符合标准高斯分布的随机数。将它们与one_hot转换后的标签连接在一起放到生成器中。

代码12-1 Mnistinfogan（续）

```
47 batch_size = 10
48 classes_dim = 10    # 10 个类别
49 con_dim = 2      # 隐含信息变量的维度
50 rand_dim = 38
51 n_input  = 784
52
53 x = tf.placeholder(tf.float32, [None, n_input])
54 y = tf.placeholder(tf.int32, [None])
55
56 z_con = tf.random_normal((batch_size, con_dim)) #2列
57 z_rand = tf.random_normal((batch_size, rand_dim)) #38列
58 z = tf.concat(axis=1, values=[tf.one_hot(y, depth = classes_dim), z_con, z_rand]) # z的函数为50
59 gen = generator(z)
60 genout= tf.squeeze(gen, -1)
61
62 # 判别器的标准结果
63 y_real = tf.ones(batch_size) #真
64 y_fake = tf.zeros(batch_size) #假
65
66 # discriminator
67 disc_real, class_real, _ = discriminator(x) #真样本的输出
68 disc_fake, class_fake, con_fake = discriminator(gen) #模拟
69 pred_class = tf.argmax(class_fake, dimension=1)
```

对应判别器的结果，定义了一个值全为0的数组y_fake和一个值全为1的y_real，并且将x与生成的模拟数据gen放到判别器中，得到对应的输出。

5. 定义损失函数与优化器

判别器中，判别结果的loss有两个：真实输入的结果与模拟输入的结果。将二者结合在一起生成loss_d。生成器的loss为自己输出的模拟数据，让它在判别器中为真，定义为loss_g。

然后还要定义网络中共有的loss值：真实的标签与输入真实样本判别出的标签、真实的标签与输入模拟样本判别出的标签、隐含信息的重构误差。然后创建两个优化器，将它们放到对应的优化器中。

这里用了一个技巧：将判别器的学习率设小，将生成器的学习率设大一些。这么做是为了让生成器有更快的进化速度来模拟真实数据，优化同样用AdamOptimizer方法。具体代码如下。

代码12-1 Mnistinfogan（续）

```
70 # 判别器loss
71 loss_d_r = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits=logits=disc_real, labels=y_real))
72 loss_d_f = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits=disc_fake, labels=y_fake))
73 loss_d = (loss_d_r + loss_d_f) / 2 #判别器的loss
```

*****ebook converter DEMO Watermarks*****

```
74 # 生成器loss
75 loss_g = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_
(logits=disc_fake, labels=y_real)) #生成器的loss
76 # 计算 factor loss
77 loss_cf = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_
logits=logits=class_fake, labels=y)) #分类正确，但生成的样本错了
78 loss_cr = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_
logits=logits=class_real, labels=y)) #生成的样本与分类都正确，但
79 loss_c =(loss_cf + loss_cr) / 2
80 # 隐含信息变量的loss
81 loss_con =tf.reduce_mean(tf.square(con_fake-z_con))
82
83 # 获得可训练的学习参数列表
84 t_vars = tf.trainable_variables()
85 d_vars = [var for var in t_vars if 'discriminator' in var.name]
86 g_vars = [var for var in t_vars if 'generator' in var.name]
87
88 disc_global_step = tf.Variable(0, trainable=False)
89 gen_global_step = tf.Variable(0, trainable=False)
90
91 train_disc = tf.train.AdamOptimizer(0.0001).minimize(loss_
+ loss_con, var_list = d_vars, global_step = disc_global_step)
92 train_gen = tf.train.AdamOptimizer(0.001).minimize(loss_
loss_con, var_list = g_vars, global_step = gen_global_step)
```

所谓的AC-GAN就是将loss_cr加入到loss_c中。如果没有loss_cr，令loss_c= loss_cf，对于网络生成模拟数据是不影响的，但是却会损失真实分类与模拟数据间的对应关系。

6. 开始训练与测试

建立session，在循环里使用run来运行前面构建的两个优化器。

代码12-1 Mnistinfogan（续）

```
93 training_epochs = 3
```

*****ebook converter DEMO Watermarks*****

```
94 display_step = 1
95
96 with tf.Session() as sess:
97     sess.run(tf.global_variables_initializer())
98
99     for epoch in range(training_epochs):
100         avg_cost = 0.
101         total_batch = int(mnist.train.num_examples/batch_size)
102
103         # 遍历全部数据集
104         for i in range(total_batch):
105
106             batch_xs, batch_ys = mnist.train.next_batch(batch_size)
107             feeds = {x: batch_xs, y: batch_ys}
108
109             # 输入数据, 运行优化器
110             l_disc, _, l_d_step = sess.run([loss_d, train_op, global_step], feeds)
111             l_gen, _, l_g_step = sess.run([loss_g, train_op, global_step], feeds)
112
113         # 显示训练中的详细信息
114         if epoch % display_step == 0:
115             print("Epoch:", '%04d' % (epoch + 1), "cost=", l_disc, l_gen)
116
117         print("完成!")
118         # 测试
119         print ("Result:", loss_d.eval({x: mnist.test.images|y:mnist.test.labels[:batch_size]}),
120               , loss_g.eval({x: mnist.test.images[:batch_size],y:mnist.test.labels[:batch_size]})
```

测试部分分别使用loss_d和loss_g的eval来完成。运行代码后输出如下：

```
Extracting /data/train-images-idx3-ubyte.gz
Extracting /data/train-labels-idx1-ubyte.gz
Extracting /data/t10k-images-idx3-ubyte.gz
Extracting /data/t10k-labels-idx1-ubyte.gz
Epoch: 0001 cost= 0.536611855  0.795714
Epoch: 0002 cost= 0.610126615  0.928032
Epoch: 0003 cost= 0.699066639  1.10242
完成!
```

*****ebook converter DEMO Watermarks*****

Result: 0.56922 1.00881

整个数据集运行3次后，通过模型的测试结果可以看到，判别的误差在0.57左右，基本可以认为对真假数据无法分辨。

7. 可视化

可视化部分会生成两个图片：原样本与对应的模拟数据图片、利用隐含信息生成的模拟样本图片。

- 原样本与对应的模拟数据图片会将对应的分类、预测分类、隐含信息一起打印出来。
- 利用隐含信息生成的模拟样本图片会在整个[0, 1]空间里均匀抽样，与样本的标签混合在一起，生成模拟数据。

代码12-1 Mnistinfogan（续）

```
121 # 根据图片模拟生成图片
122     show_num = 10
123     gensimple,d_class,inputx,inputy,con_out = sess.run(
124         [genout,pred_class,x,y,con_fake], feed_dict={x:
125             images[:batch_size],y: mnist.test.labels[:batch_size]
126             }
126     f, a = plt.subplots(2, 10, figsize=(10, 2))
127     for i in range(show_num):
128         a[0][i].imshow(np.reshape(inputx[i], (28, 28)))
129         a[1][i].imshow(np.reshape(gensimple[i], (28, 28)))
130         print("d_class",d_class[i],"inputy",inputy[i],"con_out[i])
```

```

131     plt.draw()
132     plt.show()
133     #将隐含信息分布对应的图片打印出来
134     my_con=tf.placeholder(tf.float32, [batch_size,2])
135     myz = tf.concat(axis=1, values=[tf.one_hot(y, depth
136         my_con, z_rand)])
137     mygen = generator(myz)
138     mygenout= tf.squeeze(mygen, -1)
139
140     my_con1 = np.ones([10,2])
141     a = np.linspace(0.0001, 0.99999, 10)
142     y_input= np.ones([10])
143     figure = np.zeros((28 * 10, 28 * 10))
144     my_rand = tf.random_normal((10, rand_dim))
145     for i in range(10):
146         for j in range(10):
147             my_con1[j][0]=a[i]
148             my_con1[j][1]=a[j]
149             y_input[j] = j
150             mygenoutv = sess.run(mygenout, feed_dict={y:y_ir
151             con:my_con1})
152             for jj in range(10):
153                 digit = mygenoutv[jj].reshape(28, 28)
154                 figure[i * 28: (i + 1) * 28,
155                     jj * 28: (jj + 1) * 28] = digit
156
157     plt.figure(figsize=(10, 10))
158     plt.imshow(figure, cmap='Greys_r')
159     plt.show()

```

运行代码后，生成如下结果，输出图片如图12-3所示。

```

d_class 7 inputy 7 con_out [ 1.92287825e-05  1.04916848e-0
d_class 2 inputy 2 con_out [ 0.86672944  0.00166412]
d_class 1 inputy 1 con_out [ 0.00043415  0.06153901]
d_class 0 inputy 0 con_out [ 0.00313404  0.00186323]
d_class 4 inputy 4 con_out [ 0.01356777  0.9993856 ]
d_class 1 inputy 1 con_out [ 2.54907101e-01  1.13632974e-0
d_class 4 inputy 4 con_out [ 0.95273513  0.74673545]
d_class 9 inputy 9 con_out [ 4.87649202e-01  7.44661302e-0
d_class 5 inputy 5 con_out [ 0.00222825  0.99024838]
d_class 9 inputy 9 con_out [ 6.79908317e-06  3.18196639e-0

```

*****ebook converter DEMO Watermarks*****

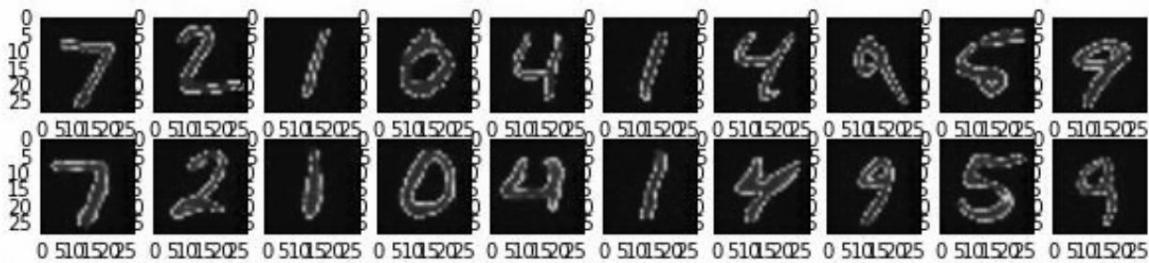


图12-3 InfoGAN实例结果1

在上面的结果中，可以很容易观察到，除了可控的类别信息一致外，隐含信息中某些维度具有非常显著的语义信息。例如，第二个元素“2”的第一个维度数值很大，表现出来就是倾斜很大，同样第5个元素“4”会看上去粗一些，这与其第二个维度的数值很大也是有关的。所以显然网络模型已经学到了MNIST数据集的重要信息（主成分）。将隐含信息对应的0、1间的数值抽样配合类别标签的图像生成结果如图12-4所示。

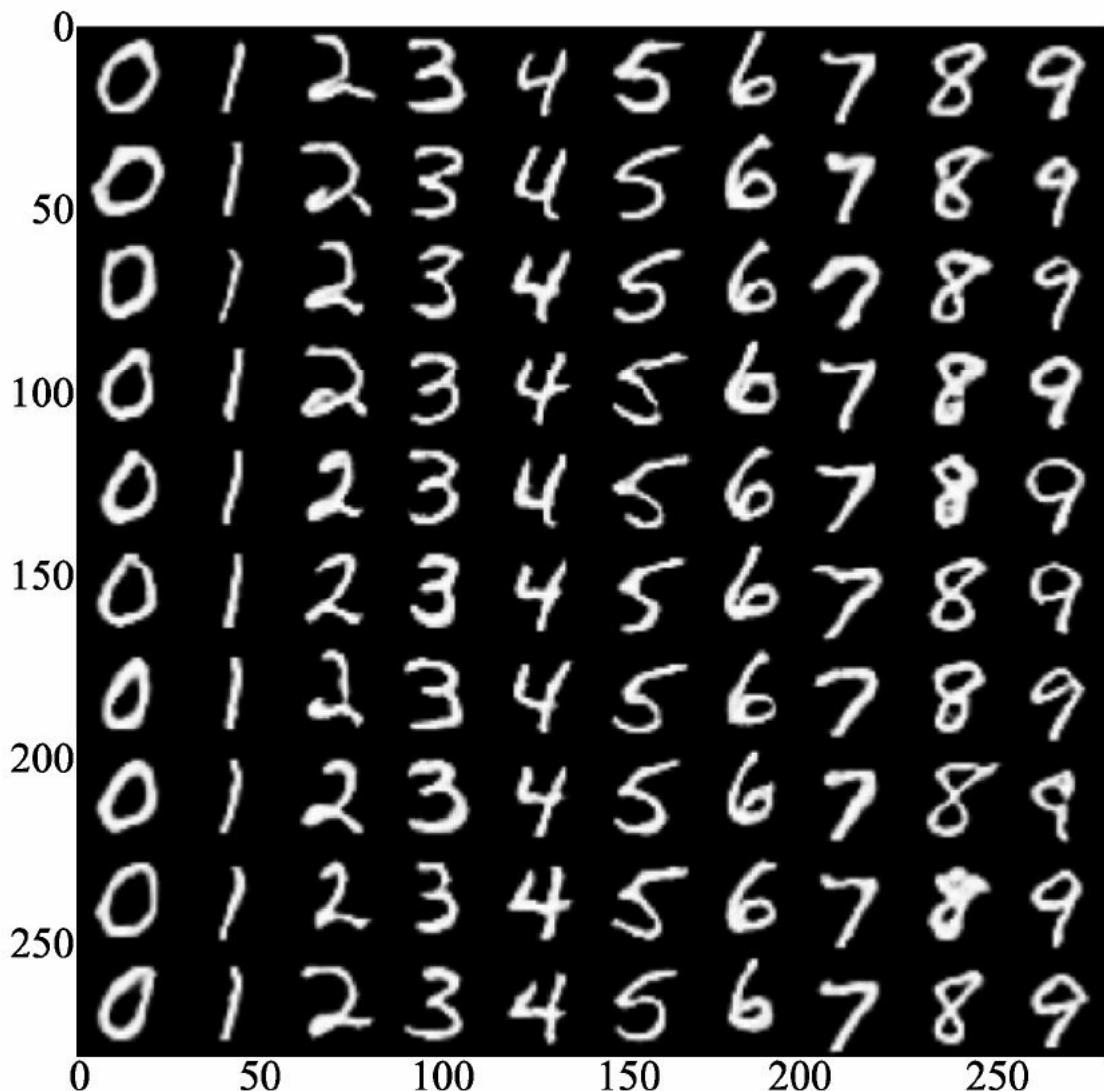


图12-4 InfoGAN实例结果2

12.3.4 练习题

在前面的例子中找到如下代码，并修改：

```
loss_cf = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_
(logits=class_fake, labels=y)) #分类正确，但生成的样本错了
loss_cr = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_
(logits=class_real, labels=y)) #生成的样本与分类正确，但是与输入的
loss_c =(loss_cf + loss_cr) / 2
```

令`loss_c`分别等于`loss_cr`和`loss_cf`。运行代码
观察结果，体验`loss_cr`和`loss_cf`两个`loss`值的作
用。

12.4 AEGAN：基于自编码器的GAN

在前面我们学习了自编码AE，而AEGAN就是GAN与AE的结合。AE的基本原理是特征的映射，即将高维特征压缩到低维特征，而在特征重建过程中只能模拟输入的单个体样本来输出结果（变分自解码除外），而AEGAN的优势在于在重建过程中可以生成与自己类似的样本，其功效等同于变分自解码器。

12.4.1 AEGAN原理及用途介绍

AEGAN的理论比变分自解码简单得多，单纯在GAN之后加个自解码网络即可。通过GAN可以利用噪声生成模拟数据的特点，使用自解码完成特征到图像的反向映射，从而实现一个即可将数据映射到低维空间，又可以将低维还原模拟分布数据的网络。具体结构如图12-5所示。

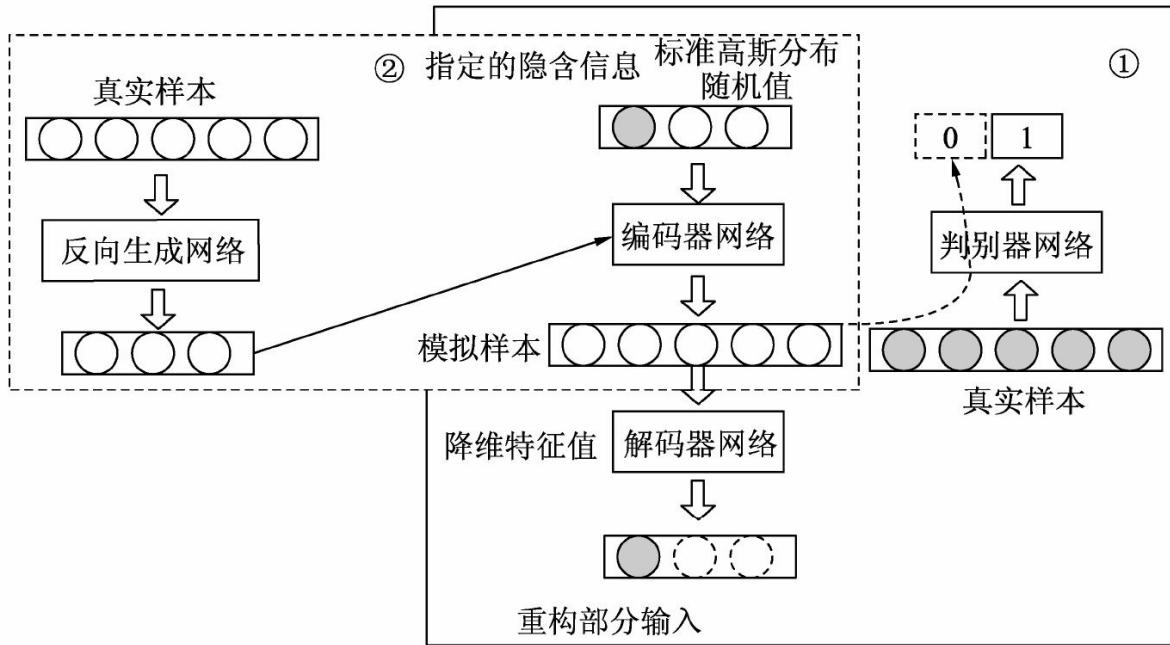


图12-5 AEGAN结构

图12-5所示为一个在InfoGAN上面嫁接一个自编码网络，原本自编码的编码器在这里叫做反向生成网络，它的解码器就是GAN的编码器网络。AEGAN网络训练分为两步。

(1) 用传统方式训练一个GAN，图12-5中训练的GAN为InfoGAN。

(2) 固定GAN网络，利用自编码网络来训练反向生成网络。这样得到的反向生成网络就具有高维到低维映射的能力了。

AEGAN的原理是先固定复杂样本分布作为网络输入，再慢慢调整网络输出去匹配标准高斯分布。对抗自编码器，严格来说应该不算GAN的变种，因为它的思路方向与GAN相反（GAN的思

路是将低维数据确定，向高维数据去匹配），所以对抗自编码器应该属于自编码器的变种会更为合适。回忆下前面学过的变分自编码器，其中编码器得出均值和方差是通过公式计算，然后用比较与标准高斯分布KL散度来拟合高斯分布的。在这里判别器就起到公式计算的等同作用，目的都是辅助解码器生成标准高斯分布的数据，即拉近编码器分布与标准高斯分布间的距离，只不过这部分的公式用神经网络来代替，通过多次迭代来完成，大大降低了模型依赖公式的复杂度。此外，对于扩展性也有极好的提升，只需要改变判别式中真实样本的输入分布，就可以得到不同分布的编码器，而不需要再为具体分布单独设计算法及公式了。

12.4.2 实例89：使用AEGAN对MNIST数据集压缩特征及重建

本例演示在MNIST数据集上使用AEGAN模型进行特征压缩及重建，并且加入标签信息的loss函数同时实现了AC-GAN网络。其中的D和G都是用卷积网络来实现的。具体实现可以分为以下几个步骤。

实例描述

通过使用前面的InfoGAN网络例子，在其基础上添加自编码网络，将InfoGAN的参数固定，

*****ebook converter DEMO Watermarks*****

训练反向生成器（自编码网络中的编码器），并将生成的模型用于MNIST数据集样本重建，得到相似样本。

本实例在代码“12-1 Mnistinfogan.py”的基础上添加自编码网络功能，具体步骤如下。

1. 添加反向生成器

在代码“12-1 Mnistinfogan.py”中添加反向生成器inversegenerator函数。该函数的功能是将图片生成特征码，其结构与判别器类似，均为生成器的反向操作，即使用两个卷积层，再接两个全连接层。代码如下。

代码12-2 aegan

```
01 #生成器函数
02 def generator(x):
03     .....
04
05 #反向生成器定义，结构与判别器类似
06 def inversegenerator(x):
07     reuse = len([t for t in tf.global_variables() if t.name == 'inversegenerator']) > 0
08     with tf.variable_scope('inversegenerator', reuse=reuse):
09         #使用了两个卷积层
10         x = tf.reshape(x, shape=[-1, 28, 28, 1])
11         x = slim.conv2d(x, num_outputs=64, kernel_size=[4, 4],
12                         stride=2, activation_fn=leaky_relu)
13         x = slim.conv2d(x, num_outputs=128, kernel_size=[4, 4],
14                         stride=2, activation_fn=leaky_relu)
15         #两个全连接
16         x = slim.flatten(x)
17         shared_tensor = slim.fully_connected(x, num_outputs=10,
18                                             activation_fn = leaky_relu)
```

```
16         z = slim.fully_connected(shared_tensor, num_output)
17         activation_fn = leaky_relu)
18     return z
```

2. 添加自编码网络代码

自编码网络的输入并不是真实图片，而是生成器生成的图片generator（z），通过inversegenerator来压缩特征，生成与生成器输入噪声一样的维度，然后再将生成器generator当成自编码中的解码器重建出原始生成的图片。

将自编码还原的图片与GAN中生成器生成的输入图片进行平方差的计算，得到自编码的损失值loss_ae。

代码12-2 aegan（续）

```
18 .....
19 gen = generator(z)
20 genout= tf.squeeze(gen, -1)
21
22 #自编码网络
23 aelearning_rate =0.01
24 igen = generator(inversegenerator(generator(z)))
25 loss_ae = tf.reduce_mean(tf.pow(gen - igen, 2))
26
27 #输出
28 igenout = generator(inversegenerator(x))
29
30 # 判别器结果标签
31 y_real = tf.ones(batch_size)                      #真
32 y_fake = tf.zeros(batch_size)                     #假
33 .....
```

3. 添加自编码网络的训练参数列表，定义优化器

自编码网络的训练参数与前面的GAN几乎一样，直接复制然后改个名字即可，本例中将使用MonitoredTrainingSession（对于MonitoredTrainingSession不熟悉的读者，可以看本书第4章的检查点保存部分）来管理检查点文件，所以定义了global_step。定义train_ae优化器，并将global_step放入优化器中。

代码12-2 aegan（续）

```
34 # 获得训练时需要更新的学习参数列表
35 t_vars = tf.trainable_variables()
36 d_vars = [var for var in t_vars if 'discriminator' in var.name]
37 g_vars = [var for var in t_vars if 'generator' in var.name]
38 ae_vars = [var for var in t_vars if 'inversegenerator' in var.name]
39
40 gen_global_step = tf.Variable(0, trainable=False)
41 global_step = tf.contrib.framework.get_or_create_global_step()
42
43 train_disc = tf.train.AdamOptimizer(0.0001).minimize(loss_disc +
+ loss_con, var_list = d_vars, global_step = global_step)
44 train_gen = tf.train.AdamOptimizer(0.001).minimize(loss_gen +
loss_con, var_list = g_vars, global_step = gen_global_step)
45 train_ae = tf.train.AdamOptimizer(aelearning_rate).minimize(
var_list = ae_vars, global_step = global_step)
46 training_GANepochs = 3      #训练GAN迭代3次数据集
47 training_aeepochs = 6      #训练AE迭代3次数据集(从3到6)
```

本例中需要一下训练GAN和AE两个网络，使用MonitoredTrainingSession管理后就只能有一个global_step，于是将global_step分段来管理两个

网络的训练。每一次迭代训练都会遍历整个MNIST数据集，先让第一个网络GAN迭代3次，然后再让第二个网络AE迭代3次。

4. 启动session依次训练GAN与AE网络

使用MonitoredTrainingSession创建session。令程序2分钟保存一次检查点文件，先训练GAN然后训练AE，最终将结果打印出来。

代码12-2 aegan（续）

```
48 with tf.train.MonitoredTrainingSession(checkpoint_dir='lcaecheckpoints', save_checkpoint_secs =120) as sess:
49
50     total_batch = int(mnist.train.num_examples/batch_size)
51     print("ae_global_step.eval(session=sess)",global_step
      (session=sess),int(global_step.eval(session=sess)/total_
52
53     for epoch in range( int(global_step.eval(session=sess)
      batch),training_GANepochs):
54         avg_cost = 0.
55
56         # 遍历全部数据集
57         for i in range(total_batch):
58
59             batch_xs, batch_ys = mnist.train.next_batch(t
60             feeds = {x: batch_xs, y: batch_ys}
61
62             # 输入数据，运行优化器
63             l_disc, _, l_d_step = sess.run([loss_d, tra
      step],feeds)
64             l_gen, _, l_g_step = sess.run([loss_g, tra
      global_step],feeds)
65
66             # 显示训练中的详细信息
67             if epoch % display_step == 0:
68                 print("Epoch:", '%04d' % (epoch + 1), "cost='
      .format(l_disc),l_gen)
69
```

*****ebook converter DEMO Watermarks*****

```

70     print("GAN完成!")
71     # 测试
72     print ("Result:", loss_d.eval({x: mnist.test.images[:batch_size],y:mnist.test.labels[:batch_size]},session = sess),loss_(mnist.test.images[:batch_size],y:mnist.test. labels[:batch_size]},session = sess))
73
74     # 根据图片模拟生成图片
75     show_num = 10
76     gensimple,inputx = sess.run(
77         [genout,x], feed_dict={x: mnist.test.images[:batch_size],y:mnist.test.labels[:batch_size]}) 
78
79     f, a = plt.subplots(2, 10, figsize=(10, 2))
80     for i in range(show_num):
81         a[0][i].imshow(np.reshape(inputx[i], (28, 28)))
82         a[1][i].imshow(np.reshape(gensimple[i], (28, 28)))
83
84     plt.draw()
85     plt.show()
86
87     #开始ae
88 print("ae_global_step.eval(session=sess)",global_step.eval(session=sess),int(global_step.eval(session=sess)/total_batch))
89     for epoch in range(int(global_step.eval(session=sess)/batch),training_aeepochs):
90         avg_cost = 0.
91
92         # 遍历全部数据集
93         for i in range(total_batch):
94
95             batch_xs, batch_ys = mnist.train.next_batch(batch_size)
96             feeds = {x: batch_xs, y: batch_ys}
97
98             # 输入数据, 运行优化器
99             l_ae, _, ae_step = sess.run([loss_ae, train_ae, global_step], feeds)
100
101         # 显示训练中的详细信息
102         if epoch % display_step == 0:
103             print("Epoch:", '%04d' % (epoch + 1), "cost=",
104             "%.10f" % l_ae)
105
106         # 测试
107         print ("Result:", loss_ae.eval({x: mnist.test.images[:batch_size],y:mnist.test.labels[:batch_size]},session = sess))
108
109         # 根据图片模拟生成图片

```

*****ebook converter DEMO Watermarks*****

```
110     gensimple, inputx = sess.run(  
111         [igenout, x], feed_dict={x: mnist.test.images[:batch_size],  
112          mnist.test.labels[:batch_size]})  
113     f, a = plt.subplots(2, 10, figsize=(10, 2))  
114     for i in range(show_num):  
115         a[0][i].imshow(np.reshape(inputx[i], (28, 28)))  
116         a[1][i].imshow(np.reshape(gensimple[i], (28, 28)))  
117  
118     plt.draw()  
119     plt.show()
```

由于global_step是整个的迭代次数，而自定义的training_aepochs是代表整个数据集迭代的次数，所以需要在循环之前将global_step转化一下，换算成迭代次数，见代码第90行中for里面的内容。运行代码，结果如下，输出图片如图12-6和图12-7所示。

```
Extracting /data/train-images-idx3-ubyte.gz  
Extracting /data/train-labels-idx1-ubyte.gz  
Extracting /data/t10k-images-idx3-ubyte.gz  
Extracting /data/t10k-labels-idx1-ubyte.gz  
INFO:tensorflow>Create CheckpointSaverHook.  
INFO:tensorflow:Saving checkpoints for 0 into log/aecheckpo:  
ae_global_step.eval(session=sess) 0 0  
.....  
INFO:tensorflow:global_step/sec: 66.8421  
Epoch: 0001 cost= 0.725493670 1.14361  
.....  
INFO:tensorflow:global_step/sec: 67.0216  
INFO:tensorflow:Saving checkpoints for 7429 into log/aecheck  
INFO:tensorflow:global_step/sec: 17.1345  
.....  
Epoch: 0002 cost= 0.590400815 0.877365  
INFO:tensorflow:global_step/sec: 63.1296  
INFO:tensorflow:global_step/sec: 67.9339  
INFO:tensorflow:Saving checkpoints for 15170 into log/aeche  
INFO:tensorflow:global_step/sec: 16.9023  
INFO:tensorflow:global_step/sec: 67.2034  
.....
```

*****ebook converter DEMO Watermarks*****

```
Epoch: 0003 cost= 0.527337492 1.45355
GAN完成!
Result: 0.523087 1.48371
.....
ae_global_step.eval(session=sess) 16500 3
INFO:tensorflow:global_step/sec: 43.4022
.....
Epoch: 0004 cost= 0.026241459
.....
INFO:tensorflow:global_step/sec: 138.887
Epoch: 0005 cost= 0.027687110
.....
INFO:tensorflow:global_step/sec: 142.044
INFO:tensorflow:Saving checkpoints for 29870 into log/aecho...
.....
INFO:tensorflow:global_step/sec: 141.241
Epoch: 0006 cost= 0.024392122
Result: 0.0210641
```



图12-6 GAN结果

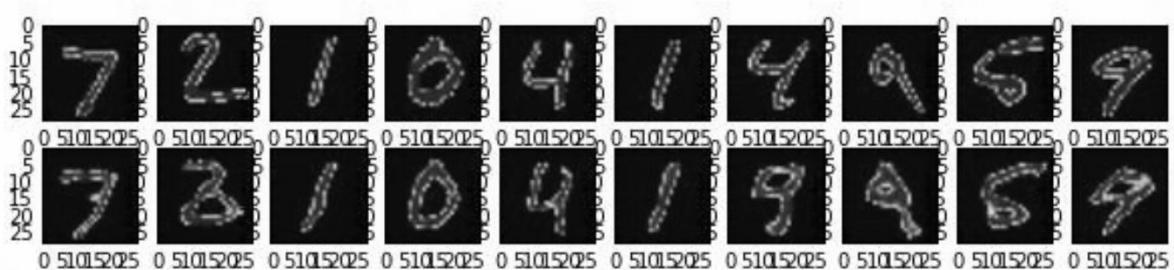


图12-7 AEGAN结果

从图12-6中可以看出，InfoGan只会生成属于原始数据分布的图片。从图12-7中可以看出，AEGAN会生成与原始图片更相近的图片。

这种网络有压缩特征与重建两部分用途，重建样本常用来处理图像的恢复与重建，还可以将重建的模拟数据保存起来以扩充数据集，甚至可以应用在超分辨率重建部分；对于压缩特征部分，可以应用在搜索相似图片的领域。

12.5 WGAN-GP：更容易训练的GAN

WGAN-GP又称为具有梯度惩罚（Gradient Penalty）的WGAN（Wasserstein GAN），是WGAN的升级版，一般可以全面代替WGAN。但是为了让读者了解WGAN-GP，还是先来介绍WGAN。

12.5.1 WGAN：基于推土机距离原理的GAN

1. 原始GAN的问题即原因

实际训练中，GAN存在着训练困难、生成器和判别器的loss无法指示训练进程、生成样本缺乏多样性等问题。这与GAN的机制有关。

GAN最终达到对抗的纳什均衡只是一个理想状态，而现实情况中得到的结果都是中间状态（伪平衡）。大部分的情况是，随着训练的次数越多判别器D的效果越好，会导致一直可以将生成器G的输出与真实样本区分开。

这是因为生成器G是从低维空间向高维空间（复杂的样本空间）映射，其生成的样本分布空间 P_g 难以充满整个真实样本的分布空间 P_r 。即两个分布完全没有重叠的部分，或者它们重叠的部分可以忽略，这样就使得判别器D总会将它们分

开。

为什么可以忽略呢？放在二维空间中会更好理解一些。在二维平面中随机取两条曲线，两条曲线上的点可以代表二者的分布，要想判别器无法分辨它们，需要两个分布融合在一起，即它们之间需要存在重叠线段，然而这样的概率为0；另一方面，即使它们很可能会存在交叉点，但是相比于两条曲线而言，交叉点比曲线低一个维度，长度（测度）为0代表它只是一个点，代表不了分布情况，所以可以忽略。

这样会带来什么后果呢？假设先将D训练得足够好，然后固定D，再来训练G，通过实验会发现G的loss无论怎么更新也无法收敛到最小值，而是无限接近log2。这个log2可以理解为Pg与Pr两个样本分布的距离。loss值恒定即表明G的梯度为0，无法再通过训练来优化自己。

所以在原始GAN的训练中，判别器训练得太好，会使生成器梯度消失，生成器loss降不下去；判别器训练得不好，会使生成器梯度不准，四处乱跑。只有判别器训练到中间状态最佳，但是这个尺度很难把握，甚至在同一轮训练的前后不同阶段，这个状态出现的时段都不一样，是个完全不可控的情况。

2. WGan介绍

WGAN (Wasserstein Gan) , Wasserstein是指 Wasserstein距离，又叫Earth-Mover (EM) 推土机距离。

WGAN的思想是将生成的模拟样本分布Pg与原始样本分布Pr组合起来，当成所有可能的联合分布的集合。然后可以从中采样得到真实样本与模拟样本，并能够计算二者的距离，还可以算出距离的期望值。这样就可以通过训练，让网络在所有可能的联合分布中对这个期望值取下界的方向优化，也就是将两个分布的集合拉到一起。这样原来的判别式就不再是判别真伪的功能了，而是计算两个分布集合距离的功能。所以将其称为评论器更加合适，同样，最后一层的sigmoid也需要去掉了。

为了实现计算Wasserstein距离的功能，我们将这部分交给神经网络去拟合。为了简化公式，现在就让神经网络拟合如下函数，见式 (12-1)：

$$|f(x_1) - f(x_2)| \leq k|x_1 - x_2| \quad \text{式 (12-1)}$$

$f(x)$ 可以理解成神经网络的计算，让判别器来实现将 $f(x_1)$ 与 $f(x_2)$ 的距离变换为 $x_1 - x_2$ 的绝对值 $\times k$ ($K \geq 0$)。K代表函数 $f(x)$ 的 Lipschitz 常数，这样两个分布集合的距离就可以表示成 $D(\text{real}) - D(G(x))$ 的绝对值 $\times k$ 了，

这个k可以理解成梯度，即在神经网络 $f(x)$ 中 x 的梯度绝对值会小于K。

将k忽略整理后可以得到二者分布的式子，见式（12-2）：

$$L = D(\text{real}) - D(G(x)) \quad \text{式 (12-2)}$$

现在要做的就是将L当成目标来计算loss，G将希望生成的结果 P_g 越来越接近 P_r ，所以需要通过训练让距离L最小化。因为生成器G与第一项无关，所以G的loss可以简化为式（12-3）。

$$G(\text{loss}) = -D(G(x)) \quad \text{式 (12-3)}$$

而D的任务是区分它们，所以希望二者距离变大，所以loss需要取反，得到式（12-4）。

$$D(\text{loss}) = D(G(x)) - D(\text{real}) \quad \text{式 (12-4)}$$

同样，通过D的loss值也可以看出G的生成质量，即loss越小代表距离越近，则生成的质量越高。

而对于前面的梯度限制，WGAN直接使用了截断（clipping）的方式。这个方式在实际应用中有问题，所以后来又产生了其升级版WGAN-GP。

12.5.2 WGAN-GP：带梯度惩罚项的WGAN

1. WGAN问题即原因

前面介绍了原始WGAN的Lipschitz限制的施加方式不对，使用Weight clipping方式太过生硬。每当更新完一次判别器的参数之后，就检查判别器的所有参数的绝对值有没有超过一个阈值，比如0.01，如果说有的话就把这些参数截断（clip）回[-0.01, 0.01]的范围内。

Lipschitz限制本意是当输入的样本稍微变化后，判别器给出的分数不能发生太剧烈的变化。通过在训练过程中保证判别器的所有参数有界，就保证了判别器不能对两个略微不同的样本给出天差地别的分数值，从而间接实现了Lipschitz限制。

然而，这种渴望与判别器本身的目的相矛盾。在判别器中，是希望loss尽可能地大，才能拉大真假样本的区别，这种情况会导致在判别器中通过loss算出的梯度会沿着loss越来越大的方向变化，然而经过Weight clipping后每一个网络参数又被独立地限制了取值范围（如[-0.01, 0.01]），这种结果只能是所有的参数走向极端，要么取最大值（如0.01）要么取最小值（如-0.01），判别器没能充分利用自身的模型能力，经过它回传给生成器的梯度也会跟着变差。

如果判别器是一个多层网络，Weight clipping还会导致梯度消失或者梯度爆炸。原因是，如果我们把Clipping threshold设得稍微小了一点，每经过一层网络，梯度就变小一点，多层之后就会指数衰减；反之，如果设得稍微大了一点，每经过一层网络，梯度就会变大一点，多层之后就会指数爆炸。然而在实际应用中很难做到设置适宜，让生成器获得恰到好处的回传梯度。

2. WGAN-GP介绍

WGAN-GP中的GP是梯度惩罚（Gradient penalty）的意思。它是替换Weight clipping的一种方法。通过直接设置一个额外的梯度惩罚项，来实现判别器的梯度不超过K。

例如式（12-5）和式（12-6）中：

$$\text{Norm} = \text{tf.gradients}(D(X_{\text{inter}}), [X_{\text{inter}}]) \quad \text{式 (12-5)}$$

$$\text{grad_pen} = \text{MSE}(\text{Norm} - k) \quad \text{式 (12-6)}$$

MSE为平方差公式， X_{inter} 为整个联合分布空间的x取样，即梯度惩罚项grad_pen为求整个联合分布空间的x对应D的梯度与k的平方差。

判别器尽可能拉大真假样本的分数差距，希望梯度越大越好，变化幅度越大越好，所以判别器在充分训练之后，其梯度Norm其实就会在k附近。因此可以把上面的loss改成要求梯度Norm离k

越近越好， k 可以是任何数，我们就简单地把 k 定为1，再跟WGAN原来的判别器loss加权合并，就得到新的判别器loss，见式（12-7）：

$$L = D(\text{real}) - D(G(x)) + \lambda \text{MSE}(\text{tf.gradients}(D(X_{\text{inter}}), [X_{\text{inter}}]) - 1) \quad \text{式 (12-7)}$$

即式（12-8）：

$$L = D(\text{real}) - D(G(x)) + \lambda \times \text{grad_pen} \quad \text{式 (12-8)}$$

λ 为梯度惩罚参数，可以用来调节梯度惩罚的力度。

grad_pen 是需要从 P_g 与 P_r 的联合空间里采样。对于整个样本空间而言，需要抓住生成样本集中区域、真实样本集中区域及夹在它们中间的区域，即先随机取一个 $0 \sim 1$ 的随机数，令一对真假样本分别按随机数的比例加和来生成 X_{inter} 的采样，见式（12-9）和式（12-10）：

$$\text{eps} = \text{tf.random_uniform}([\text{shape}], \text{minval}=0., \text{maxval}=1.) \quad \text{式 (12-9)}$$

$$X_{\text{inter}} = \text{eps} \times \text{real} + (1. - \text{eps}) \times G(x) \quad \text{式 (12-10)}$$

这样把 X_{inter} 代入到式（12-5）中，就得到最终版本的判别器loss。

```
eps = tf.random_uniform([shape], minval=0., maxval=1.)
X_inter = eps*real + (1. - eps)*G(x)
L = D(real) - D(G(x)) + lambda * MSE(tf.gradients(D(X_inter), [X_inter]), [X_inter])
```

在WGAN-GP相关论文的实验中，Gradient penalty能够显著提高训练速度，解决了原始WGAN生成器梯度二值化问题（如图12-8a）与梯度消失爆炸问题（如图12-8b）。



注意：由于我们是对每个样本独立地施加梯度惩罚，所以判别器的模型架构中不能使用Batch Normalization，因为它会引入同一个batch中不同样本的相互依赖关系。如果需要，可以选择其他的normalization方法，如Layer Normalization、Weight Normalization和Instance Normalization，这些方法就不会引入样本之间的依赖。WGAN-GP的作者推荐的是Layer Normalization。

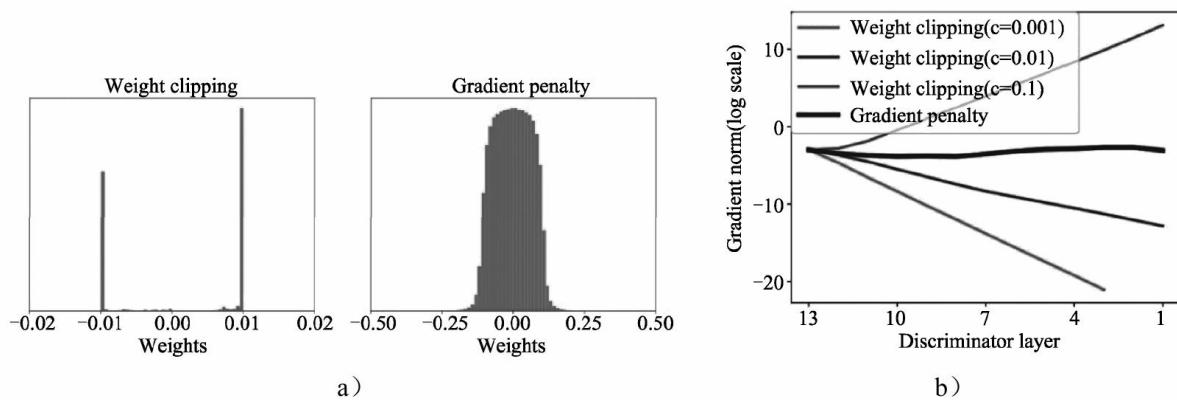


图12-8 WGAN-GP优势（该图片来源于WGAN-GP相关论文）

12.5.3 实例90：构建WGAN-GP生成MNIST数据集

在掌握了理论之后，下面通过一个例子对WGAN-GP有个更深刻的理解。

本例演示在MNIST数据集上使用WGAN-GP网络模型生成模拟数据。这次的D和G用最简单的全连接网络来实现。

实例描述

通过使用WGAN-GP网络学习MNIST数据特征，并生成以假乱真的MNIST模拟样本。

具体实现可以分为如下几个步骤。

1. 引入头文件并加载MNIST数据

假设MNIST数据放在本地磁盘跟目录的data下，同样使用slim库建立网络模型。代码如下：

代码12-3 wgan-gp

```
01 import tensorflow as tf
02 from tensorflow.examples.tutorials.mnist import input_data
03 import os
04 import numpy as np
05 from scipy import misc,ndimage
06 import tensorflow.contrib.slim as slim
07 #from tensorflow.python.ops import init_ops
08
09 mnist = input_data.read_data_sets("/data/", one_hot=True)
```

2. 定义生成器与判别器

*****ebook converter DEMO Watermarks*****

由于复杂部分都放在loss方面了，所以生成器G和判别器D就会简单一些，各自有3个全连接层。生成器最终输出与MNIST图片相同维度的数据作为模拟样本。判别器的输出不需要再有激活函数，输出维度为1的数值用来表示其结果。

代码12-3 wgan-gp（续）

```
10 def G(x):#生成器
11     reuse = len([t for t in tf.global_variables() if t.name.startswith('generator')]) > 0
12     with tf.variable_scope('generator', reuse = reuse):
13         x = slim.fully_connected(x, 32, activation_fn = tf.nn.relu)
14         x = slim.fully_connected(x, 128, activation_fn = tf.nn.relu)
15         x = slim.fully_connected(x, mnist_dim, activation_fn = tf.nn.sigmoid)
16     return x
17
18 def D(X):#判别器
19     reuse = len([t for t in tf.global_variables() if t.name.startswith('discriminator')]) > 0
20     with tf.variable_scope('discriminator', reuse=reuse):
21         X = slim.fully_connected(X, 128, activation_fn = tf.nn.relu)
22         X = slim.fully_connected(X, 32, activation_fn = tf.nn.relu)
23         X = slim.fully_connected(X, 1, activation_fn = None)
24     return X
```

3. 定义网络模型与loss

生成的模拟数据为random_Y，与前面所描述的一致；生成器的Loss为-D（random_Y）；而判别器的loss为D（random_Y） - D（real_X）再加上一个联合分布样本梯度的惩罚项grad_pen；惩罚项的采样X_inter由一部分Pg分布和一部分Pr分布组成。同时对D（X_inter）求梯度得到

*****ebook converter DEMO Watermarks*****

grad_pen。具体代码如下：

代码12-3 wgan-gp（续）

```
25 real_X = tf.placeholder(tf.float32, shape=[batch_size, mr
26 random_X = tf.placeholder(tf.float32, shape=[batch_size,
27 random_Y = G(random_X)
28
29 eps = tf.random_uniform([batch_size, 1], minval=0., maxva
30 X_inter = eps*real_X + (1. - eps)*random_Y #按照eps比例生成
31 grad = tf.gradients(D(X_inter), [X_inter])[0]
32 grad_norm = tf.sqrt(tf.reduce_sum((grad)**2, axis=1))
33 grad_pen = 10 * tf.reduce_mean(tf.nn.relu(grad_norm - 1.)
34
35 D_loss = tf.reduce_mean(D(random_Y)) -tf.reduce_mean(D(re
grad_pen
36 G_loss = -tf.reduce_mean(D(random_Y))
```

4. 定义优化器并开始训练

通过前面定义的命名空间，找到生成器和判别器的训练参数，通过AdamOptimizer进行优化训练。

代码12-3 wgan-gp（续）

```
37 # 获得各个网络中各自的训练参数
38 t_vars = tf.trainable_variables()
39 d_vars = [var for var in t_vars if 'discriminator' in var.name]
40 g_vars = [var for var in t_vars if 'generator' in var.name]
41 print(len(t_vars),len(d_vars))
42 #定义D和G的优化器
43 D_solver = tf.train.AdamOptimizer(1e-4, 0.5).minimize(D_
list=d_vars)
44 G_solver = tf.train.AdamOptimizer(1e-4, 0.5).minimize(G_
list=g_vars)
45
```

```
46 training_epochs =100
47
48 with tf.Session() as sess:
49     sess.run(tf.global_variables_initializer())
50     if not os.path.exists('out/'):
51         os.makedirs('out/')
52
53     for epoch in range(training_epochs):
54         total_batch = int(mnist.train.num_examples/batch_
55
56         # 遍历全部数据集
57         for e in range(total_batch):
58             for i in range(5):
59                 real_batch_X, _ = mnist.train.next_batch(1)
60                 random_batch_X = np.random.uniform(-1, 1,
61                 random_dim)
62                 _, D_loss_ = sess.run([D_solver,D_loss], 1
63                 {real_X:real_batch_X, random_X:random_batch_
64                 random_batch_X = np.random.uniform(-1, 1, (ba
65                 random_dim))
66                 _, G_loss_ = sess.run([G_solver,G_loss], feed_
67                 X:random_batch_X})
```

在session中优先让判别器学习次数多一些，让判别器每训练5次，生成器优化一次。
WGAN_GP不会因为判别器准确度太高而引起生成器梯度消失的问题，好的判别器只会让生成器有更好的模拟效果。

5. 可视化结果

这次我们把生成的结果用图片的方式保存起来，并生成到硬盘上。每10次的全样本迭代会生成一次图片，图片的位置为本地代码文件所在目录下的out文件夹内。代码如下：

代码12-3 wgan-gp (续)

```
64 if epoch % 10 == 0:  
65     print ('epoch %s, D_loss: %s, G_loss: %s'%(ep  
G_loss_))  
66     n_rows = 6  
67     check_imgs = sess.run(random_Y, feed_dict={r  
batch_X}).reshape((batch_size, width, height))|  
rows]  
68     imgs = np.ones((width*n_rows+5*n_rows+5, hei  
rows+5))  
69     for i in range(n_rows*n_rows):  
70         num1 = (i%n_rows)  
71         num2 = np.int32(i/n_rows)  
72         imgs[5+5*num1+width*num1:5+5*num1+width+w  
num2+height*num2:5+5*num2+height+height*num2  
imgs[i]  
73  
74     misc.imsave('out/%s.png'%(epoch/10), imgs)  
75  
76     print("完成!")
```

运行代码后，生成如下结果：

```
epoch 0, D_loss: -4.15614, G_loss: 0.35294  
epoch 10, D_loss: -2.5528, G_loss: 1.48789  
epoch 20, D_loss: -2.21916, G_loss: 1.0337  
epoch 30, D_loss: -1.87463, G_loss: 0.875138  
epoch 40, D_loss: -1.65764, G_loss: 0.752094  
epoch 50, D_loss: -1.40312, G_loss: 0.967182  
epoch 60, D_loss: -1.16828, G_loss: 0.772282  
epoch 70, D_loss: -1.20912, G_loss: 1.03305  
epoch 80, D_loss: -1.02528, G_loss: 1.05023  
epoch 90, D_loss: -0.922399, G_loss: 1.31767  
完成!
```

可以看到D_loss的值在逐渐变小，表明生成的模拟样本质量越来越高。来到本地的out文件夹下，找到10张图片，如图12-9所示（这里举例出3*****ebook converter DEMO Watermarks*****

张)。

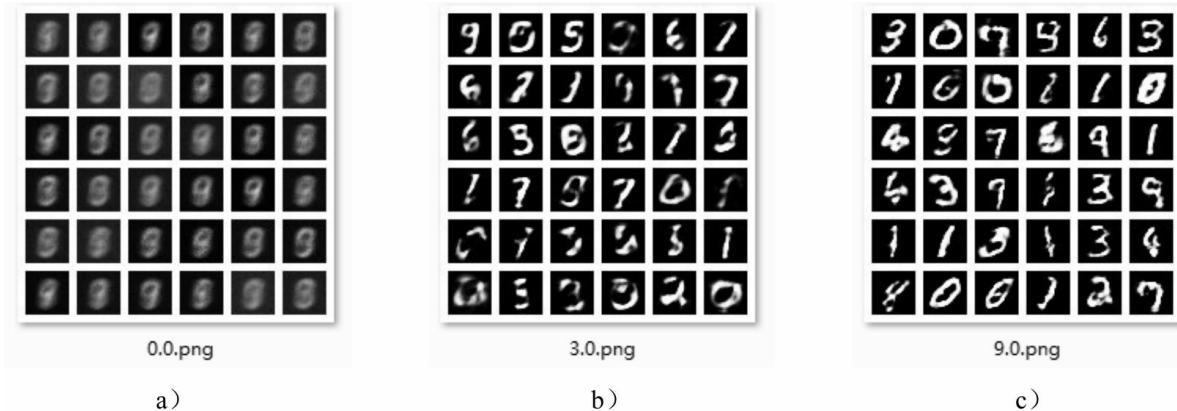


图12-9 WGAN-GP结果

图12-9a为第一次迭代时的输出，图12-9c为第10次的输出。可以看到，在WGAN-PG的判别器严格要求下，生成器的模拟数据越来越逼真。

12.5.4 练习题

把前面的例子代码loss部分分别改成如下两种情况。

(1) 第一种情况：

```
D_loss = tf.reduce_mean(D(random_Y)) - tf.reduce_mean(D(real_Y))
G_loss = tf.reduce_mean(D(random_Y))
```

(2) 第二种情况：

```
D_loss = tf.reduce_mean(D(real_X)) - tf.reduce_mean(D(random_X))
```

*****ebook converter DEMO Watermarks*****

```
G_loss = tf.reduce_mean(D(random_Y))
```

猜想一下会产生什么样的效果？为什么会这样？通过运行实际代码验证你的假设。

12.6 LSGAN（最小乘二GAN）：具有WGAN同样效果的GAN

前文已经介绍过GAN是以对抗的方式逼近概率分布。但是直接使用该方法，会随着判别器越来越好而生成器无法与其对抗，进而形成梯度消失的问题。所以不论是WGAN，还是本节中的LSGAN，都是试图使用不同的距离度量，从而构建一个不仅稳定，同时还收敛迅速的生成对抗网络。

下面就来一起学习一下LSGAN。

12.6.1 LSGAN介绍

WGAN使用的是Wasserstein理论来构建度量距离。而LSGAN使用了另一个方法，即使用了更加平滑和非饱和梯度的损失函数——最小乘二来代替原来的Sigmoid交叉熵。这是由于L2正则独有的特性，在数据偏离目标时会有一个与其偏离距离成比例的惩罚，再将其拉回来，从而使数据的偏离不会越来越远。

相对于WGAN而言，LSGAN的loss简单很多。直接将传统的GAN中的softmax变为平方差即可。

判别器的loss:

```
D_loss=tf.reduce_sum(tf.square(D(real_X)-1) + tf.square(D(ran
```

生成器的loss:

```
G_loss = tf.reduce_sum(tf.square(D(random_Y)-1))/2
```

为什么要除以2？和以前的原理一样，在对平方求导时会得到一个系数2，与事先的 $1/2$ 运算正好等于1，使公式更加完整。

12.6.2 实例91：构建LSGAN生成MNIST模拟数据

本例中直接修改“12-1 Mnistinfogan.py”代码中的loss函数，将其改成LSGAN网络。

实例描述

通过使用LSGAN网络学习MNIST数据特征，并生成以假乱真的MNIST模拟样本。

下面给出具体步骤。

1. 修改判别器

将判别器的最后一层输出disc改成使用Sigmoid的激活函数。代码如下：

代码12-4 mnistLsgan

```
01 def discriminator(x, num_classes=10, num_cont=2):
02
03     .....
04     disc = slim.fully_connected(shared_tensor, num_out_
activation_fn=tf.nn.sigmoid)
05     disc = tf.squeeze(disc, -1)
06     recog_cat = slim.fully_connected(recog_shared, n_
outputs=num_classes, activation_fn=None)
07     recog_cont = slim.fully_connected(recog_shared, r_
outputs=num_cont, activation_fn=tf.nn.sigmoid)
08     return disc, recog_cat, recog_cont
```

2. 修改loss值

将原有的loss_d与loss_g改成平方差形式，原有的y_real与y_fake不再需要了，可以删掉，其他代码不用变动。

代码12-4 MnistLsgan（续）

```
09 .....
10 # 判别器discriminator
11 disc_real, class_real, _ = discriminator(x)
12 disc_fake, class_fake, con_fake = discriminator(gen)
13 pred_class = tf.argmax(class_fake, dimension=1)
14
15 # 判别器loss
16 #loss_d_r = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_w_
(logits=disc_real, labels=y_real))
17 #loss_d_f = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_w_
(logits=disc_fake, labels=y_fake))
18 #最小乘二loss
```

```
19 loss_d = tf.reduce_sum(tf.square(disc_real-1) + tf.square(fake))/2  
20 loss_g = tf.reduce_sum(tf.square(disc_fake-1))/2  
21 .....
```

3. 运行代码生成结果

运行代码，生成结果如下，输出图片如图12-10所示。

```
Epoch: 0001 cost= 2.074717045 0.93645  
Epoch: 0002 cost= 2.024495363 1.88027  
Epoch: 0003 cost= 2.158437967 2.78284  
完成!  
Result: 1.71483 3.07138  
d_class 7 inputy 7 con_out [ 0.16134234 0.03605343]  
d_class 2 inputy 2 con_out [ 0.30764639 0.98185432]  
d_class 1 inputy 1 con_out [ 0.11353409 0.02166406]  
d_class 0 inputy 0 con_out [ 2.32195278e-04 2.08523397e-0  
d_class 4 inputy 4 con_out [ 0.355297 0.94447494]  
d_class 1 inputy 1 con_out [ 1.33050963e-01 1.69226732e-0  
d_class 4 inputy 4 con_out [ 0.17757109 0.78396767]  
d_class 9 inputy 9 con_out [ 6.99081238e-06 2.24134132e-0  
d_class 5 inputy 5 con_out [ 0.87434149 0.98944479]  
d_class 9 inputy 9 con_out [ 0.00770722 0.00958756]
```

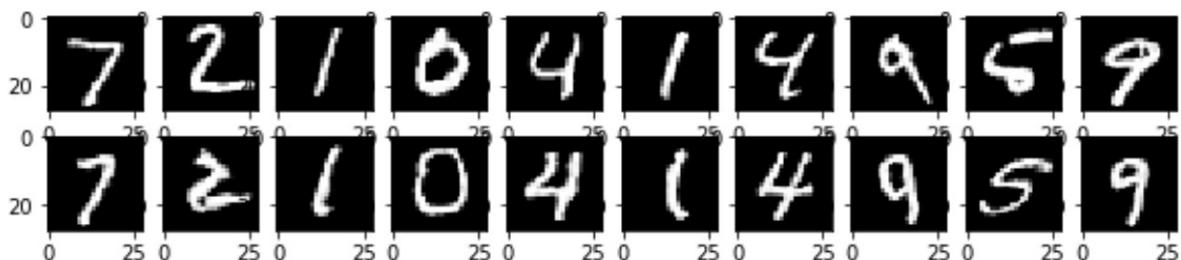


图12-10 LSGAN例子结果

可见LSGAN也可以产生与WGAN一样的效果。



注意： WGAN与LSGAN谁更好呢？答案是很难一概而论，只能具体问题具体分析。在实际实现中还会有很多细节决定最终的结果，不同的技术使用都会对结果造成相应的影响。

12.7 GAN-cls：具有匹配感知的判别器

本节介绍一种GAN网络增强技术——具有匹配感知的判别器。前面讲过，在InfoGAN中，使用了ACGAN的方式进行指导模拟数据与生成数据的对应关系。在GAN-cls中该效果会以更简单的方式来实现，即增强判别器的功能，令其不仅能判断图片真伪，还能判断匹配真伪。

12.7.1 GAN-cls的具体实现

GAN-cls的具体做法是，在原有的GAN网络上，将判别器的输入变为图片与对应标签的连接数据。这样判别器的输入特征中就会有生成图像的特征与对应标签的特征。然后用这样的判别器分别对真实标签与真实图片、假标签与真实图片、真实标签与假图片进行判断，预期的结果依次为真、假、假，在训练的过程中沿着这个方向收敛即可。而对于生成器，则不需要做任何改动。这样简单的一步就完成了生成根据标签匹配的模拟数据功能。

12.7.2 实例92：使用GAN-cls技术实现生成标签匹配的模拟数据

本例中直接修改“12-4 mnistLSgan.py”代码中

*****ebook converter DEMO Watermarks*****

的判别器函数。演示GAN-cls技术的使用。

实例描述

在代码“12-4 mnistLSgan.py”的基础上，使用GAN-cls技术对判别器进行改造，并通过输入错误的样本标签让判别器学习样本与标签的匹配，从而优化生成器，使生成器最终生成与标签一致的样本，实现与ACGAN等同的效果。

下面给出具体步骤。

1. 修改判别器

在代码“12-4 mnistLSgan.py”的基础上，将判别器的输入改成x与y，新增加的y代表输入的样本标签；在内部处理中，先通过全连接网络将y变为与图片一样维度的映射，并调整为图片相同的形状，使用concat将二者连接到一起统一处理。后续的处理过程是一样的，两个卷积后再接两个全连接，最后一层输出disc。代码如下：

代码12-5 GAN-cls

```
01 .....
02 def discriminator(x,y):
03     reuse = len([t for t in tf.global_variables() if t.name
04 ('discriminator')]) > 0
05     with tf.variable_scope('discriminator', reuse=reuse):
06         y = slim.fully_connected(y, num_outputs=n_input,
07         fn = leaky_relu)
```

```
06         y = tf.reshape(y, shape=[-1, 28, 28, 1]) #将y统一)
07         x = tf.reshape(x, shape=[-1, 28, 28, 1])
08     #将二者连接到一起，统一处理
09     x= tf.concat(axis=3, values=[x,y])
10         x = slim.conv2d(x, num_outputs = 64, kernel_size=
11                         stride=2, activation_fn=leaky_relu)
12         x = slim.conv2d(x, num_outputs=128, kernel_size=
13                         stride=2, activation_fn=leaky_relu)
14         x = slim.flatten(x)
15         shared_tensor = slim.fully_connected(x, num_outpu
16                                         activation_fn = leaky_relu)
17             disc = slim.fully_connected(shared_tensor, num_o
18                                         activation_fn=tf.nn.sigmoid)
19             disc = tf.squeeze(disc, -1)
20
21     return disc
```

2. 添加错误标签输入符，构建网络结构

添加错误标签misy，同时在判别器中分别将真实样本与真实标签、生成的图像gen与真实标签、真实样本与错误标签组成的输入传入判别器中。



注意： 这里是将3种输入的x与y分别按照batch_size维度连接变为判别器的一个输入的。生成结果后再使用split函数将其裁成3个结果disc_real、disc_fake和disc_mis，分别代表真实样本与真实标签、生成的图像gen与真实标签、真实样本与错误标签所对应的判别值。这么写会使代码看上去简洁一些，当然也可以一个一个地输入x、y，然后调用三次判别器，效果是一样的。

由于本例中不需要InfoGAN模型，将“12-4
*****ebook converter DEMO Watermarks*****”

mnistLSgan.py”代码中的隐含信息z_con部分全部去掉。代码如下：

代码12-5 GAN-cls (续)

```
18 x = tf.placeholder(tf.float32, [None, n_input])      #输入
19 y = tf.placeholder(tf.int32, [None])                #]
20 misy = tf.placeholder(tf.int32, [None])              #]
21
22 z_rand = tf.random_normal((batch_size, rand_dim)) #38列
23 z = tf.concat(axis=1, values=[tf.one_hot(y, depth = classes_dim), z_rand])#50列
24 gen = generator(z)
25 genout= tf.squeeze(gen, -1)
26
27 # 判别器discriminator
28 xin=tf.concat([x, tf.reshape(gen, shape=[-1,784]),x],0)
29 yin=tf.concat([tf.one_hot(y, depth = classes_dim),tf.one_
30 disc_all = discriminator(xin,yin)
31 disc_real,disc_fake,disc_mis =tf.split(disc_all,3)
32
33 #构建loss
34 loss_d = tf.reduce_sum(tf.square(disc_real-1) + ( tf.square(disc_fake)+tf.square(disc_mis))/2 )/2
35 loss_g = tf.reduce_sum(tf.square(disc_fake-1))/2
36 .....
```

在计算判别器的loss时，同样使用LSGAN方式，并且将错误部分的loss变为disc_fake与disc_mis的和，然后再除以2。因为对于生成器生成的样本与错误的输入标签，判别器都应该将其判断为错误。

3. 使用MonitoredTrainingSession创建session，开始训练

定义global_step，使用MonitoredTrainingSession创建session，来管理检查点文件，在session中构建错误标签数据，训练模型。

代码12-5 GAN-cls（续）

```
37 .....
38
39 global_step = tf.train.get_or_create_global_step() #使用M
40
41 train_disc = tf.train.AdamOptimizer(0.0001).minimize(loss_
list = d_vars, global_step = global_step)
42 train_gen = tf.train.AdamOptimizer(0.001).minimize(loss_(
list = g_vars, global_step = gen_global_step)
43
44 training_epochs = 3                      #整体数据集迭代3次
45 display_step = 1                         #每迭代一次显示一次输出信息
46
47 with tf.train.MonitoredTrainingSession(checkpoint_dir='l
checkpointsnew', save_checkpoint_secs = 60) as sess:
48
49     total_batch = int(mnist.train.num_examples/batch_size)
50     print("global_step.eval(session=sess)",global_step.e
(session=sess),int(global_step.eval(session=sess)/total_
51     for epoch in range( int(global_step.eval(session=sess
batch),training_epochs):
52         avg_cost = 0.
53
54     # 遍历全部数据集
55     for i in range(total_batch):
56         batch_xs, batch_ys = mnist.train.next_batch(t
57         _, mis_batch_ys = mnist.train.next_batch(batc
58         feeds = {x: batch_xs, y: batch_ys,misy:mis_ba
59
60         # 输入数据，运行优化器
61         l_disc, _, l_d_step = sess.run([loss_d, tra
step],feeds)
62         l_gen, _, l_g_step = sess.run([loss_g, tra
step],feeds)
63 .....
```

运行代码，生成如下结果，输出图片如图12-11所示。

完成！

result: 1.17139 0.829812

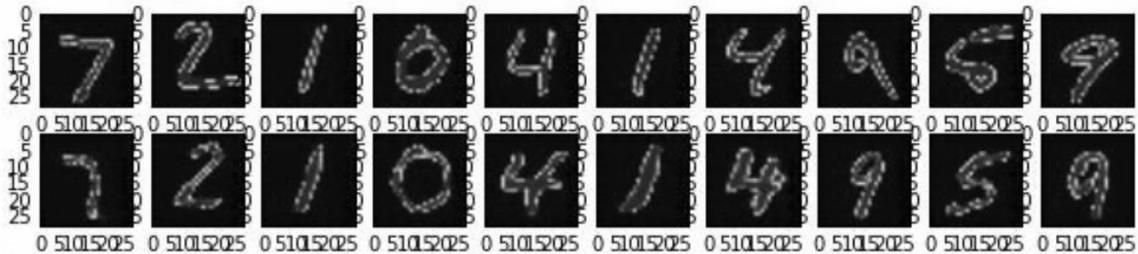


图12-11 GAN-cls结果

如图12-11所示，使用GAN-cls技术同样也实现了生成与标签对应的样本，而且整体代码的运算要比ACGAN简洁很多。

12.8 SRGAN——适用于超分辨率重建的GAN

SRGAN属于GAN理论在超分辨率重建(SR)方面的应用。在学习SRGAN之前有必要先了解一下SR领域的相关技术。

12.8.1 超分辨率技术

1. SR（超分辨率重建）技术介绍

SR（Super-Resolution，超分辨率）技术，是指从观测到的低分辨率图像重建出相应的高分辨率图像，在监控设备、卫星图像和医学影像等领域都有重要的应用价值，该技术也可应用于马赛克图片的恢复应用场景中。

SR可分为两类：从多张低分辨率图像重建出高分辨率图像，和从单张低分辨率图像重建出高分辨率图像。基于深度学习的SR，主要是基于单张低分辨率的重建方法，即Single Image Super-Resolution（SISR）。

SISR是一个逆问题。对于一个低分辨率图像，可能存在许多不同的高分辨率图像与之对应，为了让逆向图片的结果更接近真实图片，则

需要让模型在一定约束下，指定某个领域中来进行可逆训练，而这个约束，就是指现有的低分辨率像素的色度信息与位置信息。为了能让模型更好地学习并利用这个信息，基于深度学习的SR通过神经网络直接通过优化低分辨率图像到高分辨率图像的损失函数loss来进行端到端训练，以实现超分辨率重建功能。

2. 深度学习中的SR方法

在GAN出现之前，先是以SRCNN、DRCN为主的SR方法。该方法的大体思想是将低分辨率像素先扩展到高分辨率的像素大小，然后通过卷积方式训练网络，优化其与真实高分辨率图片的loss，最终形成模型。并且在这一方法上也总结了不少经验参数，如在SRCNN中，使用3层步长为1的同卷积，分别为（ 9×9 的64输出、 1×1 的32输出、 5×5 的3输出）效果会更好。

后来出现了另一种比较高效的方法ESPCN（实时的基于卷积神经网络的图像超分辨率方法）。ESPCN的核心概念是亚像素卷积层（sub-pixel convolutional layer），即先在原有的低像素图片上做卷积操作，最终输出一个含有多feature map的结果，保证总像素点与高分辨率的像素点总和是一致的，然后将多张低分辨率图片合并成一张高分辨率图片。例如，假设需要将低分辨率图片的像素扩大2倍（从 128×128 扩大到

256×256 ），就直接将其进行卷积操作，最终输出放大倍数的平方（ 2×2 ）个feature map，即 [batch_size, W, H, 4]（如果是RGB彩色图片就会是[batch_size, W, H, 12]）。以灰度图为例，将4个图片中的第一个像素取出成为重构图中的4个像素，依此类推，在重构图中的每个 2×2 区域都是由这4幅图对应位置的像素组成，最终形成形状为[batch_size, $2 \times W$, $2 \times H$, 1]大小的高分辨率图像。这个变换被称为sub-pixel convolution，如图12-12所示。

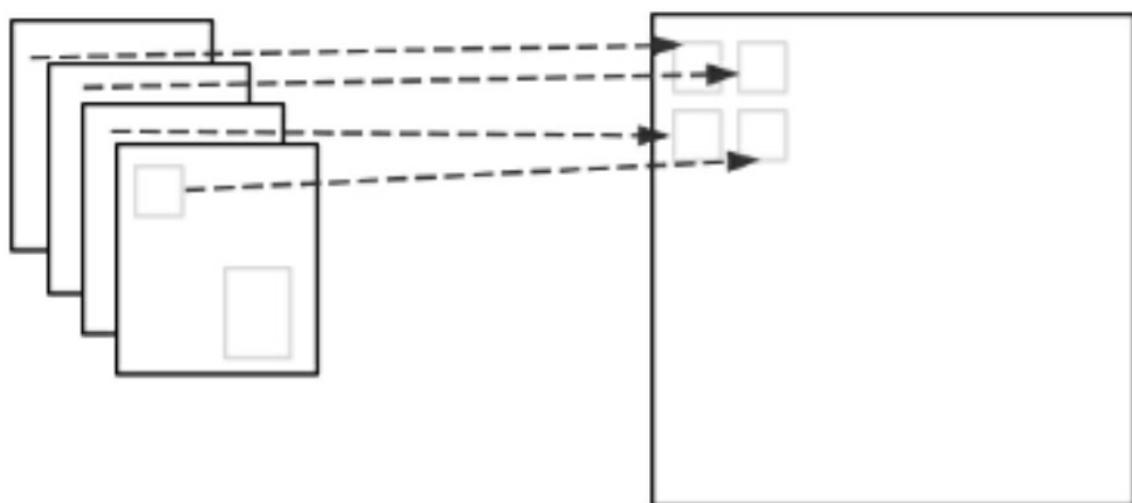


图12-12 ESPCN图例

sub-pixel convolution的方法只在最后一层进行图像低分辨率到高分辨率的大小变换，保证了前面的卷积运算均在低分辨率图像上进行，得到了更高的运算效率。

另外，基于视频图像的SR方法还有 VESPCN，这里不再展开介绍，有兴趣的读者可

*****ebook converter DEMO Watermarks*****

以自行学习。

3. TensorFlow中的图片变换函数

在TensorFlow中变化分辨率的函数主要是`tf.image.resize_images`，其具体原型如下：

```
def resize_images(images,
                  size,
                  method=ResizeMethod.BILINEAR,
                  align_corners=False):
```

前两个参数分别是输入的图片及要变化的尺寸，图片的形状为[batch, height, width, channels]或[height, width, channels]均可。第3个参数method的取值如下。

- `ResizeMethod.BILINEAR`: 表示使用双线性插值算法变化图片。
- `ResizeMethod.NEAREST_NEIGHBOR`: 表示使用邻近值插值算法变化图片。
- `ResizeMethod.BICUBIC`: 表示使用双立方插值算法变化图片。
- `ResizeMethod.AREA`: 表示使用面积插值算法变化图片。

具体算法这里不展开介绍，后面会通过实例演示其各个算法的效果。

还可以直接使用内部函数来做类似的处理，例如：

```
tf.image.resize_bicubic(images, size, align_corners=None, name=None)
tf.image.resize_nearest_neighbor(images, size, align_corners=None)
tf.image.resize_bilinear(images, size, align_corners=None, name=None)
```

与前面不同的是，images的格式只支持一种[batch, height, width, channels]。

4. TensorFlow的图像变化函数汇总

TensorFlow的图像变化函数如表12-1所示。

表12-1 图像变化函数汇总

操作	说明
tf.image.resize_images	见前面的介绍
tf.image.crop_to_bounding_box	按照指定框剪辑
tf.image.flip_left_right	水平反转
tf.image.flip_up_down	上线反转
tf.image.rot90(input,k=1)	旋转，k=1、2、3分别代表90°、180°和270°
tf.image.rgb_to_grayscale	RGB格式转化为灰度

12.8.2 实例93：ESPCN实现MNIST数据集的超分辨率重建

本实例主要通过对MNIST数据集实现超分辩

率的重建，来示范TensorFlow中相关图片变化的函数用法和效果，以及ESPCN的网络结构。该实例共分为以下几步骤。

实例描述

通过使用ESPCN网络，在MNIST数据集上将低分辨率图片复原成高分辨率图片，并与其他复原函数的生成结果进行比较。

1. 引入头文件，构建低分辨率样本

在头文件部分导入slim库，使用resize_bicubic来构建缩小4倍的低分辨率样本，将 28×28 的像素变为 14×14 （长、宽各缩小2倍）。

代码12-6 mnistEspcn

```
01 import tensorflow as tf
02 import matplotlib.pyplot as plt
03 import numpy as np
04 import tensorflow.contrib.slim as slim
05 from tensorflow.examples.tutorials.mnist import input_data
06 mnist = input_data.read_data_sets("/data/", one_hot=True)
07
08 batch_size = 30      # 获取样本的批次大小
09 n_input = 784        # MNIST data 输入(img shape: 28*28)
10 n_classes = 10       # MNIST 列别 (0-9, 一共10类)
11
12 # 待输入的样本图片
13 x = tf.placeholder("float", [None, n_input])
14 img = tf.reshape(x, [-1, 28, 28, 1])
15 # 缩小image
16 x_small = tf.image.resize_bicubic(img, (14, 14)) # 缩小一倍
```

2. 通过TensorFlow函数实现超分辨率

分别使用bicubic、nearest_neighbor和bilinear方法将分辨率还原，为了后续比较效果。

代码12-6 mnistEspcn（续）

```
17 x_bicubic = tf.image.resize_bicubic(x_small, (28, 28))#双
18 x_nearest = tf.image.resize_nearest_neighbor(x_small, (28, 28))
19 x_bilin = tf.image.resize_bilinear(x_small, (28, 28))
```

3. 建立ESPCN网络结构

建立一个简单的三层卷积网络：第1层使用 5×5 的卷积核，输出64通道的图片，slim卷积函数中使用的是默认激活函数Relu；第2层使用 3×3 的卷积核，输出的是32通道；最后一层使用 3×3 卷积核，生成4通道图片。这个4通道需要和恢复超分辨率缩放范围对应，4代表长、宽各放大2倍。接着使用tf.depth_to_space函数，将多张图片合并成一张图片。

tf.depth_to_space函数的意思是将深度数据按照块的模式展开重新排列，第一个输入是原始数据，第二个输入是块的尺寸，输入2则代表尺寸为 2×2 的块。而深度就是生成图片的通道数，即将每个通道对应的像素值填充到指定大小的块中。

代码12-6 mnistEspcn（续）

*****ebook converter DEMO Watermarks*****

```
20 #ESPCN网络结构
21 net = slim.conv2d(x_small, 64, 5)
22 net = slim.conv2d(net, 32, 3)
23 net = slim.conv2d(net, 4, 3)
24 net = tf.depth_to_space(net, 2)
25 print("net.shape", net.shape)
```

4. 构建loss及优化器

将图片重新调整形状（`reshape`）为
(batch_size, 784) 的形状，通过平方差来计算
loss，设定学习率为0.01，通过AdamOptimizer进
行优化。

代码12-6 mnistEspcn（续）

```
26 y_pred = tf.reshape(net, [-1, 784])
27
28 cost = tf.reduce_mean(tf.pow(x - y_pred, 2))
29 optimizer = tf.train.AdamOptimizer(0.01).minimize(cost)
```

5. 建立session，运行

令数据即循环100次。启动session进行迭代训
练。

代码12-6 mnistEspcn（续）

```
30 training_epochs = 100
31 display_step = 20
32
33 with tf.Session() as sess:
```

*****ebook converter DEMO Watermarks*****

```
34     sess.run(tf.global_variables_initializer())
35     total_batch = int(mnist.train.num_examples/batch_size)
36     # 启动循环开始训练
37     for epoch in range(training_epochs):
38         # 遍历全部数据集
39         for i in range(total_batch):
40             batch_xs, batch_ys = mnist.train.next_batch(1)
41             _, c = sess.run([optimizer, cost], feed_dict={x:batch_xs, y:batch_ys})
42             # 显示训练中的详细信息
43             if epoch % display_step == 0:
44                 print("Epoch:", '%04d' % (epoch+1),
45                       "cost=", "{:.9f}".format(c))
46
47     print("完成!")
```

6. 图示结果

为了比较效果，将原始图片、低分辨率图片、各种算法的变化图片及模型恢复的图片一起显示出来。代码如下：

代码12-6 mnistEspcn（续）

```
48 show_num = 10
49     encode_s, encode_b, encode_n ,encode_bi,y_predv= sess.run(
50         [x_small,x_bicubic,x_nearest,x_bilin,y_pred], feed_dict={x:mnist.test.images[:show_num]}) 
51
52     f, a = plt.subplots(6, 10, figsize=(10, 6))
53     for i in range(show_num):
54         a[0][i].imshow(np.reshape(mnist.test.images[i], (28, 28)))
55         a[1][i].imshow(np.reshape(encode_s[i], (14, 14)))
56         a[2][i].imshow(np.reshape(encode_b[i], (28, 28)))
57         a[3][i].imshow(np.reshape(encode_n[i], (28, 28)))
58         a[4][i].imshow(np.reshape(encode_bi[i], (28, 28)))
59         a[5][i].imshow(np.reshape(y_predv[i], (28, 28)))
60
plt.show()
```

运行代码，显示图片如图12-13所示。

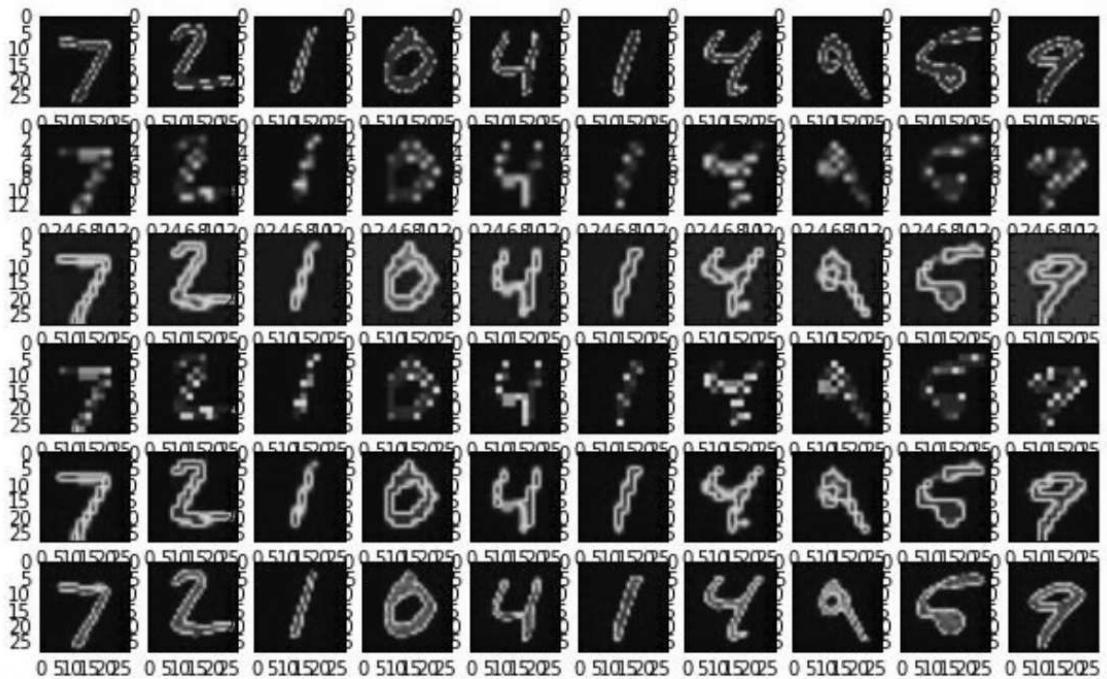


图12-13 ESPCN实例MNIST结果

最后一行是模型恢复的数据，可以看到，清晰度完全超过前面几行。Bicubic和bilinear效果还可以，nearest_neighbor是最差的。

12.8.3 实例94：ESPCN实现flowers数据集的超分辨率重建

前面的MNIST数据集轻巧方便，非常适合对模型的演示与理解。下面学习对彩色图片进行超分辨率的重建。彩色图片与MNIST样本不同的地方主要是，图片变为了3通道，并且像素点更多，而MNIST像素点更稀疏。所以应用在训练模型上，会有一些细节进行调节。通过对本例的学

*****ebook converter DEMO Watermarks*****

习，读者可以掌握对全彩色图片的一些处理技巧，这也是本例的主要意义。

本例主要实现对flowers数据集的图片处理。flowers数据集在第11章中的slim部分介绍过。本例同样还是使用slim模块进行数据的操作。另外flowers是尺寸不一的数据样本，所以在本例中也可以借鉴统一尺寸处理的方法。同样，本例还会示范TensorFlow中相关图片变化的函数用法和效果，以及ESPCN的网络结构。该实例共分为以下几步。

实例描述

通过使用ESPCN网络，在flower数据集上将低分辨率图片复原成高分辨率图片并与其他复原函数的生成结果进行比较。

1. 引入头文件，创建样本数据源

同样使用slim，这次使用的数据源是flowers，所以将该代码文件建立到models下面的slim下，然后就可以其引入flowers数据集了（这一步不熟悉的读者可以参考第11章的slim数据集部分）。

指定TFRecord文件夹，创建数据集。代码如下：

代码12-7 tfreco더SRESPCN

```
01 import tensorflow as tf
02 from datasets import flowers
03 import numpy as np
04 import matplotlib.pyplot as plt
05
06 slim = tf.contrib.slim
07
08 height = width = 200
09 batch_size = 4
10 DATA_DIR="D:/own/python/flower_photosos"
11
12 #选择数据集validation
13 dataset = flowers.get_split('validation', DATA_DIR)
14 #创建一个provider
15 provider = slim.dataset_data_provider.DatasetDataProvider(
    num_readers = 2)
16 #通过provider的get拿到内容
17 [image, label] = provider.get(['image', 'label'])
18 print(image.shape)
```

2. 获取批次样本并通过TensorFlow函数实现超分辨率

通过resize_image_with_crop_or_pad函数统一样本大小，大的剪掉，不够的加0填充。使用tf.train.batch函数获得指定批次数据images和labels。

代码12-7 tfreco더SRESPCN（续）

```
19 # 剪辑图片为统一大小
20 distorted_image = tf.image.resize_image_with_crop_or_pad(
    height, width) #剪辑尺寸，不够填充
21 ##########
22 images, labels = tf.train.batch([distorted_image, label],
```

*****ebook converter DEMO Watermarks*****

```
batch_size)
23 print(images.shape)
24
25 x_smalls = tf.image.resize_images(images, (np.int32(height),
int32(width/2)))# 尺寸变为原来的1/4
26 x_smalls2 = x_smalls/255.0
27 #还原
28 x_nearests = tf.image.resize_images(x_smalls, (height, width),
image.ResizeMethod.NEAREST_NEIGHBOR)
29 x_bilins = tf.image.resize_images(x_smalls, (height, width),
image.ResizeMethod.BILINEAR)
30 x_bicubics = tf.image.resize_images(x_smalls, (height, width),
image.ResizeMethod.BICUBIC)
```

先通过resize_images创建一个低分辨率图片x_smalls，然后将x_smalls通过不同算法的变化，生成对应的高分辨率图片。

3. 建立ESPCN网络结构

网络结构与上例一样，不同的是输入的图片做了归一化处理，统一除以255，使其变为0~1之间的数。最后一个卷积成输出的为12通道，代表 2×2 的缩放比例，一共3个通道，所以再乘以3。另外，各层均使用了Tanh函数，最后一层没有使用激活函数。

代码12-7 tfrecoederSRESPCN（续）

```
31 net = slim.conv2d(x_smalls2, 64, 5, activation_fn = tf.nn.relu)
32 net = slim.conv2d(net, 32, 3, activation_fn = tf.nn.tanh)
33 net = slim.conv2d(net, 12, 3, activation_fn = None)#2*2*3
34 y_predt = tf.depth_to_space(net, 2)
35
36 y_pred = y_predt*255.0
37 y_pred = tf.maximum(y_pred, 0)
```

*****ebook converter DEMO Watermarks*****

```
38 y_pred = tf.minimum(y_pred, 255)  
39  
40 dbatch=tf.concat([tf.cast(images,tf.float32),y_pred],0)
```

y_pred是由y_predt转化而来的，通过tf.maximum与tf.minimum函数将内部的值都变为0~255之间的数字。y_predt会参与损失值的计算。

dbatch是将生成的y_pred与images按照批次的维度合并起来，该张量是为了后面进行图片质量评估使用的。



注意：上面例子中y_pred进行了最大值和最小值的规整处理，这是个很常用的技巧。如果不处理，那么生成的图片会在显示时看到有亮点，使图片显得不清晰。读者可以试着将这段代码去掉，看看效果。

4. 构建loss及优化器

对于全彩色训练的学习率设定还是需要非常小心的，在这里设置为0.000001，让其缓慢地变化。由于输入的样本归一化了，所以计算loss时的images也需要归一化。代码如下：

代码12-7 tfrecoderSRESPCN（续）

```
41 cost = tf.reduce_mean(tf.pow( tf.cast(images,tf.float32),
42 optimizer = tf.train.AdamOptimizer(0.000001 ).minimize(cost)
```

5. 建立session，运行

启动session，运行15 0000次。代码如下：

代码12-7 tfrecoederSRESPCN（续）

```
43 training_epochs =150000
44 display_step =200
45
46 sess = tf.InteractiveSession()
47 sess.run(tf.global_variables_initializer())
48
49 #启动队列
50 tf.train.start_queue_runners(sess=sess)
51
52 # 启动循环开始训练
53 for epoch in range(training_epochs):
54
55     _, c = sess.run([optimizer, cost])
56     # 显示训练中的详细信息
57     if epoch % display_step == 0:
58         d_batch=dbatch.eval()
59         mse,psnr=batch_mse_psnr(d_batch)
60         ypsnr=batch_y_psnr(d_batch)
61         ssim=batch_ssimm(d_batch)
62         print("Epoch:", '%04d' % (epoch+1),
63               "cost=", "{:.9f}".format(c),"psnr",psnr,"ypr",
64
65 print("完成!")
```

在显示评估结果时，使用batch_mse_psnr、batch_y_psnr和batch_ssimm这3个函数分别对节点dbatch的值进行运算，得到图片的质量评估值。

6. 构建图片质量评估函数

SR图片质量有其自己的一套评估质量算法：常用的两个指标是PSNR（Peak Signal-to-Noise Ratio）和SSIM（Structure Similarity Index）。这两个值越高，代表重建结果的像素值和标准越接近。对于PSNR的计算有两个方法：

- 基于R、G、B，分别计算三通道中的MSE值再求平均值，然后再将结果代入求PSNR。
- 基于YUV，求图像YUV空间中的Y分量，计算Y分量的PSNR值。

对于YUV的介绍如下：

YUV（亦称YCrCb）是另一种颜色编码方法，常被欧洲电视系统所采用。Y代表亮度信号，U（R-Y）与V（B-Y）分别代表两个色差信号。在没有U和V时，就会表现为只有亮度的黑白颜色，彩色电视采用YUV空间正是为了用亮度信号Y解决彩色电视机与黑白电视机的兼容问题，使黑白电视机也能接收彩色电视信号。

YUV和RGB互相转换的公式如下（RGB取值范围均为0~255）：

$$Y = 0.299R + 0.587G + 0.114B$$
$$U = -0.147R - 0.289G + 0.436B$$

```
V = 0.615R-0.515G-0.100B  
R = Y + 1.14V  
G = Y-0.39U - 0.58V  
B = Y + 2.03U
```

将这3个指标用代码实现如下。

代码12-7 tfrecoderSRESPCN (续)

```
66 def batch_mse_psnr(dbatch):  
67     im1,im2=np.split(dbatch,2)  
68     mse=((im1-im2)**2).mean(axis=(1,2))  
69     psnr=np.mean(20*np.log10(255.0/np.sqrt(mse)))  
70     return np.mean(mse),psnr  
71 def batch_y_psnr(dbatch):  
72     r,g,b=np.split(dbatch,3, axis=3)  
73     y=np.squeeze(0.3*r+0.59*g+0.11*b)  
74     im1,im2=np.split(y,2)  
75     mse=((im1-im2)**2).mean(axis=(1,2))  
76     psnr=np.mean(20*np.log10(255.0/np.sqrt(mse)))  
77     return psnr  
78 def batch_ssim(dbatch):  
79     im1,im2=np.split(dbatch,2)  
80     imgsize=im1.shape[1]*im1.shape[2]  
81     avg1=im1.mean((1,2),keepdims=1)  
82     avg2=im2.mean((1,2),keepdims=1)  
83     std1=im1.std((1,2),ddof=1)  
84     std2=im2.std((1,2),ddof=1)  
85     cov=((im1-avg1)*(im2-avg2)).mean((1,2))*imgsize/(imgsize-2)  
86     avg1=np.squeeze(avg1)  
87     avg2=np.squeeze(avg2)  
88     k1=0.01  
89     k2=0.03  
90     c1=(k1*255)**2  
91     c2=(k2*255)**2  
92     c3=c2/2  
93     return np.mean((2*avg1*avg2+c1)*2*(cov+c3)/(avg1**2+c1)*(std1**2+std2**2+c2))
```

7. 图示结果

与前面例子类似，将原始图片与函数变化的图片及模型输出的图片一并显示。这里先定义一个函数用来统一显示。



注意： 必须要将其转化为UINT8的形式，否则图片会显示不出来。

代码12-7 tfrecoderSRESPCN（续）

```
94 def showresult(subplot,title,orgimg,thisimg,dopsnr = True
95     p = plt.subplot(subplot)
96     p.axis('off')
97     p.imshow(np.asarray(thisimg[0], dtype='uint8'))
98     if dopsnr :
99         conimg = np.concatenate((orgimg,thisimg))
100        mse,psnr=batch_mse_psnr(conimg)
101        ypsnr=batch_y_psnr(conimg)
102        ssim=batch_ssim(conimg)
103        p.set_title(title+str(int(psnr))+" y:"+str(int(
104            s)+str(ssim)))
104    else:
105        p.set_title(title)
```

接着取一批次的图片放入模型，调用 Showresult函数将生成的结果及评分值全部显示出来。

代码12-7 tfrecoderSRESPCN（续）

```
106 imagesv, label_batch,x_smallv,x_nearestv,x_bilinv,x_bicu
= sess.run([images, labels,x_smalls,x_nearests,x_bilins,x_b:
y_pred])
107 print("原",np.shape(imagesv),"缩放后的",np.shape(x_smallv
```

```
108
109 #显示
110 plt.figure(figsize=(20,10))
111
112 showresult(161,"org",imagesv,imagesv,False)
113 showresult(162,"small/4",imagesv,x_smallv,False)
114 showresult(163,"near",imagesv,x_nearestv)
115 showresult(164,"biline",imagesv,x_bilinv)
116 showresult(165,"bicubicv",imagesv,x_bicubicv)
117 showresult(166,"pred",imagesv,y_predv)
118
119 plt.show()
```

运行代码，输出如下，输出图片如图12-14所示。

```
.....
Epoch: 144801 cost= 0.003637410 psnr 23.8877 ypsnr 24.0189 ss
Epoch: 145001 cost= 0.008538806 psnr 25.0453 ypsnr 25.3529 ss
Epoch: 145201 cost= 0.005899625 psnr 28.1946 ypsnr 29.1575 ss
Epoch: 145401 cost= 0.002309756 psnr 25.6251 ypsnr 25.5808 ss
Epoch: 145601 cost= 0.004211991 psnr 25.2114 ypsnr 25.2179 ss
Epoch: 145801 cost= 0.002519545 psnr 27.9464 ypsnr 28.9226 ss
Epoch: 146001 cost= 0.005268521 psnr 20.8838 ypsnr 20.7228 ss
Epoch: 146201 cost= 0.002536027 psnr 23.988 ypsnr 24.3302 ss
Epoch: 146401 cost= 0.003322446 psnr 28.2296 ypsnr 29.4929 ss
Epoch: 146601 cost= 0.007955125 psnr 25.5261 ypsnr 26.271 ss
Epoch: 146801 cost= 0.002779651 psnr 29.1436 ypsnr 30.8613 ss
Epoch: 147001 cost= 0.005602385 psnr 24.6309 ypsnr 25.1222 ss
Epoch: 147201 cost= 0.004883423 psnr 25.3241 ypsnr 25.7193 ss
Epoch: 147401 cost= 0.005192784 psnr 26.5626 ypsnr 26.9226 ss
Epoch: 147601 cost= 0.006907145 psnr 27.7884 ypsnr 28.4125 ss
Epoch: 147801 cost= 0.008132000 psnr 26.7713 ypsnr 27.9356 ss
Epoch: 148001 cost= 0.008132160 psnr 24.6795 ypsnr 26.011 ss
Epoch: 148201 cost= 0.003620633 psnr 24.5258 ypsnr 24.9886 ss
Epoch: 148401 cost= 0.008644918 psnr 21.6561 ypsnr 21.704 ss
Epoch: 148601 cost= 0.003554154 psnr 25.849 ypsnr 25.9136 ss
Epoch: 148801 cost= 0.003299494 psnr 23.6707 ypsnr 24.2183 ss
Epoch: 149001 cost= 0.003197462 psnr 23.7814 ypsnr 24.1327 ss
Epoch: 149201 cost= 0.001375712 psnr 26.5407 ypsnr 26.7266 ss
Epoch: 149401 cost= 0.003641539 psnr 25.5488 ypsnr 26.2268 ss
Epoch: 149601 cost= 0.003025041 psnr 25.2158 ypsnr 25.65 ss
Epoch: 149801 cost= 0.001514586 psnr 27.9188 ypsnr 28.6265 ss
完成!
```

*****ebook converter DEMO Watermarks*****

原 (4, 200, 200, 3) 缩放后的 (4, 100, 100, 3) [3 0 0 1]

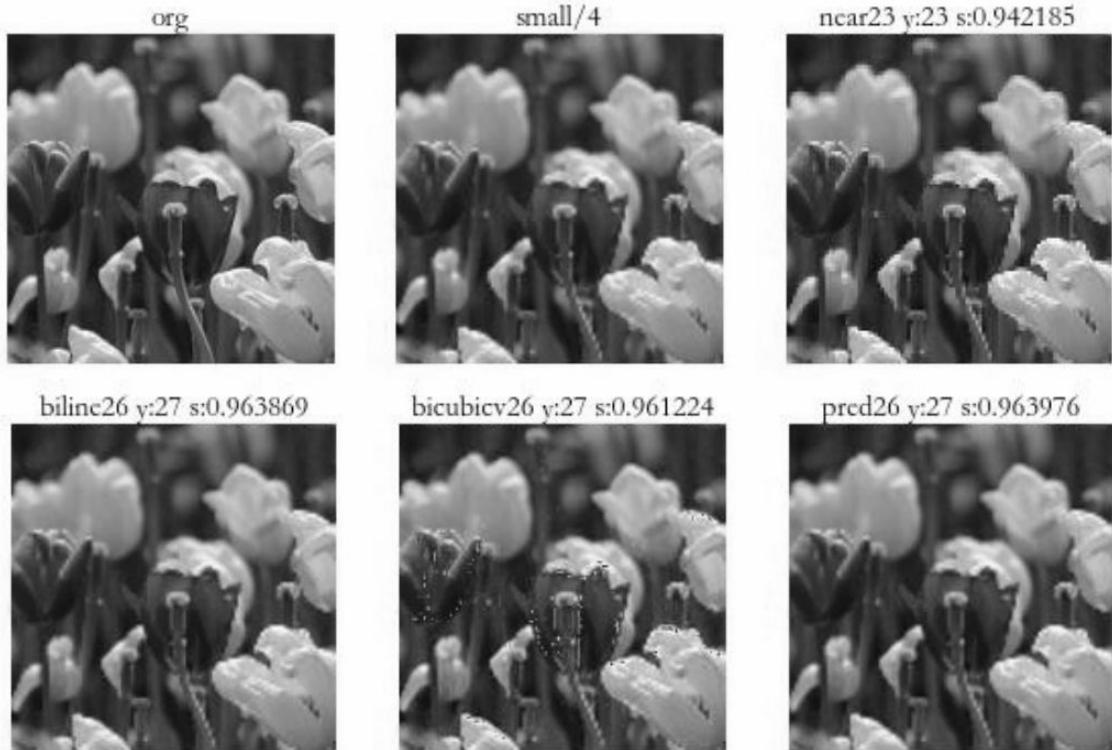


图12-14 ESPCN实例flowers结果

结果可见，使用ESPCN可以实现很好的SR效果。

本例仅将图片分辨率放大了2倍，而且与BILINEAR的比较优势不大，但没关系，下面就来演示一个更明显的例子，通过进一步优化网络将图片分辨率放大4倍。

12.8.4 实例95：使用残差网络的ESPCN

在实例94中ESPCN与BILINEAR的结果比较优势没有那么明显，这是因为普通算法在仅仅放

大两倍的图片处理上是很优秀的，另一个原因也是由于例子中的网络结构过于简单（仅三层）。下面通过对实例94的网络结构优化，实现在分辨率放大4倍任务上的图片重建。

实例描述

将flower数据集中的图片转成低分辨率，再通过使用带残差网络的ESPCN网络复原成高分辨率图片，并与其他复原函数的生成结果进行比较。

具体实现步骤如下。

1. 修改输入图片分辨率

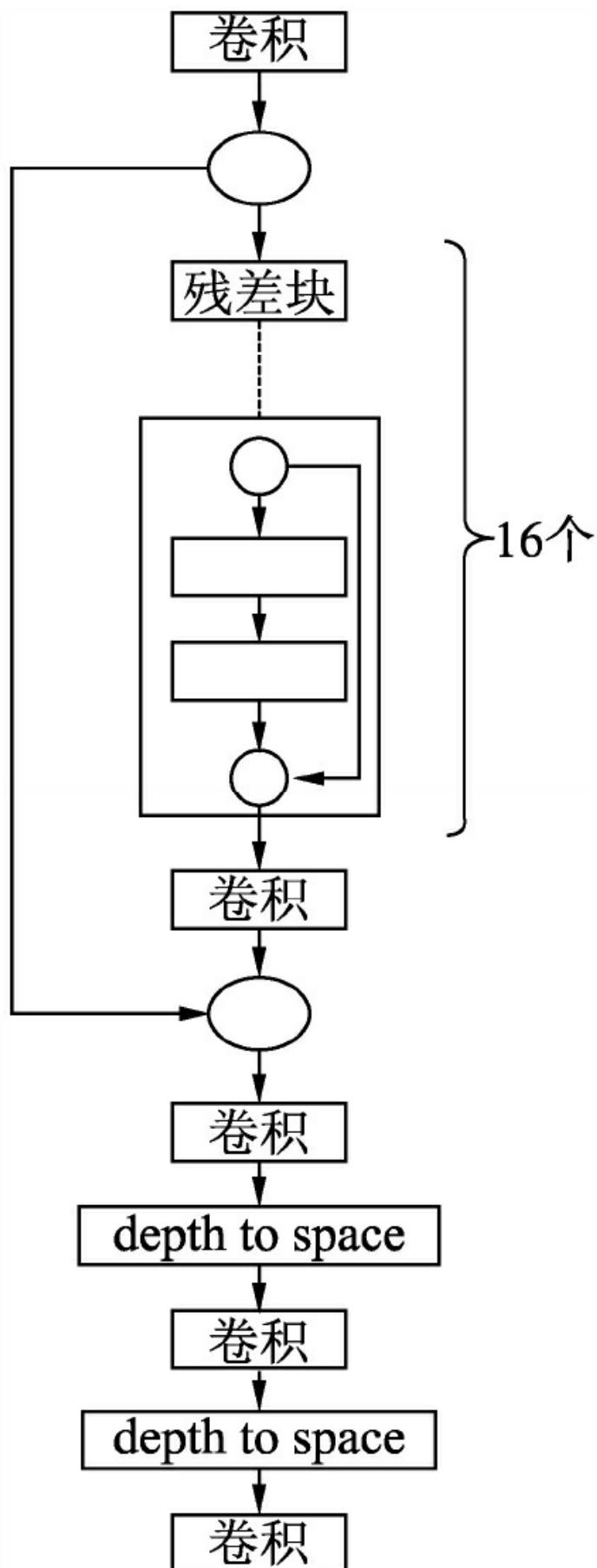
在代码“12-7 tfrecoedrSRESPCN.py”的基础上进行修改，将原来的输入尺寸由长宽各缩小一半变为长宽各缩小为原来的1/4。

代码12-8 resESPCN

```
01 .....
02 images, labels = tf.train.batch([distorted_image, label],
size=batch_size)
03 print(images.shape)
04
05 x_smalls = tf.image.resize_images(images, (np.int32(height
int32(width/4)))# 尺寸变为原来的1/16
06 x_smalls2 = x_smalls/255.0
07 .....
```

2. 添加残差网络

添加两个函数，一个是leaky_relu为leaky relu激活函数，另一个是用于生成网络残差块的函数residual_block，实现一个中间有两层卷积的残差块。接着在整个网络构造中，通过一个卷积层与一个残差层完成图像特征的转换。残差层是由16个残差块与一个卷积层组成的网络。特征转换之后再通过5层神经网络完成最终的特征修复处理过程，如图12-15所示。



*****ebook converter DEMO Watermarks*****

图12-15 ResESPCN例子结构

图12-15中，最下面的5层为修复特征数据，第1层是一个卷积层，第2层会按照 2×2 大小的像素块将第一层的结果展开，第3层与第1层一样，第4层与第2层一样，第5层也是个卷积层。连续2次变换进行放大4倍的处理，最终通过输出3通道的卷积生成最终修复图片。

代码12-8 resESPCN（续）

```
08 def leaky_relu(x, alpha=0.1, name='lrelu'):
09     with tf.name_scope(name):
10         x=tf.maximum(x, alpha*x)
11     return x
12 def residual_block(nn, i, name='resblock'):
13     with tf.variable_scope(name+str(i)):
14         conv1=slim.conv2d(nn, 64, 3, activation_fn = leaky_
normalizer_fn=slim.batch_norm)
15         conv2=slim.conv2d(conv1, 64, 3, activation_fn = leaky_
normalizer_fn=slim.batch_norm)
16         return tf.add(nn,conv2)
17
18 net = slim.conv2d(x_smalls2, 64, 5, activation_fn = leaky_
19 block=[]
20 for i in range(16):
21     block.append(residual_block(block[-1] if i else net,:_
22 conv2=slim.conv2d(block[-1], 64, 3, activation_fn = leaky_
normalizer_fn=slim.batch_norm)
23 sum1=tf.add(conv2,net)
24
25 conv3=slim.conv2d(sum1, 256, 3, activation_fn = None)
26 ps1=tf.depth_to_space(conv3,2)
27 relu2=leaky_relu(ps1)
28 conv4=slim.conv2d(relu2, 256, 3, activation_fn = None)
29 ps2=tf.depth_to_space(conv4,2) #再放大两倍
30 relu3=leaky_relu(ps2)
31 y_predt=slim.conv2d(relu3, 3, 3, activation_fn = None) #输
```

3. 修改学习率，进行网络训练

将学习率改为0.001，同样使用AdamOptimizer优化方法，循环迭代100 000次开始训练。

代码12-8 resESPCN（续）

```
32 ....
33 learn_rate =0.001
34
35 cost = tf.reduce_mean(tf.pow( tf.cast(images,tf.float32),
y_predt, 2))
36 optimizer = tf.train.AdamOptimizer(learn_rate ).minimize(
37 training_epochs =10000
```

4. 添加检测点

网络结构的修改会使单次训练的时间变长，因此有必要添加检查点文件保存功能。先对变量flags赋值定义检查点保存的路径，在session中读取到检查点文件后解析出运行的迭代次数。在range中设置起始次数，让其继续训练。

代码12-8 resESPCN（续）

```
38 ....
39 flags='b'+str(batch_size)+'_h'+str(height/4)+'_r'+str(lear
rate)+'_res'
40 if not os.path.exists('save'):
41     os.mkdir('save')
42 save_path='save/tf_'+flags
43 if not os.path.exists(save_path):
```

*****ebook converter DEMO Watermarks*****

```
44     os.mkdir(save_path)
45 saver = tf.train.Saver(max_to_keep=1) # 生成saver
46
47 sess = tf.InteractiveSession()
48 sess.run(tf.global_variables_initializer())
49
50 kpt = tf.train.latest_checkpoint(save_path)
51 print(kpt)
52 startepo= 0
53 if kpt!=None:
54     saver.restore(sess, kpt)
55     ind = kpt.find("-")
56     startepo = int(kpt[ind+1:])
57     print("startepo=",startepo)
58
59 #启动队列
60 tf.train.start_queue_runners(sess=sess)
61
62 # 启动循环开始训练
63 for epoch in range(startepo,training_epochs):
64 .....
65         print("Epoch:", '%04d' % (epoch+1),
66               "cost=", "{:.9f}".format(c),"psnr",psnr,"ypr",
67
68         saver.save(sess, save_path+"/tfrecord.cpkt", global_step=epoch)
69 print("完成!")
70 saver.save(sess, save_path+"/tfrecord.cpkt", global_step=
```

在迭代指定次数后保存检查点，并且在全部训练结束后保存检查点文件。

运行整个代码生成结果如下，输出图片如图12-16所示。

```
.....
Epoch: 4801 cost= 0.003252075 psnr 24.5383 ypsnr 25.332 ssim
Epoch: 5201 cost= 0.002841802 psnr 26.1108 ypsnr 26.599 ssim
Epoch: 5601 cost= 0.004468028 psnr 25.7363 ypsnr 26.3008 ssim
Epoch: 6001 cost= 0.004859785 psnr 26.1274 ypsnr 26.7174 ssim
Epoch: 6401 cost= 0.004147850 psnr 25.9059 ypsnr 26.7208 ssim
Epoch: 6801 cost= 0.003628785 psnr 25.7018 ypsnr 26.4992 ssim
```

*****ebook converter DEMO Watermarks*****

```
Epoch: 7201 cost= 0.002464779 psnr 23.9676 ypsnr 24.4634 ss:  
Epoch: 7601 cost= 0.003710205 psnr 24.7987 ypsnr 25.4836 ss:  
Epoch: 8001 cost= 0.002421107 psnr 27.0966 ypsnr 28.1542 ss:  
Epoch: 8401 cost= 0.003401657 psnr 25.6712 ypsnr 26.4028 ss:  
Epoch: 8801 cost= 0.003299317 psnr 26.5742 ypsnr 27.1549 ss:  
Epoch: 9201 cost= 0.002930838 psnr 26.4124 ypsnr 26.9374 ss:  
Epoch: 9601 cost= 0.003253016 psnr 25.8007 ypsnr 26.2591 ss:  
完成!  
原 (16, 256, 256, 3)缩放后的(16, 64, 64, 3)[0 2 4 1 0 4 1 3 0 0 1 1
```

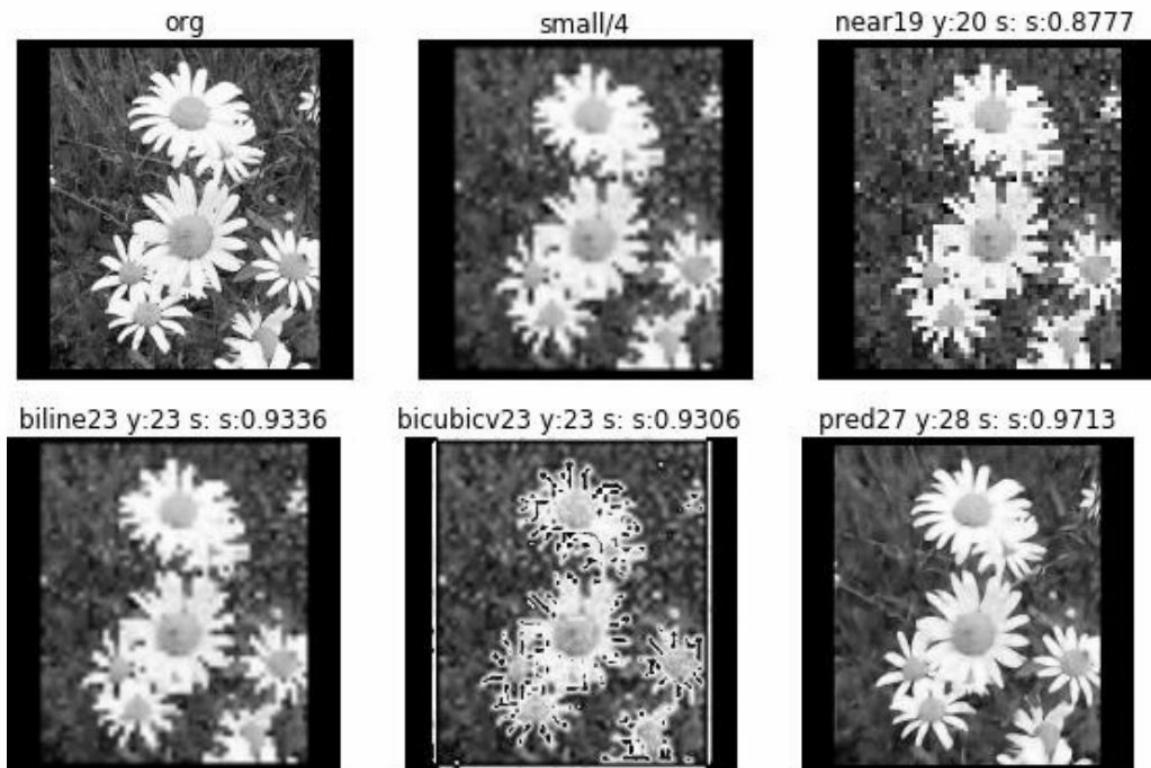


图12-16 resESPCN例子结果

图12-16中最后一幅图为模型生成的图片。放大4倍后可以看到，直接使用`resize_images`生成的图片已经得不到很好的效果，但是通过模型生成的图片却有着同样高质量的清晰度。



注意： 在构建相对较大型复杂网络结构（类似该例子的结构）进行训练时，检查点的设
*****ebook converter DEMO Watermarks*****

置是必须的，常常是运行一段，调节下参数，观察效果，然后再运行，再调节参数，再观察效果。这里有个技巧：如本例中将结构中的参数组成flags用于对检查点目录动态命名，这么做可以在调节参数时不用再额外考虑修改路径的问题，同时又能为不同参数对应的模型留下清晰的备份，便于比较。

5. 练习题

想一想，图12-16中从左向右的方向上，倒数第二幅图片为什么会如此显示？有什么办法能让其正常显示？

答案在12.8.3节中第3个小节的“注意”事项里，读者可以参考对应的代码完成该功能。

12.8.5 SRGAN的原理

在图像放大4倍以上时，前面所介绍的方法得到的结果显得过于平滑，而缺少一些细节上的真实感。这是因为，传统方法使用的代价函数是基于像素点的最小均方差（MSE），该代价函数使重建结果有较高的信噪比，但是缺少了高频信息，所以会出现过度平滑的纹理。

SRGAN的思想是，使重建的高分辨率图像与真实的高分辨率图像，无论是低层次的像素值还

是高层次的抽象特征及整体概念及风格上，都应当接近。

其中，对整体概念和风格的评估可以使用一个判别器，判断一幅高分辨率图像是由算法生成的还是真实的图像。如果一个判别器无法区分出来，那么由算法生成的图像就达到了对超分辨率修复成功的效果。

输入图片自身内容方面的损失值与来自对抗神经网络的损失值一起组成了最终的损失值（loss）。而对于自己的内容方面，基于像素点的平方差是一部分，另一部分是基于特征空间的平方差。基于特征空间特征的提取使用了VGG网络。

12.8.6 实例96：使用SRGAN实现flowers数据集的超分辨率修复

本例中用SRGAN在有基于残差网络的ESPCN上面进行SR处理，观察它能带给我们怎样的效果。由于在计算生成器loss中的一部分需要使用VGG网络来提取特征，因此本例会用到第11章中的VGG19预训练模型。为了方便训练，这里直接使用了前面训练好的残差网络ESPCN网络模型作为生成器，用其生成的图片作为判别器的输入，通过GAN的机制进行二次优化。

实例描述

将flower数据集中的图片转为低分辨率，通过使用SRGAN网络将其还原成高分辨率，并与其他复原函数的生成结果进行比较。

具体实现步骤如下。

1. 引入头文件，图片预处理

在代码“12-8 resESPCN.py”基础上进行修改，引入slim中VGG网络头文件。样本部分与前面一样，只是增加了一个对输入的图片做归一化处理。

代码12-9 rsgan

```
01 import tensorflow as tf
02 import time
03 import os
04 import numpy as np
05 import matplotlib.pyplot as plt
06 from nets import vgg
07 .....
08
09 images, labels = tf.train.batch([distorted_image, label],
size=batch_size)
10 print(images.shape)
11
12 images = tf.cast(images, tf.float32)
13 x_small1s=tf.image.resize_bicubic(images,[np.int32(height,
int32(width/4))]) # 变为原来的1/16
14 x_small1s2 = x_small1s/127.5-1 #将输入样本进行归一化处理
```

由于图片中每个像素都在0~255之间，所以除以255/2之后就会变为0~2之间的值，再减去1，就得到了x_smalls2。

2. 构建生成器

为了可以重用代码“12-8 resESPCN.py”中的模型，生成器的代码要与代码“12-8 resESPCN.py”保持一致，这里直接复制到gen函数中。

代码12-9 rsgan（续）

```
15 def gen(x_smalls2 ):
16     net = slim.conv2d(x_smalls2, 64, 5, activation_fn = leaky_relu)
17     block=[]
18     for i in range(16):
19         block.append(residual_block(block[-1] if i else net))
20         conv2=slim.conv2d(block[-1], 64, 3, activation_fn = leaky_relu,
21                         normalizer_fn=slim.batch_norm)
22         sum1=tf.add(conv2,net)
23
24         conv3=slim.conv2d(sum1, 256, 3, activation_fn = None)
25         ps1=tf.depth_to_space(conv3,2)
26         relu2=leaky_relu(ps1)
27         conv4=slim.conv2d(relu2, 256, 3, activation_fn = None)
28         ps2=tf.depth_to_space(conv4,2) #再放大两倍
29         relu3=leaky_relu(ps2)
30         y_predt=slim.conv2d(relu3, 3, 3, activation_fn = None)
31     return y_predt
```

3. VGG的预输入处理

为了得到生成器基于内容的loss，要将生成的图片与真实图片分别输入VGG网络以获得它们

*****ebook converter DEMO Watermarks*****

的特征，然后在特征空间上计算loss。所以先将低分辨率图片作为输入放进生成器gen函数中，得到生成图片resnetimg，并将图片还原成0~255区间的正常像素值。同时准备好生成器的训练参数gen_var_list为后面优化器使用做准备。



注意：本例使用了一个新方法来提取已有模型的参数：①在生成器定义好后，获取一次图（运算任务）中的所有变量；②将已有模型载入判别器后，再获取一次图（运算任务）中的所有变量。由于两次执行的时间不一样，第一次得到的仅仅是生成器gen的变量，而第二次得到的是gen和判别器的变量总和，所以从变量总和中去掉第一次的变量剩下的就是判别器的变量。这么做的目的是为了再次使用前面例子里已经训练好的模型。

使用VGG模型时，必须在输入之前对图片做RGB均值的预处理。先定义处理RGB均值的函数，然后做具体变换。

代码12-9 rsgan（续）

```
31 def rgbmeanfun(rgb):
32     _R_MEAN = 123.68
33     _G_MEAN = 116.78
34     _B_MEAN = 103.94
35     print("build model started")
36     # 将 RGB 转化成BGR
37     red, green, blue = tf.split(axis=3, num_or_size_splits=3, value=rgb)
38     bgr = tf.concat([blue, green, red], axis=3)
39
40     return bgr
```

```
    value=rgb)
38      rgbmean = tf.concat(axis=3, values=[red - _R_MEAN, gr
39      blue - _B_MEAN,]))
40      return rgbmean
41
41 resnetimg=gen(x_small1s2)
42 result=(resnetimg+1)*127.5
43 gen_var_list=tf.get_collection(tf.GraphKeys.TRAINABLE_VAF
44
45 y_pred = tf.maximum(result,0)
46 y_pred = tf.minimum(y_pred,255)
47
48 dbatch=tf.concat([images,result],0)
49 rgbmean = rgbmeanfun(dbatch)
```

RGB的处理与第11章中一样。具体的细节可以回看第11章的内容。



注意： 这里将生成的图片与真实的图片通过维度为0的concat组合在一起处理。这是一个编写代码的小技巧。因为真实图片与生成的图片其处理过程是一样的（都要经过预处理，然后放到判别器中），所以就一起打包，这相当于两个batch的数据进行处理然后塞进判别式中，得到结果后再按照打包的先后顺序将它们分开即可。这种做法只用一套代码就可以完成真实图片和生成图片的处理。

4. 计算VGG特征空间的loss

VGG中的前5个卷积层用于特征提取，所以在使用时，只取其第5个卷积层的输出节点，其他的节点可以全部忽略。那么问题来了，如何能拿

*****ebook converter DEMO Watermarks*****

到模型中的指定节点呢？可以通过slim中nets文件夹下对应的VGG源码找到对应节点的名称。这里使用了一个更简单的方法：直接在models\slim\nets文件夹下打开“vgg_test.py”文件，在第50行中可以找到testEndPoints函数，其内容如下：

```
.....  
def testEndPoints(self):  
    batch_size = 5  
    height, width = 224, 224  
    num_classes = 1000  
    with self.test_session():  
        inputs = tf.random_uniform((batch_size, height, width,  
        _, end_points = vgg.vgg_19(inputs, num_classes)  
    expected_names = [  
        'vgg_19/conv1/conv1_1',  
        'vgg_19/conv1/conv1_2',  
        'vgg_19/pool1',  
        'vgg_19/conv2/conv2_1',  
        'vgg_19/conv2/conv2_2',  
        'vgg_19/pool2',  
        'vgg_19/conv3/conv3_1',  
        'vgg_19/conv3/conv3_2',  
        'vgg_19/conv3/conv3_3',  
        'vgg_19/conv3/conv3_4',  
        'vgg_19/pool3',  
        'vgg_19/conv4/conv4_1',  
        'vgg_19/conv4/conv4_2',  
        'vgg_19/conv4/conv4_3',  
        'vgg_19/conv4/conv4_4',  
        'vgg_19/pool4',  
        'vgg_19/conv5/conv5_1',  
        'vgg_19/conv5/conv5_2',  
        'vgg_19/conv5/conv5_3',  
        'vgg_19/conv5/conv5_4',  
        'vgg_19/pool5',  
        'vgg_19/fc6',  
        'vgg_19/fc7',  
        'vgg_19/fc8'  
    ]  
    self.assertSetEqual(set(end_points.keys()), set(expected_names))
```

*****ebook converter DEMO Watermarks*****

这是一个单元测试函数，里面列举了VGG19网络结构中所有的节点名称。如上代码'vgg_19/conv5/conv5_4'就是本例中想要的节点，直接将该字符串复制放到本例代码中，如下所示。

代码12-9 rsgan（续）

```
50 #vgg 特征值
51 _, end_points = vgg.vgg_19(rgbmean, num_classes=1000, is_1
False, spatial_squeeze=False)
52 conv54=end_points['vgg_19/conv5/conv5_4']
53 print("vgg.conv5_4",conv54.shape)
54 fmap=tf.split(conv54,2)
55
56 content_loss=tf.losses.mean_squared_error(fmap[0],fmap[1]
```

由于前面通过concat将两个图片放一起来处理，得到结果后，还要使用split将其分开，接着通过平方差算出基于特征空间的loss。

5. 判别器的构建

判别器主要是通过一系列卷积层组合起来所构成的，最终使用两个全连接层实现映射到一维的输出结果。具体函数实现如下：

代码12-9 rsgan（续）

```
57 def Discriminator(dbatch, name ="Discriminator"):
*****ebook converter DEMO Watermarks*****
```

```
58     with tf.variable_scope(name):
59         net = slim.conv2d(dbatch, 64, 1, activation_fn = :
60
61         ochannels=[64,128,128,256,256,512,512]
62         stride=[2,1]
63
64         for i in range(7):
65             net = slim.conv2d(net, ochannels[i], 3, stride
66             [i%2], activation_fn = leaky_relu, normalizer_fn=:
67             batch_norm, scope='block'+str(i))
68
69         dense1 = slim.fully_connected(net, 1024, activat:
70             fn=leaky_relu)
71         dense2 = slim.fully_connected(dense1, 1, activat:
72             fn=tf.nn.sigmoid)
73
74     return dense2
```

6. 计算loss， 定义优化器

将判别器的结果裁开， 分别得到真实图片与生成图片判别的结果， 以LSGAN的方式计算生成器与判别器的loss，在生成器loss中加入基于特征空间的loss。按照前面第3步中所讲的训练参数的获取方式获得判别器训练参数disc_var_list， 使用AdamOptimizer优化loss值。代码如下：

代码12-9 rsgan（续）

```
71 disc=Discriminator(dbatch)
72 D_x,D_G_z=tf.split(tf.squeeze(disc),2)
73
74 adv_loss=tf.reduce_mean(tf.square(D_G_z-1.0))
75
76 gen_loss=(adv_loss+content_loss)
77 disc_loss=(tf.reduce_mean(tf.square(D_x-1.0))+tf.square(D_
78
79 disc_var_list=tf.get_collection(tf.GraphKeys.TRAINABLE_V
*****ebook converter DEMO Watermarks*****
```

```
80 print("len----",len(disc_var_list),len(gen_var_list))
81 for x in gen_var_list:
82     disc_var_list.remove(x)
83
84 learn_rate =0.001
85 global_step=tf.Variable(0,trainable=0,name='global_step')
86 gen_train_step=tf.train.AdamOptimizer(learn_rate).minimize(
87     gen_loss,global_step,gen_var_list)
88 disc_train_step=tf.train.AdamOptimizer(learn_rate).minimize(
89     disc_loss,global_step,disc_var_list)
```

7. 指定准备载入的预训练模型路径

这次需要对3个检查点路径进行配置，第一个是本程序的SRGAN检查点文件，第二个是srResNet检查点文件，最后一个VGG模型文件。

代码12-9 rsgan（续）

```
88 #残差网络检查点文件相关定义
89 flags='b'+str(batch_size) +'_r'+str(np.int32(height/4))+'_
90 (learn_rate) +'rsgan'
91 save_path='save/srgan_'+flags
92 if not os.path.exists(save_path):
93     os.mkdir(save_path)
94 saver = tf.train.Saver(max_to_keep=1) # 生成saver
95
96 srResNet_path='./save/tf_b16_h64.0_r0.001_res/'
97 srResNetloader = tf.train.Saver(var_list=gen_var_list) #
98
99 #VGG检查点
100 checkpoints_dir = 'vgg_19_2016_08_28'
101 init_fn = slim.assign_from_checkpoint_fn(
102     os.path.join(checkpoints_dir, 'vgg_19.ckpt'),
103     slim.get_model_variables('vgg_19'))
```



注意： 这里对于srResNet的变量恢复，要在建立Saver时传入var_list参数来指定好所要恢复的变量列表，否则默认是恢复所有变量，但是模型里却找不到其他变量，会报错误。

8. 启动session从检查点恢复变量

对于VGG模型的恢复，可以参考第11章中的内容。其他的检查点恢复的写法与前面一样。代码如下：

代码12-9 rsgan（续）

```
103 log_steps=100
104 training_epochs=16000
105
106 with tf.Session() as sess:
107
108     sess.run(tf.global_variables_initializer())
109
110     init_fn(sess)
111
112     kpt = tf.train.latest_checkpoint(srResNet_path)
113     print("srResNet_path",kpt,srResNet_path)
114     startepo= 0
115     if kpt!=None:
116         srResNetloader.restore(sess, kpt)
117         ind = kpt.find("-")
118         startepo = int(kpt[ind+1:])
119         print("srResNetloader global_step=",global_step)
120
121     kpt = tf.train.latest_checkpoint(save_path)
122     print("srgan",kpt)
123     startepo= 0
124     if kpt!=None:
125         saver.restore(sess, kpt)
126         ind = kpt.find("-")
```

```
127         startepo = int(kpt[ind+1:])
128         print("global_step=", global_step.eval(), startepo)
```

9. 启动带协调器的队列线程，开始训练

本例中涉及的参数比较多，模型比较大，会导致每次迭代时间都很长，所以加入检测点是非常有必要的。这里涉及检查点保存的粒度，如间隔太短，因为频繁地写文件会减慢训练速度，如果设置的间隔太长，中途如发生意外暂停会导致浪费了一部分训练时间，可以通过try的方式在异常捕获时再保存一次检查点，这样可以把中途的训练结果保存下来。

代码12-9 rsgan（续）

```
129 coord = tf.train.Coordinator()
130     threads = tf.train.start_queue_runners(sess, coord)
131
132     try:
133         def train(endpoint,gen_step,disc_step):
134             while global_step.eval()<=endpoint:
135
136                 if((global_step.eval()/2)%log_steps==0):
137
138                     d_batch=dbatch.eval()
139                     mse,psnr=batch_mse_psnr(d_batch)
140                     ssim=batch_ssimm(d_batch)
141                     s=time.strftime('%Y-%m-%d %H:%M:%S:'
142                         (time.time()))+'step='+str(global_step.e
143                         mse='+str(mse)+' psnr='+str(psnr)+'
144                         (ssim)'+ gen_loss='+str(gen_loss.eval())
145                         loss='+str(disc_loss.eval()))
146                     print(s)
147                     f=open('info.train_'+flags, 'a')
148                     f.write(s+'\n')
149                     f.close()
```

*****ebook converter DEMO Watermarks*****

```
146             saver.save(sess, save_path+"/srgan.cpkt", global_step=global_step.eval())
147
148             sess.run(disc_step)
149             sess.run(gen_step)
150             train(training_epochs, gen_train_step, disc_train_step)
151             print('训练完成')
152 ..... #显示部分同resEpcn例子，代码省略
153     except tf.errors.OutOfRangeError:
154         print('Done training -- epoch limit reached')
155     except KeyboardInterrupt:
156         print("Ending Training...")
157         saver.save(sess, save_path+"/srgan.cpkt", global_step=global_step.eval())
158     finally:
159         coord.request_stop()
160
161     coord.join(threads)
```

运行代码，生成结果如图12-17所示。

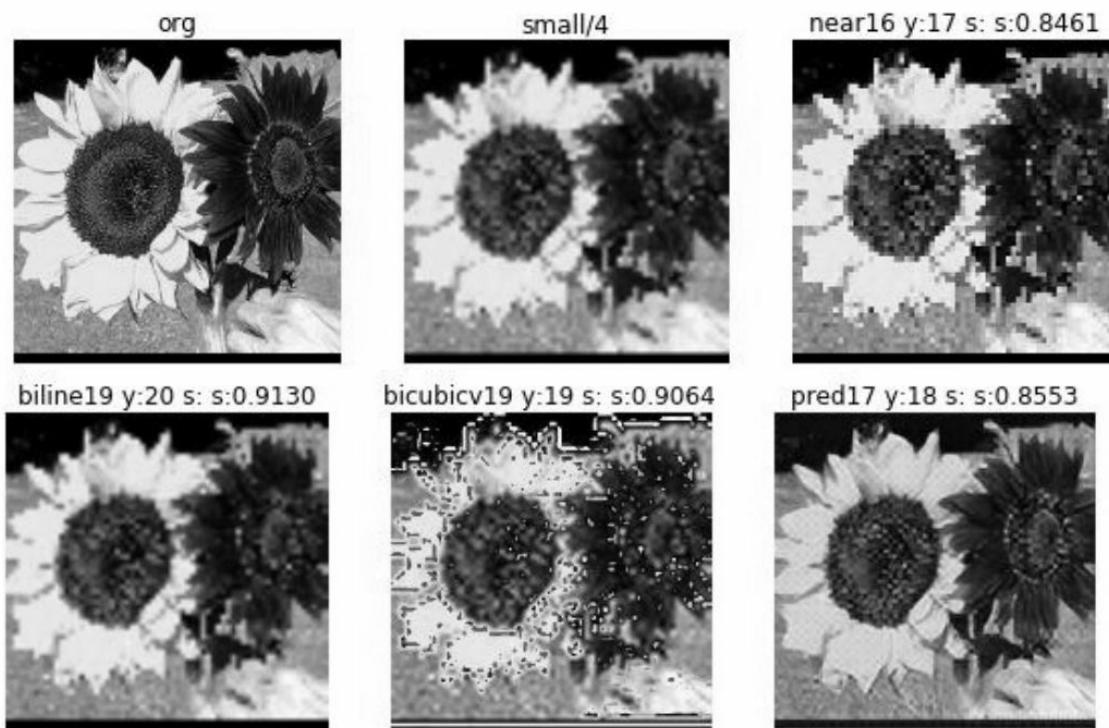


图12-17 SRGAN例子结果

图12-17中最后一张是模型生成的结果，每张图片上都有其评分值，可以看到SRGAN得到的PSNR和SSIM评价值不是最高的。但是我们肉眼看上去确实清晰了不少，并且通过有关机构对其进行MOS（Mean Opinion Score）的评价也表明，SRGAN生成的高分辨率图像看起来更真实。



注意：MOS (mean opinion score) 采用主观评定和技术评定相结合的方式。所谓主观评定就是有人为的参与，用人来评定。

该例中只演示了运行迭代1万多次的效果。如果将ResEpcn的模型与srGAN的模型分别加一个数量级，还会得到效果更优质的图片，有兴趣的读者可以自行尝试。

12.9 GAN网络的高级接口TFGAN

TFGAN是一个训练和评估生成式对抗网络(GAN)的轻量级库。它的初衷是为了让基于GAN的实验更加容易。

TFGAN也是基于估算器开发的一种应用接口，使用GANEstimator类来进行模型训练的。在TFGAN中，会使用很多已经集成的技巧(tricks)来稳定和提升GAN网络的训练效果。同时也集成了对GAN训练步骤的监视和可视化操作，以及训练后的模型评估操作，为开发者节省了大量的编码和调参时间。

TFGAN接口为开发者规范了开发GAN网络模型的标准步骤，每一个步骤都提供了全面的组建封装，使得开发者在开发GAN网络时，就像拼积木一样，按步骤选择不同的组建拼接起来即可。

TFGAN接口中规范的GAN网络开发步骤如下：

- (1) 指定网络的输入。
- (2) 使用GANModel函数来设置生成器和判别器模型。

- (3) 使用GANLoss函数来指定loss值。
- (4) 使用GANTrainOps函数来创建训练操作。
- (5) 运行训练操作。

当然开发者也可以将TFGAN中已经实现了的损失值和惩罚处理（包括推土机距离损失、梯度惩罚、互信息惩罚等），集成到原生的GAN网络或是其他框架中。

更多关于TFGAN的信息见如下链接：

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

该链接中包含的不仅仅是TFGAN的介绍，更多的是关于TFGAN实现各种GAN网络的例子代码。

在本书中，给出的均属于原生的GAN网络实例。目的是让读者掌握更底层的原理，以便于能够驾驭更有挑战性的任务。在实际应用中，使用TFGAN高级接口，可以起到事半功倍的效果，因此强烈推荐读者使用TFGAN高级接口。

12.10 总结

GAN网络可以说是2017年深度学习领域最热门的技术，从图12-18中可以看出2017年起GAN的种类已经由40多种发展到了250多种，而且速度越来越快。

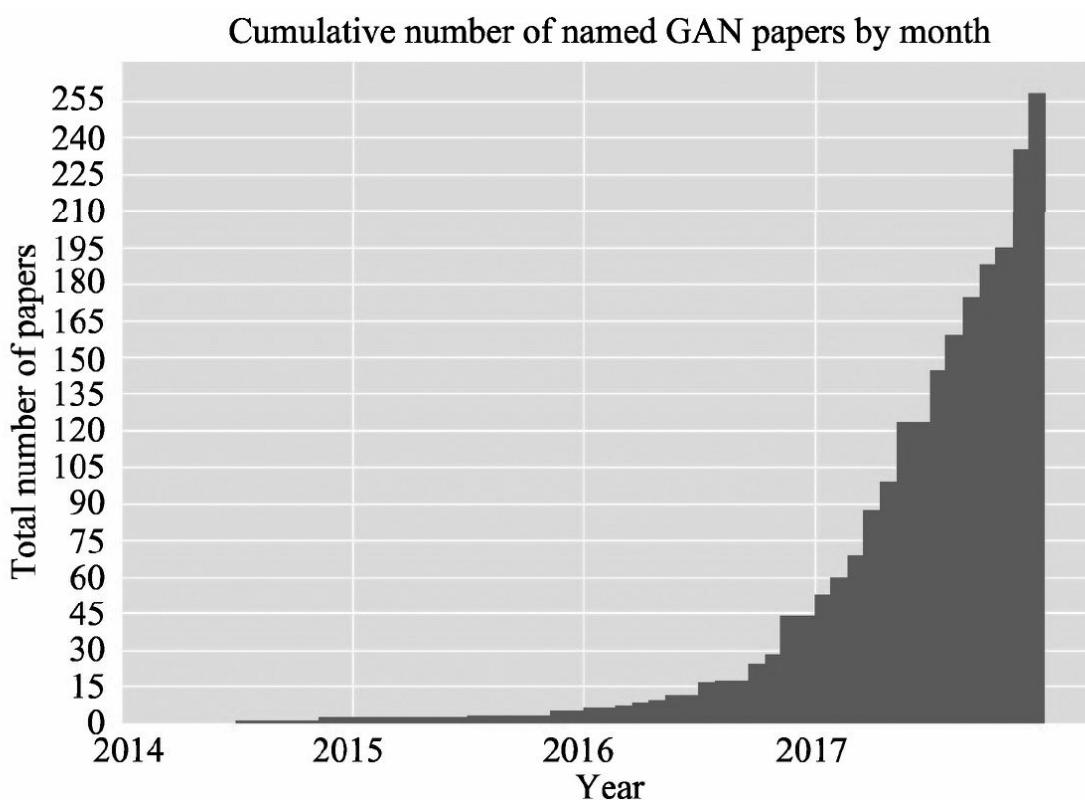


图12-18 GAN种类的发展

让网络来监督网络，使得深度学习在人工智能方向大踏步前进，由于篇幅所限，本书关于GAN网络的介绍只是冰山一角，GAN还可以实现通过文字描述生成图片；将图片生成文字，进行图像风格的转移；为人脸生成带眼镜的照片，或

将戴眼镜的照片还原成没带眼镜的照片；编写小说、诗歌等。另外，GAN在通信加密、文本分类等领域也广泛应用。在TensorFlow的官方GitHub中甚至还有使用GAN生成技术来扩充样本的工程。希望读者在结束本书的学习后不要停止脚步，人工智能的领域永无止尽，让我们一起努力用科学改变世界。