

```

47      batch_size, maxlen = tf.shape(caps_uhat)[0], tf.shape(caps_uhat)[1] # 获取批次和长度
48      out_caps_num = int(caps_uhat.get_shape()[2])
49      seqLen = tf.where(tf.equal(seqLen, 0), tf.ones_like(seqLen), seqLen)
50      mask = mkMask(seqLen, maxlen)      #mask 的形状为 (batch_size, maxlen)
51      floatmask = tf.cast(tf.expand_dims(mask, axis=-1), dtype=tf.float32) # 形状: (batch_size, maxlen, 1)
52
53      #B 的形状为 (batch_size, maxlen, out_caps_num)
54      B = tf.zeros([batch_size, maxlen, out_caps_num], dtype=tf.float32)
55      for i in range(iter_num):
56          C = tf.nn.softmax(B, axis=2) #形状: (batch_size, maxlen, out_caps_num)
57          C = tf.expand_dims(C*floatmask, axis=-1) #形状: (batch_size, maxlen, out_caps_num, 1)
58          weighted_uhat = C * caps_uhat #形状: (batch_size, maxlen, out_caps_num, out_caps_dim)
59          #S 的形状为 (batch_size, out_caps_num, out_caps_dim)
60          S = tf.reduce_sum(weighted_uhat, axis=1)
61
62          V = _squash(S, axes=[2])#shape(batch_size, out_caps_num, out_caps_dim)
63          V = tf.expand_dims(V, axis=1)#shape(batch_size, 1, out_caps_num, out_caps_dim)
64          B = tf.reduce_sum(caps_uhat * V, axis=-1)+ B #shape(batch_size, maxlen, out_caps_num)
65
66      V_ret = tf.squeeze(V, axis=[1])#shape(batch_size, out_caps_num, out_caps_dim)
67      S_ret = S
68      return V_ret, S_ret
69
70 def _squash(in_caps, axes):#定义激活函数
71     _EPSILON = 1e-9
72     vec_squared_norm = tf.reduce_sum(tf.square(in_caps), axis=axes, keepdims=True)
73     scalar_factor = vec_squared_norm / (1 + vec_squared_norm) / tf.sqrt(vec_squared_norm + _EPSILON)
74     vec_squashed = scalar_factor * in_caps
75     return vec_squashed
76
77 #定义函数, 用动态路由聚合 RNN 模型的结果信息
78 def routing_masked(in_x, xLen, out_caps_dim, out_caps_num, iter_num=3,
79                     dropout=None, is_train=False, scope=None):
80     assert len(in_x.get_shape()) == 3 and in_x.get_shape()[-1].value is not None
81     b_sz = tf.shape(in_x)[0]
82     with tf.variable_scope(scope or 'routing'):

```

```

83 caps_uhat = shared_routing_uhat(in_x, out_caps_num, out_caps_dim,
84     scope='rnn_caps_uhat')
85 attn_ctx, S = masked_routing_iter(caps_uhat, xLen, iter_num)
86 if dropout is not None:
87     attn_ctx = tf.layers.dropout(attn_ctx, rate=dropout,
88         training=is_train)
89 return attn_ctx

```

9.6.5 代码实现：用 IndyLSTM 单元搭建 RNN 模型

编写代码，实现如下逻辑。

- (1) 将 3 层 IndyLSTM 单元传入 `tf.nn.dynamic_rnn` 函数中，搭建动态 RNN 模型。
- (2) 用函数 `routing_masked` 对 RNN 模型的输出结果做基于动态路由的信息聚合。
- (3) 将聚合后的结果输入全连接网络，进行分类处理。
- (4) 用分类后的结果计算损失值，并定义优化器用于训练。

具体代码如下。

代码 9-7 用带有动态路由算法的 RNN 模型对新闻进行分类（续）

```

89 x = tf.placeholder("float", [None, maxlen])      # 定义输入占位符
90 x_len = tf.placeholder(tf.int32, [None, ])        # 定义输入序列长度占位符
91 y = tf.placeholder(tf.int32, [None, ])            # 定义输入分类标签占位符
92 nb_features = 128                                # 词嵌入维度
93 embeddings = tf.keras.layers.Embedding(num_words, nb_features)(x)
94
95 # 定义带有 IndyLSTMCell 的 RNN 模型的
96 hidden = [100, 50, 30]                            # RNN 模型的单元个数
97 stacked_rnn = []
98 for i in range(3):
99     cell = tf.contrib.rnn.IndyLSTMCell(hidden[i])
100    stacked_rnn.append(tf.nn.rnn_cell.DropoutWrapper(cell,
101        output_keep_prob=0.8))
102 mcell = tf.nn.rnn_cell.MultiRNNCell(stacked_rnn)
103
104 rnnoutputs, _ = tf.nn.dynamic_rnn(mcell, embeddings, dtype=tf.float32)
105 out_caps_num = 5                                  # 定义输出的胶囊个数
106 n_classes = 46                                   # 分类个数
107 outputs = routing_masked(rnnoutputs, x_len, int(rnnoutputs.get_shape()[-1]),
108     out_caps_num, iter_num=3)
109 pred = tf.layers.dense(outputs, n_classes, activation = tf.nn.relu)
110
111 learning_rate = 0.001

```

```

112 cost = tf.reduce_mean(tf.losses.sparse_softmax_cross_entropy(logits=pred,
    labels=y))
113 optimizer =
    tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

```

9.6.6 代码实现：建立会话，训练网络

用 `tf.data` 数据集接口的 `Iterator.from_structure` 方法获取迭代器，并按照数据集的遍历次数训练模型。具体代码如下。

代码 9-7 用带有动态路由算法的 RNN 模型对新闻进行分类（续）

```

114 iterator1 =
    tf.data.Iterator.from_structure(dataset.output_types, dataset.output_shapes)
115 one_element1 = iterator1.get_next() # 获取一个元素
116
117 with tf.Session() as sess:
118     sess.run(iterator1.make_initializer(dataset)) # 初始化迭代器
119     sess.run(tf.global_variables_initializer())
120     EPOCHS = 20 # 整个数据集迭代训练 20 次
121     for ii in range(EPOCHS):
122         alloss = [] # 数据集迭代两次
123         while True:
124             try:
125                 inp, target = sess.run(one_element1)
126                 _, loss = sess.run([optimizer, cost], feed_dict={x:
127                     inp[0], x_len:inp[1], y: target})
128                 alloss.append(loss)
129             except tf.errors.OutOfRangeError:
130                 print("step", ii+1, ": loss=", np.mean(alloss))
131                 sess.run(iterator1.make_initializer(dataset)) # 从头再来一遍
132                 break

```

代码运行后，输出如下内容：

```

step 1 : loss= 3.4340985
step 2 : loss= 2.349189
.....
step 19 : loss= 0.69928074
step 20 : loss= 0.65264946

```

结果显示，迭代 20 次之后的 loss 值约为 0.65。使用动态路由算法，会使模型训练时的收敛速度变得相对较慢。随着迭代次数的增加，模型的精度还会提高。

9.6.7 扩展：用分级网络将文章（长文本数据）分类

对于文章（长文本数据）的分类问题，可以将其样本的数据结构理解为含有多个句子，每个句子又含有多个词。本实例用“RNN 模型+动态路由算法”结构对序列词的语义进行处理，从而得到单个句子的语义。

在得到单个句子的语义之后，可以再次用“RNN 模型+动态路由算法”结构，对序列句子的语义进行处理，得到整个文章的语义，如图 9-14 所示。

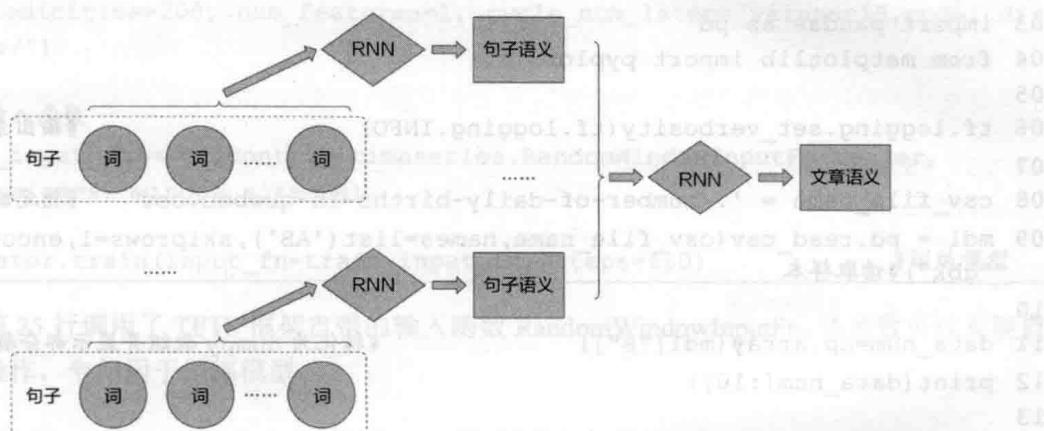


图 9-14 长文本分类结构

如图 9-14 所示，通过连续两个“RNN 模型+动态路由算法”结构，就可以实现长文本的分类功能。有兴趣的读者可以自行尝试一下。

9.7 实例 52：用 TFTS 框架预测某地区每天的出生人数

实例描述

现有记录着某地区从 1979 年 1 月 1 日至 1990 年 12 月 31 日每天出生人数的历史数据。要求：训练模型进行拟合，从而预测出未来指定天数内每天出生的人数。

9.7.1 准备样本

本实例使用的样本是某地区从 1979 年 1 月 1 日至 1990 年 12 月 31 日，每天的出生人数。样本的来源见以下地址：

<https://datamarket.com/data/set/235j/number-of-daily-births-in-quebec-jan-01-1977-to-dec-31-1990#!ds=235j&display=line>

9.7.2 代码实现：数据预处理——制作 TFTS 框架中的读取器

TFTS 框架支持 3 种创建读取器（Reader）的方式：

- 从 Numpy 数组中创建读取器。
- 从 TFRecords 文件中创建读取器。
- 从 CSV 文件中创建读取器。

本实例用 `tf.contrib.timeseries.NumpyReader` 函数和 Numpy 数组创建读取器。具体代码如下。

代码 9-8 时间序列问题

```

01 import numpy as np
02 import tensorflow as tf
03 import pandas as pd
04 from matplotlib import pyplot
05
06 tf.logging.set_verbosity(tf.logging.INFO)          #输出系统日志
07
08 csv_file_name = './number-of-daily-births-in-quebec.csv'    #指定样本文件
09 md1 = pd.read_csv(csv_file_name, names=list('AB'), skiprows=1, encoding =
"gbk")#读取样本
10
11 data_num=np.array(md1["B"])                      #转化为 numpy 数组并显示部分数据
12 print(data_num[:10])
13
14 x = np.array(range(len(data_num)))               #设置序列
15 data = {
16     tf.contrib.timeseries.TrainEvalFeatures.TIMES: x,
17     tf.contrib.timeseries.TrainEvalFeatures.VALUES: data_num,
18 }
19
20 reader = tf.contrib.timeseries.NumpyReader(data)      #创建 reader

```

9.7.3 代码实现：用 TFTS 框架定义模型，并进行训练

本实例用 TFTS 框架的内置函数 `StructuralEnsembleRegressor` 进行数据的拟合。该函数与估算器的用法类似。

- 在定义时，需要传入一个输入函数并指定若干参数。
- 在训练时，直接调用估算器的 `train` 方法即可。

函数 `StructuralEnsembleRegressor` 实现了一个结构化的回归模型。在使用时，该函数的常用参数如下。

- `periodicities`: 指定数据的拟合周期。
- `num_features`: 输入样本的维度。
- `cycle_num_latent_values`: 参与运算的潜在变量序列个数。其值越大，则运行得越慢，精度越高。
- `model_dir`: 模型保存路径。

**提示：**

有关该函数的更多参数，请参见代码中函数 StructuralEnsembleRegressor 的定义。

因为 TFTS 框架是估算器的一种具体实现，所以也支持估算器的参数设置（可以通过配置类 tf.contrib.learn.RunConfig 为估算器指定训练参数）。具体代码如下。

代码 9-8 时间序列问题（续）

```

21 estimator = tf.contrib.timeseries.StructuralEnsembleRegressor( # 定义模型
22   periodicities=200, num_features=1, cycle_num_latent_values=15, model_dir
23   ="mode/")
24 # 定义输入函数
25 train_input_fn = tf.contrib.timeseries.RandomWindowInputFn(reader,
26   batch_size=4, window_size=64) # 读取数据
27 estimator.train(input_fn=train_input_fn, steps=600) # 训练模型

```

代码第 25 行调用了 TFTS 框架自带的输入函数 RandomWindowInputFn。该函数可以实现自动乱序的操作，专门用于训练模型。

9.7.4 代码实现：用 TFTS 框架评估模型

TFTS 框架的评估方法与估算器的评估方法一致，都是使用 estimator.evaluate 方法进行的。具体代码如下。

代码 9-8 时间序列问题（续）

```

28 evaluation_input_fn = tf.contrib.timeseries.WholeDatasetInputFn(reader)
29 # 评估模型
30 evaluation = estimator.evaluate(input_fn=evaluation_input_fn, steps=1)
31 print(evaluation.keys()) # 打印评估结果
32 print(evaluation['loss']) # 打印评估结果中的 loss 值

```

代码第 28 行调用了 TFTS 框架的输入函数 WholeDatasetInputFn。该函数将指定的数据全部输入模型里，并且只输入一次，专门用于评估或预测模型。

TFTS 框架的评估结果中含有的信息量较大，具体见 9.7.6 小节的运行结果。

9.7.5 代码实现：用模型进行预测，并将结果可视化

TFTS 框架的预测方法与估算器框架的预测方法一致，都是使用 estimator.predict 方法。

这里调用输入函数 tf.contrib.timeseries.predict_continuation_input_fn 来设置模型预测时的输入数据和预测步数。该函数的作用是：在评估结果的基础上，让 estimator.predict 方法输出后续指定步数的预测值。具体代码如下。

代码 9-8 时间序列问题（续）

```

33 (predictions,) = tuple(estimator.predict() #预测模型
34     input_fn=tf.contrib.timeseries.predict_continuation_input_fn(
35         evaluation, steps=2))
36 print("predictions:", predictions)
37
38 times = evaluation["times"][0][-20:] #取后 20 个内容进行显示
39 observed = evaluation["observed"][0, :, 0][-20:] #获得原始数据 observed
40 mean = np.squeeze(np.concatenate(
41     [evaluation["mean"][0][-20:], predictions["mean"]], axis=0))
42 variance = np.squeeze(np.concatenate(
43     [evaluation["covariance"][0][-20:], predictions["covariance"]], axis=0))
44 all_times = np.concatenate([times, predictions["times"]], axis=0)
45 upper_limit = mean + np.sqrt(variance) #根据方差和均值，算出该序列的取值范围
46 lower_limit = mean - np.sqrt(variance)
47
48 #定义函数，可视化结果
49 def make_plot(name, training_times, observed, all_times, mean, upper_limit,
50   lower_limit):
51   pyplot.figure()
52   pyplot.plot(training_times, observed, "b", label="training series")
53   pyplot.plot(all_times, mean, "r", label="forecast")
54   pyplot.plot(all_times, upper_limit, "g", label="forecast upper bound")
55   pyplot.plot(all_times, lower_limit, "g", label="forecast lower bound")
56   pyplot.fill_between(all_times, lower_limit, upper_limit, color="grey",
57       alpha="0.2")
58   pyplot.axvline(training_times[-1], color="k", linestyle="--")
59   pyplot.xlabel("time")
60   pyplot.ylabel("observations")
61   pyplot.legend(loc=0)
62   pyplot.title(name)
63 make_plot("Structural ensemble", times, observed, all_times, mean, upper_limit,
64   lower_limit)

```

在模型评估和预测的过程中，会输出每个时间段的均值和方差。根据该均值和方差可以得到该值的分布区间。

代码第 49 行，用函数 make_plot 将整个数据及预测的数据区间一起显示出来。

9.7.6 运行程序

将代码运行后，输出结果如下。

(1) 输出评估结果。

```

dict_keys(['covariance', 'log_likelihood', 'loss', 'mean', 'observed',
'start_tuple', 'times', 'global_step'])

```

1.1670636

从输出结果可以看到，输出的评估结果是一个字典类型的数据。该字典中含有每个时刻的数据分布情况（covariance、log_likelihood、mean）、原始值（observed）、损失值（loss）及训练步数（global_step）。

(2) 输出预测结果。

predictions:

```
{'mean': array([[237.21950126], [246.91376098]]), 'covariance': array([[[[1054.86319966]], [[1302.53715562]]]]), 'times': array([5113, 5114], dtype=int64)}
```

输出预测结果是未来两天的出生人口数（取均值）是 237、247。结果中的 times 是输出的序列次数。

另外，程序也输出了可视化预测结果，如图 9-15 所示。

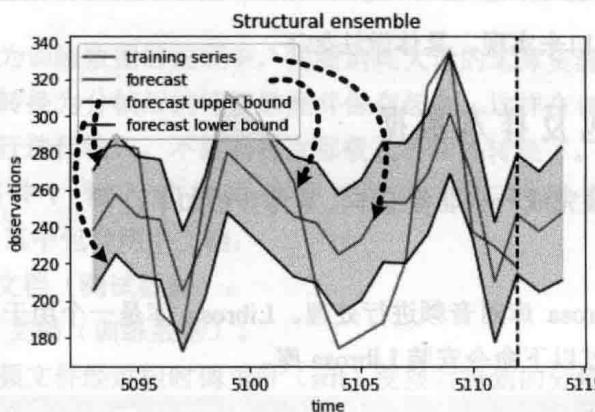


图 9-15 TFTS 框架中模型的预测结果



提示：

在 TFTS 框架中训练模型时，生成的日志信息较多。可以将代码第 6 行（`tf.logging.set_verbosity(tf.logging.INFO)`）注释掉，或设置额外的日志级别，以减少信息输出量。

9.7.7 扩展：用 TFTS 框架进行异常值检测

如图 9-15 所示，用 TFTS 框架中的模型可以预测出一个序列数据未来的分布空间。利用这个功能可以实现基于序列数据的异常值检测。

如果真实值在预测值范围之内，则认为是合理的值；否则就认为是异常的值。当然这只是个方向，在实际中还要配合很多技术来提升模型的精度。

例如：

- (1) 用 TFTS 框架训练模型，得出模型的预测范围与真实值之间的距离，通过后续模型对距离与分类的关系进行拟合。
- (2) 用 TFTS 框架定义模型，用于对多变量进行拟合。

(3) 在 TFTS 框架中用自定义的 RNN 模型进行更高精度的序列数据拟合等。

更多的例子可以参考如下链接：

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

9.8 实例 53：用 Tacotron 模型合成中文语音（TTS）

Tacotron 模型是谷歌公司推出的一个端到端的 TTS（语音合成）模型。该模型使用带有注意力机制的 Seq2Seq 框架。它所合成的语音效果，可以迟到以假乱真的效果。

实例描述

有一批音频数据和对应的文字及拼音文本。下面让 Tacotron 模型对其进行学习并拟合拼音与音频的对应关系，并根据具体的拼音输入获得对应的音频发音。

本实例用 `tf.keras` 接口来实现。具体做法如下。

9.8.1 准备安装包及样本数据

在项目开始前，需要完成一些准备工作，具体介绍如下。

1. 安装 librosa 库

在本实例中，用 `librosa` 库对音频进行处理。`Librosa` 库是一个用于音频分析和音乐分析的 Python 工具包。可以通过以下命令安装 `Librosa` 库。

```
pip install librosa
```

该工具包中封装了很多函数，可以实现音频处理、音频特征提取、绘图处理等功能。具体可以参考如下地址。

- 音频处理：<http://librosa.github.io/librosa/core.html>
- 音频特征提取：<http://librosa.github.io/librosa/feature.html>
- 绘图处理：<http://librosa.github.io/librosa/display.html>



提示：

安装完 `librosa` 库后，在代码中用“`import librosa`”语句进行导入时，有时会报如下错误：

`AttributeError: module 'numba' has no attribute 'jit'`

可以通过重新安装 `numba` 库进行解决，具体命令如下：

```
conda install numba
```

2. 部署样本数据

本实例采用的是清华大学发布的语料库 `data_thchs30`。

出于学习目的，这里只使部分语料进行训练。部署的方法如下所示。

- (1) 将数据解压缩。
 - (2) 在解压缩后的 data_thchs30\test 目录下，随意取出几个音频文件。
 - (3) 将选出的文件放到 mytest 目录里。
 - (4) 将 mytest 目录与 doc 目录一同放在 data_thchs30 目录下。
- 完整的目录结构如图 9-16 所示。

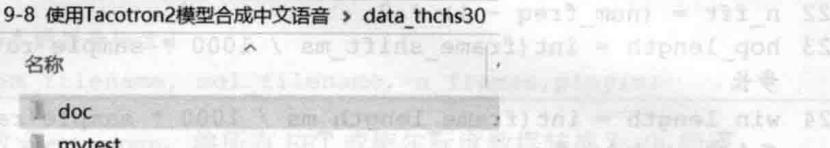


图 9-16 data_thchs30 目录结构

9.8.2 代码实现：将音频数据分帧并转为梅尔频谱

音频样本在被转换为训练数据的过程中，需要消耗大量的运算资源。所以有必要在样本预处理环节，将音频文件转换为分帧后的特征数据并保存起来。这样在训练模型的过程中，直接用转换好的音频数据进行迭代输入，不需要每次都载入再额外转换了。

在语音合成项目中，输入的样本是中文文字对应的拼音。该样本的路径是 data_thchs30/doc/trans，其中包含两个文档：

- test.syllable.txt 文档（测试数据）。
- train.syllable.txt 文档（训练数据）。

样本的标签是将音频文件经过短时傅里叶（stft）变换后得到的分帧数据与梅尔频率谱特征数据。

编写代码，将音频文件转换为分帧后的特征数字数据，并为其匹配对应的拼音文本。具体代码如下。

代码 9-9 样本预处理

```

01 import os
02 from multiprocessing import cpu_count
03 from tqdm import tqdm
04 from concurrent.futures import ProcessPoolExecutor #载入多进程库
05 from functools import partial
06 import numpy as np
07 import glob
08 from scipy import signal
09 import librosa
10
11 max_frame_num=1000 #定义每个音频文件的最大帧数
12 sample_rate=16000 #定义音频文件的采样率
13
14 num_freq=1025 #振幅频率
15 num_mels=80 #定义 Mel bands 特征的个数

```

```

16
17 frame_length_ms=50 #定义stft算法中的重叠窗口(用时间来表示)
18 frame_shift_ms=12.5 #定义stft算法中的移动步长(用时间来表示)
19
20 preemphasis=0.97#用于数字滤波器的阈值
21 #stft算法中使用的窗口(因为声音的真实频率只有正的,而fft变换是对称的,所以需要加上负
22 n_fft = (num_freq - 1) * 2
23 hop_length = int(frame_shift_ms / 1000 * sample_rate)#定义stft算法中的帧移
24 step长
25 win_length = int(frame_length_ms / 1000 * sample_rate)#定义stft算法中的相邻
26 两个窗口的重叠长度
27
28 ref_level_db=20 #控制峰值的阈值
29 min_level_db=-100 #指定dB最小值,用于归一化
30
31 #创建一个Mel filter, shape=(n_mels, 1 + n_fft/2), 即(n_mels, num_freq)
32 _mel_basis = librosa.filters.mel(sample_rate, n_fft, n_mels=num_mels)
33
34 def spectrogram(D):#定义函数,实现dB频谱转换
35     S = 20 * np.log10(np.maximum(1e-5, D)) - ref_level_db #转换为dB频谱
36     return np.clip((S - min_level_db) / -min_level_db, 0, 1)#归一化
37
38 def melspectrogram(D): #转换为mel特征
39     mel = np.dot(_mel_basis, D) #通过与矩阵点积计算,将分帧结果转换为mel特征
40     return spectrogram(mel)
41
42 def _process_utterance(out_dir, index, wav_path,pinyin):#样本预处理函数
43     #按照16000的采样率读取音频
44     wav, _ = librosa.core.load(wav_path, sr=sample_rate)
45     #对波形文件进行数字滤波处理
46     emphasis = signal.lfilter([1, -preemphasis], [1], wav)
47
48     #用短时傅里叶变换将音频分帧
49     D=np.abs(librosa.stft(emphasis, n_fft, hop_length, win_length))
50
51     #计算原始声音分帧后的时频图
52     linear_spectrogram = spectrogram(D).astype(np.float32)
53     n_frames = linear_spectrogram.shape[1] #返回帧的个数
54     if n_frames > max_frame_num: #如帧数过长,则直接舍去
55         return None
56
57     #计算原始声音分帧后的mel特征时频图
58     mel_spectrogram = melspectrogram(D).astype(np.float32)
59

```

```

60 #保存转换后的特征数据
61 spectrogram_filename = 'thchs30-spec-%05d.npy' % index
62 mel_filename = 'thchs30-mel-%05d.npy' % index
63 np.save(os.path.join(out_dir, spectrogram_filename),
64 linear_spectrogram.T, allow_pickle=False)
65 np.save(os.path.join(out_dir, mel_filename), mel_spectrogram.T,
66 allow_pickle=False)
67 #返回特征文件名(即样本的拼音标注)
68 return (spectrogram_filename, mel_filename, n_frames, pinyin)

```

代码第32行定义了函数 spectrogram，将所有 FFT 或梅尔标度数据转换为 dB 频谱。

dB 频谱是一个没有任何单位的比值。由于它在不同的领域（常见的领域有声音、信号、增益等）有着不同的名称，因此它也具有不同的实际意义，在实际使用时，也不会有固定的计算公式。在本实例中，dB 表示声音的大小（分贝）。

9.8.3 代码实现：用多进程预处理样本并保存结果

定义主处理函数 preprocess_data，并在其内部实现如下逻辑。

(1) 用多进程调用 process_utterance 函数进行批处理转换。

(2) 保存最终的结果。

具体代码如下。

代码 9-9 样本预处理（续）

```

68 #用多进程实现音频数据的转换
69 def build_from_path(in_dir, out_dir, num_workers=1, tqdm=lambda x: x):
70
71     executor = ProcessPoolExecutor(max_workers=num_workers) #创建进程池执行器
72     futures = []
73     index = 1
74     #获取指定目录下的文件
75     wav_files = glob.glob(os.path.join(in_dir, 'mytest', '*.wav'))
76
77     #读取标注文件
78     with open(os.path.join(in_dir, r'doc/trans', 'test.syllable.txt')) as f:
79         allpinyin = {}
80         for pinyin in f:
81             indexf = pinyin.index(' ')
82             allpinyin[pinyin[:indexf]] = pinyin[indexf+1:]
83
84     #将音频文件与标注关联在一起
85     for wav_file in wav_files:
86         key = wav_file[wav_file.index('D'):-4]
87         #定义进程任务，调用处理函数

```

```

88     task = partial(_process_utterance, out_dir, index,
89     wav_file, allpinyin[key])
90     futures.append(executor.submit(task))
91     index += 1
92 
93 def preprocess_data(num_workers): # 定义函数处理样本数据
94     # 指定样本路径
95     in_dir = os.path.join(os.path.expanduser('.'), 'data_thchs30')
96     # 指定输出路径
97     out_dir = os.path.join(os.path.expanduser('.'), 'training')
98     os.makedirs(out_dir, exist_ok=True)
99     # 处理数据
100    metadata = build_from_path(in_dir, out_dir, num_workers, tqdm=tqdm)
101    # 将结果保存起来
102    with open(os.path.join(out_dir, 'train.txt'), 'w', encoding='utf-8') as
103        f:
104            for m in metadata:
105                f.write('|\'.join([str(x) for x in m]))
106            frames = sum([m[2] for m in metadata])
107            print('Wrote %d utterances, %d frames' % (len(metadata), frames))
108            print('Max input length: %d' % max(len(m[3]) for m in metadata))
109            print('Max output length: %d' % max(m[2] for m in metadata))
110
111 def main():
112     preprocess_data(cpu_count())
113
114 if __name__ == "__main__": # 运行当前模块
115     main()

```

代码运行后，会在本地 training 文件夹下生成转换好的音频数据文件，以及汇总后的统计文件 train.txt。



提示：

代码第 114 行是必需的，否则会报错误。该代码是多进程程序，所以系统创建的新进程必须位于 “if __name__ == '__main__':” 之下。在 Windows 中保护代码的主循环非常重要，这种语法可以避免在使用 processpoolexecutor 或产生新进程的任何其他并行代码时递归生成子进程。

更多有关进程方面的知识可以参考《Python 带我起飞——入门、进阶、商业实战》一书的第 10 章。

9.8.4 拆分 Tacotron 网络模型的结构

Tacotron 网络模型使用带有注意力机制的 Seq2Seq 框架。它在 Seq2Seq 框架基础上又增加了一些例如 CBHG 网络模型、残差 RNN 模型之类的细节技术。

1. Tacotron 网络模型的主体结构

Tacotron 网络模型的主体结构如下。

- 编码器用 CBHG 网络模型增加其泛化能力。
- 注意力机制的实现是在原有 BahdanauAttention 注意力接口基础上的二次封装，实现了基于内容和位置的混合注意力机制。
- 解码器使用两层带有残差的多层 RNN 模型。其中的 cell 使用的是 GRU 单元。
- 在合成音频之前对解码器的输出结果同样做了一次 CBHG 网络模型的变化，将其转换为 linear 音频特征。

对应的结构图如 9-17 所示。

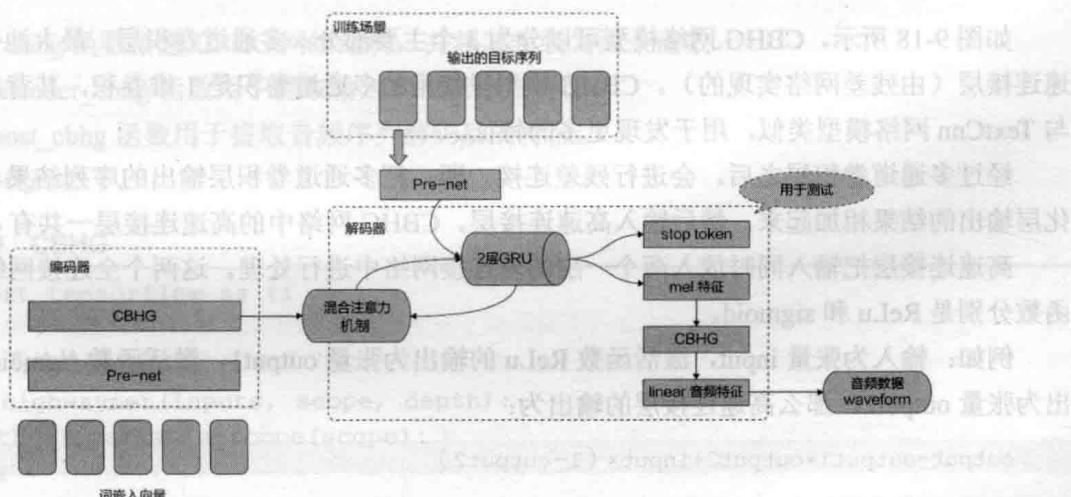


图 9-17 Tacotron 网络模型的主体结构

在图 9-17 中包含了 3 个主要的子结构：Pre-net、CBHG、混合注意力机制。其中，Pre-net 代表预处理层，对所有输入的原始数据做了两层的全连接转换，使其变化到指定的维度。另外两个子结构将在下面重点介绍。

以上结构来自 Tacotron 与 Tacotron-2 两个结构，更多内容可以参考以下两篇论文：

<https://arxiv.org/pdf/1703.10135.pdf>
<https://arxiv.org/pdf/1712.05884.pdf>

2. 介绍 CBHG 网络的结构

CBHG 网络模型常用在 NLP 任务中。其擅长提取序列字符的内在特征，在这里主要被用于提高网络的泛化效果。CBHG 网络模型的结构如图 9-18 所示。

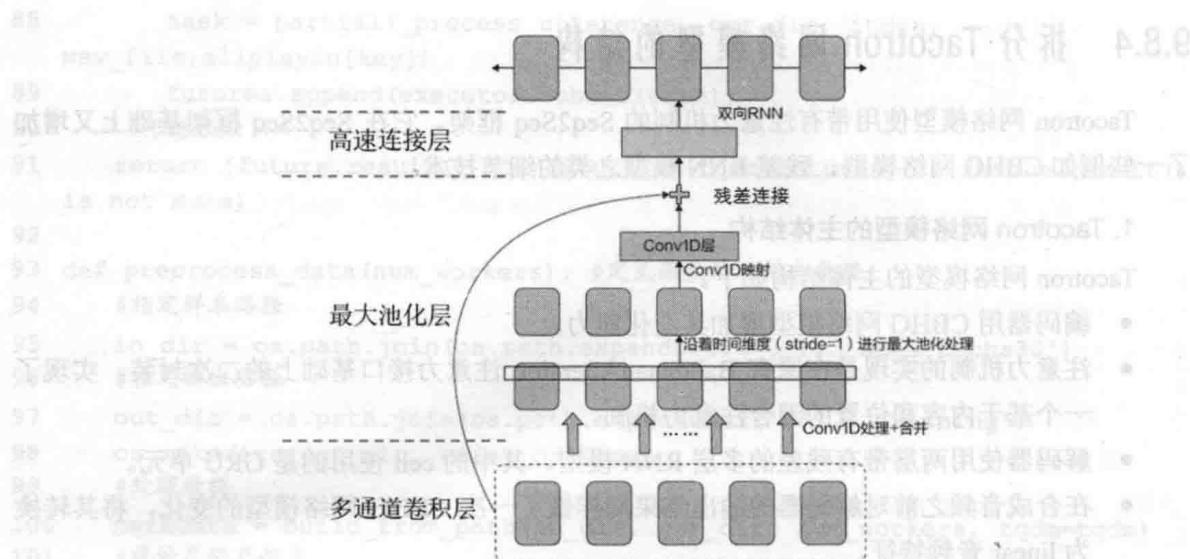


图 9-18 CBHG 网络模型的主体结构

如图 9-18 所示, CBHG 网络模型可以分为 3 个主要部分: 多通道卷积层、最大池化层、高速连接层(由残差网络实现的)。CBHG 模型里使用的多通道卷积是 1 维卷积, 其背后的理论与 TextCnn 网络模型类似, 用于发现更多的特征。

经过多通道卷积层之后, 会进行残差连接。即: 把多通道卷积层输出的序列结果与最大池化层输出的结果相加起来, 然后输入高速连接层。CBHG 网络中的高速连接层一共有 4 层。

高速连接层把输入同时放入两个一层的全连接网络中进行处理。这两个全连接网络的激活函数分别是 ReLu 和 sigmoid。

例如: 输入为张量 `input`, 激活函数 `ReLU` 的输出为张量 `output1`, 激活函数 `sigmoid` 的输出为张量 `output2`, 那么高速连接层的输出为:

```
output=output1×output2+input×(1-output2)
```

将序列中的每个样本变换之后, 再输入双向 RNN 模型中进行全序列的特征提取, 这样就完成了 CBHG 网络模型的全部过程。

3. 混合注意力机制的实现方法

在 9.1.12 小节, 介绍过混合注意力机制的原理。但是在 TensorFlow 的当前版本里并没有带混合注意力机制的模型接口, 所以需要手动实现。

在实现混合注意力机制时, 可以参考实现 BahdanauAttention 注意力机制的源代码。按照 9.1.12 小节进行修改即可。

在 BahdanauAttention 注意力机制的实现过程中, 主要包含了一个 BahdanauAttention 类与一个 `_bahdanau_score` 函数:

- BahdanauAttention 类 Bahdanau 注意力机制的主体实现, 在该类中实现了初始化方法 `__init__` 和调用方法 `__call__`。
- `_bahdanau_score` 函数用来计算经过全连接转换后的输入与中间状态结果的最终分值。

对应于 9.1.10 小节中“4. 在 TensorFlow 中的具体实现”里面的第(5)、(6)、(7)步。

混合注意力机制也是通过一个类（`LocationSensitiveAttention`）和一个函数（`_location_sensitive_score`）实现的：

- 在 `LocationSensitiveAttention` 类的 `_call_` 方法中，对上一次的注意力结果做了一次卷积与一次全连接，并与原始 `BahdanauAttention` 类中的成员变量 `query` 及成员变量 `key` 一起被送入函数 `_location_sensitive_score` 中，进行计算分数。
 - 在函数 `_location_sensitive_score` 中，除使用权重 `v` 进行相乘外，又加入偏置 `b` 的权重。
- 代码如下：

```
tf.reduce_sum(v_a * tf.tanh(keys + processed_query + processed_location + b_a), [2])
```

更详细的介绍请参考 9.8.6 小节。

9.8.5 代码实现：搭建 CBHG 网络

编写 `cbhg` 函数搭建 CBHG 网络结构，并将其封装为两个函数。

- `encoder_cbhg` 函数用于提取输入的拼音序列特征。
- `post_cbhg` 函数用于提取音频序列解码后的特征。

具体代码如下。

代码 9-10 CBHG

```
01 import tensorflow as tf
02
03 # 定义高速连接函数
04 def highwaynet(inputs, scope, depth):
05     with tf.variable_scope(scope):
06         H = tf.keras.layers.Dense(units=depth, activation='relu', name='H')(inputs)
07         T = tf.keras.layers.Dense(units=depth, activation='sigmoid', name='T',
08                               bias_initializer=tf.constant_initializer(-1.0))
09         return H * T + inputs * (1.0 - T)
10
11 def cbhg(inputs, input_lengths, is_training, scope, K, projections, depth):
12     with tf.variable_scope(scope):
13         with tf.variable_scope('conv_bank'): # 多通道卷积
14             conv_bank = []
15             for k in range(1, K+1): # 使用 same 卷积。结果的尺度与卷积核无关，与步长有关
16                 convd_output =
17                     tf.keras.layers.Conv1D(128, k, activation=tf.nn.relu,
18                                           padding='same', name = 'conv1d_%d'%k)(inputs)
18                 convd_output_bn = tf.keras.layers.BatchNormalization(
```

```

19      k) ( conv1d_output, training=is_training)
20          conv_bank.append(conv1d_output_bn)
21      conv_outputs = tf.concat(conv_bank, axis=-1)
22
23      #最大池化层
24      maxpool_output = tf.keras.layers.MaxPool1D(
25          pool_size=2, strides=1, padding='same')(conv_outputs)
26
27      #用两层卷积进行维度变换
28      proj1_output =
29          tf.keras.layers.Conv1D(projections[0], 3, activation=tf.nn.relu,
30          padding='same', name =
31          'proj_1')(maxpool_output)
32      proj1_output_bn = tf.keras.layers.BatchNormalization(name =
33          'proj_1_bn')(proj1_output, training=is_training)           #卷积后的BN处理
34      #第2层卷积
35      proj2_output = tf.keras.layers.Conv1D(projections[1], 3,
36          padding='same', name = 'proj_2')(proj1_output_bn)
37      #卷积后的BN处理
38      proj2_output_bn = tf.keras.layers.BatchNormalization(name =
39          'proj_2_bn')(proj2_output, training=is_training)
40
41      #残差连接
42      highway_input = proj2_output_bn + inputs
43
44      half_depth = depth // 2 #必须能被2整除，之后的结果是每个方向RNN的cell个数
45      assert half_depth*2 == depth, 'depth必须被2整除。'
46
47      #调整残差后的维度，与RNN的cell个数一致
48      if highway_input.shape[2] != half_depth:
49          highway_input = tf.keras.layers.Dense(half_depth)(highway_input)
50
51      #4层高速连接
52      for i in range(4):
53          highway_input = highwaynet(highway_input, 'highway_%d' % (i+1),
54          half_depth)
55
56      rnn_input = highway_input
57
58      #双向RNN
59      outputs, states =
60          tf.nn.bidirectional_dynamic_rnn(tf.keras.layers.GRUCell(half_depth),
61          tf.keras.layers.GRUCell(half_depth),
62          rnn_input,
63          sequence_length=input_lengths, dtype=tf.float32)
64
65      return tf.concat(outputs, axis=2) #将双向RNN正反方向结果组合到一起

```

```

55 #用于编码器中的 CBHG
56 def encoder_cbhg(inputs, input_lengths, is_training, depth):#depth 是 RNN
    单元个数，由于是双向的，所以它必须能被 2 整除
57     return cbhg(inputs, input_lengths, is_training, scope='encoder_cbhg',
    K=16,
58     projections=[128, inputs.shape.as_list()[2]], depth=depth)
59
60 #用于解码器中的 CBHG
61 def post_cbhg(inputs, input_dim, is_training, depth):
62     return cbhg(inputs, None, is_training, scope='post_cbhg',
    K=8, projections=[256, input_dim], depth=depth)

```

9.8.6 代码实现：构建带有混合注意力机制的模块

参考 9.1.12 小节与 9.8.4 小节的描述，实现混合注意力机制。

具体代码如下。

代码 9-11 attention

```

01 import tensorflow as tf
02 from tensorflow.contrib.seq2seq.python.ops.attention_wrapper import
    BahdanauAttention
03 from tensorflow.python.ops import array_ops, variable_scope
04
05 def _location_sensitive_score(processed_query, processed_location, keys):
06     #获取注意力的深度（全连接神经元的个数）
07     dtype = processed_query.dtype
08     num_units = keys.shape[-1].value or array_ops.shape(keys)[-1]
09
10     #定义了最后一个全连接v
11     v_a = tf.get_variable('attention_variable', shape=[num_units],
    dtype=dtype,
12     initializer=tf.contrib.layers.xavier_initializer())
13
14     #定义偏置b
15     b_a = tf.get_variable('attention_bias', shape=[num_units], dtype=dtype,
16     initializer=tf.zeros_initializer())
17     #计算注意力分数
18     return tf.reduce_sum(v_a * tf.tanh(keys + processed_query +
    processed_location + b_a), [2])
19 #平滑归一化函数，返回[batch_size, max_time]，代替softmax
20 def _smoothing_normalization(e):
21     return tf.nn.sigmoid(e) / tf.reduce_sum(tf.nn.sigmoid(e), axis=-1,
    keepdims=True)
22
23 class LocationSensitiveAttention(BahdanauAttention): #定义位置敏感注意力机
    制类

```

```

24 def __init__(self,
25     num_units, #初始化
26     memory, #实现过程中全连接的神经元个数
27     smoothing=False, #编码器（Encoder）的结果
28     cumulate_weights=True, #是否使用平滑归一化函数代替 softmax
29     name='LocationSensitiveAttention'):
30
31     #如果 smoothing 为 true, 则使用_smoothing_normalization, 否则使用 softmax
32     normalization_function = _smoothing_normalization if (smoothing == True)
33     else None
34     super(LocationSensitiveAttention, self).__init__(
35         num_units=num_units, memory=memory,
36         memory_sequence_length=None,
37         probability_fn=normalization_function,
38         name=name) #如果 probability_fn 为 None, 则基类会调用 softmax
39
40     self.location_convolution = tf.layers.Conv1D(filters=32,
41         kernel_size=(31, ), padding='same', use_bias=True,
42         bias_initializer=tf.zeros_initializer(),
43         name='location_features_convolution')
44
45     self.location_layer = tf.layers.Dense(units=num_units, use_bias=False,
46         dtype=tf.float32, name='location_features_layer')
47     self._cumulate = cumulate_weights
48
49     def __call__(self, query, #解码器中间态结果 [batch_size, query_depth]
50                 state): #上一次的注意力 [batch_size, alignments_size]
51         with variable_scope.variable_scope(None,
52 "Location_Sensitive_Attention", [query]):
53
54             #全连接处理 query 特征 [batch_size, query_depth] -> [batch_size,
55             attention_dim]
56             processed_query = self.query_layer(query) if self.query_layer else
57             query
58             #维度扩展 -> [batch_size, 1, attention_dim]
59             processed_query = tf.expand_dims(processed_query, 1)
60
61             #维度扩展 [batch_size, max_time] -> [batch_size, max_time, 1]
62             expanded_alignments = tf.expand_dims(state, axis=2)
63             #通过卷积获取位置特征 [batch_size, max_time, filters]
64             f = self.location_convolution(expanded_alignments)
65             #经过全连接变化 [batch_size, max_time, attention_dim]
66             processed_location_features = self.location_layer(f)
67
68             #计算注意力的分数 [batch_size, max_time]
69             energy = _location_sensitive_score(processed_query,
70             processed_location_features, self.keys)
71
72
73

```

```

64     #计算最终的注意力结果 [batch_size, max_time], 11-07-2017 10:57:50
65     alignments = self._probability_fn(energy, state) 11-07-2017 10:57:50
66
67     #是否需要将返回累加后的注意力作为状态值 11-07-2017 10:57:50
68     if self._cumulate: 11-07-2017 10:57:50
69         next_state = alignments + state 11-07-2017 10:57:50
70     else: 11-07-2017 10:57:50
71         next_state = alignments 11-07-2017 10:57:50
72
73     return alignments, next_state 11-07-2017 10:57:50

```

代码第 28 行通过参数 `cumulate_weights` 来决定是否使用累加注意力功能。

如果参数 `cumulate_weights` 为 `True`，则表示使用累加注意力功能。程序会将解码器每次计算的注意力累加起来，作为下一次计算注意力时的状态值。

9.8.7 代码实现：构建自定义 wrapper

注意力机制需要与 `wrapper` 函数一起使用。但是本实例的解码器在注意力机制的前后，还需要做一些其他的操作。因为 TensorFlow 中原有的 `wrapper` 函数无法满足要求，所以需要自定义一个 `TacotronDecoderwrapper` 函数。

在 `TacotronDecoderwrapper` 函数中，具体需要实现以下几个步骤。

- (1) 对输入的真实值 y ，做两层全连接的预处理变换。
- (2) 加入注意力分值，然后再做一次全连接。将混合后的特征输入解码器的 RNN 模型中。
- (3) 对输出的结果做混合注意力计算。
- (4) 将得到的注意力与解码器的 RNN 模型结果连接，作为最终的解码特征。
- (5) 对解码特征做一个全连接，生成下一个序列的音频 mel 特征。
- (6) 对解码特征做一个全连接，生成下一个序列结束符号。

由于多个连续音频帧代表一个音素（拼音中的一个发音），并且代表一个音素的多个连续音频帧的 mel 特征值一般都不会差别太大，所以在解码时，可以对音频帧进行分段处理（不需要对所有的音频帧进行逐个处理）。这种方式可以提升整个模型的处理性能。

具体实现步骤如下。

- (1) 设置一个参数（见如下代码第 12 行中的 `outputs_per_step` 参数）。
- (2) 按照该参数的步长从目标音频帧中采样。
- (3) 将采样的结果送入解码器中。
- (4) 解码器会按照参数的值生成下一时刻的音频帧数。

具体代码如下。

代码 9-12 TacotronDecoderwrapper

```

01 import tensorflow as tf
02 from tensorflow.python.framework import ops, tensor_shape
03 from tensorflow.python.ops import array_ops, check_ops, rnn_cell_impl,
04     tensor_array_ops

```

```

04 from tensorflow.python.util import nest
05 from tensorflow.contrib.seq2seq.python.ops import attention_wrapper
06
07 attention = __import__("9-11_attention")
08 LocationSensitiveAttention = attention.LocationSensitiveAttention
09
10 class TacotronDecoderwrapper(tf.nn.rnn_cell.RNNCell):
11     #初始化
12     def __init__(self, encoder_outputs, is_training, rnn_cell, num_mels,
13      outputs_per_step):
14         super(TacotronDecoderwrapper, self).__init__()
15
16         self._training = is_training
17         self._attention_mechanism = LocationSensitiveAttention(256,
18 encoder_outputs)#[N, T_in, attention_depth=256]
19         self._cell = rnn_cell
20         self._frame_projection = tf.keras.layers.Dense(units=num_mels *
21 outputs_per_step, name='projection_frame')#形状为[N, T_out/r, M*r]
22         self._stop_projection = tf.keras.layers.Dense(units=outputs_per_step,
23 name='projection_stop')
24         self._attention_layer_size =
25         self._attention_mechanism.values.get_shape()[-1].value
26
27     def _batch_size_checks(self, batch_size, error_message):
28         return [check_ops.assert_equal(batch_size,
29             self._attention_mechanism.batch_size,
30             message=error_message)]
31
32     @property
33     def output_size(self):
34         return self._output_size
35
36     def state_size(self):#返回的状态大小
37         return tf.contrib.seq2seq.AttentionWrapperState(
38             cell_state=self._cell._cell.state_size,
39             time=tensor_shape.TensorShape([]),
40             attention=self._attention_layer_size,
41             alignments=self._attention_mechanism.alignments_size,
42             alignment_history=(), attention_state = ())
43
44     def zero_state(self, batch_size, dtype):#返回一个0状态

```

```

44     with ops.name_scope(type(self).__name__ + "ZeroState",
45         values=[batch_size]):
46         cell_state = self._cell.zero_state(batch_size, dtype)
47         error_message = (
48             "When calling zero_state of TacotronDecoderCell %s: " %
49             self._base_name +
50             "Non-matching batch sizes between the memory "
51             "(encoder output) and the requested batch size.")
52         with ops.control_dependencies(
53             self._batch_size_checks(batch_size, error_message)):
54             cell_state = nest.map_structure(
55                 lambda s: array_ops.identity(s, name="checked_cell_state"),
56                 cell_state)
57
58     return tf.contrib.seq2seq.AttentionWrapperState(
59         cell_state=cell_state,
60         time=array_ops.zeros([], dtype=tf.int32),
61         attention=rnn_cell_impl._zero_state_tensors(self._attention_layer_size,
62             batch_size, dtype),
63         alignments=self._attention_mechanism.initial_alignments(batch_size,
64             dtype),
65         alignment_history=tensor_array_ops.TensorArray(dtype=dtype,
66             size=0, dynamic_size=True),
67         attention_state = tensor_array_ops.TensorArray(dtype=dtype,
68             size=0, dynamic_size=True)
69     )
70
71     # 定义类的调用方法，将当前时刻的真实值与 Decoder 输出的状态值传入，进行下一时刻的预测
72     def __call__(self, inputs, state):
73
74         drop_rate = 0.5 if self._training else 0.0 # 设置 dropout 的丢弃参数
75
76         # 对输入预处理
77         with tf.variable_scope('decoder_prenet'):#两个全连接转化 mel 特征
78             for i, size in enumerate([256, 128]):
79                 dense = tf.keras.layers.Dense(units=size, activation=tf.nn.relu,
80                     name='dense_%d' % (i+1))(inputs)
81                 inputs = tf.keras.layers.Dropout(rate=drop_rate,
82                     name='dropout_%d' % (i+1))(dense, training=self._training)
83
84         # 加入注意力特征
85         rnn_input = tf.concat([inputs, state.attention], axis=-1)
86
87         # 将连接后的结果经过一个全连接变换，再传入解码器的 RNN 模型中
88         rnn_output, next_cell_state =
89             self._cell(tf.keras.layers.Dense(256)(rnn_input), state.cell_state)
90
91         # 计算本次注意力

```

```

81     context_vector, alignments, cumulated_alignments = attention_wrapper._compute_attention(self._attention_mechanism,
82         rnn_output, state.alignments, None) #state.alignments 为上一次的累计注意力
83
84     #保存历史 alignment(与原始的 AttentionWrapper 一致)
85     alignment_history = state.alignment_history.write(state.time, alignments)
86
87     #返回本次的 wrapper 状态
88     next_state = tf.contrib.seq2seq.AttentionWrapperState(time=state.time
89 + 1, cell_state=next_cell_state, attention=context_vector,
90     alignments=cumulated_alignments, alignment_history=alignment_history,
91     attention_state=state.attention_state)
92
93     #计算本次结果: 将解码器输出与注意力结果用 concat 函数连接起来, 作为最终的输入
94     projections_input = tf.concat([rnn_output, context_vector], axis=-1)
95
96     #两个全连接分别预测输出的下一个结果和停止标志<stop_token>
97     cell_outputs = self._frame_projection(projections_input) #得到下一次
98     outputs_per_step 个帧的 mel 特征
99     stop_tokens = self._stop_projection(projections_input)
100    if self._training==False: #测试时需要加上 sigmoid。
101        stop_tokens = tf.nn.sigmoid(stop_tokens)
102
103    return (cell_outputs, stop_tokens), next_state

```

代码第 99 行进行应用场景的区别。只有在测试模型时才会使用激活函数 sigmoid。原因是在训练模型时，计算损失值 loss 部分会使用 sigmoid_cross_entropy 函数，在这个过程中包含用激活函数 sigmoid 对生成的 stop_tokens 特征进行处理，所以此处不再需要激活函数 sigmoid 了。

9.8.8 代码实现：构建自定义采样器

9.1.13 小节介绍了 Seq2Seq 框架的多种现有采样接口和自定义采样接口。但是这些采样接口都满足不了本实例的采样器需求。本小节还需要实现一个自定义的采样器，原因是：本实例的采样器在计算是否停止采样的过程中需要引入额外的变量 stop_token_preds，该变量是 TacotronDecoderwrapper 的输出结果之一。另外，在采样时还需要对原始的 mel 特征按照指定的步长进行抽取采样，这也是原有接口所不支持的。

下面仿照 tf.contrib.seq2seq.GreedyEmbeddingHelper 采样器的实现，对 __init__、initialize、sample、next_inputs 这四个方法进行重构。具体代码如下。

代码 9-13 TacotronHelpers

```

01 import numpy as np
02 import tensorflow as tf
03 from tensorflow.contrib.seq2seq import Helper
04

```

```

05 def _go_frames(batch_size, output_dim): #输入的目标序列以 0 开始。作为<GO>标志
06     return tf.tile([[0.0]], [batch_size, output_dim])
07
08 class TacoTrainingHelper(Helper):#训练场景中的采样接口
09     def __init__(self, targets, output_dim, r): #targets 形状为[N, T_out, D]
10         with tf.name_scope('TacoTrainingHelper'):
11             self._batch_size = tf.shape(targets)[0] #获得批次
12             self._output_dim = output_dim
13             self._reduction_factor = r
14
15             #对输入数据进行步长为 r 的采样。在每 r(5) 个 mel 中取一个作为下一时刻的 y
16             self._targets = targets[:, r-1::r, :]
17
18             num_steps = tf.shape(self._targets)[1] #获得序列长度(采样后的最大步数)
19             self._lengths = tf.tile([num_steps], [self._batch_size])#构建 RNN 模型
20             输入所用的长度矩阵
21
22     @property
23     def batch_size(self):
24         return self._batch_size
25
26     @property
27     def token_output_size(self):#输出的大小为 5
28         return self._reduction_factor
29
30     @property
31     def sample_ids_shape(self):
32         return tf.TensorShape([])
33
34     @property
35     def sample_ids_dtype(self):
36         return np.int32
37
38     def initialize(self, name=None): #初始化时，设置输入值为 0，代表 go
39         return (tf.tile([False], [self._batch_size]),
40         _go_frames(self._batch_size, self._output_dim))
41
42     def sample(self, time, outputs, state, name=None): #补充接口
43         return tf.tile([0], [self._batch_size])
44
45     #取 time 时刻的数据传入
46     def next_inputs(self, time, outputs, state, name=None, **unused_kwargs):
47         with tf.name_scope(name or 'TacoTrainingHelper'):
48             finished = (time + 1 >= self._lengths) #判断是否结束
49             next_inputs = self._targets[:, time, :] #下一时刻的输入标签
50             return (finished, next_inputs, state)

```

```

50 class TacoTestHelper(Helper): #测试场景中的采样接口
51     def __init__(self, batch_size, output_dim, r):
52         with tf.name_scope('TacoTestHelper'):
53             self._batch_size = batch_size
54             self._output_dim = output_dim
55             self._reduction_factor = r #采样的步长
56
57     @property
58     def batch_size(self):
59         return self._batch_size
60
61     @property
62     def token_output_size(self): #自定义属性
63         return self._reduction_factor
64
65     @property
66     def sample_ids_shape(self):
67         return tf.TensorShape([])
68
69     @property
70     def sample_ids_dtype(self):
71         return np.int32
72
73     def initialize(self, name=None):
74         return (tf.tile([False], [self._batch_size]),
75                 _go_frames(self._batch_size, self._output_dim))
76
77     def sample(self, time, outputs, state, name=None):
78         return tf.tile([0], [self._batch_size]) #返回全 0
79
80     def next_inputs(self, outputs, state, stop_token_preds, name=None,
81                     **unused_kwargs):#测试时靠 stop_token_preds 判断结束
82
83         with tf.name_scope('TacoTestHelper'):
84             #如果 stop 概率>0.5, 即为 stop 标志
85             finished = tf.reduce_any(tf.cast(tf.round(stop_token_preds),
86                                         tf.bool))
87
88             #将解码器输出的最后一帧作为下一时刻的输入
89             next_inputs = outputs[:, -self._output_dim:]
90
91             return (finished, next_inputs, state)

```

在上面代码中，分别实现了两个采样接口。

- TacoTrainingHelper 类：用于训练使用的采样接口（见代码第 8 行）。
- TacoTestHelper 类：用于测试使用的采样接口（见代码第 50 行）。

由于场景不同，二者也有各自不同的实现流程。

- 训练场景：使用的是目标输出结果作为输入采样数据，需要在 `_init_` 环节中对输入数据进行按指定步长的采样（见代码第 16 行）。在函数 `next_inputs` 中，在获取下一个采样数据时，通过判断总共的采样次数来决定是否结束采样过程（见代码第 46 行）。
- 测试场景：使用已有的解码器模型输出的当前时刻结果作为采样数据。由于解码器每次输出指定步长个 mel 特征，所以采样时只取其最后一个（见代码第 86 行）。因为在训练模型的过程中已经得到了对停止符号的判断，所以，直接通过停止符号来决定是否结束采样过程（见代码第 83 行）。

9.8.9 代码实现：构建自定义解码器

在 TensorFlow 中，提供了几种支持 Seq2Seq 框架的解码器接口，其中包括 `BasicDecoder` 接口、`BeamSearchDecoder` 接口等。这些解码器接口与 Seq2Seq 框架中的解码器模型名称相似，但意义不同，具体区别如下。

- TensorFlow 中的解码器是指代码实现过程中的一个 API，它将采样器 `Help` 与 RNN 解码器模型直接连接及耦合，即从采样器 `Help` 中采样，再传入 RNN 解码器模型中进行运算。
- Seq2Seq 框架中的解码器是相对于编码器而言的。该解码器能将编码器输出的结果解码成目标序列。在具体实现时，Seq2Seq 框架中的解码器包含了 `Help`、RNN 解码器、`Decoder`、注意力（可选）部分。

在 Seq2Seq 框架中使用解码器接口 `BasicDecoder` 的方法可以参考 9.4 节。

TensorFlow 中的原生解码器（`BasicDecoder`）接口，同样不能满足本实例的需求。原因是在 9.8.8 小节中已经修改了原始的采样器接口。即每次调用 `next_inputs` 接口进行采样时，还需要传入特征数据 `stop_token_preds`。而 TensorFlow 的原生接口不支持特征数据 `stop_token_preds` 的传入，所以需要额外开发一套。

下面仿照解码器接口 `BasicDecoder` 的实现过程，在其采样过程中和返回的结果中都加上 `stop_token` 的处理。具体代码如下。

代码 9-14 TacotronDecoder

```

01 import collections
02 import tensorflow as tf
03 from tensorflow.contrib.seq2seq.python.ops import decoder
04 from tensorflow.contrib.seq2seq.python.ops import helper as helper_py
05 from tensorflow.python.framework import ops
06 from tensorflow.python.framework import tensor_shape
07 from tensorflow.python.layers import base as layers_base
08 from tensorflow.python.ops import rnn_cell_impl
09 from tensorflow.python.util import nest
10
11 #在输出类型中，添加 token_output 作为 stop token 的输出
12 class TacotronDecoderOutput(

```

```

13     collections.namedtuple("TacotronDecoderOutput", ("rnn_output",
14         "token_output", "sample_id"))):
15     pass
16 #自定义解码器实现类，来自 Tacotron 2 的结构
17 class TacotronDecoder(decoder.Decoder):
18     #初始化
19     def __init__(self, cell, helper, initial_state, output_layer=None):
20         rnn_cell_impl.assert_like_rnncell(type(cell), cell)
21         if not isinstance(helper, helper_py.Helper):
22             raise TypeError("helper must be a Helper, received: %s" %
23                             type(helper))
24         if (output_layer is not None
25             and not isinstance(output_layer, layers_base.Layer)):
26             raise TypeError(
27                 "output_layer must be a Layer, received: %s" %
28                             type(output_layer))
29         self._cell = cell
30         self._helper = helper
31         self._initial_state = initial_state
32         self._output_layer = output_layer
33     @property
34     def batch_size(self):      #返回批次 size
35         return self._helper.batch_size
36     def _rnn_output_size(self):#返回 RNN 模型的输出尺寸
37         size = self._cell.output_size
38         if self._output_layer is None:
39             return size
40         else:
41             output_shape_with_unknown_batch = nest.map_structure(
42                 lambda s:
43                     tensor_shape.TensorShape([None]).concatenate(s),
44                     size)
45             layer_output_shape =
46             self._output_layer._compute_output_shape(output_shape_with_unknown_batch
47         )
48             return nest.map_structure(lambda s: s[1:], layer_output_shape)
49     @property
50     def output_size(self):    #返回输出 size
51         return TacotronDecoderOutput(
52             rnn_output=self._rnn_output_size(),
53             token_output=self._helper.token_output_size(),
54             sample_id=self._helper.sample_ids_shape)

```

```

54 @property
55 def output_dtype(self):      #返回输出类型
56     dtype = nest.flatten(self._initial_state)[0].dtype
57     return TacotronDecoderOutput(
58         nest.map_structure(lambda _: dtype, self._rnn_output_size()),
59         tf.float32,
60         self._helper.sample_ids_dtype)
61
62 def initialize(self, name=None):
63     #返回(finished, first_inputs, initial_state)
64     return self._helper.initialize() + (self._initial_state,)
65
66 def step(self, time, inputs, state, name=None):#执行解码的具体步骤
67
68     with ops.name_scope(name, "TacotronDecoderStep", (time, inputs,
69     state)):
70         #调用解码器cell
71         (cell_outputs, stop_token), cell_state = self._cell(inputs, state)
72         #应用指定的输出层
73         if self._output_layer is not None:
74             cell_outputs = self._output_layer(cell_outputs)
75         sample_ids = self._helper.sample(
76             time=time, outputs=cell_outputs, state=cell_state)
77
78         #调用help进行采样
79         (finished, next_inputs, next_state) = self._helper.next_inputs(
80             time=time, outputs=cell_outputs, state=cell_state,
81             sample_ids=sample_ids, stop_token_preds=stop_token)
82
83         outputs = TacotronDecoderOutput(cell_outputs, stop_token, sample_ids)
84
85     return (outputs, next_state, next_inputs, finished)

```

9.8.10 代码实现：构建输入数据集

数据集构建相对比较好理解。下面读取 9.8.3 小节保存的 metadata 文件，将里面的指定音频文件——numpy 文件载入内存，形成音频数据与拼音对应的输入样本。具体代码如下。

代码 9-15 cn_dataset

```

01 import tensorflow as tf          #载入模块
02 import os
03 import numpy as np
04 _pad,_eos = '_', '~'           #定义填充字符与结束字符
05 _padv = 0                      #定义填充的向量占位符
06 _stop_token_padv = 1            #定义标志结束的向量占位符

```

```

08     collection_name = "TextToPinyinOutput", "train_outgoing_code": 12
09 _characters = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz!`()-.:;?"
10 symbols = [_pad, _eos] + list(_characters) # 定义字典
11 index_symbols = {value:key for key, value in enumerate(symbols) }
12 print(index_symbols)
13 def text_to_id(text, maxlen, nlpipes, output_layer=None):
14     strlen = len(symbols)
15     return ''.join([symbols[i] for i in sequence if i<strlen ])
16
17 # 定义数据集
18 def mydataset(metadata_filename,outputs_per_step,batch=32,shuffleflag=True,mode = 'train'):
19
20     # 加载 metadata 文件
21     datadir = os.path.dirname(metadata_filename)
22     with open(metadata_filename, encoding='utf-8') as f:
23         print("metadata_filename",metadata_filename)
24         _metadata = [line.strip().split('!') for line in f]
25
26     # 加载拼音字符，并计算最大长度
27     inputseq = list( map(lambda x:[index_symbols[key] for key in
28         x[3] ],_metadata) )
29     seqlen = [len(x) for x in inputseq]
30
31     # 计算语音最大长度
32     Max_output_length = int(max(m[2] for m in _metadata))+1
33     # 按照 outputs_per_step 步长对最大输出长度进行取整
34     Max_output_length =[ Max_output_length + outputs_per_step - 1
35     Max_output_length%outputs_per_step,Max_output_length][Max_output_length%
36     outputs_per_step==0]
37
38     # 对输入的拼音补 0
39     inputseq = tf.keras.preprocessing.sequence.pad_sequences(inputseq,
40         padding='post',value=_padv)
41     print(inputseq)
42     print(len(inputseq[0]))
43     # 定义拼音数据集
44     datasetinputseq = tf.data.Dataset.from_tensor_slices( inputseq )
45     # 定义输入长度数据集
46     datasetseqlen = tf.data.Dataset.from_tensor_slices( seqlen )
47     # 定义全部的 metadata 数据集
48     datasetmetadata = tf.data.Dataset.from_tensor_slices( _metadata )
49     # 合并数据集

```

```

47     dataset =
48     tf.data.Dataset.zip((datasetmetadata,datasetinputseq,datasetseqlen))
49
50     def mymap(_meta,seq,seqlen):#对合并好的数据集按照指定规则进行处理
51         def _parse(meta):
52             #根据文件名加载音频数据的 np 文件
53             linear_target = np.load(os.path.join(datadir,
54                 meta.numpy()[0].decode('UTF-8') ))
55             mel_target = np.load(os.path.join(datadir,
56                 meta.numpy()[1].decode('UTF-8')))

57             #构造结束掩码
58             stop_token_target = np.asarray([0.] * len(mel_target),dtype =
59                 np.float32)
60
61             #统一对齐操作
62             linear_target =np.pad(linear_target, [(0, Max_output_length -
63                 linear_target.shape[0]), (0,0)], mode='constant', constant_values=_padv)
64             mel_target =np.pad(mel_target, [(0, Max_output_length -
65                 mel_target.shape[0]), (0,0)], mode='constant', constant_values=_padv)
66             stop_token_target =np.pad(stop_token_target, (0,
67                 Max_output_length - len(stop_token_target)), mode='constant',
68                 constant_values=_stop_token_padv)
69             #返回处理后的单条样本
70             return linear_target,mel_target,stop_token_target
71
72         linear_target,mel_target,stop_token_target = tf.py_function( _parse,
73             [_meta], [tf.float32,tf.float32,tf.float32])
74
75         return seq,seqlen,linear_target,mel_target,stop_token_target#调用第
76        三方函数进行 map 处理的返回值
77
78     dataset = dataset.map(mymap)      #对数据进行 map 处理
79     if mode=='train':                #在训练场景中进行乱序操作
80         if shuffleflag == True:      #对数据进行乱序操作
81             dataset = dataset.shuffle(buffer_size=1000)
82         dataset = dataset.repeat()
83     dataset = dataset.batch(batch)    #批次划分
84
85     iterator = dataset.make_one_shot_iterator()
86     print(dataset.output_types)
87     print(dataset.output_shapes)
88     next_element = iterator.get_next()
89
90     return next_element

```

代码第 34 行按照 outputs_per_step 步长，对最大输出长度进行取整。如果 Max_output_length%outputs_per_step==0 成立，则返回 Max_output_length。否则返回 Max_output_length + outputs_per_step - Max_output_length%outputs_per_step。

**提示：**

代码第 34 行，使用了 Python 中的条件判断语法。相关的语法介绍可以参考《Python 带我起飞——入门、进阶、商业实战》一书的 5.1.2 小节的“注意”部分。

9.8.11 代码实现：构建 Tacotron 网络

按照 9.8.4 小节描述的流程，将 Seq2Seq 框架的主要模块组合起来，搭建 Tacotron 网络。具体步骤如下。

- (1) 对输入 inputs 做词嵌入变换。
- (2) 对词嵌入结果 embedded_inputs 做两次全连接，再经过 encoder_cbhg 网络，完成编码。
- (3) 定义 RNN 模型用于整个网络的解码部分。
- (4) 实例化 TacotronDecoderwrapper 类，为 RNN 模型的解码结果添加混合注意力机制。
- (5) 定义采样器 help 对象。
- (6) 调用动态解码接口 dynamic_decode，实现 Seq2Seq 框架的循环处理，得到 mel 特征和停止标志符。
- (7) 用 post_cbhg 模型对合成后的 mel 特征进行计算，得到基于帧频的特征数据。

具体代码如下。

代码 9-16 tacotron

```

01 import tensorflow as tf
02 cbhg = __import__("9-10_cbhg")
03 encoder_cbhg = cbhg.encoder_cbhg
04 post_cbhg = cbhg.post_cbhg
05 rnnwrapper = __import__("9-12_TacotronDecoderwrapper")
06 TacotronDecoderwrapper = rnnwrapper.TacotronDecoderwrapper
07 Helpers= __import__("9-13_TacotronHelpers")
08 TacoTestHelper = Helpers.TacoTestHelper
09 TacoTrainingHelper = Helpers.TacoTrainingHelper
10 Decoder= __import__("9-14_TacotronDecoder")
11 TacotronDecoder = Decoder.TacotronDecoder
12 cn_dataset = __import__("9-15_cn_dataset")
13 symbols = cn_dataset.symbols
14 class Tacotron():
15     #初始化
16     def __init__(self, inputs,#形状为[N, input_length]。N 代表批次, input_length
      代表序列长度
17         input_lengths, #形状为[N]
18         num_mels,outputs_per_step,num_freq,
19         linear_targets=None,#形状为 [N, targets_length, num_freq],
      targets_length 代表输出序列
20         mel_targets=None, #形状为 [N, targets_length, num_mels]
21         stop_token_targets=None):

```

```

22 12. #使用tf.variable_scope()来为不同的模型部分设置不同的名字
23     with tf.variable_scope('inference') as scope:
24         is_training = linear_targets is not None
25         batch_size = tf.shape(inputs)[0]
26
27     #词嵌入转换
28     embedding_table = tf.get_variable('embedding', [len(symbols), 256],
29                                         dtype=tf.float32,
30                                         initializer=tf.truncated_normal_initializer(stddev=0.5))
31     embedded_inputs = tf.nn.embedding_lookup(embedding_table, inputs) #词嵌入形状为[N, input_lengths, 256]
32
33     #定义RNN编码器(两层全连接+encoder_cbhg)
34     drop_rate = 0.5 if is_training else 0.0
35     with tf.variable_scope('encoder_prenet'):
36         for i, size in enumerate([256, 128]):
37             dense = tf.keras.layers.Dense(units=size, activation=tf.nn.relu,
38                                         name='dense_%d' % (i+1))(embedded_inputs)
39             embedded_inputs = tf.keras.layers.Dropout(rate=drop_rate,
40                                         name='dropout_%d' % (i+1))(dense, training=is_training)
41     #最终解码特征输出的形状为[N, input_length, 256]
42     encoder_outputs = encoder_cbhg(embedded_inputs, input_lengths,
43                                     is_training, 256)
44
45     #定义RNN解码网络
46     multi_rnn_cell = tf.nn.rnn_cell.MultiRNNCell([
47         tf.nn.rnn_cell.ResidualWrapper(tf.nn.rnn_cell.GRUCell(256)),
48         tf.nn.rnn_cell.ResidualWrapper(tf.nn.rnn_cell.GRUCell(256))
49     ], state_is_tuple=True) #输出形状为[N, input_length, 256]
50
51     #实例化TacotronDecoderwrapper
52     decoder_cell = TacotronDecoderwrapper(encoder_outputs, is_training,
53                                         multi_rnn_cell,
54                                         num_mels, outputs_per_step)
55
56     if is_training:#选择不同的采样器
57         helper = TacoTrainingHelper(mel_targets, num_mels, outputs_per_step)
58     else:
59         helper = TacoTestHelper(batch_size, num_mels, outputs_per_step)
60
61     #初始化解码器状态
62     decoder_init_state = decoder_cell.zero_state(batch_size=batch_size,
63                                                 dtype=tf.float32)
64
65     max_iters=300 #解码的最大长度为300, 实际生成的长度为300×5
66
67     learning_rate_decay_fn = lambda global_step:
68         learning_rate_decay_fn(global_step)
69
70     warmup_steps = 1000 #解码时的步数
71
72     return helper, learning_rate_decay_fn, global_step

```

```

60     (decoder_outputs, stop_token_outputs, _), final_decoder_state, _ = tf.contrib.seq2seq.dynamic_decode(
61         TacotronDecoder(decoder_cell, helper,
62         decoder_init_state), maximum_iterations=max_iters)
63
64     #对输出结果进行 Reshape, 生成 mel 特征[N, outputs_per_step, num_mels]。
65     self.mel_outputs = tf.reshape(decoder_outputs, [batch_size, -1,
66     num_mels])
67     self.stop_token_outputs = tf.reshape(stop_token_outputs, [batch_size,
68     -1])
69
70     #用 CBHG 对 mel 特征后处理, 形状为[N, outputs_per_step, 256]
71     post_outputs = post_cbhg(self.mel_outputs, num_mels, is_training, 256)
72     #用全连接网络将处理后的 mel 特征还原, 输出形状为[N, outputs_per_step,
73     num_freq]
74     self.linear_outputs = tf.keras.layers.Dense(num_freq)(post_outputs)
75
76     #获取注意力机制的全部结果, 用于可视化
77     self.alignments =
78         tf.transpose(final_decoder_state.alignment_history.stack(), [1, 2, 0])
79
80     self.inputs = inputs
81     self.input_lengths = input_lengths
82     self.mel_targets = mel_targets
83     self.linear_targets = linear_targets
84     self.stop_token_targets = stop_token_targets
85
86     tf.logging.info('Initialized Tacotron model. Dimensions: ')
87     tf.logging.info(' embedding: ')
88     tf.logging.info(' encoder out: ')
89     tf.logging.info(' decoder out (r frames): ')
90     tf.logging.info(' decoder out (1 frame): ')
91     tf.logging.info(' decoder out: ')
92     tf.logging.info(' postnet out: ')
93     tf.logging.info(' postnet out: ')
94     tf.logging.info(' linear out: ')
95     tf.logging.info(' linear out: ')
96     tf.logging.info(' stop token: ')
97     tf.logging.info(' stop token: ')

```

代码第 42 行, 通过两层带有残差网络的 GRU 单元实现了解码器的主体。

带有残差网络的 RNN 模型与残差卷积网络模型的功能类似, 将深度网络结构调整为了并行网络, 可以支持更多层结构的反向传播和更好的特征表达。

9.8.12 代码实现：构建 Tacotron 网络模型的训练部分

训练模型部分主要是对损失值 loss 的计算。该损失值包括了 3 部分：Mel 特征 loss 值、stop token 的 loss 值、帧频的 loss 值。具体代码如下。

代码 9-16 tacotron（续）

```

88 def buildTrainModel(self, sample_rate, num_freq, global_step):
89     #计算 loss 值
90     with tf.variable_scope('loss') as scope:
91         #计算 mel 特征的 loss 值
92         self.mel_loss = tf.reduce_mean(tf.abs(self.mel_targets -
93                                         self.mel_outputs))
94         #计算停止符的 loss 值
95         self.stop_token_loss =
96             tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
97                             labels=self.stop_token_targets,
98                             logits=self.stop_token_outputs))
99
100    l1 = tf.abs(self.linear_targets - self.linear_outputs)
101    #计算 Prioritize 的 loss 值
102    n_priority_freq = int(4000 / (sample_rate * 0.5) * num_freq)
103    self.linear_loss = 0.5 * tf.reduce_mean(l1) + 0.5 *
104        tf.reduce_mean(l1[:, :, 0:n_priority_freq])
105
106    self.loss = self.mel_loss + self.linear_loss + self.stop_token_loss
107
108    #定义优化器
109    with tf.variable_scope('optimizer') as scope:
110        initial_learning_rate=0.001
111        self.learning_rate = _learning_rate_decay(initial_learning_rate,
112            global_step)
113
114        optimizer = tf.train.AdamOptimizer(self.learning_rate)
115        gradients, variables = zip(*optimizer.compute_gradients(self.loss))
116        self.gradients = gradients
117        clipped_gradients, _ = tf.clip_by_global_norm(gradients, 1.0)
118
119        #在 BN 运算之后更新权重
120        with
121            tf.control_dependencies(tf.get_collection(tf.GraphKeys.UPDATE_OPS)):
122                self.optimize = optimizer.apply_gradients(zip(clipped_gradients,
123                    variables),
124                        global_step=global_step)
125
126    #退化学习率
127    def _learning_rate_decay(init_lr, global_step):
128        warmup_steps = 4000.0#超参数来自于 tensor2tensor:

```

```

123 step = tf.cast(global_step + 1, dtype=tf.float32)
124 return init_lr * warmup_steps**0.5 * tf.minimum(step * warmup_steps**-1.5,
    step**-0.5)

```

代码第 121 行，用退化学习率的方法来训练模型。在退化学习率的算法中，加入了 `warmup_steps` 参数，该参数的值为 4000。这个 4000 是一个经验值，该经验值来自 `tensor2tensor` 框架中的代码。经过函数 `learning_rate_decay` 所生成的退化学习率，可以使模型在训练过程中收敛得更快。读者也可以将函数 `learning_rate_decay` 直接用在自己的模型训练中。

9.8.13 代码实现：训练模型并合成音频文件

下面加载数据集并构建模型，进行训练。

在训练过程中，对模型输出的音频特征数据进行音频转换，并将转换结果保存起来。

音频转换的过程是 9.8.2 小节的逆过程，即使用反向短时傅里叶变换对音频信号进行还原。具体代码如下。

代码 9-17 train

```

01 from datetime import datetime
02 import math
03 import os
04 import time
05 import tensorflow as tf
06 import traceback
07 import numpy as np
08 from scipy import signal
09 import librosa
10 from scipy.io import wavfile
11 import matplotlib.pyplot as plt
12
13 tacotron = __import__("9-16_tacotron")
14 Tacotron = tacotron.Tacotron
15 cn_dataset = __import__("9-15_cn_dataset")
16 mydataset = cn_dataset.mydataset
17 sequence_to_text = cn_dataset.sequence_to_text
18
19 def time_string(): # 定义函数将时间转换为字符串
20     return datetime.now().strftime('%Y-%m-%d %H:%M')
21
22 max_frame_num=1000 # 定义每个音频文件的最大帧数
23 sample_rate=16000 # 定义音频文件的采样率
24 num_mels=80
25 num_freq=1025
26 outputs_per_step = 5
27

```

```

28 n_fft = (num_freq - 1) * 2 #stft 算法中使用的窗口大小(因为声音的真实频率只有正的,而 fft 变换是对称的,所以需要加上负频率)
29 frame_length_ms=50          #定义 stft 算法中的重叠窗口(用时间来表示)
30 frame_shift_ms=12.5         #定义 stft 算法中的移动步长(用时间来表示)
31 hop_length = int(frame_shift_ms / 1000 * sample_rate) #定义 stft 算法中的帧移步长
32 win_length = int(frame_length_ms / 1000 * sample_rate) #定义 stft 算法中的相邻两个窗口的重叠长度
33 preemphasis=0.97            #用于过滤声音频率的阈值
34 ref_level_db=20             #控制峰值的阈值
35 min_level_db=-100           #指定 dB 最小值,用于归一化
36
37 griffin_lim_iters=60        #Griffin-Lim 算法合成语音时的计算次数
38 power=1.5                  #设置在 Griffin-Lim 算法之前,提升振幅的参数
39
40 def _db_to_amp(x):
41     return np.power(10.0, x * 0.05)
42
43 def _denormalize(S):
44     return (np.clip(S, 0, 1) * -min_level_db) + min_level_db
45
46 def inv_preemphasis(x): #用数字滤波器恢复音频信号
47     return signal.lfilter([1], [1, -preemphasis], x)
48
49 def _griffin_lim(S):      #用 griffin lim 信号估计算法,恢复声音
50     angles = np.exp(2j * np.pi * np.random.rand(*S.shape))
51     S_complex = np.abs(S).astype(np.complex)
52     #反向短时傅里叶变换
53     y = librosa.istft(S_complex * angles, hop_length=hop_length,
54                         win_length=win_length)
55     for i in range(griffin_lim_iters):
56         angles = np.exp(1j *
57                         np.angle(librosa.stft(y, n_fft, hop_length, win_length)))
58     y = librosa.istft(S_complex * angles, hop_length=hop_length,
59                         win_length=win_length)
60     return y
61
62 def inv_spectrogram(spectrogram): #将特征信号转换成 wave 形式的声音
63     S = _db_to_amp(_denormalize(spectrogram) + ref_level_db) #将 dB 频谱转为音频特征信号
64     return inv_preemphasis(_griffin_lim(S ** power))
65
66 def save_wav(wav, path):
67     wav *= 32767 / max(0.01, np.max(np.abs(wav)))
68     #librosa.output.write_wav(path, wav.astype(np.int16), sample_rate)
69     wavfile.write(path, sample_rate, wav.astype(np.int16))
70
71 model.linear_outputs[0], model.alignments[0])

```

```

68 def train(log_dir): #训练模型
69     return init(log_dir)
70     checkpoint_path = os.path.join(log_dir, 'model.ckpt')
71     tf.logging.info('Checkpoint path: %s' % checkpoint_path)
72     #加载数据集
73     next_element = mydataset('training/train.txt', outputs_per_step=5)
74     #定义输入占位符
75     inputs = tf.placeholder(tf.int32, [None, None], 'inputs')
76     input_lengths = tf.placeholder(tf.int32, [None], 'input_lengths')
77     linear_targets = tf.placeholder(tf.float32, [None, None, num_freq],
78     'linear_targets')
78     mel_targets = tf.placeholder(tf.float32, [None, None, num_mels],
79     'mel_targets')
79     stop_token_targets = tf.placeholder(tf.float32, [None, None],
80     'stop_token_targets')
80
81     #构建模型
82     global_step = tf.Variable(0, name='global_step', trainable=False)
83     with tf.variable_scope('model') as scope:
84         model = Tacotron(inputs,
85             input_lengths, num_mels, outputs_per_step, num_freq,
86             linear_targets, mel_targets, stop_token_targets)
87         model.buildTrainModel(sample_rate, num_freq, global_step)
88     time_window = []
89     loss_window = []
90     saver = tf.train.Saver(max_to_keep=5, keep_checkpoint_every_n_hours=2)
91
92     epochs = 100000          #定义迭代训练的次数
93     checkpoint_interval = 1000 #每 1000 次保存一次检查点
94     os.makedirs(log_dir, exist_ok=True)
95     checkpoint_state = tf.train.get_checkpoint_state(log_dir)
96
97     def plot_alignment(alignment, path, info=None): #输出音频图谱
98         fig, ax = plt.subplots()
99         im = ax.imshow(
100             alignment,
101             aspect='auto',
102             origin='lower',
103             interpolation='none')
104         fig.colorbar(im, ax=ax)
105         xlabel = 'Decoder timestep'
106         if info is not None:
107             xlabel += '\n\n' + info
108         plt.xlabel(xlabel)
109         plt.ylabel('Encoder timestep')
110         plt.tight_layout()

```

```

111     plt.savefig(path, format='png')
112
113 with tf.Session() as sess:
114     sess.run(tf.global_variables_initializer())
115     #恢复检查点
116     if checkpoint_state is not None:
117         saver.restore(sess, checkpoint_state.model_checkpoint_path)
118         tf.logging.info('Resuming from checkpoint: %s' %
119         (checkpoint_state.model_checkpoint_path))
120     else:
121         tf.logging.info('Starting new training')
122
123     try:                                #迭代训练
124         for i in range(eporch):
125             seq,seqlen,linear_target,mel_target,stop_token_target =
126             sess.run(next_element)
127             start_time = time.time()
128             step, loss, opt = sess.run([global_step, model.loss,
129             model.optimize],
130                                         feed_dict={inputs: seq, input_lengths:
131             seqlen,
132                                         linear_targets:
133             linear_target,mel_targets: mel_target,
134                                         stop_token_targets:
135             stop_token_target})
136             time_window.append(time.time() - start_time)
137             loss_window.append(loss)
138             message = 'Step %-7d [% .03f sec/step, loss=% .05f, avg_loss=% .05f]' %
139             (step, sum(time_window) / max(1, len(time_window)),
140             loss, sum(loss_window) / max(1, len(loss_window)))
141             tf.logging.info(message)
142             if loss > 100 or math.isnan(loss):
143                 tf.logging.info('Loss exploded to %.05f at step %d!' % (loss,
144                 step))
145                 raise Exception('Loss Exploded')
146
147             if step % checkpoint_interval == 0:
148                 tf.logging.info('Saving checkpoint to: %s-%d' % (checkpoint_path,
149                 step))
150                 saver.save(sess, checkpoint_path, global_step=step)
151                 tf.logging.info('Saving audio and alignment...')
152                 #输出模型结果
153                 input_seq, spectrogram, alignment = sess.run([model.inputs[0],
154                 model.linear_outputs[0], model.alignments[0]],
155

```

```

150 def train(log_dir, feed_dict={inputs: seq, input_lengths: seqlen,
151                         linear_targets: linear_target, mel_targets: mel_target,
152                         stop_token_targets: stop_token_target}):
153     logging.info('Creating checkpoint links')
154     waveform = inv_spectrogram(spectrogram.T) #转换成音频数据
155     #保存音频数据
156     save_wav(waveform, os.path.join(log_dir, 'step-%d-audio.wav' % step))
157     #绘制音频图谱
158     plot_alignment(alignment, os.path.join(log_dir,
159                                         'step-%d-align.png' % step),
160                                         info=' %s, step=%d, loss=%.5f' % (time_string(), step, loss))
161     tf.logging.info('Input: %s' % sequence_to_text(input_seq))
162 except Exception as e:
163     tf.logging.info('Exiting due to exception: %s' % e)
164     traceback.print_exc()
165 with tf.Session() as sess:
166 if __name__ == '__main__':
167     tf.reset_default_graph() #重置图
168     tf.logging.set_verbosity(tf.logging.INFO) #定义输出的 log 级别
169     train(os.path.join('.', 'model-cpk')) #训练模型

```

代码运行后，会输出如下结果：

```

.....
Step 11060 [4.415 sec/step, loss=0.05607, avg_loss=0.05585]
Step 11061 [4.441 sec/step, loss=0.05787, avg_loss=0.05588]
Step 11062 [4.476 sec/step, loss=0.05712, avg_loss=0.05591]
Step 11063 [4.504 sec/step, loss=0.06074, avg_loss=0.05595]
Step 11064 [4.532 sec/step, loss=0.05887, avg_loss=0.05602]
Step 11065 [4.551 sec/step, loss=0.05710, avg_loss=0.05605]

```

在代码的同级目录下，生成了 model-cpk 文件夹。该文件夹里面包含每训练 1000 步所输出的音频图谱（如图 9-19 所示）与 wav 文件“step-11000-audio.wav”。

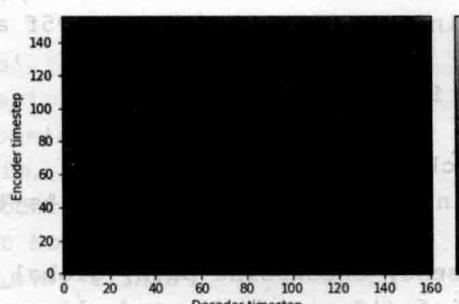


图 9-19 TTS 合成的音频图谱

9.8.14 扩展：用 pypinyin 模块实现文字到声音的转换

本实例将拼音文本转换为具体的声音。在实际应用中，在大部分的场景中是需要将文字转换为声音的，可以通过 pypinyin 模块的辅助来完成。Pypinyin 模块的功能是将文字转换为拼音，这样有了拼音之后便可以再进一步转换成声音。

下面需要先通过 pip install pypinyin 命令安装 pypinyin 模块，接着可以通过以下示例代码完成拼音的转换：

```
from pypinyin import lazy_pinyin, Style #pip install pypinyin
a = lazy_pinyin('代码医生工作室，习惯成就精品', style=Style.TONE3)
print(a)
```

代码执行之后，会看到输出了包含对应拼音的列表：

```
['dai4', 'ma3', 'yi1', 'sheng1', 'gong1', 'zuo4', 'shi4', ' ', ' ', 'xi2', 'guan4',
'cheng2', 'jiu4', 'jing1', 'pin3']
```

另外，pypinyin 模块还可以生成各种不同形式的拼音，甚至具有分词功能。例如：

```
from pypinyin.contrib.mmseg import seg
text = '代码医生工作室，习惯成就精品'
b = list(seg.cut(text))
print(b) #输出：['代', '码', '医', '生', '工', '作', '室', ' ', ' ', '习', '惯', '成',
'就', '精', '品']
```

更多的使用方法可以参考 pypinyin 模块的源码连接：

<https://github.com/mozillazg/python-pinyin>

• 马尔科模型：用于判断样本是否属于真实世界还是假想生成的。

生成模型的作用是，让生成模型能够生成与真实样本，判别模型的作用是，将合成样本与真实样本分辨出来。二者存在矛盾之处，有时一个模型放在一组后，如果判别模型生成的模拟样本会更加真实，那么判别模型的准确率就会降低，生成模型就可以被制作成生成式模型。用马尔科模型对生成模型进行评估，可以解决这个问题，因此独立处

第 4 篇 高级

10.1.3 自端端网络模型与对抗神经网络模型

本篇将介绍多模型的组合训练技术。

在训练模型过程中，可以用两个目标相反的网络模型协同训练，在训练过程中，形成对抗关系。通过对抗过程中的互相制约，促进两个网络模型的更新，从而实现更好的效果。

这种利用多个模型间的对抗关系进行训练的技术，被广泛用在模拟生成任务和攻防任务上。

第 10 章将介绍模拟生成任务，讲述了对抗神经网络模型的种类、训练对抗神经网络模型的方法，以及提升模型性能的技巧。

第 11 章将介绍攻防任务，讲述了对抗样本的制作方法、攻击模型的方法，以及提升模型健壮性的技巧。

- 第 10 章 生成式模型——能够输出内容的网络模型
- 第 11 章 网络模型攻与防——看似智能的 AI 也有其脆弱的一面

1. 半自动适应模式的批量归一化公式

所谓批量归一化中的自适应模式，就是让模型归一化（BN）算子中加上一个权重参数，通过迭代训练，使 BN 算法收敛于一个合适的正常化系数。具体来说，BN 算法中加入了自适应模式后，公式 10.1 变成了：

10.1.1 生成式模型基础入门

公式 10.1 中， γ 和 β 分别表示归一化系数和偏置项，这两个参数是根据数据集自动学习到的。 γ 和 β 是模型的参数，既然是学习到的，那么它们的值应该是随训练进度而变化的。

2. 使用自适应模式的 BN 公式

又如，通过自适应模式的 BN 公式，可以将原本的 BN 公式改写为如下的形式：

- 如果 BN 完成后带有均值归一化功能，则一定会将 scale 参数设为 False（因为自适应模式的功能会在批次层完成）。

- 如果 BN 算法后面没有线性转换层，将 scale 参数设为 True（因为使用带有自适应模式的 BN 算法效果会更好）。

具体应用可以参考 10.3.5，本书从第 8 天开始本章内容已融合于第 10 章。

第 10 章

生成式模型——能够输出内容的模型

生成式模型的主要功能是输出具体样本。该模型用在模拟生成任务中。

生成式模型包括自编码网络模型、对抗神经网络模型。这种模型输出的不再是分类或预测结果，而是符合输入样本分布空间中的一个样本个体。例如：生成与用户匹配的 3D 假牙、合成一些有趣的图片或音乐，甚至是创作小说或是编写代码。当然这些技术都比较前沿，大部分还没成熟或普及。

目前，生成式模型主要用于提升已有模型的性能。例如：

- 用生成式模型可以模拟已有样本的生成，从扩充数据集的角度提升模型的泛化能力（适用于样本不足的场景）。
- 用生成式模型可以制作目标模型的对抗样本。该对抗样本能够提升目标模型的健壮性。
- 将生成式模型嵌入到已有分类或回归任务模型里，通过损失值增加对模型的约束，从而实现精度更好的分类或回归模型。例如在胶囊网络模型中就嵌入了自编码网络模型，完成了重建损失的功能。

10.1 快速导读

在学习实例之前，有必要了解一下自编码网络模型的基础知识。

10.1.1 什么是自编码网络模型

自编码网络模型是一种输出和输入相等的模型。它是典型的非监督学习模型。输入的数据在网络模型中经过一系列特征变换，但在输出时还与输入时一样。

自编码网络模型虽然对单个样本没有意义，但对整体样本集却很有价值。它可以很好地学习到该数据集中样本的分布情况，既能对数据集进行特征压缩，实现提取数据主成分功能；又能与数据集的特征相拟合，实现生成模拟数据的功能。

10.1.2 什么是对抗神经网络模型

对抗神经网络模型由两个模型组成。

- 生成器模型：用于合成与真实样本相差无几的模拟样本。

- 判别器模型：用于判断某个样本是来自于真实世界还是模拟生成的。

生成器模型的作用是，让判别器模型将合成样本当作真实样本；判别器模型的作用是，将合成样本与真实样本分辨出来。二者存在矛盾关系。将两个模型放在一起同步训练，则生成器模型生成的模拟样本会更加真实，判别器模型对样本的判断会更加精准。生成器模型可以被当作成生成式模型，用来独立处理生成式任务；判别器模型可以被当作分类器模型，用来独立处理分类任务。

10.1.3 自编码网络模型与对抗神经网络模型的关系

自编码网络模型和对抗神经网络模型都属于多模型网络结构。二者常常混合使用，以实现更好的生成效果。

自编码网络模型和对抗神经网络模型都属于非监督（或半监督训练）模型。它们会在原有样本的分布空间中随机生成模拟数据。为了使随机生成的方式变得可控，常常会加入条件参数。

从某种角度看，如果自编码网络模型和对抗神经网络模型带上条件参数会更有价值。本书10.3节实现的AttGAN模型就是一个基于条件的模型，它由自编码网络模型和对抗神经网络模型组合而成。

10.1.4 什么是批量归一化中的自适应模式

在《深度学习之TensorFlow——入门、原理与进阶实战》一书的8.9.3小节中，介绍过批量归一化（BatchNorm, BN）算法。该算法是对一个批次图片的所有像素求均值和标准差。在深度神经网络模型中，该算法的作用是让模型更容易收敛、提高模型的泛化能力。

1. 带有自适应模式的批量归一化公式

所谓批量归一化中的自适应模式，就是在批量归一化（BN）算法中加上一个权重参数。通过迭代训练，使BN算法收敛为一个合适的值。

当BN算法中加入了自适应模式后，其数学公式见式（10.1）。

$$BN = \gamma \cdot \frac{(x - \mu)}{\sigma} + \beta \quad (10.1)$$

在式（10.1）中， μ 代表均值， σ 代表方差。这两个值都是根据当前数据运算来的。 γ 和 β 是参数，代表自适应的意思。在训练过程中，会通过优化器的反向求导来优化出合适的 γ 、 β 值。

2. 如何使用带有自适应模式的批量归一化

下面以TF-slim接口中的批量正则化函数slim.batch_norm为例。该函数的参数scale控制着式（10.1）中的 γ 。参数scale的默认值是False，代表不乘以 γ ，即不使用带有自适应模式的BN算法。

- 如果BN算法后面有其他的线性转换层，则一般会将scale参数设为False（因为自适应模式的功能会在线性层被实现）。
- 如果BN算法后面没有线性转换层，则将scale参数设为True（因为使用带有自适应模式的BN算法效果会更好）。

具体应用可以参考10.3.5小节代码。

10.1.5 什么是实例归一化

批量归一化是对一个批次图片的所有像素求均值和标准差。而实例归一化 (InstanceNorm, IN) 是对单一图片进行归一化处理，即对单个图片的所有像素求均值和标准差。

1. 实例归一化的使用场景

在对抗神经网络模型、风格转换这类生成式任务中，常用实例归一化取代批量归一化。因为，生成式任务的本质是——将生成样本的特征分布与目标样本的特征分布进行匹配。生成式任务中的每个样本都有独立的风格，不应该与批次中其他的样本产生太多联系。所以，实例归一化适用于解决这种基于个体的样本分布问题。详细说明见以下链接：

<https://arxiv.org/abs/1607.08022>

2. 如何使用实例归一化

用函数 `tf.contrib.layers.instance_norm` 和函数 `tf.contrib.slim.instance_norm` 可以实现 IN 算法。具体应用可以参考 10.3.7 小节代码。

10.1.6 了解 SwitchableNorm 及更多的归一化方法

归一化方法有很多种，除原始的 BatchNorm 算法（见 10.1.4 小节）、InstanceNorm 算法（见 10.1.5 小节）外，还有 ReNorm 算法、LayerNorm 算法、GroupNorm 算法、SwitchableNorm 算法。

下面以一个形状是[N (批次), H (高), W (宽), C (通道)]的数据为例，介绍这些算法。

1. ReNorm 算法

ReNorm 算法与 BatchNorm 算法一样，注重对全局数据的归一化，即对输入数据的形状中的 N 维度、H 维度、W 维度做归一化处理。不同的是，ReNorm 算法在 BatchNorm 算法上做了一些改进，使得模型在小批次场景中也有良好的效果。具体论文见以下链接：

<https://arxiv.org/pdf/1702.03275.pdf>

在 `tf.Keras` 接口中，在实例化 `BatchNormalization` 类后，将 `renorm` 参数设为 `True` 即可。

2. LayerNorm 算法

LayerNorm 算法是在输入数据的通道方向上，对该数据形状中的 C 维度、H 维度、W 维度做归一化处理。它主要用在 RNN 模型中，可参见《深度学习之 TensorFlow——入门、原理与进阶实战》一书的 9.4.15 小节。

在 TensorFlow 的 1.12 版本及之后的版本中，可以直接用 `tf.contrib.rnn.LayerNormLSTMCell` 类及 `tf.contrib.rnn.LayerNormBasicLSTMCell` 类创建带有 LayerNorm 算法的 RNN 模型单元。



提示：

在使用 `tf.contrib.rnn.LayerNormBasicLSTMCell` 函数时，要求输入值必须被提前归一化到 $-1 \sim 1$ 。如果输入数值在 $0 \sim 1$ 之间，则会出现损失值为“NAN”的现象。

有关 `tf.contrib.rnn.LayerNormBasicLSTMCell` 函数的具体使用实例，见本书配套资源中的代码文件“`lnonMnist(1.11 版本之后).py`”。

同时，还可以用 `tf.contrib.layers.layer_norm` 方法与 `tf.contrib.slim.layer_norm` 方法创建带有 LayerNorm 算法的网络层。

3. InstanceNorm 算法

InstanceNorm（实例归一化）算法是对输入数据形状中的 H 维度、W 维度做归一化处理。它主要用在风格化迁移之类任务的模型中。有关 InstanceNorm 的详细介绍见 10.1.5 小节。

4. GroupNorm 算法

GroupNorm 算法是介于 LayerNorm 算法和 InstanceNorm 算法之间的算法。它首先将通道分为许多组（group），再对每一组做归一化处理。

GroupNorm 算法与 ReNorm 算法的作用类似，都是为了解决 BatchNorm 算法对批次大小的依赖。具体论文见下方链接：

<https://arxiv.org/abs/1803.08494>

5. SwitchableNorm 算法

SwitchableNorm 算法是将 BN 算法、LN 算法、IN 算法结合起来使用，并为每个算法都赋予权重，让网络自己去学习归一化层应该使用什么方法。具体论文见下方链接：

<https://arxiv.org/abs/1806.10779>

具体应用方法可以参考 10.2 节的代码实例。

10.1.7 什么是图像风格转换任务

图像风格转换任务是深度学习中对抗神经网络模型所能实现的经典任务之一。用 CycleGAN 模型生成模拟梵高风格的图画，已经成为一个广为熟知的实例。除此之外，图像风格转换任务还可以实现橘子与苹果间的转化、斑马与普通马之间的转化、照片与油画之间的转化，甚至还可以根据一张风景照片生成四季下的场景图片。

图像风格转化任务也被叫作跨域生成式任务。该任务的模型（跨域生成式模型）一般由无监督或半监督方式训练生成。其技术本质是：通过对抗网络学习跨域间的关系，从而实现图像风格转换。

例如：CycleGAN 模型通过采用循环一致性损失（cycle consistency loss）和跨领域的对抗网络损失，在两个图像域之间训练两个双向的传递模型。这种模型的训练样本不用标注，只需要提供两类统一风格的图片即可，不要求一一对应。

另外还有 DiscoGAN 模型、DualGAN 模型等图像风格转换的优秀模型。读者可以自行研究。

10.1.8 什么是人脸属性编辑任务

人脸属性编辑任务可以将人脸按照指定的属性特征进行转化，例如：变换表情、添加胡子、添加眼镜、添加头帘等。这类任务早先是通过特征点区域像素替换的方法来实现的。这种方法无法做出逼真的效果，常常将替换区域做得很夸张和卡通，可以起到娱乐的效果，所以多用于社交软件中。

随着深度学习的发展，人脸属性编辑的效果变得越来越好，逼真度越来越高。通过特定的模型可以实现以假乱真的效果。

在深度学习中，人脸属性编辑任务可以被归类为图像风格转换任务中的一种。它并不是基于像素的单一替换，而是基于图片特征的深度拟合。

实现人脸属性编辑任务大致有两种方法：基于优化的方法、基于学习的方法。

1. 基于优化方法的人脸属性编辑任务

基于优化方法的人脸属性编辑任务，主要是利用神经网络模型的优化器，通过监督式训练来不断优化节点参数，从而实现人脸图片到目标属性的转化。例如：CNAI、DFI 等方法。

- CNAI 方法是计算人脸图片通过 CNN 模型处理后的特征与待转换的人脸属性特征间的损失值，并按照该损失最小化的方向优化网络模型，从而实现人脸属性编辑。
- DFI 方法是在损失计算过程中加入了欧式距离的测量方法。

这两种方法都需要通过大量次数的迭代训练，且效果相对较差。

2. 基于学习方法的人脸属性编辑任务

基于学习方法的人脸属性编辑任务，主要是通过对抗神经网络学习不同域之间的关系，从而进行转化。它是目前主流的实现方法。

在图像风格转换任务中用到的模型，都可以用来做人脸属性编辑任务。例如：在 CycleGAN 模型中加入重构损失函数，以保证图片内容的一致性（即将人脸中不需要变化的属性保持原样）。

在 CycleGAN 模型之后又出现了 StarGAN 模型。StarGAN 模型是在输入图片中加入了属性控制信息，并改良了判别器模型，在 GAN 网络结构中，除判断输入样本真假外，还对输入样本的属性进行分类。StarGAN 模型可以通过属性控制信息实现用一个模型生成多个属性的效果。

还有效果更好的 AttGAN 模型。该模型在生成器模型部分嵌入了自编码网络模型（编解码器模型架构）。这样的模型可以更深层次地拟合原数据中潜在特征和属性的关系，从而使得生成的效果更加逼真。在本书 10.3 节将介绍 AttGAN 模型的具体实现。

10.1.9 什么是 TFGan 框架

TFGan 框架是 TensorFlow 中封装好的对抗网络集成框架。整个框架的使用方法与估算器的使用方法非常相似。

用 TFGan 框架可以很方便地开发出 GAN 模型，但需要一定的学习成本。对估算器框架不熟悉的读者，不建议直接上手使用该框架。

更多实例可以参考如下链接：

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

10.2 实例 54：构建 DeblurGAN 模型，将模糊相片变清晰

在拍照时，常常因为手抖或补光不足，导致拍出的照片很模糊。可以用 DeblurGAN 模型将模糊的照片变清晰，留住精彩瞬间。

DeblurGAN 模型是一个对抗神经网络模型，由生成器模型和判别器模型组成。

- 生成器模型，根据输入的模糊图片模拟生成清晰的图片。
- 判别器模型，用在训练过程中，帮助生成器模型达到更好的效果。具体可以参考论文：
<https://arxiv.org/pdf/1711.07064.pdf>。

实例描述

有一套街景拍摄的照片数据集，其中包含清晰照片和模糊照片。

要求：

- (1) 用该数据集训练 DeblurGAN 模型，使模型具有将模糊图片转成清晰图片的能力。
- (2) DeblurGAN 模型能将数据集之外的模糊照片变清晰。

本实例的代码用 tf.keras 接口编写。具体过程如下。

10.2.1 获取样本

本实例使用 GOPRO_Large 数据集作为训练样本。GOPRO_Large 数据集里包含高帧相机拍摄的街景图片（其中的照片有的清晰，有的模糊）和人工合成的模糊照片。样本中每张照片的尺寸为 720 pixel × 1280 pixel。

1. 下载 GOPRO_Large 数据集

可以通过以下链接获取原始的 GOPRO_Large 数据集：

https://drive.google.com/file/d/1H0PIXvJH4c40pk7ou6nAwoxuR4Qh_Sa2/view

2. 部署 GOPRO_Large 数据集

在 GOPRO_Large 数据集中有若干套实景拍摄的照片。每套照片中包含有 3 个文件夹：

- 在 blur 文件夹中，放置了模糊的照片。
- 在 sharp 文件夹中，放置了清晰的照片。
- 在 blur_gamma 文件夹中，放置了人工合成的模糊照片

从 GOPRO_Large 数据集的 blur 与 sharp 文件夹里，各取出 200 张模糊与清晰的图片，放到本地代码的同级目录 image 文件夹下用作训练。其中，模糊的图片放在 image/train/A 文件夹下，清晰的图片在 image/train/B 文件夹下。

10.2.2 准备 SwitchableNorm 算法模块

SwitchableNorm 算法与其他的归一化算法一样，可以被当作函数来使用。由于在当前的 API 库里没有该代码的实现，所以需要自己编写一套这样的算法。

SwitchableNorm 算法的实现不是本节重点，其原理已经在 10.1.6 小节介绍。这里直接使用本书中的配套资源代码“switchnorm.py”即可。

直接将该代码放到本地代码文件夹下，然后将其引入。



提示：

在 SwitchableNorm 算法的实现过程中，定义了额外的变量参数。所以在运行时，需要通过会话中的 `tf.global_variables_initializer` 函数对其进行初始化，否则会报“SwitchableNorm 类中的某些张量没有初始化”之类的错误。正确的用法见 10.2.9 小节的具体实现。

10.2.3 代码实现：构建 DeblurGAN 中的生成器模型

DeblurGAN 中的生成器模型是使用残差结构来实现的。其模型的层次结构顺序如下：

- (1) 通过 1 层卷积核为 7×7 、步长为 1 的卷积变换。保持输入数据的尺寸不变。
- (2) 将第 (1) 步的结果进行两次卷积核为 3×3 、步长为 2 的卷积操作，实现两次下采样效果。
- (3) 经过 5 层残差块。其中，残差块是中间带有 Dropout 层的两次卷积操作。
- (4) 仿照 (1) 和 (2) 步的逆操作，进行两次上采样，再来一个卷积操作。
- (5) 将 (1) 的输入与 (4) 的输出加在一起，完成一次残差操作。

该结构使用“先下采样，后上采样”的卷积处理方式，这种方式可以表现出样本分布中更好的潜在特征。具体代码如下：

代码 10-1 deblurnet

```

01 from tensorflow.keras import layers as KL
02 from tensorflow.keras import models as KM
03 from switchnorm import SwitchNormalization #载入 SwitchableNorm 算法
04 ngf = 64 #定义生成器模型原始卷积核个数
05 ndf = 64 #定义判别器模型原始卷积核个数
06 input_nc = 3 #定义输入通道
07 output_nc = 3 #定义输出通道
08 n_blocks_gen = 9 #定义残差层数量
09
10 #定义残差块函数
11 def res_block(input, filters, kernel_size=(3, 3), strides=(1, 1),
12     use_dropout=False):
13     x = KL.Conv2D(filters=filters, #使用步长为 1 的卷积操作，保持输入数据的尺寸不变
14     kernel_size=kernel_size,
15     strides=strides, padding='same')(input)

```

```

15
16) (x = KL.SwitchNormalization()(x)
17) x = KL.Activation('relu')(x)
18) x = KL.SwitchNormalization()(x)
19) if use_dropout: #使用 dropout 方法
20)     x = KL.Dropout(0.5)(x)
21) x = KL.Conv2D(filters=filters, #再做一次步长为 1 的卷积操作
22)             kernel_size=kernel_size,
23)             strides=strides, padding='same')(x)
24)
25)
26) x = KL.SwitchNormalization()(x)
27) #将卷积后的结果与原始输入相加
28) merged = KL.Add()([input, x]) #残差层
29)
30) return merged
31)
32) def generator_model(image_shape, istrain=True): #构建生成器模型
33)     #构建输入层（与动态图不兼容）
34)     inputs = KL.Input(shape=(image_shape[0], image_shape[1], input_nc))
35)     #使用步长为 1 的卷积操作，保持输入数据的尺寸不变
36)     x = KL.Conv2D(filters=ngf, kernel_size=(7, 7), padding='same')(inputs)
37)     x = KL.SwitchNormalization()(x)
38)     x = KL.Activation('relu')(x)
39)
40)     n_downsampling = 2
41)     for i in range(n_downsampling): #两次下采样
42)         mult = 2**i
43)         x = KL.Conv2D(filters=ngf*mult*2, kernel_size=(3, 3), strides=2,
44)                     padding='same')(x)
45)         x = KL.SwitchNormalization()(x)
46)         x = KL.Activation('relu')(x)
47)         mult = 2**n_downsampling
48)         for i in range(n_blocks_gen): #定义多个残差层
49)             x = res_block(x, ngf*mult, use_dropout=istrain)
50)
51)         for i in range(n_downsampling): #两次上采样
52)             mult = 2**(n_downsampling - i)
53)             #x = KL.Conv2DTranspose(filters=int(ngf * mult / 2), kernel_size=(3,
54)             3), strides=2, padding='same')(x)
55)             x = KL.UpSampling2D()(x)
56)             x = KL.Conv2D(filters=int(ngf * mult / 2), kernel_size=(3, 3),
57)                         padding='same')(x)
58)             x = KL.SwitchNormalization()(x)
59)             x = KL.Activation('relu')(x)
60)
61)     #定义函数 save_all_weight，将模型中的参数保存起来
62)     (2) 定义函数 save_all_weight
63)     (3) 定义函数 save_all_weight，将模型中的参数保存起来

```

```

59     #步长为1的卷积操作
60     x = KL.Conv2D(filters=output_nc, kernel_size=(7, 7), padding='same')(x)
61     x = KL.Activation('tanh')(x)
62
63     outputs = KL.Add()([x, inputs])      #与最外层的输入完成一次大残差
64     #防止特征值域过大，进行除2操作（取平均数残差）
65     outputs = KL.Lambda(lambda z: z/2)(outputs)
66     #构建模型
67     model = KM.Model(inputs=inputs, outputs=outputs, name='Generator')
68     return model

```

代码第 11 行，通过定义函数 `res_block` 搭建残差块的结构。

代码第 32 行，通过定义函数 `generator_model` 构建生成器模型。由于生成器模型输入的是模糊图片，输出的是清晰图片，所以函数 `generator_model` 的输入与输出具有相同的尺寸。

代码第 65 行，在使用残差操作时，将输入的数据与生成的数据一起取平均值。这样做是为了防止生成器模型的返回值的值域过大。在计算损失时，一旦生成的数据与真实图片的像素数据值域不同，则会影响收敛效果。

10.2.4 代码实现：构建 DeblurGAN 中的判别器模型

判别器模型的结构相对比较简单。

- (1) 通过 4 次下采样卷积（见代码第 74~82 行），将输入数据的尺寸变小。
- (2) 经过两次尺寸不变的 1×1 卷积（见代码第 85~92 行），将通道压缩。
- (3) 经过两层全连接网络（见代码第 95~97 行），生成判别结果（0 还是 1）。

具体代码如下。

代码 10-1 deblurnodel（续）

```

69 def discriminator_model(image_shape):#构建判别器模型
70
71     n_layers, use_sigmoid = 3, False
72     inputs = KL.Input(shape=(image_shape[0],image_shape[1],output_nc))
73     #下采样卷积
74     x = KL.Conv2D(filters=ndf, kernel_size=(4, 4), strides=2,
75     padding='same')(inputs)
76     x = KL.LeakyReLU(0.2)(x)
77     nf_mult, nf_mult_prev = 1, 1
78     for n in range(n_layers):#继续3次下采样卷积
79         nf_mult_prev, nf_mult = nf_mult, min(2**n, 8)
80         x = KL.Conv2D(filters=ndf*nf_mult, kernel_size=(4, 4), strides=2,
81         padding='same')(x)
82         x = KL.BatchNormalization()(x)
83         x = KL.LeakyReLU(0.2)(x)
84     #步长为1的卷积操作，尺寸不变

```

```

85     nf_mult_prev, nf_mult = nf_mult, min(2**n_layers, 8)
86     x = KL.Conv2D(filters=ndf*nf_mult, kernel_size=(4, 4), strides=1,
87                     padding='same')(x)
88     x = KL.BatchNormalization()(x)
89     x = KL.LeakyReLU(0.2)(x)
90
91     #步长为 1 的卷积操作，尺寸不变。将通道压缩为 1
92     x = KL.Conv2D(filters=1, kernel_size=(4, 4), strides=1,
93                     padding='same')(x)
94     if use_sigmoid:
95         x = KL.Activation('sigmoid')(x)
96
97     x = KL.Flatten()(x) #两层全连接，输出判别结果
98
99     x = KL.Dense(1024, activation='tanh')(x)
100    x = KL.Dense(1, activation='sigmoid')(x)
101
102    model = KM.Model(inputs=inputs, outputs=x, name='Discriminator')
103
104    return model

```

代码 81 行，调用了批量归一化函数，使用了参数 trainable 的默认值 True。

代码 99 行，用 tf.keras 接口的 Model 类构造判别器模型 model。在使用 model 时，可以设置 trainable 参数来控制模型的内部结构。

10.2.5 代码实现：搭建 DeblurGAN 的完整结构

将判别器模型与生成器模型结合起来，构成 DeblurGAN 模型的完整结构。具体代码如下：

代码 10-1 deblurnodel（续）

```

101 def g_containing_d_multiple_outputs(generator,
102     discriminator,image_shape):
103     inputs = KL.Input(shape=(image_shape[0],image_shape[1],input_nc))
104     generated_image = generator(inputs) #调用生成器模型
105     outputs = discriminator(generated_image) #调用判别器模型
106     #构建模型
107     model = KM.Model(inputs=inputs, outputs=[generated_image, outputs])
108
109     return model

```

函数 g_containing_d_multiple_outputs 用于训练生成器模型。在使用时，需要将判别器模型的权重固定，让生成器模型不断地调整权重。具体可以参考 10.2.10 小节代码。

10.2.6 代码实现：引入库文件，定义模型参数

编写代码实现如下步骤：

- (1) 载入模型文件——代码文件“10-1 deblurnodel”。
- (2) 定义训练参数。
- (3) 定义函数 save_all_weights，将模型的权重保存起来。

具体代码如下：

代码 10-2 训练 deblur

```

01 import os
02 import datetime
03 import numpy as np
04 import tqdm
05 import tensorflow as tf
06 import glob
07 from tensorflow.python.keras.applications.vgg16 import VGG16
08 from functools import partial
09 from tensorflow.keras import models as KM
10 from tensorflow.keras import backend as K      #载入Keras的后端实现
11 deblurmodel = __import__("10-1_deblurmodel") #载入模型文件
12 generator_model = deblurmodel.generator_model
13 discriminator_model = deblurmodel.discriminator_model
14 g_containing_d_multiple_outputs =
15     deblurmodel.g_containing_d_multiple_outputs
16 RESHAPE = (360,640)      #定义处理图片的大小
17 epoch_num = 500          #定义迭代训练次数
18
19 batch_size = 4           #定义批次大小
20 critic_updates = 5       #定义每训练一次生成器模型需要训练判别器模型的次数
21 #保存模型
22 BASE_DIR = 'weights/'
23 def save_all_weights(d, g, epoch_number, current_loss):
24     now = datetime.datetime.now()
25     save_dir = os.path.join(BASE_DIR, '{}_{}'.format(now.month, now.day))
26     os.makedirs(save_dir, exist_ok=True)      #创建目录
27     g.save_weights(os.path.join(save_dir,
28         'generator_{}_{}.h5'.format(epoch_number, current_loss)), True)
29     d.save_weights(os.path.join(save_dir,
30         'discriminator_{}.h5'.format(epoch_number)), True)

```

代码第 16 行将输入图片的尺寸设为 (360,640)，使其与样本中图片的高、宽比例相对应（样本中图片的尺寸比例为 720：1280）。



提示：

在 TensorFlow 中，默认的图片尺寸顺序是“高”在前，“宽”在后。

10.2.7 代码实现：定义数据集，构建正反向模型

本小节代码的步骤如下：

- (1) 用 `tf.data.Dataset` 接口完成样本图片的载入（见代码第 29~54 行）。

- (2) 将生成器模型和判别器模型搭建起来。
- (3) 构建 Adam 优化器，用于生成器模型和判别器模型的训练过程。
- (4) 以 WGAN 的方式定义损失函数 wasserstein_loss，用于计算生成器模型和判别器模型的损失值。其中，生成器模型的损失值是由 WGAN 损失与特征空间损失（见 10.2.8 小节）两部分组成。
- (5) 将损失函数 wasserstein_loss 与优化器一起编译到可训练的判别器模型中（见代码第 70 行）。

具体代码如下：

代码 10-2 训练 deblur (续)

```

29 path = r'./image/train'
30 A_paths, =os.path.join(path, 'A', "*.png") # 定义样本路径
31 B_paths = os.path.join(path, 'B', "*.png")
32 # 获取该路径下的 png 文件
33 A_fnames, B_fnames = glob.glob(A_paths), glob.glob(B_paths)
34 # 生成 Dataset 对象
35 dataset = tf.data.Dataset.from_tensor_slices((A_fnames, B_fnames))
36
37 def _processimg(imgname): # 定义函数调整图片大小
38     image_string = tf.read_file(imgname) # 读取整个文件
39     image_decoded = tf.image.decode_image(image_string)
40     image_decoded.set_shape([None, None, None]) # 形状变化，否则下面会转化失败
41     # 变化尺寸
42     img = tf.image.resize(image_decoded, RESHAPE)
43     image_decoded = (img - 127.5) / 127.5
44     return image_decoded
45
46 def _parseone(A_fname, B_fname): # 解析一个图片文件
47     # 读取并预处理图片
48     image_A, image_B = _processimg(A_fname), _processimg(B_fname)
49     return image_A, image_B
50
51 dataset = dataset.shuffle(buffer_size=len(B_fnames)) # 转化为有图片内容的数据集
52 dataset = dataset.map(_parseone) # 将数据集按照 batch_size 划分
53 dataset = dataset.batch(batch_size)
54 dataset = dataset.prefetch(1)
55
56 # 定义模型
57 g = generator_model(RESHAPE) # 生成器模型
58 d = discriminator_model(RESHAPE) # 判别器模型
59 d_on_g = g_containing_d_multiple_outputs(g, d, RESHAPE) # 联合模型
60
61 # 定义优化器

```

```

62 d_opt = tf.keras.optimizers.Adam(lr=1E-4, beta_1=0.9, beta_2=0.999,
63     epsilon=1e-08)
64 d_on_g_opt = tf.keras.optimizers.Adam(lr=1E-4, beta_1=0.9, beta_2=0.999,
65     epsilon=1e-08)
66 #WGAN 的损失
67 def wasserstein_loss(y_true, y_pred):
68     return tf.reduce_mean(y_true*y_pred)
69 d.trainable = True
70 d.compile(optimizer=d_opt, loss=wasserstein_loss) #编译模型
71 d.trainable = False

```

代码第 70 行，用判别器模型对象的 `compile` 方法对模型进行编译。之后，将该模型的权重设置成不可训练（见代码第 71 行）。这是因为，在训练生成器模型时，需要将判别器模型的权重固定。只有这样，在训练生成器模型过程中才不会影响到判别器模型。

10.2.8 代码实现：计算特征空间损失，并将其编译到生成器模型的训练模型中

生成器模型的损失值是由 WGAN 损失与特征空间损失两部分组成。WGAN 损失已经由 10.2.7 小节的第 66 行代码实现。本小节将实现特征空间损失，并将其编译到可训练的生成器模型中去。

1. 计算特征空间损失的方法

计算特征空间损失的方法如下：

- (1) 用 VGG 模型对目标图片与输出图片做特征提取，得到两个特征数据。
- (2) 对这两个特征数据做平方差计算。

2. 特征空间损失的具体实现

在计算特征空间损失时，需要将 VGG 模型嵌入到当前网络中。这里使用已经下载好的预训练模型文件“vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5”。读者可以自行下载，也可以在本书配套资源中找到。

将预训练模型文件放在当前代码的同级目录下。并参照本书 6.7.9 小节的内容，利用 `tf.keras` 接口将其加载。

另外，在《深度学习之 TensorFlow——入门、原理与进阶实战》一书的 12.8 节的 SRGAN 模型中也讲过一个使用 TF-slim 接口计算特征空间损失的方法。有兴趣的读者可以参考那部分内容。

3. 编译生成器模型的训练模型

将 WGAN 损失函数与特征空间损失函数放到数组 `loss` 中，调用生成器模型的 `compile` 方法将损失值数组 `loss` 编译进去，实现生成器模型的训练模型。

具体代码如下：

代码 10-2 训练 deblur (续)

```

72 #计算特征空间损失
73 def perceptual_loss(y_true, y_pred,image_shape):
74     vgg = VGG16(include_top=False,
75      weights="vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5",
76      input_shape=(image_shape[0],image_shape[1],3))
77
78     loss_model = KM.Model(inputs=vgg.input,
79                           outputs=vgg.get_layer('block3_conv3').output)
80     loss_model.trainable = False
81     return tf.reduce_mean(tf.square(loss_model(y_true) -
82                                     loss_model(y_pred)))
83
84 myperceptual_loss = partial(perceptual_loss, image_shape=RESHAPE)
85
86 #构建损失
87 loss = [myperceptual_loss, wasserstein_loss]
88 loss_weights = [100, 1] #将损失值调为统一数量级
89 d_on_g.compile(optimizer=d_on_g_opt, loss=loss,
90                  loss_weights=loss_weights)
91 d.trainable = True
92
93 output_true_batch, output_false_batch = np.ones((batch_size, 1)),
94 -np.ones((batch_size, 1))
95
96 #生成数据集迭代器
97 iterator = dataset.make_initializable_iterator()
98 datatensor = iterator.get_next()

```

代码第 85 行，在计算生成器模型损失时，将损失值函数 myperceptual_loss 与损失值函数 wasserstein_loss 一起放到列表里。

代码第 86 行，定义了损失值的权重比例 [100,1]。这表示最终的损失值是：函数 myperceptual_loss 的结果乘上 100，将该积与函数 wasserstein_loss 的结果相加所得到和。



提示：

权重比例是根据每个函数返回的损失值得来的。

将 myperceptual_loss 的结果乘上 100，是为了让最终的损失值与函数 wasserstein_loss 的结果在同一个数量级上。

损失值函数 myperceptual_loss、wasserstein_loss 分别与模型 d_on_g 对象的输出值 generated_image、outputs 相对应。模型 d_on_g 对象的输出节点部分是在 10.2.5 小节代码第 106 行定义的。

在权重 weights 文件夹里找到以 “.meta” 为后缀的文件，即为生成时的生成时间序的文件。将其复制到本地路径下（作者本地的文件名为 “generator_400_0.h5”）。这个

10.2.9 代码实现：按指定次数训练模型

按照指定次数迭代调用训练函数 `pre_train_epoch`，然后在函数 `pre_train_epoch` 内遍历整个 `Dataset` 数据集，并进行训练。步骤如下：

- (1) 取一批次数据。
- (2) 训练 5 次判别器模型。
- (3) 将判别器模型权重固定，训练一次生成器模型。
- (4) 将判别器模型设为可训练，并循环第(1)步，直到整个数据集遍历结束。

具体代码如下：

代码 10-2 训练 deblur (续)

```

95 # 定义配置文件
96 config = tf.ConfigProto()
97 config.gpu_options.allow_growth = True
98 config.gpu_options.per_process_gpu_memory_fraction = 0.5
99 sess = tf.Session(config=config)          # 建立会话 (session)
100
101 def pre_train_epoch(sess, iterator, datatensor): # 迭代整个数据集进行训练
102     d_losses = []
103     d_on_g_losses = []
104     sess.run(iterator.initializer)
105
106     while True:
107         try:                                # 获取一批次的数据
108             (image.blur_batch, image.full_batch) = sess.run(datatensor)
109         except tf.errors.OutOfRangeError:      # 如果数据取完则退出循环
110             break
111
112         generated_images = g.predict(x=image.blur_batch,
113                                         batch_size=batch_size)          # 将模糊图片输入生成器模型
114
115         for _ in range(critic_updates):        # 训练 5 次判别器模型
116             d_loss_real = d.train_on_batch(image.full_batch,
117                                             output_true_batch)            # 训练，并计算还原样本的 loss 值
118
119             d_loss_fake = d.train_on_batch(generated_images,
120                                             output_false_batch)           # 训练，并计算模拟样本的 loss 值
121             d_loss = 0.5 * np.add(d_loss_fake, d_loss_real) # 二者相加，再除以 2
122             d_losses.append(d_loss)
123
124             d.trainable = False               # 固定判别器模型参数
125             d_on_g_loss = d_on_g.train_on_batch(image.blur_batch,
126                                                 [image.full_batch, output_true_batch])  # 训练并计算生成器模型 loss 值
127             d_on_g_losses.append(d_on_g_loss)

```

```

125     d.trainable = True          #恢复判别器模型参数可训练的属性
126     if len(d_on_g_losses)%10== 0:
127         print(len(d_on_g_losses),np.mean(d_losses),
128             np.mean(d_on_g_losses))
128     return np.mean(d_losses), np.mean(d_on_g_losses)
129 #初始化 SwitchableNorm 变量
130 K.get_session().run(tf.global_variables_initializer())
131 for epoch in tqdm.tqdm(range(epoch_num)):
132     #迭代训练一次数据集
133     dloss,gloss = pre_train_epoch(sess, iterator, datatensor)
134     with open('log.txt', 'a+') as f:
135         f.write('{} - {} - {}\n'.format(epoch, dloss, gloss))
136     save_all_weights(d, g, epoch, int(gloss))  #保存模型
137 sess.close()                  #关闭会话

```

代码第130行，进行全局变量的初始化。初始化之后，SwitchableNorm算法就可以正常使用了。



提示：

即便是tf.keras接口，其底层也是通过静态图上的会话（session）来运行代码的。

在代码第130行中演示了一个用tf.keras接口实现全局变量初始化的技巧：

- (1) 用tf.keras接口的后端类backend中的get_session函数，获取tf.keras接口当前正在使用的会话（session）。
- (2) 拿到session之后，运行tf.global_variables_initializer方法进行全局变量的初始化。
- (3) 代码运行后，输出如下结果：

```

1%| 6/50 [15:06<20:43:45, 151.06s/it]10 -0.4999978220462799 678.8936
20 -0.4999967348575592 680.67926
.....
1%| 7/50 [17:29<20:32:16, 149.97s/it]10 -0.49999643564224244 737.67645
20 -0.49999758243560793 700.6202
30 -0.4999980672200521 672.0518
40 -0.49999826729297636 666.23425
50 -0.4999982775449753 665.67645
.....

```

同时可以看到，在本地目录下生成了一个weights文件夹，里面放置的便是模型文件。

10.2.10 代码实现：用模型将模糊相片变清晰

在权重weights文件夹里找到以“generator”开头并且是最新生成（按照文件的生成时间排序）的文件。将其复制到本地路径下（作者本地的文件名称为“generator_499_0.h5”）。这个

模型就是 DeblurGAN 中的生成器模型部分。

按照 10.2.1 小节的步骤，在测试集中随机复制几个图片放到本地 test 目录下。与 train 目录结构一样：A 放置模糊的图片，B 放置清晰的图片。

下面编写代码来比较模型还原的效果。具体如下：

代码 10-3 使用 deblur 模型

```

01 import numpy as np
02 from PIL import Image
03 import glob
04 import os
05 import tensorflow as tf
06 deblurmodel = __import__("10-1_deblurmodel")
07 generator_model = deblurmodel.generator_model
08
09 def deprocess_image(img): # 定义图片的后处理函数
10     img = img * 127.5 + 127.5
11     return img.astype('uint8')
12
13 batch_size = 4
14 RESHAPE = (360, 640) # 定义要处理图片的大小
15
16 path = r'./image/test'
17 A_paths, B_paths = os.path.join(path, 'A', "*.png"), os.path.join(path, 'B',
18     "*.png")
19 # 获取该路径下的 png 文件
20 A_fnames, B_fnames = glob.glob(A_paths), glob.glob(B_paths)
21 # 生成 Dataset 对象
22 dataset = tf.data.Dataset.from_tensor_slices((A_fnames, B_fnames))
23
24 def _processimg(imgname): # 定义函数调整图片大小
25     image_string = tf.read_file(imgname) # 读取整个文件
26     image_decoded = tf.image.decode_image(image_string)
27     image_decoded.set_shape([None, None, None]) # 形状变化，否则下面会转化失败
28     # 变化尺寸
29     img = tf.image.resize(image_decoded, RESHAPE) # [RESHAPE[0], RESHAPE[1], 3]
30     image_decoded = (img - 127.5) / 127.5
31     return image_decoded
32
33 def _parseone(A_fname, B_fname): # 解析一个图片文件
34     # 读取并预处理图片
35     image_A, image_B = _processimg(A_fname), _processimg(B_fname)
36     return image_A, image_B
37
38 dataset = dataset.map(_parseone) # 转化为有图片内容的数据集
39 dataset = dataset.batch(batch_size) # 将数据集按照 batch_size 划分
40 dataset = dataset.prefetch(1)

```

```

40
41 #生成数据集迭代器
42 iterator = dataset.make_initializable_iterator()
43 datatensor = iterator.get_next()
44 g = generator_model(RESHAPE, False)           #构建生成器模型
45 g.load_weights("generator_499_0.h5")          #载入模型文件
46
47 #定义配置文件
48 config = tf.ConfigProto()
49 config.gpu_options.allow_growth = True
50 config.gpu_options.per_process_gpu_memory_fraction = 0.5
51 sess = tf.Session(config=config)               #建立 session
52 sess.run(iterator.initializer)
53 ii= 0
54 while True:
55     try:                                     #获取一批次的数据
56         (x_test,y_test) = sess.run(datatensor)
57     except tf.errors.OutOfRangeError:          #如果数据取完则退出循环
58         break
59     generated_images = g.predict(x=x_test, batch_size=batch_size)
60     generated = np.array([deprocess_image(img) for img in generated_images])
61     x_test = deprocess_image(x_test)
62     y_test = deprocess_image(y_test)
63     print(generated_images.shape[0])
64     for i in range(generated_images.shape[0]):  #按照批次读取结果
65         y = y_test[i, :, :, :]
66         x = x_test[i, :, :, :]
67         img = generated[i, :, :, :]
68         output = np.concatenate((y, x, img), axis=1)
69         im = Image.fromarray(output.astype(np.uint8))
70         im = im.resize( (640*3, int(640*720/1280) ) )
71         print('results{}{}.png'.format(ii,i))
72         im.save('results{}{}.png'.format(ii,i)) #将结果保存起来
73     ii+=1

```

代码第 44 行，在定义生成器模型时，需要将其第 2 个参数 `istrain` 设为 `False`。这么做的目的是不使用 `Dropout` 层。

代码执行后，系统会自动在本地文件夹的 `image/test` 目录下加载图片，并其放到模型里进行清晰化处理。最终生成的图片如图 10-1 所示。



图 10-1 DeblurGAN 的处理结果

图 10-1 中有 3 个子图。左、中、右依次为原始、模糊、生成后的图片。比较图 10-1 中的原始图片（最左侧的图片）与生成后的图片（最右侧的图片）可以发现，最右侧模型生成的图片比中间的模糊图片更为清晰。

10.2.11 练习题

如果生成器模型使用普通的归一化算法，会是什么效果？并改写代码实验一下。

答案：将 10.2.3 小节所有的 KL.SwitchNormalization 代码都替换成 KL.BatchNormalization 代码，并重新训练模型。所生成的图像如图 10-2 所示。



图 10-2 普通归一化的处理结果

用 SwitchableNorm 归一化处理的结果如图 10-3 所示。



图 10-3 使用 SwitchableNorm 归一化处理的结果

比较图 10-2 与图 10-3 中最右边的图片可以看出，图 10-2 中最右侧图片的顶部出现了一些噪声，而图 10-3 中生成的图像（最右侧的图）质量更好（消除了噪声）。因为是黑白印刷，所以效果并不太明显。

10.2.12 扩展：DeblurGAN 模型的更多妙用

DeblurGAN 模型可以提升照片的清晰度。这是一个很有商业价值的功能。

例如，在开发智能冰箱、智能冰柜项目中，用户从冰柜里拿取商品时，一般需要通过高速相机在短时间内连续拍照，并挑选出高质量的图片送入后面的 YOLO 模型进行识别。如果应用的是 DeblurGAN 模型，则可以用相对便宜的相机来替代高速相机，而 YOLO 模型的识别率又不会有太大的损失。这个方案可以大大节省硬件成本。该方案在 14.5 节还有详细描述。

另外，DeblurGAN 模型的网络结构没有将输入图片的尺寸与权重参数紧耦合，它可以处理不同尺寸的图片（请试着随意修改 10.2.9 小节代码第 14 行的尺寸值，程序仍可以正常运行）。所以说，DeblurGAN 模型应用起来更加灵活。

10.3 实例 55：构建 AttGAN 模型，对照片进行加胡子、加头帘、加眼镜、变年轻等修改

将自编码网络模型与对抗神经网络模型结合起来，通过重建学习和对抗性学习的训练方式，融合人脸的潜在特征与指定属性，生成带有指定属性特征的人脸图片。

实例描述

用 CelebA 数据集训练 AttGAN 模型。使模型能够对照片中的人物进行修改，实现为照片中的人物添加胡子、添加头帘、添加眼袋、添加眼镜、年轻化处理等 40 项属性的处理。

10.3.1 获取样本

CelebA 数据集是一个人脸数据集，其中包括人脸图片与人脸属性的标注信息。下载方法可以参考本书的 5.2.1 小节。

1. 部署样本数据

下载完 CelebA 数据集后，将其中的对齐图片数据与标注数据提取出来，用于训练。具体操作如下：

- (1) 在代码的本地文件夹下新建一个目录 data。
- (2) 将 CelebA\Img 下的 img_align_celeba.zip 解压缩，得到 img_align_celeba 文件夹，并将该文件夹放在 data 目录下。
- (3) 将 CelebA\Anno 下的 list_attr_celeba.txt 也放到 data 目录下。

2. 介绍样本的标注信息

CelebA 数据集中的标注文件 list_attr_celeba.txt 记录了每张人脸图片的多个属性特征。在标注文件 list_attr_celeba.txt 中，将人脸属性划分成了 40 个属性标签。如果图片中的人脸符合某个属性标签，则在该属性标签的位置上赋值 1，否则在该属性的标签上赋值 -1。

这 40 种人脸属性的内容如下：

'当天的小胡茬': 0,	'拱形眉毛': 1,	'漂亮': 2,	'眼袋': 3,	'没头发': 4,
'头帘': 5,	'大嘴唇': 6,	'大鼻子': 7,	'黑发': 8,	'金发': 9,
'图片模糊': 10,	'棕色头发': 11,	'浓眉毛': 12,	'胖乎乎': 13,	'双下巴': 14,
'眼镜': 15,	'山羊胡子': 16,	'灰发': 17,	'重妆': 18,	'高颧骨': 19,
'男': 20,	'嘴微微开': 21,	'小胡子': 22,	'细眼睛': 23,	'没胡子': 24,
'椭圆形脸': 25,	'苍白皮肤': 26,	'尖鼻子': 27,	'退缩发际线': 28,	'玫瑰色脸颊': 29,
'连鬓胡子': 30,	'微笑': 31,	'直发': 32,	'波浪发': 33,	'佩戴耳环': 34,
'戴帽子': 35,	'涂口红': 36,	'戴项链': 37,	'打领带': 38,	'年轻': 39

这里的标签标注并不是 one-hot 分类，人脸图片与这 40 个属性标签是多对多的关系，即一个图片可以被打上多个属性的分类标签，如图 10-4 所示。

```

1202599
25_o_Clock_Shadow Arched_Eyebrows Attractive Bags_Under_Eyes Bald Bangs Big_Lips Big_Nose Black_Hair Blond_Hair
3000001.jpg -1 1 1 -1 -1 -1 -1 -1 -1 1 -1 1 -1 -1 -1 1 1 -1 1 -1 -1 1 -1 1 -1 -1 -1 1 -1 1 -1 -1 -1 1 -1
4000002.jpg -1 -1 -1 1 -1 -1 -1 -1 -1 1 -1 -1 -1 -1 -1 1 -1 1 -1 1 -1 -1 1 -1 -1 -1 -1 -1 1 -1 -1 -1 -1 -1
5000003.jpg -1 -1 -1 -1 -1 -1 1 -1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
6000004.jpg -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
7000005.jpg -1 1 1 -1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
8000006.jpg -1 1 1 -1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
9000007.jpg 1 -1 1 1 -1 -1 1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
1000008.jpg 1 1 -1 1 -1 -1 1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
1100009.jpg -1 1 1 -1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
1200010.jpg -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
1300011.jpg -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
1400012.jpg -1 -1 1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
1500013.jpg -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

```

图 10-4 CelebA 的标注数据

从图 10-4 中可以看出，标注文件的内容主要分为 3 种数据：

- 第 1 行是总共标注的条数。
- 第 2 行是这 40 种属性的英文标签。
- 第 3 行及以下行是每个图片对应的标签，表明该图片具体带有哪个属性（1 表示具有该属性，-1 表示没有该属性）。

10.3.2 了解 AttGAN 模型的结构

AttGAN 模型属于对抗神经网络模型框架下的多模型结构。它在对抗神经网络模型框架基础上，将单一的生成器模型换成一个自编码网络模型。其整体结构描述如下。

- 生成器模型：由一个自编码网络模型构成。用自编码模型中的编码器模型来提取人脸主要潜在特征，用自编码模型中的解码器模型来生成指定属性的人脸图像。
- 判别器模型：起到约束解码器模型的作用，让解码器模型生成具有指定特征属性的人脸图像。

AttGAN 模型的完整结构如图 10-5 所示。

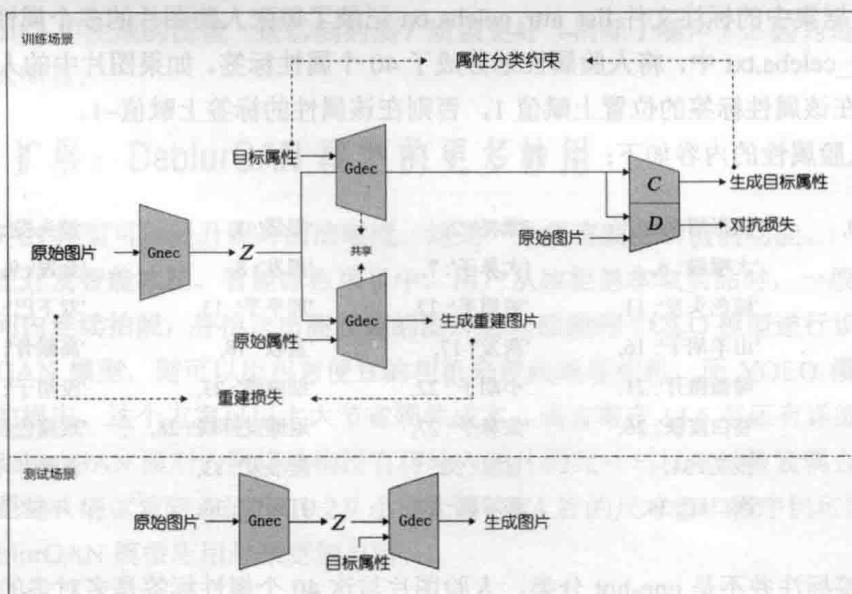


图 10-5 AttGAN 模型的完整结构

在图 10-5 中描述了 AttGAN 模型在两个场景下的完整结构：训练（Train）场景与测试（Test）场景。

- **训练场景：**体现了 AttGAN 模型的完整结构。在训练自编码模型的解码器模型时，将重建过程的损失值和对抗网络模型的损失值作为整个网络模型的损失值。该损失值将参与迭代训练过程中的反向传播过程。
- **测试场景：**直接用训练好的自编码模型生成人脸图片，不再需要对抗神经网络模型中的判别器模型部分。

1. 训练场景中模型的组成及作用

在训练场景中，模型由 3 个子模型组成：编码器模型（Genc）、解码器模型（Gdec）、判别器模型（CD）。具体描述如下。

- **编码器模型（Genc）：**将真实图片压缩成特征向量 Z 。
- **解码器模型（Gdec）：**使用了两种训练方式。一种训练方式是将样本图片与原始标签 a 组合作为输入，重建出原始图片；另一种训练方式是将样本图片与随机制作的标签 b 组合作为输入，重建出带有标签 b 中特征的图片。
- **判别器模型（CD）：**输出了两种结果。一种是分类结果（ C ），代表图片中人脸的属性；另一种是判断真伪的结果（ D ），用来区分输入是真实图片，还是生成的图片。

在 AttGAN 模型中，生成器模型的随机值并不是产生照片的随机数，而是根据原始标签变化后的标签值。照片数据在模型中只是起到重建作用。因为在人脸编辑任务中，不希望对属性之外的图像发生变化，所以重建损失可以最大化地保证个体数据原有的样子。

2. 测试场景中模型的组成及作用

在测试场景中，AttGAN 模型由两个子模型组成：

- (1) 利用编码器模型将图片特征提取出来。
- (2) 将提取的特征与指定的属性值参数一起输入编码器模型中，合成出最终的人脸图片。

更多细节可以参考论文：<https://arxiv.org/pdf/1711.10678.pdf>。

10.3.3 代码实现：实现支持动态图和静态图的数据集工具类

编写数据集工具类，对 `tf.data.Dataset` 接口进行二次封装，使其可以兼容动态图与静态图。代码如下：

代码 10-4 mydataset

```

01 import os
02 import numpy as np
03 import tensorflow as tf
04 import tensorflow.contrib.eager as tfe
05
06 class Dataset(object):          # 定义数据集类，支持动态图和静态图
07     def __init__(self):
08         self._dataset = None

```

```

09         self._iterator = None
10         self._batch_op = None
11         self._sess = None
12         self._is_eager = tf.executing_eagerly()
13         self._eager_iterator = None
14
15     def __del__(self):          #重载 del 方法
16         if self._sess:          #在静态图中，在销毁对象时需要关闭 session
17             self._sess.close()
18
19     def __iter__(self):        #重载迭代器方法
20         return self
21
22     def __next__(self):        #重载 next 方法
23         try:
24             b = self.get_next()
25         except:
26             raise StopIteration
27         else:
28             return b
29     next = __next__
30     def get_next(self):        #获取下一个批次的数据
31         if self._is_eager:
32             return self._eager_iterator.get_next()
33         else:
34             return self._sess.run(self._batch_op)
35
36     def reset(self, feed_dict={}): #重置数据集迭代器指针（用于整个数据集循环迭代）
37         if self._is_eager:
38             self._eager_iterator = tfe.Iterator(self._dataset)
39         else:
40             self._sess.run(self._iterator.initializer, feed_dict=feed_dict)
41
42     def _build(self, dataset, sess=None):    #构建数据集
43         self._dataset = dataset
44
45         if self._is_eager:                  #直接返回动态图中的数据集迭代器对象
46             self._eager_iterator = tfe.Iterator(dataset)
47         else:                            #在静态图中，需要进行初始化，并返回迭代器的 get_next 方法
48             self._iterator = dataset.make_initializable_iterator()
49             self._batch_op = self._iterator.get_next()
50             if sess:
51                 self._sess = sess
52             else:                          #如果没有传入 session，则需要自己创建一个
53                 self._sess = tf.Session()
54             try:
55                 self.reset()

```

```

56     except: #处理路径不存在的异常
57     for name in pass: #忽略不存在的文件
58         @property
59         def dataset(self): #返回 dataset 属性
60             return self._dataset
61
62         @property
63         def iterator(self): #返回 iterator 属性
64             return self._iterator
65
66         @property
67         def batch_op(self): #返回 batch_op 属性
68             return self._batch_op

```

整个代码相对比较好理解，就是内部维护了一套动态图和静态图各自的迭代关系。使用的都是 Python 基础语法方面的知识。如果这部分代码不太懂，可以参考《Python 带我起飞——入门、进阶、商业实战》中“第 9 章 类——面向对象的编程方案”相关内容。

10.3.4 代码实现：将 CelebA 做成数据集

制作 Dataset 数据集可以分成两个主要部分：

- 函数 `disk_image_batch_dataset`，用来将具体的图片和标签数据拼装成 Dataset 数据集。
- 类 `Celeba` 继承于 10.3.3 小节的 `Dataset` 类。在该类中实现了具体图片数据的转化函数 `_map_func` 与一个静态方法 `check_attribute_conflict`。静态方法 `check_attribute_conflict` 的作用是将标签中与指定属性冲突的标志位清零。

具体代码如下：

代码 10-4 mydataset（续）

```

69 #从指定的图片目录中读取图片，并转成数据集
70 def disk_image_batch_dataset(img_paths, batch_size, labels=None,
71     filter=None, drop_remainder=True,
72     map_func=None, shuffle=True, repeat=-1):
73     if labels is None: #将传入的图片路径与标签转成 Dataset 数据集
74         dataset = tf.data.Dataset.from_tensor_slices(img_paths)
75     elif isinstance(labels, tuple):
76         dataset = tf.data.Dataset.from_tensor_slices((img_paths,) +
77             tuple(labels))
77     else:
78         dataset = tf.data.Dataset.from_tensor_slices((img_paths, labels))
79
80     if filter: #支持调用外部传入的 filter 处理函数
81         dataset = dataset.filter(filter)
82
83     def parse_func(path, *label): #定义数据集的 map 处理函数，用来读取图片

```

```

84     img = tf.read_file(path)
85     img = tf.image.decode_png(img, 3)
86     return (img,) + label
87
88 if map_func:                                #支持调用外部传入的map处理函数
89     def map_func_(*args):
90         return map_func_(*parse_func(*args))
91     dataset = dataset.map(map_func_, num_parallel_calls=num_threads)
92 else:
93     dataset = dataset.map(parse_func, num_parallel_calls=num_threads)
94
95 if shuffle:                                    #乱序操作
96     dataset = dataset.shuffle(buffer_size)
97 #按批次划分
98 dataset = dataset.batch(batch_size, drop_remainder = drop_remainder)
99 dataset = dataset.repeat(repeat).prefetch(prefetch_batch)#设置缓存
100 return dataset
101
102 class Celeba(Dataset):
103     #定义人脸属性
104     att_dict={'5_o_Clock_Shadow': 0, 'Arched_Eyebrows': 1, 'Attractive': 2,
105               'Bags_Under_Eyes': 3, 'Bald': 4, 'Bangs': 5, 'Big_Lips': 6,
106               'Big_Nose': 7, 'Black_Hair': 8, 'Blond_Hair': 9, 'Blurry': 10,
107               'Brown_Hair': 11, 'Bushy_Eyebrows': 12, 'Chubby': 13,
108               'Double_Chin': 14, 'Eyeglasses': 15, 'Goatee': 16,
109               'Gray_Hair': 17, 'Heavy_Makeup': 18, 'High_Cheekbones': 19,
110               'Male': 20, 'Mouth_Slightly_Open': 21, 'Mustache': 22,
111               'Narrow_Eyes': 23, 'No_Beard': 24, 'Oval_Face': 25,
112               'Pale_Skin': 26, 'Pointy_Nose': 27, 'Receding_Hairline': 28,
113               'Rosy_Cheeks': 29, 'Sideburns': 30, 'Smiling': 31,
114               'Straight_Hair': 32, 'Wavy_Hair': 33, 'Wearing_Earrings': 34,
115               'Wearing_Hat': 35, 'Wearing_Lipstick': 36,
116               'Wearing_Necklace': 37, 'Wearing_Necktie': 38, 'Young': 39}
117
118     def __init__(self, data_dir, attrs, img_resize, batch_size,
119                  shuffle=True, repeat=-1, sess=None, mode='train',
120                  crop=True):
121         super(Celeba, self).__init__()
122         #定义数据路径
123         list_file = os.path.join(data_dir, 'list_attr_celeba.txt')
124         img_dir_jpg = os.path.join(data_dir, 'img_align_celeba')
125         img_dir_png = os.path.join(data_dir, 'img_align_celeba_png')
126
127         #读取文本数据
128         names = np.loadtxt(list_file, skiprows=2, usecols=[0], dtype=np.str)
129         if os.path.exists(img_dir_png):      #将图片的文件名收集起来

```

```

129     img_paths = [os.path.join(img_dir_png, name.replace('jpg', 'png'))]
   for name in names]
130     elif os.path.exists(img_dir_jpg):
131         img_paths = [os.path.join(img_dir_jpg, name) for name in names]
132     print(img_dir_png, img_dir_jpg)
133     #读取每个图片的属性标志
134     att_id = [Celeba.att_dict[att] + 1 for att in attrs]
135     labels = np.loadtxt(list_file, skiprows=2, usecols=att_id,
   dtype=np.int64)
136
137     if img_resize == 64:
138         offset_h = 40
139         offset_w = 15
140         img_size = 148
141     else:
142         offset_h = 26
143         offset_w = 3
144         img_size = 170
145
146     def _map_func(img, label):
147         #从位于(offset_h, offset_w)的图像的左上角像素开始对图像裁剪
148         img = tf.image.crop_to_bounding_box(img, offset_h, offset_w,
   img_size, img_size)
149         #用双向插值法缩放图片
150         img = tf.image.resize(img, [img_resize, img_resize],
   tf.image.ResizeMethod.BICUBIC)
151         img = tf.clip_by_value(img, 0, 255) / 127.5 - 1#归一化处理
152     def GetDataset(mode):
153         return img, label
154
155     drop_remainder = True
156     if mode == 'test':
157         drop_remainder = False
158     shuffle = False
159     repeat = 1
160     img_paths = img_paths[182637:]
161     labels = labels[182637:]
162     elif mode == 'val':
163         img_paths = img_paths[182000:182637]
164         labels = labels[182000:182637]
165     else:
166         img_paths = img_paths[:182000]
167         labels = labels[:182000]
168     #创建数据集
169     dataset =
disk_image_batch_dataset(img_paths=img_paths, labels=labels,

```

```

170     map_func=_map_func, decode_png(img, 3), batch_size=batch_size, num_parallel_calls=4)
171     return img, label
172     : (pqf_drop_remainder=drop_remainder,
173         shuffle=shuffle, repeat=repeat)
174     self._bulid(dataset, sess) #构建数据集
175     self._img_num = len(img_paths) #计算总长度
176     def __len__(self): #重载 len 函数
177         return self._img_num #返回数据集的总长度
178     dataset = dataset.map(parse_func, num_parallel_calls=4)
179     @staticmethod #定义一个静态方法，实现将冲突类别清零
180     def check_attribute_conflict(att_batch, att_name, att_names):
181         def _set(att, value, att_name):
182             if att_name in att_names:
183                 att[att_names.index(att_name)] = value
184             att_id = att_names.index(att_name)
185             for att in att_batch: #循环处理批次中的每个反向标签
186                 if att_name in ['Bald', 'Receding_Hairline'] and att[att_id] == 1:
187                     _set(att, 0, 'Bangs') #没头发属性和退缩发际线属性与头帘属性冲突
188                     elif att_name == 'Bangs' and att[att_id] == 1:
189                         _set(att, 0, 'Bald')
190                         _set(att, 0, 'Receding_Hairline')
191                     elif att_name in ['Black_Hair', 'Blond_Hair', 'Brown_Hair',
192                         'Gray_Hair'] and att[att_id] == 1:
193                         for n in ['Black_Hair', 'Blond_Hair', 'Brown_Hair',
194                             'Gray_Hair']:
195                             if n != att_name: #头发颜色只能取一种
196                                 _set(att, 0, n)
197                         elif att_name in ['Straight_Hair', 'Wavy_Hair'] and att[att_id]
198                             == 1:
199                             for n in ['Straight_Hair', 'Wavy_Hair']:
200                                 if n != att_name: #直发属性和波浪属性
201                                     _set(att, 0, n)
202                                     elif att_name in ['Mustache', 'No_Beard'] and att[att_id] == 1:
203                                         for n in ['Mustache', 'No_Beard']: #有胡子属性和没胡子属性
204                                             if n != att_name:
205                                                 _set(att, 0, n)
206         return att_batch

```

在代码第 104 行中，手动定义了人脸属性的字典。该字典的属性名称与顺序要与 10.3.1 小节介绍的样本标注中的一致。在整个项目中，都会用这个字典来定位图片的具体属性。

代码第 137 行是一个对输入图片主要内容增强的小技巧：先按照一定尺寸将图片主要内容剪辑下来，再将其转化为指定的尺寸，从而实现将主要内容区域放大的效果。因为本实例使用

的人脸数据集是经过对齐预处理后的图片（高为 218 pixel，宽为 178 pixel），所以可以用人为调好的数值进行裁剪。

代码第 137~144 行的意思是：如果使用 64 pixel×64 pixel 大小的图片，则从原始图片的(15,40)坐标处裁剪 148 pixel×148 pixel 大小的区域；如果使用其他尺寸大小的图片，则从原始图片的(3,26)坐标处裁剪 170 pixel×170 pixel 大小的区域。

裁剪后的图片将被用双向插值法缩放为指定大小的图片。



提示：

更多变化图片尺寸的方法，请参考《深度学习之 TensorFlow——入门、原理与进阶实战》一书的 12.7.1 小节。

10.3.5 代码实现：构建 AttGAN 模型的编码器

模型编码器模型由多个卷积层组成。每一层在进行卷积操作后，都会做批量归一化处理(BN)。另外，用一个列表 zs 将每层的处理结果收集起来一起返回。

编码器模型的结果和列表 zs 中的中间层特征会在 10.3.6 小节的解码器模型中被使用。

具体代码如下：

代码 10-5 AttGANmodels

```

01 import tensorflow as tf
02 import tensorflow.contrib.slim as slim
03
04 MAX_DIM = 64 * 16          # 卷积输出的最小维度
05 def Genc(x, dim=64, n_layers=5, is_training=True):
06     with tf.variable_scope('Genc', reuse=tf.AUTO_REUSE):
07         z = x
08         zs = []
09         for i in range(n_layers):      # 循环卷积操作
10             d = min(dim * 2**i, MAX_DIM)
11             z = slim.conv2d(z, d, 4, 2, activation_fn=tf.nn.leaky_relu)
12             z = slim.batch_norm(z, scale=True, updates_collections=None,
13             is_training=is_training)      # 批量归一化处理
14         zs.append(z)
15     return zs

```

在代码第 12 行的批量归一化 (BN) 处理中，调用了 slim.batch_norm 函数。该函数的几个重要参数说明如下。

- **scale**: 是否使用自适应模式（见 10.1.4 小节）。这里将 scale 设为了 True，表示使用自适应模式（见代码第 12 行）。
- **updates_collections**: 设置更新移动均值 μ 和移动方差 σ^2 的 OP (操作符)。默认值为 tf.GraphKeys.UPDATE_OPS，表示在 BN 处理时，会将更新移动均值 μ 和移动方差 σ^2 的操作符保存在 tf.GraphKeys.UPDATE_OPS 里。此时并不会对移动均值和移动方差做真正

的更新操作，而是等待外部代码来触发该 OP（操作符）执行（见 5.2.7 小节）。这里将参数 `updates_collections` 设为了 `None`，表示在 BN 处理时每次都强制更新移动均值 μ 和移动方差 σ （见代码第 12 行）。这种方式可以保证移动均值 μ 和移动方差 σ 实时更新。但在分布式运行时，这种方式会对性能影响很大。

- `decay`: 估计移动平均值的衰减系数。它与训练步数相对应。即需要训练 $1/(1-\text{decay})$ 步，才能够真正收敛。该参数默认值为 0.999，表示至少需要的训练步数为 $1/(1-0.999) = 1000$ 步才能够使模型真正收敛。

10.3.6 代码实现：构建含有转置卷积的解码器模型

解码器模型是由注入层、短连接层、多个转置卷积层构成的。

- 注入层：将标签信息按照解码器模型中间层的尺寸 $[h,w]$ 复制 $h \times w$ 份，变成形状为 $[\text{batch}, h, w, \text{标签属性个数}]$ 的矩阵。然后用 `concat` 函数将该矩阵与解码器模型中间层信息连接起来，一起传入下一层进行转置卷积操作。
- 短连接：将 10.3.5 小节编码器模型中间层信息与对应的解码器模型中间层信息用 `concat` 函数结合起来，一起传入下一层进行转置卷积操作。
- 转置卷积层：通过将卷积核转置并进行反卷积操作。该网络层具有信息还原的功能。

解码器模型中转置卷积层的数量要与编码器模型中卷积层的数量一致，各为 5 层。编码器模型与解码器模型的结构如图 10-6 所示。

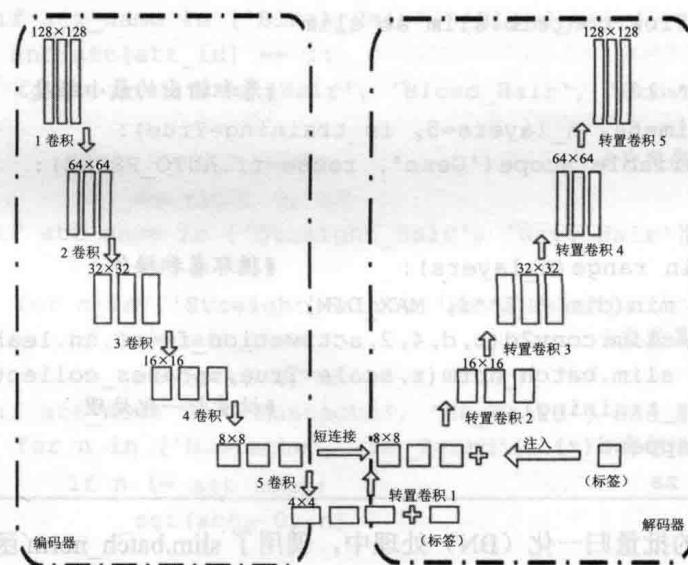


图 10-6 编码器模型与解码器模型的结构

按照图 10-6 中的结构，解码器模型的处理流程如下：

- (1) 将编码器模型的结果加入标签信息作为原始数据。
- (2) 在第 1 层进行转置卷积后加入短连接信息。
- (3) 将标签通过注入层与第 (2) 步的结果连接起来。

(4) 依次再通过4层转置卷积，得到与原始图片尺寸相同(128 pixel×128 pixel)的输出。

其中，短连接层的数量与注入层的数量是可以通过参数调节的。这里使用的参数为1，代表各使用1层。

具体代码如下：

代码 10-5 AttGANmodels (续)

```

15 def Gdec(zs, _a, dim=64, n_layers=5, shortcut_layers=1, inject_layers=0,
16   is_training=True):
17   shortcut_layers = min(shortcut_layers, n_layers - 1) # 定义短连接层
18   inject_layers = min(inject_layers, n_layers - 1) # 定义注入层
19   def _concat(z, z_, _a):                         # 定义函数，实现 concat 操作
20     feats = [z]
21     if z_ is not None:                           # 追加短连接层信息
22       feats.append(z_)
23     if _a is not None:                          # 追加注入层的标签信息
24       # 调整标签维度，与解码器模型的中间层一致
25       _a = tf.reshape(_a, [-1, 1, 1, _a.get_shape()[-1]])
26       # 按照解码器模型中间层输出的尺寸进行复制
27       _a = tf.tile(_a, [1, z.get_shape()[1], z.get_shape()[2], 1])
28     feats.append(_a)
29   return tf.concat(feats, axis=3)               # 对特征进行 concat 操作
30
31   with tf.variable_scope('Gdec', reuse=tf.AUTO_REUSE):
32     z = _concat(zs[-1], None, _a)                 # 将编码器模型结果与标签结合起来
33     for i in range(n_layers):                   # 5 层转置卷积
34       if i < n_layers - 1:
35         d = min(dim * 2** (n_layers - 1 - i), MAX_DIM)
36         z = slim.conv2d_transpose(z, d, 4, 2, activation_fn=tf.nn.relu)
37         z = slim.batch_norm(z, scale=True, updates_collections=None,
38           is_training=is_training)
39         if shortcut_layers > i:                  # 实现短连接层
40           z = _concat(z, zs[n_layers - 2 - i], None)
41         if inject_layers > i:                  # 实现注入层
42           z = _concat(z, None, _a)
43         else:
44           x = slim.conv2d_transpose(z, 3, 4, 2, activation_fn=tf.nn.tanh)
45           # 对最后一层的结果进行特殊处理
46
47   return x

```

代码第43行，对最后一层的结果做了激活函数tanh的转化，将最终结果变成与原始图片归一化处理后一样的值域(-1~1之间)。



提示：

这里分享一个在实际训练中得出的经验：激活函数leaky_relu配合卷积神经网络的效果

要比激活函数 `relu` 好。所以可以看到，在 10.3.5 小节中的编码器模型部分使用的是激活函数 `leaky_relu`，而在本节的解码器模型部分使用的是激活函数 `relu`。

10.3.7 代码实现：构建 AttGAN 模型的判别器模型部分

判别器模型相对简单。步骤如下：

- (1) 用 5 层卷积网络对输入数据进行特征提取。
- (2) 在第(1)步的 5 层卷积网络中，每次卷积操作之后，都进行一次实例归一化（10.1.5 小节）处理。实例归一化可以帮助卷积网络更好地对独立样本个体进行特征提取。
- (3) 将第(1)步的结果分成两份，分别通过 2 层全连接网络，得到判别真伪的结果与判别分类的结果。
- (4) 将最终的判别真伪的结果与判别分类的结果返回。

具体代码如下。

代码 10-5 AttGANmodels（续）

```

45 def D(x, n_att, dim=64, fc_dim=MAX_DIM, n_layers=5):
46     with tf.variable_scope('D', reuse=tf.AUTO_REUSE):
47         y = x
48         for i in range(n_layers):          #5 层卷积网络
49             d = min(dim * 2**i, MAX_DIM)
50             y= slim.conv2d(y,d,4,2,
51                             normalizer_fn=slim.instance_norm,activation_fn=tf.nn.leaky_relu)
51             print(y.shape,y.shape.ndims)
52             if y.shape.ndims > 2:      #大于2维，需要展开。变成2维的再做全连接
53                 y = slim.flatten(y)
54             #用2层全连接辨别真伪
55             logit_gan = slim.fully_connected(y, fc_dim,activation_fn
56 =tf.nn.leaky_relu )
56             logit_gan = slim.fully_connected(logit_gan, 1,activation_fn =None )
57             #用2层全连接进行分类
58             logit_att = slim.fully_connected(y, fc_dim,activation_fn
59 =tf.nn.leaky_relu )
59             logit_att = slim.fully_connected(logit_att, n_att,activation_fn
60 =None )
61             return logit_gan, logit_att
62
63 def gradient_penalty(f, real, fake=None):      #计算 WGAN-gp 的惩罚项
64     def _interpolate(a, b=None):                  #定义联合分布空间的取样函数
65         with tf.name_scope('interpolate'):
66             if b is None:
67                 beta = tf.random_uniform(shape=tf.shape(a), minval=0.,
68                                         maxval=1.)
68                 _, variance = tf.nn.moments(a, range(a.shape.ndims))

```

```

69         b = a + 0.5 * tf.sqrt(variance) * beta
70         shape = [tf.shape(a)[0]] + [1] * (a.shape.ndims - 1)
71     val_dat # 定义取样的随机数
72     val_dat alpha = tf.random_uniform(shape=shape, minval=0., maxval=1.)
73     x_sample inter = a + alpha * (b - a)      # 联合空间取样
74     b_sample inter.set_shape(a.get_shape().as_list())
75     f(x) return inter
76
77     with tf.name_scope('gradient_penalty'):
78         x = _interpolate(real, fake)      # 在联合分布空间取样
79         pred = f(x)
80         if isinstance(pred, tuple):
81             pred = pred[0]
82         grad = tf.gradients(pred, x)[0]      # 计算梯度惩罚项
83         norm = tf.norm(slim.flatten(grad), axis=1)
84         gp = tf.reduce_mean((norm - 1.)**2)
85     return gp

```

代码第 63 行是一个计算对抗网络惩罚项的函数。该惩罚项源于 WGAN-gp 对抗神经网络模型。如果在 WGAN 模型与 LSGAN 模型中添加了惩罚项，则分别变成了 WGAN-gp、LSGAN-gp 模型。



提示：

关于该部分的更多知识，还可以参考《深度学习之 TensorFlow——入门、原理与进阶实战》一书中 12.5 节的 WGAN-gp 模型与 12.6 节的 LSGAN 模型介绍。

10.3.8 代码实现：定义模型参数，并构建 AttGAN 模型

接下来进入模型训练环节。

首先，在静态图中构建 AttGAN 模型，并创建数据集。具体代码如下。

代码 10-6 trainattgan

```

01 from functools import partial          # 引入偏函数库
02 import traceback
03 import re                            # 引入正则库
04 import numpy as np
05 import tensorflow as tf
06 import time
07 import os
08 import scipy.misc
09 # 引入本地文件
10 mydataset = __import__("10-4 mydataset")
11 data = mydataset.data
12 AttGANmodels = __import__("10-5 AttGANmodels")
13 models = AttGANmodels.models

```

```

14
15 img_size = 128 # 定义图片尺寸
16 # 定义模型参数
17 shortcut_layers = 1 # 定义短连接层数
18 inject_layers = 1 # 定义注入层数
19 enc_dim = 64 # 定义编码维度
20 dec_dim = 64 # 定义解码维度
21 dis_dim = 64 # 定义判别器模型维度
22 dis_fc_dim = 1024 # 定义判别器模型中全连接的节点
23 enc_layers = 5 # 定义编码器模型层数
24 dec_layers = 5 # 定义解码器模型层数
25 dis_layers = 5 # 定义判别器模型器层数
26
27 # 定义训练参数
28 mode = 'wgan' # 设置计算损失的方式, 还可设为 "lsgan"
29 epoch = 200 # 定义迭代次数
30 batch_size = 32 # 定义批次大小
31 lr_base = 0.0002 # 定义学习率
32 n_d = 5 # 定义训练间隔, 训练 n_d 次判别器模型伴随一次生成器模型
33 # 定义生成器模型的随机方式
34 b_distribution = 'none' # 还可以取值: uniform、truncated_normal
35 thres_int = 0.5 # 训练时, 特征的上下限值域
36 # 测试时特征属性的上下限值域
37 test_int = 1.0 # 一般要大于训练时的值域, 使特征更加明显
38 n_sample = 32
39
40 # 定义默认属性
41 att_default = ['Bald', 'Bangs', 'Black_Hair', 'Blond_Hair', 'Brown_Hair',
    'Bushy_Eyebrows', 'Eyeglasses', 'Male', 'Mouth_Slightly_Open', 'Mustache',
    'No Beard', 'Pale_Skin', 'Young']
42 n_att = len(att_default)
43
44 experiment_name = "128_shortcut1_inject1_None" # 定义模型的文件夹名称
45 os.makedirs('./output/%s' % experiment_name, exist_ok=True) # 创建目录
46
47 tf.reset_default_graph()
48 # 定义运行 session 的硬件配置
49 config = tf.ConfigProto(allow_soft_placement=True,
    log_device_placement=False)
50 config.gpu_options.allow_growth = True
51 sess = tf.Session(config=config)
52
53 # 建立数据集
54 tr_data = data.Celeba(r'E:\newgan\AttGAN-Tensorflow-master\data',
    att_default, img_size, batch_size, mode='train', sess=sess)
55 val_data = data.Celeba(r'E:\newgan\AttGAN-Tensorflow-master\data',
    att_default, img_size, n_sample, mode='val', shuffle=False, sess=sess)

```

```

56
57 #准备一部分评估样本，用于测试模型的输出效果
58 val_data.get_next()
59 val_data.get_next()
60 xa_sample_ipt, a_sample_ipt = val_data.get_next()
61 b_sample_ipt_list = [a_sample_ipt]      #保存原始样本标签，用于重建
62 for i in range(len(att_default)):        #每个属性生成一个标签
63     tmp = np.array(a_sample_ipt, copy=True)
64     tmp[:, i] = 1 - tmp[:, i]            #将指定属性取反，去掉显像属性的冲突项
65     tmp = data.Celeba.check_attribute_conflict(tmp, att_default[i],
66     att_default)
66     b_sample_ipt_list.append(tmp)
67
68 #构建模型
69 Genc = partial(models.Genc, dim=enc_dim, n_layers=enc_layers)
70 Gdec = partial(models.Gdec, dim=dec_dim, n_layers=dec_layers,
    shortcut_layers=shortcut_layers, inject_layers=inject_layers)
71 D = partial(models.D, n_att=n_att, dim=dis_dim, fc_dim=dis_fc_dim,
    n_layers=dis_layers)

```

代码第 58~66 行，根据评估样本的标签数据来合成多个目标标签。这些目标标签将被输入模型中用于生成指定的人脸图片。具体步骤如下：

- (1) 用数据集生成一部分评估样本及对应的标签。
- (2) 从默认属性 `att_default` (见代码第 41 行) 中取出一个属性索引。
- (3) 用第 (2) 步的属性索引，在样本标签中找到对应的属性值，将其取反。
- (4) 将取反后的标签保存起来，完成一个目标标签的制作。
- (5) 用 `for` 循环遍历默认属性 `att_default`，在循环中实现第 (2) ~ (4) 步的操作，合成多个目标标签。

在合成目标标签的过程中，每个目标标签只在原来的标签上改变了一个属性。这样做可以使输出的效果更加明显。

在代码第 69~71 行，用偏函数分别对编码器模型、解码器模型、判别器模型进行二次封装，将常量参数固定起来。

10.3.9 代码实现：定义训练参数，搭建正反向模型

定义学习率、输入样本、模拟标签相关的占位符，并构建正反向模型。

1. 搭建 AttGAN 模型正向结构的步骤

按照 10.3.2 小节中 AttGAN 模型正向结构的描述实现如下步骤：

- (1) 用编码器模型提取特征。
- (2) 将提取后的特征与样本标签一起输入解码器模型，重建输入的人脸图片。
- (3) 将第(1)步提取后的特征与模拟标签一起输入解码器模型，完成模拟人脸图片的生成。
- (4) 将第(3)步的模拟人脸图片与真实的图片输入判别器，模型进行图片真伪的判断和属

性分类的计算。

1. 搭建 AttGAN 模型中的技术细节

在标签计算之前，统一进行一次值域变化，将标签的值域从 0~1 变为 -0.5~0.5，见代码第 75 行。

在模拟标签部分，代码中给出了 3 种方法：直接乱序、用 uniform 随机值进行变化、用 truncated_normal 随机值进行变化，见代码第 77~82 行。

完整的代码如下。

代码 10-6 trainattgan（续）

```

72 lr = tf.placeholder(dtype=tf.float32, shape=[]) # 定义学习率占位符
73 xa = tr_data.batch_op[0]                         # 定义获取训练图片数据的 OP
74 a = tr_data.batch_op[1]                         # 定义获取训练标签数据的 OP
75 _a = (tf.cast(a, tf.float32) * 2 - 1) * thres_int      # 改变标签值域
76 b = tf.random_shuffle(a)                         # 打乱属性标签的对应关系，用于生成器模型的输入
77 if b_distribution == 'none':                   # 构建生成器模型的随机值标签
78     _b = (tf.cast(b, tf.float32) * 2 - 1) * thres_int
79 elif b_distribution == 'uniform':
80     _b = (tf.cast(b, tf.float32) * 2 - 1) * tf.random_uniform(tf.shape(b)) *
81     (2 * thres_int)
82 elif b_distribution == 'truncated_normal':
83     _b = (tf.cast(b, tf.float32) * 2 - 1) * (tf.truncated_normal(tf.shape(b))
84     + 2) / 4.0 * (2 * thres_int)
85
86 xa_sample = tf.placeholder(tf.float32, [None, img_size, img_size, 3])
85 _b_sample = tf.placeholder(tf.float32, [None, n_att])
86
87 # 构建生成器模型
88 z = Genc(xa)                                     # 用编码器模型提取特征
89 xb_ = Gdec(z, _b)                                # 将编码器模型输出的特征配合随机属性，生成人脸图片（用于对抗）
90 with tf.control_dependencies([xb_]):
91     xa_ = Gdec(z, _a) # 将编码器模型输出的特征配合原有标签属性，生成人脸图片（用于重建）
92
93 # 构建判别器模型
94 xa_logit_gan, xa_logit_att = D(xa)
95 xb_logit_gan, xb_logit_att = D(xb_)
96
97 # 计算判别器模型损失
98 if mode == 'wgan':                               # 用 wgan-gp 方式
99     wd = tf.reduce_mean(xa_logit_gan) - tf.reduce_mean(xb_logit_gan)
100    d_loss_gan = -wd
101    gp = models.gradient_penalty(D, xa, xb_)      # 用梯度惩罚方法计算判别器损失
102 elif mode == 'lsgan':                            # 用 lsgan-gp 方式
103    xa_gan_loss = tf.losses.mean_squared_error(tf.ones_like(xa_logit_gan),
103        xa_logit_gan)

```