

```

66     image = tf.reshape(image, [256, 256, 3])
67
68     label = tf.cast(features['label'], tf.int32)      #转换标签类型
69
70     if flag == 'train':      #如果是训练使用，则应将其归一化，并按批次组合
71         image = tf.cast(image, tf.float32) * (1. / 255) - 0.5 #归一化
72         img_batch, label_batch = tf.train.batch([image, label], #按照批次组合
73                                         batch_size=batch_size,
74                                         capacity=20)
75
76     return image, label
77
78 TFRecordfilenames = ["mydata.tfrecords"]
79 image, label = read_and_decode(TFRecordfilenames, flag='test') #以测试的方式
打开数据集

```

函数 `read_and_decode` 接收的参数有：TFRecord 文件名列表（`filenames`）、运行模式（`flag`）、划分的批次（`batch_size`）。

- 如果是测试模式，则返回一个标签数据，代表被测图片的计算结果。
- 如果是训练模式，则返回一个列表，其中包含一批次样本数据的计算结果。

代码第 78、79 行调用了函数 `read_and_decode`，并向函数 `read_and_decode` 的参数 `flag` 设置为 `test`，代表是以测试模式加载数据集。该函数被执行后，便可以在会话（`session`）中通过队列的方式读取数据了。



提示：

如果要以训练模式加载数据集，则直接将函数 `read_and_decode` 的参数 `flag` 设置为 `train` 即可。完整的代码可以参考本书配套资源中的代码文件“4-5 将图片文件制作成 TFRecord 数据集.py”。

4.5.5 代码实现：建立会话，将数据保存到文件

将数据保存到文件中的步骤如下：

- (1) 定义要保存文件的路径。
- (2) 建立会话（`session`），准备运行静态图。
- (3) 在会话（`session`）中启动一个带有协调器的队列线程。
- (4) 用会话（`session`）的 `run` 方法获得数据，并将数据保存到指定路径下。

具体代码如下：

代码 4-5 将图片文件制作成 TFRecord 数据集（续）

```

80 saveimgpath = 'show\\'
#定义保存图片的路径
81 if tf.gfile.Exists(saveimgpath):
#如果存在 saveimgpath，则将其删除

```

```

82     tf.gfile.DeleteRecursively(saveimgpath)
83 tf.gfile.MakeDirs(saveimgpath)          #创建 saveimgpath 路径
84
85 #开始一个读取数据的会话
86 with tf.Session() as sess:
87     sess.run(tf.local_variables_initializer()) #初始化本地变量，没有这句会报错
88
89     coord= tf.train.Coordinator()           #启动多线程
90     threads= tf.train.start_queue_runners(coord=coord)
91     myset = set([])                      #建立集合对象，用于存放子文件夹
92
93     try:
94         i = 0
95         while True:
96             example, examplelab = sess.run([image,label]) #取出 image 和 label
97             examplelab = str(examplelab)
98             if examplelab not in myset:
99                 myset.add(examplelab)
100                tf.gfile.MakeDirs(saveimgpath+examplelab)
101                img=Image.fromarray(example, 'RGB')           #转换 Image 格式
102                img.save(saveimgpath+examplelab+'/'+str(i)+'_Label_'+'.jpg') #
保存图片
103                print( i)
104                i = i+1
105            except tf.errors.OutOfRangeError:           #定义取完数据的异常处理
106                print('Done Test -- epoch limit reached')
107            finally:
108                coord.request_stop()
109                coord.join(threads)
110                print("stop()")

```

代码第 82 行是删除指定目录的操作，也可以用代码 shutil.rmtree(saveimgpath) 来实现。

在代码第 91 行，建立了集合对象 myset，用于按数据的标签来建立子文件夹。

在代码第 95 行，用无限循环的方式从训练集里不停地取数据。当训练集里的数据被取完之后，会触发 tf.errors.OutOfRangeError 异常。

在代码第 98 行，会判断是否有新的标签出现。如果没有新的标签出现，则将数据存到已有的文件夹里；如果有新的标签出现，则接着创建新的子文件夹（见代码第 100 行）。

4.5.6 运行程序

程序运行后，输出以下结果：

```
loading sample dataset..
```

```
100%|██████████| 20/20 [00:00<00:00, 246.26it/s]
```

```
0
```

```
1
```

```

.....
18
19
Done Test -- epoch limit reached
stop()

```

执行之后，在本地路径下会发现有一个 show 的文件夹，里面放置了生成的图片，如图 4-8 所示。



图 4-8 转化后的 man 和 woman 样本

show 文件夹中有两个子文件夹：0 和 1。0 文件夹中放置的是男人图片，1 文件夹中放置的是女人图片。

4.6 实例 6：将内存对象制作成 Dataset 数据集

`tf.data.Dataset` 接口是一个可以生成 Dataset 数据集的高级接口。用 `tf.data.Dataset` 接口来处理数据集会使代码变得简单。这也是目前 TensorFlow 主推的一种数据集处理方式。

实例描述

生成一个模拟 $y \approx 2x$ 的数据集，将数据集的样本和标签分别以元组和字典类型存放为两份。建立两个 `Dataset` 数据集：一个被传入元组类型的样本，另一个被传入字典类型的样本。

对这两个数据集做以下操作，并比较结果：

- (1) 处理数据源是元组类型的数据集，将前 5 个数据依次显示出来。
- (2) 处理数据源是字典类型的数据集，将前 5 个数据依次显示出来。
- (3) 处理数据源是元组类型的数据集，按照每批次 10 个样本的格式进行划分，并将前 5 个批次的数据依次显示出来。
- (4) 对数据源是字典类型的数据集中的 y 变量做变换，将其转化成整形。然后将前 5 个数据依次显示出来。
- (5) 对数据源是元组类型的数据集进行乱序操作，将前 5 个数据依次显示出来。

本节先介绍 `tf.data.Dataset` 接口的基本使用方法，然后介绍 `Dataset` 数据集的具体操作。

4.6.1 如何生成 Dataset 数据集

`tf.data.Dataset` 接口是通过创建 `Dataset` 对象来生成 `Dataset` 数据集的。`Dataset` 对象可以表示

为一系列元素的封装。

有了 Dataset 对象之后，就可以在其上直接做乱序（shuffle）、元素变换（map）、迭代取值（iterate）等操作。

Dataset 对象可以由不同的数据源转化而来。在 tf.data.Dataset 接口中，有三种方法可以将内存中的数据转化成 Dataset 对象，具体如下。

- `tf.data.Dataset.from_tensors`: 根据内存对象生成 Dataset 对象。该 Dataset 对象中只有一个元素。
- `tf.data.Dataset.from_tensor_slices`: 根据内存对象生成 Dataset 对象。内存对象是列表、元组、字典、Numpy 数组等类型。另外，该方法也支持 TensorFlow 中的张量类型。
- `tf.data.Dataset.from_generator`: 根据生成器对象生成 Dataset 对象。具体可以参考 8.3 节的自然语言处理（NLP）实例。

这几种方法的使用基本类似。本实例中使用的是 `tf.data.Dataset.from_tensor_slices` 接口。



提示：

在使用 `tf.data.Dataset.from_tensor_slices` 之类的接口时，如果传入了嵌套 list 类型的对象，则必须保证 list 中每个嵌套元素的长度都相同，否则会报错。

正确使用举例：

```
Dataset.from_tensor_slices([[1, 2], [1, 2]]) #list 里有两个子 list，并且长度相同
```

错误使用举例：

```
Dataset.from_tensor_slices([[1, 2], [1]]) #list 里有两个子 list，并且长度不同
```

4.6.2 如何使用 Dataset 接口

使用 Dataset 接口的操作步骤如下：

- (1) 生成数据集 Dataset 对象。
- (2) 对 Dataset 对象中的样本进行变换操作。
- (3) 创建 Dataset 迭代器。
- (4) 在会话（session）中将数据取出。

其中，第（1）步是必备步骤，第（2）步是可选步骤。

1. Dataset 接口所支持的数据集操作

在 `tf.data.Dataset` 接口的 API 中，支持的数据集变换操作有：有乱序（shuffle）、自定义元素变换（map）、按批次组合（batch）、重复（repeat）等。

2. Dataset 接口在不同框架中的应用

第（3）步和第（4）步是在静态图中使用数据集的步骤，作用是取出数据集中的数据。在实际应用中，第（3）步和第（4）步会随着 Dataset 对象所应用的框架不同而有所变化。例如：

- 在动态图框架中，可以直接迭代 Dataset 对象进行取数据（见本书 9.3 节中 Dataset 数据

集的使用实例)。

- 在估算器框架中，可以直接将 Dataset 对象封装成输入函数来进行取数据（见本书 9.4 节中 Dataset 数据集的使用实例）。

4.6.3 tf.data.Dataset 接口所支持的数据集变换操作

在 TensorFlow 中封装了 tf.data.Dataset 接口的多个常用函数，见表 4-1。

表 4-1 tf.data.Dataset 接口的常用函数

函 数	描 述
range(*args)	<p>根据传入的数值范围，生成一系列整数数字组成的数据集。其中，传入参数与 Python 中的 xrange 函数一样，共有 3 个：start(起始数字)、stop(结束数字)、step(步长)。</p> <p>例：import tensorflow as tf Dataset =tf.data.Dataset</p> <pre>Dataset.range(5) == [0, 1, 2, 3, 4] Dataset.range(2, 5) == [2, 3, 4] Dataset.range(1, 5, 2) == [1, 3] Dataset.range(1, 5, -2) == [] Dataset.range(5, 1) == [] Dataset.range(5, 1, -2) == [5, 3]</pre>
zip(datasets)	<p>将输入的多个数据集按内部元素顺序重新打包成新的元组序列。它与 Python 中的 zip 函数意义一样。更多内容可参考《Python 带我起飞——入门、进阶、商业实战》一书 5.3.5 小节。</p> <p>例：import tensorflow as tf Dataset =tf.data.Dataset</p> <pre>a = Dataset.from_tensor_slices([1, 2, 3]) b = Dataset.from_tensor_slices([4, 5, 6]) c = Dataset.from_tensor_slices((7, 8), (9, 10), (11, 12)) d = Dataset.from_tensor_slices([13, 14]) Dataset.zip((a, b)) == {(1, 4), (2, 5), (3, 6)} Dataset.zip((a, b, c)) == {(1, 4, (7, 8)), (2, 5, (9, 10)), (3, 6, (11, 12))} Dataset.zip((a, d)) == {(1, 13), (2, 14)}</pre>
concatenate(dataset)	<p>将输入的序列(或数据集)数据连接起来。</p> <p>例：import tensorflow as tf Dataset =tf.data.Dataset</p> <pre>a = Dataset.from_tensor_slices([1, 2, 3]) b = Dataset.from_tensor_slices([4, 5, 6, 7]) a.concatenate(b) == {1, 2, 3, 4, 5, 6, 7}</pre>

续表

函数	描述
list_files(file_pattern, shuffle=None)	<p>获取本地文件，将文件名做成数据集。提示：文件名是二进制形式。</p> <p>例：在本地路径下有以下3个文件：</p> <ul style="list-style-type: none"> • facelib\one.jpg • facelib\two.jpg • facelib\嘴炮.jpg <p>制作数据集代码：</p> <pre>import tensorflow as tf Dataset = tf.data.Dataset dataset = Dataset.list_files('facelib*.jpg')</pre> <p>得到的数据集：</p> <pre>{ b'facelib\\two.jpg' b'facelib\\one.jpg' b'facelib\\xe5\x98\xb4\xe7\x82\xae.jpg'}</pre> <p>生成的二进制可以转成字符串来显示。</p> <p>例：</p> <pre>str1 = b'facelib\\xe5\x98\xb4\xe7\x82\xae.jpg' print(str1.decode()) 输出：facelib\嘴炮.jpg</pre> <p>更多二进制与字符串转化信息，可参考《Python带我起飞——入门、进阶、商业实战》一书8.5节</p>
repeat(count=None)	<p>生成重复的数据集。输入参数count代表重复的次数。</p> <p>例：import tensorflow as tf Dataset = tf.data.Dataset a = Dataset.from_tensor_slices([1, 2, 3]) a.repeat(1) == {1, 2, 3, 1, 2, 3}</p> <p>也可以无限次重复，例如：a.repeat()</p>
shuffle(buffer_size, seed=None, reshuffle_each_iteration=None)	<p>将数据集的内部元素顺序随机打乱。参数说明如下。</p> <ul style="list-style-type: none"> • buffer_size：随机打乱元素排序的大小（越大越混乱）。 • seed：随机种子。 • reshuffle_each_iteration：是否每次迭代都随机乱序。 <p>例：import tensorflow as tf Dataset = tf.data.Dataset a = Dataset.from_tensor_slices([1, 2, 3, 4, 5]) a.shuffle(1) == {1, 2, 3, 4, 5} a.shuffle(10) == {4, 1, 3, 2, 5}</p>
batch(batch_size, drop_remainder)	<p>将数据集的元素按照批次组合。参数说明如下。</p> <ul style="list-style-type: none"> • batch_size：批次大小。 • drop_remainder：是否忽略批次组合后剩余的数据。 <p>例：import tensorflow as tf Dataset = tf.data.Dataset a = Dataset.from_tensor_slices([1, 2, 3, 4, 5]) a.batch(1) == {[1], [2], [3], [4], [5]} a.batch(2) == {[1, 2], [3, 4], [5]}</p>

续表

函数	描述
padded_batch(batch_size, padded_shapes, padding_values=None)	为数据集的每个元素补充 padding_values 值。参数说明如下。 <ul style="list-style-type: none"> • batch_size: 生成的批次。 • padded_shapes: 补充后的样本形状。 • padding_values: 所需要补充的值（默认为 0）。 <p>例: data1 = tf.data.Dataset.from_tensor_slices([[1, 2], [1, 3]]) data1 = data1.padded_batch(2, padded_shapes=[4]) == {[1 2 0 0] [1 3 0 0]} 在每条数据后面补充两个 0, 使其形状变为 [4]</p>
map(map_func, num_parallel_calls=None)	通过 map_func 函数将数据集中的每个元素进行处理转换, 返回一个新的数据集。参数说明如下。 <ul style="list-style-type: none"> • map_func: 处理函数。 • num_parallel_calls: 并行的处理的线程个数。 <p>例: import tensorflow as tf Dataset = tf.data.Dataset a = Dataset.from_tensor_slices([1, 2, 3, 4, 5]) a.map(lambda x: x + 1) == {2, 3, 4, 5, 6}</p>
flat_map(map_func)	将整个数据集放到 map_func 函数中去处理, 并将处理完的结果展平。 <p>例: import tensorflow as tf Dataset = tf.data.Dataset a = Dataset.from_tensor_slices([[1, 2, 3], [4, 5, 6]]) a.flat_map(lambda x: Dataset.from_tensors(x)) == {[1 2 3] [4 5 6]} 将数据集展平后返回</p>
interleave(map_func, cycle_length, block_length=1)	控制元素的生成顺序函数。参数说明如下。 <ul style="list-style-type: none"> • map_func: 每个元素的处理函数。 • cycle_length: 循环处理元素个数。 • block_length: 从每个元素所对应的组合对象中, 取出的个数。 <p>例: 在本地路径下有以下 4 个文件: <ul style="list-style-type: none"> • testset\1mem.txt • testset\1sys.txt • testset\2mem.txt • testset\2sys.txt mem 的文件为每天的内存信息, 内容为: 1day 9:00 CPU mem 110 1day 9:00 GPU mem 11 sys 的文件为每天的系统信息, 内容为: 1day 9:00 CPU 11.1 1day 9:00 GPU 91.1 现要将每天的内存信息和系统信息按照时间的顺序放到数据集中。 def parse_fn(x): print(x)</p>

续表

函数	描述
interleave(predicate, cycle_length=1, block_length=1)	<pre> return x dataset = (Dataset.list_files('testset/*txt', shuffle=False) .interleave(lambda x: tf.data.TextLineDataset(x).map(parse_fn, num_parallel_calls=1), cycle_length=2, block_length=2)) 生成的数据集为: b'1day 9:00 CPU mem 110' b'1day 9:00 GPU mem 11' b'1day 9:00 CPU 11.1' b'1day 9:00 GPU 91.1' b'1day 10:00 CPU mem 210' b'1day 10:00 GPU mem 21' b'1day 10:00 CPU 11.2 b'1day 10:00 GPU 91.2' b'1day 11:00 CPU mem 310' b'1day 11:00 GPU mem 31' </pre> <p>本实例的完整代码及数据文件在随书的配套资源中，见代码文件“4-6_interleave_例子.py”</p>
filter(predicate)	<p>将整个数据集中的元素按照函数 predicate 进行过滤，留下使函数 predicate 返回为 True 的数据。</p> <p>例：import tensorflow as tf</p> <pre> dataset = tf.data.Dataset.from_tensor_slices([1.0, 2.0, 3.0, 4.0, 5.0]) dataset = dataset.filter(lambda x: tf.less(x, 3)) == { [1.0 2.0] } 过滤掉大于 3 的数字 </pre>
apply(transformation_func)	<p>将一个数据集转换为另一个数据集。</p> <p>例：data1 = np.arange(50).astype(np.int64)</p> <pre> dataset = tf.data.Dataset.from_tensor_slices(data1) dataset = dataset.apply((tf.contrib.data.group_by_window(key_func=lambda x: x%2, reduce_func=lambda _, els: els.batch(10), window_size=20))) == { [0 2 4 6 8 10 12 14 16 18] [20 22 24 26 28 30 32 34 36 38] [1 3 5 7 9 11 13 15 17 19] [21 23 25 27 29 31 33 35 37 39] [40 42 44 46 48] [41 43 45 47 49] } </pre> <p>该代码内部执行逻辑如下：</p> <ol style="list-style-type: none"> (1) 将数据集中偶数行与奇数行分开。 (2) 以 window_size 为窗口大小，一次取 window_size 个偶数行和 window_size 个奇数行。 (3) 在 window_size 中，按照指定的批次 batch 进行组合，并将处理后的数据集返回。
shard(num_shards, index)	用在分布式训练场景中，代表将数据集分为 num_shards 份，并取第 index 份数据

续表

函数	描述
prefetch(buffer_size)	设置从数据集中取数据时的最大缓冲区。buffer_size 是缓冲区大小。推荐将 buffer_size 设置成 tf.data.experimental.AUTOTUNE，代表由系统自动调节缓存大小。

表 4-1 中完整的代码在随书的配套资源代码文件“4-7 Dataset 对象的操作方法.py”中。

一般来讲，处理数据集比较合理的步骤是：

- (1) 创建数据集。
- (2) 乱序数据集（shuffle）。
- (3) 重复数据集（repeat）。
- (4) 变换数据集中的元素（map）。
- (5) 设定批次（batch）。
- (6) 设定缓存（prefetch）。



提示：

在处理数据集的步骤中，第（5）步必须放在第（3）步后面，否则在训练时会产生某批次数据不足的情况。在模型与批次数据强耦合的情况下，如果输入模型的批次数据不足，则训练过程会出错。

造成这种情况的原因是：如果数据总数不能被批次整除，则在批次组合时会剩下一些不足一批次的数据；而在训练过程中，这些剩下的数据也会进入模型。

如果先对数据集进行重复（repeat）操作，则不会在设定批次（batch）操作过程中出现剩余数据的情况。

另外，还可以在 batch 函数中将参数 drop_remainder 设为 True。这样，在设定批次（batch）操作过程中，系统将会把剩余的数据丢弃。这也可以起到避免出现批次数据不足的问题。

4.6.4 代码实现：以元组和字典的方式生成 Dataset 对象

用 tf.data.Dataset.from_tensor_slices 接口分别以元组和字典的方式，将 $y \approx 2x$ 模拟数据集转为 Dataset 对象——dataset（元组方式数据集）、dataset2（字典方式数据集）。

代码 4-8 将内存数据转成 DataSet 数据集

```

01 import tensorflow as tf
02 import numpy as np
03
04 #在内存中生成模拟数据
05 def GenerateData(datasize = 100):
06     train_X = np.linspace(-1, 1, datasize) #定义在-1~1之间连续的100个浮点数
07     train_Y = 2 * train_X + np.random.randn(*train_X.shape) * 0.3 #y=2x,
但是加入了噪声

```

```

08     return train_X, train_Y      #以生成器的方式返回
09
10 train_data = GenerateData()
11
12 #将内存数据转化成数据集
13 dataset = tf.data.Dataset.from_tensor_slices( train_data )#以元组的方式生成
   数据集
14 dataset2 = tf.data.Dataset.from_tensor_slices( {           #以字典的方式生成数据集
15         "x":train_data[0],
16         "y":train_data[1]
17     } )

```

代码第 10 行，定义的变量 train_data 是内存中的模拟数据集。

代码第 14 行，以字典方式生成的 Dataset 对象 dataset2。在 dataset2 对象中，用字符串“x”“y”作为数据的索引名称。索引名称相当于字典类型数据中的 key，用于读取数据。

4.6.5 代码实现：对 Dataset 对象中的样本进行变换操作

依照实例的要求，对 Dataset 对象中的样本依次进行批次组合、类型转换和乱序操作。具体代码如下。

代码 4-8 将内存数据转成 DataSet 数据集（续）

```

18 batchsize = 10                      #定义批次样本个数
19 dataset3 = dataset.repeat().batch(batchsize)    #按批次组合数据集
20
21 dataset4 = dataset2.map(lambda data:
   (data['x'],tf.cast(data['y'],tf.int32)) )       #转化数据集中的元素
22 dataset5 = dataset.shuffle(100)            #乱序数据集

```

在本小节代码中，一共生成了 3 个新的数据集——dataset3、dataset4、dataset5。具体解读如下：

- 代码第 18、19 行，对数据集进行批次组合操作，生成了数据集 dataset3。首先调用数据集对象 dataset 的 repeat 方法，将数据集对象 dataset 变为可以无限制重复的循环数据集；接着调用 batch 方法，将数据集对象 dataset 中的样本按照 batchsize 大小进行划分（batchsize 大小为 10，即按照 10 条一批次来划分），这样每次从数据集 dataset3 中取出的数据都是以 10 条为单位的。
- 代码第 21 行，对数据集中的元素进行自定义转化操作，生成了数据集 dataset4。这里用匿名函数将字典类型中 key 值为 y 的数据转化成整形。有关匿名函数的更多知识请参考《Python 带我起飞——入门、进阶、商业实战》一书的 6.3 节。
- 代码第 22 行，对数据集做乱序操作，生成了数据集 dataset5。这里调用了 shuffle 函数，并传入参数 100。这样可以让数据乱序得更充分。

4.6.6 代码实现：创建 Dataset 迭代器

在本实例中，通过迭代器的方式从数据集中取数据。具体步骤如下：

- (1) 调用数据集 Dataset 对象的 make_one_shot_iterator 方法，生成一个迭代器 iterator。
- (2) 调用迭代器的 get_next 方法，获得一个元素。

具体代码如下：

代码 4-8 将内存数据转成 DataSet 数据集（续）

```

23 def getone(dataset):
24     iterator = dataset.make_one_shot_iterator() #生成一个迭代器
25     one_element = iterator.get_next()           #从 iterator 里取出一个元素
26     return one_element
27
28 one_element1 = getone(dataset)               #从 dataset 里取出一个元素
29 one_element2 = getone(dataset2)              #从 dataset2 里取出一个元素
30 one_element3 = getone(dataset3)              #从 dataset3 里取出一个批次的元素
31 one_element4 = getone(dataset4)              #从 dataset4 里取出一个元素
32 one_element5 = getone(dataset5)              #从 dataset5 里取出一个元素

```

代码第 23 行的函数 getone 用于返回数据集中具体元素的张量。

代码第 28~32 行，分别将制作好的数据集 dataset、dataset2、dataset3、dataset4、dataset5 传入函数 getone，依次得到对应数据集中的第 1 个元素。



提示：

代码第 24 行，用 make_one_shot_iterator 方法创建数据集迭代器。该方法内部会自动实现迭代器的初始化。如果不使用 make_one_shot_iterator 方法，则需要在会话（session）中手动对迭代器进行初始化。如：

```

iterator = dataset.make_initializable_iterator()    #直接生成迭代器
one_element1 = iterator.get_next()                 #生成元素张量
with tf.Session() as sess:
    sess.run(iterator.initializer)                #在会话（session）中对迭代器进行初始化
    .....

```

另外，在 TensorFlow 中还有一些其他方式可用来迭代数据集，以适应更多的场景。具体可以参考 4.9 与 4.10 节。

4.6.7 代码实现：在会话中取出数据

由于运行框架是静态图，所以整个过程中的数据都是以张量类型存在的。必须将数据放入会话（session）中的 run 方法进行计算，才能得到真实的值。

定义函数 showone 与 showbatch，分别用于获取数据集中的单个数据与多个数据。

具体代码如下。

代码 4-8 将内存数据转成 DataSet 数据集（续）

```

33 def showone(one_element,datasetname):          # 定义函数，用于显示单个数据
34     print('{0:-^50}'.format(datasetname))        # 分隔符
35     for ii in range(5):
36         datav = sess.run(one_element)            # 通过静态图注入的方式传入数据
37         print(datasetname,"-",ii,"| x,y:",datav) # 分隔符
38
39 def showbatch(onebatch_element,datasetname):    # 定义函数，用于显示批次数据
40     print('{0:-^50}'.format(datasetname))
41     for ii in range(5):
42         datav = sess.run(onebatch_element)        # 通过静态图注入的方式传入数据
43         print(datasetname,"-",ii,"| x.shape:",np.shape(datav[0]),"|"
44             x[:3]:" ,datav[0][:3])
45         print(datasetname,"-",ii,"| y.shape:",np.shape(datav[1]),"|"
46             y[:3]:" ,datav[1][:3])
47
48 with tf.Session() as sess:                      # 建立会话(session)
49     showone(one_element1,"dataset1")           # 调用 showone 函数，显示一条数据
50     showone(one_element2,"dataset2")
51     showbatch(one_element3,"dataset3")          # 调用 showbatch 函数，显示一批次数据
52     showone(one_element4,"dataset4")
53     showone(one_element5,"dataset5")

```

代码第 34、40 行，是输出一个格式化字符串的功能代码。该代码会输出一个分割符，使结果看起来更工整。

有关字符串格式化模板的更多信息，可以参考《Python 带我起飞——入门、进阶、商业实战》一书的 4.4.3 小节。

4.6.8 运行程序

整个代码运行后，输出以下结果：

```

-----dataset1-----
dataset1 - 0 | x,y: (-1.0, -2.1244706266287157)
dataset1 - 1 | x,y: (-0.9797979797979798, -1.9726405683713444)
dataset1 - 2 | x,y: (-0.9595959595959596, -1.6247158752571687)
dataset1 - 3 | x,y: (-0.9393939393939394, -1.9846861456039562)
dataset1 - 4 | x,y: (-0.9191919191919192, -1.9161218907604878)

-----dataset2-----
dataset2 - 0 | x,y: {'x': -1.0, 'y': -2.1244706266287157}
dataset2 - 1 | x,y: {'x': -0.9797979797979798, 'y': -1.9726405683713444}
dataset2 - 2 | x,y: {'x': -0.9595959595959596, 'y': -1.6247158752571687}
dataset2 - 3 | x,y: {'x': -0.9393939393939394, 'y': -1.9846861456039562}
dataset2 - 4 | x,y: {'x': -0.9191919191919192, 'y': -1.9161218907604878}

-----dataset3-----

```

```

dataset3 - 0 | x.shape: (10,) | x[:3]: [-1.          -0.97979798 -0.95959596]
dataset3 - 0 | y.shape: (10,) | y[:3]: [-2.12447063 -1.97264057 -1.62471588]
dataset3 - 1 | x.shape: (10,) | x[:3]: [-0.7979798 -0.77777778 -0.75757576]
dataset3 - 1 | y.shape: (10,) | y[:3]: [-1.77361254 -1.71638089 -1.6188056 ]
dataset3 - 2 | x.shape: (10,) | x[:3]: [-0.5959596 -0.57575758 -0.555555556]
dataset3 - 2 | y.shape: (10,) | y[:3]: [-0.80146675 -1.1920661 -0.99146132]
dataset3 - 3 | x.shape: (10,) | x[:3]: [-0.39393939 -0.37373737 -0.35353535]
dataset3 - 3 | y.shape: (10,) | y[:3]: [-1.41878264 -0.97009554 -0.81892304]
dataset3 - 4 | x.shape: (10,) | x[:3]: [-0.19191919 -0.17171717 -0.15151515]
dataset3 - 4 | y.shape: (10,) | y[:3]: [-0.11564091 -0.6592607  0.16367008]
-----dataset4-----
dataset4 - 0 | x,y: (-1.0, -2)
dataset4 - 1 | x,y: (-0.9797979797979798, -1)
dataset4 - 2 | x,y: (-0.9595959595959596, -1)
dataset4 - 3 | x,y: (-0.9393939393939394, -1)
dataset4 - 4 | x,y: (-0.9191919191919192, -1)
-----dataset5-----
dataset5 - 0 | x,y: (-0.5353535353535352, -1.0249665887548258)
dataset5 - 1 | x,y: (0.39393939393939403, 0.6453621496727984)
dataset5 - 2 | x,y: (0.232323232323232325, 0.641307921857285)
dataset5 - 3 | x,y: (0.6161616161616164, 0.8879358507776747)
dataset5 - 4 | x,y: (0.7373737373737375, 1.60192581924349)

```

在结果中，每个分隔符都代表一个数据集，在分割符下面显示了该数据集中的数据。

- dataset1：元组数据的内容。
- dataset2：字典数据的内容。
- dataset3：批次数据的内容。可以看到，每个 x、y 的都有 10 条数据。
- dataset4：将 dataset2 转化后的结果。可以看到，y 的值被转成了一个整数。
- dataset5：将 dataset1 乱序后的结果。可以看到，前 5 条的 x 数据与 dataset1 中的完全不同，并且没有规律。

4.6.9 使用 tf.data.Dataset.from_tensor_slices 接口的注意事项

在 tf.data.Dataset.from_tensor_slices 接口中，如果传入的是列表类型对象，则系统将其中的元素当作数据来处理；而如果传入的是元组类型对象，则将其中的元素当作列来拆开。这是值得注意的地方。

下面举例演示：

代码 4-9 from_tensor_slices 的注意事项

```

01 import tensorflow as tf
02
03 #传入列表对象
04 dataset1 = tf.data.Dataset.from_tensor_slices( [1,2,3,4,5] )
05 def getone(dataset):
06     iterator = dataset.make_one_shot_iterator() #生成一个迭代器
07     one_element = iterator.get_next()           #从 iterator 里取出一个元素

```

```

08     return one_element
09
10 one_element1 = getone(dataset1)
11
12 with tf.Session() as sess:           #建立会话(session)
13     for i in range(5):               #通过for循环打印所有的数据
14         print(sess.run(one_element1)) #用sess.run读出Tensor值

```

运行代码，输出以下结果：

```

1
2
3
4
5

```

结果中显示了列表中的所有数据，这是正常的结果。

1. 错误示例

如果将代码第4行传入的列表对象改成元组对象，则代码如下：

```
dataset1 = tf.data.Dataset.from_tensor_slices((1,2,3,4,5)) #传入元组对象
```

代码运行后将会报错，输出以下结果：

```

.....
IndexError: list index out of range

```

报错的原因是：函数from_tensor_slices自动将外层的元组拆开，将里面的每个元素当作一个列的数据。由于每个元素只是一个具体的数字，并不是数组，所以报错。

2. 修改办法

将数据中的每个数字改成数组，即可避免错误发生，具体代码如下：

```

dataset1 = tf.data.Dataset.from_tensor_slices( ([1],[2],[3],[4],[5]) )
one_element1 = getone(dataset1)
with tf.Session() as sess:           #建立会话(session)
    print(sess.run(one_element1))   #用sess.run读出Tensor值

```

则代码运行后，输出以下结果：

```
(1, 2, 3, 4, 5)
```

4.7 实例7：将图片文件制作成Dataset数据集

本实例将前面4.5节与4.6节的内容综合起来，将图片转为Dataset数据集，并进行更多的变换操作。

实例描述

有两个文件夹，分别放置男人与女人的照片。

现要求：

- (1) 将两个文件夹中的图片制作成 Dataset 的数据集；
- (2) 对图片进行尺寸大小调整、随机水平翻转、随机垂直翻转、按指定角度翻转、归一化、随机明暗度变化、随机对比度变化操作，并将其显示出来。

在图片训练过程中，一个变形丰富的数据集会使模型的精度与泛化性成倍地提升。一套成熟的代码，可以使开发数据集的工作简化很多。

本实例中使用的样本与 4.5 节实例中使用的样本完全一致。具体的样本内容可参考 4.5.1 小节。

4.7.1 代码实现：读取样本文件的目录及标签

定义函数 `load_sample`，用来将样本图片的目录名称及对应的标签读入内存。该函数与 4.5.2 小节中介绍的 `load_sample` 函数完全一样。具体代码可参考 4.5.2 小节。

4.7.2 代码实现：定义函数，实现图片转换操作

定义函数 `distorted_image`，用 TensorFlow 自带的 API 实现单一图片的变换处理。函数 `distorted_image` 的结果不能直接输出，需要通过会话形式进行显示。

具体代码如下：

代码 4-10 将图片文件制作成 Dataset 数据集

```

01 def distorted_image(image, size, ch=1, shuffleflag = False, cropflag = False,
02                      brightnessflag=False, contrastflag=False): # 定义函数
03     distorted_image = tf.image.random_flip_left_right(image)
04
05     if cropflag == True: # 随机裁剪
06         s = tf.random_uniform((1,2), int(size[0]*0.8), size[0], tf.int32)
07         distorted_image = tf.random_crop(distorted_image,
08                                         [s[0][0], s[0][0], ch])
09         # 上下随机翻转
10         distorted_image = tf.image.random_flip_up_down(distorted_image)
11         if brightnessflag == True: # 随机变化亮度
12             distorted_image =
13                 tf.image.random_brightness(distorted_image, max_delta=10)
14             if contrastflag == True: # 随机变化对比度
15                 distorted_image =
16                     tf.image.random_contrast(distorted_image, lower=0.2, upper=1.8)
17             if shuffleflag==True:
18                 distorted_image = tf.random_shuffle(distorted_image) # 沿着第 0 维打乱
19                 # 顺序
20
21     return distorted_image

```

在函数`_distorted_image`中使用的图片处理方法在实际应用中很常见。这些方法是数据增强操作的关键部分，主要用在模型的训练过程中。

4.7.3 代码实现：用自定义函数实现图片归一化

定义函数`norm_image`，用来实现对图片的归一化。由于图片的像素值是0~255之间的整数，所以直接除以255便可以得到归一化的结果。具体代码如下：

代码4-10 将图片文件制作成Dataset数据集（续）

```
17 def _norm_image(image, size, ch=1, flattenflag = False): # 定义函数，实现归一化，并且拍平
18     image_decoded = image/255.0
19     if flattenflag==True:
20         image_decoded = tf.reshape(image_decoded, [size[0]*size[1]*ch])
21     return image_decoded
```

本实例只用最简单的归一化处理，将图片的值域变化为0~1之间的小数。在实际开发中，还可以将图片的值域变化为-1~1之间的小数，让其具有更大的值域。

4.7.4 代码实现：用第三方函数将图片旋转30°

定义函数`random_rotated30`实现图片旋转功能。在函数`random_rotated30`中，用`skimage`库函数将图片旋转30°。`skimage`库需要额外安装，具体的安装命令如下：

```
pip install scikit-image
```

在整个数据集的处理流程中，对图片的操作都是基于张量进行变化的。因为第三方函数无法操作TensorFlow中的张量，所以需要对其进行额外的封装。

用`tf.py_function`函数可以将第三方库函数封装为一个TensorFlow中的操作符（OP）。具体代码如下：

代码4-10 将图片文件制作成Dataset数据集（续）

```
22 from skimage import transform
23 def _random_rotated30(image, label):# 定义函数，实现图片随机旋转操作
24
25     def _rotated(image): # 封装好的 skimage 模块，将进行图片旋转 30°
26         shift_y, shift_x = np.array(image.shape.as_list()[:2], np.float32) /
27             2.0
28         tf_rotate = transform.SimilarityTransform(rotation=np.deg2rad(30))
29         tf_shift = transform.SimilarityTransform(translation=[-shift_x,
30             -shift_y])
31         tf_shift_inv = transform.SimilarityTransform(translation=[shift_x,
32             shift_y])
33         image_rotated = transform.warp(image, (tf_shift + (tf_rotate +
34             tf_shift_inv)).inverse)
35         return image_rotated
```

```

32
33     def _rotatedwrap():
34         image_rotated = tf.py_function(_rotated,[image],[tf.float64]) #调
35         return tf.cast(image_rotated,tf.float32)[0]
36
37     a = tf.random_uniform([1],0,2,tf.int32) #实现随机功能
38     image_decoded = tf.cond(tf.equal(tf.constant(0),a[0]),lambda:
39         image,_rotatedwrap)
40     return image_decoded, label

```

为了实现随机转化的功能，使用了 TensorFlow 中的 `tf.cond` 方法，用来根据随机条件判断是否需要对本次图片进行旋转（见代码第 38 行）。



提示：

本实例使用第三方函数进行图片旋转处理，主要是为了演示函数 `tf.py_function` 的使用方法。如果仅要实现旋转的功能，则可以直接用 TensorFlow 中的函数 `tf.contrib.image.rotate` 来实现。具体用法见 11.2.7 小节的实例。

4.7.5 代码实现：定义函数，生成 Dataset 对象

在函数 `dataset` 中，用内置函数 `_parseone` 将所有的文件名称转化为具体的图片内容，并返回 `Dataset` 对象。具体代码如下：

代码 4-10 将图片文件制作成 Dataset 数据集（续）

```

41 def dataset(directory,size,batchsize,random_rotated=False):      #定义函数,
42     创建数据集
43     """ parse dataset """
44     (filenames,labels),_ = load_sample(directory,shuffleflag=False) #载入文
件名称与标签
45     def _parseone(filename, label):          #解析一个图片文件
46         """读取并处理每张图片"""
47         image_string = tf.read_file(filename)      #读取整个文件
48         image_decoded = tf.image.decode_image(image_string)
49         image_decoded.set_shape([None, None, None]) #对图片做扭曲变化
50         image_decoded = _distorted_image(image_decoded,size)
51         image_decoded = tf.image.resize(image_decoded, size) #变化尺寸
52         image_decoded = _norm_image(image_decoded,size) #归一化
53         image_decoded = tf.cast(image_decoded, dtype=tf.float32)
54         label = tf.cast(tf.reshape(label, []),dtype=tf.int32) #将
55         label 转为张量
56         return image_decoded, label
57     #生成 Dataset 对象
58     dataset = tf.data.Dataset.from_tensor_slices((filenames, labels))

```

```

57 dataset = dataset.map(_parseone)           #转化为图片数据集
58
59 if random_rotated == True:
60     dataset = dataset.map(_random_rotated30)
61
62 dataset = dataset.batch(batchsize)         #批次组合数据集
63
64 return dataset

```

4.7.6 代码实现：建立会话，输出数据

首先，定义两个函数——showresult 和 showimg，用于将图片数据进行可视化输出。

接着，创建两个数据集 dataset、dataset2：

- dataset 是一个批次为 10 的数据集，支持随机反转、尺寸转化、归一化操作。
- dataset2 在 dataset 的基础上，又支持将图片旋转 30°。

定义好数据集后建立会话（session），然后通过会话（session）的 run 方法获得数据并将其显示出来。

具体代码如下：

代码 4-10 将图片文件制作成 Dataset 数据集（续）

```

65 def showresult(subplot,title,thisimg):          #显示单个图片
66     p=plt.subplot(subplot)
67     p.axis('off')
68     p.imshow(thisimg)
69     p.set_title(title)
70
71 def showimg(index,label,img,ntop):               #显示结果
72     plt.figure(figsize=(20,10))                  #定义显示图片的宽、高
73     plt.axis('off')
74     ntop = min(ntop,9)
75     print(index)
76     for i in range(ntop):
77         showresult(100+10*ntop+1+i,label[i],img[i])
78     plt.show()
79
80 def getone(dataset):
81     iterator = dataset.make_one_shot_iterator() #生成一个迭代器
82     one_element = iterator.get_next()           #从 iterator 里取出一个元素
83     return one_element
84
85 sample_dir=r"man_woman"
86 size = [96,96]
87 batchsize = 10
88 tdataset = dataset(sample_dir,size,batchsize)
89 tdataset2 = dataset(sample_dir,size,batchsize,True)

```

```

90 print(tdataset.output_types)          #打印数据集的输出信息
91 print(tdataset.output_shapes)
92
93 one_element1 = getone(tdataset)      #从 tdataset 里取出一个元素
94 one_element2 = getone(tdataset2)     #从 tdataset2 里取出一个元素
95
96 with tf.Session() as sess:          #建立会话 (session)
97     sess.run(tf.global_variables_initializer()) #初始化
98
99     try:
100         for step in np.arange(1):
101             value = sess.run(one_element1)
102             value2 = sess.run(one_element2)
103             #显示图片
104             showimg(step,value2[1],np.asarray(value2[0]*255,np.uint8),10)
105             showimg(step,value2[1],np.asarray(value2[0]*255,np.uint8),10)
106
107     except tf.errors.OutOfRangeError:    #捕获异常
108         print("Done!!!")

```

这部分代码与 4.6.7 小节、4.5.5 小节中的代码比较类似，不再详述。

4.7.7 运行程序

整个代码运行后，输出如下结果：



图 4-9 实例 7 程序运行结果 (a)



图 4-9 实例 7 程序运行结果 (b)

在输出结果中有两张图：

- 图 4-9 (a) 是数据集 tdataset 中的内容。该数据集对原始图片进行了随机裁剪，并将尺寸变成了边长为 96pixel 的正方形。
- 图 4-9 (b) 是数据集 tdataset2 中的内容。该数据集在 tdataset 的变换基础上，进行了随机 30° 的旋转。



提示：

skimage 库是一个很强大的图片转化库，读者还可以在其中找到更多有关图片变化的

功能。

本实例中介绍了第三方库与 tf.data.Dataset 接口结合使用的方法，需要读者掌握。通过这个方法可以将所有的第三方库与 tf.data.Dataset 接结合起来使用，以实现更强大的数据预处理功能。

4.8 实例 8：将 TFRecord 文件制作成 Dataset 数据集

tf.data 接口是生成 Dataset 数据集的高级接口，可以将多种格式的样本文件转化成 Dataset 数据集。其中包括：文本格式、二进制格式、TFRecord 格式。在 tf.data 接口中，针对不同格式的样本文件，提供了对应的转化函数。具体如下。

- `tf.data.TextLineDataset`: 根据文本文件生成 Dataset 对象。支持单个或多个文件读取，将文件中的每一行转化为 Dataset 对象中的每个元素。该方法可以用来读入 CSV 文件，见 4.6.3 小节中表 4-1 的内容描述和 7.2 节的实例代码。
- `tf.data.FixedLengthRecordDataset`: 该方法专门用于读入数据源是二进制格式的文件，根据二进制文件生成 Dataset 对象。它支持单个或多个文件读取。在使用时，需要传入文件列表和每次读取二进制数据的长度。该方法会将文件中指定的二进制长度转化为 Dataset 对象中的每个元素。
- `tf.data.TFRecordDataset`: 该方法用于读入数据源是 TFRecord 格式的文件(见 4.1.2 小节)。它可以将 TFRecord 文件中的 TFExample 对象转成 Dataset 数据集中的元素。



提示：

如果是在 Linux 中使用 tf.data 接口，则样本的文件名尽量用英文。

作者在测试时发现，1.8 版本的 `tf.data.TextLineDataset` 接口在 Ubuntu 16.04 系统中，找不到文件名为中文的文件。如果文件名使用英文命名，则会省去很多额外的麻烦。

本实例用 tf.data 接口将 TFRecord 文件制作成 Dataset 数据集。

实例描述

有一个 TFRecord 格式的数据集，里面的内容为男人与女人的照片。

现要求：将 TFRecord 格式的数据集载入内存中，将其转化为 Dataset 数据集，并将返回的图片显示出来。

在程序中使用 TFRecord 格式的数据集，是 TensorFlow 早期版本的主流开发方式。在学习或工作过程中，也常会遇到 TFRecord 格式的数据集。下面就来演示如何用 `tf.data.Dataset` 接口加载 TFRecord 格式的数据集。

4.8.1 样本介绍

将 4.5 节实例中生成的 TFRecord 格式文件复制一份，放到代码同级目录下，如图 4-10 所示。

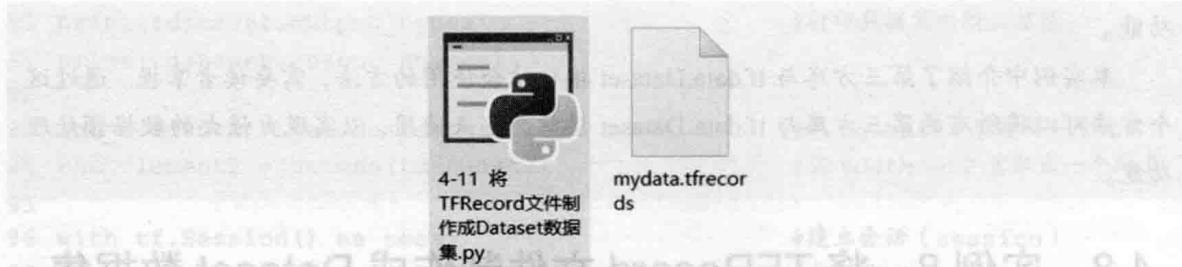


图 4-10 TFRecord 格式文件

4.8.2 代码实现：定义函数，生成 Dataset 对象

定义函数 dataset，并实现以下步骤：

- (1) 用 `tf.data.TFRecordDataset` 接口读取 TFRecord 数据文件，并将其转为 Dataset 对象。
- (2) 用 Dataset 对象的 `map` 方法将该数据集中的每条数据放到内置函数 `_parseone` 中，解析成具体的图片与标签。
- (3) 用 Dataset 对象的 `batch` 方法将数据集按批次组合，并将组合后的结果返回。

具体代码如下：

代码 4-11 将 TFRecord 文件制作成 Dataset 数据集

```

01 import tensorflow as tf
02 from PIL import Image
03 import numpy as np
04 import matplotlib.pyplot as plt
05
06 def dataset(directory,size,batchsize):          # 定义函数，创建数据集
07     """ parse dataset """
08     def _parseone(example_proto):                  # 解析一个图片文件
09         """ Reading and handle image """
10         # 定义解析的字典
11         dics = {}
12         dics['label'] = tf.FixedLenFeature(shape=[],dtype=tf.int64)
13         dics['img_raw'] = tf.FixedLenFeature(shape=[],dtype=tf.string)
14         # 解析一行样本
15         parsed_example = tf.parse_single_example(example_proto,dics)
16
17         image = tf.decode_raw(parsed_example['img_raw'],out_type=tf.uint8)
18         image = tf.reshape(image, size)
19         image = tf.cast(image,tf.float32)*(1./255)-0.5 # 对图像数据做归一化处理
20
21         label = parsed_example['label']
22         label = tf.cast(label,tf.int32)
23         label = tf.one_hot(label, depth=2, on_value=1)    # 转为 One-hot 编码
24         return image,label
25

```

```

26     dataset = tf.data.TFRecordDataset(directory)
27     dataset = dataset.map(_parseone)
28     dataset = dataset.batch(batchsize)           #按批次组合数据集
29     dataset = dataset.prefetch(batchsize)
30     return dataset

```

在代码第8行的内建函数_parseone中可以看到，整个过程与4.5.4小节读取TFRecord数据集的方式非常相似——也是定义一个字典作为参数，并按照该字典的形状和类型来解析数据。

4.8.3 代码实现：建立会话输出数据

本小节代码的步骤如下：

- (1) 定义两个函数——showresult和showimg，用于将图片数据进行可视化输出。
- (2) 创建一个批次为10条数据的数据集dataset对象。
- (3) 在dataset对象中，将每10条数据组合在一起形成一个批次。
- (4) 建立会话(session)，并通过会话(session)的run方法获得数据。
- (5) 将取到的数据显示出来。

具体代码如下：

代码4-11 将TFRecord文件制作成Dataset数据集（续）

```

31 def showresult(subplot,title,thisimg):                      #显示单个图片
32     p = plt.subplot(subplot)
33     p.axis('off')
34     p.imshow(thisimg)
35     p.set_title(title)
36
37 def showimg(index,label,img,ntop):                         #显示结果
38     plt.figure(figsize=(20,10))                            #定义显示图片的宽、高
39     plt.axis('off')
40     ntop = min(ntop,9)
41     print(index)
42     for i in range(ntop):
43         showresult(100+10*ntop+1+i,label[i],img[i])
44     plt.show()
45
46 def getone(dataset):
47     iterator = dataset.make_one_shot_iterator() #生成一个迭代器
48     one_element = iterator.get_next()          #从iterator里取出一个元素
49     return one_element
50
51 sample_dir=['mydata.tfrecords']
52 size = [256,256,3]
53 batchsize = 10
54 tdataset = dataset(sample_dir,size,batchsize)
55

```

```

56 print(tdataset.output_types)          # 打印数据集的输出信息
57 print(tdataset.output_shapes)
58 one_element1 = getone(tdataset)       # 从 tdataset 里取出一个元素
59
60
61 with tf.Session() as sess:           # 建立会话 (session)
62     sess.run(tf.global_variables_initializer()) # 初始化
63 try:                                # 异常处理
64     for step in np.arange(1):
65         value = sess.run(one_element1)
66         showimg(step,value[1],
67                    np.asarray((value[0]+0.5)*255,np.uint8),10) # 显示图片
68     except tf.errors.OutOfRangeError: # 捕获异常
69         print("Done!!!!")

```

这部分代码与前面 4.7.7 小节比较类似，不再详述。

4.8.4 运行程序

整个代码运行后，输出以下结果：



图 4-11 实例 8 程序运行结果

在输出结果中可以看到，每张图片的标题都显示了两个数字。这是由于，在处理数据集过程中对标签数据进行了 one-hot 编码（见代码第 32 行）。

4.9 实例 9：在动态图中读取 Dataset 数据集

从 TensorFlow 1.4 版本开始，动态图（读者对动态图先有一个概念即可，在 6.1.3 小节会详细介绍）的功能越来越完善。到了 1.8 版本，对 `tf.data.Dataset` 接口的支持变得更加友好。使用动态图操作 `Dataset` 数据集，就如同从普通序列对象中取数据一样简单。

在 TensorFlow 2.0 版本中，动态图已经取代静态图成为系统默认的开发框架。

实例描述

将 4.7 节中的数据以 TensorFlow 动态图的方式显示出来。

该实例重用了 4.7 节的部分代码制作数据集，然后用动态图框架读取数据集的内容。

4.9.1 代码实现：添加动态图调用

在代码的最开始位置引入相关模块，并启用动态图功能。

**提示：**

启动动态图的语句必须在其他所有语句之前执行（见下面代码第8行）。

代码如下：

代码 4-12 在动态图里读取 Dataset 数据集

```

01 import os
02 import tensorflow as tf
03
04 from sklearn.utils import shuffle
05 import numpy as np
06 import matplotlib.pyplot as plt
07
08 tf.enable_eager_execution()          #启用动态图
09 print("TensorFlow 版本: {}".format(tf.__version__))    #打印版本，确保是 1.8
以后的版本
10 print("Eager execution: {}".format(tf.executing_eagerly()))  #验证动态图
是否启动

```

4.9.2 制作数据集

制作数据集的内容与4.7节完全一致。可以将4.7.1~4.7.6小节中的代码完全移到本实例中。

4.9.3 代码实现：在动态图中显示数据

将4.7.1~4.7.6小节中的代码复制到本实例中之后，接着添加以下代码即可将数据内容显示出来。

代码 4-12 在动态图里读取 Dataset 数据集（续）

```

11 for step,value in enumerate(tdataset):
12     showimg(step, value[1].numpy(),np.asarray( value[0]*255,np.uint8),10)
#显示图片

```

可以看到，这次的代码中没有再建立会话，而是直接将数据集用for循环的方式进行迭代读取。这就是动态图的便捷之处。

代码第12行中，对象value是一个带有具体值的张量。这里用该张量的numpy方法将张量value[1]中的值取出来。同样，还可以用np.asarray的方式直接将张量value[0]转化为numpy类型的数组。

代码运行后显示以下结果：

```

TensorFlow 版本: 1.13.1
Eager execution: True
loading sample dataset..
loading sample dataset..
loading sample dataset..

```

```
(tf.float32, tf.int32)
(TensorShape([Dimension(None), Dimension(96), Dimension(96), Dimension(None)]),
TensorShape([Dimension(None)]))
```

0



图 4-12 实例 9 程序运行结果 (a)

1



图 4-12 实例 9 程序运行结果 (b)

本实例用 `tf.data.Dataset` 接口的可迭代特性，实现对数据的读取。

更多数据集迭代器的用法见 4.10 节。

4.9.4 实例 10：在 TensorFlow 2.x 中操作数据集

下面在代码文件“4-12 在动态图里读取 Dataset 数据集.py”的基础之上稍加调整，将其升级成可以支持 TensorFlow 2.x 版本的代码。

完整代码如下：

代码 4-13 在动态图里读取 Dataset 数据集_tf2 版

```
01 import os
02 import tensorflow as tf
03 from PIL import Image
04 from sklearn.utils import shuffle
05 import numpy as np
06 import matplotlib.pyplot as plt
07 print("TensorFlow 版本: {}".format(tf.__version__))
08 print("Eager execution: {}".format(tf.executing_eagerly()))
09
10 def load_sample(sample_dir, shuffleflag = True):
11     '''递归读取文件。只支持一级。返回文件名、数值标签、数值对应的标签名'''
12     print ('loading sample dataset..')
13     lfilenames = []
14     labelsnames = []
15     for (dirpath, dirnames, filenames) in os.walk(sample_dir):
16         for filename in filenames:
17             filename_path = os.sep.join([dirpath, filename])
18             lfilenames.append(filename_path) #添加文件名
19             labelsnames.append( dirpath.split('\\')[-1] )#添加文件名对应的标签
20
21     lab= list(sorted(set(labelsnames))) #生成标签名称列表
```

```

22     labdict=dict( zip( lab ,list(range(len(lab)))) ) #生成字典
23
24     labels = [labdict[i] for i in labelsnames]
25     if shuffleflag == True:
26         return
27     else:
28         return
29     (np.asarray( lfilenames),np.asarray( labels)),np.asarray(lab)
30     directory='man_woman\\' #定义样本路径
31     (filenames,labels),_ =load_sample(directory,shuffleflag=False)
32     #定义函数，实现增强数据操作
33     def _distorted_image(image,size,ch=1,shuffleflag = False,cropflag =
False,
34                           brightnessflag=False,contrastflag=False):
35         distorted_image =tf.image.random_flip_left_right(image)
36
37         if cropflag == True: #随机裁剪
38             s = tf.random.uniform((1,2),int(size[0]*0.8),size[0],tf.int32)
39             distorted_image = tf.image.random_crop(distorted_image,
[s[0][0],s[0][0],ch])
40         #上下随机翻转
41         distorted_image = tf.image.random_flip_up_down(distorted_image)
42         if brightnessflag == True: #随机变化亮度
43             distorted_image =
tf.image.random_brightness(distorted_image,max_delta=10)
44         if contrastflag == True: #随机变化对比度
45             distorted_image =
tf.image.random_contrast(distorted_image,lower=0.2, upper=1.8)
46         if shuffleflag==True:
47             distorted_image = tf.random.shuffle(distorted_image)#沿着第 0 维打乱
顺序
48         return distorted_image
49
50     #定义函数，实现归一化，并且拍平
51     def _norm_image(image,size,ch=1,flattenflag = False):
52         image_decoded = image/255.0
53         if flattenflag==True:
54             image_decoded = tf.reshape(image_decoded, [size[0]*size[1]*ch])
55         return image_decoded
56     from skimage import transform
57     def _random_rotated30(image, label): #定义函数，实现图片随机旋转操作
58         def _rotated(image): #用封装好的 skimage 模块将图片旋转 30°
59             shift_y, shift_x = np.array(image.shape[:2],np.float32) / 2.
60             tf_rotate = transform.SimilarityTransform(rotation=np.deg2rad(30))

```

```

61         tf_shift = transform.SimilarityTransform(translation=[-shift_x,
62 -shift_y])
63         tf_shift_inv = transform.SimilarityTransform(translation=[shift_x,
shift_y])
64         image_rotated = transform.warp(image, (tf_shift + (tf_rotate +
tf_shift_inv)).inverse)
65         return image_rotated
66
67     def _rotatedwrap():
68         image_rotated = tf.py_function(_rotated,[image],[tf.float64]) #调用第三方函数
69
70         return tf.cast(image_rotated,tf.float32)[0]
71
72     a = tf.random.uniform([1],0,2,tf.int32)#实现随机功能
73     image_decoded = tf.cond(tf.equal(tf.constant(0),a[0]),lambda:
image,_rotatedwrap)
74
75     return image_decoded, label
76 #定义函数，创建数据集
77 def dataset(directory,size,batchsize,random_rotated=False):
78     #载入文件的名称与标签
79     (filenames,labels),_ =load_sample(directory,shuffleflag=False)
80     def _parseone(filename, label): #解析一个图片文件
81         image_string = tf.io.read_file(filename) #读取整个文件
82         image_decoded = tf.image.decode_image(image_string)
83         image_decoded.set_shape([None, None, None])
84         image_decoded = _distorted_image(image_decoded,size)#扭曲图片
85         image_decoded = tf.image.resize(image_decoded, size) #变化尺寸
86         image_decoded = _norm_image(image_decoded,size)#归一化
87         image_decoded = tf.cast(image_decoded,dtype=tf.float32)
88         #将label 转为张量
89         label = tf.cast(tf.reshape(label, []),dtype=tf.int32 )
90
91         return image_decoded, label
92     #生成Dataset 对象
93     dataset = tf.data.Dataset.from_tensor_slices((filenames, labels))
94     dataset = dataset.map(_parseone) #有图片内容的数据集
95     if random_rotated == True:
96         dataset = dataset.map(_random_rotated30)
97     dataset = dataset.batch(batchsize) #批次划分数据集
98     return dataset
99
100    def showresult(subplot,title,thisimg): #显示单个图片
101        p =plt.subplot(subplot)
102        p.axis('off')
103        p.imshow(thisimg)
104        p.set_title(title)

```

```

103
104 def showimg(index,label,img,ntop):          #显示图片结果
105     plt.figure(figsize=(20,10))             #定义显示图片的宽、高
106     plt.axis('off')
107     ntop = min(ntop,9)
108     print(index)
109     for i in range(ntop):
110         showresult(100+10*ntop+1+i,label[i],img[i])
111     plt.show()
112
113 sample_dir=r"man_woman"
114 size = [96,96]
115 batchsize = 10
116 tdataset = dataset(sample_dir,size,batchsize)
117 tdataset2 = dataset(sample_dir,size,batchsize,True)
118 print(tdataset.output_types)               #打印数据集的输出信息
119 print(tdataset.output_shapes)
120
121 for step,value in enumerate(tdataset): #显示图片
122     showimg(step, value[1].numpy(),np.asarray(value[0]*255,np.uint8),10)

```

在TensorFlow 2.x版本中，将TensorFlow 1.x版本中的部分函数名字进行了调整，具体如下：

- 将函数tf.random_uniform改成了tf.random.uniform（见代码第38行）。
- 将函数tf.random_crop改成了tf.image.random_crop（见代码第39行）。
- 将函数tf.random_shuffle改成了tf.random.shuffle（见代码第47行）。
- 将函数tf.read_file改成了tf.io.read_file（见代码第79行）。

这些变化的函数名，都是可以通过工具自动转化的。读者可以直接使用TensorFlow 2.x版本中提供的工具，对TensorFlow 1.x版本的代码进行升级。具体命令如下：

```
tf_upgrade_v2 --infile "1.x的代码文件" -outfile "2.x的代码文件"
```

具体实例还可以参考本书6.13节。

代码运行后，可以输出与4.9.3小节一样的结果。这里不再详述。



提示：

如果将本实例代码第7行换作启动动态图的语句，即：

```
tf.enable_eager_execution()
```

则该代码也可以在TensorFlow 1.13.1版本上正常运行。这说明了一个问题：TensorFlow 2.x版本的内部代码与TensorFlow 1.13.1版本非常接近。TensorFlow 1.13.1版本既可以支持TensorFlow 1.x版本，又可以部分支持TensorFlow 2.x版本，具有更好的兼容性。

4.10 实例11：在不同场景中使用数据集

本节将演示数据集的其他几种迭代方式，分别对应不同的场景。

实例描述

在内存中定义一个数组，将其转化成 **Dataset** 数据集。在训练模型、测试模型、使用模型的场景中使用数据集，将数组中的内容输出来。

4.6、4.7、4.8 节中关于数据集的使用，更符合于训练模型场景的用法。可以通过用 `tf.data.Dataset` 接口的 `repeat` 方法来实现数据集的循环使用。在实际训练中，只能控制训练模型的迭代次数，无法直观地控制数据集的遍历次数。

4.10.1 代码实现：在训练场景中使用数据集

为了指定数据集的遍历次数，在创建迭代器时使用了 `from_structure` 方法，该方法没有自动初始化功能，所以需要在会话（`session`）中对其进行初始化。当整个数据集遍历结束后，会产生 `tf.errors.OutOfRangeError` 异常。通过在捕获 `tf.errors.OutOfRangeError` 异常的处理函数中对迭代器再次进行初始化的方式，将数据集内部的指针清零，让数据集可以再次从头遍历。



提示：

虽然在多次迭代过程中会频繁调用迭代器初始化函数，但这并不会影响整体性能。系统只是对迭代器做了初始化，并不是将整个数据集进行重新设置，所以这种方案是可行的。

具体代码如下：

代码 4-14 在不同场景中使用数据集

```

01 import tensorflow as tf
02
03 dataset1 = tf.data.Dataset.from_tensor_slices([1,2,3,4,5])#定义训练数据集
04
05 #创建迭代器
06 iterator1 = tf.data.Iterator.from_structure(dataset1.output_types,
07                                               dataset1.output_shapes)
08 one_element1 = iterator1.get_next()#获取一个元素
09
10 with tf.Session() as sess2:
11     sess2.run(iterator1.make_initializer(dataset1)) #初始化迭代器
12     for ii in range(2):#将数据集迭代两次
13         while True:#通过 for 循环打印所有的数据
14             try:
15                 print(sess2.run(one_element1))#调用 sess.run 读出 Tensor 值
16             except tf.errors.OutOfRangeError:
17                 print("遍历结束")
18             sess2.run(iterator1.make_initializer(dataset1))
19             break

```

整体代码运行后，输出以下结果：

```

1   #遍历数据集，将每条数据都输出到控制台，输出更多数据提升 8.01A
2
3
4
5
遍历结束
1
2
3
4
5
遍历结束

```

从结果中可以看出，整个数据集迭代运行了两遍。



提示：

代码中第6行的 `tf.data.Iterator.from_structure` 方法还可以换成 `dataset1.make_initializer`，一样可以实现通过初始化的方法实现从头遍历数据集的效果。

例如，代码中的第6~11行可以写成如下：

```

iterator = dataset1.make_initializer()      #直接生成迭代器
one_element1 = iterator.get_next()          #生成元素张量
with tf.Session() as sess2:                  #在会话（session）中需要对迭代器进行初始化
    sess.run(iterator.initializer)

```

4.10.2 代码实现：在应用模型场景中使用数据集

在应用模型场景中，可以将实际数据注入 Dataset 数据集中的元素张量，来实现输入操作。具体代码如下：

代码 4-14 在不同场景中使用数据集（续）

```
20   print(sess2.run(one_element1,{one_element1:356}))  #往数据集中注入数据
```

代码第20行，将数字“356”注入到张量 `one_element1` 中。此时的张量 `one_element1` 起到占位符的作用，这也是在使用模型场景中常用的做法。

整个代码运行后，输出以下结果：

```
356
```

从输出结果可以看出，“356”这个数字已经进入张量图并成功输出到屏幕上。



提示：

这种方式与迭代器的生成方式无关，所以它不仅适用于通过 `from_structure` 生成的迭代器，也适用于通过 `make_one_shot_iterator` 方法生成的迭代器。

4.10.3 代码实现：在训练与测试混合场景中使用数据集

在训练 AI 模型时，一般会有两个数据集：一个用于训练，一个用于测试。在 TensorFlow 中提供了一个便捷的方式，可以在训练过程中对训练与测试的数据源进行灵活切换。

具体的方式为：

- (1) 创建两个 Dataset 对象，一个用于训练、一个用于测试。
- (2) 分别建立两个数据集对应的迭代器——iterator（训练迭代器）、iterator_test（测试迭代器）。
- (3) 在会话中，分别建立两个与迭代器对应的句柄——iterator_handle（训练迭代器句柄）、iterator_handle_test（测试迭代器句柄）。
- (4) 生成占位符，用于接收迭代器句柄。
- (5) 生成关于占位符的迭代器，并定义其 get_next 方法取出的张量。

在运行时，直接将用于训练或测试的迭代器句柄输入占位符，即可实现数据源的使用。具体代码如下：

代码 4-14 在不同场景中使用数据集（续）

```

21 dataset1 = tf.data.Dataset.from_tensor_slices([1,2,3,4,5]) #创建训练 Dataset
    对象
22 iterator = dataset1.make_one_shot_iterator() #生成一个迭代器
23
24 dataset_test = tf.data.Dataset.from_tensor_slices([10,20,30,40,50]) #创建测试 Dataset 对象
25 iterator_test = dataset1.make_one_shot_iterator() #生成一个迭代器
26 #适用于测试与训练场景中的数据集方式
27 with tf.Session() as sess:
28     iterator_handle = sess.run(iterator.string_handle()) #创建迭代器句柄
29     iterator_handle_test = sess.run(iterator_test.string_handle()) #创建迭代器句柄
30
31 handle = tf.placeholder(tf.string, shape=[]) #定义占位符
32 iterator3 = tf.data.Iterator.from_string_handle(handle,
33 iterator.output_types)
34 one_element3 = iterator3.get_next() #获取元素
35 print(sess.run(one_element3,{handle: iterator_handle})) #取出元素
36 print(sess.run(one_element3,{handle: iterator_handle_test}))
```

运行代码后，显示以下结果：

```

1
10
```

其中，1 是训练集的第一个数据，10 是训练集中第 1 个数据。

由于篇幅限制，制作数据集的介绍到这里就结束了。

4.11 tf.data.Dataset 接口的更多应用

目前，tf.data.Dataset 接口是 TensorFlow 中主流的数据集接口。在编写自己的模型程序时，建议优先使用 tf.data.Dataset 接口。



提示：

本章除介绍了主流的 Dataset 数据集外，还介绍了一些其他形式的数据集（例如：内存对象数据集、TFRecord 格式的数据集）。这些内容是为了让读者对数据集这部分知识有一个全面的掌握，这样在阅读别人代码，或在别人的代码上做二次开发时，就不会出现技术盲区。

用 tf.data.Dataset 接口还可以将更多其他类型的样本制作成数据集。另外，也可以对 tf.data.Dataset 接口进行二次封装，使 tf.data.Dataset 接口用起来更为简单。



提示：

10.3.3 小节还介绍了一个同时支持静态图与动态图的工具类。它是对原有 tf.data.Dataset 接口的封装。读者可以直接拿来使用，以提升编写代码的效率。

更多的内容可以参考官网中的教程。

第 5 章

10分钟快速训练自己的图片分类模型

本章重点讲解微调技术，即用自己的数据集在预训练模型上进行二次训练。该技术可以在样本较少的情况下快速地训练出自己的可用模型。

通过本章的学习，读者可以用成熟模型快速训练出自己的图片分类器。

5.1 快速导读

在学习实例之前，有必要了解一下模型的基础知识。

5.1.1 认识模型和模型检查点文件

1. 什么是模型

模型是通过神经网络训练得来的，是机器运算后所产出的结果。用 TensorFlow 开发程序，最终的目的就是要得到模型。有了模型之后，便可以用模型来做一些对应的回归、分类等任务。

模型默认存在内存中，会随着程序的关闭而销毁。在关闭当前程序时，为了防止模型丢失，一般会把模型保存到文件里，以便下次使用。保存模型的文件，就是模型文件。

2. 什么是模型中的检查点

模型中的检查点，就好比游戏中的还原点。在训练模型过程中，可以将模型以文件的方式保存到硬盘上。这样在之后的训练中可以直接载入上次生成的检查点文件，接着上次的结果继续训练。

在训练中，引入检查点是非常有用的。在训练模型时，难免会出现中断的情况。及时将模型的训练成果保存下来，这样即使出现中断情况，也不会耽误模型的训练进度。

5.1.2 了解“预训练模型”与微调（Fine-Tune）

预训练模型等同于检查点文件。在使用时，既可以将检查点文件载入已有模型，接着训练；也可以将检查点文件载入到别的模型中，做二次开发。

这样，新的模型就会在原有模型的训练结果之上再进行训练，从而大大缩短了训练时间。这种二次开发被叫作微调（Fine-Tune）。

可以说微调是一种转移学习的技巧。它是指，将一个已经在相关任务上训练过的模型用在

新模型中重新使用，继续训练。在本章中，将调用一个通过 ImageNet 数据集训练好的模型，将该模型中的“提取图像特征”能力转移到现有的分类任务上。

该方法适用于中等量级（几千到几万个）的数据集。如果是大型数据集（数百万个），还是建议重头训练比较好。

5.1.3 学习 TensorFlow 中的预训练模型库——TF-Hub 库

TF-Hub 库是 TensorFlow 中专门用于预训练模型的库，其中包含很多在大型数据集上训练好的模型。如需在较小的数据集上实现识别任务，则可以通过微调这些预训练模型来实现。另外，它还能够提升原有模型在具体场景中的泛化能力，加快训练的速度。

TF-Hub 库可以支持 TensorFlow 的 1.x 与 2.x 版本。

1. 安装 TF-Hub 库

该库独立于 TensorFlow 安装包。如想使用，则需要额外安装。可以在命令行里输入以下命令：

```
pip install tensorflow-hub
```

2. TF-Hub 库的说明

在 GitHub 网站上还有 TF-Hub 库的源码链接，其中包含了众多详细的说明文档。地址如下：

```
https://github.com/tensorflow/hub
```

有兴趣的读者可以根据该链接中的文档内容自行学习。

5.2 实例 12：通过微调模型分辨男女

本实例是在第 3 章和第 4 章的基础上实现的。利用第 4 章的数据集制作方法，制作自己的数据集；然后使用自己的数据集，在已有的模型上展开二次训练。如何使用已有模型是第 3 章的内容；而如何用已有的模型做二次训练，则是本实例的内容。

实例描述

有一组照片，分为男人和女人。

训练模型来学习这些照片，让模型能够找到其中的规律。接着，用该模型对图片中的人物进行识别，区分其性别是“男”还是“女”。

本实例中，用 NASNet_A_Mobile 模型来做二次训练。具体过程分为 4 步：

- (1) 准备样本。
- (2) 准备 NASNet_A_Mobile 模型。
- (3) 编写代码进行二次训练。
- (4) 用已经训练好的模型进行测试。

5.2.1 准备工作

1. 准备样本

通过以下链接下载 CelebA 数据集：

<http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html>

待数据集下载完后，将其解压缩，并手动分出一部分男人与女人的照片。

在本实例中，一共用 20000 张图片来训练模型，其中：

- 训练样本由 8421 张男性头像和 11599 张女性头像构成（在 train 文件夹下）。

- 测试样本由 10 张男性头像和 10 张女性头像构成（在 val 文件夹下）。

部分样本数据如图 5-1 所示。

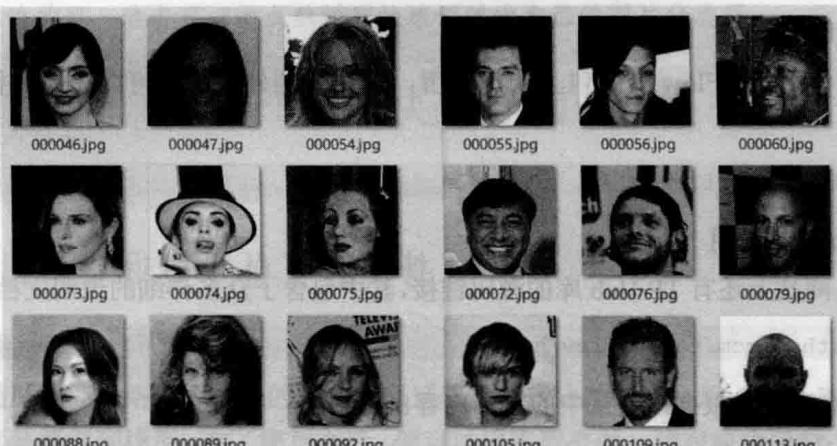


图 5-1 男女数据集样本示例

将样本整理好后，统一放到 data 文件夹下。

2. 准备代码环境并预训练模型

具体步骤如下。

- (1) 下载与部署 slim 模块。

该部分的内容与 3.1 节完全一样，这里不再详述。

- (2) 下载 NASNet_A_Mobile 模型。

该部分的内容与 3.1 节类似。在图 3-2 中，找到“nasnet-a_mobile_04_10_2017.tar.gz”的下载链接，将其下载并解压缩。

- (3) 完整代码文件的结构。

本实例是通过 4 个代码文件来实现的，具体文件及描述如下。

- 5-1 mydataset.py：处理男女图片数据集的代码。
- 5-2 model.py：加载预训练模型 NASNet_A_Mobile 并进行微调的代码。
- 5-3 train.py：训练模型的代码。
- 5-4 test.py：测试模型的代码。

部署时，将这4个代码文件与slim模块、NASNet_A_Mobile模型、样本一起放到一个文件夹下。完整代码文件的结构如图5-2所示。

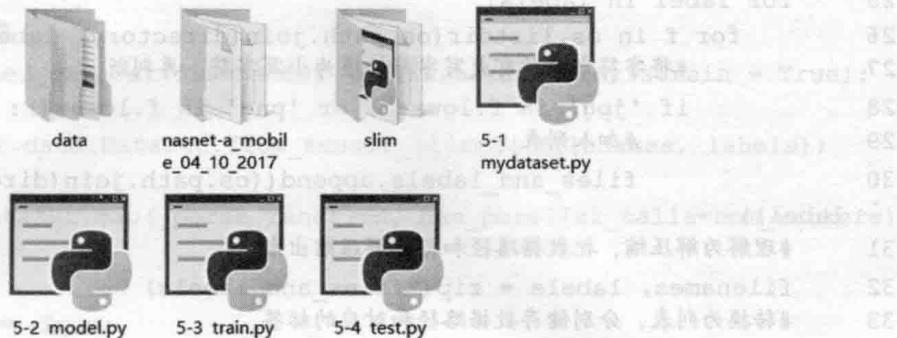


图5-2 分辨男女实例的文件结构

5.2.2 代码实现：处理样本数据并生成Dataset对象

本实例中，直接将数据集的相关操作封装到代码文件“5-1 mydataset.py”中。在该文件中包含用于训练与测试的数据集。

- 在训练模式下，会对数据进行乱序处理。
- 在测试模式下，按照数据的原始顺序直接使用。

这部分的知识在第4章已经有全面的介绍，这里不再详述。完整代码如下：

代码5-1 mydataset

```

01 import tensorflow as tf
02 import sys
03 nets_path = r'slim'                                #加载环境变量
04 if nets_path not in sys.path:
05     sys.path.insert(0,nets_path)
06 else:
07     print('already add slim')
08 from nets.nasnet import nasnet                  #导出nasnet
09 slim = tf.contrib.slim                            #载入TF-slim接口
10 image_size = nasnet.build_nasnet_mobile.default_image_size
11 from preprocessing import preprocessing_factory    #图像处理
12
13 import os
14 def list_images(directory):
15     """
16         获取所有directory中的所有图片和标签
17     """
18
19     #返回path指定的文件夹所包含的文件或文件夹的名字列表
20     labels = os.listdir(directory)
21     #对标签进行排序，以便训练和验证按照相同的顺序进行

```

```

22     labels.sort() #对文件夹中的所有子文件夹进行排序，根据字母顺序
23     #创建文件标签列表
24     files_and_labels = []
25     for label in labels:
26         for f in os.listdir(os.path.join(directory, label)):
27             #将字符串中所有大写字符转换为小写字符，再判断
28             if 'jpg' in f.lower() or 'png' in f.lower():
29                 #加入列表
30                 files_and_labels.append((os.path.join(directory, label, f),
31 label))
32     #理解为解压缩，把数据路径和标签解压缩出来
33     filenames, labels = zip(*files_and_labels)
34     #转换为列表，分别储存数据路径和对应的标签
35     filenames = list(filenames)
36     labels = list(labels)
37     #列出分类总数，比如两类：['man', 'woman']
38     unique_labels = list(set(labels))
39
40     label_to_int = {}
41     #循环列出数据和数据下标，给每个分类打上标签{'woman': 2, 'man': 1, none: 0}
42     for i, label in enumerate(sorted(unique_labels)):
43         label_to_int[label] = i+1
44     print(label, label_to_int[label])
45     #把每个标签化为0、1这种形式
46     labels = [label_to_int[l] for l in labels]
47     print(labels[:6], labels[-6:])
48     return filenames, labels #返回储存数据路径和对应转换后的标签
49 num_workers = 2 #定义并行处理数据的线程数量
50
51 #图像批量预处理
52 image_preprocessing_fn =
53     preprocessing_factory.get_preprocessing('nasnet_mobile',
54     is_training=True)
55 image_eval_preprocessing_fn =
56     preprocessing_factory.get_preprocessing('nasnet_mobile',
57     is_training=False)
58
59 def _parse_function(filename, label): #定义图像解码函数
60     image_string = tf.read_file(filename)
61     image = tf.image.decode_jpeg(image_string, channels=3)
62     return image, label
63
64 def training_preprocess(image, label): #定义函数，调整图像的大小
65     image = image_preprocessing_fn(image, image_size, image_size)
66     return image, label

```

```

64 def val_preprocess(image, label):      # 定义评估图像的预处理函数
65     image = image_eval_preprocessing_fn(image, image_size, image_size)
66     return image, label
67
68 # 创建带批次的数据集
69 def creat_batched_dataset(filenames, labels, batch_size, isTrain = True):
70
71     dataset = tf.data.Dataset.from_tensor_slices((filenames, labels))
72
73     dataset = dataset.map(_parse_function, num_parallel_calls=num_workers)
    # 对图像进行解码
74
75     if isTrain == True:
76         dataset = dataset.shuffle(buffer_size=len(filenames))
    # 打乱数据顺序
77
78     dataset = dataset.map(training_preprocess,
    num_parallel_calls=num_workers) # 调整图像大小
79
80     else:
81         dataset = dataset.map(val_preprocess, num_parallel_calls=num_workers)
    # 调整图像大小
82
83     return dataset.batch(batch_size)           # 返回批次数据
84
85 # 根据目录返回数据集
86 def creat_dataset_fromdir(directory, batch_size, isTrain = True):
87     filenames, labels = list_images(directory)
88     num_classes = len(set(labels))
89     dataset = creat_batched_dataset(filenames, labels, batch_size, isTrain)
90
91     return dataset, num_classes

```

代码第 11 行导入了 `preprocessing_factory` 函数。该函数是 `slim` 模块中封装好的工厂函数，用于生成模型的预处理函数。用该函数对样本进行操作（见代码第 60、61 行），可以提升开发效率，并能够减小出错的可能性。

工厂函数的知识点，属于 Python 基础知识，这里不再详述。有兴趣的读者可以参考《Python 带我起飞——入门、进阶、商业实战》一书的 6.10 节。



提示：

这里用了一个技巧——仿照原 `NASNet_A_Mobile` 模型的分类方法，在对分类标签排序时，将标签为 0 的分类空出来，男人的分类是 1，女人的分类是 2。

代码第 42 行用到的变量 `unique_labels` 是从集合对象转化而来的。在使用时，需要对变量 `unique_labels` 固定顺序，所以用 `sorted` 函数进行变换。如果不对变量 `unique_labels` 固定顺序，在下次启动时，有可能出现标签序号与名称对应不上的现象。在多次中断多次训练的场景中，标签序号与名称对应不上的现象会使模型的准确率飘忽不定。这种问题很难排查。

5.2.3 代码实现：定义微调模型的类 MyNASNetModel

在微调模型的实现中，统一通过定义类 MyNASNetModel 来实现。在类 MyNASNetModel 中，大致可分为两大动作：初始化设置、构建模型。

- 初始化设置：定义构建模型时的必要参数。
- 构建模型：针对训练、测试、应用三种情况，分别构建不同的模型。在训练过程中，还需要加载预训练模型及微调模型。

定义类 MyNASNetModel，并对模型的设置进行初始化。具体如下：

代码 5-2 model

```

01 import sys
02 nets_path = r'slim' # 加载环境变量
03 if nets_path not in sys.path:
04     sys.path.insert(0,nets_path)
05 else:
06     print('already add slim')
07
08 import tensorflow as tf
09 from nets.nasnet import nasnet # 导出 nasnet
10 slim = tf.contrib.slim
11
12 import os
13 mydataset = __import__("5-1_mydataset")
14 creat_dataset_fromdir = mydataset.creat_dataset_fromdir
15
16 class MyNASNetModel(object):
17     """微调模型类 MyNASNetModel
18     """
19     def __init__(self, model_path=''):
20         self.model_path = model_path # 原始模型的路径

```

代码第 20 行是初始化 MyNASNetModel 类的操作。变量 model_path 指的是“要加载的原始预训练模型”。该操作只有在训练模式下才有意义。在测试和应用模式下，该变量可以为 None（空值）。

5.2.4 代码实现：构建 MyNASNetModel 类中的基本模型

在构建模型的过程中，无论是训练、测试还是应用，都需要载入最基本的 NASNet_A_Mobile 模型。这里通过定义 MyNASNetModel 类的 MyNASNet 方法来实现。具体的实现方式与 3.3 节的实现方式基本一致。

不同的是：3.3 节构建的是 PNASNet 模型结构，本节构建的是 NASNet_A_Mobile 模型结构。

代码 5-2 model（续）

```
21     def MyNASNet(self,images,is_training):
```

```

22     arg_scope = nasnet.nasnet_mobile_arg_scope() #获得模型命名空间
23     with slim.arg_scope(arg_scope):
24         #构建NASNet Mobile 模型
25         logits, end_points =
26     nasnet.build_nasnet_mobile(images, num_classes = self.num_classes+1,
27     is_training=is_training)
28
29     global_step = tf.train.get_or_create_global_step()#定义记录步数的张量
30
31     return logits,end_points,global_step #返回有用的张量

```

代码第 25 行，在调用 `nasnet.build_nasnet_mobile` 方法时，向 `num_classes` 参数里传的值是“分类的个数 `self.num_classes` 加 1”。其中：

- 分类的个数 `self.num_classes` 的值是 2，表示男人和女人两类。该值是在 5.2.7 小节的 `build_model` 方法中被赋值的。
- 1 表示是一个空（None）类，即模型预测不出男还是女的情况。

5.2.5 代码实现：实现 MyNASNetModel 类中的微调操作

微调操作是针对训练场景的。它通过定义 `MyNASNetModel` 类中的 `FineTuneNASNet` 方法来实现。微调操作主要是对预训练模型的权重参数进行选择性恢复。

预训练模型 `NASNet_A_Mobile` 是在 `ImgNet` 数据集上训练的，有 1000 个分类。而本实例中识别男女的任务只有两个分类。所以，最后两个输出层的超参不应该被恢复（由于分类不同，导致超参的个数不同）。在实际使用时，最后两层的参数需要对其初始化并单独训练。

代码 5-2 model（续）

```

30     def FineTuneNASNet(self,is_training):    #实现微调模型的网络操作
31         model_path = self.model_path
32
33         exclude = ['final_layer','aux_7'] #恢复超参，除 exclude 外的超参全部恢复
34         variables_to_restore =
35             slim.get_variables_to_restore(exclude=exclude)
36         if is_training == True:
37             init_fn = slim.assign_from_checkpoint_fn(model_path,
38             variables_to_restore)
39         else:
40             init_fn = None
41
42         tuning_variables = [] #将没有恢复的超参数收集起来，用于微调训练过程
43         for v in exclude:
44             tuning_variables += slim.get_variables(v)
45
46         return init_fn, tuning_variables

```

代码中，首先用 `exclude` 列表将不需要恢复的网络节点收集起来（见代码第 33 行）。

接着，将预训练模型中的超参值赋给剩下的节点，完成了预训练模型的载入（见代码第 36 行）。

最后，用 `tuning_variables` 列表将不需要恢复的网络节点权重收集起来（见代码第 40 行），用于微调训练过程。



提示：

这里介绍一个技巧——如何获得 `exclude` 中的元素（见代码第 33 行）。具体方法是：通过额外执行代码 `tf.global_variables()`，将张量图中的节点打印出来；从里面找到最后两层的节点，并将该节点的名称填入代码中。

另外，在找到节点后，还可以用 `slim.get_variables` 函数来检查该名称的节点是否正确。

例如：可以将 `slim.get_variables('final_layer')` 的返回值打印出来，观察张量图中是否有 `final_layer` 节点。

5.2.6 代码实现：实现与训练相关的其他方法

在 `MyNASNetModel` 类中，还需要定义与训练操作相关的其他方法，具体如下。

- `build_acc_base` 方法：用于构建评估模型的相关节点。
- `load_cpk` 方法：用于载入及生成模型的检查点文件。
- `build_model_train` 方法：用于构建训练模型中的损失函数及优化器等操作节点。

具体代码如下：

代码 5-2 model（续）

```

45     def build_acc_base(self, labels):#定义评估函数
46         #返回张量中最大值的索引
47         self.prediction = tf.cast(tf.argmax(self.logits, 1), tf.int32)
48         #计算 prediction、labels 是否相同
49         self.correct_prediction = tf.equal(self.prediction, labels)
50         #计算平均值
51         self.accuracy =
52             tf.reduce_mean(tf.cast(self.correct_prediction), tf.float32)
53             #将正确率最高的 5 个值取出来，计算平均值
54             self.accuracy_top_5 =
55                 tf.reduce_mean(tf.cast(tf.nn.in_top_k(predictions=self.logits,
56                 targets=labels, k=5), tf.float32))
56
57     def load_cpk(self, global_step, sess, begin = 0, saver= None, save_path =
58     None):
59         if begin == 0:
60             save_path=r'./train_nasnet' #定义检查点文件的路径
61             if not os.path.exists(save_path):
62                 print("there is not a model path:", save_path)
63             saver = tf.train.Saver(max_to_keep=1) #生成 saver

```

```

61     return saver, save_path
62 else:
63     kpt = tf.train.latest_checkpoint(save_path) #查找最新的检查点文件
64     print("load model:", kpt)
65     startepo= 0 #计步
66     if kpt!=None:
67         saver.restore(sess, kpt) #还原模型
68         ind = kpt.find("-")
69         startepo = int(kpt[ind+1:])
70         print("global_step=", global_step.eval(), startepo)
71     return startepo
72
73 def build_model_train(self, images,
74     labels, learning_rate1, learning_rate2, is_training):
75     self.logits, self.end_points,
76     self.global_step= self.MyNASNet(images, is_training=is_training)
77     self.step_init = self.global_step.initializer
78
79     self.init_fn, self.tuning_variables = self.FineTuneNASNet(
80         is_training=is_training)
81     #定义损失函数
82     tf.losses.sparse_softmax_cross_entropy(labels=labels,
83         logits=self.logits)
84     loss = tf.losses.get_total_loss()
85     #定义微调训练过程的退化学习率
86     learning_rate1=tf.train.exponential_decay(
87         learning_rate=learning_rate1, global_step=self.global_step,
88         decay_steps=100, decay_rate=0.5)
89     #定义联调训练过程的退化学习率
90     learning_rate2=tf.train.exponential_decay(
91         learning_rate=learning_rate2, global_step=self.global_step,
92         decay_steps=100, decay_rate=0.2)
93     last_optimizer = tf.train.AdamOptimizer(learning_rate1) #优化器
94     full_optimizer = tf.train.AdamOptimizer(learning_rate2)
95     update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
96     with tf.control_dependencies(update_ops): #更新批量归一化中的参数
97         #定义模型优化器
98         self.last_train_op = last_optimizer.minimize(loss,
99             self.global_step, var_list=self.tuning_variables)
100        self.full_train_op = full_optimizer.minimize(loss,
101            self.global_step)
102
103        self.build_acc_base(labels) #定义评估模型的相关指标
104        #写入日志, 支持tensorBoard操作
105        tf.summary.scalar('accuracy', self.accuracy)
106        tf.summary.scalar('accuracy_top_5', self.accuracy_top_5)
107

```

```

106     #将收集的所有默认图表并合并
107     self.merged = tf.summary.merge_all()
108     #写入日志文件
109     self.train_writer = tf.summary.FileWriter('./log_dir/train')
110     self.eval_writer = tf.summary.FileWriter('./log_dir/eval')
111     #定义要保持到检查点文件中的变量
112     self.saver, self.save_path = self.load_ckp(self.global_step, None)

```

代码第 82 行，用 `tf.losses.sparse_softmax_cross_entropy` 函数计算 loss 值，函数会将 loss 值添加到内部集合 `ops.GraphKeys.LOSSES` 中。

代码第 84 行，用 `tf.losses.get_total_loss` 函数从 `ops.GraphKeys.LOSSES` 集合中取出所有的 loss 值。

在代码第 96 行，在反向优化时，用 `tf.control_dependencies` 函数对批量归一化操作中的均值与方差进行更新。函数 `tf.control_dependencies` 的作用是，将依赖运行的功能添加到 `last_train_op` 与 `full_train_op` 的操作上。即：在执行代码 `last_train_op` 与 `full_train_op`（见代码第 98、99 行）之前，需要先执行 `tf.GraphKeys.UPDATE_OPS` 中的 OP。

`tf.GraphKeys.UPDATE_OPS` 中的 OP 就是更新 BN 中的移动均值 (μ) 和移动方差 (σ) 的实际操作。在调用 TF-slim 接口中的 BN 函数时，默认不会直接更新移动均值 (μ) 和移动方差 (σ)。而是将其封装为一个 OP（静态图中的操作符）放到 `tf.GraphKeys.UPDATE_OPS` 中。关于这部分知识，在 10.3.5 小节还会涉及。

5.2.7 代码实现：构建模型，用于训练、测试、使用

在 `MyNASNetModel` 类中，定义了 `build_model` 方法用于构建模型。在 `build_model` 方法中，用参数 `mode` 来指定模型的具体使用场景。具体代码如下：

代码 5-2 model (续)

```

113     def build_model(self, mode='train', testdata_dir='./data/val',
114                     traindata_dir='./data/train',
115                     batch_size=32, learning_rate1=0.001, learning_rate2=0.001):
116         if mode == 'train':
117             tf.reset_default_graph()
118             #创建训练数据和测试数据的 Dataset 数据集
119             dataset, self.num_classes =
120                 creat_dataset_fromdir(traindata_dir, batch_size)
121             testdataset, _ =
122                 creat_dataset_fromdir(testdata_dir, batch_size, isTrain = False)
123             #创建一个可初始化的迭代器
124             iterator = tf.data.Iterator.from_structure(dataset.output_types,
125                 dataset.output_shapes)
126             #读取数据
127             images, labels = iterator.get_next()

```

```

125 import tensorflow as tf
126 learning_rate1 = 0.001
127 learning_rate2 = 0.0001
128 num_epochs = 70
129 self.build_model_train(images,
130                         labels, learning_rate1, learning_rate2, is_training=True)
131 self.global_init = tf.global_variables_initializer() # 定义全局初始化OP
132 tf.get_default_graph().finalize() # 将后续的图设为只读
133 elif mode == 'test':
134     tf.reset_default_graph()
135     # 创建测试数据的 Dataset 数据集
136     testdataset, self.num_classes =
137         creat_dataset_fromdir(testdata_dir, batch_size, isTrain = False)
138     # 创建一个可初始化的迭代器
139     iterator =
140         tf.data.Iterator.from_structure(testdataset.output_types,
141                                         testdataset.output_shapes)
142     # 读取数据
143     self.images, labels = iterator.get_next()
144     self.test_init_op = iterator.make_initializer(testdataset)
145     self.logits, self.end_points, self.global_step =
146         self.MyNASNet(self.images, is_training=False)
147     self.saver, self.save_path = self.load_ckp(self.global_step, None)
148     # 定义用于操作检查点文件的相关变量
149     # 评估指标
150     self.build_acc_base(labels)
151     tf.get_default_graph().finalize() # 将后续的图设为只读
152     elif mode == 'eval':
153         tf.reset_default_graph()
154         # 创建测试数据的 Dataset 数据集
155         testdataset, self.num_classes =
156             creat_dataset_fromdir(testdata_dir, batch_size, isTrain = False)
157         # 创建一个可初始化的迭代器
158         iterator =
159             tf.data.Iterator.from_structure(testdataset.output_types,
160                                             testdataset.output_shapes)
161             # 读取数据
162             self.images, labels = iterator.get_next()
163             self.logits, self.end_points, self.global_step =
164                 self.MyNASNet(self.images, is_training=False)

```

```

160         self.saver, self.save_path = self.load_ckp(self.global_step, None)
          # 定义用于操作检查点文件的相关变量
161     tf.get_default_graph().finalize() # 将后续的图设为只读

```

代码第 115 行，对 mode 进行了判断，获得当前的使用场景。并根据不同的使用场景实现不同的代码分支。针对训练、测试、使用这三个场景，构建的步骤几乎一样，具体如下：

- (1) 清空张量图（见代码第 116、133、150 行）。
- (2) 生成数据集（见代码第 118、136、152 行）。
- (3) 定义网络结构（见代码第 129、144、159 行）。

代码第 147 行用 build_acc_base 方法生成评估节点，用于评估模型。



提示：

在每个操作分支的最后部分都加了代码“tf.get_default_graph().finalize()”（见代码第 131、148、161 行），这是一个很好的习惯。

该代码的功能是把图锁定，之后如想添加任何新的操作则都会产生错误。这么做的意图是：防止在后面训练或是测试过程中，由于开发人员疏忽在图中添加额外的图操作。

如果在循环内部额外定义了其他张量，则会使整体性能大大下降，然而这种错误又很难发现。所以，利用锁定图的方法可以避免这种情况的发生。

5.2.8 代码实现：通过二次迭代来训练微调模型

训练微调模型的操作是在代码文件“5-3 train.py”中单独实现的。与正常的训练方式不同，这里用两次迭代的方式。

- 第 1 次迭代：微调模型，固定预训练模型载入的权重，只训练最后两层。
- 第 2 次迭代：联调模型，用更小的学习率训练全部节点。

先将 MyNASNetModel 类实例化，再用其 build_model 方法构建模型，然后用会话(session)开始训练。具体代码如下：

代码 5-3 train

```

import tensorflow as tf
model = __import__("5-2 model")
MyNASNetModel = model.MyNASNetModel

batch_size = 32
train_dir = 'data/train'
val_dir = 'data/val'

learning_rate1 = 1e-1 # 定义两次迭代的学习率
learning_rate2 = 1e-3
# 初始化模型
mymode = MyNASNetModel(r'nasnet-a_mobile_04_10_2017\model.ckpt')

```

```

18 mymode.build_model('train',val_dir,train_dir,batch_size,learning_rate1 ,
19 learning_rate2 )                                #载入模型
20 #微调的迭代次数 epochs1 = 20
21 num_epochs1 = 20                                 #微调的迭代次数 epochs
22 num_epochs2 = 200                                #联调的迭代次数
23 #载入模型
24 with tf.Session() as sess:
25     sess.run(mymode.global_init)                 #初始全局节点
26     step = 0                                     ("step", "global_step")
27     mymode.load_cpk(mymode.global_step,sess,1,mymode.saver,mymode.save_path)
28     #载入模型
29     print(step)                                #微调
30     if step == 0:
31         mymode.init_fn(sess)                   #载入预训练模型的权重
32     #输出进度
33     for epoch in range(num_epochs1):
34         print('Starting1 epoch %d / %d' % (epoch + 1, num_epochs1))
35         #用训练集初始化迭代器
36         sess.run(mymode.train_init_op)          #数据集从头开始
37         while True:
38             try:
39                 step += 1
40                 #预测, 合并图, 训练
41                 acc,accuracy_top_5, summary, _ = sess.run([mymode.accuracy,
42 mymode.accuracy_top_5,mymode.merged,mymode.last_train_op])
43             except tf.errors.OutOfRangeError:        #数据集指针在最后
44                 print("train1:",epoch, " ok")
45                 mymode.saver.save(sess, mymode.save_path+ "/mynasnet.cpkt",
46 global_step=mymode.global_step.eval())
47                 break
48
49         sess.run(mymode.step_init)               #微调结束, 计数器从0开始
50
51 #整体训练
52 for epoch in range(num_epochs2):
53     print('Starting2 epoch %d / %d' % (epoch + 1, num_epochs2))
54     sess.run(mymode.train_init_op)
55     while True:
56         try:

```

```

160    print(step)
161    step += 1
162    # 预测, 合并图, 训练
163    acc, summary, _ = sess.run([mymode.accuracy, mymode.merged,
164                               mymode.full_train_op])
165
166    # 代码对训练步数进行了判断, 如果当前的使用目录, 开始时不使用模型, 实现
167    # 不同的代码分支
168    mymode.train_writer.add_summary(summary, step) # 写入日志文件
169
170    if step % 100 == 0:
171        print(f'step: {step} train2 accuracy: {acc}')
172    except tf.errors.OutOfRangeError:
173        print("train2:", epoch, " ok")
174        mymode.saver.save(sess, mymode.save_path + "/mynasnet.cpkt",
175                          global_step=mymode.global_step.eval())
176        break

```

将以上代码运行后，会在本地“train_nasnet”文件夹中生成训练好的模型文件。

5.2.9 代码实现：测试模型

测试模型的操作是在代码文件“5-4 test.py”中单独实现的。下面用测试数据集评估现有模型，并且将单张图片放到模型里进行预测。

1. 定义测试模型所需要的功能函数

首先，定义函数 check_accuracy，以实现准确率的计算。

接着，定义函数 check_sex，以实现男女性别的识别。

具体代码如下：

代码 5-4 test

```

01 import tensorflow as tf
02 model = __import__("5-2 model")
03 MyNASNetModel = model.MyNASNetModel
04
05 import sys
06 nets_path = r'slim' # 加载环境变量
07 if nets_path not in sys.path:
08     sys.path.insert(0, nets_path)
09 else:
10     print('already add slim')
11
12 from nets.nasnet import nasnet # 载入 nasnet 模型
13 slim = tf.contrib.slim # 载入 TF-slim 接口
14 image_size = nasnet.build_nasnet_mobile.default_image_size # 获得输入尺寸
15 224
16 import numpy as np
17 from PIL import Image

```

```

18
19 batch_size = 32
20 test_dir = 'data/val'
21
22 def check_accuracy(sess):
23     """
24     测试模型准确率
25     """
26     sess.run(mymode.test_init_op)          # 初始化测试数据集
27     num_correct, num_samples = 0, 0        # 定义正确个数和总个数
28     i = 0
29     while True:
30         i+=1
31         print('i',i)
32         try:
33             #计算 correct_prediction
34             correct_pred,accuracy,logits =
35             sess.run([mymode.correct_prediction,mymode.accuracy,mymode.logits])
36             #累加 correct_pred
37             num_correct += correct_pred.sum()
38             num_samples += correct_pred.shape[0]
39             print("accuracy",accuracy,logits)
40
41         except tf.errors.OutOfRangeError:      #捕获异常，数据用完后自动跳出
42             print('over')
43             break
44
45     acc = float(num_correct) / num_samples    #计算并返回准确率
46     return acc
47
48 #定义函数用于识别男女
49 def check_sex(imgdir,sess):
50     img = Image.open(image_dir)            #读入图片
51     if "RGB"!=img.mode :                  #检查图片格式
52         img = img.convert("RGB")
53
54     img = np.asarray(img.resize((image_size,image_size))),   #图像预处理
55
56     dtype=np.float32).reshape(1,image_size,image_size,3)
57     img = 2 *( img / 255.0)-1.0
58     #将图片传入 nasnet 模型的输入中，得出预测结果
59     prediction = sess.run(mymode.logits, {mymode.images: img})
60     print(prediction)
61
62     pre = prediction.argmax()           #返回张量中值最大的索引
63     print(pre)

```

```

63
64     if pre == 1: img_id = 'man'
65     elif pre == 2: img_id = 'woman'
66     else: img_id = 'None'
67     plt.imshow( np.asarray((img[0]+1)*255/2,np.uint8) ) #将张量转为图像
68     plt.show()
69     print(img_id,"--",image_dir) #返回类别
70
71     return pre

```

2. 建立会话，进行测试

首先，建立会话（session），对模型进行测试。

接着，将两张图片输入模型，进行男女的判断。

具体代码如下：

代码 5-4 test (续)

```

71 mymode = MyNASNetModel() #初始化模型
72 mymode.build_model('test',test_dir) #载入模型
73
74 with tf.Session() as sess: #载入模型
75     mymode.load_cpk(mymode.global_step,sess,1,mymode.saver,
76     mymode.save_path)
77
78     #测试模型的准确性
79     val_acc = check_accuracy(sess)
80     print('Val accuracy: %f\n' % val_acc)
81
82     #单张图片测试
83     image_dir = 'tt2t.jpg' #选取测试图片
84     check_sex(image_dir,sess)
85
86     image_dir = test_dir + '\\woman' + '\\000001.jpg' #选取测试图片
87     check_sex(image_dir,sess)
88
89     image_dir = test_dir + '\\man' + '\\000003.jpg' #选取测试图片
90     check_sex(image_dir,sess)

```

该程序使用的模型文件，只迭代训练了 100 次（如果要提高效果，则可以再多训练几次）。

代码运行后，输出以下结果。

(1) 显示测试集的输出结果：

```
i 1
accuracy 0.90625 [[-3.813714    1.4075054   1.1485975 ]
 [-7.3948846   6.220533   -1.4093535 ]
 [-1.9391974   3.048838   0.21784738]
 [-3.873174    4.530942   0.43135062]
.....
```

```

[-3.8561587  2.7012844 -0.3634925 ]
[-4.4860134  4.7661724 -0.67080706]
[-2.9615571  2.8164086  0.71033645]]
i 2
accuracy 0.90625 [[ -6.6900268 -2.373093   6.6710057 ]
[ -4.1005263  0.74619263  4.980012 ]
[ -5.6469827  0.39027584  1.2689826 ]]
.....
[ -5.8080773  0.9121424  3.4134243 ]
[ -4.242001   0.08483959  4.056322 ]
i 3
over
Val accuracy: 0.906250

```

上面显示的是测试集中 man 和 woman 文件夹中的图片的计算结果。最终模型的准确率为 90%。

(2) 显示单张图片的运行结果：

```

[[ -4.8022223  1.9008529  1.9379601]]
2

```

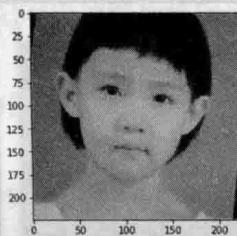


图 5-3 分辨男女测试图片 (a)

```

woman -- tt2t.jpg
[[-6.181205 -2.9042015  6.1356106]]
2

```

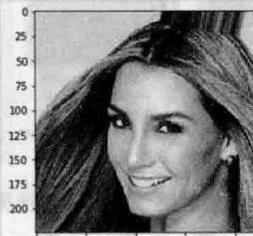


图 5-3 分辨男女测试图片 (b)

```

woman -- data/val/woman\000001.jpg
[[-4.896065  1.7791721  1.3118265]]
1

```

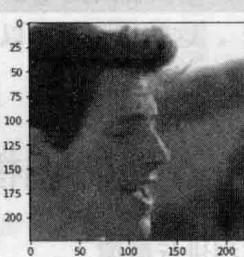


图 5-3 分辨男女测试图片 (c)

```
man -- data/val/man/000003.jpg
```

上面显示了 3 张图片，分别为：自选图片、测试数据集中的女人图片、测试数据集中的男人图片，每张图片下面显示了模型识别的结果。可以看到，结果与图片内容一致。

5.3 扩展：通过摄像头实时分辨男女

下面在 5.2.9 小节的例子基础上加入摄像头的采集功能，这样便可以实现实时分辨男女。



提示：

由于本书重点内容聚焦在深度学习部分，所以摄像头采集部分不再介绍，有兴趣的读者可以参考《Python 带我起飞——入门、进阶、商业实战》一书的第 14 章。那里有完整的人脸识别实例及配套代码。

将摄像头采集的图片输入本实例的模型中即可实现。最终呈现的效果如图 5-4 所示。



图 5-4 通过摄像头实时分辨男女

5.4 TF-slim 接口中的更多成熟模型

在 3.1 节下载 PNASNet 模型部分，可以看到图 3-2 中有很多其他模型（VGG、ResNet、Inception v4、Inception-ResNet-v2 等）。这些模型都可以被下载，并使用本节实例中的方法进行二次训练。

5.5 实例 13：用 TF-Hub 库微调模型以评估人物的年龄

本节将使用 TF-Hub 库对预训练模型进行微调。

实例描述

有一组照片，每个文件夹的名称为具体的年龄，里面放的是该年纪的人物图片。微调 TF-Hub 库，让模型学习这些样本，找到其中的规律，可以根据具体人物的图片来评估人物的年龄。

本实例与 5.2 节的实例一样，都是让 AI 模型具有人眼的评估能力。即便是通过人眼来观察他人的外表，也不能准确判断出被观察人的性别和年纪。所以在应用中，模型的准确度应该与用人眼的估计值来比对，并不能与被测目标的真实值来比对。

5.5.1 准备样本

本实例所用的样本来自于 IMDB-WIKI 数据集。IMDB-WIKI 数据集中包含与年龄匹配对应的人物图片。该数据集的介绍及下载地址可以参考以下链接：

<https://data.vision.ee.ethz.ch/cvl/rrothe/imdb-wiki/>

因为该数据集相对粗糙（有些年纪对应的图片特别少），所以需要在该数据集的基础上做一些简单的调整：

- 补充了一些与年龄匹配的人物图片。
- 删掉了若干不合格的样本。

整理后的图片一共有 105500 张，如图 5-5 所示。

图 5-5 显示的是数据集中的文件。文件夹的名称代表年龄，文件夹里面放的是该年纪的人物图片。

读者可以直接使用本书配套的数据集，将该数据集（IMBD-WIKI 文件夹）放到当前代码的本地同级文件夹下即可使用。

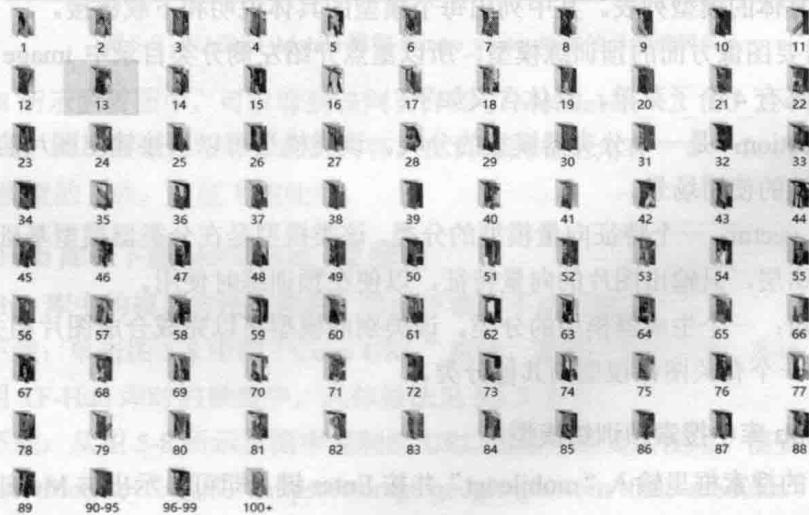


图 5-5 数据集中的文件

5.5.2 下载 TF-Hub 库中的模型

安装 TF-Hub 库的具体方法见 5.1.3 小节。在安装完成之后，可以按照以下步骤进行操作。

1. 找到 TF-Hub 库中的模型下载链接

在 GitHub 网站中找到 TF-Hub 库中所提供的模型及下载地址，具体网址如下（国内可能访问不了，请读者自行想办法）：

<https://tfhub.dev/>

打开该网页后，可以看到在列表中有很多模型及下载链接，如图 5-6 所示。

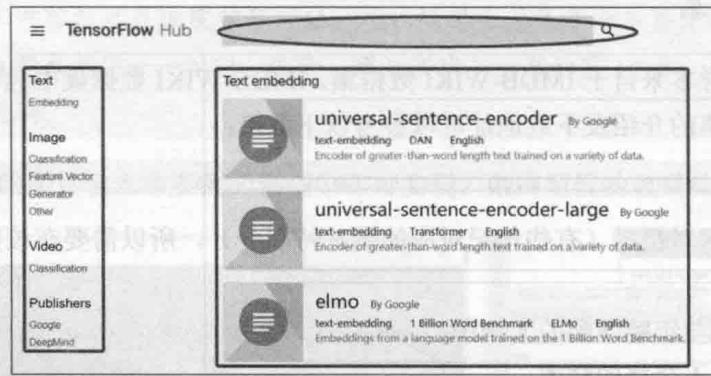


图 5-6 预训练模型列表

在图 5-6 可以分为 3 部分，具体如下：

- 最顶端是搜索框。可以通过该搜索框搜索想要下载的预训练模型。
- 左侧是模型的分类目录。将 TF-Hub 库中的预训练模型按照文本、图像、视频、发布者进行分类。
- 右侧是具体的模型列表。其中列出每个模型的具体说明和下载链接。

因为本例需要图像方面的预训练模型，所以重点介绍左侧分类目录中 image 下的内容。在 image 分类下方还有 4 个子菜单，具体含义如下：

- Classification:** 是一个分类器模型的分类。该类模型可以直接输出图片的预测结果。用于端到端的使用场景。
- Feature_vector:** 一个特征向量模型的分类。该类模型是在分类器模型基础上去掉了最后两个网络层，只输出图片的向量特征，以便在预训练时使用。
- Generator:** 一个生成器模型的分类。该类别的模型可以完成合成图片相关的任务。
- Other:** 一个有关图像模型的其他分类。

2. 在 TF-Hub 库中搜索预训练模型

在图 5-6 中的搜索框里输入“mobilenet”并按 Enter 键，即可显示出与 MobileNet 相关的模型，如图 5-7 所示。



图 5-7 搜索 MobileNet 预训练模型

在图 5-7 右侧的列表部分，可以找到 MobileNet 模型。以 MobileNet_v2_100_224 模型为例（图 5-7 右侧列表中的最下方 2 行），该模型有两个版本：classification 与 feature_vector。

单击图 5-7 右侧列表中的最后下面一行，进入 MobileNet_v2_100_224 模型 classification 版本的详细说明页面，如图 5-8 所示。

图 5-8 NASNet_Mobile 模型 feature_vector 版本的详细说明页

在如图 5-8 所示的页面中，可以看到该网页介绍了 MobileNet_v2_100_224 模型的来源、训练、使用、微调，以及历史日志等方面的内容。在页面的右上角有一个“Copy URL”按钮，该按钮可以复制模型的下载，方便下载使用。

3. 在 TF-Hub 库中下载 MobileNet_V2 模型

下载 TF-Hub 库中的模型方法有两种：自动下载和手动下载。

- 自动下载：单击图 5-8 中的“Copy URL”按钮，复制下载的 URL 地址，并将该地址填入调用 TF-Hub 库时的参数中。具体做法见 5.5.3 小节。
- 手动下载：从图 5-8 所示页面中复制的 URL 地址不能直接使用，需要将其前半部分的“<https://tfhub.dev>”换成“<https://storage.googleapis.com/tfhub-modules>”，并在 URL 后加上“.tar.gz”。

以 MobileNet_v2_100_224（简称 MobileNet_V2）模型的 classification 版本为例，手动下载的步骤如下。

(1) 单击 5-8 中的“Copy URL”按钮，所得到的 URL 地址如下：

```
https://tfhub.dev/google/imagenet/mobilenet_v2_100_224/feature_vector/2
```

(2) 将其改成正常下载的地址。具体如下：

```
https://storage.googleapis.com/tfhub-modules/google/imagenet/mobilenet_v2_100_224/feature_vector/2.tar.gz
```

(3) 用下载工具按照(2)中的地址进行下载。

5.5.3 代码实现：测试 TF-Hub 库中的 MobileNet_V2 模型

为了验证 TF-Hub 库中的模型效果，本小节将使用与第 3 章类似的代码：将 3 张图片输入 MobileNet_V2 模型的 classification 版本中，观察其输出结果。

编写代码载入 MobileNet_V2 模型，具体代码如下：

代码 5-5 测试 TF-Hub 库中的 NASNet_Mobile 模型

```

01 from PIL import Image
02 from matplotlib import pyplot as plt
03 import numpy as np
04 import tensorflow as tf
05 import tensorflow_hub as hub
06
07 with open('中文标签.csv', 'r+') as f:                      # 打开文件
08     labels = list(map(lambda x:x.replace(',', ' '), list(f)))  # 读取文件
09     print(len(labels), type(labels), labels[:5])             # 显示输出中文标签
10
11 sample_images = ['hy.jpg', 'ps.jpg', '72.jpg']           # 定义待测试图片路径
12
13 # 加载分类模型
14 module_spec = hub.load_module_spec("https://tfhub.dev/google/imagenet/mobilenet_v2_100_224/classification/2")
15 # 获得模型的输入图片尺寸
16 height, width = hub.get_expected_image_size(module_spec)
17
18 input_imgs = tf.placeholder(tf.float32, [None, height, width, 3]) # 定义占位符
19 images = 2 * (input_imgs / 255.0) - 1.0                      # 归一化图片
20
21 module = hub.Module(module_spec)                                # 将模型载入张量图
22
23 logits = module(images) # 获得输出张量，其形状为[batch_size, num_classes]
24
25 y = tf.argmax(logits, axis = 1)                                  # 获得结果的输出节点
26 with tf.Session() as sess:
27     sess.run(tf.global_variables_initializer())
28     sess.run(tf.tables_initializer())

```

```

29
30     def preimg(img):                                # 定义图片预处理函数
31         return np.asarray(img.resize((height, width)),
32                             dtype=np.float32).reshape(height, width, 3)
33
34     # 获得原始图片与预处理图片
35     batchImg = [ preimg( Image.open(imgfilename) ) for imgfilename in
36     sample_images ]
37
38     # 将样本输入模型
39     yv, img_norm = sess.run([y, images], feed_dict={input_imgs: batchImg})
40     print(yv, np.shape(yv))                           # 显示输出结果
41
42     def showresult(yy, img_norm, img_org):           # 定义显示图片函数
43         plt.figure()
44         p1 = plt.subplot(121)
45         p2 = plt.subplot(122)
46         p1.imshow(img_org)                          # 显示图片
47         p1.axis('off')
48         p1.set_title("organization image")
49
50         p2.imshow((img_norm * 255).astype(np.uint8))   # 显示图片
51         p2.axis('off')
52         p2.set_title("input image")
53         plt.show()
54
55         print(yy, labels[yy])
56
57     for yy, img1, img2 in zip(yv, batchImg, orgImg):    # 显示每条结果及图片
58         showresult(yy, img1, img2)

```

在代码第 14 行，用 TF-Hub 库中的 `load_module_spec` 函数加载 MobileNet_V2 模型。该步骤是通过将 TF-Hub 库中的模型链接（Module URL="https://tfhub.dev/google/imagenet/mobilenet_v2_100_224/classification/2"）传入函数 `load_module_spec` 中来完成的。

在链接里可以找到该模型文件的名字：mobilenet_v2_100_224。TF-Hub 库中的命名都非常规范，从名字上便可了解该模型的相关信息：

- 模型是 MobileNet_V2。
- 神经元节点是 100%（无裁剪）。
- 输入的图片尺寸是 224。

得到模型之后，便将模型文件载入图中（见代码第 21 行），并获得输出张量（见代码第 23 行），然后通过会话（session）完成模型的输出结果。

运行代码后，显示以下结果：

```

1001 <class 'list'> ['背景 known  \n', '丁鲷      \n', '金鱼      \n', '大白鲨      \n',
虎鲨      \n']

```

```

INFO:tensorflow:Downloading TF-Hub Module 'https://tfhub.dev/google/imagenet/
mobilenet_v2_100_224/classification/2'.
.....
INFO:tensorflow:Initialize variable module/MobilenetV2/expanded_conv_9/project/
weights:0 from checkpoint b'C:\\Users\\ljh\\AppData\\Local\\Temp\\tfhub_modules\\
bb6444e8248f8c581b7a320d5ff53061e4506c19\\variables\\variables' with MobilenetV2/
expanded_conv_9/project/weights
[852 490 527] (3,)

```

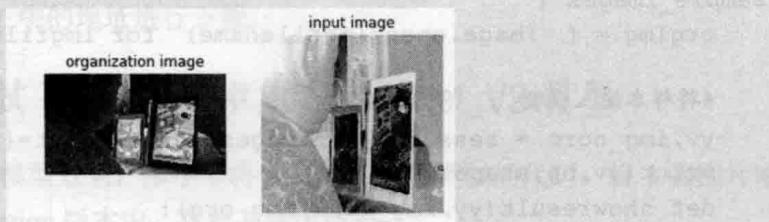


图 5-9 测试 MobileNet_V2 模型结果 (a)

852 电视

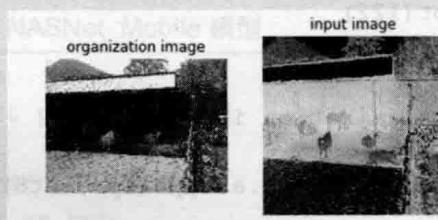


图 5-9 测试 MobileNet_V2 模型结果 (b)

490 围栏

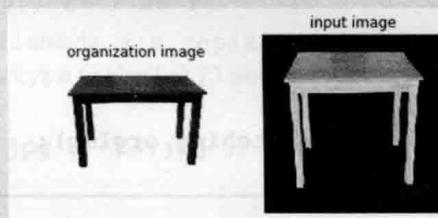


图 5-9 测试 MobileNet_V2 模型结果 (c)

527 书桌

在显示的结果中，可以分为两部分内容：

- 第 1 行是标签内容。
- 从第 2 行开始，所有以“INFO:”开头的信息都是模型加载具体参数时的日志信息。

在每条信息中都能够看到一个相同的路径：“checkpoint b'C:\\Users\\ljh\\AppData\\Local\\Temp\\tfhub_modules\\bb6444e8248f8c581b7a320d5ff53061e4506c19” ，这表示系统将 mobilenet_v2_100_224 模型下载到 C:\\Users\\ljh\\AppData\\Local\\Temp\\tfhub_modules\\bb6444e8248f8c581b7a320d5ff53061e4506c19 目录下。

如果想要让模型缓存到指定的路径下，则需要在系统中设置环境变量 TFHUB_CACHE_DIR。例如，以下语句表示将模型下载到当前目录下的 my_module_cache 文件夹中。

```
TFHUB_CACHE_DIR= ./my_module_cache
```

提示：

如果由于网络原因导致模型无法下载成功，还可以将本书的配套模型资源复制到当前代码同级目录下，并传入当前模型文件的路径。具体操作是，将代码第14行换为以下代码：

```
module_spec = hub.load_module_spec("mobilenet_v2_100_224")
```

在最后一条的INFO信息之后便是模型的预测结果。

提示：

如果感觉输出的INFO内容太多，则可以在代码的最前面加上“tf.logging.set_verbosity(tf.logging.ERROR)”来关闭info信息输出。

5.5.4 用TF-Hub库微调MobileNet_V2模型

在TF-Hub库的GitHub网站上提供了微调模型的代码文件，运行该代码可以直接微调现有模型。该文件的地址如下：

```
https://github.com/tensorflow/hub/raw/master/examples/image\_retraining/retrain.py
```

将代码文件下载后，直接用命令行的方式运行，便可以对模型进行微调。

1. 修改TF-Hub库中的代码BUG

当前代码存在一个隐含的BUG：在某一类的数据样本相对较少的情况下，运行时会产生错误。需要将其修改后才可以正常运行。

在“retrain.py”代码文件中的函数get_random_cached_bottlenecks里添加代码（见代码第477行），当程序在产生错误时，让其再去执行一次随机选取类别的操作（见代码第515~525行）。具体代码如下：

代码retrain(片段)

```
...
477 def get_random_cached_bottlenecks(sess, image_lists, how_many, category,
478                                     bottleneck_dir, image_dir, jpeg_data_tensor,
479                                     decoded_image_tensor, resized_input_tensor,
480                                     bottleneck_tensor, module_name):
481     ...
507     class_count = len(image_lists.keys())
508     bottlenecks = []
509     ground_truths = []
510     filenames = []
511     if how_many >= 0:
512         # Retrieve a random sample of bottlenecks.
513         for unused_i in range(how_many):
514             IsErr = True          #添加检测异常标志
515             while IsErr==True:    #如果出现异常就再运行一次
516                 ...
```

```

517     try:
518         label_index = random.randrange(class_count)
519         label_name = list(image_lists.keys())[label_index]
520         image_index = random.randrange(MAX_NUM_IMAGES_PER_CLASS + 1)
521         image_name = get_image_path(image_lists, label_name,
522                                     image_index,
523                                     image_dir, category)
524         IsErr = False    #没有异常
525     except ZeroDivisionError:
526         continue        #出现异常，再运行一次
...

```

2. 用命令行运行微调程序

将代码文件“retrain.py”与 5.5.1 小节准备的样本数据、5.5.2 小节下载的 MobileNet_V2 模型文件一起放到当前代码的同级目录下。在命令行窗口中输入以下命令：

```
python retrain.py --image_dir ./IMBD-WIKI --tfhub_module mobilenet_v2_100_224_feature_vector
```

也可以输入以下命令，直接从网上下载 MobileNet_V2 模型，并进行微调。

```
python retrain.py --image_dir ./IMBD-WIKI --tfhub_module https://tfhub.dev/google/imagenet/mobilenet_v2_100_224/feature_vector/2
```

程序运行之后，会显示如图 5-10 所示界面。



图 5-10 微调 MobileNet_V2 模型结束

从图 5-10 中可以看到，生成的模型被放在默认路径下（根目录下的 tmp 文件夹里）。来到该路径下（作者本地的路径是“G:\tmp”），可以看到微调模型程序所生成的文件，如图 5-11 所示。

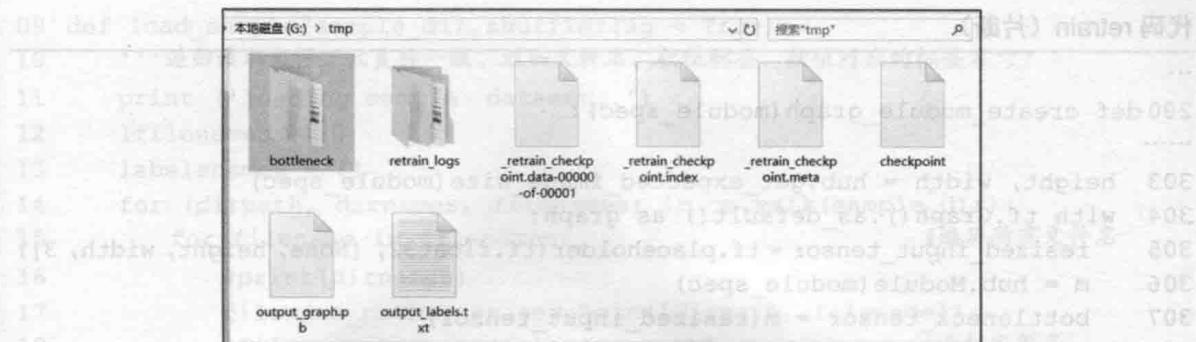


图 5-11 微调 MobileNet_V2 模型后生成的文件

在图 5-11 中可以看到有两个文件夹。

- **bottleneck:** 用预训练模型 MobileNet_V2 将图片转化成的特征值文件。
- **retrain_logs:** 微调模型过程中的日志文件。该文件可以通过 TensorBoard 显示出来（TensorBoard 的使用方法见 13.3.2 小节）。

其他的文件是训练后生成的模型。每个模型文件的具体意义在第 6 章会有介绍。



提示：

本实例只是一个例子，重点在演示 TF-Hub 的使用。因为实例中所使用的数据集质量较低，所以训练效果并不是太理想。读者可以按照本实例的方法使用更优质的数据集训练出更好的模型。

3. 支持更多的命令行操作

代码文件“retrain.py”是一个很强大的训练脚本。在使用时，还可以通过修改参数实现更多的配置。

本实例只演示了部分参数的使用，其他的参数都用默认值，例如：迭代训练 4000 次，学习率为 0.01，批次大小为 100，训练集占比为 80%，测试集与验证集各占比 10% 等。

可以通过以下命令获得该脚本的全部参数说明。

```
python retrain.py -h
```

5.5.5 代码实现：用模型评估人物的年龄

用代码文件“retrain.py”微调后的模型是以扩展名为“pb”的文件存在的（在图 5-11 中，第 2 行的左数第 1 个）。该模型文件属于冻结图文件。冻结图的知识在第 13 章会详细讲解。

将冻结图格式的模型载入内存，便可以评估人物的年纪。

1. 找到模型中的输入、输出节点

冻结图文件中只有模型的具体参数。如果想使用它，则还需要知道与模型文件对应的输入和输出节点。

这两个节点都可以在代码文件“retrain.py”中找到。以输入节点为例，具体代码如下：

代码 retrain (片断)

```

...
290 def create_module_graph(module_spec):
.....
303     height, width = hub.get_expected_image_size(module_spec)
304     with tf.Graph().as_default() as graph:
305         resized_input_tensor = tf.placeholder(tf.float32, [None, height, width, 3])
306         m = hub.Module(module_spec)
307         bottleneck_tensor = m(resized_input_tensor)
308         wants_quantization = any(node.op in FAKE_QUANT_OPS
309                                 for node in graph.as_graph_def().node)
310     return graph, bottleneck_tensor, resized_input_tensor, wants_quantization
...

```

从代码文件“retrain.py”的第 305 行代码可以看到，输入节点的张量是一个占位符——placeholder。

**提示：**

直接使用 `print(placeholder.name)` 和 `print(final_result.name)` 两行代码即可将输入节点和输出节点的名称打印出来。

将输入节点和输出节点的名称记下来，填入代码文件“5-6 用微调后的 mobilenet_v2 模型评估人物的年龄.py”中，便可以实现模型的使用。

更多有关张量的介绍可以参考《深度学习之 TensorFlow——入门、原理与进阶实战》的 4.4.2 小节。

2. 加载模型并评估结果

将本书的配套图片样例文件“22.jpg”和“tt2t.jpg”放到代码的同级目录下，用于测试模型。同时把生成的模型文件夹“tmp”也复制到本地代码的同级目录下。

这部分代码可以分为 3 部分。

- 样本文件加载部分（见代码第 1~34 行）：这部分重用了本书 4.7 节的代码。
- 加载冻结图（见代码第 35~69 行）：读者可以先有一个概念，在第 13 章还有详细讲解。
- 图片结果显示部分（见代码第 70~94 行）：这部分重用了本书 3.4 节中显示部分的代码。

完整的代码如下：

代码 5-6 用模型评估人物的年龄

```

01 from PIL import Image
02 from matplotlib import pyplot as plt
03 import numpy as np
04 import tensorflow as tf
05
06 from sklearn.utils import shuffle
07 import os
08

```

```

09 def load_sample(sample_dir, shuffleflag = True):
10     '''递归读取文件。只支持一级。返回文件名、数值标签、数值对应的标签名'''
11     print ('loading sample dataset..')
12     lfilenames = []
13     labelsnames = []
14     for (dirpath, dirnames, filenames) in os.walk(sample_dir):
15         for filename in filenames: #遍历所有文件名
16             #print(dirnames)
17             filename_path = os.sep.join([dirpath, filename])
18             lfilenames.append(filename_path) #添加文件名
19             labelsnames.append( dirpath.split('\\')[-1] )#添加文件名对应的标签
20
21     lab= list(sorted(set(labelsnames))) #生成标签名称列表
22     labdict=dict( zip( lab ,list(range(len(lab)))) ) #生成字典
23
24     labels = [labdict[i] for i in labelsnames]
25     if shuffleflag == True:
26         return
27     shuffle(np.asarray( lfilenames),np.asarray( labels)),np.asarray(lab)
28     else:
29         return
30     (np.asarray( lfilenames),np.asarray( labels)),np.asarray(lab)
31 #载入标签
32 data_dir = 'IMBD-WIKI\\' #定义文件的路径
33 _,labels = load_sample(data_dir,False) #载入文件的名称与标签
34 print(labels) #输出load_sample返回的标签字符串
35 sample_images = ['22.jpg', 'tt2t.jpg'] #定义待测试图片的路径
36
37 tf.logging.set_verbosity(tf.logging.ERROR)
38 tf.reset_default_graph()
39 #分类模型
40 thissavedir= 'tmp'
41 PATH_TO_CKPT = thissavedir +'/output_graph.pb'
42 od_graph_def = tf.GraphDef()
43 with tf.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
44     serialized_graph = fid.read()
45     od_graph_def.ParseFromString(serialized_graph)
46     tf.import_graph_def(od_graph_def, name='')
47
48 fenlei_graph = tf.get_default_graph()
49
50 height,width = 224,224
51
52 with tf.Session(graph=fenlei_graph) as sess:
53     result = fenlei_graph.get_tensor_by_name('final_result:0')

```

```

54     input_imgs = fenlei_graph.get_tensor_by_name('Placeholder:0')
55     y = tf.argmax(result, axis=1)
56
57     def preimg(img):
58         reimg = np.asarray(img.resize((height, width)), dtype=np.float32).reshape(height, width, 3)
59         normimg = 2 * (reimg / 255.0) - 1.0
60         return normimg
61
62
63     # 获得原始图片与预处理图片
64     batchImg = [preimg(Image.open(imgfilename)) for imgfilename in
65                 sample_images]
66     orgImg = [Image.open(imgfilename) for imgfilename in sample_images]
67
68     yv = sess.run(y, feed_dict={input_imgs: batchImg}) # 输入模型
69     print(yv)
70
71     print(yv, np.shape(yv)) # 显示输出结果
72
73     def showresult(yy, img_norm, img_org):
74         plt.figure()
75         p1 = plt.subplot(121)
76         p2 = plt.subplot(122)
77         p1.imshow(img_norm) # 显示图片
78         p1.axis('off')
79         p1.set_title("organization image")
80
81         img = ((img_norm + 1) / 2) * 255
82         p2.imshow(np.asarray(img, np.uint8)) # 显示图片
83         p2.axis('off')
84         p2.set_title("input image")
85
86         print("索引: ", yy, ", ", "年纪: ", labels[yy])
87
88     for yy, img1, img2 in zip(yv, batchImg, orgImg): # 显示每条结果及图片
89         showresult(yy, img1, img2)

```

代码第 41 行，指定了要加载的模型动态图文件。

代码第 53 行，指定了与模型文件对应的输入节点“final_result:0”。

代码第 54 行，指定了与模型文件对应的输出节点“Placeholder:0”。

代码运行后显示以下结果：

```

['1' '10' '100+' '11' '12' '13' '14' '15' '16' '17' '18' '19' '2' '20' '21' '22' '23'
'24' '25' '26' '27' '28' '29' '3' '30' '31' '32' '33' '34' '35' '36' '37' '38' '39' '4'
'40' '41' '42' '43' '44' '45' '46' '47' '48' '49' '5' '50' '51' '52' '53' '54' '55' '56'
'57' '58' '59' '6' '60' '61' '62' '63' '64' '65' '66' '67' '68' '69' '7' '70' '71' '72'

```

```
'73' '74' '75' '76' '77' '78' '79' '8' '80' '81' '82' '83' '84' '85' '86' '87' '88' '89'
'9' '90-95' '96-99']
```

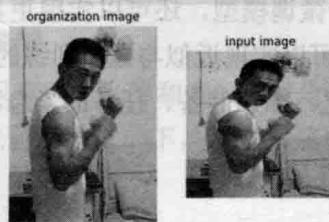


图 5-12 年纪预测结果 (a)

索引: 32, 年纪: 38

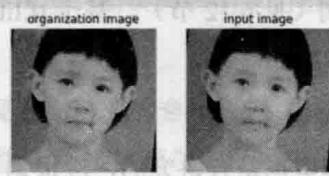


图 5-12 年纪预测结果 (b)

索引: 1, 年纪: 10

输出结果可以分为两部分:

- 第 1 部分是标签的内容。
- 第 2 部分是评估的结果。

在第 2 部分中，每张图片的下面都会显示这个图片的评估结果，其中包括：在模型中的标签索引、该索引对应的标签名称。

5.5.6 扩展：用 TF-Hub 库中的其他模型处理不同领域的分类任务

TF-Hub 库中实现了一个通用的模型框架，它不仅可以处理图像方面的任务，还可以处理很多其他领域的任务。



提示：

可以通过 5.5.2 小节中介绍的预训练模型下载方法获取更多领域的预训练模型。

另外，还可以在 GitHub 网站上的 TF-Hub 主页中找到更多的示例代码。其中包括了文本处理、微调、模型创建、模型使用等多种操作的代码演示。

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

同时，本书第 13 章会通过一个创建 TF-Hub 模型的例子，来详细介绍 TF-Hub 库的相关知识。

5.6 总结

本节将对微调模型方面的技术做一下总结，包括微调的方法及模型选取的方法。

1. 用 TF-Hub 库与 TF-slim 接口微调模型的区别

TF-Hub 库冻结了已有的权重，操作简单，对训练硬件相对要求不高。但它只能微调最后的

输出层，不支持整体联调。

TF-slim 接口不仅仅可以用于微调模型，还可以实现更灵活的训练方式：既可以完全实现 TF-Hub 库中模型的微调方式，也可以实现近似与重新训练的微调方式。

读者可以根据自己的硬件情况、知识储备、任务的紧急程度、对准确度的要求程度来自行选择。

2. 微调模型的更多方法

在 TensorFlow 中，微调模型的方法有很多种，还可以基于 tf.keras 接口进行微调（见 6.10 节），基于 T2T 框架接口进行微调（见 6.12 节），基于 tf.lite 接口进行微调（见 13.3 节）。读者可以根据不同的应用场景灵活运用。

3. 在微调过程中，如何选取预训练模型

在微调过程中，选取预训练模型也是有讲究的，应根据不同的应用场景来定。建议按照以下规则进行选取。

- 单独使用的预训练模型：如果样本量充足，则可以首选精度最高的模型；如果样本量不足，则可以使用 ResNet 模型。
- 嵌入到模型中的预训练模型：需要根据模型的功能来定。
 - 如果模型的输入尺寸固定，则优先 ResNet 模型（例如 8.7 节）。
 - 如果模型的输入尺寸不固定，则可以使用类似 VGG 模型这种支持输入变长尺寸的模型（例如 10.2 节）。



提示：

以上在实际工作中还是应根据具体的网络特征来定。例如，YOLO V3 模型（一个知名的目标识别模型）中就用 Darknet-53 模型作为嵌入层，而非 ResNet 模型（见 8.5 节）。

- 在嵌入式上运行的预训练模型：优先选择 TensorFlow 中提供的裁剪后的模型（见 13.3 节）。

在选取模型的建议中，多次提到了 ResNet 模型。原因是，ResNet 模型在 Imgnet 数据集上输出的特征向量所表现的泛化能力是最强的。具体可以参考以下论文：

<https://arxiv.org/pdf/1805.08974.pdf>

另外，微调模型只是适用于样本不足或运算资源不足的情况下。如果样本不足，则模型微调后的精度与泛化能力会略低于原有的预训练模型；如果样本充足，最好还是使用精度最高的模型，从头开始训练。因为：在样本充足情况下，能在 Imgnet 数据集上表现出高精度的模型，在自定义数据集上也同样可以。

5.7 练习题

由于篇幅有限，本章只针对 TF-slim 接口与 TF-Hub 库各介绍了一个实例。读者还可以在此基础上做更多的练习，真正掌握实际的用法。

5.7.1 基于 TF-slim 接口的练习

1. 使用输出两个分类结果的模型

在实例 11 中，虽然输出结果只有两个（男和女），但是在模型搭建时使用了 3 个分类（又加了一个 None 分类）。读者可以自行尝试一下，看看搭建模型时，使用输出两个分类结果的模型能否正常工作。想想为什么？

2. 尝试从 0 开始训练模型，体会微调与完整训练的区别

在实例 11 中，使用的是预训练模型。如果读者的算力资源充足，则可以尝试从 0 开始训练模型，感受二者的区别。

3. 自己动手准备数据集，实现更高精度的专用模型

在 5.3 节中，介绍了一个用摄像头连接该模型的应用扩展。读者可以尝试用 opencv 库来独立完成该程序（可以参考 13.5 节中 opencv 的使用方法）。另外，读者还可以通过自己的摄像头收集一些与应用场景中一致的样本数据，然后仿照本实例的方法进行训练。

理论上，用自己收集的样本进行训练所得到的模型，会比用本实例中的数据集训练所得到的模型有更高的准确度。因为，训练样本更接近真实样本。

4. 更换模型，实现更高精度的效果

在实例 11 里用的是 NASNet_A_Mobile 模型，该模型相对较小，速度较快，但是准确率偏低。还可以使用其他模型（例如 PNASNet 模型）来进行训练，以达到更好的准确度。读者可以选几个其他的模型尝试一下训练效果。

5. 自由发挥分类任务，玩转图片分类器

如果前面的知识都掌握了，读者可以自行尝试完成一些图片分类的任务。从制作数据集开始，到选择模型、编写代码、训练模型。只要细心就会发现，日常生活中有很多场景都可以用图片分类功能来解决问题。尝试着用本章所学知识来解决它们。

5.7.2 基于 TF-Hub 库的练习

1. 用预处理样本来优化模型

在实例 12 中，使用的是端到端模式对图片中的人物进行年纪评估。还可以对样本进行预处理，只把头像部分截取出来，然后进行训练。看看是否会有更好的效果。

2. 使用更丰富的数据集

在实例 12 中，使用的数据集质量不是太高。还可以写一个爬虫来自己收集数据集。爬虫的做法可以参考《Python 带我起飞——入门、进阶、商业实战》一书的第 11 章。

具体做法是：在百度图片中按照年纪依次进行搜索，将返回的图片结果用爬虫截取下来；然后用自己收集的数据来训练模型，并对目标图片进行测试，观察其准确度。

3. 使用更大的模型或全局微调来提升准确度

将 5.5 节中的模型换作 PNASNet 模型，可以进一步提升准确度。另外还可以仿照 5.7.1 小节中用 TF-slim 接口进行全局微调，这样也可以将准确度提升。读者都可以自己尝试一下。

又）委员会根据财政部预算司有关情况，建议是两个阶段从提出申请之日起增加至自行选取并类表个两出率优势，加些对非特许者，不一定要自己顶着办。（类似 2011 年一个“同

2. 微调模型的更多方法

在 TensorFlow 中，微调模型的接口非常丰富，吸附率为 0.从左到右，S-6.10 然而部分从左到右，再到最右侧的参数，如图所示。读者可以根据不同的应用场景选择使用。

3. 在微调过程中，如何选取预训练模型的参数方案，建议读者参考自 S-6.10 以来的“迁移学习”部分。通过对比不同参数方案的准确率，从中选择最佳的参数组合。数据集与自身或均值匹配，快慢。（迁移学习的“迁移学习”部分有详细介绍）

4. 建议使用卷积神经网络的模型而不是全连接模型，因为前者能更好地捕捉特征，且训练速度更快。

5. 在微调中的预训练模型，本例中选择的是 ResNet-50 模型，因为其准确率较高且模型的

6. 如果模型的输入尺寸固定，则优先 ResNet 集成的更简单，吸附率为 0.从左到右，S-6.10 为 0.5，而卷积神经网络的输入尺寸则需要根据实际情况进行调整，如图所示。如果数据集的一张图片的尺寸为 224x224，则建议将其调整为 256x256，如果数据集的一张图片的尺寸为 128x128，则建议将其调整为 160x160，以此类推。

7. 在微调过程中，建议使用 Adam 优化器，因为其收敛速度更快，且稳定性更好。如果使用 SGD 优化器，则建议设置一个较小的步长，如图所示。

8. 在微调过程中，建议使用交叉熵损失函数，因为其计算速度快且稳定，如图所示。如果使用其他损失函数，则建议将其设置为 0.5，以保证训练过程的稳定性。

9. 在微调过程中，建议使用批次大小为 16 或 32，因为其训练速度更快，且稳定性更好。如果使用批次大小为 1，则建议将其设置为 8，以保证训练过程的稳定性。如果使用批次大小为 64，则建议将其设置为 32，以保证训练过程的稳定性。

10. 在微调过程中，建议使用批次大小为 16 或 32，因为其训练速度更快，且稳定性更好。如果使用批次大小为 1，则建议将其设置为 8，以保证训练过程的稳定性。如果使用批次大小为 64，则建议将其设置为 32，以保证训练过程的稳定性。

11. 在微调过程中，建议使用批次大小为 16 或 32，因为其训练速度更快，且稳定性更好。如果使用批次大小为 1，则建议将其设置为 8，以保证训练过程的稳定性。如果使用批次大小为 64，则建议将其设置为 32，以保证训练过程的稳定性。

由于篇幅有限，只能简要地介绍几种常用的微调方法，希望读者能够在实践的基础上做更多的练习，真正掌握实际的用法。

第 6 章

用TensorFlow编写训练模型的程序

本章介绍如何用 TensorFlow 编写训练模型的程序。通过本章的学习，读者可以掌握多种模型的编写方法，并能够使用几种常用的框架训练模型。

6.1 快速导读

在学习实例之前，有必要了解一下训练模型的基础知识。

6.1.1 训练模型是怎么一回事

训练模型是指，通过程序的反复迭代来修正神经网络中各个节点的值，从而实现具有一定拟合效果的算法。

在训练神经网络的过程中，数据的流向有两个：正向和反向。

- 正向负责预测生成结果，即沿着网络节点的运算方向一层一层地计算下去。
- 反向负责优化调整模型参数，即用链式求导将误差和梯度从输出节点开始一层一层地传递归去，对每层的参数进行调整。

训练模型的完整的步骤如下：

- (1) 通过正向生成一个值，然后计算该值与真实标签之间的误差。
- (2) 利用反向求导的方式，将误差从网络的最后一层传到前一层。
- (3) 对前一层中的参数求偏导，并按照偏导结果的方向和大小来调整参数。
- (4) 通过循环的方式，不停地执行 (1) (2) (3) 这 3 步操作。从整个过程中可以看到，步骤 (1) 的误差越来越小。这表示模型中的参数所需要调整的幅度越来越小，模型的拟合效果越来越好。

在反向的优化过程中，除简单的链式求导外，还可以加入一些其他的算法，使得训练过程更容易收敛。

在 TensorFlow 中，反向传播的算法已经被封装到具体的函数中，读者只需要明白各种算法的特点即可。使用时，可以根据适用的场景直接调用对应的 API，不再需要手动实现。

6.1.2 用“静态图”方式训练模型

“静态图”是 TensorFlow 1.x 版本中张量流的主要运行方式。其运行机制是将“定义”与“运

行”相分离。相当于：先用程序搭建起一个结构（即在内存中构建一个图），让数据（张量流）按照图中的结构顺序进行计算，最终运行出结果。

1. 了解静态图的操作方式

静态图的操作方式可以抽象成两种：模型构建和模型运行。

- 模型构建：从正向和反向两个方向搭建好模型。
- 模型运行：在构建好模型后，通过多次迭代的方式运行模型，实现训练的过程。

在 TensorFlow 中，每个静态图都可以理解成一个任务。所有的任务都要通过会话(session)才能运行。

2. 在 TensorFlow 1.x 版本中使用静态图

在 TensorFlow 1.x 版本中使用静态图的步骤如下：

- (1) 定义操作符（调用 `tf.placeholder` 函数）。
- (2) 构建模型。
- (3) 建立会话（调用 `tf.Session` 之类的函数）。
- (4) 在会话里运行张量流并输出结果。

3. 在 TensorFlow 2.x 版本中使用静态图

在 TensorFlow 2.x 版本中，使用静态图的步骤与在 TensorFlow 1.x 版本中使用静态图的步骤完全一致。

但是，由于静态图不是 TensorFlow 2.x 版本中的默认工作模式，所以在使用时还需要注意两点：

- (1) 在代码的最开始处，用 `tf.compat.v1.disable_v2_behavior` 函数关闭动态图模式（见 6.1.3 小节）。
- (2) 将 TensorFlow 1.x 版本中的静态图接口，替换成 `tf.compat.v1` 模块下的对应接口。例如：
 - 将函数 `tf.placeholder` 替换成函数 `tf.compat.v1.placeholder`。
 - 将函数 `tf.Session` 替换成函数 `tf.compat.v1.Session`。

6.1.3 用“动态图”方式训练模型

“动态图”（eager）是在 TensorFlow 1.3 版本之后出现的。到了 1.11 版本时，它已经变得较完善。在 TensorFlow 2.x 版本中，它已经变成了默认的工作方式。

动态图主要是在原始的静态图上做了编程模式的优化。它使得使用 TensorFlow 变得更简单、更直观。

例如，调用函数 `tf.matmul` 后，在动态图与静态图中的区别如下：

- 在动态图中，程序会直接得到两个矩阵相乘的值。
- 在静态图中，程序只会生成一个 OP（操作符）。该 OP 必须在绘画中使用 `run` 方法才能进行真正的计算，并输出结果。

1. 了解动态图的编程方式

所谓的动态图是指，代码中的张量可以像 Python 语法中的其他对象一样直接参与计算。不再需要像静态图那样用会话（session）对张量进行运算。

2. 在TensorFlow 1.x 版本中使用动态图

启用动态图，只需要在程序的最开始处加上以下代码：

```
tf.enable_eager_execution()
```

这行代码的作用是——开启动态图的计算功能。



提示：

代码“`tf.enable_eager_execution()`”必须在所有的代码之前执行，否则会报错。

3. 在TensorFlow 2.x 版本中使用动态图

在TensorFlow 2.x 版本中，已经将动态图设为了默认的工作模式。使用动态图时，直接编写代码即可。

TensorFlow 1.x 中的 `tf.enable_eager_execution` 函数在TensorFlow 2.x 版本中已经被删除，另外在TensorFlow 2.x 版本中还提供了关闭动态图与启用动态图的两个函数。

- 关闭动态图函数：`tf.compat.v1.disable_v2_behavior`。
- 启用动态图函数：`tf.compat.v1.enable_v2_behavior`。

4. 动态图的原理及不足

在创建动态图的过程中，默认也建立了一个会话（session）。所有的代码都在该会话（session）中进行，而且该会话（session）具有进程相同的生命周期。这表示：当前程序中只能有一个会话（session），并且该会话一直处于打开状态，无法被关闭。

动态图的不足之处是：在动态图中，无法实现多会话（session）操作。

对于习惯了多会话（session）开发模式的用户，需要将静态图中的多会话逻辑转化单会话逻辑后才可以移植到动态图中。

6.1.4 什么是估算器框架接口（Estimators API）

估算器框架接口（Estimators API）是TensorFlow中的一种高级API。它提供了一整套训练模型、测试模型的准确率，以及生成预测的方法。

用户在估算器框架中开发模型，只需要实现对应的方法即可。整体的数据流向搭建，全部交给估算器框架来做。估算器框架内部会自动实现：检查点文件的导出与恢复、保存TensorBoard的摘要、初始化变量、异常处理等操作。



提示：

TensorFlow 2.x 版本可以完全兼容 TensorFlow 1.x 版本的估算器框架代码。用估算器框架开发模型代码，不需要考虑版本移植的问题。

1. 估算器框架的组成

估算器框架是在 `tf.layers` 接口（见 6.1.5 小节）上构建而成的。估算器框架可以分为三个主要部分。

- 输入函数：主要由 `tf.data.Dataset` 接口组成，可以分为训练输入函数（`train_input_fn`）和测试输入函数（`eval_input_fn`）。前者用于输出数据和训练数据，后者用于输出验证数据和测试数据。
- 模型函数：由模型（`tf.layers` 接口）和监控模块（`tf.metrics` 接口）组成，主要用来实现训练模型、测试（或验证）模型、监控模型参数状况等功能。
- 估算器：将各个部分“粘合”起来，控制数据在模型中的流动与变换，并控制模型的各种行为（运算）。它类似于计算机中的操作系统。

2. 估算器中的预置模型

估算器框架除支持自定义模型外，还提供了一些封装好的常用模型，例如：基于线性的回归和分类模型（`LinearRegressor`、`LinearClassifier`）、基于深度神经网络的回归和分类模型（`DNNRegressor`、`DNNClassifier`）等。直接使用这些模型，可以省去大量的开发时间。在第 7 章中会介绍模型的具体使用。

3. 基于估算器开发的高级模型

在 TensorFlow 中，还有两个基于估算器开发的高级模型框架——TFTS 与 TF-GAN。

- TFTS：专用于处理序列数据的通用框架。
- TF-GAN：专用于处理对抗神经网络（GAN）的通用框架。

在 9.7 节会有 TFTS 框架的具体介绍及详细实例。

4. 估算器的利与弊

估算器框架的价值主要是，对模型的训练、使用等流程化的工作做了高度集成。它适用于封装已经开发好的模型代码。它会使整体的工程代码更加简洁。该框架的弊端是：由于对流程化的工作集成度太高，导致在开发模型过程中无法精确控制某个具体的环节。

综上所述，估算器框架不适用于调试模型的场景，但适用于对成熟模型进行训练、使用的场景。

6.1.5 什么是 `tf.layers` 接口

`tf.layers` 接口是一个与 TF-slim 接口类似的 API，该接口的设计是与神经网络中“层”的概念相匹配的。

例如，在用 `tf.layers` 接口开发含有多个卷积层、池化层的神经网络时，会针对每一层网络定义一个以“`tf.layers.`”开头的函数，然后再将这些神经网络层依次连接起来。

`tf.layers` 接口的所有函数都可以在本地的以下路径中找到：

```
Anaconda3\lib\site-packages\tensorflow\tools\api\generator\api\layers\__init__.py
```

在源码中，可以通过查看函数定义的方法了解每个 `tf.layers` 接口的用法。具体操作如下：

(1) 用鼠标右击指定的函数名。

(2) 在弹出的菜单中选择“go to definition”命令，如图6-1所示。

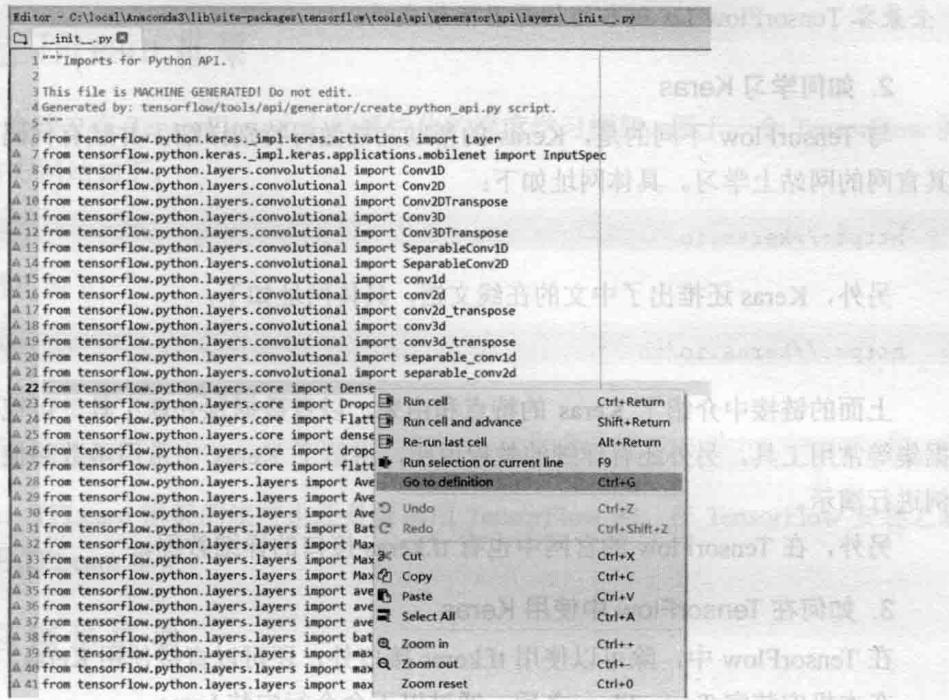


图6-1 tf.layers函数

tf.layers接口常用于动态图中，而TF-slim接口则更多地应用在静态图中。



提示：

用tf.layers接口开发模型代码，需要考虑版本移植的问题。在TensorFlow 2.x版本中，所有tf.layers接口都需要被替换为tf.compat.v1.layers。

另外，在TensorFlow 2.x版本中，tf.layers模块更多用于tf.keras接口的底层实现。如果是开发新项目，则建议直接使用tf.keras接口。如果要重构已有的项目，也建议使用tf.keras接口进行替换。

6.1.6 什么是tf.keras接口

tf.keras接口是TensorFlow中支持Keras语法的高级API。它可以将用Keras语法实现的代码程序移植到TensorFlow上来运行。

1. 什么是Keras

Keras是一个用Python编写的高级神经网络接口。它是目前最通用的前端神经网络接口。

基于Keras开发的代码可以在TensorFlow、CNTK、Theano等主流的深度学习框架中直接运行。在TensorFlow 2.x版本中用tf.keras接口在动态图上开发模型是官网推荐的主流方法之一。

**提示：**

用 tf.keras 接口开发模型代码，不需要考虑版本移植的问题。TensorFlow 2.x 版本可以完全兼容 TensorFlow 1.x 版本的估算器框架代码。

2. 如何学习 Keras

与 TensorFlow 不同的是，Keras 的帮助文档做得特别详细，并赋有代码实例。可以直接在其官网的网站上学习。具体网址如下：

```
https://keras.io
```

另外，Keras 还推出了中文的在线文档，具体网址如下：

```
https://keras.io/zh
```

上面的链接中介绍了 Keras 的特点和由来，以及数据预处理工具、可视化工具、集成的数据集等常用工具。另外还有详细的教程说明，讲解了 Keras 中常用函数的使用方法，以及用实例进行演示。

另外，在 TensorFlow 的官网中也有 tf.keras 接口的详细教程。

3. 如何在 TensorFlow 中使用 Keras

在 TensorFlow 中，除可以使用 tf.keras 接口外，还可以直接使用 Keras。

在本机安装完 TensorFlow 之后，通过以下命令行安装 keras。

```
pip install keras
```

这时使用的 Keras 代码，会默认将 TensorFlow 作为后端来进行运算。

4. Keras 与 tf.keras 接口

在开发过程中，所有的 Keras 都可以用 tf.keras 接口来无缝替换（具体细节略有一点差别，可以忽略）。

在开发算法原型时，可以直接用 tf.keras 接口中集成的数据集（如 boston_housing、cifar10、cifar100、fashion_mnist、imdb、mnist、reuters 等）来快速验证模型的效果。

当然，在实际开发中，每种不同的高级接口都有它的学习成本。读者应根据自己对某个 API 的熟练程度选取适合自己的 API。

6.1.7 什么是 tf.js 接口

tf.js（TensorFlow.js）是基于 JavaScript 的 TensorFlow 支持库，它可以用浏览器 API（比如 WebGL）来加速计算。这意味着，TensorFlow 程序可以运行在不同的环境当中，让 AI 无处不在。

tf.js 接口的出现，对大量的 web 开发工程师是一件好事。它使得“用 JavaScript 开发 AI”变成可能。

更多信息可以参考以下链接:

<https://js.tensorflow.org>

6.1.8 什么是TFLearn框架

TFLearn是一个建立在TensorFlow之上的模块化的深度学习框架，属于一个TensorFlow的第三方API，其官方网站如下：

<http://tflearn.org>

对应的代码链接如下：

<https://github.com/tflearn/tflearn>

可以通过以下的pip命令安装TFLearn：

`pip install tflearn`

类似于Keras，TFLearn框架的底层也还是要调用TensorFlow的。在TensorFlow安装之后才可以安装和使用TFLearn框架。

6.1.9 该选择哪种框架

与TensorFlow相关的多种API已经非常多。对于使用者来讲，没必要把全部的API都学精。所有的API从使用角度来看，大致可以分为3个层面：

- 对于网络单层的封装（TF-slim、tf.layers）。
- 对于处理框架的封装（Estimators、eager）。
- 对于框架及网络的整体封装（TFLearn、tf.keras）。



提示：

读者可以根据自己的知识基础和使用场景，选择一至两种API并学精它，便于在自己开发模型时使用。

至于其他的API，大致了解一下即可，能够达到从GitHub网站上下载源码并进行简单的修改、调试的地步就可以了。

1. 从学习的角度分析

从学习的角度来讲，原生的API是必须要学的。它可以最大化地掌控TensorFlow程序。有了这个基础再去了解其他API就不会费劲。上面说的3个层面的API，建议每一个层面都挑选一个去了解即可。额外强调的是，tf.keras接口还是非常值得去认真学习的，因为：在整个GitHub网站上的代码中，使用Keras实现的深度学习项目占比很高。

2. 从工程的角度

从工程的角度来讲，推荐使用tf.keras、Estimators、eager这三种框架。因为这三种是在动态图中，也可以用`with tf.device`方式对硬件资源进行指定。

TensorFlow 2.x 版本中支持的主流框架，具有很好的技术延续性。在实际开发中，根据不同的开发场景，给出的搭配建议如下。

- 在开发并调试模型的场景中，推荐用 `tf.keras` 接口搭建模型，并在 `eager` 框架进行训练和调参。动态图框架有更好的灵活性，可以对网络的各个环节进行改动。
- 在对成熟模型进行训练的场景中，在模型开发工作结束之后，可以用 `tf.keras` 接口中 `model` 类的集成方法或将模型代码封装在 `Estimators` 框架中，进行训练或评估等操作。
- 在对外发布模型的源代码的场景中，在公布开源模型或项目交付时，也会将模型代码封装在 `Estimators` 框架中。`Estimators` 框架对模型的流程化代码进行了高度的集成，可以使源码变得更加简洁。

6.1.10 分配运算资源与使用分布策略

在 TensorFlow 中，分配 GPU 的运算资源是很常见的事情。大体可以分为 3 种情况：

- 为整个程序指定 GPU 卡。
- 为整个程序指定所占的 GPU 显存。
- 在程序内部调配不同的 OP（操作符）到指定 GPU 卡。

通过指定硬件的运算资源，可以提高系统的运算性能，从而缩短模型的训练时间。在实现时，可以调用底层的接口进行手动调配；也调用上层的高级接口，进行分布策略的应用。具体的做法如下：

1. 为整个程序指定 GPU 卡

主要是通过设置 `CUDA_VISIBLE_DEVICES` 变量来实现的。例如：

```
CUDA_VISIBLE_DEVICES=1      #代表只使用序号(device)为1的卡
CUDA_VISIBLE_DEVICES=0,1    #代表只使用序号(device)为0和1的卡
CUDA_VISIBLE_DEVICES="0,1" #代表只使用序号(device)为0和1的卡
CUDA_VISIBLE_DEVICES=0,2,3 #代表只使用序号(device)为0、2、3的卡，序号为1的卡不可见
CUDA_VISIBLE_DEVICES=""   #代表不使用GPU卡
```

设置该变量有两种方式：

(1) 命令行方式。

在通过命令行运行程序时，可以在“`python`”前加上“`CUDA_VISIBLE_DEVICES`”，如下所示：

```
root@user-NULL:~/test# CUDA_VISIBLE_DEVICES=1 python 要运行的Python程序.py
```

(2) 在程序中设置。

在程序的最开始处添加以下代码：

```
import os
os.environ["CUDA_VISIBLE_DEVICES"] = "0"
```

`CUDA_VISIBLE_DEVICES` 的值可以是字符串类型，也可以是数值类型。

**提示：**

设置 CUDA_VISIBLE_DEVICES，主要是为了让程序对指定的 GPU 卡可见。这时系统只会对可见的 GPU 卡编号。在运行时，这个编号并不代表 GPU 卡的真正序号。

例如：

设置 CUDA_VISIBLE_DEVICES=1，则运行程序后会显示当前任务是在 device:GPU:0 上运行的。见下面的输出信息：

```
2018-06-24 06:24:53.535524: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1053]
Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 10764 MB memory)
-> physical GPU (device: 0, name: Tesla K80, pci bus id: 0000:86:00.0, compute capability: 3.7)
```

这说明，当前程序会把系统中的序号为“1”的卡当作自己的第0块卡来使用。

2. 为整个程序指定所占的 GPU 显存

在 TensorFlow 中，为整个程序分配 GPU 显存的方式，主要是靠构建 tf.ConfigProto 类来实现的。tf.ConfigProto 类可以理解成一个容器。在以下网址可以找到该类的定义：

```
https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/protobuf/config.proto
```

在上述链接中可以看到各种定制化选项的定义。这些定制化选项，都可以放置到 tf.ConfigProto 类中。例如：RPCOptions、RunOptions、GPUOptions、graph_options 等。

可以通过定义 GPUOptions 来控制运算时的硬件资源分配，例如：使用哪个 GPU、需要占用多大缓存等。在 6.4 节还会通过一个具体的例子演示如何使用 tf.ConfigProto 类。

3. 在程序内部，调配不同的 OP（操作符）到指定 GPU 卡

在代码前使用 tf.device 语句，可以指定当前的语句在哪个设备上运行。例如：

```
with tf.device('/cpu:0'):
```

表示当前代码在第 0 块 CPU 上运行。

4. 其他配置相关的选项

其他与指派设备的选项如下。

(1) 自动选择运行设备：allow_soft_placement。

如果 tf.device 指派的设备不存在或者不可用，为防止程序发生等待或异常，可以设置 tf.ConfigProto 中的参数 allow_soft_placement=True，表示允许 TensorFlow 自动选择一个存在并且可用的设备来运行操作。

(2) 记录设备指派情况：log_device_placement。

设置 tf.ConfigProto 中参数 log_device_placement = True，可以得到 operations 和 Tensor 被指派到哪个设备（几号 CPU 或几号 GPU）上的运行信息，并在终端显示。

5. 动态图的设备指派

在动态图中，也可以用 with tf.device 方法对硬件资源进行指派。

除此之外，还可以调用动态图中张量的 `gpu`、`cpu` 方法来进行硬件资源的指派。以下面代码为例：

```
import tensorflow as tf
import tensorflow.contrib.eager as tfe
tfe.enable_eager_execution()          # 启动动态图
print(tf.contrib.eager.num_gpus())    # 获得当前 GPU 个数
x = tf.random_normal([10, 10])       # 定义一个张量
x_gpu0 = x.gpu()                   # 通过该张量的 gpu 方法，将其复制到 GPU 上执行，默认是 0 号 GPU
x_cpu = x.cpu()                   # 通过该张量的 cpu 方法，将其复制到 CPU 上执行

_ = tf.matmul(x_gpu0, x_gpu0)      # 在第 0 号 GPU 上运行乘法
_ = tf.matmul(x_cpu, x_cpu)        # 在 CPU 上运行乘法

if tfe.num_gpus() > 1:             # 当 GPU 个数大于 1 时
    x_gpu1 = x.gpu(1)              # 将该张量复制到第 1 号 GPU 上
    _ = tf.matmul(x_gpu1, x_gpu1)  # 在第 1 号 GPU 上运行乘法
```

6. 使用分布策略

分配运算资源的最简单方式就是使用分布策略。使用分布策略也是官方推荐主流方式。该方式针对几种常用的训练场景，将资源分配的算法封装成不同的分布策略。用户在训练模型时，只需要选择对应的分布策略即可。运行时，系统会按照该策略中的算法进行资源分配，使机器的运算性能最大化的发挥出来。

(1) 具体的分布策略及对应的场景如下。

- **MirroredStrategy**（镜像策略）：该策略适用于一机多 GPU 的场景，将计算任务均匀地分配到每块 GPU 上。
- **CollectiveAllReduceStrategy**（集合规约策略）：该策略适用于分布训练场景，用多台机器训练一个模型任务。先将每台机器上使用 `MirroredStrategy` 策略进行训练，再将多台机器的结果进行规约合并。
- **ParameterServerStrategy**（参数服务器策略）：适用于分布训练场景。也是用多台机器来训练一个模型任务。在训练过程中，使用参数服务器来统一管理每个 GPU 的训练参数。

(2) 使用方式。

分布策略的使用方式非常简单。需要实例化一个分布策略对象，并将其作为参数传入训练模型中。以 `MirroredStrategy` 策略为例，实例化的代码如下：

```
distribution = tf.contrib.distribute.MirroredStrategy()
```

实例化后的对象 `distribution` 可以传入 `tf.keras` 接口中 `model` 类的 `fit` 方法中，用于训练。例如：

```
model.compile(loss='mean_squared_error',
               optimizer=tf.train.GradientDescentOptimizer(learning_rate=0.2),
               distribute=distribution)
```

也可以传入估算器的 `RunConfig` 中，生成配置对象 `config`，并将该对象传入估算器的