Chapter 13

Setup

In [1]:
```
import sys

assert sys.version_info >= (3, 7)
```

In [2]:
```
from packaging import version
import sklearn

assert version.parse(sklearn.__version__) >= version.parse("1.0.1")
```

In [3]:
```
import tensorflow as tf

assert version.parse(tf.__version__) >= version.parse("2.8.0")
```

1. tf.data API

整个tf.data API都围绕着 **tf.data.Dataset** 的概念展开：这表示一系列数据项。通常您将使用逐渐从磁盘中读取数据的数据集，但为了简单起见，让我们使用 **tf.data.Dataset.from_tensor_slices()** 从一个简单的数据张量创建一个数据集:

In [4]:
```
import tensorflow as tf

X = tf.range(10)  # any data tensor

dataset = tf.data.Dataset.from_tensor_slices(X)

dataset
```

Out[4]:
```
<TensorSliceDataset element_spec=TensorSpec(shape=(), dtype=tf.int32, name=None)>
```

from_tensor_slices() 函数取一个张量并创建一个 **tf.data.Dataset** ，其元素是X沿第一个维度的所有切片，因此这个数据集包含10项：张量0、1、2、...，9。在这种情况下，如果我们使用tf.data.Dataset.range(10)，我们将获得相同的数据集（除了元素将是64位整数，而不是32位整数）

In [7]:
```
for item in dataset:
    print(item)
```

```
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)
tf.Tensor(3, shape=(), dtype=int32)
tf.Tensor(4, shape=(), dtype=int32)
tf.Tensor(5, shape=(), dtype=int32)
tf.Tensor(6, shape=(), dtype=int32)
tf.Tensor(7, shape=(), dtype=int32)
tf.Tensor(8, shape=(), dtype=int32)
tf.Tensor(9, shape=(), dtype=int32)
```

tf.data API是一个流式API：您可以非常有效地遍历数据集的项，但该API不是为索引或切片而设计的。

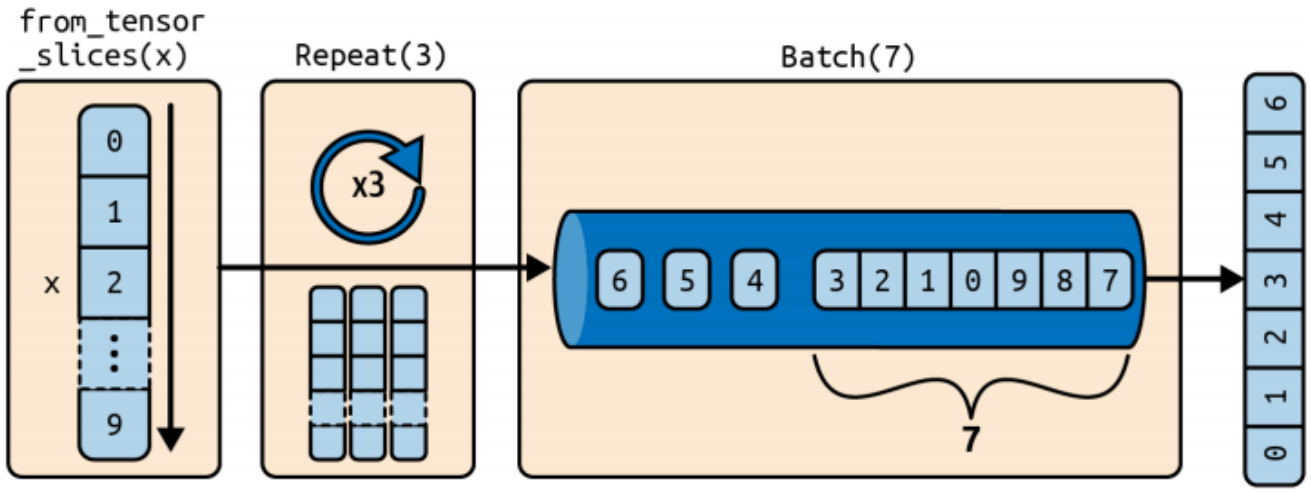
数据集还可以包含张量的元组，或者名称/张量对的字典，甚至是嵌套的元组和张量的字典。当分割元组、字

典或嵌套结构时，数据集将只对其包含的张量进行切片，同时保留元组/字典结构。例如：

In [11]:
```
X_nested = {"a": ([1, 2, 3], [4, 5, 6]), "b": [7, 8, 9]}
dataset = tf.data.Dataset.from_tensor_slices(X_nested)
for item in dataset:
    print(item)
```

```
{'a': (<tf.Tensor: shape=(), dtype=int32, numpy=1>, <tf.Tensor: shape=(), dtype=int32, n
umpy=4>), 'b': <tf.Tensor: shape=(), dtype=int32, numpy=7>}
{'a': (<tf.Tensor: shape=(), dtype=int32, numpy=2>, <tf.Tensor: shape=(), dtype=int32, n
umpy=5>), 'b': <tf.Tensor: shape=(), dtype=int32, numpy=8>}
{'a': (<tf.Tensor: shape=(), dtype=int32, numpy=3>, <tf.Tensor: shape=(), dtype=int32, n
umpy=6>), 'b': <tf.Tensor: shape=(), dtype=int32, numpy=9>}
```

1.1 链式转换（Chaining Transformations）



一旦您有了一个数据集，您就可以通过调用它的转换方法来对其应用各种转换。每个方法都返回一个新的数据集，因此您可以像这样链接转换：

In [12]:
```
dataset = tf.data.Dataset.from_tensor_slices(tf.range(10))
dataset = dataset.repeat(3).batch(7)
for item in dataset:
    print(item)
```

```
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int32)
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int32)
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int32)
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int32)
tf.Tensor([8 9], shape=(2,), dtype=int32)
```

在本例中，我们首先调用原始数据集上的 **repeat()** 方法，它返回一个新的数据集，重复原始数据集的项三次。当然，这不会将内存中的所有数据复制三次！如果您不带参数调用此方法，则新数据集将永远重复源数据集，因此遍历该数据集的代码将必须决定何时停止。

然后我们在这个新的数据集上调用 **batch()** 方法，这再次创建一个新的数据集。这一个将把前一个数据集的项目分组为7个项。

最后，我们遍历这个最终数据集的项。**batch()** 方法必须输出大小为2而不是7的最终批处理，但是如果希望删除最终批处理，则可以用 **drop_remainder=True** 调用 batch()，这样所有批处理都具有完全相同的大小。

数据集方法不修改数据集，它们会创建新的数据集。因此，请确保保留对这些新数据集的引用（例如，使用 dataset = ...），否则什么也不会发生。

您还可以通过调用 **map()** 方法来转换这些项。

例如，这将创建一个新的数据集，其中所有批都乘以2：

In [13]:
```
dataset = dataset.map(lambda x: x * 2)  # x is a batch
for item in dataset:
    print(item)
```

```
tf.Tensor([ 0  2  4  6  8 10 12], shape=(7,), dtype=int32)
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int32)
tf.Tensor([ 8 10 12 14 16 18  0], shape=(7,), dtype=int32)
tf.Tensor([ 2  4  6  8 10 12 14], shape=(7,), dtype=int32)
tf.Tensor([16 18], shape=(2,), dtype=int32)
```

map() 方法是您将调用的用于对数据应用任何预处理的方法。有时这将包括相当密集的计算，比如重塑或旋转图像，所以您通常希望生成多个线程来加速速度。这可以通过将 **num_parallel_calls** 参数设置为要运行的线程数或 **tf.data.AUTOTUNE** 来实现。请注意，您传递给map()方法的函数必须可以转换为TF函数（请参见第12章）。

也可以使用 **filter()** 方法简单地过滤数据集。

例如，此代码创建了一个数据集，其中只包含总和大于50的批块：

In [14]:
```
dataset = dataset.filter(lambda x: tf.reduce_sum(x) > 50)
for item in dataset:
    print(item)
```

```
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int32)
tf.Tensor([ 8 10 12 14 16 18  0], shape=(7,), dtype=int32)
tf.Tensor([ 2  4  6  8 10 12 14], shape=(7,), dtype=int32)
```

您通常只需要查看一个数据集中的几个项。您可以使用 **take()** 方法：

In [18]:
```
for item in dataset.take(2):
    print(item)
```

```
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int32)
tf.Tensor([ 8 10 12 14 16 18  0], shape=(7,), dtype=int32)
```

1.2 数据洗牌（Shuffling the Data）

正如我们在第4章中所讨论的，当训练集中的实例是独立的和同分布的（IID）时，梯度下降效果最好。确保这一点的一种简单方法是使用 **shuffle()** 方法来洗牌实例。

它将创建一个新的数据集，该数据集将首先用源数据集的第一项填充一个缓冲区。然后，每当要求它输入一个项时，它就会从缓冲区中随机取出一个项，并用源数据集中的一个新项替换它，直到它完全遍历源数据集。此时，它将继续从缓冲区中随机抽取出项，直到它为空。

您必须指定缓冲区的大小，并且使其足够大是很重要的，否则改组将不会非常有效。只是不要超过你所拥有的内存量，尽管即使你有很多内存量，也没有必要超出数据集的大小。如果您在每次运行程序时都想要相同的随机顺序，则可以提供一个随机种子。

例如，下面的代码创建并显示一个包含整数0到9的数据集，重复两次，使用大小为4的缓冲区和42的随机种子进行打乱，并用批处理大小为7进行批处理：

In [19]:
```
dataset = tf.data.Dataset.range(10).repeat(2)
dataset = dataset.shuffle(buffer_size=4, seed=42).batch(7)
```

```
for item in dataset:
    print(item)
```

```
tf.Tensor([1 4 2 3 5 0 6], shape=(7,), dtype=int64)
tf.Tensor([9 8 2 0 3 1 4], shape=(7,), dtype=int64)
tf.Tensor([5 7 9 6 7 8], shape=(6,), dtype=int64)
```

如果您在一个被打乱的数据集上调用 **repeat()** ，默认情况下，它将在每次迭代中生成一个新的顺序。这通常是一个好主意，但是如果您希望在每次迭代中重复使用相同的顺序（例如，用于测试或调试），您可以在调用 **shuffle()** 时设置 **reshuffle_each_iteration=False** 。

对于不适合内存的大型数据集，这种简单的洗牌缓冲区方法可能是不够的，因为缓冲区与数据集相比会很小。

一种解决方案是洗牌源数据本身（例如，在Linux上，您可以使用shuf命令洗牌文本文件）。这肯定会改善洗牌很多！即使源数据被打乱，您通常也会希望打乱更多，或者在每个时代重复相同的顺序，模型最终可能会有偏差（例如，由于源数据的顺序中偶然出现的一些虚假模式）。

为了对实例进行更多的洗牌，一种常见的方法是将源数据分割成多个文件，然后在训练期间以随机的顺序读取它们。但是，位于同一文件中的实例最终仍然会彼此接近。为了避免这种情况，您可以随机选择多个文件并同时读取它们，交错它们的记录。然后，在此基础上，您可以使用 **shuffle()** 方法添加一个洗牌缓冲区。如果这听起来需要很多工作，不用担心：tf.dataAPI在几行代码中使所有这些成为可能。让我们来看看你能如何做到这一点。

1.3 多个文件的交错行（Interleaving Lines from Multiple Files）

首先，假设您已经加载了加州住房数据集，将其打乱（除非它已经被打乱），并将其分成训练集、验证集和测试集。然后你将每个集合分割成许多CSV文件，每个文件看起来都是这样的（每一行包含8个输入特性加上目标房子值的中值）：

In [20]:
```
# fetches, splits and normalizes the California housing dataset

from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split

housing = fetch_california_housing()

X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target.reshape(-1, 1), random_state=42)

X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)
```

In [21]:
```
# split the dataset into 20 parts and save it to CSV files

import numpy as np
from pathlib import Path

def save_to_csv_files(data, name_prefix, header=None, n_parts=10):
    housing_dir = Path() / "datasets" / "housing"
    housing_dir.mkdir(parents=True, exist_ok=True)
    filename_format = "my_{}_{:02d}.csv"

    filepaths = []
    m = len(data)
    chunks = np.array_split(np.arange(m), n_parts)
    for file_idx, row_indices in enumerate(chunks):
        part_csv = housing_dir / filename_format.format(name_prefix, file_idx)
```

```
        filepaths.append(str(part_csv))
        with open(part_csv, "w") as f:
            if header is not None:
                f.write(header)
                f.write("\n")
            for row_idx in row_indices:
                f.write(",".join([repr(col) for col in data[row_idx]]))
                f.write("\n")
    return filepaths

train_data = np.c_[X_train, y_train]
valid_data = np.c_[X_valid, y_valid]
test_data = np.c_[X_test, y_test]

header_cols = housing.feature_names + ["MedianHouseValue"]

header = ",".join(header_cols)

train_filepaths = save_to_csv_files(train_data, "train", header, n_parts=20)
valid_filepaths = save_to_csv_files(valid_data, "valid", header, n_parts=10)
test_filepaths = save_to_csv_files(test_data, "test", header, n_parts=10)
```

In [22]:
```
print("".join(open(train_filepaths[0]).readlines()[:4]))
```

```
MedInc,HouseAge,AveRooms,AveBedrms,Population,AveOccup,Latitude,Longitude,MedianHouseVal
ue
3.5214,15.0,3.0499445061043287,1.106548279689234,1447.0,1.6059933407325193,37.63,-122.4
3,1.442
5.3275,5.0,6.490059642147117,0.9910536779324056,3464.0,3.4433399602385686,33.69,-117.39,
1.687
3.1,29.0,7.5423728813559325,1.591525423728813,1328.0,2.2508474576271187,38.44,-122.98,
1.621
```

让我们还假设 **train_filepaths** 包含训练文件路径的列表（你也有valid_filepaths和test_filepaths）：

In [23]:
```
train_filepaths
```

Out[23]:
```
['datasets\\housing\\my_train_00.csv',
 'datasets\\housing\\my_train_01.csv',
 'datasets\\housing\\my_train_02.csv',
 'datasets\\housing\\my_train_03.csv',
 'datasets\\housing\\my_train_04.csv',
 'datasets\\housing\\my_train_05.csv',
 'datasets\\housing\\my_train_06.csv',
 'datasets\\housing\\my_train_07.csv',
 'datasets\\housing\\my_train_08.csv',
 'datasets\\housing\\my_train_09.csv',
 'datasets\\housing\\my_train_10.csv',
 'datasets\\housing\\my_train_11.csv',
 'datasets\\housing\\my_train_12.csv',
 'datasets\\housing\\my_train_13.csv',
 'datasets\\housing\\my_train_14.csv',
 'datasets\\housing\\my_train_15.csv',
 'datasets\\housing\\my_train_16.csv',
 'datasets\\housing\\my_train_17.csv',
 'datasets\\housing\\my_train_18.csv',
 'datasets\\housing\\my_train_19.csv']
```

或者，您也可以使用文件模式；例如，train_filepaths="datasets/housing/my*train*.csv"。现在让我们创建一个只包含这些文件路径的数据集：

In [25]:
```
filepath_dataset = tf.data.Dataset.list_files(train_filepaths,
                                              seed=42)
```

```
filepath_dataset
```

Out[25]:
```
<ShuffleDataset element_spec=TensorSpec(shape=(), dtype=tf.string, name=None)>
```

默认情况下，**list_files()** 函数返回一个打乱文件路径的数据集。一般来说，这是一件好事，但如果出于某种原因，可以设置 **shuffle=False** 。

In [26]:
```
# shows that the file paths are shuffled
for filepath in filepath_dataset:
    print(filepath)
```

```
tf.Tensor(b'datasets\\housing\\my_train_05.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_16.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_01.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_17.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_00.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_14.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_10.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_02.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_12.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_19.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_07.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_09.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_13.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_15.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_11.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_18.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_04.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_06.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_03.csv', shape=(), dtype=string)
tf.Tensor(b'datasets\\housing\\my_train_08.csv', shape=(), dtype=string)
```

接下来，您可以调用 **interleave()** 方法，一次读取五个文件，并交错它们的行。您还可以使用跳过 **skip()** 方法跳过每个文件的第一行，即头行)：

In [27]:
```
n_readers = 5
dataset = filepath_dataset.interleave(
    lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
    cycle_length=n_readers)
```

interleave() 方法将创建一个数据集，该数据集将从 **filepath_dataset** 中提取5个文件路径，对于每个数据路径，它将调用您给它的函数（在本例中是一个 **lambda** ）来创建一个新的数据集（在本例中是一个 **TextLineDataset** ）。

需要说明的是，在这个阶段将总共有7个数据集：文件路径数据集、交错数据集，以及由交错数据集内部创建的5个TextLineDatasets。当您遍历交错数据集时，它将循环通过这五个TextLineDatasets，每次从每个数据集中读取一行，直到所有数据集都退出项目。然后，它将从filepath_dataset中获取接下来的五个文件路径，并以相同的方式交织它们，以此类推，直到文件路径耗尽。为了实现交错工作，最好有相同长度的文件；否则，最长文件的结尾将不会交错。

默认情况下，**interleave()** 不使用并行性；它只是按顺序从每个文件中读取一行。如果您想让它实际上并行读取文件，您可以将 **interleave()** 方法的 **num_parallel_calls** 参数设置为您想要的线程数（请回想一下，map() 方法也有这个参数）。你甚至可以将它设置为 **tf.data.AUTOTUNE** 使TensorFlow根据可用的CPU动态选择正确的线程数量。让我们来看看这个数据集现在包含了什么：

In [28]:
```
for line in dataset.take(5):
    print(line)
```

```
tf.Tensor(b'4.5909,16.0,5.475877192982456,1.0964912280701755,1357.0,2.9758771929824563,3
3.63,-117.71,2.418', shape=(), dtype=string)
tf.Tensor(b'2.4792,24.0,3.4547038327526134,1.1341463414634145,2251.0,3.921602787456446,3
4.18,-118.38,2.0', shape=(), dtype=string)
tf.Tensor(b'4.2708,45.0,5.121387283236994,0.953757225433526,492.0,2.8439306358381504,37.
48,-122.19,2.67', shape=(), dtype=string)
tf.Tensor(b'2.1856,41.0,3.7189873417721517,1.0658227848101265,803.0,2.0329113924050635,3
2.76,-117.12,1.205', shape=(), dtype=string)
tf.Tensor(b'4.1812,52.0,5.701388888888889,0.9965277777777778,692.0,2.4027777777777777,3
3.73,-118.31,3.215', shape=(), dtype=string)
```

可以将一个文件路径列表传递给TextLineDataset构造函数：它将一行一行地浏览每个文件。如果您还将 **num_parallel_reads** 参数设置为大于1的数字，那么数据集将并行读取该数量的文件，并交错它们的行（无需调用交错（）方法）。但是，它不会打乱文件，也不会跳过标题行。

1.4 预处理数据（Preprocessing the Data）

现在我们有了一个housing数据集，它将每个实例作为包含字节字符串的张量返回，我们需要做一些预处理，包括解析字符串和缩放数据。让我们实现几个自定义函数，来执行此预处理：

In [29]:
```
# compute the mean and standard deviation of each feature

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(X_train)
```

Out[29]:
```
▾ StandardScaler
StandardScaler()
```

In [30]:
```
X_mean, X_std = scaler.mean_, scaler.scale_  # extra code
n_inputs = 8

def parse_csv_line(line):
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
    fields = tf.io.decode_csv(line, record_defaults=defs)
    return tf.stack(fields[:-1]), tf.stack(fields[-1:])

def preprocess(line):
    x, y = parse_csv_line(line)
    return (x - X_mean) / X_std, y
```

代码解释：

1. 首先，代码假设我们已经预先计算了训练集中每个特征的平均值和标准分布。X_mean 和 X_std 只是一维张量（或NumPy数组），包含8个浮点数，每个输入特征一个。这可以使用Scikit-Learn StandardScaler在足够大的随机样本上完成。在本章的后面，我们将使用一个Keras预处理层来代替。
2. parse_csv_line() 函数接受一条CSV行并解析它。为了帮助实现这一点，它使用了tf.io.decode_csv()函数，它需要两个参数：第一个是要解析的行，第二个是包含CSV文件中每个列的默认值的数组。这个数组（defs）不仅告诉TensorFlow每个列的默认值，而且还告诉列的数量及其类型。在这个例子中，我们告诉它，所有特性列都是浮点，缺失值应该默认值为零，但我们提供一个空数组tf.float32最后列（目标）：数组告诉Tensor流这列包含浮点数，但没有默认值，如果它遇到一个缺失值将产生一个异常。
3. tf.io.decode_csv() 函数返回一个标量张量列表（每列一个），但我们需要返回一个一维张量数组。所以我们在除了最后一个张量外的所有张量上调用tf.stack()：这将把这些张量堆叠到一个一维数组中。然后我

们对目标值做同样的事情：这使得它成为一个具有单一值的一维张量数组，而不是一个标量张量。tf.io.decode_csv() 函数已完成，因此它返回输入特性和目标。

4. 最后，自定义 **preprocess()** 函数只调用parse_csv_line()函数，通过减去特征均值，然后减以特征标准差来对输入特征进行缩放，并返回一个包含缩放特征和目标的元组。

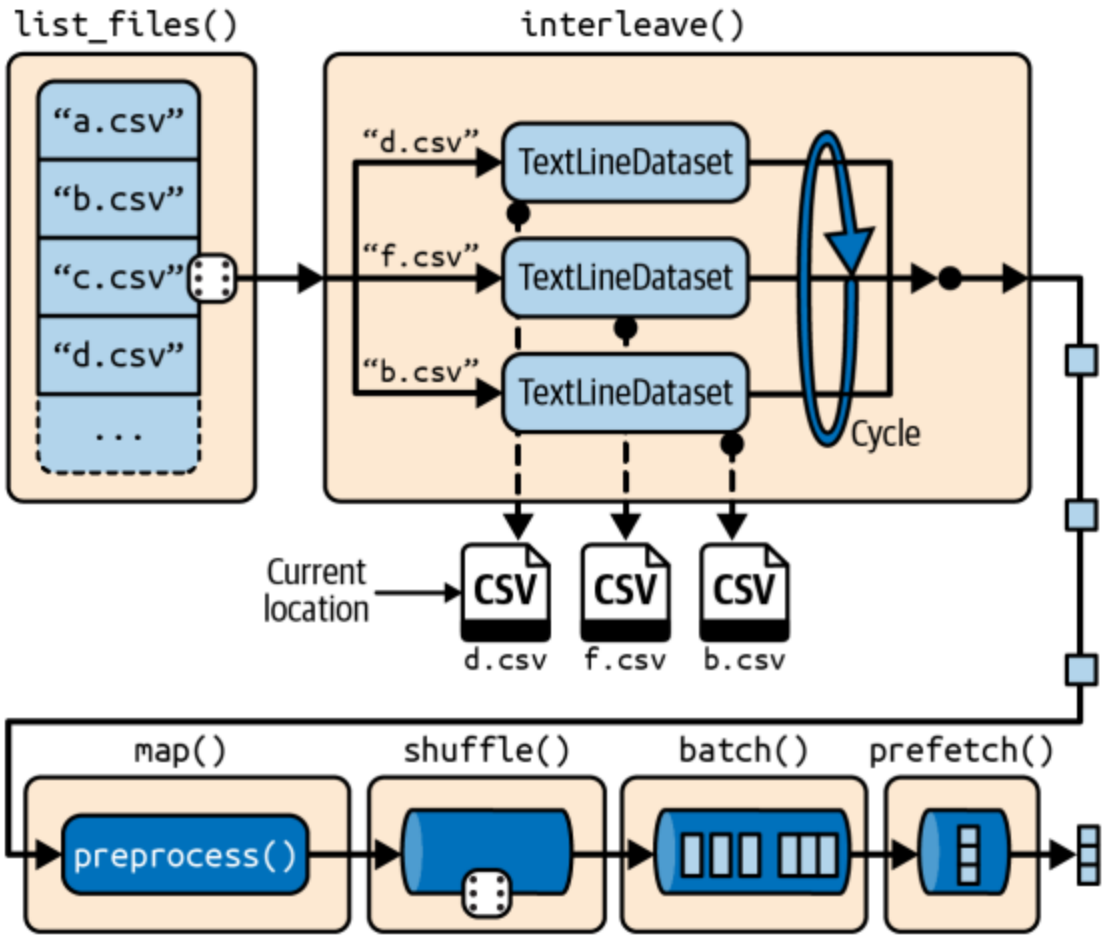
In [31]:
```
preprocess(b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782')
```

Out[31]:
```
(<tf.Tensor: shape=(8,), dtype=float32, numpy=
 array([ 0.16579159,  1.216324  , -0.05204564, -0.39215982, -0.5277444 ,
        -0.2633488 ,  0.8543046 , -1.3072058 ], dtype=float32)>,
 <tf.Tensor: shape=(1,), dtype=float32, numpy=array([2.782], dtype=float32)>)
```

看起来不错！**preprocess()** 函数可以将一个实例从一个字节字符串转换为一个漂亮的缩放张量，并具有相应的标签。我们现在可以使用数据集的map()方法来将 **preprocess()** 函数应用于数据集中的每个样本。

1.5 将所有东西放在一起（Putting Everything Together）

为了使代码更可重用，让我们将我们到目前为止讨论的所有内容放到另一个助手函数中；它将创建并返回一个数据集，该数据集将有效地从多个CSV文件加载加州住房数据，预处理，洗牌，并进行批处理：



In [32]:
```
def csv_reader_dataset(filepaths, n_readers=5, n_read_threads=None,
                       n_parse_threads=5, shuffle_buffer_size=10_000,
                       seed=42, batch_size=32):

    dataset = tf.data.Dataset.list_files(filepaths, seed=seed)

    dataset = dataset.interleave(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
        cycle_length=n_readers, num_parallel_calls=n_read_threads)
```

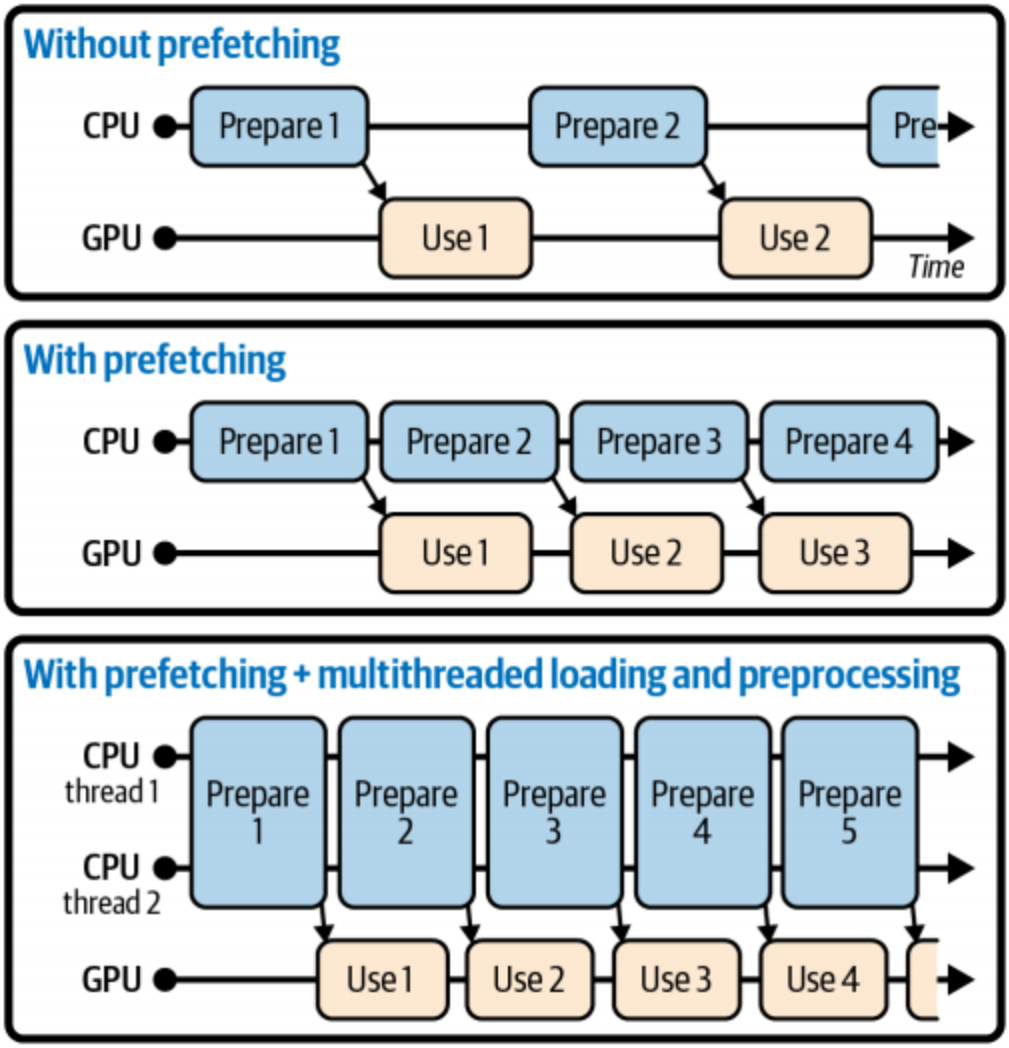
```
    dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)

    dataset = dataset.shuffle(shuffle_buffer_size, seed=seed)

    return dataset.batch(batch_size).prefetch(1)
```

In [33]:
```
# show the first couple of batches produced by the dataset

example_set = csv_reader_dataset(train_filepaths, batch_size=3)

for X_batch, y_batch in example_set.take(2):
    print("X =", X_batch)
    print("y =", y_batch)
    print()
```

```
X = tf.Tensor(
[[-1.3957452  -0.04940685 -0.22830808  0.22648273  2.2593622   0.35200632
   0.9667386  -1.4121602 ]
 [ 2.7112627  -1.0778131   0.69413143 -0.14870553  0.51810503  0.3507294
  -0.82285154  0.80680597]
 [-0.13484643 -1.868895    0.01032507 -0.13787179 -0.12893449  0.03143518
   0.2687057   0.13212144]], shape=(3, 8), dtype=float32)
y = tf.Tensor(
[[1.819]
 [3.674]
 [0.954]], shape=(3, 1), dtype=float32)

X = tf.Tensor(
[[ 0.09031774  0.9789995   0.1327582  -0.13753782 -0.23388447  0.10211545
   0.97610843 -1.4121602 ]
 [ 0.05218809 -2.0271113   0.2940109  -0.02403445  0.16218767 -0.02844518
   1.4117942  -0.93737936]
 [-0.672276    0.02970133 -0.76922584 -0.15086786  0.4962024  -0.02741998
  -0.7853724   0.77182245]], shape=(3, 8), dtype=float32)
y = tf.Tensor(
[[2.725]
 [1.205]
 [1.625]], shape=(3, 1), dtype=float32)
```

1.6 预取（Prefetching）

通过在自定义csv_reader_dataset()函数的末尾调用 **prefetch(1)** ，我们创建了一个数据集，它将尽力始终领先一个批处理。换句话说，当我们的训练算法在一批处理时，数据集已经在并行处理下一批处理（例如，从磁盘读取数据并进行预处理）。这可以显著提高性能，如图



如果我们还确保加载和预处理是多线程（通过设置num_parallel_calls当调用 interleave() 和 map() ），我们可以利用多个CPU核心，并希望使准备一批数据比在GPU上运行一个训练步骤更短：这样GPU将几乎100%利用（除了数据传输时间从CPU到GPU），并且训练将运行得更快。

如果你计划购买一个GPU卡，它的处理能力和内存大小当然是非常重要的（特别是，大量的RAM对于大型计算机视觉或自然语言处理模型是至关重要的）。对于良好的性能，同样重要的是GPU的内存带宽；这是它每秒可以进入或退出RAM的gb数据数。

如果数据集足够小，则可以通过使用数据集的 **cache()** 方法将其内容缓存到RAM来显著加快训练速度。通常应该在加载和预处理数据之后，但在洗牌、重复、批处理和预取之前。这样，每个实例将只被读取和预处理一次（而不是每个epoch一次），但数据仍然将在每个epoch进行不同的打乱，下一批仍然将提前准备。

In [34]:
```
# list all methods of the tf.data.Dataset class
for m in dir(tf.data.Dataset):
    if not (m.startswith("_") or m.endswith("_")):
        func = getattr(tf.data.Dataset, m)
        if hasattr(func, "__doc__"):
            print("● {:21s}{}".format(m + "()", func.__doc__.split("\n")[0]))
```

```
● apply()                Applies a transformation function to this dataset.
● as_numpy_iterator()    Returns an iterator which converts all elements of the dataset to
numpy.
● batch()                Combines consecutive elements of this dataset into batches.
● bucket_by_sequence_length()A transformation that buckets elements in a `Dataset` by le
ngth.
● cache()                Caches the elements in this dataset.
● cardinality()          Returns the cardinality of the dataset, if known.
```

```
• choose_from_datasets()Creates a dataset that deterministically chooses elements from `
datasets`.
• concatenate()        Creates a `Dataset` by concatenating the given dataset with this
dataset.
• element_spec()       The type specification of an element of this dataset.
• enumerate()          Enumerates the elements of this dataset.
• filter()             Filters this dataset according to `predicate`.
• flat_map()           Maps `map_func` across this dataset and flattens the result.
• from_generator()     Creates a `Dataset` whose elements are generated by `generator`.
(deprecated arguments)
• from_tensor_slices() Creates a `Dataset` whose elements are slices of the given tensor
s.
• from_tensors()       Creates a `Dataset` with a single element, comprising the given t
ensors.
• get_single_element() Returns the single element of the `dataset`.
• group_by_window()    Groups windows of elements by key and reduces them.
• interleave()         Maps `map_func` across this dataset, and interleaves the results.
• list_files()         A dataset of all files matching one or more glob patterns.
• load()               Loads a previously saved dataset.
• map()                Maps `map_func` across the elements of this dataset.
• options()            Returns the options for this dataset and its inputs.
• padded_batch()       Combines consecutive elements of this dataset into padded batche
s.
• prefetch()           Creates a `Dataset` that prefetches elements from this dataset.
• random()             Creates a `Dataset` of pseudorandom values.
• range()              Creates a `Dataset` of a step-separated range of values.
• reduce()             Reduces the input dataset to a single element.
• rejection_resample() A transformation that resamples a dataset to a target distributio
n.
• repeat()             Repeats this dataset so each original value is seen `count` time
s.
• sample_from_datasets()Samples elements at random from the datasets in `datasets`.
• save()               Saves the content of the given dataset.
• scan()               A transformation that scans a function across an input dataset.
• shard()              Creates a `Dataset` that includes only 1/`num_shards` of this dat
aset.
• shuffle()            Randomly shuffles the elements of this dataset.
• skip()               Creates a `Dataset` that skips `count` elements from this datase
t.
• snapshot()           API to persist the output of the input dataset.
• take()               Creates a `Dataset` with at most `count` elements from this datas
et.
• take_while()         A transformation that stops dataset iteration based on a `predica
te`.
• unbatch()            Splits elements of a dataset into multiple elements.
• unique()             A transformation that discards duplicate elements of a `Dataset`.
• window()             Returns a dataset of "windows".
• with_options()       Returns a new `tf.data.Dataset` with the given options set.
• zip()                Creates a `Dataset` by zipping together the given datasets.
```

1.7 在Keras中使用数据集（Using the Dataset with Keras）

现在，我们可以使用我们之前编写的自定义 **csv_reader_dataset()** 函数来为训练集、验证集和测试集创建一个数据集。训练集将在每个历元上都被打乱（请注意，验证集和测试集也将被打乱，即使我们并不真的需要这样做）：

In [35]:
```
train_set = csv_reader_dataset(train_filepaths)
valid_set = csv_reader_dataset(valid_filepaths)
test_set = csv_reader_dataset(test_filepaths)
```

现在，您可以简单地使用这些数据集来构建和训练一个Keras模型。当您调用模型的 **fit()** 方法时，您传递 train_set 而不是 X_train，y_train，并传递 **validation_data=validset** 而不是 **validation_data=**

(X_valid,y_valid) 。fit() 方法将处理每个epoch重复一次训练数据集，在每个epoch使用不同的随机顺序：

In [36]:
```
# for reproducibility
tf.keras.backend.clear_session()
tf.random.set_seed(42)
```

In [42]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(30, activation="relu",
                          kernel_initializer="he_normal",
                          input_shape=X_train.shape[1:]),
    tf.keras.layers.Dense(1),
])
model.compile(loss="mse", optimizer="sgd")
model.fit(train_set, validation_data=valid_set, epochs=5)
```

```
Epoch 1/5
363/363 [==============================] - 1s 1ms/step - loss: 0.8034 - val_loss: 36.436
0
Epoch 2/5
363/363 [==============================] - 0s 982us/step - loss: 0.7672 - val_loss: 1.60
25
Epoch 3/5
363/363 [==============================] - 0s 953us/step - loss: 0.4789 - val_loss: 4.46
23
Epoch 4/5
363/363 [==============================] - 0s 919us/step - loss: 0.4380 - val_loss: 0.39
02
Epoch 5/5
363/363 [==============================] - 0s 937us/step - loss: 0.4404 - val_loss: 21.1
081
```

Out[42]:
```
<keras.callbacks.History at 0x275d127bc10>
```

In [38]:
```
test_mse = model.evaluate(test_set)
new_set = test_set.take(3)  # pretend we have 3 new samples
y_pred = model.predict(new_set)  # or you could just pass a NumPy array
```

```
162/162 [==============================] - 0s 652us/step - loss: 0.3926
3/3 [==============================] - 0s 8ms/step
```

与其他集合不同，**new_set** 通常不包含标签。如果是这样，就像这里的情况一样，Keras 就会忽略他们。注意，在所有这些情况下，如果您愿意，您仍然可以使用NumPy数组而不是数据集（当然，它们需要首先被加载和预处理）。

如果你想构建自己的自定义训练循环（如第12章讨论），您可以迭代训练集，非常自然：

In [39]:
```
# defines the optimizer and loss function for training
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
loss_fn = tf.keras.losses.mean_squared_error

n_epochs = 5

for epoch in range(n_epochs):
    for X_batch, y_batch in train_set:
        # extra code – perform one Gradient Descent step
        #              as explained in Chapter 12
        print("\rEpoch {}/{}".format(epoch + 1, n_epochs), end="")
        with tf.GradientTape() as tape:
            y_pred = model(X_batch)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

```
Epoch 5/5
```

事实上，甚至可以创建一个TF函数（见第12章），为整个epoch训练模型。这真的可以加快训练的速度

In [40]:
```
@tf.function
def train_one_epoch(model, optimizer, loss_fn, train_set):
    for X_batch, y_batch in train_set:
        with tf.GradientTape() as tape:
            y_pred = model(X_batch)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))

optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
loss_fn = tf.keras.losses.mean_squared_error
for epoch in range(n_epochs):
    print("\rEpoch {}/{}".format(epoch + 1, n_epochs), end="")
    train_one_epoch(model, optimizer, loss_fn, train_set)
```

```
Epoch 5/5
```

在Keras中，**compile()** 方法的 **steps_per_execution** 参数允许您定义 **fit()** 方法在每次调用tf时将处理的批数。它用于训练。默认值仅为1，所以如果您将其设置为50，您通常会看到显著的性能改进。但是，Keras回调的 **on*batch*()** 方法只每50批调用一次。

如果您满意CSV文件（或您正在使用的任何其他格式），您不必使用 **TFRecords** 。俗话说，如果它没有坏掉，就不要修理它！当训练期间的瓶颈是加载和解析数据时，**TFRecords** 很有用。

2. TFRecord格式（The TFRecord Format）

TFRecord 格式是TensorFlow存储大量数据和有效读取它的首选格式。它是一种非常简单的二进制格式，只包含一个不同大小的二进制记录序列（每个记录由一个长度、一个检查长度是否被损坏的CRC校验和组成，然后是实际数据，最后是数据的CRC校验和）。您可以使用 **tf.io.TFRecordWriter** 类轻松创建TFRecord文件：

In [43]:
```
with tf.io.TFRecordWriter("my_data.tfrecord") as f:
    f.write(b"This is the first record")
    f.write(b"And this is the second record")
```

然后你就可以使用 **tf.data.TFRecordDataset** 读取一个或多个TFRecord文件：

In [44]:
```
filepaths = ["my_data.tfrecord"]
dataset = tf.data.TFRecordDataset(filepaths)

for item in dataset:
    print(item)
```

```
tf.Tensor(b'This is the first record', shape=(), dtype=string)
tf.Tensor(b'And this is the second record', shape=(), dtype=string)
```

默认情况下，**TFRecordDataset** 将一个一个地读取文件，但是您可以通过将文件路径列表传递给构造函数并将 **num_parallel_reads** 设置为大于1的数字，使其并行读取多个文件并交错它们的记录。或者，您也可以通过使用 **list_files()** 和 **interleave()** 来获得与我们之前读取多个CSV文件相同的结果。

In [45]:
```
# shows how to read multiple files in parallel and interleave them

filepaths = ["my_test_{}.tfrecord".format(i) for i in range(5)]

for i, filepath in enumerate(filepaths):
```

```
    with tf.io.TFRecordWriter(filepath) as f:
        for j in range(3):
            f.write("File {} record {}".format(i, j).encode("utf-8"))

dataset = tf.data.TFRecordDataset(filepaths, num_parallel_reads=3)

for item in dataset:
    print(item)
```

```
tf.Tensor(b'File 0 record 0', shape=(), dtype=string)
tf.Tensor(b'File 1 record 0', shape=(), dtype=string)
tf.Tensor(b'File 2 record 0', shape=(), dtype=string)
tf.Tensor(b'File 0 record 1', shape=(), dtype=string)
tf.Tensor(b'File 1 record 1', shape=(), dtype=string)
tf.Tensor(b'File 2 record 1', shape=(), dtype=string)
tf.Tensor(b'File 0 record 2', shape=(), dtype=string)
tf.Tensor(b'File 1 record 2', shape=(), dtype=string)
tf.Tensor(b'File 2 record 2', shape=(), dtype=string)
tf.Tensor(b'File 3 record 0', shape=(), dtype=string)
tf.Tensor(b'File 4 record 0', shape=(), dtype=string)
tf.Tensor(b'File 3 record 1', shape=(), dtype=string)
tf.Tensor(b'File 4 record 1', shape=(), dtype=string)
tf.Tensor(b'File 3 record 2', shape=(), dtype=string)
tf.Tensor(b'File 4 record 2', shape=(), dtype=string)
```

2.1 压缩的TFRecord文件（Compressed TFRecord Files）

压缩TFRecord文件有时会很有用，特别是当它们需要通过网络连接加载时。您可以通过设置选项参数来创建一个压缩的TFRecord文件：

In [49]:
```
options = tf.io.TFRecordOptions(compression_type="GZIP")
with tf.io.TFRecordWriter("my_compressed.tfrecord", options) as f:
    f.write(b"Compress, compress, compress!")
```

在读取压缩的TFRecord文件时，您需要指定压缩类型：

In [50]:
```
dataset = tf.data.TFRecordDataset(["my_compressed.tfrecord"],
                                  compression_type="GZIP")
```

In [51]:
```
# eshows that the data is decompressed correctly
for item in dataset:
    print(item)
```

```
tf.Tensor(b'Compress, compress, compress!', shape=(), dtype=string)
```

2.2 协议缓冲区的简介（A Brief Introduction to Protocol Buffers）

尽管每个记录都可以使用您想要的任何二进制格式，TFRecord文件通常包含序列化的协议缓冲区（也称为**protobufs**）。这是一种可移植的、可扩展的、高效的二进制格式，2001年谷歌开发，2008年开源；protobufs 现在被广泛使用，特别是在谷歌的远程过程调用系统 gRPC 中。它们是使用一种简单的简单语言定义的：

In [4]:
```
%%writefile person.proto
syntax = "proto3";
message Person {
    string name = 1;
    int32 id = 2;
    repeated string email = 3;
}
```

```
Overwriting person.proto
```

这个 **protobuf** 定义说我们使用的是protobuf格式的版本3，它指定每个 **Person** 对象可以（可选地）有字符串类型的 **name** ，一个int32类型的 **id** ，以及0个或多个 **email** 字段，每个都是类型字符串。数字1、2和3是字段标识符：它们将在每个记录的二进制表示中使用。一旦您在一个.proto文件中有了一个定义，您就可以编译它了。这需要原型协议，即原型编译器，以Python（或其他语言）生成访问类。

请注意，通常在TensorFlow中使用的protobuf定义已经为您编译，它们的Python类是TensorFlow库的一部分，因此您不需要使用协议。您所需要知道的是如何在Python中使用protobuf访问类。

为了说明基本原理，让我们看一个简单的例子，它使用为Person protobuf生成的访问类（代码在注释中解释）：

In [5]:
```
!protoc person.proto --python_out=. --descriptor_set_out=person.desc --include_imports
```

In [6]:
```
%ls person*
```

```
 驱动器 C 中的卷是 Windows
 卷的序列号是 7A93-55F0

 C:\Users\tu'tu\Desktop\一些课外内容\机器学习\神经网络与深度学习(Edition 3) 的目录

2023/01/28  14:41                92 person.desc
2023/01/28  14:40               114 person.proto
2023/01/28  14:41               986 person_pb2.py
               3 个文件          1,192 字节
               0 个目录 81,678,643,200 可用字节
```

In [7]:
```
from person_pb2 import Person  # import the generated access class
```

In [8]:
```
person = Person(name="Al", id=123, email=["a@b.com"])  # create a Person
print(person)  # display the Person
```

```
name: "Al"
id: 123
email: "a@b.com"
```

In [9]:
```
person.name  # read a field
```

Out[9]:
```
'Al'
```

In [10]:
```
person.name = "Alice"  # modify a field
```

In [11]:
```
person.email[0]  # repeated fields can be accessed like arrays
```

Out[11]:
```
'a@b.com'
```

In [12]:
```
person.email.append("c@d.com")  # add an email address
```

In [13]:
```
serialized = person.SerializeToString()  # serialize person to a byte string

serialized
```

Out[13]:
```
b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
```

In [14]:
```
person2 = Person()  # create a new Person

person2.ParseFromString(serialized)  # parse the byte string (27 bytes long)
```

Out[14]:
```
27
```

In [15]:
```
person == person2  # now they are equal
```

Out[15]:
```
True
```

简而言之，我们导入由 **protoc** 生成的 **Person** 类，创建一个实例并使用它，可视化它，读写一些字段，然后使用 **SerializeToString()** 方法序列化它。这是准备通过网络保存或传输的二进制数据。当读取或接收这个二进制数据时，我们可以使用 **ParseFromString()** 方法来解析它，并得到一个被序列化的对象的副本。

您可以将序列化的 **Person** 对象保存到一个TFRecord文件中，然后加载和解析它：一切都可以正常工作。然而，**ParseFromString()** 并不是一个 TensorFlow 操作，因此您不能在 **tf.data** pipeline的预处理函数中使用它（除非将它包装在 **tf.py_function()** 操作中，这将使代码更慢，也更不具有可移植性，正如您在第12章中看到的那样）。但是，您可以使用 **tf.io.decode_proto()** 函数，它可以解析您想要的任何 **protobuf** ，前提是您给它提供 **protobuf** 定义（请参见笔记本的示例）。也就是说，在实践中，您通常希望使用 TensorFlow 提供专用解析操作的预定义协议对象。现在让我们看看这些预定义的 **protobuf** 。

In [16]:
```
# shows how to use the tf.io.decode_proto() function

person_tf = tf.io.decode_proto(
    bytes=serialized,
    message_type="Person",
    field_names=["name", "id", "email"],
    output_types=[tf.string, tf.int32, tf.string],
    descriptor_source="person.desc")

person_tf.values
```

Out[16]:
```
[<tf.Tensor: shape=(1,), dtype=string, numpy=array([b'Alice'], dtype=object)>,
 <tf.Tensor: shape=(1,), dtype=int32, numpy=array([123])>,
 <tf.Tensor: shape=(2,), dtype=string, numpy=array([b'a@b.com', b'c@d.com'], dtype=objec
t)>]
```

2.3 TensorFlow Protobufs

TFRecord 文件中通常使用的主要 **protobuf** 是 **Example**，它表示数据集中的一个实例。它包含一个已命名特性的列表，其中每个特性都可以是字节字符串列表、浮点数列表或整数列表。以下是 **protobuf** 的定义（来自TensorFlow的源代码）：

```
syntax = "proto3";
message BytesList { repeated bytes value = 1; }
message FloatList { repeated float value = 1 [packed = true]; }
message Int64List { repeated int64 value = 1 [packed = true]; }
message Feature {
    oneof kind {
        BytesList bytes_list = 1;
        FloatList float_list = 2;
        Int64List int64_list = 3;
    }
};
message Features { map<string, Feature> feature = 1; };
message Example { Features features = 1; };
```

BytesList、**FloatList**和**Int64List**的定义是直接定义的。注意，**[packed = true]** 用于重复的数值域，用于非常有效的编码。**Feature** 包含**BytesList**、**FloatList**或**Int64List**。**Features**（带有s）包含一个字典，它将特征名

称映射到相应的特征值。最后，一个 **Example** 只包含一个 **Features** 对象。

为什么甚至定义了**Example**，因为它只包含一个 **Features** 对象？嗯，TensorFlow的开发者可能有一天会决定添加更多的字段。只要新的 **Example** 定义仍然包含 **features** 字段，具有相同的ID，它将向后兼容。这种可扩展性是 **protobufs** 的伟大特性之一。

下面是你可以创建一个 **tf.train.Example** 的方法。与前面表示同一个 person：

In [18]:
```
from tensorflow.train import BytesList, FloatList, Int64List
from tensorflow.train import Feature, Features, Example

person_example = Example(
    features=Features(
        feature={
            "name": Feature(bytes_list=BytesList(value=[b"Alice"])),
            "id": Feature(int64_list=Int64List(value=[123])),
            "emails": Feature(bytes_list=BytesList(value=[b"a@b.com",
                                                          b"c@d.com"]))
        }))
```

代码有点冗长和重复，但是您可以很容易地将它包装在一个小的辅助函数中。现在我们有了一个 **Example**，我们可以通过调用它的 **SerializeToString()** 方法来序列化它，然后将结果数据写入TFRecord文件。让我们写五次，假装我们有多次关联：

In [19]:
```
with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:
    for _ in range(5):
        f.write(person_example.SerializeToString())
```

通常你会写五个以上的 **Examples**！通常，您将创建一个转换脚本，该脚本从当前格式（例如CSV文件）读取，为每个实例创建一个 **Example** protobuf，将它们序列化，并将它们保存到几个TFRecord文件中，最好在此过程中对它们进行洗牌。这需要一些工作，所以再次确认它是必要的（也许您的管道对CSV文件工作得很好）。

2.4 加载和解析 Example（Loading and Parsing Examples）

要加载序列化的 **Example** protobuf，我们将再次使用 **tf.data.TFRecordDataset**，并用 **tf.io.parse_single_example()** 解析每个 **Example**。它至少需要两个参数：一个包含序列化数据的字符串标量张量，以及对每个特征的描述。

描述是一个字典，它将每个特征名称映射到一个 **tf.io.FixedLenFeature** 描述符，指示特征的形状、类型和默认值，或者一个 **tf.io.VarLenFeature** 描述符，仅指示类型，如果特征列表的长度可能 变化（例如“emails”功能）。

下面的代码定义了一个描述字典，然后创建一个 **TFRecordDataset**，并对其应用一个自定义预处理函数来解析该数据集包含的每个序列化的 **Example** protobuf：

In [20]:
```
feature_description = {
    "name": tf.io.FixedLenFeature([], tf.string, default_value=""),
    "id": tf.io.FixedLenFeature([], tf.int64, default_value=0),
    "emails": tf.io.VarLenFeature(tf.string),
}

def parse(serialized_example):
    return tf.io.parse_single_example(serialized_example, feature_description)
```

```
dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).map(parse)

for parsed_example in dataset:
    print(parsed_example)
```

```
{'emails': <tensorflow.python.framework.sparse_tensor.SparseTensor object at 0x000002738
212D7F0>, 'id': <tf.Tensor: shape=(), dtype=int64, numpy=123>, 'name': <tf.Tensor: shape
=(), dtype=string, numpy=b'Alice'>}
{'emails': <tensorflow.python.framework.sparse_tensor.SparseTensor object at 0x000002738
865C0A0>, 'id': <tf.Tensor: shape=(), dtype=int64, numpy=123>, 'name': <tf.Tensor: shape
=(), dtype=string, numpy=b'Alice'>}
{'emails': <tensorflow.python.framework.sparse_tensor.SparseTensor object at 0x000002738
212D7F0>, 'id': <tf.Tensor: shape=(), dtype=int64, numpy=123>, 'name': <tf.Tensor: shape
=(), dtype=string, numpy=b'Alice'>}
{'emails': <tensorflow.python.framework.sparse_tensor.SparseTensor object at 0x000002738
8666670>, 'id': <tf.Tensor: shape=(), dtype=int64, numpy=123>, 'name': <tf.Tensor: shape
=(), dtype=string, numpy=b'Alice'>}
{'emails': <tensorflow.python.framework.sparse_tensor.SparseTensor object at 0x000002738
8666820>, 'id': <tf.Tensor: shape=(), dtype=int64, numpy=123>, 'name': <tf.Tensor: shape
=(), dtype=string, numpy=b'Alice'>}
```

固定长度的特征被解析为正则张量，而可变长度的特征被解析为稀疏张量。你可以使用 **tf.sparse.to_dense()** 将稀疏张量转换为稠密张量，但在这种情况下，只访问它的值更简单：

In [21]:
```
tf.sparse.to_dense(parsed_example["emails"], default_value=b"")
```

Out[21]:
```
<tf.Tensor: shape=(2,), dtype=string, numpy=array([b'a@b.com', b'c@d.com'], dtype=objec
t)>
```

In [22]:
```
parsed_example["emails"].values
```

Out[22]:
```
<tf.Tensor: shape=(2,), dtype=string, numpy=array([b'a@b.com', b'c@d.com'], dtype=objec
t)>
```

您可能希望使用 **tf.io.parse_example()** 逐批解析示例，而不是使用 **tf.io.parse_single_example()** 逐个解析它们：

In [23]:
```
def parse(serialized_examples):
    return tf.io.parse_example(serialized_examples, feature_description)

dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).batch(2).map(parse)

for parsed_examples in dataset:
    print(parsed_examples)  # two examples at a time
```

```
{'emails': <tensorflow.python.framework.sparse_tensor.SparseTensor object at 0x000002738
A180E80>, 'id': <tf.Tensor: shape=(2,), dtype=int64, numpy=array([123, 123], dtype=int6
4)>, 'name': <tf.Tensor: shape=(2,), dtype=string, numpy=array([b'Alice', b'Alice'], dty
pe=object)>}
{'emails': <tensorflow.python.framework.sparse_tensor.SparseTensor object at 0x000002738
A021C40>, 'id': <tf.Tensor: shape=(2,), dtype=int64, numpy=array([123, 123], dtype=int6
4)>, 'name': <tf.Tensor: shape=(2,), dtype=string, numpy=array([b'Alice', b'Alice'], dty
pe=object)>}
{'emails': <tensorflow.python.framework.sparse_tensor.SparseTensor object at 0x000002738
A04FEE0>, 'id': <tf.Tensor: shape=(1,), dtype=int64, numpy=array([123], dtype=int64)>,
'name': <tf.Tensor: shape=(1,), dtype=string, numpy=array([b'Alice'], dtype=object)>}
```

In [24]:
```
parsed_examples
```

Out[24]:
```
{'emails': <tensorflow.python.framework.sparse_tensor.SparseTensor at 0x2738a04fee0>,
 'id': <tf.Tensor: shape=(1,), dtype=int64, numpy=array([123], dtype=int64)>,
 'name': <tf.Tensor: shape=(1,), dtype=string, numpy=array([b'Alice'], dtype=object)>}
```

最后，**BytesList** 可以包含您需要的任何二进制数据，包括任何序列化的对象。

例如，您可以使用 **tf.io.encode_jpeg()** 使用JPEG格式编码图像，并将此二进制数据放在 **BytesList** 中。之后，当您的代码读取TFRecord时，它将从解析 **Example** 开始，然后它将需要调用 **tf.io.decode_jpeg()** 来解析数据并获得原始映像（或者您可以使用 **tf.io.decode_image()** ，它可以解码任何BMP、GIF、JPEG或PNG映像）。您还可以将使用 **tf.io.serialize_tensor()** 序列化的任何张量存储在字节列表中，然后将生成的字节字符串放在字节列表特性中。

稍后，当您解析TFRecord时，您可以使用 **tf.io.parse_tensor()** 来解析这些数据。

In [28]:
```
import matplotlib.pyplot as plt
from sklearn.datasets import load_sample_images

img = load_sample_images()["images"][0]
plt.imshow(img)
plt.axis("off")
plt.title("Original Image")
plt.show()
```

In [30]:
```
data = tf.io.encode_jpeg(img)

example_with_image = Example(features=Features(feature={
    "image": Feature(bytes_list=BytesList(value=[data.numpy()]))}))

serialized_example = example_with_image.SerializeToString()

with tf.io.TFRecordWriter("my_image.tfrecord") as f:
    f.write(serialized_example)
```

In [32]:
```
feature_description = { "image": tf.io.VarLenFeature(tf.string) }

def parse(serialized_example):
    example_with_image = tf.io.parse_single_example(serialized_example,
                                                    feature_description)
    return tf.io.decode_jpeg(example_with_image["image"].values[0])
    # or you can use tf.io.decode_image() instead

dataset = tf.data.TFRecordDataset("my_image.tfrecord").map(parse)

for image in dataset:
```

```
    plt.imshow(image)
    plt.axis("off")
    plt.show()
```

In [33]:
```
tensor = tf.constant([[0., 1.], [2., 3.], [4., 5.]])

serialized = tf.io.serialize_tensor(tensor)

serialized
```

Out[33]:
```
<tf.Tensor: shape=(), dtype=string, numpy=b'\x08\x01\x12\x08\x12\x02\x08\x03\x12\x02\x08
\x02"\x18\x00\x00\x00\x00\x00\x00\x80?\x00\x00\x00@\x00\x00@@\x00\x00\x80@\x00\x00\xa0
@'>
```

In [34]:
```
tf.io.parse_tensor(serialized, out_type=tf.float32)
```

Out[34]:
```
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[0., 1.],
       [2., 3.],
       [4., 5.]], dtype=float32)>
```

In [35]:
```
sparse_tensor = parsed_example["emails"]
serialized_sparse = tf.io.serialize_sparse(sparse_tensor)
serialized_sparse
```

Out[35]:
```
<tf.Tensor: shape=(3,), dtype=string, numpy=
array([b'\x08\t\x12\x08\x12\x02\x08\x02\x12\x02\x08\x01"\x10\x00\x00\x00\x00\x00\x00\x00
\x00\x01\x00\x00\x00\x00\x00\x00\x00',
       b'\x08\x07\x12\x04\x12\x02\x08\x02"\x10\x07\x07a@b.comc@d.com',
       b'\x08\t\x12\x04\x12\x02\x08\x01"\x08\x02\x00\x00\x00\x00\x00\x00\x00'],
      dtype=object)>
```

In [36]:
```
BytesList(value=serialized_sparse.numpy())
```

Out[36]:
```
value: "\010\t\022\010\022\002\010\002\022\002\010\001\"\020\000\000\000\000\000\000\000
\000\001\000\000\000\000\000\000\000"
value: "\010\007\022\004\022\002\010\002\"\020\007\007a@b.comc@d.com"
value: "\010\t\022\004\022\002\010\001\"\010\002\000\000\000\000\000\000\000"
```

正如您所看到的，**Example** protobuf 非常灵活，因此对于大多数用例，它可能就足够了。但是，在处理列表的列表时，使用它可能有点麻烦。

例如，假设您要对文本文档进行分类。每个文档都可以表示为一个句子列表，其中每个句子都被表示为一个单词列表。也许每个文档都有一个注释列表，其中每个注释都被表示为一个单词列表。可能也有一些上下文数据，如文档的作者、标题和发布日期。TensorFlow 的 **SequenceExample** 就是为这种用例而设计的。

2.5 使用SequenceExample Protobuf来处理列表的列表（Handling Lists of Lists Using the SequenceExample Protobuf）

下面是 **SequenceExample** protobuf的定义：

```
message FeatureList { repeated Feature feature = 1; };
message FeatureLists { map<string, FeatureList> feature_list = 1; };
message SequenceExample {
    Features context = 1;
    FeatureLists feature_lists = 2;
};
```

SequenceExample 包含一个上下文数据的 **Features** 对象和一个包含一个或多个已命名的 **FeatureLists** 对象（例如，一个特性列表名为“**content**”和另一个名为“**comments**”）。每个 **FeatureList** 包含一个 **Feature** 对象列表，每个对象可以是字节字符串列表、64位整数列表或浮点数列表（在本例中，每个 **Feature** 将代表一个句子或注释，可能以单词标识符列表的形式）。

构建 **SequenceExample**、序列化和解析它类似于构建、序列化和解析示例，但是必须使用 **tf.io.parse_single_sequence_example()** 来解析单个 **SequenceExample** 或 **tf.io.parse_sequence_example()** 来解析批处理。

这两个函数都返回一个包含上下文特性（作为字典）和特性列表（也作为字典）的元组。如果特征列表包含不同大小的序列（如前面的示例中），您可能希望使用 **tf.RaggedTensor.from_sparse()** 将它们转换为不规则张量。

In [37]:
```
from tensorflow.train import FeatureList, FeatureLists, SequenceExample

context = Features(feature={
    "author_id": Feature(int64_list=Int64List(value=[123])),
    "title": Feature(bytes_list=BytesList(value=[b"A", b"desert", b"place", b"."])),
    "pub_date": Feature(int64_list=Int64List(value=[1623, 12, 25]))
})

content = [["When", "shall", "we", "three", "meet", "again", "?"],
           ["In", "thunder", ",", "lightning", ",", "or", "in", "rain", "?"]]
comments = [["When", "the", "hurlyburly", "'s", "done", "."],
            ["When", "the", "battle", "'s", "lost", "and", "won", "."]]

def words_to_feature(words):
    return Feature(bytes_list=BytesList(value=[word.encode("utf-8")
                                               for word in words]))

content_features = [words_to_feature(sentence) for sentence in content]
comments_features = [words_to_feature(comment) for comment in comments]

sequence_example = SequenceExample(
    context=context,
    feature_lists=FeatureLists(feature_list={
        "content": FeatureList(feature=content_features),
        "comments": FeatureList(feature=comments_features)
    }))
```

In [38]:
```
sequence_example
```

Out[38]:
```
context {
  feature {
    key: "author_id"
    value {
      int64_list {
        value: 123
      }
    }
  }
  feature {
    key: "pub_date"
    value {
      int64_list {
        value: 1623
        value: 12
        value: 25
      }
    }
  }
  feature {
    key: "title"
    value {
      bytes_list {
        value: "A"
        value: "desert"
        value: "place"
        value: "."
      }
    }
  }
}
feature_lists {
  feature_list {
    key: "comments"
    value {
      feature {
        bytes_list {
          value: "When"
          value: "the"
          value: "hurlyburly"
          value: "\'s"
          value: "done"
          value: "."
        }
      }
      feature {
        bytes_list {
          value: "When"
          value: "the"
          value: "battle"
          value: "\'s"
          value: "lost"
          value: "and"
          value: "won"
          value: "."
        }
      }
    }
  }
  feature_list {
    key: "content"
    value {
      feature {
        bytes_list {
```

```
            value: "When"
            value: "shall"
            value: "we"
            value: "three"
            value: "meet"
            value: "again"
            value: "?"
          }
        }
        feature {
          bytes_list {
            value: "In"
            value: "thunder"
            value: ","
            value: "lightning"
            value: ","
            value: "or"
            value: "in"
            value: "rain"
            value: "?"
          }
        }
      }
    }
}
```

In [39]:
```
serialized_sequence_example = sequence_example.SerializeToString()
```

In [40]:
```
context_feature_descriptions = {
    "author_id": tf.io.FixedLenFeature([], tf.int64, default_value=0),
    "title": tf.io.VarLenFeature(tf.string),
    "pub_date": tf.io.FixedLenFeature([3], tf.int64, default_value=[0, 0, 0]),
}
sequence_feature_descriptions = {
    "content": tf.io.VarLenFeature(tf.string),
    "comments": tf.io.VarLenFeature(tf.string),
}
```

In [41]:
```
parsed_context, parsed_feature_lists = tf.io.parse_single_sequence_example(
    serialized_sequence_example, context_feature_descriptions,
    sequence_feature_descriptions)
parsed_content = tf.RaggedTensor.from_sparse(parsed_feature_lists["content"])
```

In [42]:
```
parsed_context
```

Out[42]:
```
{'title': <tensorflow.python.framework.sparse_tensor.SparseTensor at 0x2738b220940>,
 'author_id': <tf.Tensor: shape=(), dtype=int64, numpy=123>,
 'pub_date': <tf.Tensor: shape=(3,), dtype=int64, numpy=array([1623,   12,   25], dtype=
int64)>}
```

In [43]:
```
parsed_context["title"].values
```

Out[43]:
```
<tf.Tensor: shape=(4,), dtype=string, numpy=array([b'A', b'desert', b'place', b'.'], dty
pe=object)>
```

In [44]:
```
parsed_feature_lists
```

Out[44]:
```
{'comments': <tensorflow.python.framework.sparse_tensor.SparseTensor at 0x2738b220610>,
 'content': <tensorflow.python.framework.sparse_tensor.SparseTensor at 0x2738b220100>}
```

In [45]:
```
print(tf.RaggedTensor.from_sparse(parsed_feature_lists["content"]))
```

```
<tf.RaggedTensor [[b'When', b'shall', b'we', b'three', b'meet', b'again', b'?'],
 [b'In', b'thunder', b',', b'lightning', b',', b'or', b'in', b'rain', b'?']]>
```

3. Keras预处理层（Keras Preprocessing Layers）

为神经网络准备数据通常需要对数字特征进行规范化，对分类特征和文本进行编码，裁剪和调整图像大小等等。这里有以下几种选择：

1. 在准备训练数据文件时，预处理可以提前做好，可以使用您喜欢的任何工具，如NumPy、Pandas或Scikit-Learn。你会需要在生产中应用完全相同的预处理步骤，以确保您的生产模型接收与训练的类似的预处理输入。
2. 或者，您可以在使用 **tf.data** 加载数据的同时对数据进行预处理，通过使用该数据集的 **map()** 方法对数据集的每个元素应用预处理函数。同样，您将需要在生产过程中应用相同的预处理步骤。
3. 最后一种方法是在模型中直接包含预处理层，这样它就可以在训练期间预处理所有的输入数据，然后在生产中使用相同的预处理层。

本章的其余部分将介绍这最后一种方法。

Keras提供了许多预处理层，您可以包含在模型中：它们可以应用于数字特征、分类特征、图像和文本。

我们将在下一节中介绍数字和分类特征，以及基本的文本预处理，我们将在第14章介绍图像预处理，在第16章介绍更高级的文本预处理。

3.1 归一化层（The Normalization Layer）

正如我们在第10章中所看到的，Keras提供了一个 **Normalization** 层，我们可以使用它来标准化输入特征。我们可以在创建层时指定每个特征的均值和方差，或者更简单地说，在拟合模型之前将训练集传递给层的 **adapt()** 方法，这样层可以在训练前自己测量特征的均值和方差：

In [47]:
```
# extra code – fetches, splits and normalizes the California housing dataset

from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target.reshape(-1, 1), random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)
```

In [48]:
```
tf.random.set_seed(42)  # extra code – ensures reproducibility

norm_layer = tf.keras.layers.Normalization()

model = tf.keras.models.Sequential([
    norm_layer,
    tf.keras.layers.Dense(1)
])

model.compile(loss="mse",
              optimizer=tf.keras.optimizers.SGD(learning_rate=2e-3))

# computes the mean and variance of every feature
norm_layer.adapt(X_train)

model.fit(X_train, y_train,
          validation_data=(X_valid, y_valid),
          epochs=5)
```

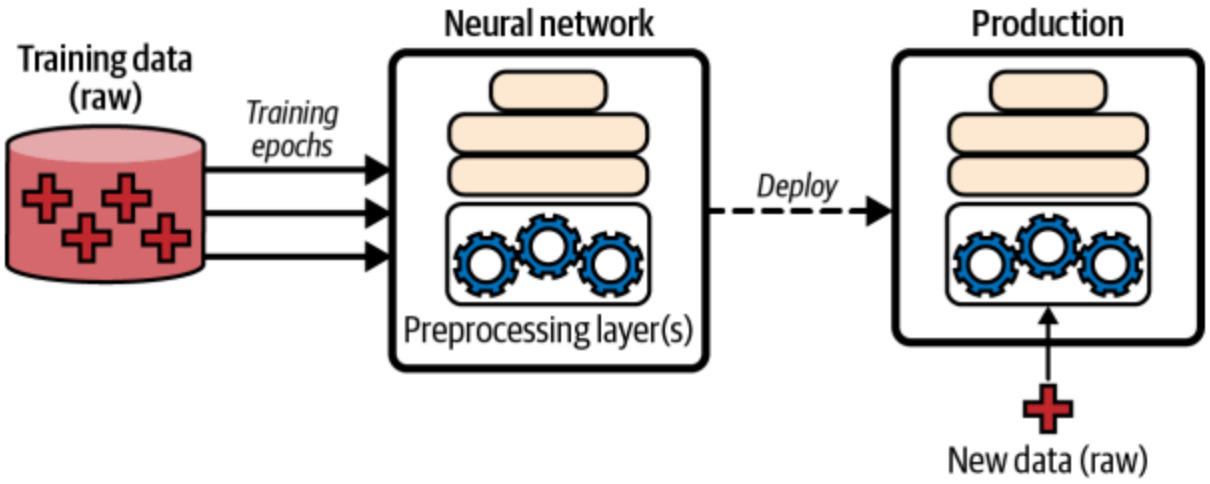
```
Epoch 1/5
```

```
363/363 [==============================] - 1s 1ms/step - loss: 2.6423 - val_loss: 1.0434
Epoch 2/5
363/363 [==============================] - 0s 795us/step - loss: 0.8810 - val_loss: 2.21
39
Epoch 3/5
363/363 [==============================] - 0s 867us/step - loss: 0.7648 - val_loss: 1.39
60
Epoch 4/5
363/363 [==============================] - 0s 813us/step - loss: 0.7127 - val_loss: 0.95
83
Epoch 5/5
363/363 [==============================] - 0s 849us/step - loss: 0.6780 - val_loss: 0.65
03
```

Out[48]:
```
<keras.callbacks.History at 0x2738d9f9df0>
```

传递给 **adapt()** 方法的数据样本必须足够大以代表你的数据集，但它不需要完整的训练集：Normalization 层，几百实例随机抽样训练集通常足以得到一个好的估计的特征均值和方差。

由于我们在模型中包含了 **Normalization** 层，所以我们现在可以将这个模型部署到生产中，而不必再次担心规范化：模型将可以处理它。



在模型中直接包含预处理层是很好和简单的，但它会降低训练速度（在归一化层的情况下只是非常轻微）：实际上，由于预处理是在训练过程中动态执行的，所以每个 epoch 发生一次。

我们可以在训练前对整个训练集进行一次标准化，这样可以做得更好。为了做到这一点，我们可以以一种独立的方式使用标准化层（很像一个 Scikit-Learn StandardScaler）：

In [50]:
```
norm_layer = tf.keras.layers.Normalization()
norm_layer.adapt(X_train)
X_train_scaled = norm_layer(X_train)
X_valid_scaled = norm_layer(X_valid)
```

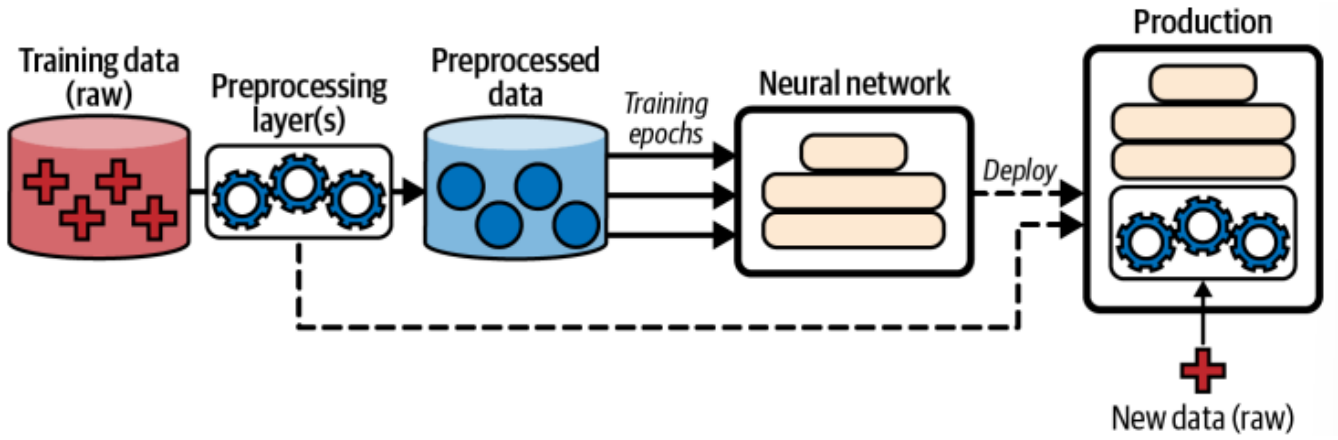
现在我们可以在缩放的数据上训练一个模型，这次没有规范化层：

In [52]:
```
tf.random.set_seed(42)  # extra code – ensures reproducibility

model = tf.keras.models.Sequential([tf.keras.layers.Dense(1)])

model.compile(loss="mse",
              optimizer=tf.keras.optimizers.SGD(learning_rate=2e-3))

model.fit(X_train_scaled, y_train,
          epochs=5,
          validation_data=(X_valid_scaled, y_valid))
```

```
Epoch 1/5
363/363 [==============================] - 1s 1ms/step - loss: 2.8950 - val_loss: 1.0112
Epoch 2/5
363/363 [==============================] - 0s 772us/step - loss: 0.7619 - val_loss: 1.54
94
Epoch 3/5
363/363 [==============================] - 0s 835us/step - loss: 0.6421 - val_loss: 1.17
36
Epoch 4/5
363/363 [==============================] - 0s 767us/step - loss: 0.6136 - val_loss: 0.81
75
Epoch 5/5
363/363 [==============================] - 0s 815us/step - loss: 0.5967 - val_loss: 0.56
40
```

Out[52]:
```
<keras.callbacks.History at 0x2738b8eeb80>
```

这应该会加快训练速度。但是现在，当我们将输入部署到生产中时，模型不会对其进行预处理。为了解决这个问题，我们只需要创建一个新的模型，它同时包装了已适应的规范化层和我们刚刚训练的模型。然后，我们可以将这个最终的模型部署到生产中，它将负责预处理其输入和进行预测：



In [54]:
```
final_model = tf.keras.Sequential([norm_layer, model])

X_new = X_test[:3]  # pretend we have a few new instances (unscaled)

y_pred = final_model(X_new)  # preprocesses the data and makes predictions
```

In [55]:
```
y_pred
```

Out[55]:
```
<tf.Tensor: shape=(3, 1), dtype=float32, numpy=
array([[0.92027915],
       [1.653259  ],
       [2.3451328 ]], dtype=float32)>
```

现在我们有了最好的选择：训练是快速的，因为我们只在训练开始前对数据进行一次预处理，而最终的模型可以动态地预处理其输入，而不会有任何预处理不匹配的风险。

此外，Keras 预处理层与 **tf.data** API 配合得很好。

例如，可以将 **tf.data.Dataset** 传递给预处理层的 **adapt()** 方法。 也可以使用数据集的 **map()** 方法将 Keras 预处理层应用于 **tf.data.Dataset** 。

例如，以下是如何将经过调整的归一化层应用于数据集中每个批次的输入特征：

In [56]:
```
# \creates a dataset to demo applying the norm_layer using map()
dataset = tf.data.Dataset.from_tensor_slices((X_train, y_train)).batch(5)
```

```
dataset = dataset.map(lambda X, y: (norm_layer(X), y))
```

最后，如果您需要比Keras预处理层提供的更多的功能，那么您可以始终编写自己的Keras层，就像我们在第12章中讨论的那样。

例如，如果规范化层不存在，则可以使用以下自定义层获得类似的结果：

In [59]:
```
import numpy as np

class MyNormalization(tf.keras.layers.Layer):
    def adapt(self, X):
        self.mean_ = np.mean(X, axis=0, keepdims=True)
        self.std_ = np.std(X, axis=0, keepdims=True)

    def call(self, inputs):
        eps = tf.keras.backend.epsilon()  # a small smoothing term
        return (inputs - self.mean_) / (self.std_ + eps)
```

In [60]:
```
my_norm_layer = MyNormalization()

my_norm_layer.adapt(X_train)

X_train_scaled = my_norm_layer(X_train)
```

接下来，让我们看看另一个Keras预处理层：离散层（Discretization layer）。

3.2 离散层（The Discretization Layer）

离散化层的目标是通过将值范围（称为**bin**）映射到类别，将数值特征转换为分类特征。这有时对于具有多模态分布的特征或与目标具有高度非线性关系的特征很有用。

例如，下面的代码将一个数字年龄特征映射到三个类别，分别小于18岁、18岁到50岁（不包括在内）和50岁或以上：

In [61]:
```
age = tf.constant([[10.], [93.], [57.], [18.], [37.], [5.]])

discretize_layer = tf.keras.layers.Discretization(bin_boundaries=[18., 50.])

age_categories = discretize_layer(age)

age_categories
```

Out[61]:
```
<tf.Tensor: shape=(6, 1), dtype=int64, numpy=
array([[0],
       [2],
       [2],
       [1],
       [1],
       [0]], dtype=int64)>
```

在这个示例中，我们提供了所需的容器边界。如果您愿意，您可以提供您想要的容器数量，然后调用层的**adapt()** 方法，让它根据值百分位数找到合适的箱子边界。

例如，如果我们设置了 **num_bins=3** ，那么bin边界将位于第33和第66个百分位数以下的值处（在本例中，在值10和37处）：

In [62]:
```
discretize_layer = tf.keras.layers.Discretization(num_bins=3)
```

```
discretize_layer.adapt(age)

age_categories = discretize_layer(age)

age_categories
```

Out[62]:
```
<tf.Tensor: shape=(6, 1), dtype=int64, numpy=
array([[1],
       [2],
       [2],
       [1],
       [2],
       [0]], dtype=int64)>
```

诸如此类的类别标识符通常不应该直接传递给神经网络，因为它们的值不能进行有意义的比较。相反，它们应该进行编码，例如使用独热编码。现在让我们来看看如何做到这一点。

3.3 类别编码层（The CategoryEncoding Layer）

当只有几个类别（例如，少于十几个或两个），那么独热编码通常是一个不错的选择（如第2章所述）。为此，Keras提供了类别编码层。

例如，让我们对我们刚刚创建的年龄类别特性进行独热编码：

In [63]:
```
onehot_layer = tf.keras.layers.CategoryEncoding(num_tokens=3)

onehot_layer(age_categories)
```

Out[63]:
```
<tf.Tensor: shape=(6, 3), dtype=float32, numpy=
array([[0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.]], dtype=float32)>
```

如果您尝试一次编码多个分类特征（只有当它们都使用相同的类别时才有意义），**CategoryEncoding** 类默认情况下执行多热编码：输出张量将为任何输入特征中的每个类别包含一个1。例如：

In [64]:
```
two_age_categories = np.array([[1, 0], [2, 2], [2, 0]])
onehot_layer(two_age_categories)
```

Out[64]:
```
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[1., 1., 0.],
       [0., 0., 1.],
       [1., 0., 1.]], dtype=float32)>
```

如果您认为知道每个类别发生的次数是有用的，您可以在创建类别编码层时设置 **output_mode="count"** ，在这种情况下，输出张量将包含每个类别出现的次数。在前面的示例中，输出将完全相同，除了第二行，它将成为 **[0.，0.，2.]** 。

In [65]:
```
onehot_layer = tf.keras.layers.CategoryEncoding(num_tokens=3, output_mode="count")

onehot_layer(two_age_categories)
```

Out[65]:
```
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[1., 1., 0.],
       [0., 0., 2.],
       [1., 0., 1.]], dtype=float32)>
```

请注意，多热编码和计数编码都会丢失信息，因为不可能知道每个活动类别来自哪个特性。

例如，[0,1]和[1,0]都被编码为[1.,1.,0.]。如果您想避免这种情况，那么您需要对每个特性分别进行独热编码，并连接输出。

In [66]:
```
onehot_layer = tf.keras.layers.CategoryEncoding(num_tokens=3 + 3)

onehot_layer(two_age_categories + [0, 3])  # adds 3 to the second feature
```

Out[66]:
```
<tf.Tensor: shape=(3, 6), dtype=float32, numpy=
array([[0., 1., 0., 1., 0., 0.],
       [0., 0., 1., 0., 0., 1.],
       [0., 0., 1., 1., 0., 0.]], dtype=float32)>
```

在这个输出中，前三列对应于第一个特征，最后三列对应于第二个特征。这使得模型能够区分这两个特征。然而，它也增加了输入给模型的特征的数量，因此需要更多的模型参数。很难预先知道是单个多热编码还是每个特性的独热编码效果最好：它取决于任务，而且您可能需要测试这两个选项。

In [67]:
```
# shows another way to one-hot encode each feature separately
onehot_layer = tf.keras.layers.CategoryEncoding(num_tokens=3,
                                                output_mode="one_hot")
tf.keras.layers.concatenate([onehot_layer(cat)
                             for cat in tf.transpose(two_age_categories)])
```

Out[67]:
```
<tf.Tensor: shape=(3, 6), dtype=float32, numpy=
array([[0., 1., 0., 1., 0., 0.],
       [0., 0., 1., 0., 0., 1.],
       [0., 0., 1., 1., 0., 0.]], dtype=float32)>
```

In [68]:
```
# shows another way to do this, using tf.one_hot() and Flatten
tf.keras.layers.Flatten()(tf.one_hot(two_age_categories, depth=3))
```

Out[68]:
```
<tf.Tensor: shape=(3, 6), dtype=float32, numpy=
array([[0., 1., 0., 1., 0., 0.],
       [0., 0., 1., 0., 0., 1.],
       [0., 0., 1., 1., 0., 0.]], dtype=float32)>
```

现在您可以使用独热或多热来编码分类整数特性。但是，分类文本功能又如何呢？为此，您可以使用串查找（StringLookup）层。

3.4 字符串查找层（The StringLookup Layer）

让我们使用Keras字符串查找层来独热编码城市特征：

In [69]:
```
cities = ["Auckland", "Paris", "Paris", "San Francisco"]

str_lookup_layer = tf.keras.layers.StringLookup()

str_lookup_layer.adapt(cities)

str_lookup_layer([["Paris"], ["Auckland"], ["Auckland"], ["Montreal"]])
```

Out[69]:
```
<tf.Tensor: shape=(4, 1), dtype=int64, numpy=
array([[1],
       [3],
       [3],
       [0]], dtype=int64)>
```

我们首先创建一个字符串查找层，然后我们使它适应数据：它发现有三个不同的类别。然后，我们使用该层对一些城市进行编码。默认情况下，它们被编码为整数。未知类别被映射到0，就像本例中的“Montreal”一样。已知的类别从1开始编号，从最常见的类别到最不常见的类别。

方便的是，如果您在创建字符串查找层时设置了 **output_mode="one_hot"** ，它将为每个类别输出一个热向

量，而不是一个整数：

In [71]:
```
str_lookup_layer = tf.keras.layers.StringLookup(output_mode="one_hot")

str_lookup_layer.adapt(cities)

str_lookup_layer([["Paris"], ["Auckland"], ["Auckland"], ["Montreal"]])
```

```
WARNING:tensorflow:5 out of the last 367 calls to <function PreprocessingLayer.make_adapt_function.<locals>.adapt_step at 0x000002738FEE0280> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for  more details.
```

Out[71]:
```
<tf.Tensor: shape=(4, 4), dtype=float32, numpy=
array([[0., 1., 0., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [1., 0., 0., 0.]], dtype=float32)>
```

如果训练集非常大，则可以方便地将该层仅调整为训练集的一个随机子集。在这种情况下，层的 **adapt()** 方法可能会错过一些较罕见的类别。默认情况下，它会将它们全部映射到类别0，使它们被模型无法区分。

为了减少这种风险（同时仍然只在训练集的一个子集上调整该图层），您可以将 **num_oov_indices** 设置为一个大于1的整数。这是要使用的词汇表外（**out-of-vocabulary，OOV**）桶的数量：使用哈希函数调制OOV桶的数量，每个未知类别将伪随机地映射到其中一个OOV桶。这将允许该模型至少区分一些罕见的类别。例如：

In [72]:
```
str_lookup_layer = tf.keras.layers.StringLookup(num_oov_indices=5)

str_lookup_layer.adapt(cities)

str_lookup_layer([["Paris"], ["Auckland"], ["Foo"], ["Bar"], ["Baz"]])
```

```
WARNING:tensorflow:6 out of the last 368 calls to <function PreprocessingLayer.make_adapt_function.<locals>.adapt_step at 0x000002738D744DC0> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for  more details.
```

Out[72]:
```
<tf.Tensor: shape=(5, 1), dtype=int64, numpy=
array([[5],
       [7],
       [4],
       [3],
       [4]], dtype=int64)>
```

由于有五个OOV桶，第一个已知的类别的ID现在是5（“Paris”）。但是“Foo”、“Bar”和“Baz”是未知的，所以它们都被映射到一个OOV桶中。“Bar”有自己的专用桶（ID为3），但遗憾的是，“Foo”和“Baz”碰巧被映射到同一个桶（ID为4），所以它们在模型中仍然无法区分。这被称为哈希碰撞。减少碰撞风险的唯一方法是增加OOV桶的数量。然而，这也将增加类别的总数，一旦类别是一个热编码，这将需要更多的RAM和额外的模型参数。所以，不要增加这个数字太多。

这种将类别伪随机地映射到桶中的想法被称为**哈希技巧**。Keras提供了一个专门的图层：哈希（Hashing）层。

3.5 哈希层（The Hashing Layer）

对于每个类别，Keras哈希层计算一个散列，调制桶（或“箱子”）的数量。映射完全是伪随机的，但在运行和平台之间是稳定的（即，只要箱子的数量不变，相同的类别总是映射到相同的整数）。

例如，让我们使用哈希层对一些城市进行编码：

In [73]:
```
hashing_layer = tf.keras.layers.Hashing(num_bins=10)

hashing_layer([["Paris"], ["Tokyo"], ["Auckland"], ["Montreal"]])
```

Out[73]:
```
<tf.Tensor: shape=(4, 1), dtype=int64, numpy=
array([[0],
       [1],
       [9],
       [1]], dtype=int64)>
```

这一层的好处是它根本不需要进行调整，这有时可能很有用，特别是在核心外设置中（当数据集太大而无法容纳内存时）。然而，我们再次遇到了一个哈希碰撞：“Tokyo”和“Montreal”被映射到相同的ID，使它们被模型无法区分。所以，通常最好坚持使用StringLookup层。

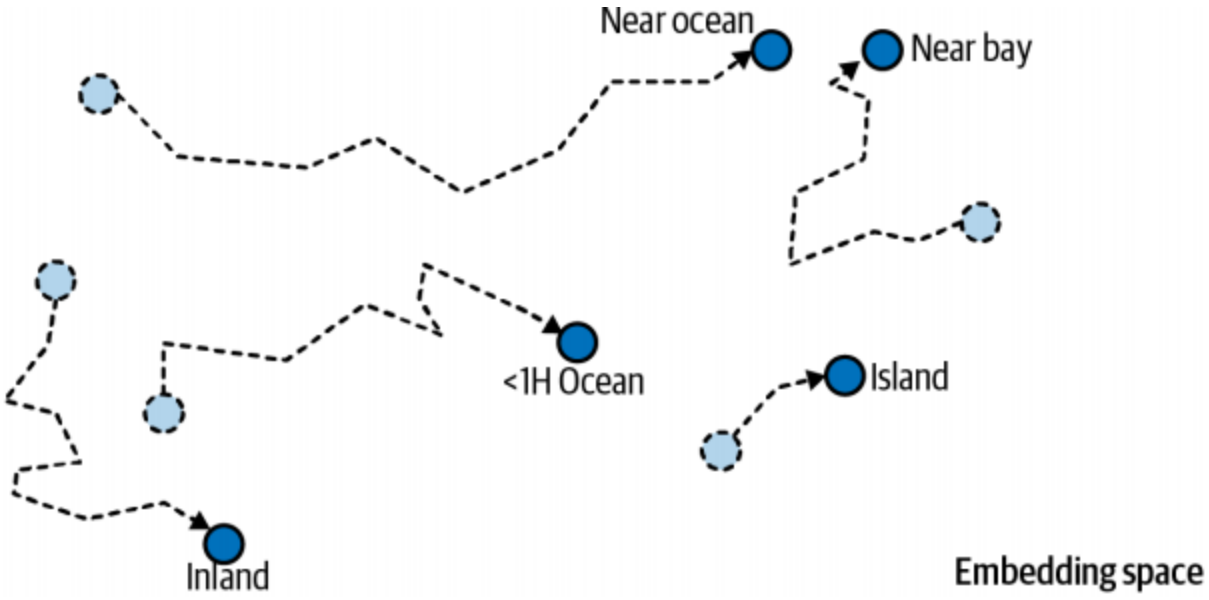
现在让我们来看看另一种编码类别的方法：可训练的嵌入（trainable embeddings）。

3.6 使用嵌入来编码分类特征

嵌入是对一些高维数据的密集表示，例如词汇表中的类别或单词。如果有50,000个可能的类别，那么一个热编码将产生一个50,000维的稀疏向量（即，主要包含零）。相比之下，嵌入将是一个相对较小的密集向量；例如，只有100维。

在深度学习中，嵌入通常是随机初始化的，然后通过梯度下降和其他模型参数进行训练。例如，加州住房数据集中的“NEARBAY”类别最初可以用一个随机向量表示，如[0.131,0.890]，而“NEAR OCEAN”类别可能用另一个随机向量表示，如[0.631,0.791]。在本例中，我们使用二维嵌入，但是维数是一个你可以调整的超参数。

由于这些嵌入是可训练的，它们将在训练中逐渐改善；在这种情况下，由于它们代表了相当相似的类别，梯度下降最终肯定会推动它们更紧密在一起，而它会使它们远离“INLAND”类别的嵌入。事实上，表示得越好，神经网络就越容易做出准确的预测，所以训练往往会使嵌入成为有用的数据表示。这被称为表示学习（您将在第17章中看到其他类型的表示学习）。



Keras提供了一个嵌入层，它包装了一个嵌入矩阵：这个矩阵每个类别有一行，每个嵌入维数有一列。默认情况下，它是随机初始化的。要将类别ID转换为嵌入，嵌入层只需向上查找并返回与该类别对应的行。

例如，让我们初始化一个包含五行和二维嵌入的嵌入层，并使用它来编码一些类别：

In [74]:
```
tf.random.set_seed(42)

embedding_layer = tf.keras.layers.Embedding(input_dim=5, output_dim=2)

embedding_layer(np.array([2, 4, 2]))
```

Out[74]:
```
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[ 0.04608688, -0.01342333],
       [-0.03691798, -0.02986037],
       [ 0.04608688, -0.01342333]], dtype=float32)>
```

由于该层还没有被训练，所以这些编码只是随机的。嵌入层是随机初始化的，因此在模型之外使用它作为独立的预处理层是没有意义的，除非您使用预先训练的权重来初始化它。

如果您想嵌入一个分类文本属性，您可以简单地链接一个字符串查找层和一个嵌入层，如下所示：

In [75]:
```
tf.random.set_seed(42)

ocean_prox = ["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]

str_lookup_layer = tf.keras.layers.StringLookup()

str_lookup_layer.adapt(ocean_prox)

lookup_and_embed = tf.keras.Sequential([
    tf.keras.layers.InputLayer(input_shape=[], dtype=tf.string),  # WORKAROUND
    str_lookup_layer,
    tf.keras.layers.Embedding(input_dim=str_lookup_layer.vocabulary_size(),
                              output_dim=2)
])

lookup_and_embed(np.array(["<1H OCEAN", "ISLAND", "<1H OCEAN"]))
```

Out[75]:
```
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[-0.04670948, -0.0068708 ],
       [-0.02286313,  0.01121503],
       [-0.04670948, -0.0068708 ]], dtype=float32)>
```

警告：Keras 2.8.0（问题 #16101）中存在一个错误，它阻止使用 StringLookup 层作为顺序模型的第一层。幸运的是，有一个简单的解决方法：只需添加一个 InputLayer 作为第一层。

请注意，嵌入矩阵中的行数需要等于词汇量大小：这是类别的总数，包括已知的类别和OOV桶（默认情况下只有一个）。字符串查找类的 **vocabulary_size()** 方法方便地返回这个数字。

在这个示例中，我们使用了二维嵌入，但根据经验法则，嵌入通常有10到300个维度，这取决于任务、词汇表大小和训练集的大小。您将必须调优这个超参数。

把所有的东西放在一起，我们现在可以创建一个Keras模型，它可以处理一个分类的文本特征和常规的数字特征，并学习对每个类别（以及每个OOV桶）的嵌入：

In [79]:
```
# set seeds and generates fake random data
# (feel free to load the real dataset if you prefer)
tf.random.set_seed(42)
np.random.seed(42)
```

```
X_train_num = np.random.rand(10_000, 8)
X_train_cat = np.random.choice(ocean_prox, size=10_000)
y_train = np.random.rand(10_000, 1)

X_valid_num = np.random.rand(2_000, 8)
X_valid_cat = np.random.choice(ocean_prox, size=2_000)
y_valid = np.random.rand(2_000, 1)

num_input = tf.keras.layers.Input(shape=[8], name="num")
cat_input = tf.keras.layers.Input(shape=[], dtype=tf.string, name="cat")

cat_embeddings = lookup_and_embed(cat_input)

encoded_inputs = tf.keras.layers.concatenate([num_input, cat_embeddings])

outputs = tf.keras.layers.Dense(1)(encoded_inputs)

model = tf.keras.models.Model(inputs=[num_input, cat_input],
                              outputs=[outputs])

model.compile(loss="mse", optimizer="sgd")

history = model.fit((X_train_num, X_train_cat), y_train,
                    epochs=5,
                    validation_data=((X_valid_num, X_valid_cat), y_valid))
```

```
Epoch 1/5
313/313 [==============================] - 1s 1ms/step - loss: 0.1360 - val_loss: 0.0933
Epoch 2/5
313/313 [==============================] - 0s 854us/step - loss: 0.0891 - val_loss: 0.08
70
Epoch 3/5
313/313 [==============================] - 0s 930us/step - loss: 0.0851 - val_loss: 0.08
44
Epoch 4/5
313/313 [==============================] - 0s 856us/step - loss: 0.0837 - val_loss: 0.08
35
Epoch 5/5
313/313 [==============================] - 0s 870us/step - loss: 0.0831 - val_loss: 0.08
31
```

这个模型需要两个输入： **num_input**，每个实例包含8个数字特征，再加上**cat_input**，每个实例包含一个分类文本输入。

该模型使用我们之前创建的 **lookup_and_embed** 模型，将每个海洋邻近类别编码为相应的可训练嵌入。接下来，它使用 **concatenate()** 函数连接数字输入和嵌入，以产生完整的编码输入，这些输入准备被提供给神经网络。此时我们可以添加任何类型的神经网络，但为了简单起见，我们只添加一个密集的输出层，然后我们用我们刚刚定义的输入和输出创建Keras模型。接下来，我们编译模型并训练它，同时传递数值输入和分类输入。

正如您在第10章中看到的，由于输入层被命名为“**num**”和“**cat**”，因此我们还应该使用字典而不是元组将训练数据传递给 **fit()** 方法： **{"num": X_train_num, "cat": X_train_cat}** 。或者，我们也可以传递一个包含批的 **tf.data.Dataset** ，每个批分别表示为 **((X_batch_num, X_batch_cat), y_batch)** 或 **({"num": X_batch_num, "cat": X_batch_cat}, y_batch)** 。当然，验证数据也是如此。

In [92]:
```
# shows that the model can also be trained using a tf.data.Dataset
train_set = tf.data.Dataset.from_tensor_slices(
    ((X_train_num, X_train_cat), y_train)).batch(32)

valid_set = tf.data.Dataset.from_tensor_slices(
    ((X_valid_num, X_valid_cat), y_valid)).batch(32)
```

```
history = model.fit(train_set, epochs=5,
                    validation_data=valid_set)
```

```
Epoch 1/5
313/313 [==============================] - 0s 1ms/step - loss: 0.0829 - val_loss: 0.0830
Epoch 2/5
313/313 [==============================] - 0s 879us/step - loss: 0.0828 - val_loss: 0.08
29
Epoch 3/5
313/313 [==============================] - 0s 864us/step - loss: 0.0828 - val_loss: 0.08
29
Epoch 4/5
313/313 [==============================] - 0s 881us/step - loss: 0.0828 - val_loss: 0.08
29
Epoch 5/5
313/313 [==============================] - 0s 940us/step - loss: 0.0828 - val_loss: 0.08
29
```

In [93]:
```
# shows that the dataset can contain dictionaries
train_set = tf.data.Dataset.from_tensor_slices(
    ({"num": X_train_num, "cat": X_train_cat}, y_train)).batch(32)

valid_set = tf.data.Dataset.from_tensor_slices(
    ({"num": X_valid_num, "cat": X_valid_cat}, y_valid)).batch(32)

history = model.fit(train_set, epochs=5, validation_data=valid_set)
```

```
Epoch 1/5
313/313 [==============================] - 0s 1ms/step - loss: 0.0828 - val_loss: 0.0829
Epoch 2/5
313/313 [==============================] - 0s 830us/step - loss: 0.0828 - val_loss: 0.08
29
Epoch 3/5
313/313 [==============================] - 0s 855us/step - loss: 0.0828 - val_loss: 0.08
28
Epoch 4/5
313/313 [==============================] - 0s 965us/step - loss: 0.0828 - val_loss: 0.08
28
Epoch 5/5
313/313 [==============================] - 0s 870us/step - loss: 0.0828 - val_loss: 0.08
28
```

独热编码接着一个密集层（没有激活功能，没有偏差）相当于一个嵌入层。然而，嵌入层使用更少的计算，因为它避免了多次乘零——当嵌入矩阵的大小增长时，性能差异变得明显。密集层的权值矩阵起着嵌入矩阵的作用。

例如，使用大小为20的独热向量和大小为10单位的密集层相当于使用包含 **input_dim=20** 和 **output_dim=10** 的嵌入层。

因此，使用比嵌入层之后的层中的单元数更多的嵌入维度将是一种浪费。

好吧，现在您已经学会了如何编码分类特性，那么是时候将注意力转向文本预处理了。

3.7 文本预处理（Text Preprocessing）

Keras为基本的文本预处理提供了一个文本向量化层。就像字符串查找层一样，您必须在创建时向它传递一个词汇表，或者让它使用 **adapt()** 方法从一些训练数据中学习词汇表。让我们来看一个例子：

In [94]:
```
train_data = ["To be", "!(to be)", "That's the question", "Be, be, be."]

text_vec_layer = tf.keras.layers.TextVectorization()
```

```
text_vec_layer.adapt(train_data)

text_vec_layer(["Be good!", "Question: be or be?"])
```

Out[94]:
```
<tf.Tensor: shape=(2, 4), dtype=int64, numpy=
array([[2, 1, 0, 0],
       [6, 2, 1, 2]], dtype=int64)>
```

这两句话“Be good!”以及“Question: be or be?”分别编码为 [2,1,0,0] 和 [6,2,1,2] 。词汇是从训练数据中的四句话中学习的：“be” = 2，“to” = 3 等。

为了构建词汇表，**adapt()** 方法首先将训练句子转换为小写，去掉标点符号，这就是为什么“Be”、“be”和“be？”都被编码为“be”=2。接下来，句子在空格上被分割，得到的单词按降频进行排序，产生最终的词汇表。在编码句子时，未知词被编码为 1 。最后，由于第一句比第二句短，所以用 0 填充。

文本向量化层有很多选择。例如，如果您愿意，您可以通过设置 **standardize=None** 来保存大小写和标点符号，或者您可以传递任何您想要的标准化函数作为 **standardize** 参数。您可以通过设置 **split=None** 来防止分隔，也可以传递您自己的分隔函数。您可以设置 **output_sequence_length** 参数以确保输出序列都被裁剪或填充到所需的长度，或者您可以设置 **ragged=True** 来得到一个不规则的张量而不是规则的张量。请查看文档以获得更多选项。

In [95]:
```
text_vec_layer = tf.keras.layers.TextVectorization(ragged=True)

text_vec_layer.adapt(train_data)

text_vec_layer(["Be good!", "Question: be or be?"])
```

Out[95]:
```
<tf.RaggedTensor [[2, 1], [6, 2, 1, 2]]>
```

单词 ID 必须进行编码，通常使用嵌入层：我们将在第16章中这样做。或者，您可以将文本向量化层的 **output_mode** 参数设置为“**multi_hot**”或“**count**”，以获得相应的编码。然而，简单地计算单词通常并不理想：像“to”和“the”这样的单词如此频繁，它们根本无关紧要，然而，像“basketball”这样罕见的单词信息更丰富。

因此，与其将输出模式设置为“multi_hot”或“count”，通常最好将其设置为“**tf_idf**”，这表示term-frequency × inverse-document-frequency（TF-IDF）。这与计数编码类似，但是在训练数据中经常出现的单词被降低加权，相反，罕见的单词被提高加权。

例如：

In [96]:
```
text_vec_layer = tf.keras.layers.TextVectorization(output_mode="tf_idf")

text_vec_layer.adapt(train_data)

text_vec_layer(["Be good!", "Question: be or be?"])
```

Out[96]:
```
<tf.Tensor: shape=(2, 6), dtype=float32, numpy=
array([[0.96725637, 0.6931472 , 0.        , 0.        , 0.        ,
        0.        ],
       [0.96725637, 1.3862944 , 0.        , 0.        , 0.        ,
        1.0986123 ]], dtype=float32)>
```

有许多TF-IDF变体，但文本向量化层的方式是通过该单词的数量乘以权重，权重也就是 $log(1 + d/(f + 1))$ ，其中 d 是句子的总数（即文档）训练数据，f 是多少训练句子包含给定的单词。

例如，在这种情况下，在训练数据中有 $d = 4$ 个句子，而单词“be”出现在其中的 $f = 3$ 个中。因为“be”这个词出现在“Question: be or be?”，它被编码为 $2 \times log(1 + 4/(1 + 3)) \approx 1.3862944$ 。“question”这个词只出

现一次，但由于它是一个不太常见的单词，它的编码几乎是一样高的：
$1 \times log(1 + 4/(1 + 1)) \approx 1.0986123$。请注意，平均权重用于表示未知的单词。

In [98]:
```
2 * np.log(1 + 4 / (1 + 3))
```

Out[98]:
```
1.3862943611198906
```

In [99]:
```
1 * np.log(1 + 4 / (1 + 1))
```

Out[99]:
```
1.0986122886681098
```

这种文本编码方法使用起来很简单，它可以为基本的自然语言处理任务提供相当好的结果，但它有几个重要的局限性：它只适用于用空格分隔单词的语言，它不区分同音异义词（例如，“to bear” 与 “teddy bear”），它不会向你的模型提示“evolution”和“evolutionary”等词是相关的，等等。如果你使用 multi-hot、count 或 TF-IDF 编码， 那么单词的顺序就丢失了。 那么还有哪些选择呢？

一种选择是使用 **TensorFlow Text** 库，它提供比 TextVectorization 层更高级的文本预处理功能。 例如，它包括几个能够将文本拆分为小于单词的标记的子词分词器，这使得模型可以更容易地检测到“evolution”和“evolutionary”有一些共同点（第 16 章将详细介绍子词分词）。

另一种选择是使用预先训练好的语言模型组件。现在让我们来看看这个问题。

3.8 使用预先训练好的语言模型组件（Using Pretrained Language Model Components）

TensorFlow Hub 库使您可以很容易地在自己的模型中重用预训练过的模型组件，可用于文本、图像、音频等。这些模型组件被称为 **modules** 。只需浏览 **TF Hub** 存储库，找到您需要的存储库，并将代码示例复制到您的项目中，该模块将自动下载并捆绑到一个Keras层中，您可以直接包含在您的模型中。模块通常同时包含预处理代码和预训练的权重，而且它们通常不需要额外的训练（当然，模型的其余部分肯定需要训练）。

例如，有一些强大的预训练语言模型。最强大的是相当大的（几GB），所以作为一个快速的例子，让我们使用 **nnlm-en-dim50** 模块，版本2，这是一个相当基本的模块，将原始文本作为输入并输出50维的句子嵌入。我们将导入 TensorFlow Hub并使用它来加载模块，然后使用该模块将两个句子编码为向量：

In [ ]:
```
import tensorflow_hub as hub

hub_layer = hub.KerasLayer("https://tfhub.dev/google/nnlm-en-dim50/2")

sentence_embeddings = hub_layer(tf.constant(["To be", "Not to be"]))

sentence_embeddings.numpy().round(2)
```

hub.KerasLayer 层从给定的URL中下载该模块。这个参数模块是一个句子编码器：它将字符串作为输入，并将每个模块编码为一个向量（在这种情况下，是一个50维向量）。在内部，它解析字符串（在空格上分割单词），并使用嵌入矩阵嵌入每个单词，该矩阵在一个巨大的语料库上预先训练：Google News 7B 语料库（70亿个单词长！）。然后计算所有单词嵌入的平均值，结果是句子嵌入。

您只需要在模型中包含这个 **hub_layer** 层，这样您就准备好开始了。请注意，这种特殊的语言模型是在英语语言上训练的，但有许多其他语言可用，以及多语言模型。

最后但并非最不重要的是，通过Hugging Face提供的优秀的开源 **Transformers** 库也使您可以很容易地在自己的模型中包含强大的语言模型组件。您可以浏览 **Hugging Face Hub** ，选择您想要的模型，并使用提供的代码示例开始。它过去只包含语言模型，但现在已经扩展到包括图像模型等。

我们将在第16章中更深入地回到自然语言处理问题。现在让我们来看看Keras的图像预处理层。

3.9 图像预处理层

Keras预处理API包括三个图像预处理层：

1. **tf.keras.layers.Resizing** 可将输入的图像的大小调整到所需的大小。例如，**Resizing(height=100, width=200)** 将每个图像的大小调整为100×200，可能使图像扭曲。如果您设置 **crop_to_aspect_ratio=True** ，则图像将被裁剪到目标图像比，以避免失真。
2. **tf.keras.layers.Rescaling** 将重新调整像素值。例如，**Rescaling(scale=2/255, offset=-1)** ，可将值从0→255缩放到-1→1。
3. **tf.keras.layers.CenterCrop** 裁剪图像，只保留所需高度和宽度的中心补丁。

例如，让我们加载几个示例图像并居中裁剪它们。为此，我们将使用 Scikit-Learn 的 **load_sample_images()** 函数； 这会加载两张彩色图像，一张是中国寺庙，另一张是花朵（这需要 **Pillow** 库）：

In [109…
```
from sklearn.datasets import load_sample_images

images = load_sample_images()["images"]

crop_image_layer = tf.keras.layers.CenterCrop(height=100, width=100)

cropped_images = crop_image_layer(images)
```

In [111…
```
plt.imshow(images[0])
plt.axis("off")
plt.show()
```

In [112…
```
plt.imshow(cropped_images[0] / 255)
plt.axis("off")
plt.show()
```

Keras 还包括几个用于数据增强的层，例如 RandomCrop、RandomFlip、RandomTranslation、RandomRotation、RandomZoom、RandomHeight、 RandomWidth和RandomContrast。 这些层仅在训练期间处于活动状态，并且它们随机对输入图像应用一些变换（它们的名称是不言自明的）。 数据增强会人为地增加训练集的大小，这通常会提高性能，只要转换后的图像看起来像真实的（未增强的）图像。 我们将在下一章更详细地介绍图像处理。

在底层，Keras 预处理层基于 TensorFlow 的低级 API。

例如，归一化层使用 **tf.nn.moments()** 计算均值和方差，离散化层使用 **tf.raw_ops.Bucketize()**，分类编码使用 **tf.math.bincount()**，IntegerLookup 和 StringLookup 使用 **tf.lookup** 包，哈希和文本向量化使用 **tf.strings** 包中的几个操作，嵌入使用 **tf.nn.embedding_lookup()**，图像预处理层使用 **tf.image** 包中的操作。

如果 Keras 预处理 API 不能满足您的需求，您可能偶尔需要直接使用 TensorFlow 的低级 API。

4. TensorFlow数据集项目

TensorFlow 数据集 (TFDS) 项目使得加载常见数据集变得非常容易，从 MNIST 或 Fashion MNIST 等小型数据集到 ImageNet 等大型数据集（您将需要相当多的磁盘空间！）。该列表包括图像数据集、文本数据集（包括翻译数据集）、音频和视频数据集、时间序列等等。 可以访问 https://homl.info/tfds 查看完整列表以及每个数据集的描述。 您还可以查看 **Know Your Data**，这是一种用于探索和理解 TFDS 提供的许多数据集的工具。

你可以导入 tensorflow_datasets，通常是作为 tfds，然后调用 **tfds.load()** 函数，这将下载你想要的数据（除非它之前已经下载过）并将数据作为数据集字典返回（通常用于训练和一个用于测试，但这取决于您选择的数据集）。

例如，让我们下载 MNIST：

In [118…

```
import tensorflow_datasets as tfds
```

```
datasets = tfds.load(name="mnist")
mnist_train, mnist_test = datasets["train"], datasets["test"]
```

```
Downloading and preparing dataset Unknown size (download: Unknown size, generated: Unkno
wn size, total: Unknown size) to C:\Users\tu'tu\tensorflow_datasets\mnist\3.0.1...
Dl Completed...: 0 url [00:00, ? url/s]
Dl Size...: 0 MiB [00:00, ? MiB/s]
Extraction completed...: 0 file [00:00, ? file/s]
Generating splits...:   0%|          | 0/2 [00:00<?, ? splits/s]
Generating train examples...: 0 examples [00:00, ? examples/s]
Shuffling C:\Users\tu'tu\tensorflow_datasets\mnist\3.0.1.incompletePEUX3I\mnist-train.tf
record*...:   0%|      …
Generating test examples...: 0 examples [00:00, ? examples/s]
Shuffling C:\Users\tu'tu\tensorflow_datasets\mnist\3.0.1.incompletePEUX3I\mnist-test.tfr
ecord*...:   0%|      …
Dataset mnist downloaded and prepared to C:\Users\tu'tu\tensorflow_datasets\mnist\3.0.1.
 Subsequent calls will reuse this data.
```

load() 函数可以打乱它下载的文件：只需设置 **shuffle_files=True**。然而，这可能是不够的，所以最好再打乱一些训练数据。

然后您可以应用您想要的任何转换（通常是洗牌、批处理和预取），然后您就可以训练您的模型了。 这是一个简单的例子：

In [119…
```
for batch in mnist_train.shuffle(10_000, seed=42).batch(32).prefetch(1):
    images = batch["image"]
    labels = batch["label"]
    # [...] do something with the images and labels
```

请注意，数据集中的每个项目都是一个包含特征和标签的字典。 但是 Keras 期望每个项目都是一个包含两个元素（同样是特征和标签）的元组。 您可以使用 **map()** 方法转换数据集，如下所示：

In [120…
```
mnist_train = mnist_train.shuffle(10_000, seed=42).batch(32)

mnist_train = mnist_train.map(lambda items: (items["image"], items["label"]))

mnist_train = mnist_train.prefetch(1)
```

但是通过设置 **as_supervised=True** 让 **load()** 函数为你做这件事更简单（显然这只适用于标记数据集）。

最后，TFDS 提供了一种使用 **split** 参数拆分数据的便捷方法。 例如，如果想使用前90%的训练集进行训练，剩下的10%进行验证，整个测试集进行测试，那么可以设置 **split=["train[:90%]", "train[90%:]", "test"]** 。**load()** 函数将返回所有三个集合。 这是一个完整的示例，使用 TFDS 加载和拆分 MNIST 数据集，然后使用这些集来训练和评估一个简单的 Keras 模型：

In [122…
```
train_set, valid_set, test_set = tfds.load(
    name="mnist",
    split=["train[:90%]", "train[90%:]", "test"],
    as_supervised=True
)

train_set = train_set.shuffle(10_000, seed=42).batch(32).prefetch(1)

valid_set = valid_set.batch(32).cache()

test_set = test_set.batch(32).cache()

tf.random.set_seed(42)

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
```

```
    tf.keras.layers.Dense(10, activation="softmax")
])

model.compile(loss="sparse_categorical_crossentropy",
              optimizer="nadam",
              metrics=["accuracy"])

history = model.fit(train_set, validation_data=valid_set, epochs=5)

test_loss, test_accuracy = model.evaluate(test_set)
```

```
Epoch 1/5
1688/1688 [==============================] - 3s 1ms/step - loss: 9.6797 - accuracy: 0.83
51 - val_loss: 6.2637 - val_accuracy: 0.8743
Epoch 2/5
1688/1688 [==============================] - 2s 896us/step - loss: 5.6372 - accuracy: 0.
8786 - val_loss: 5.6044 - val_accuracy: 0.8787
Epoch 3/5
1688/1688 [==============================] - 1s 864us/step - loss: 5.0631 - accuracy: 0.
8828 - val_loss: 5.6720 - val_accuracy: 0.8788
Epoch 4/5
1688/1688 [==============================] - 2s 880us/step - loss: 4.8132 - accuracy: 0.
8863 - val_loss: 5.4716 - val_accuracy: 0.8793
Epoch 5/5
1688/1688 [==============================] - 2s 894us/step - loss: 4.6871 - accuracy: 0.
8879 - val_loss: 5.6099 - val_accuracy: 0.8793
313/313 [==============================] - 0s 1ms/step - loss: 5.1157 - accuracy: 0.8858
```

恭喜，您已经完成了这一技术性很强的章节！ 你可能觉得它离神经网络的抽象之美有点远，但事实是深度学习往往涉及大量数据，知道如何高效地加载、解析和预处理数据是一项至关重要的技能。 在下一章中，我们将研究卷积神经网络，它是用于图像处理和许多其他应用的最成功的神经网络架构之一。