3.4 电影评论分类：二分类问题

二分类问题可能是应用最广泛的机器学习问题。在这个例子中，你将学习根据电影评论的文字内容将其划分为正面或负面。

3.4.1 IMDB 数据集

本节使用 IMDB 数据集，它包含来自互联网电影数据库（IMDB）的 50 000 条严重两极分化的评论。数据集被分为用于训练的 25 000 条评论与用于测试的 25 000 条评论，训练集和测试集都包含 50% 的正面评论和 50% 的负面评论。

为什么要将训练集和测试集分开？因为你不应该将训练机器学习模型的同一批数据再用于测试模型！模型在训练数据上的表现很好，并不意味着它在前所未见的数据上也会表现得很好，而且你真正关心的是模型在新数据上的性能（因为你已经知道了训练数据对应的标签，显然不再需要模型来进行预测）。例如，你的模型最终可能只是**记住**了训练样本和目标值之间的映射关系，但这对在前所未见的数据上进行预测毫无用处。下一章将会更详细地讨论这一点。

与 MNIST 数据集一样，IMDB 数据集也内置于 Keras 库。它已经过预处理：评论（单词序列）已经被转换为整数序列，其中每个整数代表字典中的某个单词。

下列代码将会加载 IMDB 数据集（第一次运行时会下载大约 80MB 的数据）。

代码清单 3-1 加载 IMDB 数据集

```
from keras.datasets import imdb

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(
    num_words=10000)
```

参数 `num_words=10000` 的意思是仅保留训练数据中前 10 000 个最常出现的单词。低频单词将被舍弃。这样得到的向量数据不会太大，便于处理。

`train_data` 和 `test_data` 这两个变量都是评论组成的列表，每条评论又是单词索引组成的列表（表示一系列单词）。`train_labels` 和 `test_labels` 都是 0 和 1 组成的列表，其中 0 代表**负面**（negative），1 代表**正面**（positive）。

```
>>> train_data[0]
[1, 14, 22, 16, ... 178, 32]

>>> train_labels[0]
1
```

由于限定为前 10 000 个最常见的单词，单词索引都不会超过 10 000。

```
>>> max([max(sequence) for sequence in train_data])
9999
```

下面这段代码很有意思，你可以将某条评论迅速解码为英文单词。

```
word_index = imdb.get_word_index()  <-- word_index 是一个将单词映射为整数索引的字典
reverse_word_index = dict(
    [(value, key) for (key, value) in word_index.items()])
decoded_review = ' '.join(
    [reverse_word_index.get(i - 3, '?') for i in train_data[0]])
```

键值颠倒，将整数索引映射为单词

将评论解码。注意，索引减去了 3，因为 0、1、2 是为“padding”（填充）、“start of sequence”（序列开始）、“unknown”（未知词）分别保留的索引

3.4.2　准备数据

你不能将整数序列直接输入神经网络。你需要将列表转换为张量。转换方法有以下两种。

- 填充列表，使其具有相同的长度，再将列表转换成形状为 `(samples, word_indices)` 的整数张量，然后网络第一层使用能处理这种整数张量的层（即 `Embedding` 层，本书后面会详细介绍）。
- 对列表进行 one-hot 编码，将其转换为 0 和 1 组成的向量。举个例子，序列 `[3, 5]` 将会被转换为 10 000 维向量，只有索引为 3 和 5 的元素是 1，其余元素都是 0。然后网络第一层可以用 `Dense` 层，它能够处理浮点数向量数据。

下面我们采用后一种方法将数据向量化。为了加深理解，你可以手动实现这一方法，如下所示。

代码清单 3-2　将整数序列编码为二进制矩阵

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))  <-- 创建一个形状为 (len(sequences), dimension) 的零矩阵
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.  <-- 将 results[i] 的指定索引设为 1
    return results

x_train = vectorize_sequences(train_data)  <-- 将训练数据向量化
x_test = vectorize_sequences(test_data)    <-- 将测试数据向量化
```

样本现在变成了这样：

```
>>> x_train[0]
array([ 0.,  1.,  1., ...,  0.,  0.,  0.])
```

你还应该将标签向量化，这很简单。

```
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

现在可以将数据输入到神经网络中。

3.4.3　构建网络

输入数据是向量，而标签是标量（1 和 0），这是你会遇到的最简单的情况。有一类网

络在这种问题上表现很好，就是带有 relu 激活的全连接层（Dense）的简单堆叠，比如 Dense(16, activation='relu')。

传入 Dense 层的参数（16）是该层隐藏单元的个数。一个**隐藏单元**（hidden unit）是该层表示空间的一个维度。我们在第 2 章讲过，每个带有 relu 激活的 Dense 层都实现了下列张量运算：

```
output = relu(dot(W, input) + b)
```

16 个隐藏单元对应的权重矩阵 W 的形状为 (input_dimension, 16)，与 W 做点积相当于将输入数据投影到 16 维表示空间中（然后再加上偏置向量 b 并应用 relu 运算）。你可以将表示空间的维度直观地理解为“网络学习内部表示时所拥有的自由度”。隐藏单元越多（即更高维的表示空间），网络越能够学到更加复杂的表示，但网络的计算代价也变得更大，而且可能会导致学到不好的模式（这种模式会提高训练数据上的性能，但不会提高测试数据上的性能）。

对于这种 Dense 层的堆叠，你需要确定以下两个关键架构：

- 网络有多少层；
- 每层有多少个隐藏单元。

第 4 章中的原则将会指导你对上述问题做出选择。现在你只需要相信我选择的下列架构：

- 两个中间层，每层都有 16 个隐藏单元；
- 第三层输出一个标量，预测当前评论的情感。

中间层使用 relu 作为激活函数，最后一层使用 sigmoid 激活以输出一个 0~1 范围内的概率值（表示样本的目标值等于 1 的可能性，即评论为正面的可能性）。relu（rectified linear unit，整流线性单元）函数将所有负值归零（见图 3-4），而 sigmoid 函数则将任意值“压缩”到 [0, 1] 区间内（见图 3-5），其输出值可以看作概率值。

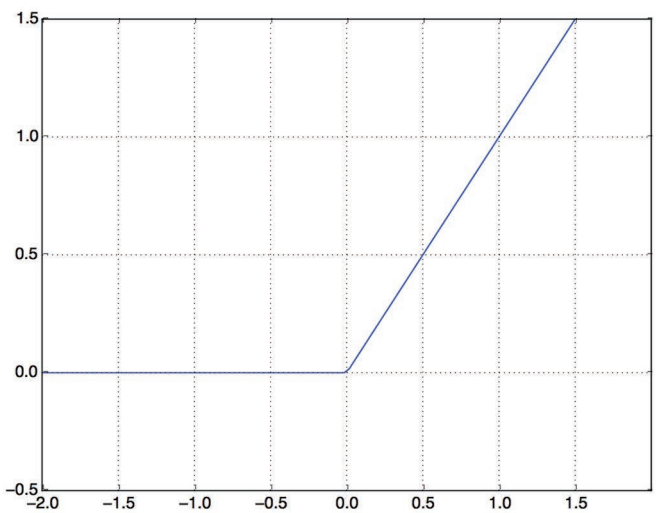

图 3-4 整流线性单元函数

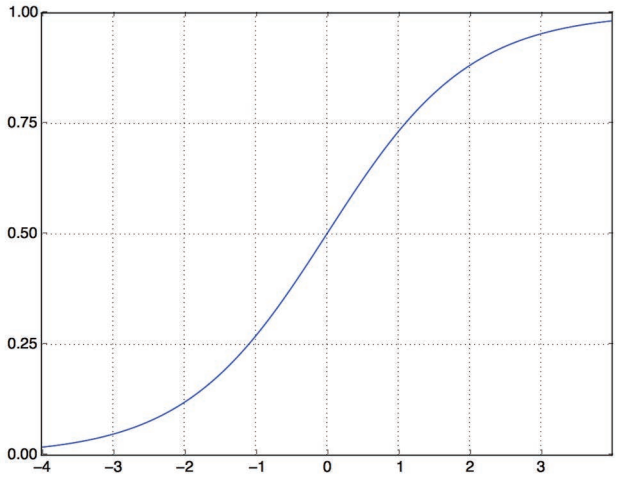

图 3-5　sigmoid 函数

图 3-6 显示了网络的结构。代码清单 3-3 是其 Keras 实现，与前面见过的 MNIST 例子类似。

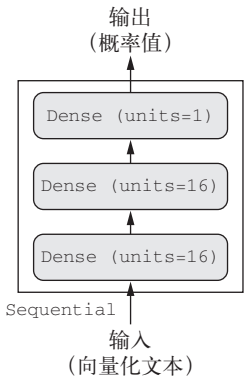

图 3-6　三层网络

代码清单 3-3　模型定义

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

什么是激活函数？为什么要使用激活函数？

如果没有 relu 等激活函数（也叫**非线性**），Dense 层将只包含两个线性运算——点积和加法：

```
output = dot(W, input) + b
```

这样 Dense 层就只能学习输入数据的**线性变换**（仿射变换）：该层的**假设空间**是从输入数据到 16 位空间所有可能的线性变换集合。这种假设空间非常有限，无法利用多个表示层的优势，因为多个线性层堆叠实现的仍是线性运算，添加层数并不会扩展假设空间。

为了得到更丰富的假设空间，从而充分利用多层表示的优势，你需要添加非线性或激活函数。relu 是深度学习中最常用的激活函数，但还有许多其他函数可选，它们都有类似的奇怪名称，比如 prelu、elu 等。

最后，你需要选择损失函数和优化器。由于你面对的是一个二分类问题，网络输出是一个概率值（网络最后一层使用 sigmoid 激活函数，仅包含一个单元），那么最好使用 binary_crossentropy（二元交叉熵）损失。这并不是唯一可行的选择，比如你还可以使用 mean_squared_error（均方误差）。但对于输出概率值的模型，**交叉熵**（crossentropy）往往是最好的选择。交叉熵是来自于信息论领域的概念，用于衡量概率分布之间的距离，在这个例子中就是真实分布与预测值之间的距离。

下面的步骤是用 rmsprop 优化器和 binary_crossentropy 损失函数来配置模型。注意，我们还在训练过程中监控精度。

代码清单 3-4　编译模型

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

上述代码将优化器、损失函数和指标作为字符串传入，这是因为 rmsprop、binary_crossentropy 和 accuracy 都是 Keras 内置的一部分。有时你可能希望配置自定义优化器的参数，或者传入自定义的损失函数或指标函数。前者可通过向 optimizer 参数传入一个优化器类实例来实现，如代码清单 3-5 所示；后者可通过向 loss 和 metrics 参数传入函数对象来实现，如代码清单 3-6 所示。

代码清单 3-5　配置优化器

```
from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

代码清单 3-6　使用自定义的损失和指标

```
from keras import losses
from keras import metrics

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss=losses.binary_crossentropy,
              metrics=[metrics.binary_accuracy])
```

3.4.4　验证你的方法

为了在训练过程中监控模型在前所未见的数据上的精度，你需要将原始训练数据留出 10 000 个样本作为验证集。

代码清单 3-7　留出验证集

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

现在使用 512 个样本组成的小批量，将模型训练 20 个轮次（即对 `x_train` 和 `y_train` 两个张量中的所有样本进行 20 次迭代）。与此同时，你还要监控在留出的 10 000 个样本上的损失和精度。你可以通过将验证数据传入 `validation_data` 参数来完成。

代码清单 3-8　训练模型

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])

history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

在 CPU 上运行，每轮的时间不到 2 秒，训练过程将在 20 秒内结束。每轮结束时会有短暂的停顿，因为模型要计算在验证集的 10 000 个样本上的损失和精度。

注意，调用 `model.fit()` 返回了一个 `History` 对象。这个对象有一个成员 `history`，它是一个字典，包含训练过程中的所有数据。我们来看一下。

```
>>> history_dict = history.history
>>> history_dict.keys()
dict_keys(['val_acc', 'acc', 'val_loss', 'loss'])
```

字典中包含 4 个条目，对应训练过程和验证过程中监控的指标。在下面两个代码清单中，我们将使用 Matplotlib 在同一张图上绘制训练损失和验证损失（见图 3-7），以及训练精度和验证精度（见图 3-8）。请注意，由于网络的随机初始化不同，你得到的结果可能会略有不同。

代码清单 3-9 绘制训练损失和验证损失

```
import matplotlib.pyplot as plt

history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

epochs = range(1, len(loss_values) + 1)

plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

'bo' 表示蓝色圆点

'b' 表示蓝色实线

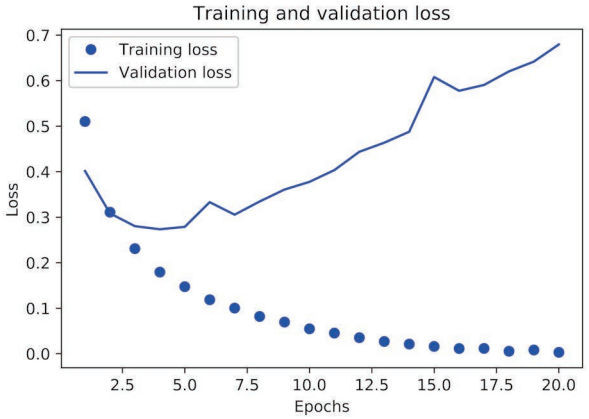

图 3-7 训练损失和验证损失

代码清单 3-10 绘制训练精度和验证精度

```
plt.clf()
acc = history_dict['acc']
val_acc = history_dict['val_acc']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```

清空图像

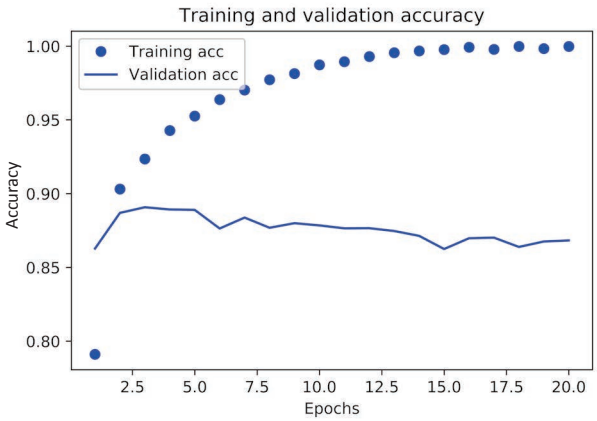

图 3-8　训练精度和验证精度

如你所见，训练损失每轮都在降低，训练精度每轮都在提升。这就是梯度下降优化的预期结果——你想要最小化的量随着每次迭代越来越小。但验证损失和验证精度并非如此：它们似乎在第四轮达到最佳值。这就是我们之前警告过的一种情况：模型在训练数据上的表现越来越好，但在前所未见的数据上不一定表现得越来越好。准确地说，你看到的是**过拟合**（overfit）：在第二轮之后，你对训练数据过度优化，最终学到的表示仅针对于训练数据，无法泛化到训练集之外的数据。

在这种情况下，为了防止过拟合，你可以在 3 轮之后停止训练。通常来说，你可以使用许多方法来降低过拟合，我们将在第 4 章中详细介绍。

我们从头开始训练一个新的网络，训练 4 轮，然后在测试数据上评估模型。

代码清单 3-11　从头开始重新训练一个模型

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
```

最终结果如下所示。

```
>>> results
[0.2929924130630493, 0.88327999999999995]
```

这种相当简单的方法得到了 88% 的精度。利用最先进的方法，你应该能够得到接近 95% 的精度。

3.4.5 使用训练好的网络在新数据上生成预测结果

训练好网络之后，你希望将其用于实践。你可以用 predict 方法来得到评论为正面的可能性大小。

```
>>> model.predict(x_test)
array([[ 0.98006207]
       [ 0.99758697]
       [ 0.99975556]
       ...,
       [ 0.82167041]
       [ 0.02885115]
       [ 0.65371346]], dtype=float32)
```

如你所见，网络对某些样本的结果非常确信（大于等于 0.99，或小于等于 0.01），但对其他结果却不那么确信（0.6 或 0.4）。

3.4.6 进一步的实验

通过以下实验，你可以确信前面选择的网络架构是非常合理的，虽然仍有改进的空间。

- 前面使用了两个隐藏层。你可以尝试使用一个或三个隐藏层，然后观察对验证精度和测试精度的影响。
- 尝试使用更多或更少的隐藏单元，比如 32 个、64 个等。
- 尝试使用 mse 损失函数代替 binary_crossentropy。
- 尝试使用 tanh 激活（这种激活在神经网络早期非常流行）代替 relu。

3.4.7 小结

下面是你应该从这个例子中学到的要点。

- 通常需要对原始数据进行大量预处理，以便将其转换为张量输入到神经网络中。单词序列可以编码为二进制向量，但也有其他编码方式。
- 带有 relu 激活的 Dense 层堆叠，可以解决很多种问题（包括情感分类），你可能会经常用到这种模型。
- 对于二分类问题（两个输出类别），网络的最后一层应该是只有一个单元并使用 sigmoid 激活的 Dense 层，网络输出应该是 0~1 范围内的标量，表示概率值。
- 对于二分类问题的 sigmoid 标量输出，你应该使用 binary_crossentropy 损失函数。
- 无论你的问题是什么，rmsprop 优化器通常都是足够好的选择。这一点你无须担心。
- 随着神经网络在训练数据上的表现越来越好，模型最终会过拟合，并在前所未见的数据上得到越来越差的结果。一定要一直监控模型在训练集之外的数据上的性能。

3.5 新闻分类：多分类问题

上一节中，我们介绍了如何用密集连接的神经网络将向量输入划分为两个互斥的类别。但

如果类别不止两个，要怎么做？

本节你会构建一个网络，将路透社新闻划分为 46 个互斥的主题。因为有多个类别，所以这是**多分类**（multiclass classification）问题的一个例子。因为每个数据点只能划分到一个类别，所以更具体地说，这是**单标签、多分类**（single-label, multiclass classification）问题的一个例子。如果每个数据点可以划分到多个类别（主题），那它就是一个**多标签、多分类**（multilabel, multiclass classification）问题。

3.5.1　路透社数据集

本节使用**路透社数据集**，它包含许多短新闻及其对应的主题，由路透社在 1986 年发布。它是一个简单的、广泛使用的文本分类数据集。它包括 46 个不同的主题：某些主题的样本更多，但训练集中每个主题都有至少 10 个样本。

与 IMDB 和 MNIST 类似，路透社数据集也内置为 Keras 的一部分。我们来看一下。

代码清单 3-12　加载路透社数据集

```
from keras.datasets import reuters

(train_data, train_labels), (test_data, test_labels) = reuters.load_data(
    num_words=10000)
```

与 IMDB 数据集一样，参数 `num_words=10000` 将数据限定为前 10 000 个最常出现的单词。我们有 8982 个训练样本和 2246 个测试样本。

```
>>> len(train_data)
8982
>>> len(test_data)
2246
```

与 IMDB 评论一样，每个样本都是一个整数列表（表示单词索引）。

```
>>> train_data[10]
[1, 245, 273, 207, 156, 53, 74, 160, 26, 14, 46, 296, 26, 39, 74, 2979,
3554, 14, 46, 4689, 4329, 86, 61, 3499, 4795, 14, 61, 451, 4329, 17, 12]
```

如果好奇的话，你可以用下列代码将索引解码为单词。

代码清单 3-13　将索引解码为新闻文本

```
word_index = reuters.get_word_index()
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
decoded_newswire = ' '.join([reverse_word_index.get(i - 3, '?') for i in
    train_data[0]])
```

注意，索引减去了 3，因为 0、1、2 是为“padding”（填充）、“start of sequence”（序列开始）、“unknown”（未知词）分别保留的索引

样本对应的标签是一个 0~45 范围内的整数，即话题索引编号。

```
>>> train_labels[10]
3
```

3.5.2 准备数据

你可以使用与上一个例子相同的代码将数据向量化。

代码清单 3-14 编码数据

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data) <-- 将训练数据向量化
x_test = vectorize_sequences(test_data)   <-- 将测试数据向量化
```

将标签向量化有两种方法：你可以将标签列表转换为整数张量，或者使用 one-hot 编码。one-hot 编码是分类数据广泛使用的一种格式，也叫**分类编码**（categorical encoding）。6.1 节给出了 one-hot 编码的详细解释。在这个例子中，标签的 one-hot 编码就是将每个标签表示为全零向量，只有标签索引对应的元素为 1。其代码实现如下。

```
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results

one_hot_train_labels = to_one_hot(train_labels)   <-- 将训练标签向量化
one_hot_test_labels = to_one_hot(test_labels)     <-- 将测试标签向量化
```

注意，Keras 内置方法可以实现这个操作，你在 MNIST 例子中已经见过这种方法。

```
from keras.utils.np_utils import to_categorical

one_hot_train_labels = to_categorical(train_labels)
one_hot_test_labels = to_categorical(test_labels)
```

3.5.3 构建网络

这个主题分类问题与前面的电影评论分类问题类似，两个例子都是试图对简短的文本片段进行分类。但这个问题有一个新的约束条件：输出类别的数量从 2 个变为 46 个。输出空间的维度要大得多。

对于前面用过的 `Dense` 层的堆叠，每层只能访问上一层输出的信息。如果某一层丢失了与分类问题相关的一些信息，那么这些信息无法被后面的层找回，也就是说，每一层都可能成为信息瓶颈。上一个例子使用了 16 维的中间层，但对这个例子来说 16 维空间可能太小了，无法学会区分 46 个不同的类别。这种维度较小的层可能成为信息瓶颈，永久地丢失相关信息。

出于这个原因，下面将使用维度更大的层，包含 64 个单元。

代码清单 3-15　模型定义

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

关于这个架构还应该注意另外两点。

- 网络的最后一层是大小为 46 的 Dense 层。这意味着，对于每个输入样本，网络都会输出一个 46 维向量。这个向量的每个元素（即每个维度）代表不同的输出类别。
- 最后一层使用了 softmax 激活。你在 MNIST 例子中见过这种用法。网络将输出在 46 个不同输出类别上的**概率分布**——对于每一个输入样本，网络都会输出一个 46 维向量，其中 output[i] 是样本属于第 i 个类别的概率。46 个概率的总和为 1。

对于这个例子，最好的损失函数是 categorical_crossentropy（分类交叉熵）。它用于衡量两个概率分布之间的距离，这里两个概率分布分别是网络输出的概率分布和标签的真实分布。通过将这两个分布的距离最小化，训练网络可使输出结果尽可能接近真实标签。

代码清单 3-16　编译模型

```
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

3.5.4　验证你的方法

我们在训练数据中留出 1000 个样本作为验证集。

代码清单 3-17　留出验证集

```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]

y_val = one_hot_train_labels[:1000]
partial_y_train = one_hot_train_labels[1000:]
```

现在开始训练网络，共 20 个轮次。

代码清单 3-18　训练模型

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

最后，我们来绘制损失曲线和精度曲线（见图 3-9 和图 3-10）。

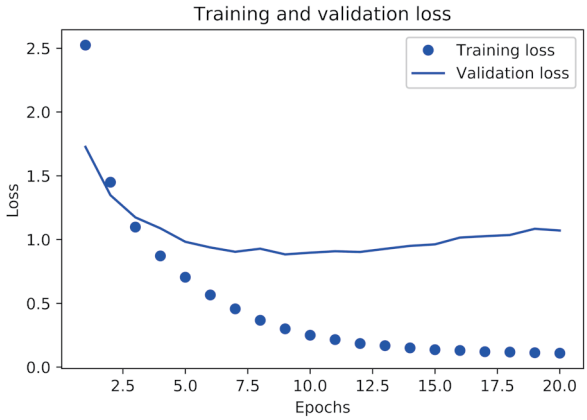

图 3-9 训练损失和验证损失

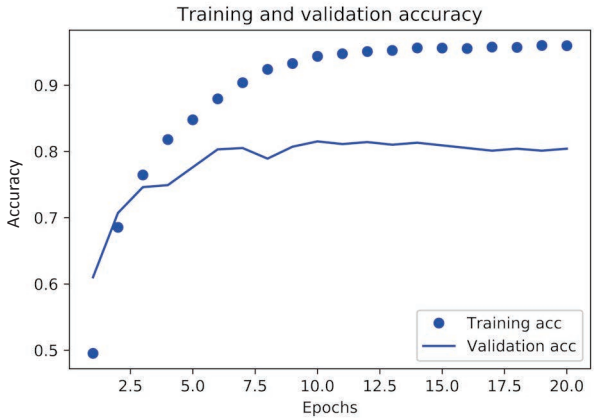

图 3-10 训练精度和验证精度

代码清单 3-19 绘制训练损失和验证损失

```
import matplotlib.pyplot as plt

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

代码清单 3-20　绘制训练精度和验证精度

```
plt.clf()   <—— 清空图像

acc = history.history['acc']
val_acc = history.history['val_acc']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```

网络在训练 9 轮后开始过拟合。我们从头开始训练一个新网络，共 9 个轮次，然后在测试集上评估模型。

代码清单 3-21　从头开始重新训练一个模型

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(partial_x_train,
          partial_y_train,
          epochs=9,
          batch_size=512,
          validation_data=(x_val, y_val))
results = model.evaluate(x_test, one_hot_test_labels)
```

最终结果如下。

```
>>> results
[0.9565213431445807, 0.79697239536954589]
```

这种方法可以得到约 80% 的精度。对于平衡的二分类问题，完全随机的分类器能够得到 50% 的精度。但在这个例子中，完全随机的精度约为 19%，所以上述结果相当不错，至少和随机的基准比起来还不错。

```
>>> import copy
>>> test_labels_copy = copy.copy(test_labels)
>>> np.random.shuffle(test_labels_copy)
>>> hits_array = np.array(test_labels) == np.array(test_labels_copy)
>>> float(np.sum(hits_array)) / len(test_labels)
0.18655387355298308
```

3.5.5 在新数据上生成预测结果

你可以验证，模型实例的 predict 方法返回了在 46 个主题上的概率分布。我们对所有测试数据生成主题预测。

代码清单 3-22 在新数据上生成预测结果

```
predictions = model.predict(x_test)
```

predictions 中的每个元素都是长度为 46 的向量。

```
>>> predictions[0].shape
(46,)
```

这个向量的所有元素总和为 1。

```
>>> np.sum(predictions[0])
1.0
```

最大的元素就是预测类别，即概率最大的类别。

```
>>> np.argmax(predictions[0])
4
```

3.5.6 处理标签和损失的另一种方法

前面提到了另一种编码标签的方法，就是将其转换为整数张量，如下所示。

```
y_train = np.array(train_labels)
y_test = np.array(test_labels)
```

对于这种编码方法，唯一需要改变的是损失函数的选择。对于代码清单 3-21 使用的损失函数 categorical_crossentropy，标签应该遵循分类编码。对于整数标签，你应该使用 sparse_categorical_crossentropy。

```
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])
```

这个新的损失函数在数学上与 categorical_crossentropy 完全相同，二者只是接口不同。

3.5.7 中间层维度足够大的重要性

前面提到，最终输出是 46 维的，因此中间层的隐藏单元个数不应该比 46 小太多。现在来看一下，如果中间层的维度远远小于 46（比如 4 维），造成了信息瓶颈，那么会发生什么？

代码清单 3-23 具有信息瓶颈的模型

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

```
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(partial_x_train,
          partial_y_train,
          epochs=20,
          batch_size=128,
          validation_data=(x_val, y_val))
```

现在网络的验证精度最大约为 71%，比前面下降了 8%。导致这一下降的主要原因在于，你试图将大量信息（这些信息足够恢复 46 个类别的分割超平面）压缩到维度很小的中间空间。网络能够将**大部分**必要信息塞入这个四维表示中，但并不是全部信息。

3.5.8　进一步的实验

- 尝试使用更多或更少的隐藏单元，比如 32 个、128 个等。
- 前面使用了两个隐藏层，现在尝试使用一个或三个隐藏层。

3.5.9　小结

下面是你应该从这个例子中学到的要点。

- 如果要对 *N* 个类别的数据点进行分类，网络的最后一层应该是大小为 *N* 的 `Dense` 层。
- 对于单标签、多分类问题，网络的最后一层应该使用 `softmax` 激活，这样可以输出在 *N* 个输出类别上的概率分布。
- 这种问题的损失函数几乎总是应该使用分类交叉熵。它将网络输出的概率分布与目标的真实分布之间的距离最小化。
- 处理多分类问题的标签有两种方法。
  - 通过分类编码（也叫 one-hot 编码）对标签进行编码，然后使用 `categorical_crossentropy` 作为损失函数。
  - 将标签编码为整数，然后使用 `sparse_categorical_crossentropy` 损失函数。
- 如果你需要将数据划分到许多类别中，应该避免使用太小的中间层，以免在网络中造成信息瓶颈。

3.6　预测房价：回归问题

前面两个例子都是分类问题，其目标是预测输入数据点所对应的单一离散的标签。另一种常见的机器学习问题是**回归**问题，它预测一个连续值而不是离散的标签，例如，根据气象数据预测明天的气温，或者根据软件说明书预测完成软件项目所需要的时间。

注意　不要将**回归**问题与 **logistic 回归**算法混为一谈。令人困惑的是，logistic 回归不是回归算法，而是分类算法。