第 10 章 循环神经网络

在前面的章节中，我们学习了普通神经网络和卷积神经网络。在一般情况下，普通神经网络通过增加隐藏层提高模型复杂度，同时控制过拟合，在很多实际问题中可以取得很好的预测效果；当数据的自变量具有一定的复杂结构时（如，图片像素点），卷积神经网络可以进一步提高预测准确率。但是，当处理具有序列特征的数据（序列数据）时，无论是普通神经网络，还是卷积神经网络都无法取得很好的效果。这是因为无论是普通神经网络还是卷积神经网络都无法有效地考虑序列数据观测点之间的关系。

然而，序列数据广泛存在于生产生活中，如天气数据、经济数据、语言数据等。因此, 循环神经网络（Recurrent Neural Network，RNN）便应运而生了。在现实生活中，循环神经网络在很多领域获得了极大成功，如机器翻译、语音识别、自动文本归纳、手势识别以及步态识别等。

在本章及本书后面的章节中，我们将基于自然语言处理问题，陆续学习循环神经网络的原理和应用。特别的，本章将介绍：

- 词嵌入（Word Embedding），一种用数字向量表达字母、字或者词的方法；
- 简单循环神经网络（Vanilla Recurrent Neural Network）。

10.1 分析 IMDB 电影评论数据

10.1.1 IMDB 电影评论数据

首先看一个例子，IMDB 电影评论数据。IMDB 数据包含了 5 万条电影评论，以及这 5 万条电影评论代表的关于电影的情绪，喜欢（POSITIVE）或不喜欢（NEGATIVE）。先读入数据。在下面的代码中，`lambda x:x[:-1]`表示定义一个没有名字的函数，该函数的参数为 `x`，`x[:-1]`表示去除 `x` 的最后一个字符；函数 `map()`对第二个参数列表的每一个元素调用第一个参数的函数。例如，`map(lambda x:x[:-1], f.readlines())`将去除 `f.readlines()`载入的每一句话的最后一个字符。

```
import numpy as np
with open('./data/reviews.txt', 'r') as f:
    raw_reviews = list(map(lambda x:x[:-1], f.readlines()))

with open('./data/labels.txt', 'r') as f:
    raw_labels = list(map(lambda x:x[:-1].upper(),
```

```
f.readlines()))
```

下面代码列出一些数据观测点（一个观测点包含一条电影评论和它对应的情绪）。可以看到，电影评论表达的情绪和因变量是比较吻合的。例如，第一条评论开始第一句话是“one of the best love stories i have ever seen. ”，因变量是“POSITIVE”。

```
def print_label_and_review(i):
    print(raw_labels[i] + "\t:\t" + raw_reviews[i][:50] + "...")
print("Labels \t\t : \t Reviews\n..... \n")

print_label_and_review(2134)
print_label_and_review(4998)
print_label_and_review(5297)
print_label_and_review(6267)
print_label_and_review(12816)
print_label_and_review(21934)

Labels       :   Reviews
.....

POSITIVE    :   one of the best love stories i have ever seen .
it ...
POSITIVE    :   this schiffer guy is a real genius  the movie
is of...
NEGATIVE    :   if you haven  t seen this  it  s terrible . it
is  ...
NEGATIVE    :   comment this movie is impossible . is terrible
ve ...
POSITIVE    :   adrian pasdar is excellent is this film . he
makes ...
POSITIVE    :   excellent episode movie ala pulp fiction .
days   ...
```

在电影评论网站上，有些观众看电影之后喜欢对电影进行评论，但是没有明确表明对电影喜欢与否。因此，我们希望可以通过分析这些数据，建立模型，之后把电影评论输入模型，模型可以判断发表电影评论的观众喜欢或者不喜欢所评论的电影。因此，人们也称类似的数据分析为情绪分析。基于该数据的数据结构和建模目的，我们可以建立一个分类模型实现该目标，如图 10-1 所示。

图 10-1 分类模型分析电影评论的情绪

在分类模型中，输入数据和输出数据必须都是数字。而在该数据中，输入数据和输出数据都是文字。因此需要先把文字信息转化为数字信息，即需要先完成图 10-2 虚线框的步骤。

图 10-2 分类模型分析电影评论情绪，文字信息转化为数字信息（虚线框部分）

这里容易处理的是输出数据，因为输出数据只有两种可能，POSITIVE 和 NEGATIVE。我们可以把 POSTIVE 编码为 1，NEGATIVE 编码为 0。代码如下，

```
targets = list()
for label in raw_labels:
    if label == "POSITIVE":
        targets.append(1)
    else:
        targets.append(0)
```

输入数据转化为数字信息是比较麻烦的，因为，

- 电影评论是文字，而且单词数量很多。
- 评论的长短不一，而之前学过的神经网络模型输入向量的长度都是固定的。

该数据的建模目标是通过输入电影评论预测观众是否喜欢所评论的电影。换句话说，希望建立电影评论与观众情绪的一种关系，这样可以通过建立的关系得到观众情绪的预测。我们可以先问自己一个问题：当我们阅读一条电影评论后，如何判断评论所蕴含的情绪？很直观的一种方法是：当一个或者多个褒义词出现在评论中（如，excellent（出色的），perfect（完美的），amazing（迷人的），wonderful（奇妙的）），我们会认为评论是正面的（POSITIVE）；当一个或者多个贬义词出现在评论中（如，worst（最糟的），awful（可怕的），waste（浪费），poor（不好的），terrible（糟糕的），disappointment（失望）），我们会认为评论是负面的（NEGATIVE）。总的来说，我们认为褒义词意味着正面评价，贬义词意味着负面评价。

换句话说，我们可以通过查看评论中出现了哪些褒义词或者贬义词来判断评论的情绪。基于这样的逻辑，如果模型可以实现类似的判断流程，那么模型便可以对评论做出预测。为了实现这一点，我们首先统计每个句子出现的单词。

- 记录每一条评论出现的单词（下面代码中的 `tokens`），以及所有评论出现的单词（下面代码中的 `vocab`）。

```
tokens = list()
vocab = set()
for sentence in raw_reviews:
    words_in_sent = set()
    for word in sentence.split(" "):
        words_in_sent.add(word)
        vocab.add(word)
    tokens.append(list(words_in_sent))
vocab = list(vocab)
```

下面可以看到前 3 条评论包含的单词量（不重复计算，即，如果一个单词多次出现在评论中，只记一次）为 93，92，232。在所有评论中，出现的单词（包括标点等各种字符）数量为 74074。

```
print("First review contains %2d unique words;" %
(len(tokens[0])))
print("Second review contains %2d unique words;" %
(len(tokens[1])))
print("Third review contains %3d unique words;" %
(len(tokens[2])))
print("Total words contained in all reviews: %5d" %
(len(vocab)))

First review contains 93 unique words;
Second review contains 92 unique words;
Third review contains 232 unique words;
Total words contained in all reviews: 74074
```

- 对每个单词进行编码，即让每个单词对应一个数字。为了方便，按照单词出现的先后顺序对单词编码。在 Python 中，可以使用字典实现单词编码。如下面代码的 word2index。

```
word2index = {}
for idx, word in enumerate(vocab):
    word2index[word] = idx
dict(list(word2index.items())[0:10])

{'': 0,
 'flopped': 1,
 'tranquilizers': 2,
 'hyuck': 3,
 'huddled': 4,
 'cordell': 5,
 'ustinov': 6,
 'sergio': 7,
 'freemasons': 8,
 'seeping': 9}
```

- 对每一条评论进行编码，每一条评论用长度相同的向量表示。向量的长度为单词总数量 74074；向量的大部分元素都是 0，只有评论出现的单词的编码对应位置设为 1。即，每一条评论的编码记录了该评论出现过哪些单词。在下面代码中，reviews_arr 每一行表示一条评论的信息，reviews_arr 的行数为评论数，列数为单词总数量；reviews 是一个列

表，列表元素表示一条评论所有单词对应的编码。因此，reviews_arr 和 reviews 包含一样的信息。

例如，如果有两条评论，分别为“I love this love story！”和“this movie is terrible”。两条评论出现的单词总数为 8，假设对这 8 个单词的编码为 {'!':0,'love':1,'terrible':2,'I':3,'movie':4, 'this':5, 'is':6, 'story':7}。那么可以使用下面的 reviews_arr 和 reviews 表示这两条评论蕴含的信息：

- reviews_arr 是一个2×8的矩阵，

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

- reviews 是一个长度为 2 的列表[[0,1,3,5,7], [2,4,5,6]]。

对于 IMDB 数据，可以通过如下代码得到 reviews_arr 和 reviews。

```
reviews = list()
reviews_arr = np.zeros((len(tokens), len(vocab)))
for i, sentence in enumerate(tokens):
    sent_indices = list()
    for word in sentence:
        sent_indices.append(word2index[word])
        reviews_arr[i][word2index[word]] = 1
    reviews.append(sent_indices)
print("reviews 包含的第一条评论的编码：\n", reviews[0])
print("reviews_arr 包含的第一条评论的编码前 100 个元素：\n",
reviews_arr[0][0:100])
```

```
reviews 包含的第一条评论的编码：
 [0, 48318, 22779, 70804, 44198, 33550, 27847, 61668, 9244,
1474, 10676, 37944, 6952, 3780, 12149, 53296, 18142, 59026,
2959, 6682, 63144, 34473, 27628, 55075, 70598, 66894, 65496,
26243, 22007, 59975, 53054, 20529, 48419, 51347, 19674, 61175,
27079, 1848, 53678, 32234, 33397, 17962, 56263, 66661, 57130,
43726, 3046, 29685, 23764, 42261, 14250, 43439, 42850, 55163,
33164, 57180, 22965, 24108, 73681, 11157, 26339, 29449, 43796,
50307, 58032, 67337, 29162, 73120, 10554, 26633, 6534, 52324,
49729, 11754, 12647, 36994, 21278, 11483, 44134, 50336, 30374,
32357, 10922, 35846, 502, 45408, 45955, 37016, 49772, 57814,
6896, 60782, 68281]
reviews_arr 包含的第一条评论的编码前 100 个元素：
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0.
 0. 0. 0. 0.]
```

10.1.2 神经网络模型（IMDB 数据）

至此，我们已经完成了图 10-2 建模过程中的虚线方框部分。

1. 把文字数据转化成数字；
2. 经过变换，每一条评论的编码长度都是一致的；
3. 对输出数据进行 0，1 编码。

现在已经准备好建立神经网络分类模型。我们将建立具有 1 个隐藏层的神经网络。隐藏层和输出层的激活函数都是 sigmoid 函数。输入层的节点数为评论中出现过的单词数量，74074，隐藏层节点数为 20，输出层节点数为 1。在这里，从输入层到隐藏层没有截距项。图 10-3 显示了该神经网络模型的输入层、隐藏层、输出层的节点数、输入层到隐藏层的权重、隐藏层到输出层的权重和截距项，以及隐藏层和输出层的激活函数。

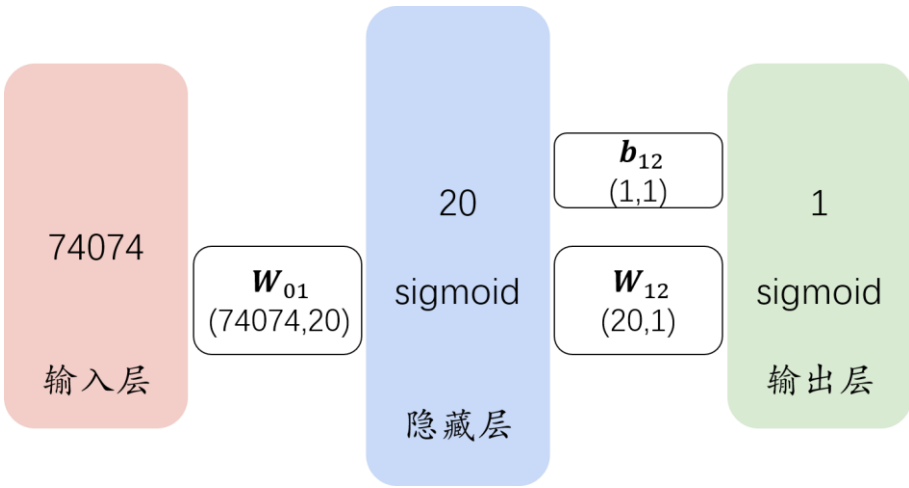

图 10-3 神经网络模型，隐藏层节点数为 20，输入层到隐藏层没有截距项

下面的代码实现了图 10-3 所示的神经网络模型。

```
np.random.seed(1)

def sigmoid(x):
    return 1.0/(1.0 + np.exp(-x))

def sigmoid2deriv(output):
    return output * (1-output)

train_reviews, train_targets = reviews[:20000], targets[:20000]
test_reviews, test_targets = reviews[-5000:], targets[-5000:]

lr, epochs = 0.1, 2
hidden_size = 20

weights_0_1 = np.random.normal(loc=0, scale=0.1, \
```

```
                                    size=(len(vocab), hidden_size))
weights_1_2 = np.random.normal(loc=0, scale=0.1, \
                                    size=(hidden_size, 1))
b_1_2 = 0
for e in range(epochs):
    train_correct_cnt = 0
    for i in range(len(train_reviews)):
        layer_0, target = train_reviews[i], train_targets[i]
        #-------------------------------
        # 使用函数 np.sum 计算 layer_1，而不是矩阵乘法
        layer_1 = sigmoid(np.sum(weights_0_1[layer_0], axis=0,\
                                              keepdims=True))
        layer_2 = sigmoid(np.dot(layer_1, weights_1_2) + b_1_2)

        if (np.abs(layer_2 - target) < 0.5):
            train_correct_cnt += 1

        layer_2_delta = layer_2 - target
        layer_1_delta = layer_2_delta.dot(weights_1_2.T) * \
                                           sigmoid2deriv(layer_1)

        b_1_2 -= lr * layer_2_delta
        weights_1_2 -= lr * layer_1.T.dot(layer_2_delta)
        #-------------------------------
        # 更新 weights_0_1
        weights_0_1[layer_0] -= lr * layer_1_delta

    val_correct_cnt = 0
    for i in range(len(test_reviews)):
        layer_0, target = test_reviews[i], test_targets[i]
        layer_1 = sigmoid(np.sum(weights_0_1[layer_0], axis=0,\
                                                keepdims=True))
        layer_2 = sigmoid(np.dot(layer_1, weights_1_2) + b_1_2)
        if (np.abs(layer_2 - target) < 0.5):
            val_correct_cnt += 1
    print("Train_acc: %0.3f; Val_acc: %0.3f" % \
          (train_correct_cnt/len(train_reviews), \
           val_correct_cnt / len(test_reviews)))
Train_acc: 0.831; Val_acc: 0.856
Train_acc: 0.918; Val_acc: 0.844
```

代码的大部分与之前章节学习的内容是一样的；不同点有两个：第一是计算 `layer_1` 的方式，第二是更新权重矩阵 `weights_0_1` 的方式。为了更方便比较不同点，我们把现在和之前学习的代码不一样的地方列出来。

- 下面是以前计算 `layer_1` 的方式。`layer_0` 为 `reviews_arr` 的一行，维度为1×74074。然后使用矩阵乘法得到 `layer_1`。

```
train_reviews_arr = reviews_arr[:20000]
train_targets = targets[:20000]
layer_0, target = train_reviews_arr[1:2], train_targets[1:2]

# 使用函数矩阵乘法计算 layer_1
layer_1 = sigmoid(np.dot(layer_0, weights_0_1))
```

- 下面是现在计算 layer_1 的方式。现在让 reviews 作为输入数据。还记得吗？reviews 是一个列表，列表元素包含对应评论所有单词的编码。layer_0 为列表的一个元素，然后取 weights_0_1 中 layer_0 对应的列，对 weights_0_1[layer_0]列求和。

```
train_reviews, train_targets = reviews[:20000], targets[:20000]
layer_0, target = train_reviews[1], train_targets[1]
# 使用函数 np.sum 计算 layer_1，而不是矩阵乘法
layer_1 = sigmoid(np.sum(weights_0_1[layer_0], axis=0,
                                                keepdims=True))
```

上面两种方式的计算结果是完全一样的。如图 10-4 所示，一个行向量乘以一个矩阵，行向量大部分都是 0，有些位置为 1；那么，相乘的结果等同于在矩阵中，取出与行向量为 1 的元素对应的行（例如，图 10-4 中 weights_0_1 矩阵带有阴影的两行），然后按列求和。

$$\begin{pmatrix}0 & 1 & 0 & 0 & 1 & 0\end{pmatrix} \times \begin{pmatrix}13 & 27 & -5 & 10 & 1 & 2\\ 2 & -1 & 3 & 0 & 1 & 2\\ 10 & 5 & 17 & 24 & 2 & 3\\ 1 & 25 & 14 & 3 & 3 & 21\\ 3 & 2 & 1 & 0 & 4 & 5\\ 1 & 12 & 12 & 31 & 1 & 0\end{pmatrix} = \begin{pmatrix}5 & 1 & 4 & 0 & 5 & 7\end{pmatrix}$$

layer_0　　weights_0_1　　layer_1

图 10-4 快速计算矩阵乘法

上面两种方式的区别在于计算速度。假设行向量和矩阵的维度分别为$1 \times m$，$m \times n$，通常m很大。

- 如果对行向量和矩阵进行矩阵乘法，那么需要$m \times n$次乘法和$m \times n$次加法。
- 如果使用现在的做法，只需要$k \times n$次加法，而且$k \ll m$，k为向量中不为 0 的元素个数。

下面的代码分别采用两种方式计算整个训练数据的 layer_1，并使用 Jupyter Notebook 的魔法功能%%time 记录两种方法分别的运行时间。可以看到，两种方法的计算结果确实完全相同，但是以前的方式运行时间是现在方式的 10 倍以上。

```
%%time
layer_1_old = np.zeros((20000, 20))
for i in range(20000):
    layer_0, target = train_reviews_arr[i:i+1], \
                                        train_targets[i:i+1]
```

```
    layer_1_old[i] = sigmoid(np.dot(layer_0, weights_0_1))
Wall time: 14.5 s

%%time
layer_1_new = np.zeros((20000, 20))
for i in range(20000):
    layer_0, target = train_reviews[i], train_targets[i]
    layer_1_new[i] = sigmoid(np.sum(weights_0_1[layer_0], \
                                    axis=0, keepdims=True))
Wall time: 960 ms

# 判断两种方式的计算结果是否相同
np.allclose(layer_1_old, layer_1_new)
True
```

接着考虑更新权重矩阵 `weights_0_1` 的方式。同样的，首先把以前的和现在的方式都列出来。

- 下面是以前更新 `weights_0_1` 的方式。注意，在下面代码中，`layer_0` 为 `train_reviews_arr` 的一行。采用矩阵乘法的方式计算 `weights_0_1` 梯度的过程如图 10-5 所示。从计算结果可以看到，`weights_0_1` 的梯度大部分都是 0，只有 `layer_0` 为 1 的元素对应的两行不为 0。因此，在这一次迭代中，`weights_0_1` 只有两行权重不为 0，且两行权重的梯度是一样的，其他行的梯度都为 0。

```
layer_0, target = train_reviews_arr[1:2], train_targets[1:2]
weights_0_1 -= lr*layer_0.T.dot(layer_1_delta)
```

$$\begin{pmatrix}0\\1\\0\\0\\1\\0\end{pmatrix} \times \begin{pmatrix}0.2 & 2 & 0.1 & 3 & 0.9 & 0.5\end{pmatrix} = \begin{pmatrix}0 & 0 & 0 & 0 & 0 & 0\\0.2 & 2 & 0.1 & 3 & 0.9 & 0.5\\0 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 0 & 0 & 0\\0.2 & 2 & 0.1 & 3 & 0.9 & 0.5\\0 & 0 & 0 & 0 & 0 & 0\end{pmatrix}$$

layer_0.T　　layer_1_delta　　gradient of weights_0_1

图 10-5 计算权重矩阵W_{01}的梯度

- 下面是现在更新 `weights_0_1` 的方式。在下面代码中，`layer_0` 为 `train_reviews` 列表的一个元素。从以前计算 `weights_0_1` 的梯度的结果已知 `weights_0_1` 的梯度只有 `layer_0` 的元素对应的行不为 0，且 `weights_0_1` 的梯度不为 0 的行的梯度都等于 `layer_1_delta`。基于这些观察，更新 `weights_0_1` 时，只需要针对 `weights_0_1` 中 `layer_0` 元素对应的行，让这些行减 `layer_1_delta` 和 `lr` 的乘积。这两种更新 `weights_0_1` 方式的结果完全相同，区别在于现在的方式计算速度要快得多。

```
layer_0, target = train_reviews[1], train_targets[1]
weights_0_1[layer_0] -= lr * layer_1_delta
```

10.2 词嵌入

在 10.1 节中，我们为 IMDB 模型建立了一个神经网络模型。在模型训练过程中，使用计算复杂度较少的方式计算 layer_1。现在，我们再次审视权重矩阵 weights_0_1。在模型中，单词通过 one-hot 编码；在编码向量中，只有单词编码对应的位置为 1，其余的元素值都为 0。

从图 10-6 可以看到，当 layer_0 表示一个单词时，layer_1 即为权重矩阵 weights_0_1 中单词编码对应的行。神经网络模型计算得到 layer_1 之后，再通过 layer_1 计算输出层判断该单词所代表的情绪状态。从这个角度来说，layer_1 包含了 layer_0 所代表的信息。或者说，在 IMDB 数据中，通过训练模型，我们希望模型的长度为 20 的 layer_1 可以表示长度为 74074 的 layer_0 所蕴含的信息，进而通过 layer_1 预测评论情绪。因此可以认为 layer_1 也是该单词的编码。这时（输入数据为一个单词 one-hot 编码），layer_0 和 layer_1 都是单词的编码，它们都蕴含了单词的信息。直观上，layer_0 和 layer_1 的不同之处有如下两点。

- layer_0 为离散型的编码（元素只可以取 0 和 1），而 layer_1 为连续型的（元素可以是任意实数）。
- layer_0 编码向量长度很长（在 IMDB 数据中，编码向量等于单词的总数，74074），而 layer_1 的维度通常较小（在 IMDB 数据中，layer_1 的维度为 20）。在实际中，即使单词数量很多，layer_1 的维度为 200-300 通常也足够了。

$$\begin{pmatrix}0 & 1 & 0 & 0 & 0 & 0\end{pmatrix} \times \begin{pmatrix}13 & 27 & -5 & 10 & 1 & 2\\ 2 & -1 & 3 & 0 & 1 & 2\\ 10 & 5 & 17 & 24 & 2 & 3\\ 1 & 25 & 14 & 3 & 3 & 21\\ 3 & 2 & 1 & 0 & 4 & 5\\ 1 & 12 & 12 & 31 & 1 & 0\end{pmatrix} = \begin{pmatrix}2 & -1 & 3 & 0 & 1 & 2\end{pmatrix}$$

layer_0　　weights_0_1　　layer_1

图 10-6 词嵌入

所以，任意一个单词都能在矩阵 weights_0_1 中找到对应的行，作为该单词的编码。这个过程称为**词嵌入**。layer_1 的维度称为词嵌入维度。我们也可以把权重矩阵看成一个查找表（look-up table），根据单词编号可以在表中找到对单词的编码，如图 10-7 所示。

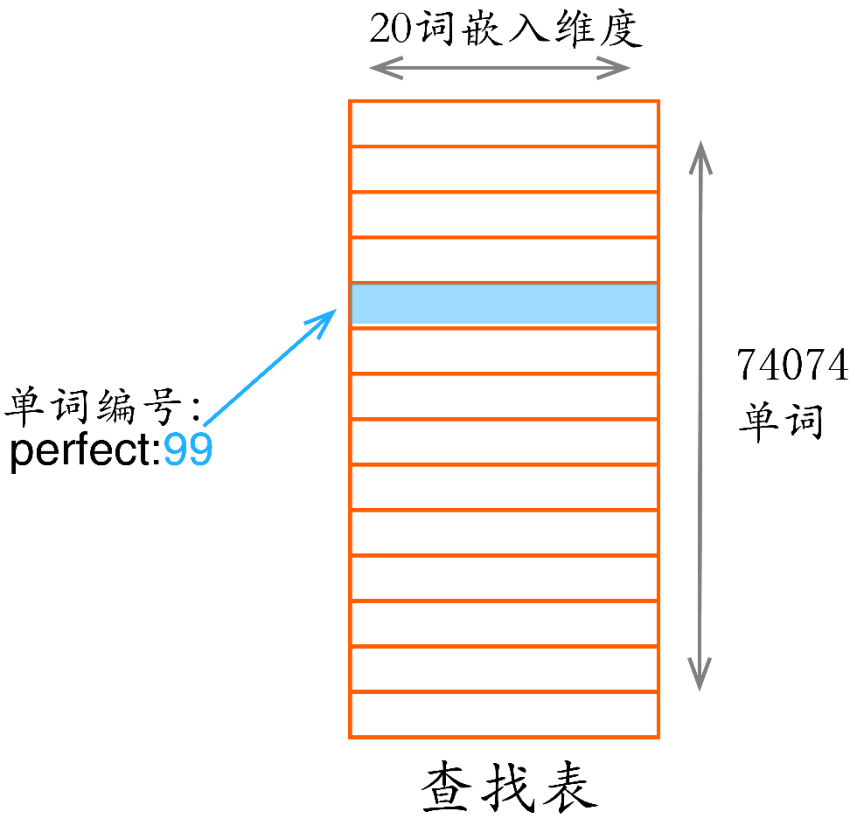

图 10-7 词嵌入-查找表

我们已经看到词嵌入的一个优点，它可以用较小的维度对单词进行编码。这样可以有效地表达单词蕴含的信息。其实，词嵌入还有另外一个优点，相似意思的单词可能有相近的向量表示。为了验证这一点，我们先确定一个目标单词，比如“perfect”，然后找出与“perfect”的向量最近的单词，观察这些单词是否意思与“perfect”相近。这里采用欧式距离衡量向量的“相近”程度，即$\boldsymbol{v}$和$\boldsymbol{u}$的距离定义为$d = \sqrt{\sum(v_i - u_i)^2}$。

下面代码定义函数 `similar()`，该函数可以找出与目标单词编码距离最短的 10 个单词。代码需要用到函数包 collections 的类 Counter，Counter 可以方便对距离进行排序。

```
from collections import Counter

def similar(target='perfect'):
    target_index = word2index[target]  # target 的编码
    scores = Counter()
    for word, index in word2index.items():
        # 计算所有单词与 target 的距离
        raw_diff = weights_0_1[index] - \
                        weights_0_1[target_index]
        squared_diff = raw_diff * raw_diff
```

```
            scores[word] = - np.sqrt(np.sum(squared_diff))
        return scores.most_common(10)
print(similar('perfect'))

[('perfect', -0.0), ('amazing', -0.6626543443142086),
('incredible', -0.7645675403756697), ('superb', -
0.7806366585028585), ('excellent', -0.7903178732866969),
('loved', -0.8276581288623271), ('wonderful', -
0.8366806442056482), ('highly', -0.8808036787802515),
('enjoyable', -0.9099089096747288), ('favorite', -
0.9140890439208232)]

print(similar('terrible'))

[('terrible', -0.0), ('poorly', -0.7436572656539531), ('lame',
-0.7515908855558356), ('disappointing', -0.776169872113042),
('mess', -0.7905753693182044), ('fails', -0.794331307577956),
('dull', -0.7980887062553197), ('avoid', -0.8158677823569627),
('worse', -0.8372545413595637), ('save', -0.8495447242356247)]
```

可以看到，与“perfect”单词最相近的单词有“amazing”，“incredible”，“superb”，“excellent”，“loved”，“wonderful”等，在词义上，这些单词也与“perfect”相似；与“terrible”最相近的有“poorly”，“lame”，“disappointing”，“mess”，“fails”，“dull”等，在词义上这些单词也与“terrible”相似。

为什么词嵌入会有这样的效果呢？我们先回忆一下刚刚建立的图 10-3 所示的神经网络模型。在模型训练过程中，总的目标是使得损失函数$\sum -(y_i \log(p_i) + (1-y_i)\log(1-p_i))$变小。也就是说，当$y_i = 1$或者评论的情绪是正面时，要让$p_i$尽可能大；当$y_i = 0$或者评论的情绪是负面时，要让$p_i$尽可能小。一种可能的情况是，包含褒义词的评论都得到相近的 `layer_1`，这些相近的 `layer_1` 都得到比较大的p_i；同样的，包含贬义词的评论也都得到相近的 `layer_1`，这些相近的 `layer_1` 都得到比较小的p_i。让包含褒义词的评论都得到相近的 `layer_1` 的一种方式就是让褒义词具有相近的词嵌入；让包含贬义词的评论都得到相近的 `layer_1` 的一种方式就是让贬义词具有相近的词嵌入。也就是说，如果两个单词表达了相似的情绪，那么神经网络模型就有较大可能会让这两个单词有相近的词嵌入。

10.3 循环神经网络

10.3.1 引入循环神经网络

在本节中，我们将学习循环神经网络（Recurrent Neural Network, RNN）。先回顾 10.2 节的词嵌入，以及建立的电影评论情绪分析神经网络模型。我们依然关注神经网络模型中词嵌入部分，即图 10-8 的虚线框部分。

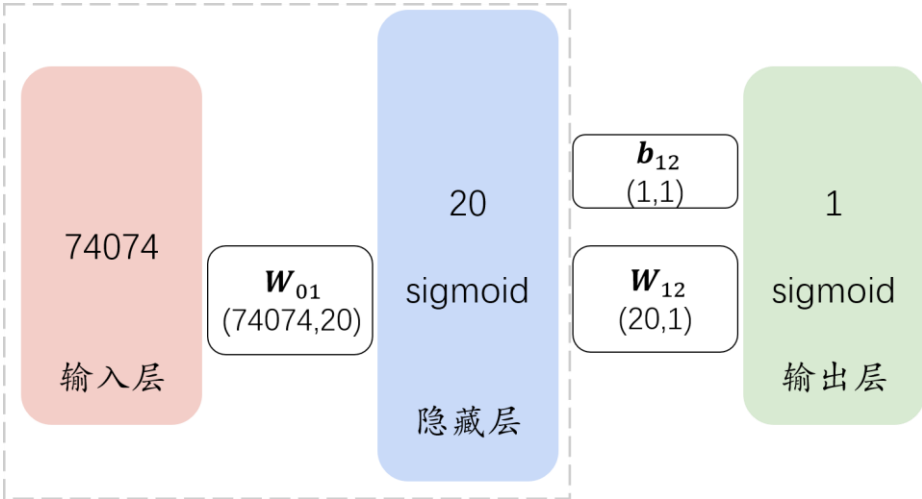

图 10-8 简单神经网络-词嵌入

为了更好地理解图 10-8 虚线框的运算，考虑一个简单例子。该例子共有 5 个单词，分别是["I","love","deep","learning","story"]，5 个单词的编码分别是[0,1,2,3,4]，接着对 5 个单词做 one-hot 编码，且假设$\boldsymbol{W}_{01}$的维度为5×3。那么，在隐藏层中，对句子“I love deep learning”的编码就是矩阵$\boldsymbol{W}_{01}$的前 4 行按列求和，即$\boldsymbol{W}_{01}[0,] + \boldsymbol{W}_{01}[1,] + \boldsymbol{W}_{01}[2,] + \boldsymbol{W}_{01}[3,]$，如图 10-9 所示。

这个模型实质还是普通的神经网络模型，只是在建模过程中，增加了对文字信息的处理。该模型有一个潜在缺陷：没有考虑句子的单词顺序。例如，“deep learning love I”的编码和“I love deep learning”是一样的。但是，我们希望神经网络模型能够考虑句子的单词顺序，因为单词顺序可能影响最终预测，以及在更一般情况下，考虑数据的序列信息。

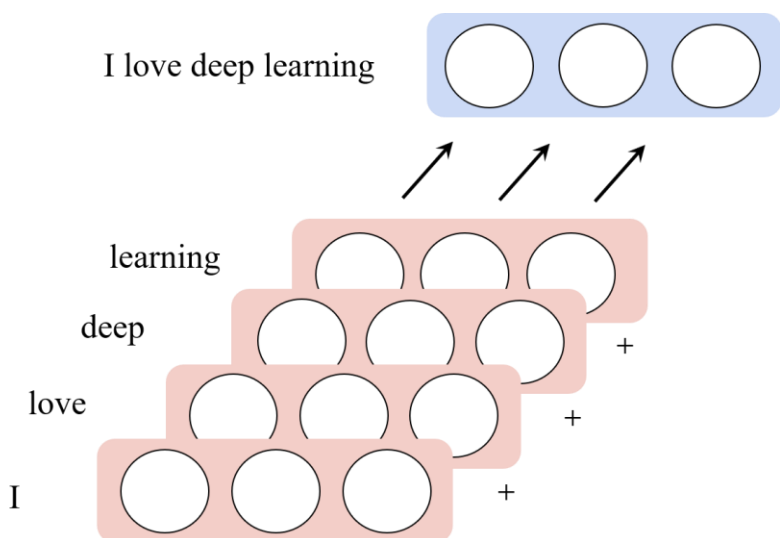

图 10-9 句子的表示，词嵌入求和

我们先对图 10-9 的计算做一个看似无用的变换。在图 10-9 中，“I love deep learning”的编码为$\boldsymbol{W}_{01}[0,]+\boldsymbol{W}_{01}[1,]+\boldsymbol{W}_{01}[2,]+\boldsymbol{W}_{01}[3,]$。而在图 10-10 中，我们让“I”的词嵌入乘以一个单位矩阵（Identity Matrix），再加上“love”的词嵌入，所得到的和再乘以单位矩阵，等等。在这个简单例子中，词嵌入维度为 3，单位矩阵为

$$\boldsymbol{I}=\boldsymbol{I}_3=\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}$$

根据图 10-10，“I love deep learning”的编码为$((\boldsymbol{W}_{01}[0,]\boldsymbol{I}_3+\boldsymbol{W}_{01}[1,])\boldsymbol{I}_3+\boldsymbol{W}_{01}[2,])\boldsymbol{I}_3+\boldsymbol{W}_{01}[3,]$。任何矩阵乘以单位矩阵都等于其本身，因此，图 10-10 和图 10-9 所示的两种计算方式的结果是完全一样的。

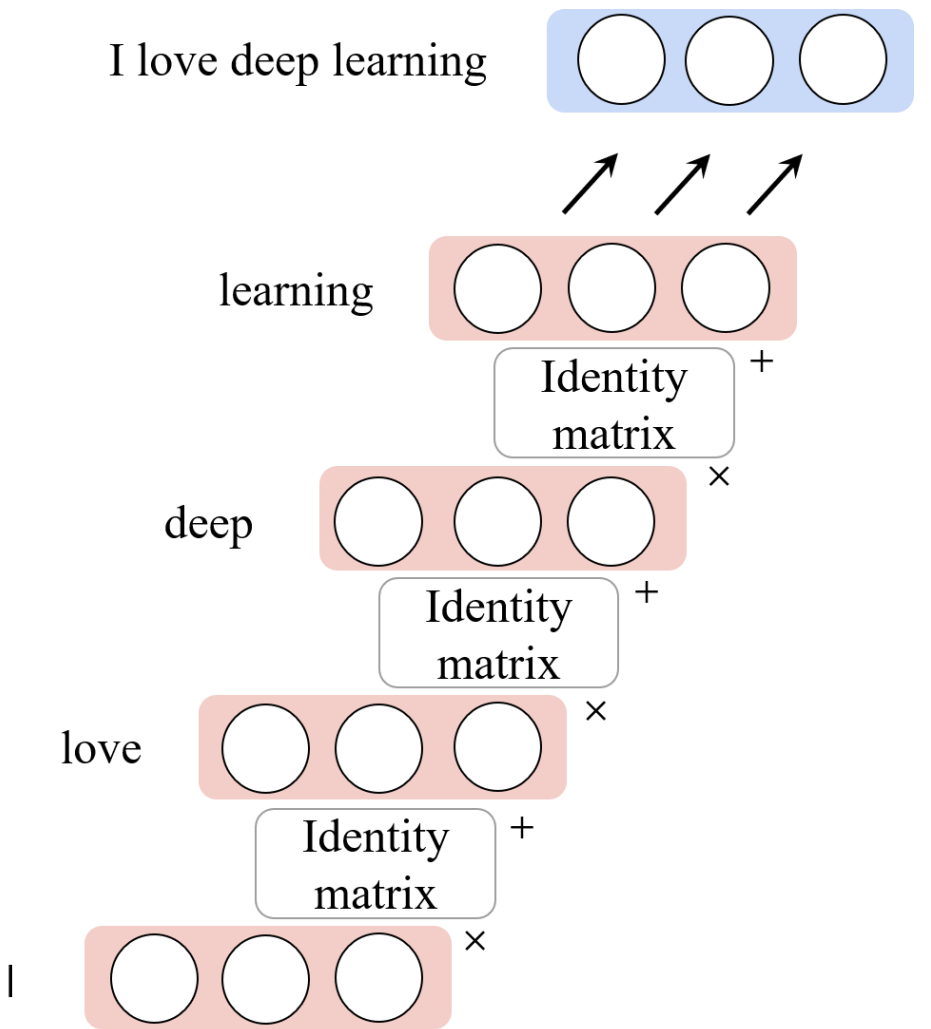

图 10-10 句子的表示，词嵌入乘以单位矩阵后再求和

我们也可以把图 10-10 所示的计算过程用图 10-11 表示。在图 10-11 中，隐藏层矩形框从左到右分别记为$\boldsymbol{h}_0$，$\boldsymbol{h}_1$，$\boldsymbol{h}_2$，$\boldsymbol{h}_3$，$\boldsymbol{h}_4$，其中，$\boldsymbol{h}_0$是一个元素全为 0 的矩阵。在这个简单例子中，$\boldsymbol{h}_0$为一个维度为 3 的行向量，即$\boldsymbol{h}_0 = (0,0,0)$。图 10-11 的计算方式为，

- $\boldsymbol{h}_1 = \boldsymbol{h}_0\boldsymbol{I}_3 + \boldsymbol{W}_{01}[0,]$
- $\boldsymbol{h}_2 = \boldsymbol{h}_1\boldsymbol{I}_3 + \boldsymbol{W}_{01}[1,]$
- $\boldsymbol{h}_3 = \boldsymbol{h}_2\boldsymbol{I}_3 + \boldsymbol{W}_{01}[2,]$
- $\boldsymbol{h}_4 = \boldsymbol{h}_3\boldsymbol{I}_3 + \boldsymbol{W}_{01}[3,]$

句子“I love deep learning”的编码为$\boldsymbol{h}_4$。同样的，$\boldsymbol{h}_4$等于图 10-9 的计算结果。

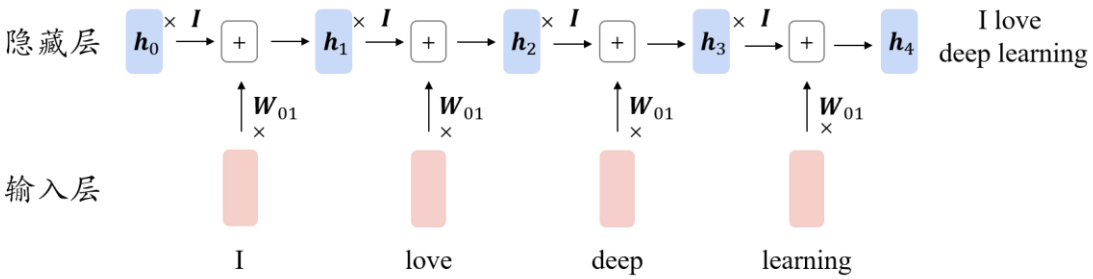

图 10-11 句子的表示，词嵌入乘以单位矩阵后再求和（第二种表示方式）

可以看到$\boldsymbol{h}_4$等于神经网络模型（图 10-8）的隐藏层。在图 10-11 中，增加一个输出层，即为一个完整的神经网络。图 10-3 所示的具有 1 个隐藏层的神经网络模型也可以用图 10-12 的形式表示。

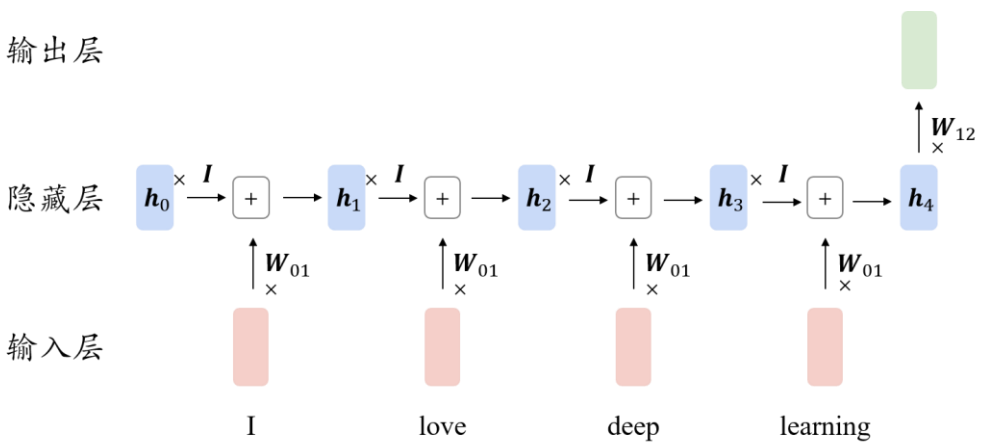

图 10-12 神经网络模型，图 10-3 所示神经网络的另一种表示方式

到现在为止，我们用图 10-9、图 10-10、图 10-11 表示了同一个句子的编码。用图 10-8 和图 10-12 表示同一个神经网络模型。根据上面的分析，我们知道该模型无法考虑序列的关系。例如，输入“I love deep learning”和“deep learning

love I”的预测结果将是一样的。现在，我们把上面模型进一步推广，使模型可以考虑数据的序列关系。图 10-13 表示了我们的第一个循环神经网络，总体结构与图 10-12 类似，但是有 3 个不同点。

1. 在图 10-12 中，$\boldsymbol{h}_0$，$\boldsymbol{h}_1$，$\boldsymbol{h}_2$，$\boldsymbol{h}_3$都乘以一个单位矩阵；但是，在图 10-13 中，$\boldsymbol{h}_0$，$\boldsymbol{h}_1$，$\boldsymbol{h}_2$，$\boldsymbol{h}_3$都乘以一个权重矩阵$\boldsymbol{W}_{hh}$，$\boldsymbol{W}_{hh}$不是单位矩阵，将在模型训练中得到。注意，每一个$\boldsymbol{h}_i$，$i=0,1,2,3$都乘以同样的权重矩阵，记为$\boldsymbol{W}_{hh}$（表示从隐藏层到隐藏层权重矩阵，Hidden layer to Hidden layer）。
2. 在图 10-12 中，$\boldsymbol{h}_i$是$\boldsymbol{h}_{i-1}$，$i=1,2,3,4$的一个线性函数，例如，$\boldsymbol{h}_1=\boldsymbol{h}_0\boldsymbol{I}_{3\times3}+\boldsymbol{W}_{01}[0,]$；在图 10-13 中，计算$\boldsymbol{h}_i$的过程中增加了一个激活函数，使得$\boldsymbol{h}_i$是$\boldsymbol{h}_{i-1}$的一个非线性函数。例如，$\boldsymbol{h}_1=f(\boldsymbol{h}_0\boldsymbol{W}_{hh}+\boldsymbol{W}_{01}[0,])$，$f(\cdot)$是一个激活函数。
3. 第 3 个不同点是符号略有不同。为了体现不同权重的功能，在图 10-13 中，我们把$\boldsymbol{W}_{01}$记为$\boldsymbol{W}_{ih}$（表示从输入层到隐藏层的权重，Input layer to Hidden layer），$\boldsymbol{W}_{12}$记为$\boldsymbol{W}_{ho}$（表示从隐藏层到输出层权重，Hidden layer to Output layer）。

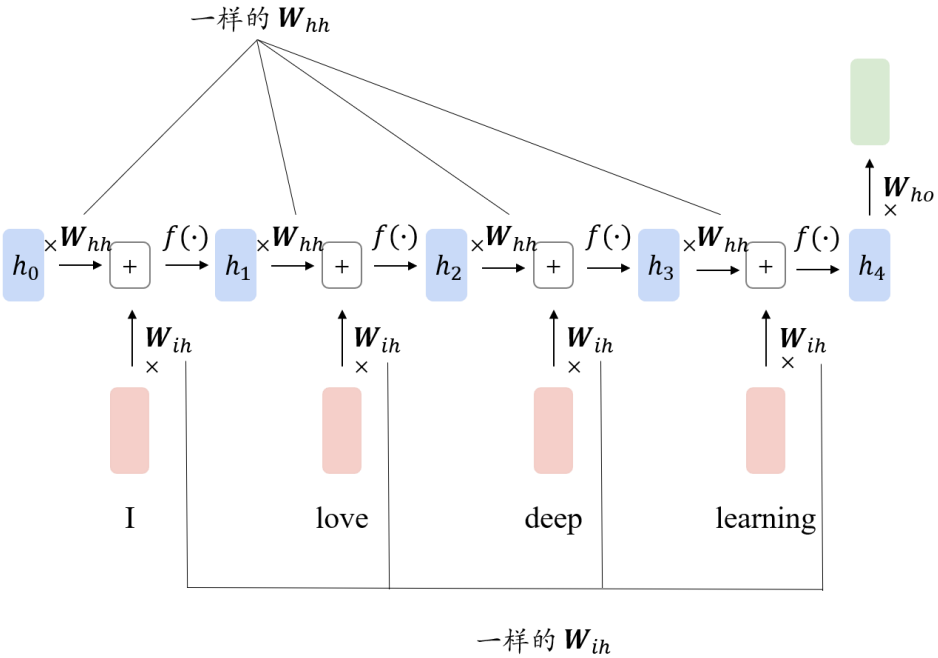

图 10-13 循环神经网络

下面列出了该循环神经网络模型的计算过程。在 RNN 模型中，通常隐藏层到隐藏层和隐藏层到输出层的运算都包含截距项，$\boldsymbol{b}_{hh}$和$\boldsymbol{b}_{ho}$。为了更加简洁，图 10-13 省略了截距项$\boldsymbol{b}_{hh}$和$\boldsymbol{b}_{ho}$。

- $\boldsymbol{h}_1 = f(\boldsymbol{h}_0\boldsymbol{W}_{hh} + \boldsymbol{x}_0\boldsymbol{W}_{ih} + \boldsymbol{b}_{hh})$
- $\boldsymbol{h}_2 = f(\boldsymbol{h}_1\boldsymbol{W}_{hh} + \boldsymbol{x}_1\boldsymbol{W}_{ih} + \boldsymbol{b}_{hh})$
- $\boldsymbol{h}_3 = f(\boldsymbol{h}_2\boldsymbol{W}_{hh} + \boldsymbol{x}_2\boldsymbol{W}_{ih} + \boldsymbol{b}_{hh})$
- $\boldsymbol{h}_4 = f(\boldsymbol{h}_3\boldsymbol{W}_{hh} + \boldsymbol{x}_3\boldsymbol{W}_{ih} + \boldsymbol{b}_{hh})$
- $\text{output} = \text{sigmoid}(\boldsymbol{h}_4\boldsymbol{W}_{ho} + \boldsymbol{b}_{ho})$

其中，$\boldsymbol{x}_0$，$\boldsymbol{x}_1$，$\boldsymbol{x}_2$，$\boldsymbol{x}_3$分别为“I love deep learning”4 个单词的 one-hot 编码。

因为$\boldsymbol{W}_{hh}$不再是单位矩阵，且引入了激活函数$f(\cdot)$；“ I love deep learning”的编码将会与“deep learning love I”不同。图 10-13 所示的模型即循环神经网络模型，RNN！在循环神经网络中，$\boldsymbol{h}_0$，$\boldsymbol{h}_1$，$\boldsymbol{h}_2$，$\boldsymbol{h}_3$，$\boldsymbol{h}_4$都称为隐藏层。

有时候，为了方便，我们也会把 RNN 示意图画成如图 10-14 的简洁形式。图 10-14 表示多个输入数据按顺序输入到模型中，最后得到一个输出值。就像电影评论数据中一样，输入评论的多个单词后，得到一个电影评论的预测值。

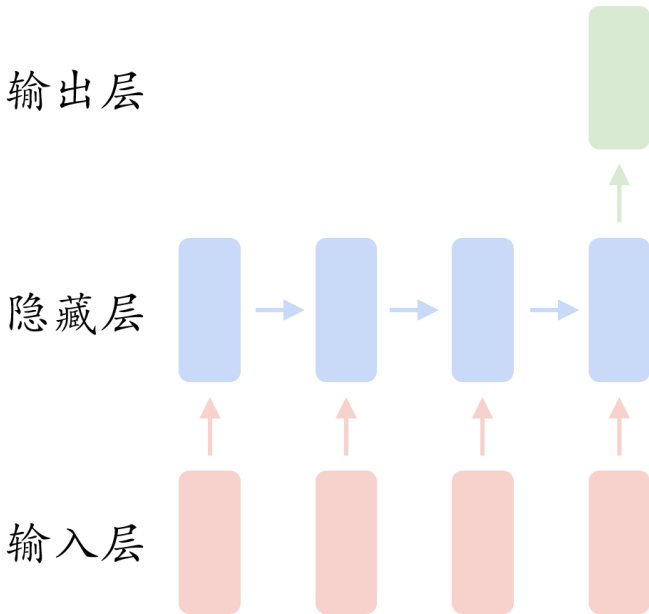

图 10-14 循环神经网络（简洁形式）

在实际应用中，根据任务不同，RNN 模型有多种不同的形式。下面列出了 5 种较为常见的形式。

（1）一个输入到一个输出（one to one），RNN 退化为普通的神经网络。

（2）一个输入到多个输出（one to many），例如输入是一幅图片，输出为图片的名称（名称为不固定长度的短语或者句子）。

（3）多个输入到一个输出（many to one），例如，电影评论数据，输入为评论（长度不固定，且序列很重要的句子），输出为评论代表的情绪。

（4）多个输入到多个输出（many to many），例如，机器翻译，输入为一句话（长度不固定，且序列很重要的句子），整个句子输入之后，输出是另一种语言的一句话（长度不固定，且序列很重要的句子）。

（5）多个输入到多个输出（many to many），例如，识别视频中出现的物体，输入为一长段视频，输出为每一小段视频中出现的物体。

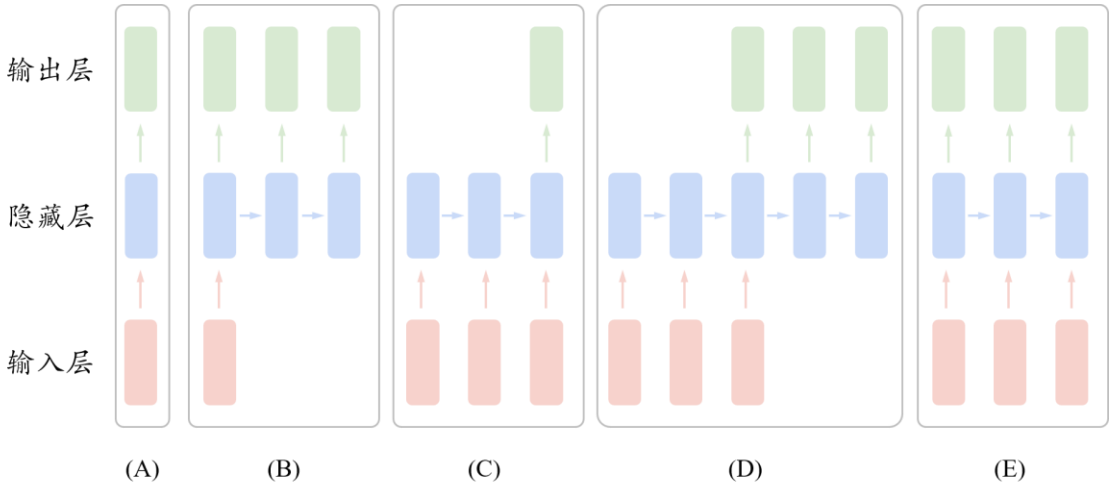

图 10-15 5 种循环神经网络

10.4 从零开始实现 RNN

根据任务不同，RNN 的结构也各异。但是，总的来说，无论是哪种结构的 RNN，其基本原理和计算方法都是类似的。在这里，我们将以图 10-15(E)所示的 RNN 结构为例学习 RNN 的正向传播算法、反向传播算法以及梯度下降法等模型训练过程。

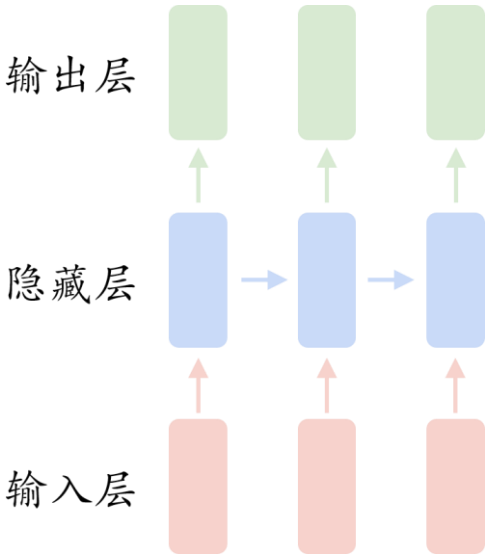

图 10-16 循环神经网络（多个输入对多个输出）

10.4.1 莎士比亚作品

在本节中，我们尝试一个非常具有挑战性的任务：像莎士比亚一样写作！具体来说，我们希望模型可以通过一些给定字符（字母、标点等符号），预测下一个字符。训练模型时，使用莎士比亚作品作为输入数据训练模型，因此，也希望该模型的输出可以具备莎士比亚的风格。

首先，读入莎士比亚作品。下面列出了莎士比亚作品的前 200 个字符，可以看到文本是以戏剧剧本形式呈现的。

```
np.random.seed(1)
with open("./data/shakespeare.txt", 'r') as file:
    data = file.read()
print(data[0:200])

First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you
```

接着简单处理数据。chars 包含了作品中出现的 65 个字符，包括英文大小写字母、标点符号、空格符、换行符等。载入的莎士比亚作品包含了 1115394 个字符。同时，我们还为每个字符创建了编号，得到两个字典型数据：char_to_indx 和 index_to_char。char_to_indx 把字符对应到编码；index_to_char 把编码对应到字符。

```
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print ('Data has %d characters, and %d unique ones.' % \
                                        (data_size, vocab_size))
char_to_index = { ch:i for i,ch in enumerate(chars) }
index_to_char = { i:ch for i,ch in enumerate(chars) }

Data has 1115394 characters, and 65 unique ones.
```

下面列出了 char_to_indx 中的 10 个元素，字母 T 的编号为 0，字母 g 的编号为 1，等等。

```
dict(list(char_to_index.items())[0:10])
{'T': 0,
 'g': 1,
 'M': 2,
 't': 3,
 'h': 4,
 '3': 5,
 'N': 6,
 'o': 7,
 'B': 8,
 'O': 9}
```

在正式开始实现 RNN 模型之前，我们先看一个简单的例子，较为直观地看看什么是根据一些字符预测下一个字符。现在，假设训练数据很简单，只有一个单词，hello。这时，`chars` 为 4 个字符，['h','e','l','o']，这 4 个字符分别编码为{'h':0, 'e':1, 'l':2, 'o':3}，并且 4 个字符的 one-hot 编码分别为

$$'h': \begin{pmatrix}1\\0\\0\\0\end{pmatrix} \quad 'e': \begin{pmatrix}0\\1\\0\\0\end{pmatrix} \quad 'l': \begin{pmatrix}0\\0\\1\\0\end{pmatrix} \quad 'o': \begin{pmatrix}0\\0\\0\\1\end{pmatrix}$$

训练模型时，计算流程如图 10-17 所示。

- 输入字符h，得到输出层各个字母的概率值，然后依据这些概率值，随机抽取一个字符，计算损失函数。
- 接着，输入e，通过h，e得到输出层各个字母的概率值，然后依据这些概率值，随机抽取一个字符，计算损失函数。
- 然后，输入l，通过h，e，l得到输出层各个字母的概率值，然后依据这些概率值，随机抽取一个字符，计算损失函数。
- 最后，输入l，通过h，e，l，l得到输出层各个字母的概率值，然后依据这些概率值，随机抽取一个字符，计算损失函数。

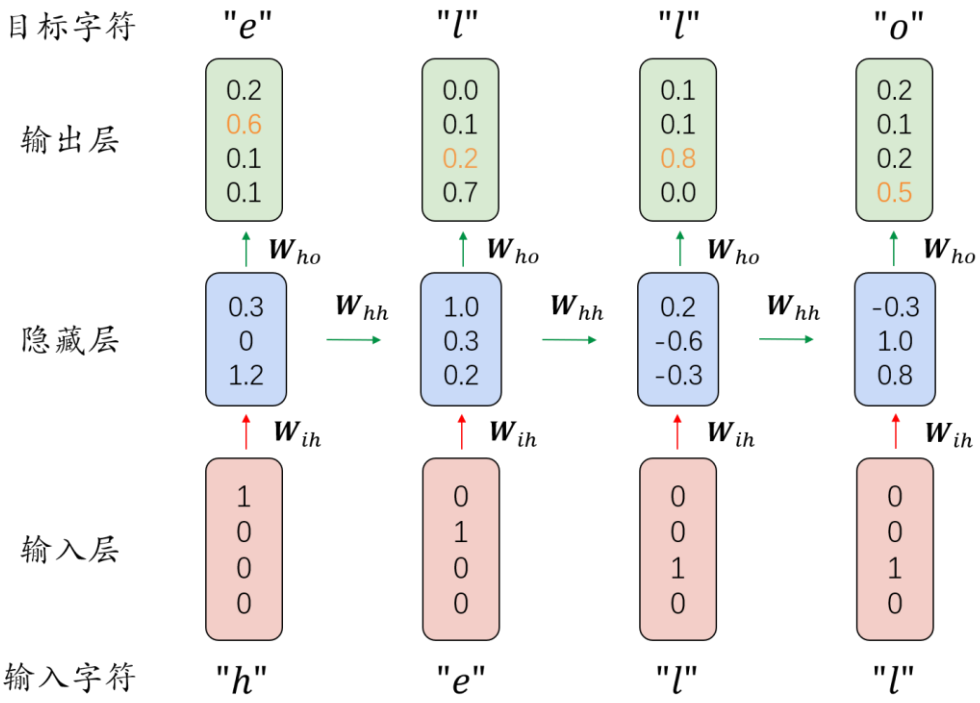

图 10-17 循环神经网络，模型训练

使用模型时，计算流程如图 10-18 所示。

- 输入字符h，得到输出层各个字母的概率值，然后依据这些概率值，随机抽取一个字符，例如，抽取到字符e。
- 接着，输入抽取到的字符e，通过h，e，得到输出层各个字母的概率值，然后依据这些概率值，随机抽取一个字符，例如，抽取到字符o。
- 然后，输入o，通过h，e，o，得到输出层各个字母的概率值，然后依据这些概率值，随机抽取一个字符，例如，抽取到字符l。
- 最后，输入l，通过h，e，o，l，得到输出层各个字母的概率值，然后依据这些概率值，随机抽取一个字符，例如，抽取到字符o。

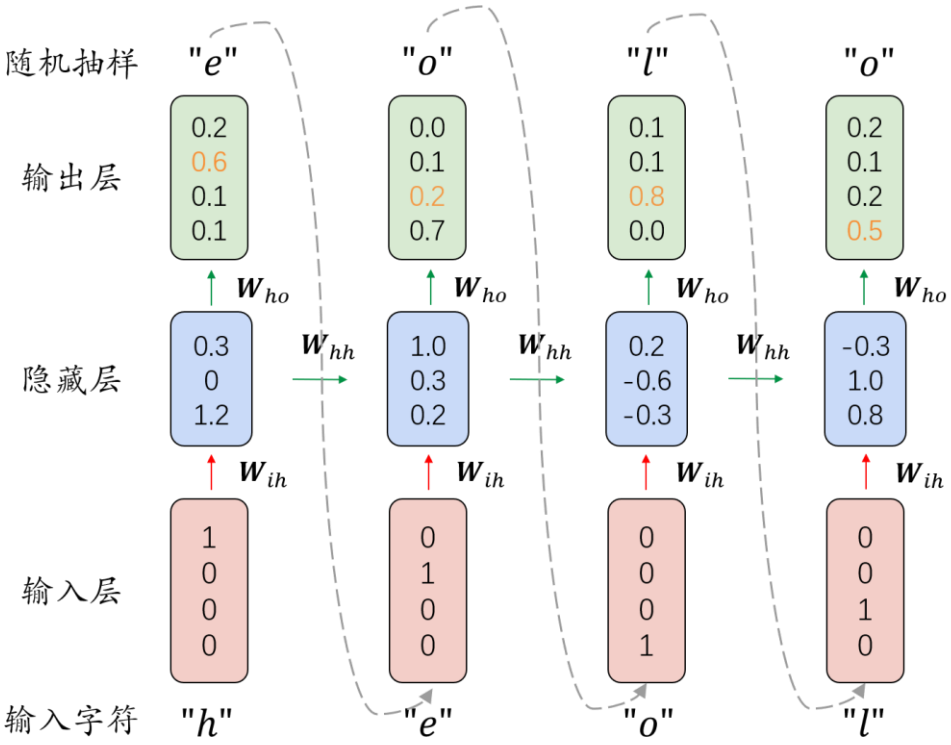

图 10-18 循环神经网络，模型预测

10.4.2 正向传播算法

我们开始学习 RNN 模型的具体训练方法。在图 10-13 的模型中，有 5 个需要训练的参数，3 个权重矩阵和 2 个截距项向量，$\boldsymbol{W}_{ih}$，$\boldsymbol{W}_{hh}$，$\boldsymbol{W}_{ho}$，$\boldsymbol{b}_{hh}$，$\boldsymbol{b}_{ho}$。图 10-19 画出了图 10-13 的循环神经网络正向传播算法计算图。在计算图中，省略了截距项$\boldsymbol{b}_{hh}$和$\boldsymbol{b}_{ho}$。图 10-19 看上去非常复杂，其实只要看明白了虚线框部分便可以明白整个计算图。整个计算图只是重复虚线框的计算 4 次。在简单的例子中，$\boldsymbol{x}_0$，$\boldsymbol{x}_1$，$\boldsymbol{x}_2$，$\boldsymbol{x}_3$分别为h，e，l，l的 one-hot 编码；$\boldsymbol{y}_0$，$\boldsymbol{y}_1$，$\boldsymbol{y}_2$，$\boldsymbol{y}_3$分别为e，l，l，o的 one-hot 编码；h_{-1}为一个维度元素全为 0 的行向量。输入数据为$\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3$，目标数据$\boldsymbol{y}_0, \boldsymbol{y}_1, \boldsymbol{y}_2, \boldsymbol{y}_3$。注意，在这里，初始隐藏层记为$\boldsymbol{h}_{-1}$。

根据图 10-19 计算图，对于简单例子，正向传播算法计算步骤可以用文字描述如下。

- 输入数据$\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3$，目标数据$\boldsymbol{y}_0, \boldsymbol{y}_1, \boldsymbol{y}_2, \boldsymbol{y}_3$，初始隐藏层$\boldsymbol{h}_{-1}$
- 循环，让 $t = 0,1,2,3$
  - 计算隐藏层$\boldsymbol{h}_t$，$\boldsymbol{h}_t = f((\boldsymbol{h}_{t-1}\boldsymbol{W}_{hh} + \boldsymbol{b}_{hh}) + \boldsymbol{x}_t\boldsymbol{W}_{ih})$
  - 计算输出值，$\boldsymbol{o}_t = \text{softmax}(\boldsymbol{h}_t\boldsymbol{W}_{ho} + \boldsymbol{b}_{ho})$

- 计算损失函数，$L_t = \ell(\boldsymbol{y}_t, \boldsymbol{o}_t)$。在莎士比亚写作的例子中，可以采用交叉熵作为损失函数。

- 计算损失函数，损失函数为 4 个输出值$\boldsymbol{o}_0, \boldsymbol{o}_1, \boldsymbol{o}_2, \boldsymbol{o}_3$的损失函数的和，$L = L_0 + L_1 + L_2 + L_3$。

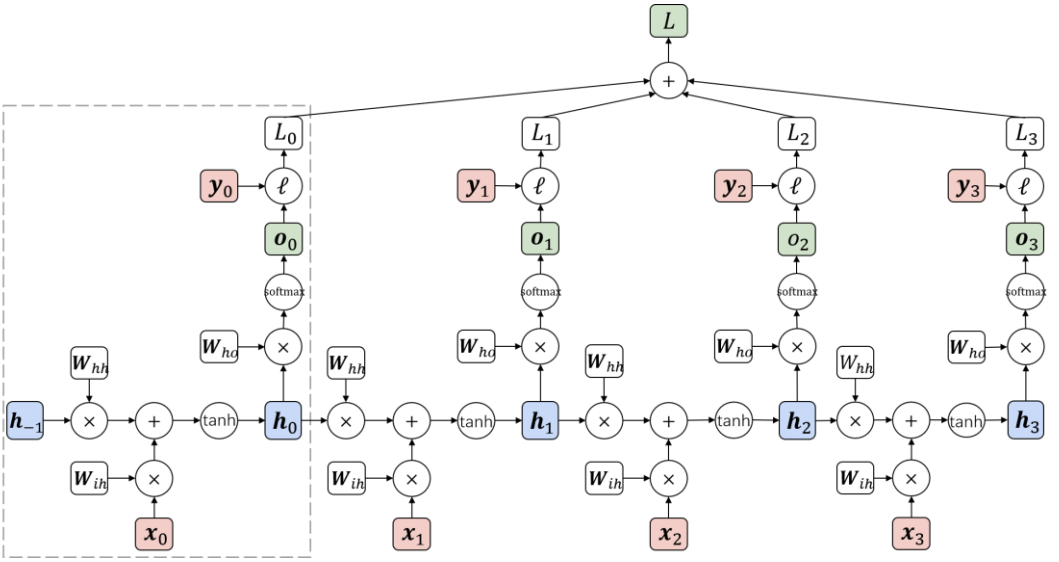

图 10-19 循环神经网络，正向传播算法计算图。从左至右，从下至上看

在下面代码中，逐步定义函数 `lossFun()`，该函数有 3 个参数：

- `inputs`：当前输入的字符编号；
- `targets`：目标字符编号；
- `prev_hidden`：前一个隐藏层状态，初始值为元素全为 0 的向量。

函数 `lossFun()`的第一部分实现了正向传播算法。需要训练的参数有 5 个，$\boldsymbol{W}_{ih}$，$\boldsymbol{W}_{hh}$，$\boldsymbol{W}_{ho}$，$\boldsymbol{b}_{hh}$，$\boldsymbol{b}_{ho}$。$\boldsymbol{W}_{ih}$，$\boldsymbol{W}_{hh}$，$\boldsymbol{W}_{ho}$都从方差为10^{-4}的正态分布产生；$\boldsymbol{b}_{hh}$，$\boldsymbol{b}_{ho}$的初始值都设为 0。这里使用 tanh 函数作为隐藏层激活函数。需要注意的是，图 10-19 的正向传播算法计算图是根据简单例子画的（输入序列长度固定为 4）；在函数 `lossFun()`中，我们对一般的情况进行编程（允许输入序列长度是给定的任意数）。

```
"""
参数初始化
w_ih: 从输入层到隐藏层权重矩阵
w_hh: 从隐藏层到隐藏层权重矩阵
w_ho: 从隐藏层到输出层权重矩阵
b_hh: 从隐藏层到隐藏层截距项
b_ho: 从隐藏层到输出层截距项
"""
hidden_size = 20
w_ih = np.random.randn(vocab_size, hidden_size)*0.01
w_hh = np.random.randn(hidden_size, hidden_size)*0.01
```

```
w_ho = np.random.randn(hidden_size, vocab_size)*0.01
b_hh = np.zeros((1, hidden_size))
b_ho = np.zeros((1, vocab_size))

"""
函数 lossFun()的正向传播算法部分
inputs, targets：输入数据、目标数据，都是整数序列(字符编码)
prev_hidden：隐藏层初始值
返回损失函数、参数梯度以及最后一个隐藏层
"""
def lossFun(inputs, targets, prev_hidden):

    input_states, hidden_states, output_states = {}, {}, {}
    hidden_states[-1] = np.copy(prev_hidden)
    loss = 0

    # 第一部分：正向传播算法
    for t in range(len(inputs)):
        # 字符的 one-hot 编码
        input_states[t] = np.zeros((1, vocab_size))
        input_states[t][0, inputs[t]] = 1
        # 计算隐藏层
        hidden_states[t] = np.tanh(np.dot(input_states[t], w_ih)\
                   + (np.dot(hidden_states[t-1], w_hh) + b_hh))
        # 计算输出层加权值
        logits = np.dot(hidden_states[t], w_ho) + b_ho
        # 计算输出层
        output_states[t] = np.exp(logits) / np.sum(np.exp(logits))
        # 预测误差
        loss += -np.log(output_states[t][0, targets[t]])
```

10.4.3 反向传播算法

现在介绍反向传播算法。只要把正向传播算法计算图的箭号都反方向，即得到反向传播算法的计算图，如图 10-20 所示。

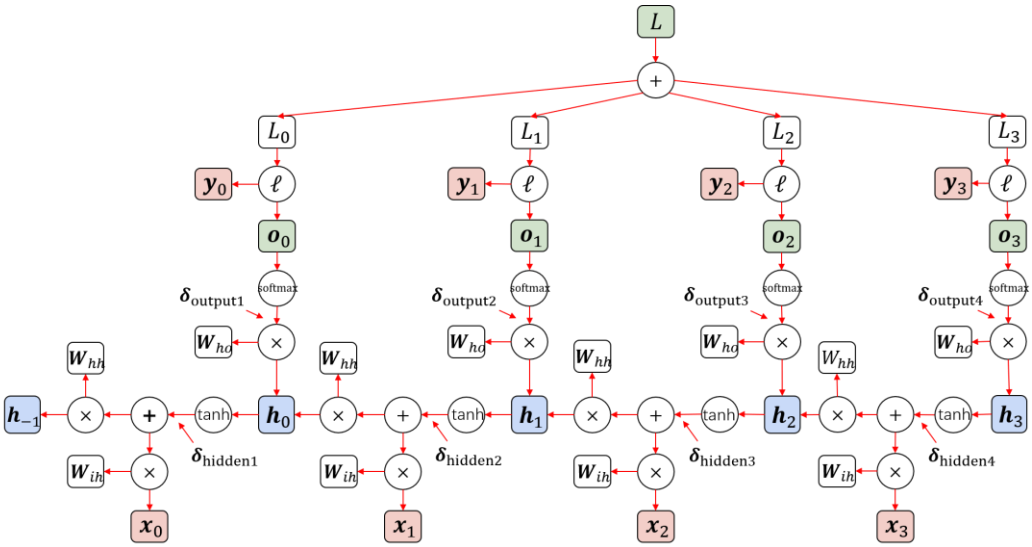

图 10-20 循环神经网络，反向传播算法计算图。从右至左，从上至下看

根据反向传播算法计算图，可以按照如下步骤计算变量$\boldsymbol{W}_{ih}$，$\boldsymbol{W}_{hh}$，$\boldsymbol{W}_{ho}$，$\boldsymbol{b}_{hh}$，$\boldsymbol{b}_{ho}$的梯度。

- $\frac{\partial L}{\partial \boldsymbol{W}_{ho}}$，$\frac{\partial L}{\partial \boldsymbol{b}_{ho}}$，$\frac{\partial L}{\partial \boldsymbol{W}_{hh}}$，$\frac{\partial L}{\partial \boldsymbol{b}_{hh}}$，$\frac{\partial L}{\partial \boldsymbol{W}_{ih}}$的初始值都设为 0，维度分别与$\boldsymbol{W}_{ho}$，$\boldsymbol{b}_{ho}$，$\boldsymbol{W}_{hh}$，$\boldsymbol{b}_{hh}$，$\boldsymbol{W}_{ih}$相同。设置 `grad_hidden_next` 的初始值全为 0，维度与隐藏层的维度相同。
- 循环，让 $t = 3,2,1,0$
  - 计算delta_output_t（图 10-20 中 softmax 函数处的箭号标识）：$\text{delta\_output}_t = \boldsymbol{o}_t - \boldsymbol{y}_t$（假设输出层激活函数为 softmax）
  - 计算L_t关于$\boldsymbol{W}_{ho}$，$\boldsymbol{b}_{ho}$的梯度：$\frac{\partial L_t}{\partial \boldsymbol{W}_{ho}} = \boldsymbol{h}_t^{\top}(\boldsymbol{o}_t - \boldsymbol{y}_t)$和$\frac{\partial L_t}{\partial \boldsymbol{b}_{ho}} = (\boldsymbol{o}_t - \boldsymbol{y}_t)$
  - 更新$\frac{\partial L}{\partial \boldsymbol{W}_{ho}}$和$\frac{\partial L}{\partial \boldsymbol{b}_{ho}}$：$\frac{\partial L}{\partial \boldsymbol{W}_{ho}} \mathrel{+}= \frac{\partial L_t}{\partial \boldsymbol{W}_{ho}}$，$\frac{\partial L}{\partial \boldsymbol{b}_{ho}} \mathrel{+}= \frac{\partial L_t}{\partial \boldsymbol{b}_{ho}}$
  - 计算 L_t关于$\boldsymbol{h}_t$的梯度：$\frac{\partial L_t}{\partial \boldsymbol{h}_t} = (\boldsymbol{o}_t - \boldsymbol{y}_t)\boldsymbol{W}_{ho}^{\top}$
  - 计算 delta_hidden_t（图 10-20 中 tanh 函数处的箭号标识），$\text{delta\_hidden}_t = (\frac{\partial L_t}{\partial \boldsymbol{h}_t} + \text{grad\_hidden\_next}) \circ (1 - \boldsymbol{h}_t^2)$
  - 计算L关于$\boldsymbol{W}_{hh}$，$\boldsymbol{b}_{hh}$，$\boldsymbol{W}_{ih}$的梯度:
    - $\frac{\partial L}{\partial \boldsymbol{W}_{hh}} \mathrel{+}= \boldsymbol{h}_{t-1}^{\top}\text{delta\_hidden}_t$
    - $\frac{\partial L}{\partial \boldsymbol{W}_{ih}} \mathrel{+}= \boldsymbol{x}_t^{\top}\text{delta\_hidden}_t$
    - $\frac{\partial L}{\partial \boldsymbol{b}_{hh}} \mathrel{+}= \text{delta\_hidden}_t$
  - 计算grad_hidden_next，$\text{grad\_hidden\_next} = \text{delta\_hidden}_t \times \boldsymbol{W}_{hh}^{\top}$

在函数 lossFun()的第二部分，根据上面算法编程实现了反向传播算法。为了使梯度下降法迭代更平稳，在函数 lossFun()中，使用函数 np.clip()对参数梯度进行截断，把参数梯度控制在−2到2之间。

```
"""
inputs, targets: 输入数据，目标数据，都是整数序列(字符编码)
prev_hidden: 隐藏层初始值
返回损失函数，参数梯度，最后一个隐藏层
"""
def lossFun(inputs, targets, prev_hidden):

    input_states, hidden_states, output_states = {}, {}, {}
    hidden_states[-1] = np.copy(prev_hidden)
    loss = 0

    # 第一部分：正向传播算法
    for t in range(len(inputs)):
        input_states[t] = np.zeros((1, vocab_size))
        input_states[t][0, inputs[t]] = 1
        hidden_states[t] = np.tanh(np.dot(input_states[t], w_ih)
                    + (np.dot(hidden_states[t-1], w_hh) + b_hh))
        logits = np.dot(hidden_states[t], w_ho) + b_ho
        output_states[t] = np.exp(logits)/np.sum(np.exp(logits))
        loss += -np.log(output_states[t][0, targets[t]])

    # 第二部分：反向传播算法
    grad_w_ih, grad_w_hh, grad_w_ho = np.zeros_like(w_ih), \
                     np.zeros_like(w_hh), np.zeros_like(w_ho)
    grad_b_hh, grad_b_ho = np.zeros_like(b_hh), \
                                          np.zeros_like(b_ho)
    grad_hidden_next = np.zeros_like(hidden_states[0])
    for t in reversed(range(len(inputs))):
        delta_output = np.copy(output_states[t])
        delta_output[0,targets[t]] -= 1
        grad_w_ho += np.dot(hidden_states[t].T, delta_output)
        grad_b_ho += delta_output
        grad_hidden = np.dot(delta_output, w_ho.T) + \
                                            grad_hidden_next
        delta_hidden = (1 - hidden_states[t]*hidden_states[t])\
                                                * grad_hidden

        grad_b_hh += delta_hidden
        grad_w_ih += np.dot(input_states[t].T, delta_hidden)
        grad_w_hh += np.dot(hidden_states[t-1].T, delta_hidden)
        grad_hidden_next = np.dot(delta_hidden, w_hh.T)
    for grad_param in [grad_w_ih, grad_w_hh, grad_w_ho, \
                                    grad_b_hh, grad_b_ho]:

        np.clip(grad_param, -2, 2, out=grad_param)
    return loss, grad_w_ih, grad_w_hh, grad_w_ho, grad_b_hh, \
           grad_b_ho, hidden_states[len(inputs)-1]
```

下面定义函数 sample()，该函数有两个参数，init_chars 为输入的初始小段字符序列，n 为输出的字符长度。在函数 sample()中，通过 init_chars 的字符序列得到一个隐藏层状态 hidden。产生字符序列过程中，每次输入一个字符，通过正向传播算法得到下一个字符的预测概率向量。然后依据该预测概率向量随机抽取下一个字符，再把抽取到的字符作为输入字符。

```
# 函数 sample()，使用正向传播算法，通过随机抽样得到预测字符
def sample(init_chars, n):
    """
      从模型中随机抽样，得到一个正数序列
      h 是隐藏层状态
    """
    hidden = np.zeros((1, hidden_size))
    s = []
    for t in range(len(init_chars) + n):
        if t < (len(init_chars)):
            ix = char_to_index[init_chars[t]]
            input = np.zeros((1, vocab_size))
            input[0, ix] = 1
        else:
            logits = np.dot(hidden, w_ho) + b_ho
            prob = np.exp(logits) / np.sum(np.exp(logits))
            ix = np.random.choice(range(vocab_size),   \
                                  p=prob.ravel())
            input = np.zeros((1,vocab_size))
            input[0,ix] = 1

        hidden = np.tanh(np.dot(input, w_ih) + \
                         (np.dot(hidden, w_hh) + b_hh))
        s.append(ix)

    return s
```

最后使用上面定义的函数 lossFun()训练模型，并且使用函数 sample()得到预测字符序列。RNN 字符序列长度为 25。为了让训练过程更加稳定，我们使用了两个小技巧。

- smooth_loss 的初始值为一个较大的数，在训练过程中，每次小比例更新训练误差。这么可以让输出的损失函数的值更加平稳。
- 更新权重和截距项时，让梯度除以对应梯度累加和的开根号。这么可以让学习步长逐步变小。

每迭代 10 次，我们以 First Citizen:为起始序列随机抽样产生 200 个字符。

```
epochs = 30
# 超参数设置：隐藏层长度；输入序列长度；学习步长
hidden_size, seq_length, lr = 20, 25, 0.1
seq_num = int((len(data)-1)/seq_length)

# 设置累积梯度为，初始值为 0
```

```
mem_w_ih, mem_w_hh, mem_w_ho = np.zeros_like(w_ih), \
                      np.zeros_like(w_hh), np.zeros_like(w_ho)
mem_b_hh, mem_b_ho = np.zeros_like(b_hh), np.zeros_like(b_ho)

smooth_loss = -np.log(1.0/vocab_size)*seq_length
for e in range(epochs):
    prev_hidden = np.zeros((1, hidden_size))
    for i in range(seq_num):
        # 准备输入数据，输出数据。
        seq_start, seq_end = i*seq_length, (i+1)*seq_length
        inputs = [char_to_index[ch] for ch in \
                                data[seq_start:seq_end]]
        targets = [char_to_index[ch] for ch  in \
                        data[(seq_start+1):(seq_end+1)]]

        # 调用函数 lossFun，得到预测误差，参数梯度，隐藏层
        loss, grad_w_ih, grad_w_hh, grad_w_ho, grad_b_hh,\
         grad_b_ho, prev_hidden = \
                     lossFun(inputs, targets, prev_hidden)
        smooth_loss = smooth_loss * 0.999 + loss * 0.001

        for param, grad_param, mem in \
          zip([w_ih, w_hh, w_ho, b_hh, b_ho],\
      [grad_w_ih, grad_w_hh, grad_w_ho, grad_b_hh, grad_b_ho],\
      [mem_w_ih, mem_w_hh, mem_w_ho, mem_b_hh, mem_b_ho]):
            mem += np.abs(grad_param)
            param -= lr * grad_param/np.sqrt(mem + 1e-8)

  # 随机抽样，得到预测
    if(e % 5 == 0 or e == (epochs - 1)):
        print('Epoch %d, loss: %f' % (e, smooth_loss))
        sample_ix = sample(data[0:14], 200)
        s = ''.join(index_to_char[ix] for ix in sample_ix)
        print('----\n %s \n----' % (s))

Epoch 0, loss: 53.249057
----
 First Citizen:
Kin' nstedyer sowi! Cear,
And a apdridbed your hiwith.
Pay the siges lare' shalles wils:
Aw.
Wo mime thou fe lpothes:
But tow thath, tive,
The theasptenttorp sang mususay that unate!
I.

SION:
A:
A o
----
Epoch 10, loss: 51.379775
----
 First Citizen: he it batticen aid my madlues dot wo dissienfop.
```

```
TERWIO:
I' fillrt.

ERSPERTIO:
Whous.

PBick:
SVouliges:
Tour nom xey now me trexturood
Andace in!

TANUMITABES:
Nor ber me tall anever have not thee
----
Epoch 20, loss: 51.202619
----
 First Citizen: couraly ud minn
Thriclion as saxt ene bo low my amis here ste?

GLENUC:

RDUTAY:
Mare may, I sheathr;

ID II:
You, mord not sar more spicher ard,
I plofor:
Groek saands oable
-Pran; thin he cors Good
----
Epoch 29, loss: 51.090442
----
 First Citizen: thine on gide the ro beay that sho's for arrt
in Frow to.
Their this stenive a. thou Torlans wasters gofe, reswaid:
Lattsent?
I.
But you, wralf faindowice, you, the maly il now you his tell
prokes' l
----
```

10.4 本章小结

在本章中，我们学习了词嵌入和卷积神经网络（RNN）。词嵌入可以使用较少的维度对单词高效编码，而且意思相似的单词有着相近的词嵌入编码。通过建立隐藏层之间的关系，RNN 可以更加有效地使用数据的序列特征，从而提高神经网络模型的预测精度。

与之前章节学过的模型相比，RNN 模型的训练过程更加复杂。你可以尝试自己推导 RNN 的正向传播算法、反向传播算法以及独立从零开始实现 RNN 模型，巩固自己对 RNN 的理解。

从莎士比亚的例子可以看到，RNN 模型从莎士比亚作品中学习了部分莎士比亚的风格，如输出的文本类似剧本的形式，产生的字符中用空格分开不同的单词，用标点符号分割句子，有些单词的拼写是对的，等等。但是，总的来说，该 RNN 模型的表现还远远没达到理想水平。在接下来几章中，我们将学习在实践中常用的预测效果更好地 RNN 模型，LSTM 和 GRU。

习题

1. 分析 IMDB 数据，建立图 10-3 所示的神经网络模型。建模过程中，尝试以前章节使用的计算 `layer_1` 和 `weights_0_1` 的方式，即，使用矩阵乘法计算 `layer_1`，以及 `weights_0_1` 的梯度。比较以前章节学习的方式和本章学习的方式的计算结果和运算时间。
2. 分析 IMDB 数据，建立神经网络模型。但是，现在我们把每条评论编码为单词出现的频率。模型的表现会更好吗？想想有什么办法可以改进模型。例如，and，you，the 等单词在每条评论中出现频率都很高，而这些单词并没有提供情绪信息。尝试去除类似 and，you，the 等无用单词的干扰，预测准确度提高了吗？
3. 在像莎士比亚写作的例子中，尝试更长的输入，输出序列；尝试不同的隐藏层长度；尝试不同的梯度截断范围；看看不同的超参数设置对模型结果的影响。
4. 在像莎士比亚写作的例子中，我们对每个字母编码，逐次输入一个字符，预测一个字母。请尝试对每个单词进行编码，逐次输入一个单词，预测一个单词。这时，单词的数量将会很大，请使用词嵌入提高代码运行效率。