

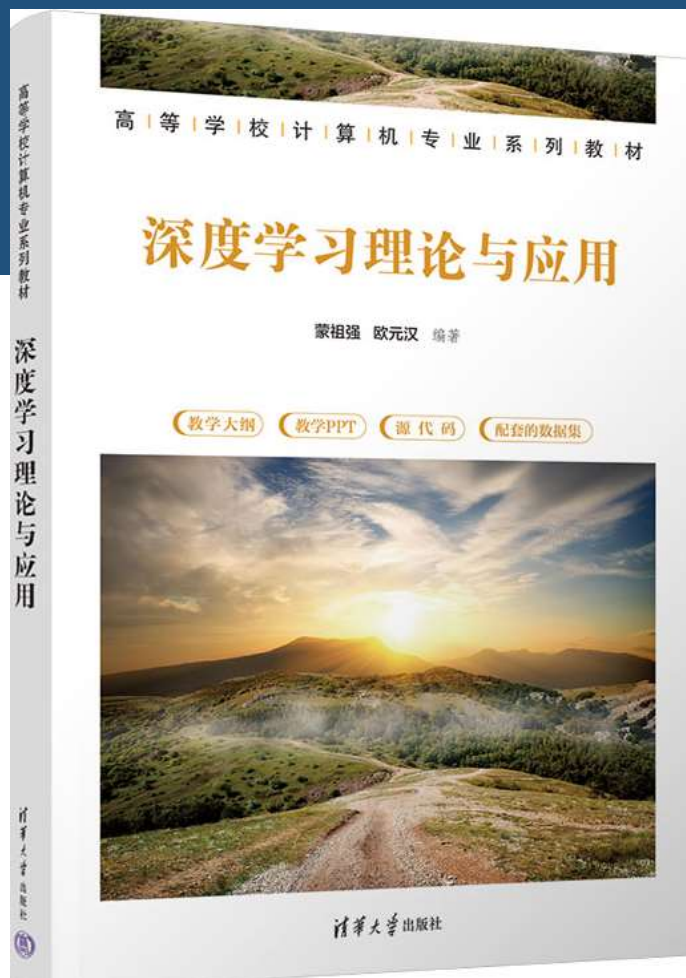
深度学习理论与应用

Deep Learning Theory and Applications

蒙祖强，欧元汉 编著

教材

全国各大
书店网店
均有销售

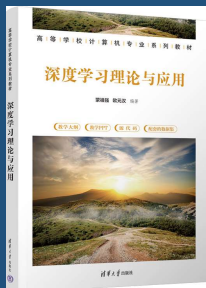


- **教学大纲：** 提供面向教育工程认证的教学大纲
- **教学PPT：** 提供课堂教学用的PPT课件
- **源代码：** 提供教材涉及的全部源代码
- **数据集：** 提供教材示例、案例用到的全部数据集

获取教学资源：

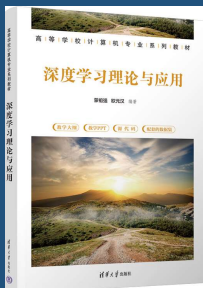
http://www.tup.tsinghua.edu.cn/booksCenter/book_09988101.html

教材：蒙祖强，欧元汉 编著. 深度学习理论与应用. 北京：清华大学出版社，2023年7月. (书号: 978-7-302-63508-6)



第7章 循环神经网络

蒙祖强，欧元汉 编著. 深度学习理论与应用. 北京: 清华大学出版社，2023年7月.



本章内容

contents

7.1 一个简单的循环神经网络——航空旅客出行人数预测

7.2 循环神经网络

7.3 长短时记忆网络(LSTM)

7.4 文本的表示

7.5 基于LSTM的文本分类

7.6 基于LSTM的文本生成

7.1 一个简单的循环神经网络——航空旅客出行人数预测



廣西大學
GUANGXI UNIVERSITY

7.1.1 程序代码

【例7.1】创建一个简单的循环神经网络，用于预测某一国际航空公司每月旅客出行人数。

- **航空出行人数数据集：**该数据集记录了1949 年到1960 年间每月搭乘本公司航班出行的旅客人数，一共有12 月/ 年 \times 12 年=144 条数据。文件名为international-airline-passengers.csv（位于./data 目录下）。
- **数据集示例：**右图该文件的内容结构：

	A	B	C
1	time	passengers	
2	Jan-49	112	
3	Feb-49	118	
4	Mar-49	132	
5	Apr-49	129	
6	May-49	121	
7	Jun-49	135	
8	Jul-49	148	
9	Aug-49	148	
10	Sep-49	136	
11	Oct-49	119	
12	Nov-49	104	
13	Dec-49	118	
14	Jan-50	115	
15	Feb-50	126	
16	Mar-50	141	

7.1 一个简单的循环神经网络——航空旅客出行人数预测

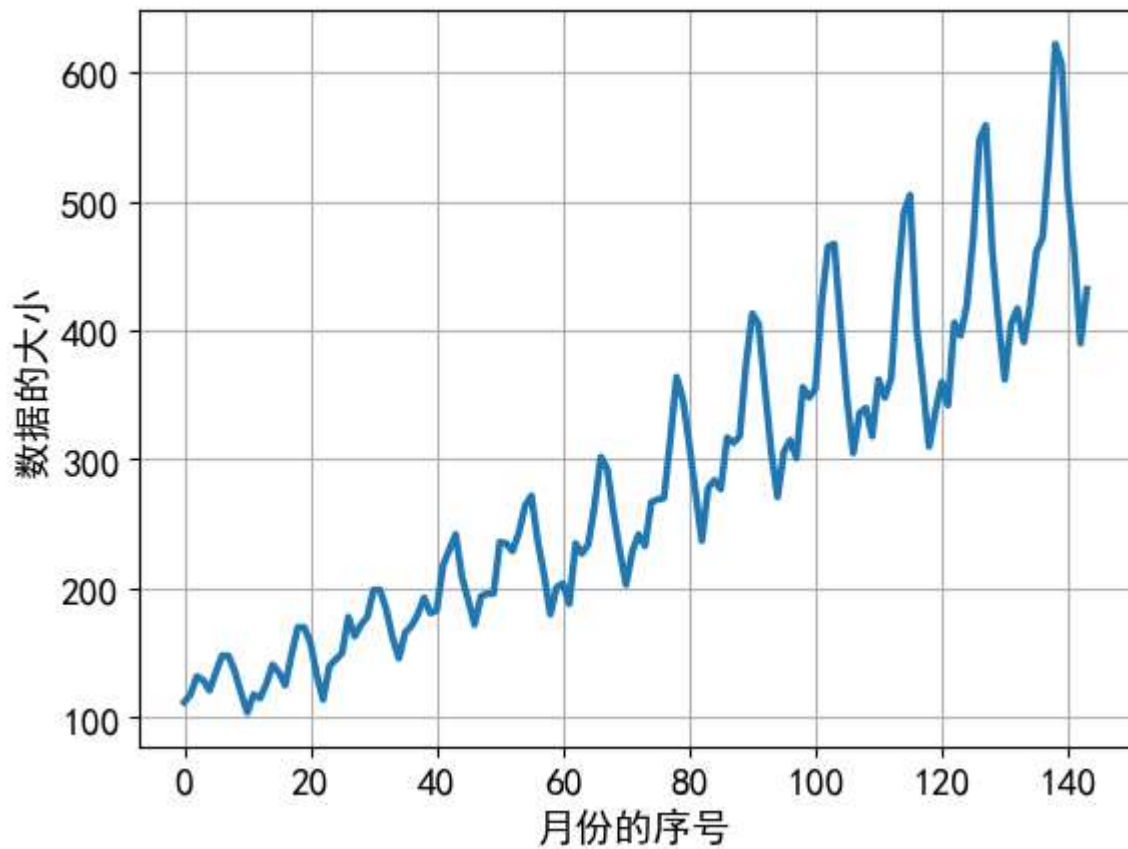


廣西大學
GUANGXI UNIVERSITY

7.1.1 程序代码

数据有着明显的**先后关系**。而且数据分布很有规律，是一种序列结构的数据。数据的变化趋势如右图：

可以从不同**粒度层**对其进行建模。**最细的粒度层**是以单个数据为单位来研究这类数据变化趋势的预测。**中等粒度层**是以季度为单位，即一共有48个季度，每个季度由3个数字构成的向量来表示。**最粗的粒度层**是以年为单位，即一共有12年。



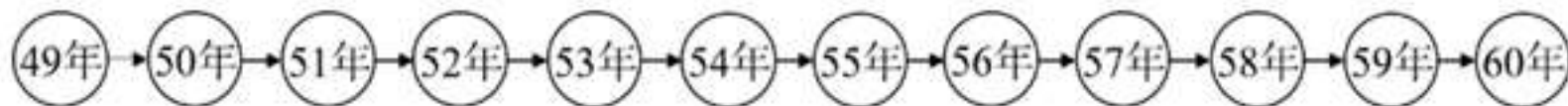
7.1 一个简单的循环神经网络——航空旅客出行人数预测



廣西大學
GUANGXI UNIVERSITY

7.1.1 程序代码

本例正是以年为单位来研究数据变化的规律，得到长度为12 的数据序列，如下图



循环神经网络正是处理序列数据的“利器”，它可以利用前面若干个数据来预测当前的数据。循环神经网络在处理数据对象时，它不是单独考虑当前的对象而是利用其前面的若干对象来“综合”考虑，然后做出当前的决策。

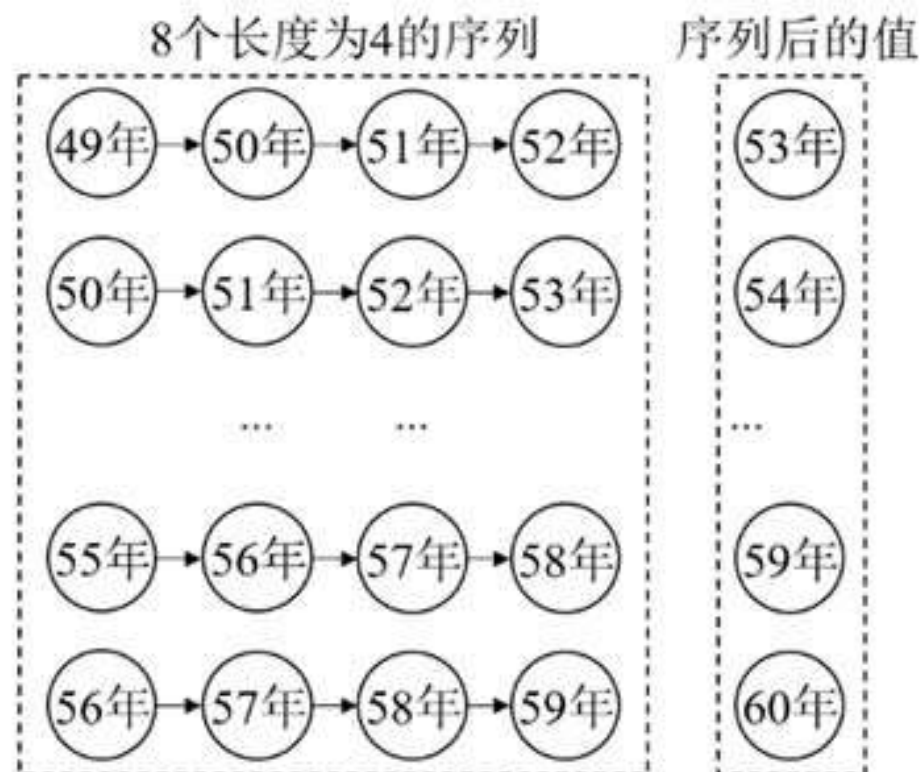
7.1 一个简单的循环神经网络——航空旅客出行人数预测



廣西大學
GUANGXI UNIVERSITY

7.1.1 程序代码

本例中，这个“若干个”被设置为4个，用前面连续4个年份来预测当前的年份。并用如右图所示构造网络训练的数据集。利用原来长度为12的总序列，可以构造出8个长度为4的子序列及子序列后的值，以此来构造训练集。



7.1 一个简单的循环神经网络——航空旅客出行人数预测



廣西大學
GUANGXI UNIVERSITY

7.1.1 程序代码

然后用LSTM（一种循环神经网络）来处理序列，其核心代码如下：

```
class Air_Model(nn.Module):
    def __init__(self):
        super(Air_Model, self).__init__()
        #构建LSTM 网络
        self.lstm = nn.LSTM(input_size=vec_dim, hidden_size=10, num_layers=1, \
                             batch_first=True, bidirectional=False, bias=True)
        self.linear = nn.Linear(10, vec_dim)
    def forward(self, x): #torch.Size([1, 4, 12])
        _, (h_out, _) = self.lstm(x)
        h_out = h_out.view(x.shape[0], -1) #
        o = self.linear(h_out)
        return o
```

（完整代码见教材P185页）

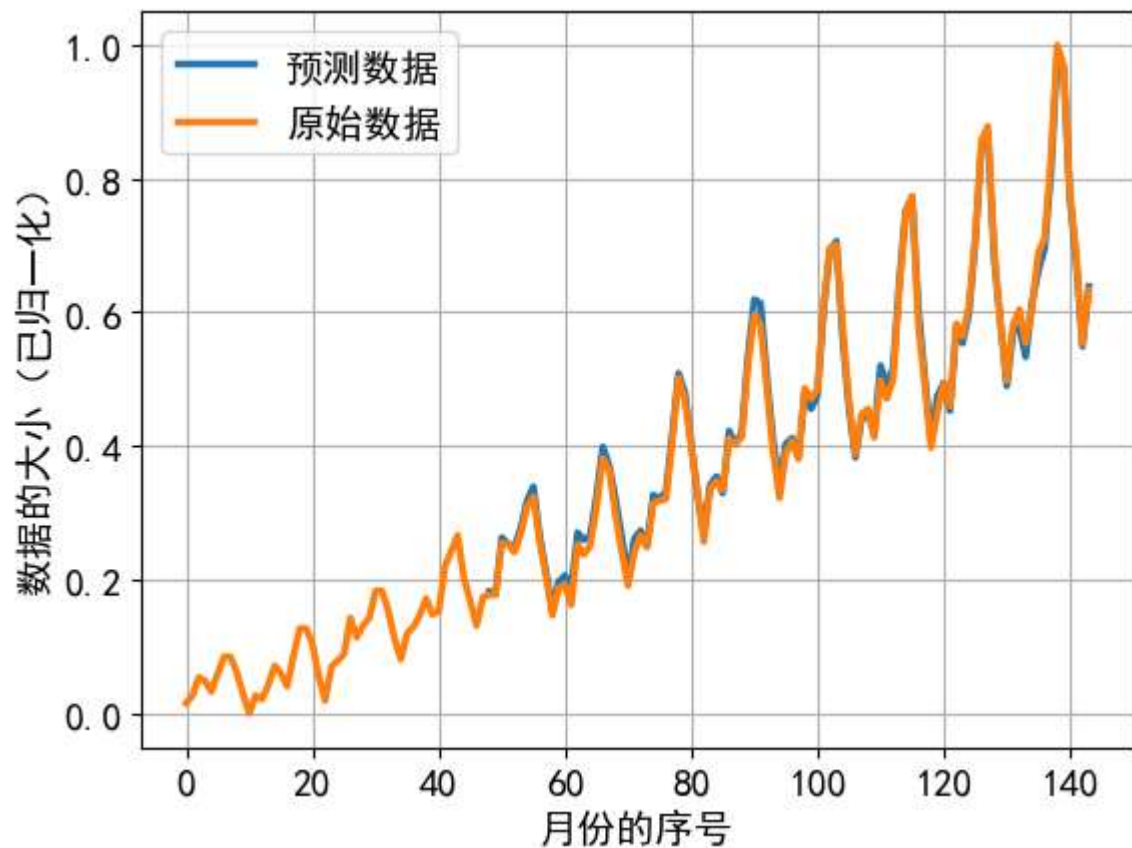
7.1 一个简单的循环神经网络——航空旅客出行人数预测



廣西大學
GUANGXI UNIVERSITY

7.1.1 程序代码

运行上述代码，产生如右图所示的结果。在图中，将原始数据用一条曲线表示（红色），预测的数据也用一条曲线来表示（蓝色）。从图中可以看到，这两条曲线几乎是重合在一起了，这说明所建模型对这类序列数据的预测能力是比较强的。



7.1 一个简单的循环神经网络——航空旅客出行人数预测



廣西大學
GUANGXI UNIVERSITY

7.1.2 代码解释

程序从文件international-airline-passengers.csv 中读取数据以后，以年为单位构建长度为12 的序列，序列中的元素（年）则由12 个月的数据构成的向量来表示，即序列中的元素表示为长度为12 的向量。然后，取序列中第1 至4 个元素构成一个子序列，取第5 个元素为该子序列后面的元素；再取第2 至5 个元素构成另一个子序列，取第6 个元素为其后续的元素，…，一共有8 个这样的子序列和它们的后续元素。相应代码如下：

7.1 一个简单的循环神经网络——航空旅客出行人数预测



廣西大學
GUANGXI UNIVERSITY

7.1.2 代码解释

```
train_x, train_y = [], []  
#构造8 个长度为4 的子序列及其后续的值 (seq_len=4)  
for i in range(data.shape[0] - seq_len):  
    tmp_x = data[i:i + seq_len, :] #子序列  
    tmp_y = data[i + seq_len, :] #子序列后面的值  
    train_x.append(tmp_x)  
    train_y.append(tmp_y)  
train_x = torch.FloatTensor(train_x) #张量化  
train_y = torch.FloatTensor(train_y)
```

7.1 一个简单的循环神经网络——航空旅客出行人数预测



廣西大學
GUANGXI UNIVERSITY

7.1.2 代码解释

运行结果：张量train_x 和train_y 的形状分别为(8, 4, 12)和(8, 12)，这就是后面所建立模型的训练数据集。接着，在程序中创建了名为Air_Model 的类，该类主要调用了LSTM 来构建一个循环神经网络，代码如下：

```
self.lstm = nn.LSTM(input_size=vec_dim, hidden_size=10, num_layers=1, \
                    batch_first=True, bidirectional=False, bias=True)
```

参数说明：input_size=vec_dim 表示序列中每个元素被表示为长度为vec_dim 的向量（此处vec_dim=12），hidden_size=10 表示网络隐含层的神经元个数为10，num_layers=1 表示只有一个隐含层。LSTM 要求输入的张量必须为3 维张量。batch_first=True 表示输入张量的第1 维用于表示批量的大小，第2 维用于表示序列的长度。

7.1 一个简单的循环神经网络——航空旅客出行人数预测



廣西大學
GUANGXI UNIVERSITY

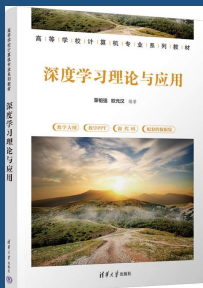
7.1.2 代码解释

LSTM: 它可以对给定的若干个对象（本例为4个）进行“综合考虑”，提取它们的整体特征，然后将这些特征送入其他网络作进一步处理。在本例中，将这些特征输入到一个全连接网络层。相关代码如下：

```
_, (h_out, _) = self.lstm(x) #x 的形状为torch.Size([1, 4, 12]), h_out 的形状为torch.Size([1, 1, 10])  
h_out = h_out.view(x.shape[0], -1)  
o = self.linear(h_out) #o 的形状为torch.Size([1, 12])
```

其中，h_out 返回的是输入子序列中最后一个对象（元素）输入时形成的特征，这个特征实际上就是对前四个对象综合的结果。然后，该特征被扁平化，送入一个全连接网络层 self.linear，该网络层输出的o 就是长度为12 的向量。

本例训练的目的就是让该输出向量尽可能接近输入序列后面紧跟的元素的向量，所以使用均方差损失函数 torch.nn.MSELoss()。



本章内容

contents

7.1 一个简单的循环神经网络——航空旅客出行人数预测

7.2 循环神经网络

7.3 长短时记忆网络(LSTM)

7.4 文本的表示

7.5 基于LSTM的文本分类

7.6 基于LSTM的文本生成

7.2 循环神经网络



廣西大學
GUANGXI UNIVERSITY

7.2.1 循环神经网络的基本结构

提到循环神经网络，就离不开**序列数据**。最典型的序列数据就是自然语言文本。例如，假如有下面一句未说完的话：

他超额完成了任务，经理表扬了____。

空格上应该填“**他**”。这是因为把上面这些文本当作一个词的序列来分析，当遇到空格的时候自然而然地综合了前面的词，从而得出应该填“他”的决策。如果只是独立地去读每一个词，人类也很难准确填写空格上的词。

对序列数据的处理，要求在处理当前对象的时候要同时考虑处理前面对象时所产生的结果。但在本章之前介绍的神经网络都没有提供这种处理机制，因此需要构造一种能够综合前面结果的神经网络——这就是**循环神经网络**。

7.2 循环神经网络



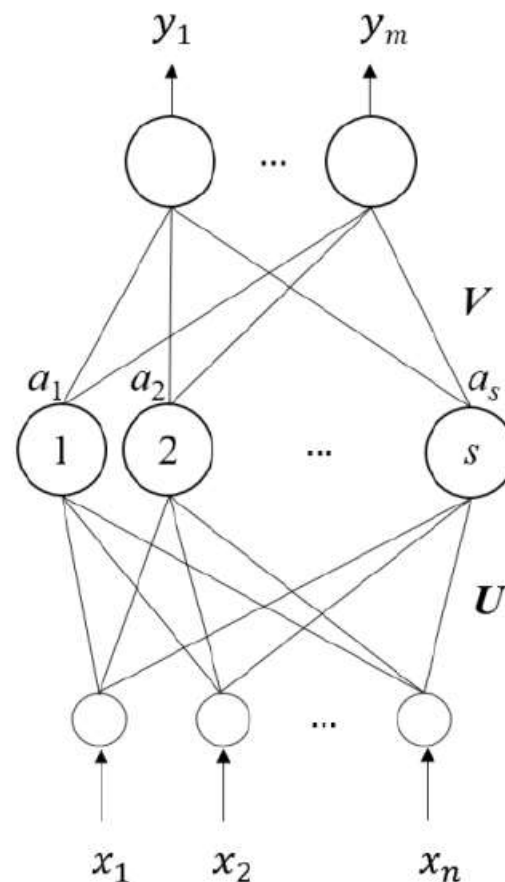
7.2.1 循环神经网络的基本结构

一个含有一层隐含层的全连接网络可以用右图(a)表示, 其中 U 和 V 分别表示输入层到隐含层之间和隐含层到输出层之间的参数构成的二维矩阵。右图(b)为其压缩式表示。

令 $\mathbf{x}=\{x_1, x_2, \dots, x_n\}$, $\mathbf{y}=\{y_1, y_2, \dots, y_m\}$ 以及 $\mathbf{a}=\{a_1, a_2, \dots, a_s\}$, 并用 σ 和 g 分别表示隐含层和输出层的激活函数, 则该网络的前向计算过程可以表示为:

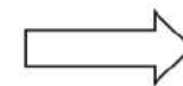
$$\mathbf{a} = \sigma(\mathbf{U}\mathbf{x} + \mathbf{b}_h)$$

$$\mathbf{y} = g(\mathbf{V}\mathbf{a} + \mathbf{b}_o)$$



(a) 全连接网络的基本结构

压缩表示



(b) 压缩表示

\mathbf{b}_h 和 \mathbf{b}_o 分别表示隐含层和输出层的偏置项向量。

7.2 循环神经网络

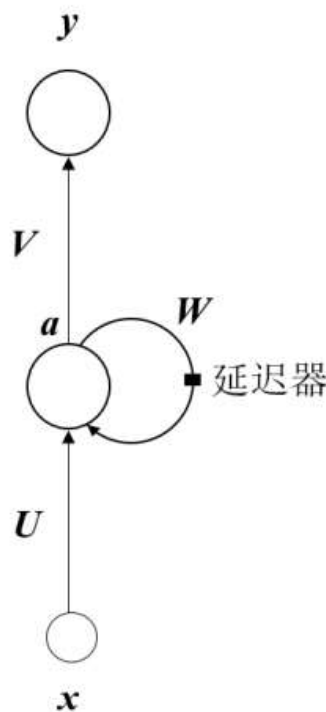


7.2.1 循环神经网络的基本结构

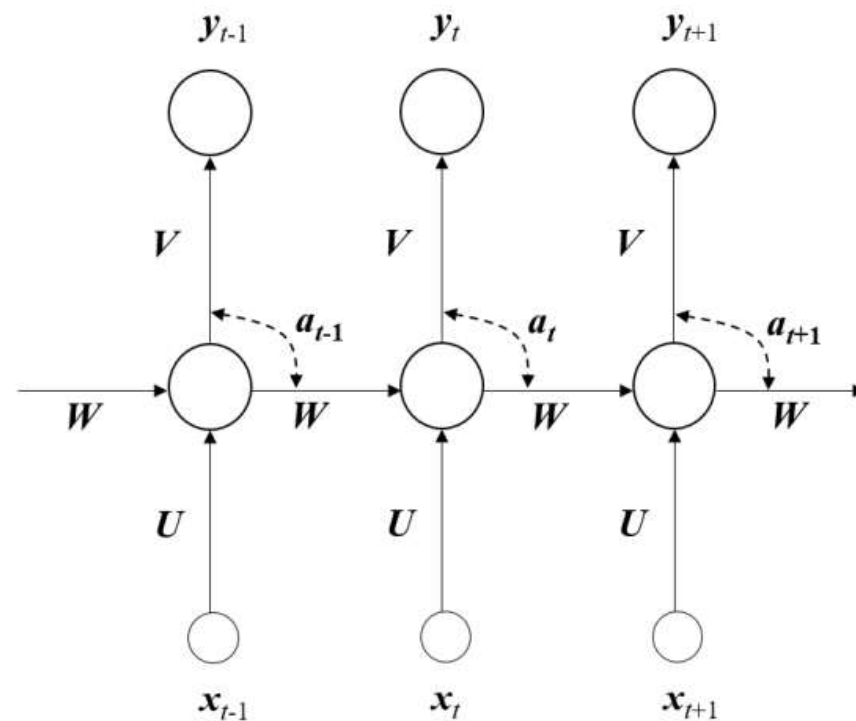
结果图解：为了使神经网络能够接受上一对象处理时形成的结果，以便用于综合处理当前的输入对象，需要隐含层节点加上一个从其输出到其输入的一个“闭环”图(a)所示。它实际上就是一个**全连接网络层**，其作用是将上一次处理时形成的输出乘以参数矩阵 W 后加到当前的输入上去。

图(a)所示的网络按照时间步展开，结果得到如图(b)所示的示意图，

其中 x_t 、 y_t 和 a_t 表示在时间步 t 时的输入、输出和隐含层激活输出等。



展开



(a) 循环神经网络的基本结构

(b) 按时间步展开的循环神经网络的逻辑结构

7.2 循环神经网络



7.2.1 循环神经网络的基本结构

循环神经网络计算：假设构建的循环神经网络用于处理长度为 T 的序列，则在计算时间步 t 的激活输出 a_t 时，需要前面的 $T-1$ 个隐含层的输出。 a_t 的计算过程如下：

$$\begin{aligned} a_t &= \sigma(Ux_t + b_h + Wa_{t-1} + b_w) \\ &= \sigma(Ux_t + b_h + W(\sigma(Ux_{t-1} + b_h + Wa_{t-2} + b_w)) + b_w) \\ &= \sigma(Ux_t + b_h + W(\sigma(Ux_{t-1} + b_h + W(Ux_{t-2} + b_h + Wa_{t-3} + b_w) + b_w)) + b_w) \\ &= \dots \dots \end{aligned}$$

其中， b_h 和 b_w 分别为隐含层和“延迟器”层的偏置项向量，它们也是待学习的参数。

在往回计算到第 T 个隐含层时（从 t 开始数），其输入为 x_{t-T+1} ，激活输出为 a_{t-T+1} ，同时需要前一个隐含层的输出 a_{t-T} （最初的 a_0 通过随机初始化产生）。

越靠前的 a_{t-i} ($i \in \{0, 1, \dots, T-1\}$) 其对当前 a_t 的影响就越小。如果 T 太大，那么这种影响几乎可以忽略不计。因此，这种循环神经网络对待处理的序列的**长度是有限制的**，即不能处理长度太长的序列，或者说不能处理长距离的依赖。

7.2 循环神经网络



7.2.2 从“零”开始构建一个循环神经网络

【例7.2】预测某一国际航空公司每月旅客出行人数。

循环神经网络在结构比较简单，它只是比全连接网络多出一个“**延迟器**”。先构建一个全连接网络，其中输入层、隐含层和输出层中的节点数分别为 n , s 和 m ，代码如下：

```
self.U = nn.Linear(n, s) #输入层到隐含层  
self.V = nn.Linear(s, m) #隐含层到输出层
```

并构造隐含层到隐含层的“延迟器”：

```
self.W = nn.Linear(s, s) #隐含层到隐含层
```

规定输入张量的形状为 $(batch_size, seq_len, vec_dim)$ ，其中 $batch_size$ 表示数据批量的大小， seq_len 表示输入序列的长度， vec_dim 为序列中表示每个元素的向量的长度。假设用 x 表示输入的张量，则 $batch_size$, seq_len , vec_dim 分别等于 $x.size(0)$, $x.size(1)$, $x.size(2)$ 。在本例中， $batch_size = 1$, $seq_len = 4$, $n = m = x.size(2) = vec_dim = 12$, s 可以自由设置。

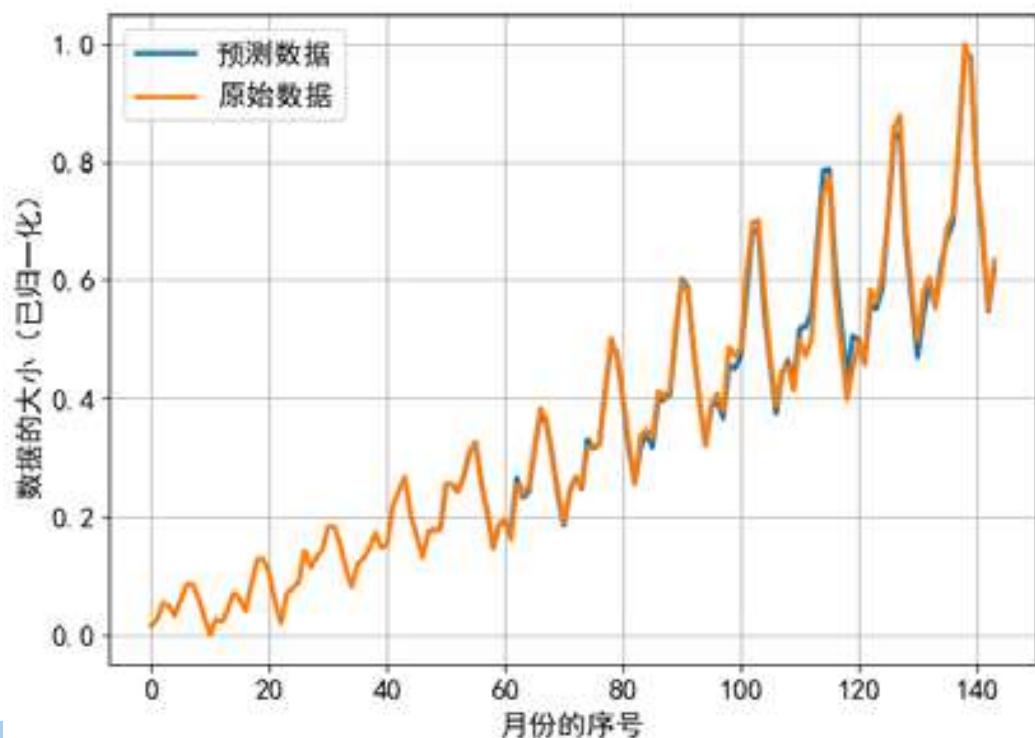
7.2 循环神经网络



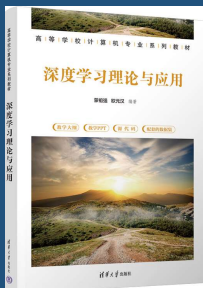
7.2.2 从“零”开始构建一个循环神经网络

实现类Air_Model（见例7.1）相似功能的类My_RNN，其完整代码如下：

运行修改后的代码，产生与例7.1 相似的结果



```
class My_RNN(nn.Module):
    def __init__(self,n=vec_dim,s=128,m=vec_dim):
        super(My_RNN, self).__init__()
        self.s = s
        self.U = nn.Linear(n, s) #输入层到隐含层
        self.V = nn.Linear(s, m) #隐含层到输出层
        self.W = nn.Linear(s, s) #隐含层到隐含层
    def forward(self, x):
        a_t_1 = torch.rand(x.size(0), self.s)
        lp = x.size(1)
        for k in range(lp):
            input1 = x[:, k, :]
            input1 = self.U(input1)
            input2 = self.W(a_t_1)
            input = input1 + input2
            a_t = torch.relu(input)
            a_t_1 = a_t
        y_t = self.V(a_t)
        return y_t
```

本章内容

contents

7.1 一个简单的循环神经网络——航空旅客出行人数预测

7.2 循环神经网络

7.3 长短时记忆网络(LSTM)

7.4 文本的表示

7.5 基于LSTM的文本分类

7.6 基于LSTM的文本生成

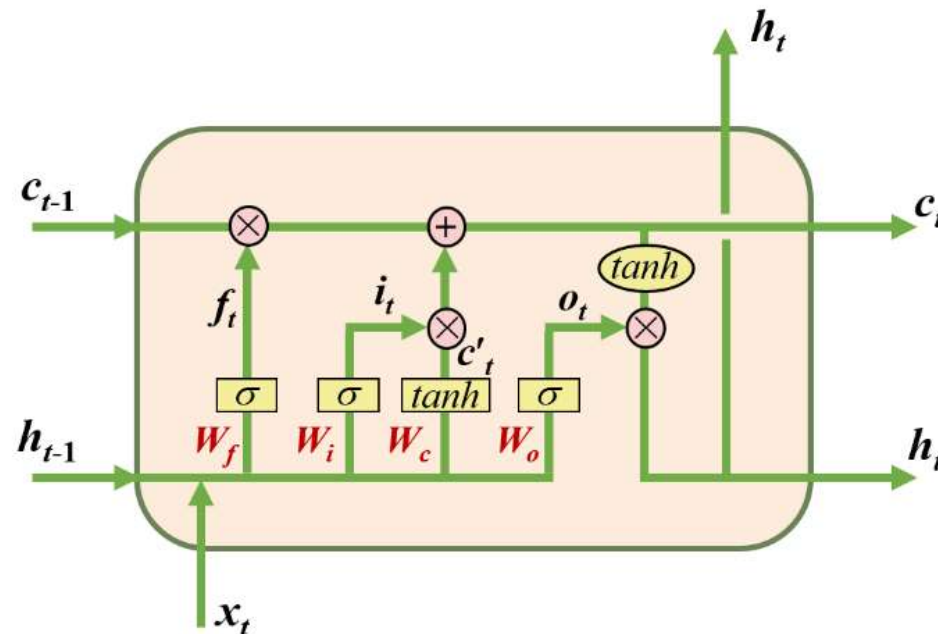
7.3 长短时记忆网络 (LSTM)



7.3.1 LSTM的结构和特点

LSTM (Long Short Term Memory Network, 长短时记忆网络) 是由德国科学家 Schmidhuber 于1997 年提出来的一种循环神经网络。LSTM 的**优点**在于, 它在一定程度上解决了原始循环神经网络遇到的**长距离依赖问题**。也就是说, 它可以处理更大长度的输入序列, 这已经在自然语言处理、图片描述、语音识别等领域中获得成功应用。

长距离依赖问题的**本质**是**梯度消失或梯度爆炸**, 这导致了原始循环神经网络不能处理较长的序列。为了解决这个缺陷, LSTM 增加了一个**长期状态** (也称细胞状态) c 。这样, 连同原来的短时状态 (也称即时状态或隐藏状态) h , 一共就两个状态 c 和 h 。 c 用于保存长期信息, h 用于保存短期信息。LSTM 的基本结构如右图。



7.3 长短时记忆网络 (LSTM)



7.3.1 LSTM的结构和特点

LSTM 主要功能：维持对 \mathbf{c} 和 \mathbf{h} 的有效控制，使得 \mathbf{c} 能够保存长期信息， \mathbf{h} 能够保持短期信息。LSTM对 \mathbf{c} 和 \mathbf{h} 的控制主要是通过三个门来实现，分别是**遗忘门 \mathbf{f}_t** 、**输入门 \mathbf{i}_t** 和**输出门 \mathbf{o}_t** 。它们的作用说明如下。

- **遗忘门 \mathbf{f}_t** ：负责控制上一时间步的长期状态 \mathbf{c}_{t-1} 有多少信息应该被遗忘，还有多少信息应该继续保存到当前长期状态 \mathbf{c}_t 中；
- **输入门 \mathbf{i}_t** ：负责控制把当前输入信息 \mathbf{x}_t 和上一时间步的短时状态 \mathbf{h}_{t-1} 有多少可以保存到当前长期状态 \mathbf{c}_t 中；
- **输出门 \mathbf{o}_t** ：负责控制把当前长期状态 \mathbf{c}_t 中的多少信息转化为当前的短时状态 \mathbf{h}_t 并输出。

门 (gate)：实际上就是一个**全连接网络层**。因此，每个门都有一个参数矩阵和一个偏置项。遗忘门 \mathbf{f}_t 、输入门 \mathbf{i}_t 和输出门 \mathbf{o}_t 的参数矩阵分别记为 \mathbf{W}_f 、 \mathbf{W}_i 和 \mathbf{W}_o ，其偏置项分别记为 \mathbf{b}_f 、 \mathbf{b}_i 和 \mathbf{b}_o 。每个门的输入是一个向量，输出是一个0到1之间的实数向量。

通过这三个门的控制，使得 \mathbf{c}_t 可以保持到当前时刻为止的历史信息，因而它是一种长期状态。

7.3 长短时记忆网络 (LSTM)



7.3.1 LSTM的结构和特点

输入门 i_t ：控制候选状态 c'_t 输入到 c_t 中的信息量，而 c'_t 是上一时间步的短时状态 h_{t-1} 和前输入信息 x_t 的综合，这种综合也是通过一个全连接网络层来实现的。该网络层的参数矩阵和一个偏置项分别记为 W_c 和 b_c 。当前计算单元会同时接收到上一计算单元的输出 h_{t-1} 和当前计算单元的输入 x_t 。对二者的处理有不同的方式：一种是**拼接处理**，另一种是**分割处理**。

1. 拼接处理方式

拼接处理：指对 h_{t-1} 和 x_t 按照最后一个维数进行拼接。例如，假设 h_{t-1} 和 x_t 的形状分别为(128, 10)和(128, 20)，则按最后一维进行拼接后得到形状为(128, 30)的张量。我们用 $[h_{t-1}, x_t]$ 表示对 h_{t-1} 和 x_t 拼接后得到的结果。将上述的门和全连接网络层表示为如下的数学公式：

7.3 长短时记忆网络 (LSTM)



7.3.1 LSTM的结构和特点

$$f_t = \sigma(W_f \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f)$$

$$i_t = \sigma(W_i \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i)$$

$$\mathbf{c}'_t = \tanh(W_c \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c)$$

$$\mathbf{o}_t = \sigma(W_o \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o)$$

$$\mathbf{c}_t = \mathbf{f}_t \otimes \mathbf{c}_{t-1} + \mathbf{i}_t \otimes \mathbf{c}'_t$$

$$\mathbf{h}_t = \mathbf{o}_t \otimes \tanh(\mathbf{C}_t)$$

其中， σ 表示某一种激活函数，“ \cdot ”表示矩阵相乘，“ \otimes ”表示向量元素乘积（即按位对向量进行相乘）。

待学习参数保存在四个参数矩阵 (\mathbf{W}_f 、 \mathbf{W}_i 、 \mathbf{W}_c 、 \mathbf{W}_o) 和四个偏置项 (\mathbf{b}_f 、 \mathbf{b}_i 、 \mathbf{b}_c 、 \mathbf{b}_o) 中。假设长期状态 \mathbf{c}_t 的维数为 n_c （不考虑批量的大小在内，下同），短期状态 \mathbf{h}_t 的维数为 n_h ，输入 \mathbf{x}_t 的维数为 n_x ，则 $[\mathbf{h}_{t-1}, \mathbf{x}_t]$ 的维数为 $n_h + n_x$ ，于是我们推知 \mathbf{W}_f 、 \mathbf{W}_i 、 \mathbf{W}_c 、和 \mathbf{W}_o 都是 $(n_h + n_x) \times n_c$ 矩阵

（因为它们都是连接 $[\mathbf{h}_{t-1}, \mathbf{x}_t]$ 到 \mathbf{c}_t 的网络层的参数矩阵），即每个矩阵有 $(n_h + n_x) \times n_c$ 个参数，一共有 $4(n_h + n_x) \times n_c$ 个参数。偏置项 \mathbf{b}_f 、 \mathbf{b}_i 、 \mathbf{b}_c 、 \mathbf{b}_o 都是向量，它们的维数跟 \mathbf{c}_t 的维数一样，都等于 n_c ，因此这些偏置项包含的待学习参数的总数为 $4n_c$ 。这样，LSTM网络的参数总量为 $4(n_h + n_x) \times n_c + 4n_c = 4(n_h \times n_c) + 4(n_x \times n_c) + 4n_c$ 。

7.3 长短时记忆网络 (LSTM)



7.3.1 LSTM的结构和特点

2. 分割处理方式

分割方式：在有的LSTM 变体（如PyTorch 版的nn.LSTM）中， h_{t-1} 和 x_t 并没有被拼接起来，而是单独分别送入相应的全连接网络，我们把这种处理方式称为**分割方式**。在分割处理方式中，上面的计算公式应改写为下列形式：

$$f_t = \sigma(W_f \cdot h_{t-1} + b_{f1} + W_f \cdot x_t + b_{f2})$$

$$i_t = \sigma(W_i \cdot h_{t-1} + b_{i1} + W_i \cdot x_t + b_{i2})$$

$$c'_t = \tanh(W_c \cdot h_{t-1} + b_{c1} + W_c \cdot x_t + b_{c2})$$

$$o_t = \sigma(W_o \cdot h_{t-1} + b_{o1} + W_o \cdot x_t + b_{o2})$$

$$c_t = f_t \otimes c_{t-1} + i_t \otimes c'_t$$

$$h_t = o_t \otimes \tanh(c_t)$$

由于 h_{t-1} 和 x_t 被独立送入不同的全连接网络中，因此就多出一个偏置项出来，从而导致偏置项包含的参数翻倍，因而整个LSTM 的参数数量也有所不同。

7.3 长短时记忆网络 (LSTM)



7.3.1 LSTM的结构和特点

$$f_t = \sigma(W_f \cdot h_{t-1} + b_{f1} + W_f \cdot x_t + b_{f2})$$

$$i_t = \sigma(W_i \cdot h_{t-1} + b_{i1} + W_i \cdot x_t + b_{i2})$$

$$c'_t = \tanh(W_c \cdot h_{t-1} + b_{c1} + W_c \cdot x_t + b_{c2})$$

$$o_t = \sigma(W_o \cdot h_{t-1} + b_{o1} + W_o \cdot x_t + b_{o2})$$

$$c_t = f_t \otimes c_{t-1} + i_t \otimes c'_t$$

$$h_t = o_t \otimes \tanh(c_t)$$

公式说明：“ $W_f \cdot h_{t-1} + b_{f1}$ ”是 h_{t-1} 到 c_t 的网络层，其参数数量为 $n_h \times n_c + n_c$ ；类似地，“ $W_f \cdot x_t + b_{f2}$ ”表示相应的网络参数数量为 $n_x \times n_c + n_c$ 。第一个公式对应的网络的参数数量为 $(n_h \times n_c + n_c) + (n_x \times n_c + n_c)$ 。而第二至第四个公式所对应的网络的参数数量同样都是 $(n_h \times n_c + n_c) + (n_x \times n_c + n_c)$ 。因此，nn.LSTM 的参数数量为 $4(n_h \times n_c + n_c) + 4(n_x \times n_c + n_c) = 4(n_h \times n_c) + 4(n_x \times n_c) + 8 n_c$ 。这比拼接方式多出 $4 n_c$ 个参数。

7.3 长短时记忆网络 (LSTM)

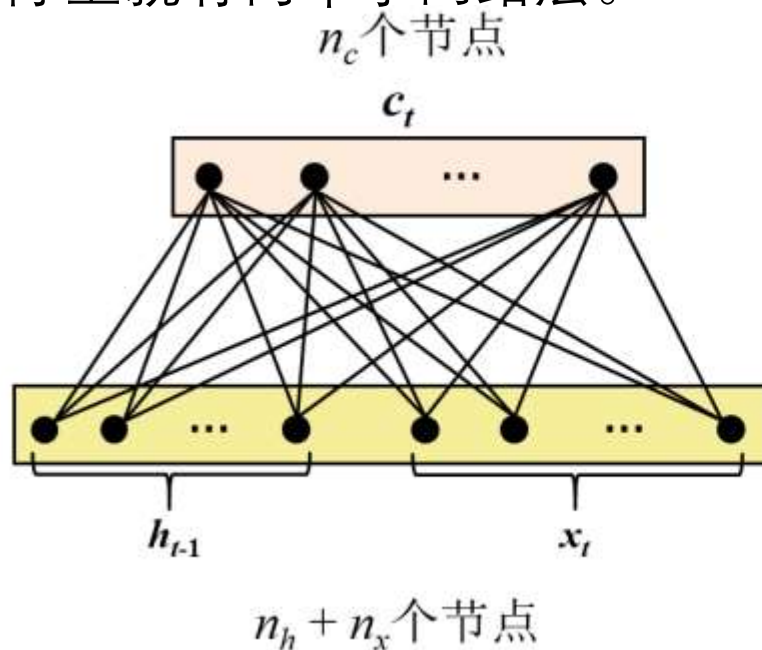


7.3.1 LSTM的结构和特点

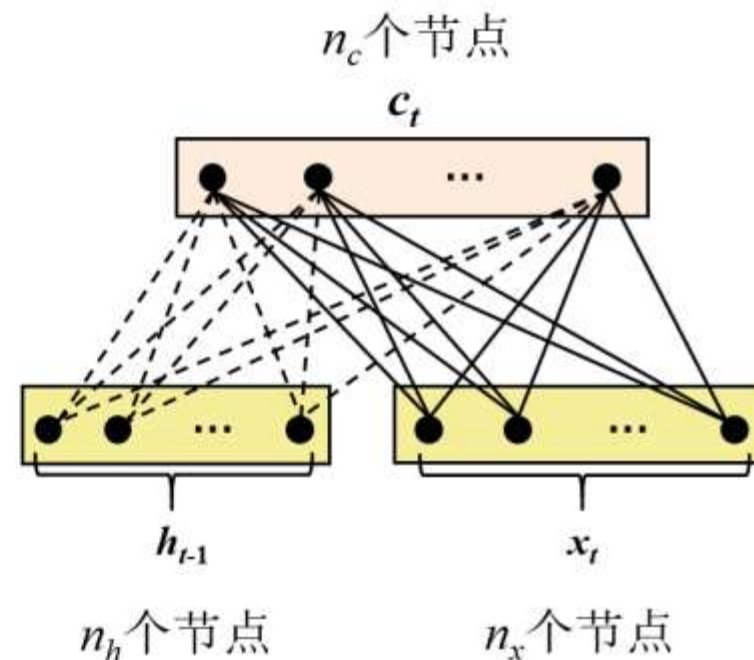
拼接处理方式和分割处理方式的区别如下图

拼接处理： h_{t-1} 和 x_t 被拼接为一个向量，因而该向量和长期状态节点构成一个全连接网络层。

分割处理： h_{t-1} 和 x_t 并没有被拼接为一个向量，而是各自单独跟长期状态节点构成自己的子网络层，实际上就有两个子网络层。



(a) 拼接处理方式



(b) 分割处理方式

7.3 长短时记忆网络 (LSTM)



7.3.2 LSTM使用方法

LSTM 的结构比较复杂，一般不需要从零开始去构造这样的网络，而是调用已经模块化的函数来实现。torch.nn 提供了LSTM 的实现模块，它的调用格式如下：

```
lstm = nn.LSTM(input_size, hidden_size, num_layers, bias, batch_first, dropout, bidirectional)
```

1. **input_size**: 对于输入序列，序列中的每个元素都需要表示成有统一规格的数值向量，而向量的长度则需要赋给参数input_size，以便LSTM 能够处理这些序列。例如，例7.1 中序列是由四个年份构成，每个年份（元素）则表示成长度为12 的数值向量，而长度12 则应该赋给该参数input_size，即input_size=12；又如，在自然语言处理中，序列通常是由若干个词构成，每个词（元素）必须表示成有相同长度的数值向量，而这个长度则必须赋给参数input_size。
2. **hidden_size**: 用于设置LSTM 中隐藏层神经元的个数，实际上就是设置长期状态c的维数 n_c 和短期状态h 的维数 n_h 。注意，在nn.LSTM 中， n_c 和 n_h 一般是相等的。也就是说，如果hidden_size = 512，则 $n_c = n_h = 512$ 。

7.3 长短时记忆网络 (LSTM)



7.3.2 LSTM使用方法

3. `num_layers`: 用于设置LSTM 的层数。例如，如果`num_layers=2`，则表示该LSTM有两层。`num_layers` 的默认值为1。
4. `bias`: 用于设置每个隐藏层神经元是否有偏置。如果`bias = True`（默认值），则表示每个神经元都有偏置项；如果`bias = False`，则没有。
5. `batch_first`: 该参数决定LSTM 接收输入张量的格式，LSTM 只能接受3 维张量的输入：
(1) 当`batch_first=True` 时，表示“批量在前，序列在后”，即输入张量的第1 维用于表示批量的大小，第2 维用于表示序列的长度，即输入张量的形状为：

`(batch_size, seq_len, vec_dim)`

其中，`batch_size` 表示批量中序列的数量（样本数量），`seq_length` 表示每个序列中元素的数量，`input_dim` 则为表示每个元素的向量的长度，该参数和参数`input_size`一般相等。

7.3 长短时记忆网络 (LSTM)



7.3.2 LSTM使用方法

(2) 当batch_first=False (默认值) 时, 表示“序列在前, 批量在后”, 即张量的第1 维用于表示序列的长度, 而第2 维才用于表示批量的大小, 即输入张量的形状为:

(seq_len, batch_size, vec_dim)

6. dropout: LSTM 模块内神经元随机丢失的比例, 仅在多层LSTM 的传递中使用
7. bidirectional: 当该参数为True 时, 表示LSTM 采用双向网络; 当为False 时 (默认值), 表示LSTM 采用单向网络。

LSTM 的参数众多, 而且参数取值的不同搭配组合会产生不同的返回结果, 加上LSTM 的工作过程比较抽象, 使得学习LSTM 的使用变得比较困难。下面从时间步上分析它的工作机制。

7.3 长短时记忆网络 (LSTM)



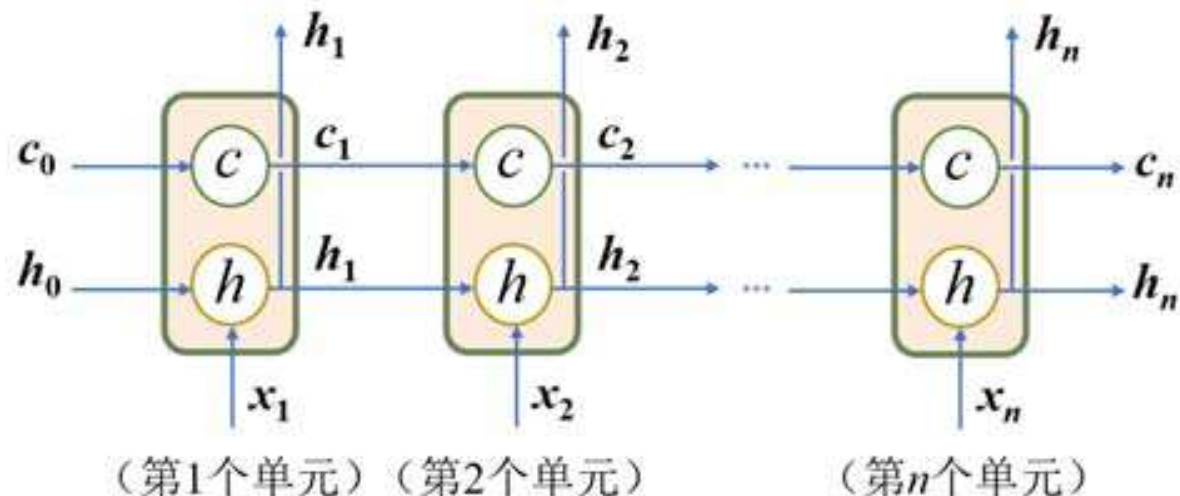
7.3.2 LSTM使用方法

LSTM使用： 调用对象lstm 的格式如下：

```
out, (hn, cn) = lstm(x)
```

其中， \mathbf{x} 为输入的张量，对象lstm 返回3 个结果，分别放在张量 \mathbf{out} , \mathbf{h}_n 和 \mathbf{c}_n 中。

假设在张量 \mathbf{x} 中序列的长度为seq_length，则LSTM 在处理过程中会产生seq_length 个时间步，在每个时间步上会形成一个逻辑计算单元，即一共有seq_length 个计算单元，如下图：



7.3 长短时记忆网络 (LSTM)

7.3.2 LSTM使用方法

参数设置:

(1) 当batch_first = False (默认值), 则输入张量 x 的形状为(seq_length, batch_size, input_dim)。例如, 如果张量 x 的形状为(30, 128, 200), 则表示 x 包含128个序列, 每个序列都包含30个元素, 每个元素用长度为200的数值向量来表示。

在运行时, 序列中的第1个元素 x_1 先被输入到LSTM中, 第1个单元利用 c_0 和 h_0 (被随机初始化, 或按某种分布对其初始化) 一起计算后输出 c_1 和 h_1 , 这时 h_1 的一个拷贝会被输出到当前单元的外部 (如课本图7-12中用向上箭头表示, 下同), 同时 c_1 和 h_1 会一起被送入到第2个单元; 然后, 序列中的第2个元素 x_2 被输入到第2个单元, 该单元利用上一单元的输出生成 c_1 和 h_1 , 计算后产生 c_2 和 h_2 并将之输出到下一个单元, 同时 h_2 的一个拷贝也会被输出到当前单元的外部; ..., 直到第 n 个单元, 它利用输入 x_n 和上一单元的输出生成 c_{n-1} 和 h_{n-1} , 计算后输出 c_n 和 h_n 。

7.3 长短时记忆网络 (LSTM)



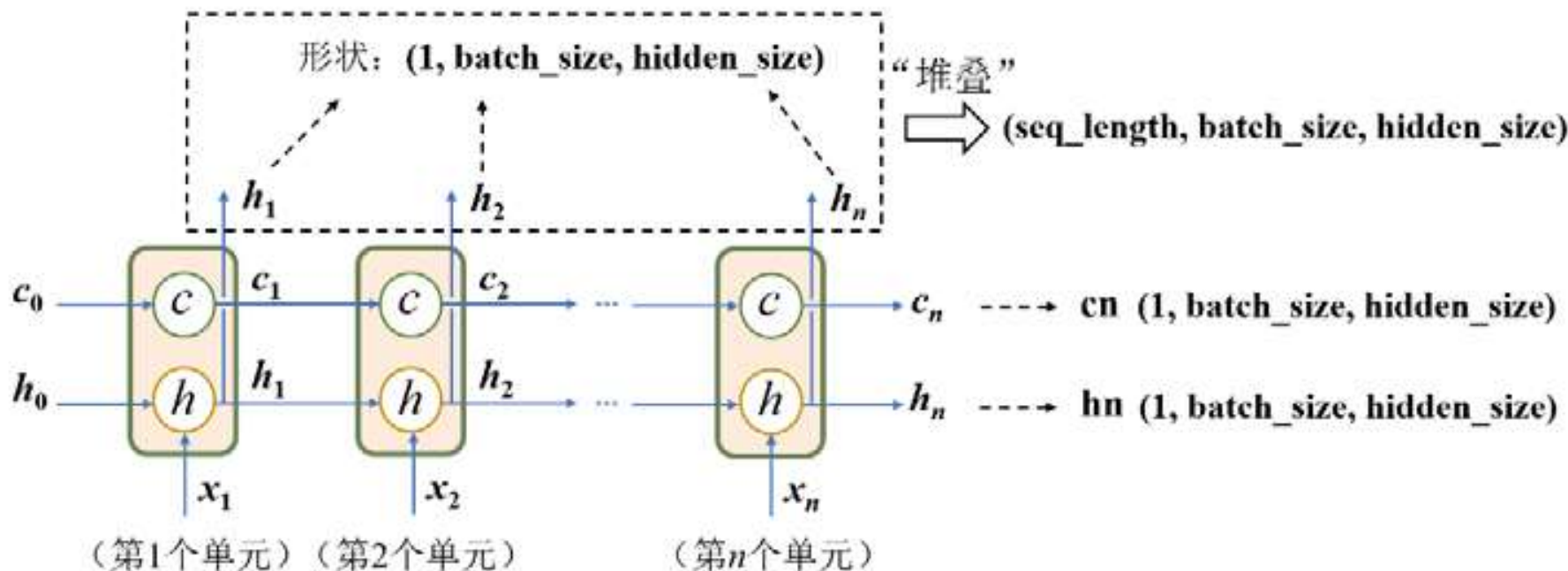
7.3.2 LSTM使用方法

在nn.LSTM 中, $\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_n$ 以及 $\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_n$ 的形状都是相同的, 均为 $(\text{num_layers} \times \text{bidirections}, \text{batch_size}, \text{hidden_size})$, 其中如果 $\text{bidirectional}=\text{True}$, 则 $\text{bidirections}=2$, 否则 $\text{bidirections}=1$ (默认值)。

张量 \mathbf{h}_n 和 \mathbf{c}_n 实际上分别保存第 n 个计算单元的两个输出: \mathbf{c}_n 和 \mathbf{h}_n 。显然, 张量 \mathbf{h}_n 和 \mathbf{c}_n 的形状也都是 $(\text{num_layers} \times \text{bidirections}, \text{batch_size}, \text{hidden_size})$ 。注意到, 每个计算单元都有一个输出 \mathbf{h}_i ($i = 1, 2, \dots, n$), 其形状也为 $(\text{num_layers} \times \text{bidirections}, \text{batch_size}, \text{hidden_size})$, 保存在张量 **out** 中。因此, 张量 **out** 的形状为 $(\text{seq_length}, \text{batch_size}, \text{hidden_size} \times \text{bidirections})$

7.3.2 LSTM使用方法

先考虑最简单和最常用的情况。假设num_layers = 1, bidirectional=False，即假设LSTM是单层、单向的（这也是默认和最简单的结构），则 $\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_n$ 、 $\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_n$ 以及张量 \mathbf{h}_n 和 \mathbf{c}_n 的形状均为(1, batch_size, hidden_size)，张量out的形状为(seq_length, batch_size, hidden_size)。对比这两种形状可以推知，每个计算单元的输出 \mathbf{h}_i ($i = 1, 2, \dots, n$)“堆叠”在一起就得到张量out，这种关系可从下图体会到。



7.3 长短时记忆网络 (LSTM)



7.3.2 LSTM使用方法

下面代码按第一维获取张量out 中的最后一个元素（张量），然后和张量 h_n 相对比：

```
lstm = torch.nn.LSTM(input_size=200, hidden_size=120, num_layers=1, \
                      batch_first=False, bidirectional=False, bias=True)
x = torch.randn(30,128,200) #随机生成输入张量x
out, (hn, cn) = lstm(x) #调用LSTM 对象lstm
print(x.shape) #输出各张量的形状
print(out.shape, hn.shape,cn.shape)
t = out[29,:].unsqueeze(0) #（按第一维）获取张量out 中的最后一个元素（即hn）
print('判断两个张量是否相等? ', '相等' if t.equal(hn) else '不相等')
```

执行上述代码，结果输出如下：

```
torch.Size([30, 128, 200])
torch.Size([30, 128, 120]) torch.Size([1, 128, 120]) torch.Size([1, 128, 120])
判断两个张量是否相等? 相等
```

7.3 长短时记忆网络 (LSTM)



7.3.2 LSTM使用方法

按第一维获取张量out 中的最后一个元素（也是一个张量），它跟最后一个单元输出的 h_n 是相等的。对于更复杂的情况，LSTM 主要是对各自的 h_i 进行相应的处理后再“堆叠”到out 中。

由张量x 表示的序列 x_1, x_2, \dots, x_n ，在经过上述n 个单元的计算后，产生的out（各个单元输出的“堆叠”）一般被视为这个序列 x_1, x_2, \dots, x_n 的特征，进而为下游应用服务（如分类等）。但在进一步使用之前，通常会按第一维对张量中的元素进行相加，如：

```
out = torch.sum(out, dim=0)
```

这时out 的形状会从(30, 128, 120)变为(128, 120)。

如果batch_first=True，则第二维表示序列长度，这时应该按第二维对张量中的元素进行相加。

7.3 长短时记忆网络 (LSTM)



廣西大學
GUANGXI UNIVERSITY

7.3.2 LSTM使用方法

张量加法：将序列中各个元素的特征加在一起，将得到的结果作为整个序列的特征。这里采用了一种假设：序列中各个元素是同等重要的。然而，在许多应用中序列中不同元素发挥的作用是不一样的，因而应采用加权求和，而不是简单的相加。也就是说，为每个元素找到一个权值，一共30个权值： $\alpha_0, \alpha_1, \dots, \alpha_{29}$ ，然后执行下列相加操作：

$$\text{out} = a_0 \cdot \text{out}[0] + a_1 \cdot \text{out}[1] + \dots + a_{29} \cdot \text{out}[29]$$

权值的寻找：这通常设置另一个网络来找（该网络有30输出节点，输出30个权值），这种网络一般是**注意力机制网络**，这将在第8章介绍。

7.3 长短时记忆网络 (LSTM)



7.3.2 LSTM使用方法

有时候 h_n (即 h_n) 也可能用作这个序列 x_1, x_2, \dots, x_n 的特征。例如, 在例7.1 中设置了下面一条语句:

```
_, (h_out, _) = self.lstm(x)
```

其作用就是使用 h_n 作为序列的特征。但是, 一般不使用 c_n (即 c_n) 作为序列的特征。

参数设置:

(2) 当`batch_first = True`, 输入张量 x 的形状应该设置为 $(batch_size, seq_length, input_dim)$ 。这时, 除了张量 out 的形状改变为 $(batch_size, seq_length, hidden_size \times bidirections)$ 以外, h_n 和 c_n 的形状没有发生改变, 仍然为 $(num_layers \times bidirections, batch_size, hidden_size)$, 同时LSTM 的工作机理也一样。

7.3 长短时记忆网络 (LSTM)



7.3.3 深度循环神经网络

前面介绍循环神经网络都只有一个隐藏层，因此它们不是真正意义上的深度神经网络。可以堆叠两个或两个以上的隐藏层，从而构造真正的深度神经网络。

在PyTorch 中，利用nn.LSTM 来构造深度神经网络就比较简单，只要把参数num_layers设置为2 及大于2 的整数即可（该参数的默认值为1）。例如，下列代码定义了由3 个隐藏层构成的深度LSTM 模型：

```
lstm = nn.LSTM(input_size=12, hidden_size=10, num_layers=3, \
               batch_first=True, bidirectional=False, bias=True)
```

7.3 长短时记忆网络 (LSTM)



7.3.3 深度循环神经网络

LSTM 模型输出的形状有改变（跟num_layers=1 的情况相比）。例如，执行下列代码：

```
x = torch.randn(1, 4, 12)
out, (h_out, c_out) = lstm(x)
print(out.shape, h_out.shape, c_out.shape)
```

h_out 和c_out 的形状均为torch.Size([3, 1, 10])，此外out 的形状为torch.Size([1, 4,10])。为此，需要对“3”进行相应的处理，例如：

```
h_out = torch.mean(h_out, dim=0, keepdim=True) #求平均值
```

h_out 的形状变为torch.Size([1, 1, 10])。

7.3 长短时记忆网络 (LSTM)



廣西大學
GUANGXI UNIVERSITY

7.3.4 双向循环神经网络

循环神经网络优点：它可以“回头看”，因而可以利用上文的信息来辅助当前的决策。但是，在有的情况下，不但需要“回头看”，而且还需要“**向后看**”，即还需要利用下文信息才能做出正确的决策。

例如，观察下面一句话：

我想出国，我准备去____雅思考试。

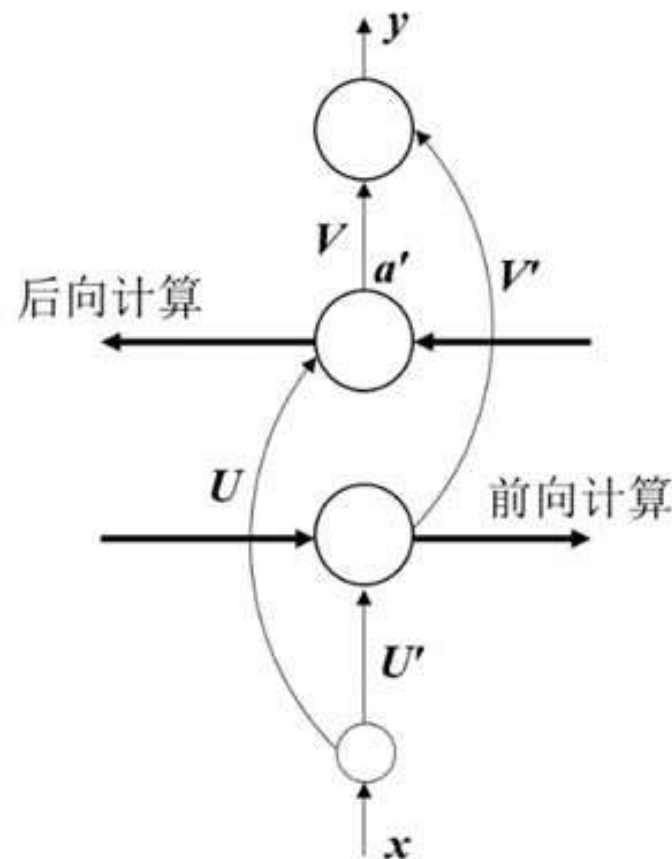
如果只看前面部分，那么横线上似乎可以填“签证”、“美国”等词，这带有很大的不确定性；但如果还看到后面的“雅思考试”，那么横线上填“参加”的概率就非常高了。这说明，有时候需要获得上下文的信息才能做出当前的决策。为此，人们提出了**双向循环神经网络**。

7.3 长短时记忆网络 (LSTM)



7.3.4 双向循环神经网络

双向循环神经网络 (Bidirectional recurrent neural network, BRNN)：在单向神经网络的基础上再增加一个隐藏层，这两个隐藏层连着相同的输入层和输出层，于是得到两个循环神经网络 (RNN)；但它们处理序列的顺序不一样，其中一个执行前向计算（由左到右），另一个执行后向计算（由右到左），它们共同给输出层提供序列中每一个元素（词）的过去和未来的上下文信息。但当采用双向循环神经网络时，输出层和隐藏层中的神经元数量都将翻倍。双向循环神经网络的这种逻辑结构可用右图表示。



7.3 长短时记忆网络 (LSTM)



7.3.4 双向循环神经网络

在PyTorch 中，利用nn.LSTM 来构造双向循环神经网络也非常容易，只需把参数bidirectional 的值设置为True 即可（其默认值为False，表示默认为单向循环神经网络）。例如，下列代码定义了一个双向循环神经网络模型：

```
lstm = nn.LSTM(input_size=12, hidden_size=10, num_layers=1, \
               batch_first=True, bidirectional=True, bias=True)
```

对于双向循环神经网络模型，除了参数个数翻倍以外，其输出的形状也产生变化。例如，执行下列代码：

```
x = torch.randn(32, 4, 12)
out, (h_out, _) = lstm(x)
print(out.shape, h_out.shape)
```

结果发现，out 和h_out 的形状分别为(32, 4, 20)和(2, 32, 10)；如果将bidirectional=True改为bidirectional=False，则out 和h_out 的形状分别为(32, 4, 10)和(1, 32, 10)。在编写代码时，需要根据这种形状的改变来调整后续代码。

7.3 长短时记忆网络 (LSTM)



7.3.5 LSTM的变体——GRU

自从LSTM 被提出以来，学者们对其进行了大量的改进，形成了多种LSTM 变体版本。其中，比较有名和成功的变体版本是**GRU**。GRU (Gated Recurrent Unit) 对LSTM 进行许多简化，其参数量明显比LSTM 的少，但却保持了和LSTM 几乎相同的效果，所以近年来GRU 变得越来越受欢迎。

在PyTorch 中，可以利用nn.GRU 来构造这种简化版的循环神经网络，其参数设置跟nn.LSTM 的参数设置是一样的。但由于在GRU 中长期状态单元和短时状态单元被合并为一个状态单元，因而其输出的张量有很大的变化。

例如，下面代码定义了一个GRU 模型：

```
gru = nn.GRU(input_size=12, hidden_size=10, num_layers=1, \
              batch_first=True, bidirectional=False, bias=True)
```

7.3 长短时记忆网络 (LSTM)



7.3.5 LSTM的变体——GRU

定义该GRU 模型所使用的参数设置跟上面定义LSTM 模型的参数设置完全一样，然而该GRU 模型只有720 个参数，而上面的LSTM 模型有960 个参数。

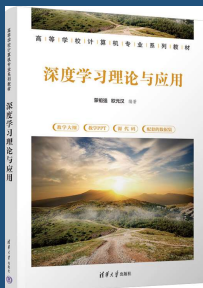
从执行下列代码输出的结果中可以看出二者返回结果的差异：

```
x = torch.randn(32, 4, 12)
out, h_out = gru(x)
print(out.shape, h_out.shape)
```

输出结果如下：

```
torch.Size([32, 4, 10]) torch.Size([1, 32, 10])
```

GRU 模型返回的结果只有两个张量，而LSTM 模型有三个张量。实际上，LSTM 模型返回的第三个张量一般很少使用，这也说明GRU 显得更为精简。



本章内容

contents

7.1 一个简单的循环神经网络——航空旅客出行人数预测

7.2 循环神经网络

7.3 长短时记忆网络(LSTM)

7.4 文本的表示

7.5 基于LSTM的文本分类

7.6 基于LSTM的文本生成

7.4 文本的表示

7.4.1 词的独热表示 (one-hot)

早期，词向量化方法主要是独热编码 (one-hot)。该方法的基本过程是，首先创建一个包含所有词的**词表**（这种词表实际上就是Python 语言中的字典，所以有时候也成为**字典**），并在词表中为每个词分配一个唯一的索引；假设词表大小为 N （即词的个数），对给定索引为 i 的词 w ，建立一个长度为 N 的向量 \mathbf{V} ，然后将向量 \mathbf{V} 中下标为 i 的分量设置为1，其他分量设置为0。这样得到的 \mathbf{V} 便是词 w 的**独热表示**，也称**独热向量**或**one-hot 向量**。

例如，对于句子“明天去看展览”，对它分词后可以得到“明天”、“去”、“看”、“展览”等四个词。首先建立词表：

`{"明天": 0, "去": 1, "看": 2, "展览": 3}`

在词表中，每个词都有一个唯一的索引。根据该词表，可以得到各词的one-hot 向量。

词	one-hot 向量
明天	(1, 0, 0, 0)
去	(0, 1, 0, 0)
看	(0, 0, 1, 0)
展览	(0, 0, 0, 1)



7.4.1 词的独热表示 (one-hot)

独热编码优点：直观、编码方便、易于理解的。

独热编码缺点：

(1) 稀疏性和高维性。在自然语言数据集中，可能有成千上万个不同的词，因而one-hot 向量的维数普遍比较高（甚至达到几十万维），而且在高维的向量中只有一个分量为1，其他分量均为0。这样，1 的分布十分稀疏，严重影响表达的效率。

(2) one-hot 向量不能表示上下文信息，无法体现词与词之间的关系。例如，任何两个不同词的one-hot 向量的欧氏距离均为0，利用欧氏距离无法计算词之间的语义相近程度等。因而，人们进一步提出了其他更出色的词表示方法。



7.4.2 Word2vec 词向量

Word2vec 意指“Word to vector”之意，它是谷歌于2013 年提出的词向量化的一种方法。Word2vec 实际上是NLP（自然语言处理）的一个工具，是一个预训练模型。其特点是，利用给定的数据集训练一个模型，该模型可以对输入的词给出相应低维、稠密的向量表示。这种向量较好地考虑上下文关系，不仅融合了**历史信息**，也融合了**“未来”的上下文信息**。相对于one-hot 向量，Word2vec 向量的维数要低得多，向量中每个分量都是不同的实数，且利用这些向量可以度量词与词之间的语义关系等。

下列代码说明了如何利用已有的语料训练一个Word2Vec，以及如何输出各个词的词向量等：

7.4 文本的表示



7.4.2 Word2vec 词向量

```
from gensim.models import Word2Vec
sentence = [['明天', '去', '看', '展览'], ['明天', '有', '图书', '展览']] #模拟预料
model = Word2Vec(sentences = sentence,
                  vector_size=5, #设置词向量的长度（在以前版本中size=5），默
                                #认值是100
                                #长度越长保留的信息越多，当vector_size 足够
                                #大时，甚至可以解决一词多义的问题
                  min_count=1, #去除词频小于1 的词汇
                  window=5)    #词向量上下文最大距离，该值越大，则上下文
                                #范围越广，默认值为5
word_vector = model.wv['明天'] #获取'明天'的词向量
print(word_vector)
print(model.wv.key_to_index)   #输出由所有词汇构成的词表（键-值对形式）
print(model.wv.index_to_key)  #输出由所有词汇构成的列表（list）
```

7.4 文本的表示



7.4.2 Word2vec 词向量

执行上述代码，输出如下结果：

```
[-0.14233617 0.12917742 0.17945977 -0.10030856 -0.07526746]  
{'展览': 0, '明天': 1, '图书': 2, '有': 3, '看': 4, '去': 5}  
['展览', '明天', '图书', '有', '看', '去']
```

这个例子告诉我们如何训练一个Word2vec 以及如何获得词的词向量等。

从这个例子大概也可以看出，Word2vec 模型是要利用已有的语料来预先训练，得到一个模型，然后提供给下游任务去使用。相对下游任务而言，这种词向量表示是静态的，无法保持与下游任务“与时俱进”。直觉告诉我们，这种与任务无关的词向量化方法在部分场景中可能大打折扣。当然，通过知识迁移方法，Word2vec 生成的向量也可以为其他表示学习提供初始化数据，从而可能加快相关任务的训练过程。

7.4 文本的表示



7.4.3 词嵌入表示

如今，比较盛行且被证明为有效的词向量化方法是与任务相关的**词嵌入**表示（Word embedding）。

词嵌入：要事先为每个词分配一个固定长度的向量，向量中的初始数值一般是通过随机初始化产生。这种向量也是低维、稠密的一种分布式表示。相对于由one-hot向量构成的稀疏的高维向量空间，由这种词向量构成的向量空间好像是嵌入到前者之中的一种稠密的低维的子空间，因而也称为**词嵌入空间**。

实际上，Word2vec 词向量构成的空间也是低维、稠密的词嵌入空间。但这种空间在任务执行之前就已经确定了的，是**静态的词向量空间**。

7.4 文本的表示



7.4.3 词嵌入表示

要介绍的是**动态的词嵌入**表示方法，这主要体现在：这种词嵌入向量（简称词向量）跟网络中的权值参数一样，在训练过程中它们也是被更新、被优化的；在训练收敛以后，所得到的向量才是相应词的向量表示。在这种方法中，词向量化过程和模型训练过程是同步进行的，模型收敛之时也是词向量化完成之时；且词向量化的结果跟具体的任务是相关，或者说这种方法是由任务同步驱动的词向量化方法。假设给定如下四个句子以及它们的类别信息：

```
sentences = ['明天去看展览', '今天加班，天气不好', '明天有图书展览', '明天去']  
labels = [1, 0, 1, 1] #0 表示一个类别，1 表示另一个类别，每个句子对应一个类别
```

构建分类模型及完成词嵌入式表示的主要步骤如下：

(1) **分词并建立词表**，代码如下：

```
sent_words = []  
vocab_dict = dict() #在Python 语言中用字典实现词表的功能  
max_len = 0 #保存最长句子的长度（词汇个数）
```

7.4 文本的表示



7.4.3 词嵌入表示

```
for sentence in sentences: #取出每一个句子
    words = list(jieba.cut(sentence)) #分词（用jieba 分词）
    if max_len < len(words): #保存最长句子的长度
        max_len = len(words)
    sent_words.append(words)
    for word in words: #统计词频
        vocab_dict[word] = vocab_dict.get(word, 0) + 1
sorted_vocab_dict = sorted(vocab_dict.items(), key=lambda kv: (kv[1], kv[0]), reverse=True)
sorted_vocab_dict = sorted_vocab_dict[:-2]
#<unk>、<pad>分别表示未知单词和填充用的单词
vocab_word2index = {'<unk>':0, '<pad>':1}
for word,_ in sorted_vocab_dict:
    if not word in vocab_word2index:
        vocab_word2index[word] = len(vocab_word2index) #构建单词的唯一编码
```

7.4 文本的表示



7.4.3 词嵌入表示

产生3 个中间结果：sent_words[], vocab_word2index, max_len，其中sent_words存放的是各序列构成的列表（sent_words 本身也是列表），各序列长度不一；vocab_word2index。是由各个词汇构成的词表，用于完成词的索引编码；max_len 用于保存最长句子的长度。它们的内容如下：

```
vocab_word2index = {'<unk>': 0, '<pad>': 1, '明天': 2, '展览': 3, '去': 4, '，': 5,  
                    '看': 6, '有': 7, '天气': 8, '图书': 9, '加班': 10}  
sent_words= [['明天', '去', '看', '展览'],  
              ['今天', '加班', '，', '，', '天气', '不好'],  
              ['明天', '有', '图书', '展览'],  
              ['明天', '去']]  
max_len = 5
```




7.4.3 词嵌入表示

(2) 完成序列的**索引编码**。

编码：用词表中的索引来表示序列中各个元素（词）的过程。由于每个索引都是整数，因而索引编码也称为整数编码。在编码过程中，可能还需要考虑等长化等问题。索引编码是词嵌入表示的一个前期步骤，在后面的学习过程中会逐步体会到。

先确定序列的长度，然后再利用词表对各个序列中的元素进行索引编码，接着用1填补长度不足 max_len 的序列，最后将各等长序列向量化。注意，只有进行了索引编码后才能进行张量化，所以索引编码就显得很重要，否则下面的步骤是无法进行的。

7.4 文本的表示



7.4.3 词嵌入表示

相关代码如下：

```
#设置序列的长度（可以设为最长句子的长度，也可以取平均值等）
max_len = int(max_len*0.9)
en_sentences = []
for words in sent_words:
    words = words[:max_len] #截取超过max_len 个元素的部分
    #对序列中的元素进行索引编码，0 表示未知词汇<unk>的编码
    en_words = [vocab_word2index.get(word, 0) for word in words]
    #对长度不足max_len 的序列，用1 填补，1 为填充词<pad>的索引
    en_words = en_words + [1] * (max_len - len(en_words))
    en_sentences.append(en_words)
#至此，形成每个序列等长的索引向量，保存在en_sentences[]中
sentences_tensor = torch.LongTensor(en_sentences) #转化为张量
labels_tensor = torch.LongTensor(labels)
```

7.4 文本的表示



7.4.3 词嵌入表示

上面代码得到两个张量：sentences_tensor 和 labels_tensor，它们的形状分别为(4, 4)和(4)。“(4, 4)”表示有4 个序列，每个序列中元素个数为4，“(4)”表示有4 个类标记，分别表示各个序列的类属。sentences_tensor 和 labels_tensor 的内容分别如下：

```
tensor([[ 2,  4,  6,  3],  
        [ 0, 10,  5,  8],  
        [ 2,  7,  9,  3],  
        [ 2,  4,  1,  1]])  
tensor([1, 0, 1, 1])
```

7.4 文本的表示



7.4.3 词嵌入表示

(3) **数据打包**：对数据进行打包，每包包含3 个序列。代码如下：

```
dataset = TensorDataset(sentences_tensor, labels_tensor)
dataloader = DataLoader(dataset=dataset, batch_size=3, shuffle=True)
```

由于总共只有4 个序列（句子），因此有一个包有3 个序列，另一个包只有1 个序列。

(4) **定义词嵌入向量空间**：为词表中每一个索引（整数）定义一个长度固定为20（也可以设置为40 等，看问题的复杂性而定）的向量，所有这些向量便构成了词嵌入向量空间。这需要定义嵌入层来实现，关键代码如下：

```
self.embedding = nn.Embedding(len(vocab_word2index), 20) #也可以设置宽度为40 等
```

7.4 文本的表示



7.4.3 词嵌入表示

其中，`nn.Embedding` 是 `nn` 模块提供的API。它实际上是创建了高度为`len(vocab_word2index)`、宽度为20 的二维实数矩阵。矩阵中的初始数据是随机初始化而形成的。也可以利用`Word2vec`等工具来初始化该矩阵，这实际上相当于应用知识迁移的方法。

例如，假设已有训练好的初始数据保存在同尺寸的张量`pretrained_weight` 中，则可以用下列语句将这些数据复制到该矩阵中（作为初始化数据）：

```
#pretrained_weight = torch.randn(len(vocab_word2index), 20) #模拟训练好的数据  
self.embedding.weight.data.copy_(pretrained_weight)
```

张量`sentences_tensor` 中的索引（正整数）正是该矩阵的行下标值，或者说，正是通过该下标值将各个词和矩阵中的行向量关联起来。需要注意的是，该矩阵中的参数跟网络权值参数一样，在训练过程中不断得到更新和优化，直到训练过程结束（收敛）后，这种更新才中止。此时，每一个行向量就成为相应词的向量表示。

7.4 文本的表示



7.4.3 词嵌入表示

嵌入向量空间需要和LSTM 一起训练， 因此它们一般放在一个类当中。这里类的定义如下：

```
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.embedding = nn.Embedding(len(vocab_word2index), 20) #嵌入层
        self.lstm = nn.LSTM(input_size=20, hidden_size=28, num_layers = 1,
                             batch_first = False, bidirectional = False, bias = True)
        self.fc = nn.Linear(28, 2)
    def forward(self, x): #torch.Size([3, 4])
        o = self.embedding(x) #torch.Size([3, 4])变为torch.Size([3, 4, 20])
        o, (_, _) = self.lstm(o)
        o = torch.sum(o, dim=1) #将各单元的输出简单相加， 形成当前序列x 的特征
        o = self.fc(o)
        return o
```

7.4 文本的表示

7.4.3 词嵌入表示

在上述代码中，理解“ $o = \text{self.embedding}(x)$ ”是关键。首先，这里的 x 是对句子进行索引编码后形成的索引张量（张量中的元素为索引，即正整数）。如果输入 x 的形状为 $(3, 4)$ ，则输出的 o 的形状为 $(3, 4, 20)$ 。可以这样理解这个变换过程：输入 x 包含3 个序列，每个序列由4 索引（正整数）组成；在经过嵌入层 self.embedding 以后，序列的总量3 保持不变，但序列中每个索引被替换成了长度为20 的数值向量。替换的方法是，以索引为行下标，在上述二维实数矩阵中找到对应的行向量，然后以此向量来替换 x 中的索引即可；执行所有这样的替换后，输出 o 的形状就变成了 $(3, 4, 20)$ 。

输入 x 在经过嵌入层 self.embedding 以后，才真正完成词从符号到向量的转变（期间经过了索引编码的过程）。此后，虽然向量中的数值还在不断被更新，但是在编程思维上可以用向量来代替相应的词了。

7.4 文本的表示



7.4.3 词嵌入表示

(5) 训练模型，形成最终的词嵌入向量空间。训练代码如下：

```
lstm_model = Model() #创建实例
optimizer = torch.optim.Adam(lstm_model.parameters(), lr=0.01)
for ep in range(100): #训练100 轮
    for i, (batch_texts, batch_labels) in enumerate(dataloader):
        output = lstm_model(batch_texts)
        #本例是个分类任务，用交叉信息上计算损失函数值
        loss = CrossEntropyLoss()(output, batch_labels)
        print(round(loss.item(), 4))
        optimizer.zero_grad() #更新参数，包括嵌入向量中的参数
        loss.backward()
        optimizer.step()
```

7.4 文本的表示



7.4.3 词嵌入表示

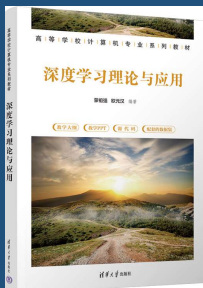
词嵌入向量空间中的数据一般是随机初始化得到的，或者是从别的地方迁移过来的，需要通过训练并待模型收敛后，形成的最终向量才是正确的词向量。

经过训练并待模型收敛后，可仿造下列代码输出任何一个词的词向量：

```
embedding = lstm_model.embedding.weight.clone() #将词嵌入向量空间转化为张量  
index = vocab_word2index['明天'] #查看'明天'的词向量  
print( np.array(embedding[index].data) )
```

上述代码用于输出'明天'的词向量，结果得到如下的向量：

```
(0.886, -1.8081, 0.5094, -0.3197, 0.2217, -0.6586, 0.6308, -0.379, -0.2269, 0.267, -1.2014,  
1.5571, -1.4393, -0.3958, 0.3539, 0.4668, -1.4754, -0.3781, 0.7225, 0.1311)
```



本章内容

contents

7.1 一个简单的循环神经网络——航空旅客出行人数预测

7.2 循环神经网络

7.3 长短时记忆网络(LSTM)

7.4 文本的表示

7.5 基于LSTM的文本分类

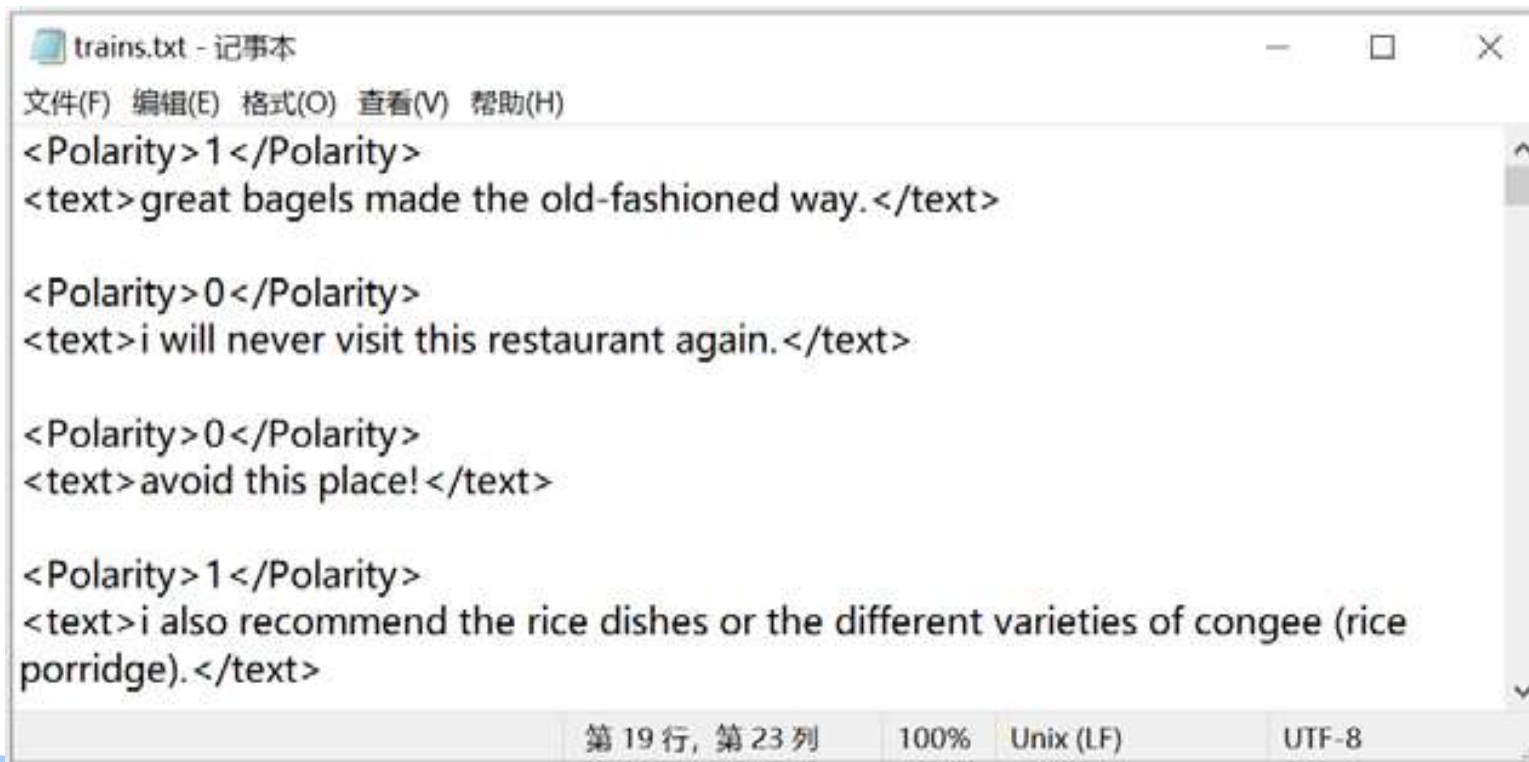
7.6 基于LSTM的文本生成

7.5 基于LSTM的文本分类



【例7.3】 对给定的英文文本及其分类标记，构建一个基于LSTM 的文本分类模型。

本例使用的数据集是一个英文文本数据集，保存在./data/corpus 目录下。数据集中每一条英文文本都放在标记对<text></text> 之间， 相应的分类标记则放在标记对<Polarity></Polarity>之间，该标记对位于标记对<text></text>的前面。相应结构如下图所示。



```
trains.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
<Polarity>1</Polarity>
<text>great bagels made the old-fashioned way.</text>

<Polarity>0</Polarity>
<text>i will never visit this restaurant again.</text>

<Polarity>0</Polarity>
<text>avoid this place!</text>

<Polarity>1</Polarity>
<text>i also recommend the rice dishes or the different varieties of congee (rice
porridge).</text>
```

第 19 行, 第 23 列 100% Unix (LF) UTF-8

7.5 基于LSTM的文本分类



1. 构建序列数据集和单词词表

定义函数`get_dataset_vocab()`，其功能是按空格对英文句子进行拆分，每一个英文句子形成一个列表，该函数返回由所有这样的列表组成的列表。此外，还返回由各个单词构成的词表，以用于单词的索引编码。

`def get_dataset_vocab(fn):` #获取文本数据集，和单词词表（完整代码见教材207页）

2.对英文文本进行索引编码

对英文文本进行索引编码，并进行数据打包，允许不同的数据批量采用不同的序列长度。

定义函数`text_loader()`，其作用是利用函数`get_dataset_vocab()`形成的数据集和单词词表，对英文文本进行索引编码。此外，对数据进行“组装”，形成数据包（批量）；同一个数据包中，以最长句子的长度作为当前数据包中每个向量的长度，不够的就填补1。

（函数`text_loader()`的代码见教材208页）

7.5 基于LSTM的文本分类



3. 定义实现分类任务的类——类Lstm_model

利用nn.Embedding()构建词向量空间，利用nn.LSTM()来搭建循环神经网络，这些都封装在类Lstm_model 当中

```
class Lstm_model(nn.Module):
    def __init__(self,vocab_size,embedding_dim,hidden_dim):
        super(Lstm_model, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim) #定义嵌入层
        self.lstm = nn.LSTM(input_size=embedding_dim, hidden_size=hidden_dim, \
                             num_layers=1, batch_first=True, bidirectional=False, bias=True)
        self.fc = nn.Linear(hidden_dim,1)
    def forward(self, x):
        o = x
        o = self.embedding(o)
        o, _ = self.lstm(o)
        o = o.sum(1)
        o = self.fc(o)
        return o
```

7.5 基于LSTM的文本分类



在上述代码中，创建了尺寸为 $\text{vocab_size} \times \text{embedding_dim}$ 的词向量空间，即一共有 vocab_size 个不同的单词，每个单词用长度为 embedding_dim 的向量来表示。LSTM的定义代码表明，在输入张量 x 中，张量的第一维必须是批量大小的信息，第二维是序列长度信息，词向量的长度为 embedding_dim 。

4. 加载数据，并进行训练

调用函数`get_dataset_vocab()`和函数`text_loader()`加载数据并进行打包，然后实例化类`Lstm_model`，执行训练。

```
for ep in range(100):
    for text_batch, label_batch in zip(train_batches, train_labels):
        text_batch, label_batch = text_batch.to(device), label_batch.to(device)
        pre_y = lstm_model(text_batch)
        #二分类问题，本例采用逻辑回归方法来解决
        loss = nn.BCEWithLogitsLoss()(pre_y.squeeze(), label_batch.float())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        print(ep, round(loss.item(),4))
```

(完整代码见教材209页)

7.5 基于LSTM的文本分类



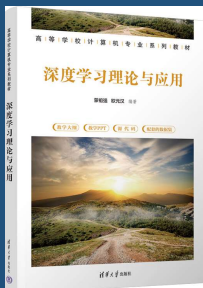
5. 模型测试

利用测试数据，对模型的分类准确率进行测试。代码如下：

```
lstm_model.eval()
acc = 0
for text_batch, label_batch in zip(test_batches, test_labels):
    text_batch, label_batch = text_batch.to(device), label_batch.to(device)
    pre_y = lstm_model(text_batch)
    pre_y = torch.sigmoid(pre_y)
    pre_y = torch.round(pre_y) #四舍五入，变成了0 或1
    pre_y = pre_y.squeeze().long()
    correct = torch.eq(pre_y, label_batch).long()
    acc += correct.sum()
print('在测试集上的准确率： {:.1f}%'.format(100.*acc/len(test_examples)))
```

运行上述代码，结果输出如下的测试结果：

在测试集上的准确率： 80.0%



本章内容

contents

7.1 一个简单的循环神经网络——航空旅客出行人数预测

7.2 循环神经网络

7.3 长短时记忆网络(LSTM)

7.4 文本的表示

7.5 基于LSTM的文本分类

7.6 基于LSTM的文本生成

7.6 基于LSTM 的文本生成



廣西大學
GUANGXI UNIVERSITY

7.6.1.语言模型与文本生成

语言模型：对任何给定一个句子前面连续的若干个词（或单词），该模型可以预测紧跟这些词后面最可能出现的词。

比如，观察前面提及的例子：

他超额完成了任务，经理表扬了____。

假设这句话所在的上下文包含的全部词汇为：'他'、'超额'、'完成'、'了'、'任务'、','、'、'、'经理'、'表扬'（'一共 8 个词汇），那么语言模型只能从这 8 个词汇中选择一个放在下划线上，实际上是预测这些词出现在下划线上的概率，其本质属于分类问题。因而可以换一种思路：把每个词汇看成是一个类别，这个选词填空的问题便是一个 8 分类问题。例如，如果上面这句文本属于'他'这个类，那么下划线上应该放上'他'；如果属于'超额'，则应该放上'超额'，等。由此可见，这种语言模型可以转化为多分类问题。但这里还遇到两个问题：

- (1) 如何构造训练数据；
- (2) 有的词汇出现频率很高，从而导致类不平衡问题。

7.6 基于LSTM 的文本生成

7.6.1.语言模型与文本生成

对于问题（1），可以利用已有的每个句子来构造训练数据。例如，假设已有一个句子：“张三超额完成了任务”，并假设序列的长度设置为4，则先在该句子分词后形成的列表的头部和尾部分别添加特殊标识符'<s>'和'<e>'（它们分别表示句子的开始和结束），得到这样的分词列表：['<e>', '张三', '超额', '完成', '了', '任务', '<e>'], 然后据此进一步构造如表所示的训练样本集。

序号	等长序列				类标记
1	'<s>'	'<pad>'	'<pad>'	'<pad>'	'张三'
2	'<s>'	'张三'	'<pad>'	'<pad>'	'超额'
3	'<s>'	'张三'	'超额'	'<pad>'	'完成'
4	'<s>'	'张三'	'超额'	'完成'	'了'
5	'张三'	'超额'	'完成'	'了'	'任务'
6	'超额'	'完成'	'了'	'任务'	'<e>'

7.6 基于LSTM 的文本生成



廣西大學
GUANGXI UNIVERSITY

7.6.1.语言模型与文本生成

这样构造训练数据的目的是希望模型能够正确预测任意序列后面的词。例如，当输入为'<s>张三'时，希望模型能够输出'超额'；当输入为'张三超额完成了'时，希望模型能够输出'任务'；当输入为'超额完成了任务'时，希望模型能够输出'<e>'（表示文本生成过程结束），等等。

可以看到，如果一个句子包含 n 个词，则可以生成 $n+1$ 条训练样本。这样，利用所有的句子便可以生成大量带标记的训练样本，而不需人工对样本进行标注。

7.6 基于LSTM 的文本生成



7.6.2. 类不平衡问题

由于词表中每个词都是一个类，所以类别就很多，这需要大量的训练样本。而且，各词出现的频率差别很大（如有的出现几百次，而有的只出现一次），因此会出现类不平衡问题。

对于问题（2），在样本已经客观存在类不平衡的情况下，可以考虑用Weighted Random Sampler 类来缓解。WeightedRandomSampler 类用于有回放的加权随机采样，产生样本的索引，然后再结合DataLoader 类完成采样功能。WeightedRandomSampler 类的调用格式如下：

```
sampler = WeightedRandomSampler(weights, replacement, num_samples)
```

其中，weights 为权值向量，其分量要跟数据集中的样本一一对应，表示相应样本的权重；replacement 应设置为True，表示有回放的抽样；num_samples 用于设置要抽样的样本数量。一个样本被抽中的次数在概率上正比于在向量weights 中设置的对应权重。

WeightedRandomSampler 返回的是样本的索引，取值范围为 $\{0, 1, \dots, \text{len}(\text{weight}) - 1\}$ 。

7.6 基于LSTM 的文本生成



7.6.2. 类不平衡问题

例如，下列代码先设置了有3 个分量的权重向量，然后调用WeightedRandomSampler来生成10 个样本的索引：

```
weights = [2,5,3]
sampler = WeightedRandomSampler(weights=weights, replacement=True,num_samples=10)
print(list(sampler))
```

执行上述代码，输出如下的结果：

```
[1, 1, 2, 1, 2, 0, 2, 0, 1, 1]
```

由于weights 中只有3 个分量，因此生成索引的范围为{0,1,2}。从结果中大致可以看出，索引0、1 和2 出现的概率分别为0.2、0.5 和0.3，分别等于2/10、5/10 和3/10。这也证实“一个样本被抽中的次数在概率上正比于在向量weights 中设置的对应权重”。

7.6 基于LSTM 的文本生成



7.6.2. 类不平衡问题

对于类不平衡问题，我们希望抽取更多小类的样本，以期在数量上保持跟大类样本大抵平衡。为此，可以先统计各类样本的数量，然后将类样本数的倒数设置为相应样本的权值，从而形成权重向量weights。

例如，下面代码先统计各个类别所包含的样本数量，然后将这个数量的倒数作为对应样本的权重，形成权重向量weights，进而结合WeightedRandomSampler 和 DataLoader，在数据集中有放回地抽取10000 条样本，并保持各个类别的样本在数量上相对平衡：

```
class_dict = dict()
for label in labels: #统计各类别词汇出现的频次
    lb = label.item()
    class_dict[lb] = class_dict.get(lb, 0) + 1
weights = []
for label in labels:
    lb = label.item()
    weights.append(class_dict[lb]) #保存样本所在类别的样本数
```

(完整代码见教材212页)

7.6 基于LSTM 的文本生成



7.6.3. 文本生成案例

【例7.4】 开发一个能自动写小说的程序。

(1) 从网上下载金庸小说的部分文本，并保存在./data 目录下的文件“金庸小说节选.txt”中，按照前面介绍的方法构建训练集和打包，并解决类不平衡问题。

完整代码见教材212页

```
path = r'./data'
name = r'金庸小说节选.txt'
fn = path+'/' + name
sentence_words, vocab_word2index = get_texts_vocab(fn) #构建数据集和编码词表
texts, labels = enAllTxts(sentence_words, vocab_word2index) #生成训练数据
vocab_index2word = { index:word for word,index in vocab_word2index.items()} #用于解码
dataset = TensorDataset(texts, labels)
#下面代码主要解决类不平衡问题:
class_dict = dict()
for label in labels:
    lb = label.item()
    class_dict[lb] = class_dict.get(lb, 0) + 1 #统计各类别词汇出现的频次
weights = [] #跟dataloader.dataset 中的数据行要一一对应
for label in labels:
    lb = label.item()
    weights.append(class_dict[lb])
weights = 1./torch.FloatTensor(weights)
sampler = WeightedRandomSampler(weights=weights,
                                replacement=True,num_samples=len(labels)*1000) #解决类不平衡问题
dataloader = DataLoader(dataset=dataset, batch_size=128, sampler=sampler, shuffle=False)
```

7.6 基于LSTM 的文本生成



7.6.3. 文本生成案例

(2) 定义自动生成文本类——Novel_mode 类。

```
class Novel_model(nn.Module):
    def __init__(self,vocab_size):
        super(Novel_model,self).__init__()
        self.embedding = nn.Embedding(vocab_size, 256)
        self.lstm = nn.LSTM(input_size=256, hidden_size=256,\
batch_first=True, num_layers=1, bidirectional = False)
        self.fc1 = nn.Linear(256, 512)
        self.fc2 = nn.Linear(512, vocab_size) #多分类问题.一共有1524 个类别
    def forward(self, x):
        o = x
        o = self.embedding(o)
        o, _ = self.lstm(o)
        o = torch.sum(o, dim=1)
        o = self.fc1(o)
        o = torch.relu(o)
        o = self.fc2(o)
        return o
```

7.6 基于LSTM 的文本生成



7.6.3. 文本生成案例

(3) 训练文本自动生成模型，代码如下。完整代码见教材214页

```
novel_model = Novel_model(vocab_size=len(vocab_word2index)).to(device)
optimizer = torch.optim.Adam(novel_model.parameters(), lr=0.01)
for ep in range(5): #迭代5 轮
    for i, (batch_texts, batch_labels) in enumerate(dataloader):
        batch_texts, batch_labels = \
            batch_texts.to(device), batch_labels.to(device)
        #torch.Size([128, 10]) ---> torch.Size([128, 1524])
        batch_out = novel_model(batch_texts)
        #多分类，用交叉熵损失函数
        loss = nn.CrossEntropyLoss()(batch_out, batch_labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if i%500==0:
            print(ep, round(loss.item(),8))
torch.save(novel_model, 'novel_model')
```

7.6 基于LSTM 的文本生成



廣西大學
GUANGXI UNIVERSITY

7.6.3. 文本生成案例

(4) 加上下面的模块引入代码:

```
import numpy as np
import torch
from torch import nn, optim
from torch.utils.data import TensorDataset, DataLoader, WeightedRandomSampler
import jieba
jieba.setLogLevel(jieba.logging.INFO) #屏蔽jieba 分词时出现的提示信息
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

7.6 基于LSTM 的文本生成



7.6.3. 文本生成案例

(5) 执行由上述代码构成的.py 文件， 会生成模型`novel_model` 以及词表`vocab_word2index` 和`vocab_index2word`， 它们都保存在磁盘上。

对给定由不超过10 个词构成的序列（开始标识符为'`<s>`'）， 模型`novel_model` 可以生成这个序列的下一个词。这样， 我们可以按照下面步骤来生成小说文本：

- ①在最开始时我们可以给模型输入由一两个词构成的序列， 假设为序列`seq`;
- ②待模型输出下一个词`w` 后， 如果`w='<e>'`， 则退出， 表示文本生成过程结束， 否则把该词加到原序列后面得到新的序列， 即 $seq \leftarrow seq + w$;
- ③截取新序列后的10 个词构成的序列， 即 $seq \leftarrow seq[-10:]$ ， 然后将`seq` 输入到该模型中并转②。

7.6 基于LSTM 的文本生成



7.6.3. 文本生成案例

基于模型`novel_model`，生成小说文本的实现代码如下。完整代码见教材P212-216

```
def getNextWord(s): #调用模型， 给定一个词序列， 生成它的下一个词
    words = list(jieba.cut(s))
    words = ['<s>'] + words
    en_words = [vocab_word2index.get(word,0) for word in words]
    en_words = en_words[len(en_words)-10:len(en_words)]
    en_words = en_words + [1]*(10-len(en_words))
    batch_texts = torch.LongTensor(en_words).unsqueeze(0).to(device)
    batch_out = novel_model(batch_texts)
    batch_out = torch.softmax(batch_out, dim=1)
    pre_index = torch.argmax(batch_out, dim=1)
    word = vocab_index2word[pre_index.item()]
    return word
```


7.6 基于LSTM 的文本生成

7.6.3. 文本生成案例

执行上述代码，在笔者计算机上输出如下结果：

黄蓉在睡梦之中见到他，只见後一个人。

可以看到，本例程序“写”的小说文本似乎还像样，至少不是胡乱生成文本。但生成的句子还不太畅通，远达不到人类的水平。

本例子只是想阐明文本生成的基本原理。通过本例子，可以掌握文本生成的基本方法。但要开发一个实用的文本生成程序，还需做更多的努力和尝试，包括加长序列的长度、下载更多的训练数据、提供强大的算力支撑、做大量的调参工作等。



本章内容：

- 循环神经网络
- 长短期记忆网络LSTM
- 文本表示
- 基于LSTM的文本分类
- 基于LSTM的文本生成