RNN

- Models for Sequences

RNN

- RNN是神经网络中的一种，它擅长对序列数据进行建模处理。循环神经网络（Recurrent Neural Network, RNN）是一类以序列（sequence）数据为输入，在序列的演进方向进行递归（recursion）且所有节点（循环单元）按链式连接的递归神经网络（recursive neural network）。

语言模型

- 语言模型（language model）是自然语言处理的重要技术。自然语言处理中最常见的数据是文本数据。我们可以把一段自然语言文本看作一段离散的时间序列。假设一段长度为T的文本中的词依次为$w_1, w_2, \cdots\cdots, w_T$，那么在离散的时间序列中，$w_t$（$1 \leqslant t \leqslant T$）可看作在时间步（time step）t的输出或标签。给定一个长度为T的词的序列$w_1, w_2, \cdots\cdots, w_T$，语言模型将计算该序列的概率：

$$P(w_1, w_2, \cdots\cdots, w_T).$$

语言模型

- 语言模型可用于提升语音识别和机器翻译的性能。例如，在语音识别中，给定一段“厨房里食油用完了”的语音，有可能会输出“厨房里食油用完了”和“厨房里石油用完了”这两个读音完全一样的文本序列。如果语言模型判断出前者的概率大于后者的概率，我们就可以根据相同读音的语音输出“厨房里食油用完了”的文本序列。在机器翻译中，如果对英文“you go first”逐词翻译成中文的话，可能得到“你走先”“你先走”等排列方式的文本序列。如果语言模型判断出“你先走”的概率大于其他排列方式的文本序列的概率，我们就可以把“you go first”翻译成“你先走”。

语言模型的计算

- 既然语言模型很有用，那该如何计算它呢？假设序列$w_1, w_2, \ldots, w_T$中的每个词是依次生成的，我们有 $P(w_1, w_2, \ldots, w_T) \approx \prod_{t=1}^{T} P(w_t \mid w_{t-(n-1)}, \ldots, w_{t-1}).$
- 例如，一段含有4个词的文本序列的概率

$$P(w_1, w_2, w_3, w_4) = P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_1, w_2)P(w_4 \mid w_1, w_2, w_3).$$

语言模型的计算

- 为了计算语言模型，我们需要计算词的概率，以及一个词在给定前几个词的情况下的条件概率，即语言模型参数。设训练数据集为一个大型文本语料库，如维基百科的所有条目。词的概率可以通过该词在训练数据集中的相对词频来计算。例如，$P(w_1)$可以计算为w_1在训练数据集中的词频（词出现的次数）与训练数据集的总词数之比。因此，根据条件概率定义，一个词在给定前几个词的情况下的条件概率也可以通过训练数据集中的相对词频计算。例如，$P(w_2 \mid w_1)$可以计算为w_1, w_2两词相邻的频率与w_1词频的比值，因为该比值即$P(w_1, w_2)$与$P(w_1)$之比；而$P(w_3 \mid w_1, w_2)$同理可以计算为w_1、w_2和w_3三词相邻的频率与w_1和w_2两词相邻的频率的比值。

n元语法

- 当序列长度增加时，计算和存储多个词共同出现的概率的复杂度会呈指数级增加。n元语法通过马尔可夫假设（虽然并不一定成立）简化了语言模型的计算。这里的马尔可夫假设是指一个词的出现只与前面n个词相关，即n阶马尔可夫链（Markov chain of order n）。如果n=1，那么有$P(w_3 \mid w_1, w_2) = P(w_3 \mid w_2)$。如果基于n−1阶马尔可夫链，我们可以将语言模型改写为

$$P(w_1, w_2, \ldots, w_T) \approx \prod_{t=1}^{T} P(w_t \mid w_{t-(n-1)}, \ldots, w_{t-1}).$$

n元语法

- 以上也叫n元语法（n-grams）。它是基于n−1阶马尔可夫链的概率语言模型。当n分别为1、2和3时，我们将其分别称作一元语法（unigram）、二元语法（bigram）和三元语法（trigram）。例如，长度为4的序列w_1,w_2,w_3,w_4在一元语法、二元语法和三元语法中的概率分别为

$$
\begin{aligned}
P(w_1, w_2, w_3, w_4) &= P(w_1)P(w_2)P(w_3)P(w_4),\\
P(w_1, w_2, w_3, w_4) &= P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_2)P(w_4 \mid w_3),\\
P(w_1, w_2, w_3, w_4) &= P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_1, w_2)P(w_4 \mid w_2, w_3).
\end{aligned}
$$

循环神经网络

- n元语法中，时间步 t 的词 w_t 基于前面所有词的条件概率只考虑了最近时间步的 n-1 个词。如果要考虑比 t-(n-1) 更早时间步的词对 w_t 的可能影响，我们需要增大 n。但这样模型参数的数量将随之呈指数级增长。
- 本节将介绍循环神经网络。它并非刚性地记忆所有固定长度的序列，而是通过隐藏状态来存储之前时间步的信息。首先我们回忆一下前面介绍过的多层感知机，然后描述如何添加隐藏状态来将它变成循环神经网络。

不含隐藏状态的神经网络

- 让我们考虑一个含单隐藏层的多层感知机。给定样本数为 n、输入个数（特征数或特征向量维度）为 d 的小批量数据样本 $\boldsymbol{X} \in \mathbb{R}^{n \times d}$。设隐藏层的激活函数为 ϕ，那么隐藏层的输出 $\boldsymbol{H} \in \mathbb{R}^{n \times h}$ 计算 $\boldsymbol{H} = \phi(\boldsymbol{X}\boldsymbol{W}_{xh} + \boldsymbol{b}_h),$
- 其中隐藏层权重参数 $\boldsymbol{W}_{xh} \in \mathbb{R}^{d \times h}$，隐藏层偏差参数 $\boldsymbol{b}_h \in \mathbb{R}^{1 \times h}$，h为隐藏单元个数。上式相加的两项形状不同，因此将按照广播机制相加。把隐藏变量 $\boldsymbol{H}$ 作为输出层的输入，且设输出个数为q（如分类问题中的类别数），输出层的输出为 $\boldsymbol{O} = \boldsymbol{H}\boldsymbol{W}_{hq} + \boldsymbol{b}_q,$
- 其中输出变量 $\boldsymbol{O} \in \mathbb{R}^{n \times q}$, 输出层权重参数 $\boldsymbol{W}_{hq} \in \mathbb{R}^{h \times q}$, 输出层偏差参数 $\boldsymbol{b}_q \in \mathbb{R}^{1 \times q}$。如果是分类问题，我们可以使用 softmax($\boldsymbol{O}$) 来计算输出类别的概率分布。

含隐藏状态的循环神经网络

- 现在我们考虑输入数据存在时间相关性的情况。假设 $\boldsymbol{X}_t \in \mathbb{R}^{n \times d}$ 是序列中时间步 t 的小批量输入，$\boldsymbol{H}_t \in \mathbb{R}^{n \times h}$ 是该时间步的隐藏变量。与多层感知机不同的是，这里我们保存上一时间步的隐藏变量 $\boldsymbol{H}_{t-1}$，并引入一个新的权重参数 $\boldsymbol{W}_{hh} \in \mathbb{R}^{h \times h}$，该参数用来描述在当前时间步如何使用上一时间步的隐藏变量。具体来说，时间步 t 的隐藏变量的计算由当前时间步的输入和上一时间步的隐藏变量共同决定：$\boldsymbol{H}_t = \phi(\boldsymbol{X}_t \boldsymbol{W}_{xh} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hh} + \boldsymbol{b}_h).$

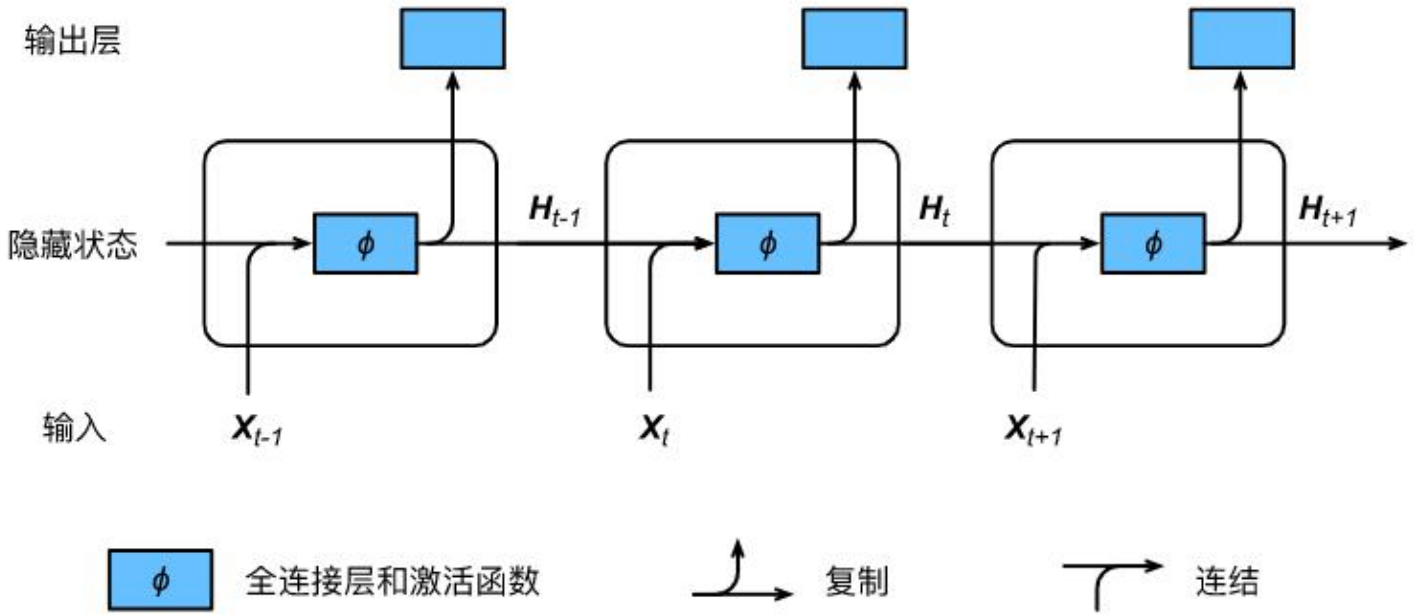
- 与多层感知机相比，我们在这里添加了 $\boldsymbol{H}_{t-1} \boldsymbol{W}_{hh}$ 一项。由上式中相邻时间步的隐藏变量 $\boldsymbol{H}_t$ 和 $\boldsymbol{H}_{t-1}$ 之间的关系可知，这里的隐藏变量能够捕捉截至当前时间步的序列的历史信息，就像是神经网络当前时间步的状态或记忆一样。因此，该隐藏变量也称为隐藏状态。由于隐藏状态在当前时间步的定义使用了上一时间步的隐藏状态，上式的计算是循环的。使用循环计算的网络即循环神经网络（recurrent neural network）。

含隐藏状态的循环神经网络

- 在时间步t，输出层的输出和多层感知机中的计算类似：$\boldsymbol{O}_t = \boldsymbol{H}_t \boldsymbol{W}_{hq} + \boldsymbol{b}_q.$
- 循环神经网络的参数包括隐藏层的权重$\boldsymbol{W}_{xh} \in \mathbb{R}^{d \times h}$、$\boldsymbol{W}_{hh} \in \mathbb{R}^{h \times h}$和偏差 $\boldsymbol{b}_h \in \mathbb{R}^{1 \times h}$，以及输出层的权重$\boldsymbol{W}_{hq} \in \mathbb{R}^{h \times q}$和偏差$\boldsymbol{b}_q \in \mathbb{R}^{1 \times q}$。值得一提的是，即便在不同时间步，循环神经网络也始终使用这些模型参数。因此，循环神经网络模型参数的数量不随时间步的增加而增长。

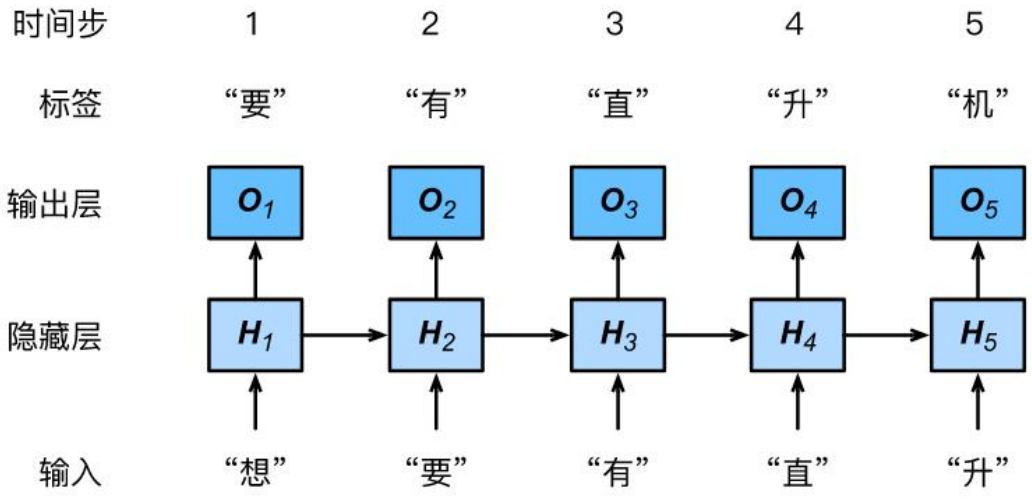
含隐藏状态的循环神经网络

- 下图展示了循环神经网络在 3 个相邻时间步的计算逻辑。在时间步 t，隐藏状态的计算可以看成是将输入 $\boldsymbol{X}_t$ 和前一时间步隐藏状态 $\boldsymbol{H}_{t-1}$ 连结后输入一个激活函数为 ϕ 的全连接层。该全连接层的输出就是当前时间步的隐藏状态 $\boldsymbol{H}_t$，且模型参数为 $\boldsymbol{W}_{xh}$ 与 $\boldsymbol{W}_{hh}$ 的连结，偏差为 $\boldsymbol{b}_h$。当前时间步 t 的隐藏状态 $\boldsymbol{H}_t$ 将参与下一个时间步 t+1 的隐藏状态 $\boldsymbol{H}_{t+1}$ 的计算，并输入到当前时间步的全连接输出层。



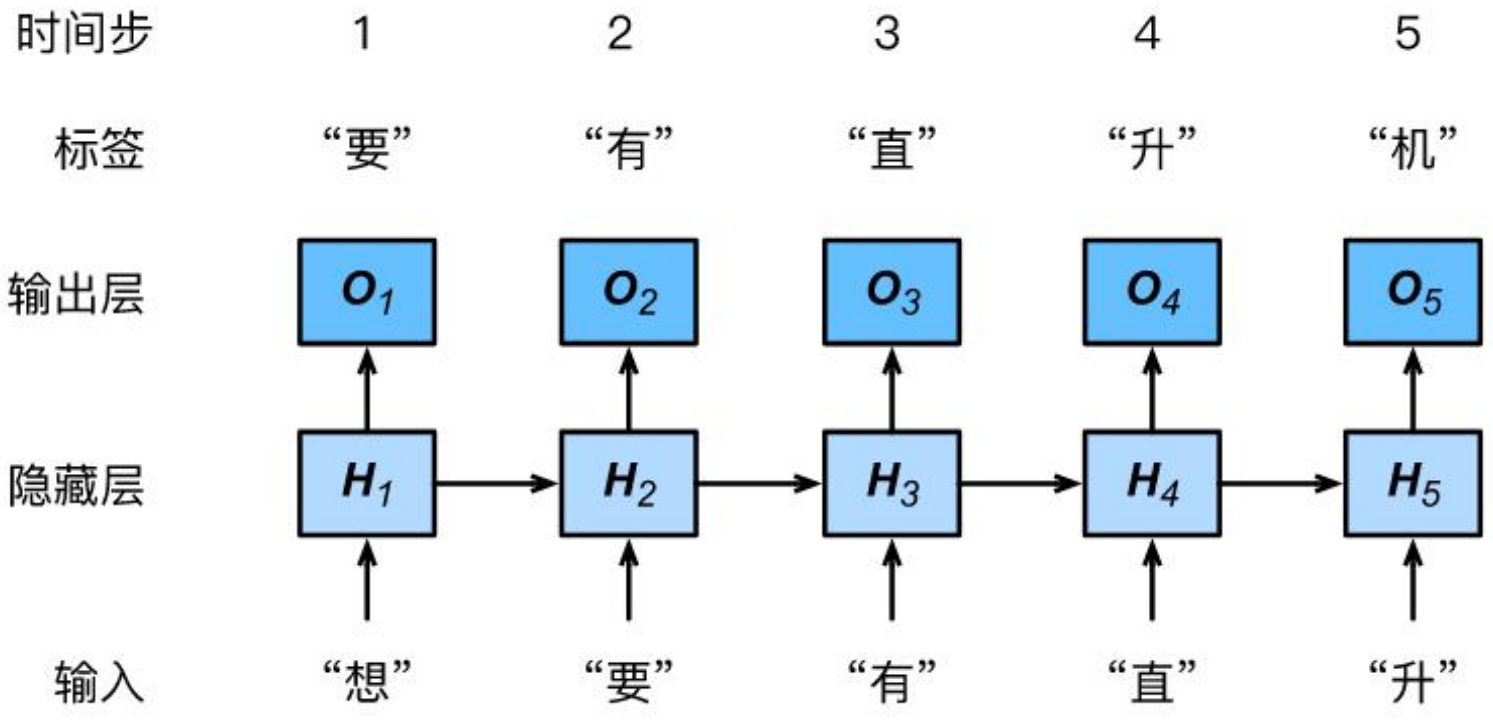
应用：基于字符级循环神经网络的语言模型

- 我们介绍如何应用循环神经网络来构建一个语言模型。设小批量中样本数为1，文本序列为“想”“要”“有”“直”“升”“机”。下图演示了如何使用循环神经网络基于当前和过去的字符来预测下一个字符。在训练时，我们对每个时间步的输出层输出使用softmax运算，然后使用交叉熵损失函数来计算它与标签的误差。在图中，由于隐藏层中隐藏状态的循环计算，时间步 3 的输出 O_3 取决于文本序列“想”“要”“有”。由于训练数据中该序列的下一个词为“直”，时间步 3 的损失将取决于该时间步基于序列“想”“要”“有”生成下一个词的概率分布与该时间步的标签“直”。



应用：基于字符级循环神经网络的语言模型

- 因为每个输入词是一个字符，因此这个模型被称为字符级循环神经网络（character-level recurrent neural network）。因为不同字符的个数远小于不同词的个数（对于英文尤其如此），所以字符级循环神经网络的计算通常更加简单。接下来将介绍它的具体实现。



语言模型数据集（周杰伦专辑歌词）

- 我们介绍如何预处理一个语言模型数据集，并将其转换成字符级循环神经网络所需要的输入格式。为此，我们收集了周杰伦从第一张专辑《Jay》到第十张专辑《跨时代》中的歌词，并在后面几节里应用循环神经网络来训练一个语言模型。当模型训练好后，我们就可以用这个模型来创作歌词。首先读取这个数据集，看看前40个字符是什么样的。

```
with zipfile.ZipFile('../../data/jaychou_lyrics.txt.zip') as zin:
    with zin.open('jaychou_lyrics.txt') as f:
        corpus_chars = f.read().decode('utf-8')
corpus_chars[:40]
```

- 输出：

```
'想要有直升机\n想要和你飞到宇宙去\n想要和你融化在一起\n融化在宇宙里\n我每天每天每'
```

建立字符索引

- 我们将每个字符映射成一个从0开始的连续整数，又称索引，方便之后的数据处理。为了得到索引，我们将数据集里所有不同字符取出来，然后将其逐一映射到索引来构造词典。接着，打印vocab_size，即词典中不同字符的个数，又称词典大小。

```
idx_to_char = list(set(corpus_chars))
char_to_idx = dict([(char, i) for i, char in enumerate(idx_to_char)])
vocab_size = len(char_to_idx)
vocab_size # 1027
```

建立字符索引

- 之后，将训练数据集中每个字符转化为索引，并打印前20个字符及其对应的索引。

```
corpus_indices = [char_to_idx[char] for char in corpus_chars]
sample = corpus_indices[:20]
print('chars:', ''.join([idx_to_char[idx] for idx in sample]))
print('indices:', sample)
```

- 输出：

```
chars: 想要有直升机 想要和你飞到宇宙去 想要和
indices: [250, 164, 576, 421, 674, 653, 357, 250, 164, 850, 217, 910, 1012, 261, 275,
  366, 357, 250, 164, 850]
```

时序数据的采样

- 在训练中我们需要每次随机读取小批量样本和标签。与之前章节的实验数据不同的是，时序数据的一个样本通常包含连续的字符。假设时间步数为5，样本序列为5个字符，即“想”“要”“有”“直”“升”。该样本的标签序列为这些字符分别在训练集中的下一个字符，即“要”“有”“直”“升”“机”。我们有两种方式对时序数据进行采样，分别是随机采样和相邻采样。

随机采样

- 下面的代码每次从数据里随机采样一个小批量。其中批量大小batch_size指每个小批量的样本数，num_steps为每个样本所包含的时间步数。 在随机采样中，每个样本是原始序列上任意截取的一段序列。相邻的两个随机小批量在原始序列上的位置不一定相毗邻。因此，我们无法用一个小批量最终时间步的隐藏状态来初始化下一个小批量的隐藏状态。在训练模型时，每次随机采样前都需要重新初始化隐藏状态。

```
def data_iter_random ( corpus_indices, batch_size, num_steps, device=None):
    num_examples = (len(corpus_indices) - 1) // num_steps
    epoch_size = num_examples // batch_size
    example_indices = list(range(num_examples))
    random.shuffle(example_indices)

    def _data(pos): # 返回从pos开始的长为num_steps的序列
        return corpus_indices[pos: pos + num_steps]
    if device is None:
        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    for i in range(epoch_size):
        i = i * batch_size # 每次读取batch_size个随机样本
        batch_indices = example_indices[i: i + batch_size]
        X = [_data(j * num_steps) for j in batch_indices]
        Y = [_data(j * num_steps + 1) for j in batch_indices]
        yield torch.tensor(X, dtype=torch.float32, device=device), torch.tensor(Y, dtype=torch.float32, device=device)
```

- 让我们输入一个从0到29的连续整数的人工序列。设批量大小和时间步数分别为2和6。打印随机采样每次读取的小批量样本的输入X和标签Y。可见，相邻的两个随机小批量在原始序列上的位置不一定相毗邻。

```
my_seq = list(range(30))
for X, Y in data_iter_random(my_seq, batch_size=2, num_steps=6):
    print('X: ', X, '\nY:', Y, '\n')
```

输出：

```
X:  tensor([[18., 19., 20., 21., 22., 23.],
        [12., 13., 14., 15., 16., 17.]])
Y: tensor([[19., 20., 21., 22., 23., 24.],
        [13., 14., 15., 16., 17., 18.]])

X:  tensor([[ 0.,  1.,  2.,  3.,  4.,  5.],
        [ 6.,  7.,  8.,  9., 10., 11.]])
Y: tensor([[ 1.,  2.,  3.,  4.,  5.,  6.],
        [ 7.,  8.,  9., 10., 11., 12.]])
```

相邻采样

- 除对原始序列做随机采样之外，我们还可以令相邻的两个随机小批量在原始序列上的位置相毗邻。这时候，我们就可以用一个小批量最终时间步的隐藏状态来初始化下一个小批量的隐藏状态，从而使下一个小批量的输出也取决于当前小批量的输入，并如此循环下去。这对实现循环神经网络造成了两方面影响：一方面，在训练模型时，我们只需在每一个迭代周期开始时初始化隐藏状态；另一方面，当多个相邻小批量通过传递隐藏状态串联起来时，模型参数的梯度计算将依赖所有串联起来的小批量序列。同一迭代周期中，随着迭代次数的增加，梯度的计算开销会越来越大。 为了使模型参数的梯度计算只依赖一次迭代读取的小批量序列，我们可以在每次读取小批量前将隐藏状态从计算图中分离出来。我们将在下一节（循环神经网络的从零开始实现）的实现中了解这种处理方式。

```
def data_iter_consecutive ( corpus_indices, batch_size, num_steps, device=None):
    if device is None:
        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    corpus_indices = torch.tensor(corpus_indices, dtype=torch.float32, device=device)
    data_len = len(corpus_indices)
    batch_len = data_len // batch_size
    indices = corpus_indices[0: batch_size*batch_len].view(batch_size, batch_len)
    epoch_size = (batch_len - 1) // num_steps
    for i in range(epoch_size):
        i = i * num_steps
        X = indices[:, i: i + num_steps]
        Y = indices[:, i + 1: i + num_steps + 1]
        yield X, Y
```

- 同样的设置下，打印相邻采样每次读取的小批量样本的输入X和标签Y。相邻的两个随机小批量在原始序列上的位置相毗邻。

```
for X, Y in data_iter_consecutive(my_seq, batch_size=2, num_steps=6):
    print('X: ', X, '\nY:', Y, '\n')
```

- 输出：

```
X:  tensor([[ 0.,  1.,  2.,  3.,  4.,  5.],
        [15., 16., 17., 18., 19., 20.]])
Y: tensor([[ 1.,  2.,  3.,  4.,  5.,  6.],
        [16., 17., 18., 19., 20., 21.]])

X:  tensor([[ 6.,  7.,  8.,  9., 10., 11.],
        [21., 22., 23., 24., 25., 26.]])
Y: tensor([[ 7.,  8.,  9., 10., 11., 12.],
        [22., 23., 24., 25., 26., 27.]])
```

循环神经网络的从零开始实现

- 我们将从零开始实现一个基于字符级循环神经网络的语言模型，并在周杰伦专辑歌词数据集上训练一个模型来进行歌词创作。首先，我们读取周杰伦专辑歌词数据集：

```
from torch import nn, optim
import torch.nn.functional as F

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

(corpus_indices, char_to_idx, idx_to_char, vocab_size) = d2l.load_data_jay_lyrics()
```

循环神经网络的从零开始实现

```
def load_data_jay_lyrics():
    """加载周杰伦歌词数据集"""
    with zipfile.ZipFile('../../data/jaychou_lyrics.txt.zip') as zin:
        with zin.open('jaychou_lyrics.txt') as f:
            corpus_chars = f.read().decode('utf-8')
    corpus_chars = corpus_chars.replace('\n', ' ').replace('\r', ' ')
    corpus_chars = corpus_chars[0:10000]
    idx_to_char = list(set(corpus_chars))
    char_to_idx = dict([(char, i) for i, char in enumerate(idx_to_char)])
    vocab_size = len(char_to_idx)
    corpus_indices = [char_to_idx[char] for char in corpus_chars]
    return corpus_indices, char_to_idx, idx_to_char, vocab_size
```

one-hot向量

- 为了将词表示成向量输入到神经网络，一个简单的办法是使用one-hot向量。假设词典中不同字符的数量为N（即词典大小vocab_size），每个字符已经同一个从0到N-1的连续整数值索引一一对应。如果一个字符的索引是整数i, 那么我们创建一个全0的长为N的向量，并将其位置为i的元素设成1。该向量就是对原字符的one-hot向量。

```
def one_hot(x, n_class, dtype=torch.float32):
    # X shape: (batch), output shape: (batch, n_class)
    x = x.long()
    res = torch.zeros(x.shape[0], n_class, dtype=dtype, device=x.device)
    res.scatter_(1, x.view(-1, 1), 1)
    return res
```

one-hot向量

- 我们每次采样的小批量的形状是(批量大小, 时间步数)。下面的函数将这样的小批量变换成数个可以输入进网络的形状为(批量大小, 词典大小)的矩阵，矩阵个数等于时间步数。也就是说，时间步 t 的输入为$\boldsymbol{X}_t \in \mathrm{R}^{n \times d}$，其中 n 为批量大小，d 为输入个数，即one-hot向量长度（词典大小）。

```
def to_onehot(X, n_class):
    # X shape: (batch, seq_len), output: seq_len elements of (batch, n_class)
    return [one_hot(X[:, i], n_class) for i in range(X.shape[1])]
```

初始化模型参数

- 接下来，我们初始化模型参数。隐藏单元个数 num_hiddens是一个超参数。

```
num_inputs, num_hiddens, num_outputs = vocab_size, 256, vocab_size
def get_params():
    def _one(shape):
        ts = torch.tensor(np.random.normal(0, 0.01, size=shape), device=device, dtype=torch.float32)
        return torch.nn.Parameter(ts, requires_grad=True)
    W_xh = _one((num_inputs, num_hiddens)) # 隐藏层参数
    W_hh = _one((num_hiddens, num_hiddens))
    b_h = torch.nn.Parameter(torch.zeros(num_hiddens, device=device, requires_grad=True))
    W_hq = _one((num_hiddens, num_outputs)) # 输出层参数
    b_q = torch.nn.Parameter(torch.zeros(num_outputs, device=device, requires_grad=True))
    return nn.ParameterList([W_xh, W_hh, b_h, W_hq, b_q])
```

定义模型

- 我们根据循环神经网络的计算表达式实现该模型。首先定义init_rnn_state函数来返回初始化的隐藏状态。它返回由一个形状为(批量大小, 隐藏单元个数)的值为0的NDArray组成的元组。使用元组是为了更便于处理隐藏状态含有多个NDArray的情况。

```
def init_rnn_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device), )
```

定义模型

- 下面的rnn函数定义了在一个时间步里如何计算隐藏状态和输出。这里的激活函数使用了tanh函数。当元素在实数域上均匀分布时，tanh函数值的均值为0。

```
def rnn(inputs, state, params):
    # inputs和outputs皆为num_steps个形状为(batch_size, vocab_size)的矩阵
    W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        H = torch.tanh(torch.matmul(X, W_xh) + torch.matmul(H, W_hh) + b_h)
        Y = torch.matmul(H, W_hq) + b_q
        outputs.append(Y)
    return outputs, (H,)
```

定义预测函数

- 定义函数基于前缀prefix（数个字符的字符串）预测接下来的num_chars个字符。

```
def predict_rnn(prefix, num_chars, rnn, params, init_rnn_state,
                num_hiddens, vocab_size, device, idx_to_char, char_to_idx):
    state = init_rnn_state(1, num_hiddens, device)
    output = [char_to_idx[prefix[0]]]
    for t in range(num_chars + len(prefix) - 1):
        X = to_onehot(torch.tensor([[output[-1]]], device=device), vocab_size)
        (Y, state) = rnn(X, state, params)
        if t < len(prefix) - 1:
            output.append(char_to_idx[prefix[t + 1]])
        else:
            output.append(int(Y[0].argmax(dim=1).item()))
    return ''.join([idx_to_char[i] for i in output])
```

裁剪梯度

- 循环神经网络中较容易出现梯度衰减或梯度爆炸。为了应对梯度爆炸，我们可以裁剪梯度（clip gradient）。假设我们把所有模型参数梯度的元素拼接成一个向量 $\boldsymbol{g}$，并设裁剪的阈值是θ。裁剪后的梯度的L_2范数不超过θ。

$$\min\left(\frac{\theta}{\|\boldsymbol{g}\|}, 1\right)\boldsymbol{g}$$

```
def grad_clipping(params, theta, device):
    norm = torch.tensor([0.0], device=device)
    for param in params:
        norm += (param.grad.data ** 2).sum()
    norm = norm.sqrt().item()
    if norm > theta:
        for param in params:
            param.grad.data *= (theta / norm)
```

困惑度

- 我们通常使用困惑度（perplexity）来评价语言模型的好坏。回忆一下（softmax回归）中交叉熵损失函数的定义。困惑度是对交叉熵损失函数做指数运算后得到的值。特别地，

✓最佳情况下，模型总是把标签类别的概率预测为1，此时困惑度为1；
✓最坏情况下，模型总是把标签类别的概率预测为0，此时困惑度为正无穷；
✓基线情况下，模型总是预测所有类别的概率都相同，此时困惑度为类别个数。

- 显然，任何一个有效模型的困惑度必须小于类别个数。在本例中，困惑度必须小于词典大小vocab_size。

定义模型训练函数

跟之前的模型训练函数相比，这里的模型训练函数有以下几点不同：

- 使用困惑度评价模型。
- 在迭代模型参数前裁剪梯度。
- 对时序数据采用不同采样方法将导致隐藏状态初始化的不同。

```
def train_and_predict_rnn(rnn, get_params, init_rnn_state, num_hiddens,
                          vocab_size, device, corpus_indices, idx_to_char,
                          char_to_idx, is_random_iter, num_epochs, num_steps,
                          lr, clipping_theta, batch_size, pred_period,
                          pred_len, prefixes):
    if is_random_iter:
        data_iter_fn = d2l.data_iter_random
    else:
        data_iter_fn = d2l.data_iter_consecutive
    params = get_params()
    loss = nn.CrossEntropyLoss()
```

```
for epoch in range(num_epochs):
    if not is_random_iter:  # 如使用相邻采样，在epoch开始时初始化隐藏状态
        state = init_rnn_state(batch_size, num_hiddens, device)
    l_sum, n, start = 0.0, 0, time.time()
    data_iter = data_iter_fn(corpus_indices, batch_size, num_steps, device)
    for X, Y in data_iter:
        if is_random_iter:  # 如使用随机采样，在每个小批量更新前初始化隐藏状态
            state = init_rnn_state(batch_size, num_hiddens, device)
        else:
        # 否则需要使用detach函数从计算图分离隐藏状态, 这是为了
        # 使模型参数的梯度计算只依赖一次迭代读取的小批量序列(防止梯度计算开销太大)
            for s in state:
                s.detach_()
```

```
inputs = to_onehot(X, vocab_size)
# outputs有num_steps个形状为(batch_size, vocab_size)的矩阵
(outputs, state) = rnn(inputs, state, params)
outputs = torch.cat(outputs, dim=0) # 拼接之后形状为(num_steps * batch_size, vocab_size)
# Y的形状是(batch_size, num_steps)，转置后变成长度为batch * num_steps 的向量，跟输出的行对应
y = torch.transpose(Y, 0, 1).contiguous().view(-1)
l = loss(outputs, y.long()) # 使用交叉熵损失计算平均分类误差

# 梯度清0
if params[0].grad is not None:
    for param in params:
        param.grad.data.zero_()
l.backward()
grad_clipping(params, clipping_theta, device)  # 裁剪梯度
d2l.sgd(params, lr, 1)  # 因为误差已经取过均值，梯度不用再做平均
l_sum += l.item() * y.shape[0]
n += y.shape[0]
```

```
if (epoch + 1) % pred_period == 0:
    print('epoch %d, perplexity %f, time %.2f sec' % (
        epoch + 1, math.exp(l_sum / n), time.time() - start))
    for prefix in prefixes:
        print(' -', predict_rnn(prefix, pred_len, rnn, params, init_rnn_state,
            num_hiddens, vocab_size, device, idx_to_char, char_to_idx))
```

训练模型并创作歌词

- 现在我们可以训练模型了。首先，设置模型超参数。我们将根据前缀“分开”和“不分开”分别创作长度为50个字符（不考虑前缀长度）的一段歌词。我们每过50个迭代周期便根据当前训练的模型创作一段歌词。

num_epochs, num_steps, batch_size, lr, clipping_theta = 250, 35, 32, 1e2, 1e-2

pred_period, pred_len, prefixes = 50, 50, ['分开', '不分开']

下面采用随机采样训练模型并创作歌词。

```
train_and_predict_rnn(rnn, get_params, init_rnn_state, num_hiddens,
                      vocab_size, device, corpus_indices, idx_to_char,
                      char_to_idx, True, num_epochs, num_steps, lr,
                      clipping_theta, batch_size, pred_period, pred_len,
                      prefixes)
```

输出：

epoch 50, perplexity 70.039647, time 0.11 sec
 - 分开 我不要再想 我不能 想你的让我 我的可 你怎么 一颗四 一颗四 我不要 一颗两 一颗四 一颗四 我
 - 不分开 我不要再 你你的外 在人 别你的让我 狂的可 语人两 我不要 一颗两 一颗四 一颗四 我不要 一
epoch 100, perplexity 9.726828, time 0.12 sec
 - 分开 一直的美栈人 一起看 我不要好生活 你知不觉 我已好好生活 我知道好生活 后知不觉 我跟了这生活
 - 不分开堡 我不要再想 我不 我不 我不要再想你 不知不觉 你已经离开我 不知不觉 我跟了好生活 我知道好生
epoch 150, perplexity 2.864874, time 0.11 sec
 - 分开 一只会停留 有不它元羞 这蜴什么奇怪的事都有 包括像猫的狗 印地安老斑鸠 平常话不多 除非是乌鸦抢
 - 不分开扫 我不你再想 我不能再想 我不 我不 我不要再想你 不知不觉 你已经离开我 不知不觉 我跟了这节奏
epoch 200, perplexity 1.597790, time 0.11 sec
 - 分开 有杰伦 干 载颗拳满的让空美空主 相爱还有个人 再狠狠忘记 你爱过我的证 有晶莹的手滴 让说些人
 - 不分开扫 我叫你爸 你打我妈 这样对吗干嘛这样 何必让它牵鼻子走 瞎 说底牵打我妈要 难道球耳 快使用双截
epoch 250, perplexity 1.303903, time 0.12 sec
 - 分开 有杰人开留 仙唱它怕羞 蜥蜴横著走 这里什么奇怪的事都有 包括像猫的狗 印地安老斑鸠 平常话不多
 - 不分开简 我不能再想 我不 我不 我不能 爱情走的太快就像龙卷风 不能承受我已无处可躲 我不要再想 我不能

循环神经网络的简洁实现

- 使用PyTorch来更简洁地实现基于循环神经网络的语言模型。首先，我们读取周杰伦专辑歌词数据集。

```
import torch
from torch import nn, optim
import torch.nn.functional as F

import d2lzh_pytorch as d2l
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
(corpus_indices, char_to_idx, idx_to_char, vocab_size) = d2l.load_data_jay_lyrics()
```

定义模型

- PyTorch中的nn模块提供了循环神经网络的实现。下面构造一个含单隐藏层、隐藏单元个数为256的循环神经网络层rnn_layer。

```
num_hiddens = 256
# rnn_layer = nn.LSTM(input_size=vocab_size, hidden_size=num_hiddens)
rnn_layer = nn.RNN(input_size=vocab_size, hidden_size=num_hiddens)
```

定义模型

- 这里rnn_layer的输入形状为(时间步数, 批量大小, 输入个数)。其中输入个数即one-hot向量长度（词典大小）。此外，rnn_layer作为nn.RNN实例，在前向计算后会分别返回输出和隐藏状态h，其中输出指的是隐藏层在**各个时间步**上计算并输出的隐藏状态，它们通常作为后续输出层的输入。需要强调的是，该“输出”本身并不涉及输出层计算，形状为(时间步数, 批量大小, 隐藏单元个数)。而nn.RNN实例在前向计算返回的隐藏状态指的是隐藏层在**最后时间步**的隐藏状态：当隐藏层有多层时，每一层的隐藏状态都会记录在该变量中；对于像长短期记忆（LSTM），隐藏状态是一个元组(h, c)，即hidden state和cell state。

定义模型

- 来看看我们的例子，输出形状为(时间步数, 批量大小, 输入个数)，隐藏状态h的形状为(层数, 批量大小, 隐藏单元个数)。

```
num_steps = 35
batch_size = 2
state = None
X = torch.rand(num_steps, batch_size, vocab_size)
Y, state_new = rnn_layer(X, state)
print(Y.shape, len(state_new), state_new[0].shape)
```

- 输出：

```
torch.Size([35, 2, 256]) 1 torch.Size([2, 256])
```

- 接下来我们继承Module类来定义一个完整的循环神经网络。它首先将输入数据使用one-hot向量表示后输入到rnn_layer中，然后使用全连接输出层得到输出。输出个数等于词典大小vocab_size。

```
class RNNModel(nn.Module):
    def __init__(self, rnn_layer, vocab_size):
        super(RNNModel, self).__init__()
        self.rnn = rnn_layer
        self.hidden_size = rnn_layer.hidden_size * (2 if rnn_layer.bidirectional else 1)
        self.vocab_size = vocab_size
        self.dense = nn.Linear(self.hidden_size, vocab_size)
        self.state = None
    def forward(self, inputs, state): # inputs: (batch, seq_len)
        X = d2l.to_onehot(inputs, self.vocab_size) # 获取one-hot向量表示 X是个list
        Y, self.state = self.rnn(torch.stack(X), state)
        output = self.dense(Y.view(-1, Y.shape[-1])) #输出形状为(num_steps * batch_size, vocab_size)
        return output, self.state
```

下面定义一个预测函数。这里的实现区别在于前向计算和初始化隐藏状态的函数接口。

```
def predict_rnn_pytorch(prefix, num_chars, model, vocab_size, device, idx_to_char, char_to_idx):
    state = None
    output = [char_to_idx[prefix[0]]] # output会记录prefix加上输出
    for t in range(num_chars + len(prefix) - 1):
        X = torch.tensor([output[-1]], device=device).view(1, 1)
        if state is not None:
            if isinstance(state, tuple): # LSTM, state:(h, c)
                state = (state[0].to(device), state[1].to(device))
            else:
                state = state.to(device)
        (Y, state) = model(X, state)
        if t < len(prefix) - 1:
            output.append(char_to_idx[prefix[t + 1]])
        else:
            output.append(int(Y.argmax(dim=1).item()))
    return ''.join([idx_to_char[i] for i in output])
```

定义模型

- 我们使用权重为随机值的模型来预测一次。

```
model = RNNModel(rnn_layer, vocab_size).to(device)
predict_rnn_pytorch('分开', 10, model, vocab_size, device, idx_to_char,
  char_to_idx)
```

- 输出：

'分开戏想暖迎凉想征凉征征'

- 接下来实现训练函数。算法同上一节的一样，但这里只使用了相邻采样来读取数据。

```
def train_and_predict_rnn_pytorch(model, num_hiddens, vocab_size, device, corpus_indices, idx_to_char,
                                  char_to_idx, num_epochs, num_steps, lr, clipping_theta, batch_size, pred_period, pred_len, prefixes):
    loss = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    model.to(device)
    state = None
    for epoch in range(num_epochs):
        l_sum, n, start = 0.0, 0, time.time()
        data_iter = d2l.data_iter_consecutive(corpus_indices, batch_size, num_steps, device) # 相邻采样
        for X, Y in data_iter:
            if state is not None:
                # 使用detach函数是为了使模型参数的梯度计算只依赖一次迭代读取的小批量序列
                if isinstance (state, tuple): # LSTM, state:(h, c)
                    state = (state[0].detach(), state[1].detach())
                else:
                    state = state.detach()
```

```
(output, state) = model(X, state) # output: 形状为(num_steps * batch_size, vocab_size)

# Y的形状是(batch_size, num_steps)，转置后再变成长度为
# batch * num_steps 的向量，这样跟输出的行一一对应
y = torch.transpose(Y, 0, 1).contiguous().view(-1)
l = loss(output, y.long())

optimizer.zero_grad()
l.backward()
# 梯度裁剪
d2l.grad_clipping(model.parameters(), clipping_theta, device)
optimizer.step()
l_sum += l.item() * y.shape[0]
n += y.shape[0]
```

```
num_epochs, batch_size, lr, clipping_theta = 250, 32, 1e-3, 1e-2 # 注意这里的学习率设置
pred_period, pred_len, prefixes = 50, 50, ['分开', '不分开']
train_and_predict_rnn_pytorch(model, num_hiddens, vocab_size, device, corpus_indices, idx_to_char, char_to_idx,
                              num_epochs, num_steps, lr, clipping_theta, batch_size, pred_period, pred_len, prefixes)
```

输出：

```
epoch 50, perplexity 10.658418, time 0.05 sec
 - 分开始我妈  想要你 我不多 让我心到的 我妈妈 我不能再想 我不多再想 我不要再想 我不多再想 我不要
 - 不分开 我想要你不你 我 你不要 让我心到的 我妈人 可爱女人 坏坏的让我疯狂的可爱女人 坏坏的让我疯狂的
.....
epoch 200, perplexity 1.034663, time 0.05 sec
 - 分开不能去吗周杰伦 才离 没要你在一场悲剧 我的完美主义 太彻底 分手的话像语言暴力 我已无能为力再提起
 - 不分开 让我面到你 爱情来的太快就像龙卷风 离不开暴风圈来不及逃 我不能再想 我不能再想 我不 我不 我不
epoch 250, perplexity 1.021437, time 0.05 sec
 - 分开 我我外的家边 你知道这 我爱不看的太  我想一个又重来不以 迷已文一只剩下回忆 让我叫带你 你你的
 - 不分开 我我想想和 是你听没不  我不能不想  不知不觉 你已经离开我 不知不觉 我跟了这节奏 后知后觉
```

通过时间反向传播

- 在前面模型中，如果不裁剪梯度，模型将无法正常训练。为了深刻理解这一现象，这里将介绍循环神经网络中梯度的计算和存储方法，即通过时间反向传播（back-propagation through time）。
- 之前我们介绍了神经网络中梯度计算与存储的一般思路，并强调正向传播和反向传播相互依赖。正向传播在循环神经网络中比较直观，而通过时间反向传播其实是反向传播在循环神经网络中的具体应用。我们需要将循环神经网络按时间步展开，从而得到模型变量和参数之间的依赖关系，并依据链式法则应用反向传播计算并存储梯度。

简单起见，我们考虑一个无偏差项的循环神经网络，且激活函数为恒等映射（$\phi(x) = x$）。设时间步 t 的输入为单样本 $\boldsymbol{x}_t \in \mathbb{R}^d$，标签为 y_t，那么隐藏状态 $\boldsymbol{h}_t \in \mathbb{R}^h$ 的计算表达式为

$$\boldsymbol{h}_t = \boldsymbol{W}_{hx}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1},$$

其中 $\boldsymbol{W}_{hx} \in \mathbb{R}^{h\times d}$ 和 $\boldsymbol{W}_{hh} \in \mathbb{R}^{h\times h}$ 是隐藏层权重参数。设输出层权重参数 $\boldsymbol{W}_{qh} \in \mathbb{R}^{q\times h}$，时间步 t 的输出层变量 $\boldsymbol{o}_t \in \mathbb{R}^q$ 计算为

$$\boldsymbol{o}_t = \boldsymbol{W}_{qh}\boldsymbol{h}_t.$$

设时间步 t 的损失为 $\ell(\boldsymbol{o}_t, y_t)$。时间步数为 T 的损失函数 L 定义为

$$L = \frac{1}{T}\sum_{t=1}^{T} \ell(\boldsymbol{o}_t, y_t).$$

我们将 L 称为有关给定时间步的数据样本的目标函数，并在本节后续讨论中简称为目标函数。

模型计算图

为了可视化循环神经网络中模型变量和参数在计算中的依赖关系，我们可以绘制模型计算图，如图6.3所示。例如，时间步3的隐藏状态$\boldsymbol{h}_3$的计算依赖模型参数$\boldsymbol{W}_{hx}$、$\boldsymbol{W}_{hh}$、上一时间步隐藏状态$\boldsymbol{h}_2$以及当前时间步输入$\boldsymbol{x}_3$。

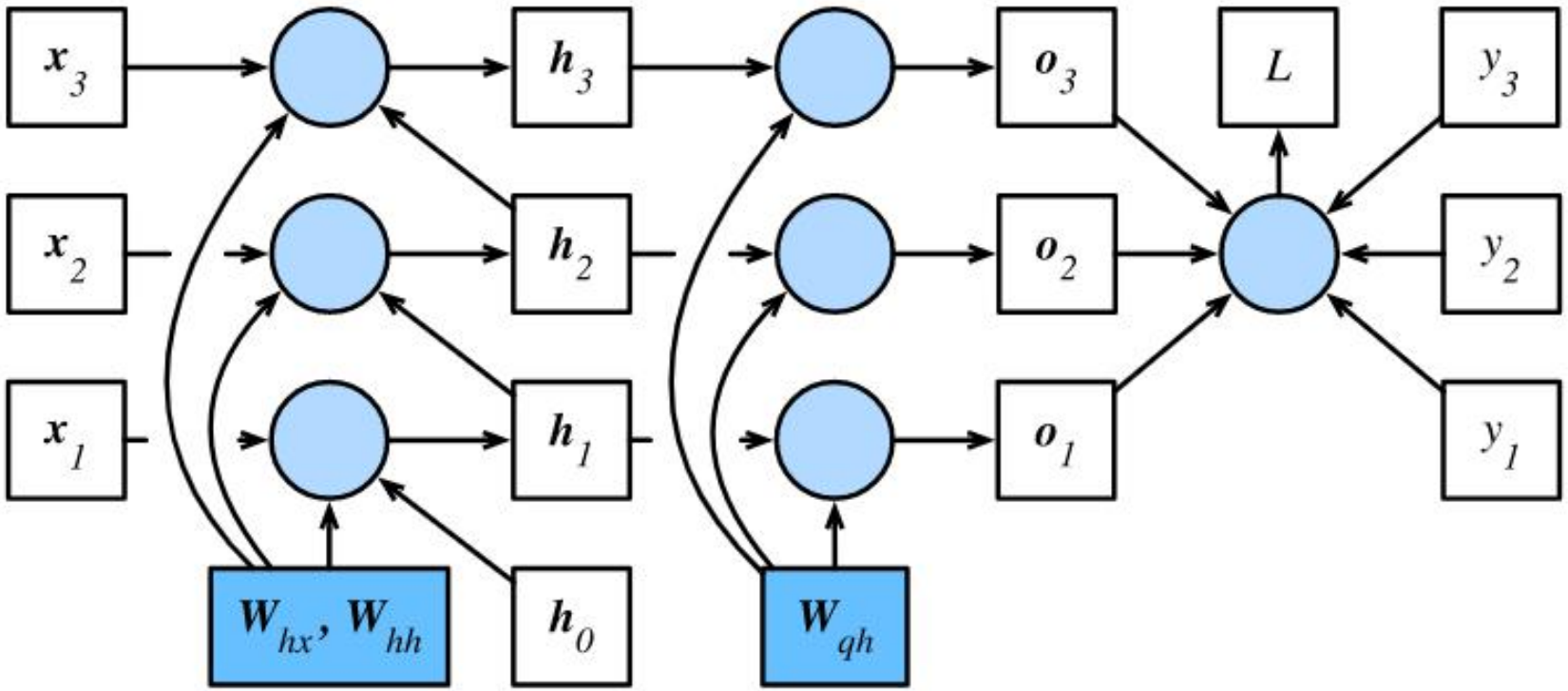


图6.3 时间步数为3的循环神经网络模型计算中的依赖关系。方框代表变量（无阴影）或参数（有阴影），圆圈代表运算符

方法

首先，目标函数有关各时间步输出层变量的梯度$\partial L/\partial \boldsymbol{o}_t \in \mathbb{R}^q$很容易计算：

$$\frac{\partial L}{\partial \boldsymbol{o}_t} = \frac{\partial \ell(\boldsymbol{o}_t, y_t)}{T \cdot \partial \boldsymbol{o}_t}.$$

下面，我们可以计算目标函数有关模型参数$\boldsymbol{W}_{qh}$的梯度$\partial L/\partial \boldsymbol{W}_{qh} \in \mathbb{R}^{q \times h}$。根据图6.3，$L$通过$\boldsymbol{o}_1, \ldots, \boldsymbol{o}_T$依赖$\boldsymbol{W}_{qh}$。依据链式法则，

$$\frac{\partial L}{\partial \boldsymbol{W}_{qh}} = \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{o}_t}, \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{W}_{qh}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{o}_t} \boldsymbol{h}_t^\top.$$

其次，我们注意到隐藏状态之间也存在依赖关系。 在图6.3中，L只通过$\boldsymbol{o}_T$依赖最终时间步T的隐藏状态$\boldsymbol{h}_T$。因此，我们先计算目标函数有关最终时间步隐藏状态的梯度$\partial L/\partial \boldsymbol{h}_T \in \mathbb{R}^h$。依据链式法则，我们得到

$$\frac{\partial L}{\partial \boldsymbol{h}_T} = \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{o}_T}, \frac{\partial \boldsymbol{o}_T}{\partial \boldsymbol{h}_T}\right) = \boldsymbol{W}_{qh}^\top \frac{\partial L}{\partial \boldsymbol{o}_T}.$$

接下来对于时间步$t < T$, 在图6.3中，L通过$\boldsymbol{h}_{t+1}$和$\boldsymbol{o}_t$依赖$\boldsymbol{h}_t$。依据链式法则， 目标函数有关时间步$t < T$的隐藏状态的梯度$\partial L/\partial \boldsymbol{h}_t \in \mathbb{R}^h$需要按照时间步从大到小依次计算：

$$\frac{\partial L}{\partial \boldsymbol{h}_t} = \text{prod}(\frac{\partial L}{\partial \boldsymbol{h}_{t+1}}, \frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t}) + \text{prod}(\frac{\partial L}{\partial \boldsymbol{o}_t}, \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{h}_t}) = \boldsymbol{W}_{hh}^\top \frac{\partial L}{\partial \boldsymbol{h}_{t+1}} + \boldsymbol{W}_{qh}^\top \frac{\partial L}{\partial \boldsymbol{o}_t}$$

将上面的递归公式展开，对任意时间步$1 \leq t \leq T$，我们可以得到目标函数有关隐藏状态梯度的通项公式

$$\frac{\partial L}{\partial \boldsymbol{h}_t} = \sum_{i=t}^{T} \left(\boldsymbol{W}_{hh}^\top \right)^{T-i} \boldsymbol{W}_{qh}^\top \frac{\partial L}{\partial \boldsymbol{o}_{T+t-i}}.$$

由上式中的指数项可见，当时间步数 T 较大或者时间步 t 较小时，目标函数有关隐藏状态的梯度较容易出现衰减和爆炸。这也会影响其他包含$\partial L/\partial \boldsymbol{h}_t$项的梯度，例如隐藏层中模型参数的梯度$\partial L/\partial \boldsymbol{W}_{hx} \in \mathbb{R}^{h \times d}$和$\partial L/\partial \boldsymbol{W}_{hh} \in \mathbb{R}^{h \times h}$。 在图6.3中，$L$通过$\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T$依赖这些模型参数。 依据链式法则，我们有

$$\frac{\partial L}{\partial \boldsymbol{W}_{hx}} = \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{h}_t}, \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}_{hx}} \right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{h}_t} \boldsymbol{x}_t^\top,$$

$$\frac{\partial L}{\partial \boldsymbol{W}_{hh}} = \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{h}_t}, \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}_{hh}} \right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{h}_t} \boldsymbol{h}_{t-1}^\top.$$

门控循环单元（GRU）

- 我们发现，当时间步数较大或者时间步较小时，循环神经网络的梯度较容易出现衰减或爆炸。虽然裁剪梯度可以应对梯度爆炸，但无法解决梯度衰减的问题。通常由于这个原因，循环神经网络在实际中较难捕捉时间序列中时间步距离较大的依赖关系。
- 门控循环神经网络（gated recurrent neural network）的提出，正是为了更好地捕捉时间序列中时间步距离较大的依赖关系。它通过可以学习的门来控制信息的流动。其中，门控循环单元（gated recurrent unit，GRU）是一种常用的门控循环神经网络。

门控循环单元

- 下面将介绍门控循环单元的设计。它引入了重置门（reset gate）和更新门（update gate）的概念，从而修改了循环神经网络中隐藏状态的计算方式。

门控循环单元

如图6.4所示，门控循环单元中的重置门和更新门的输入均为当前时间步输入$\boldsymbol{X}_t$与上一时间步隐藏状态$\boldsymbol{H}_{t-1}$，输出由激活函数为sigmoid函数的全连接层计算得到。

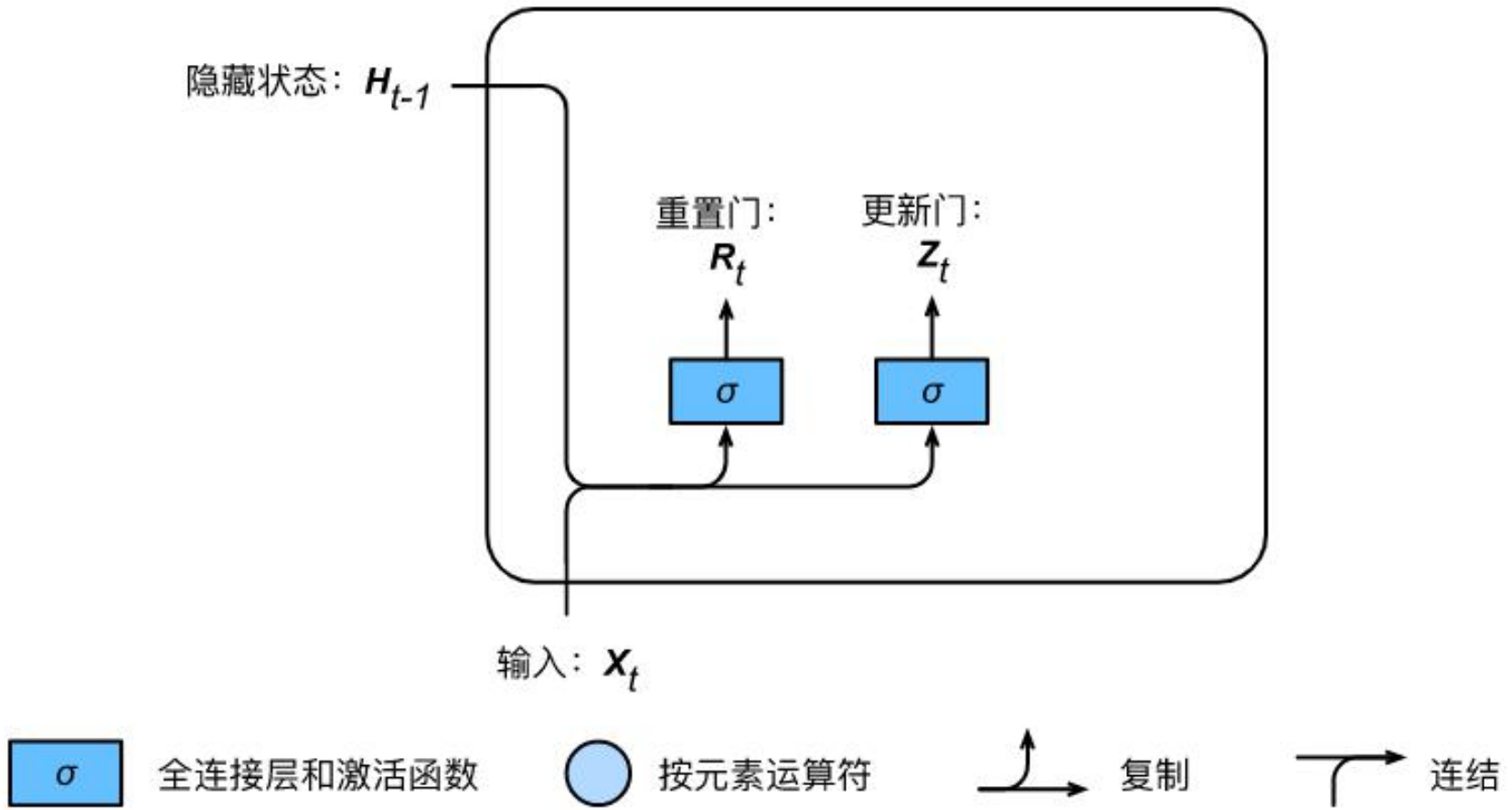


图6.4 门控循环单元中重置门和更新门的计算

门控循环单元

具体来说，假设隐藏单元个数为h，给定时间步t的小批量输入$\boldsymbol{X}_t \in \mathbb{R}^{n\times d}$（样本数为$n$，输入个数为$d$）和上一时间步隐藏状态$\boldsymbol{H}_{t-1} \in \mathbb{R}^{n\times h}$。重置门$\boldsymbol{R}_t \in \mathbb{R}^{n\times h}$和更新门$\boldsymbol{Z}_t \in \mathbb{R}^{n\times h}$的计算如下：

$$\boldsymbol{R}_t = \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xr} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hr} + \boldsymbol{b}_r),$$
$$\boldsymbol{Z}_t = \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xz} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hz} + \boldsymbol{b}_z),$$

其中$\boldsymbol{W}_{xr}, \boldsymbol{W}_{xz} \in \mathbb{R}^{d\times h}$和$\boldsymbol{W}_{hr}, \boldsymbol{W}_{hz} \in \mathbb{R}^{h\times h}$是权重参数，$\boldsymbol{b}_r, \boldsymbol{b}_z \in \mathbb{R}^{1\times h}$是偏差参数。3.8节（多层感知机）节中介绍过，sigmoid函数可以将元素的值变换到0和1之间。因此，重置门$\boldsymbol{R}_t$和更新门$\boldsymbol{Z}_t$中每个元素的值域都是$[0, 1]$

接下来，门控循环单元将计算候选隐藏状态来辅助稍后的隐藏状态计算。如图6.5所示，我们将当前时间步重置门的输出与上一时间步隐藏状态做按元素乘法（符号为⊙）。如果重置门中元素值接近0，那么意味着重置对应隐藏状态元素为0，即丢弃上一时间步的隐藏状态。如果元素值接近1，那么表示保留上一时间步的隐藏状态。然后，将按元素乘法的结果与当前时间步的输入连结，再通过含激活函数tanh的全连接层计算出候选隐藏状态，其所有元素的值域为$[-1, 1]$。

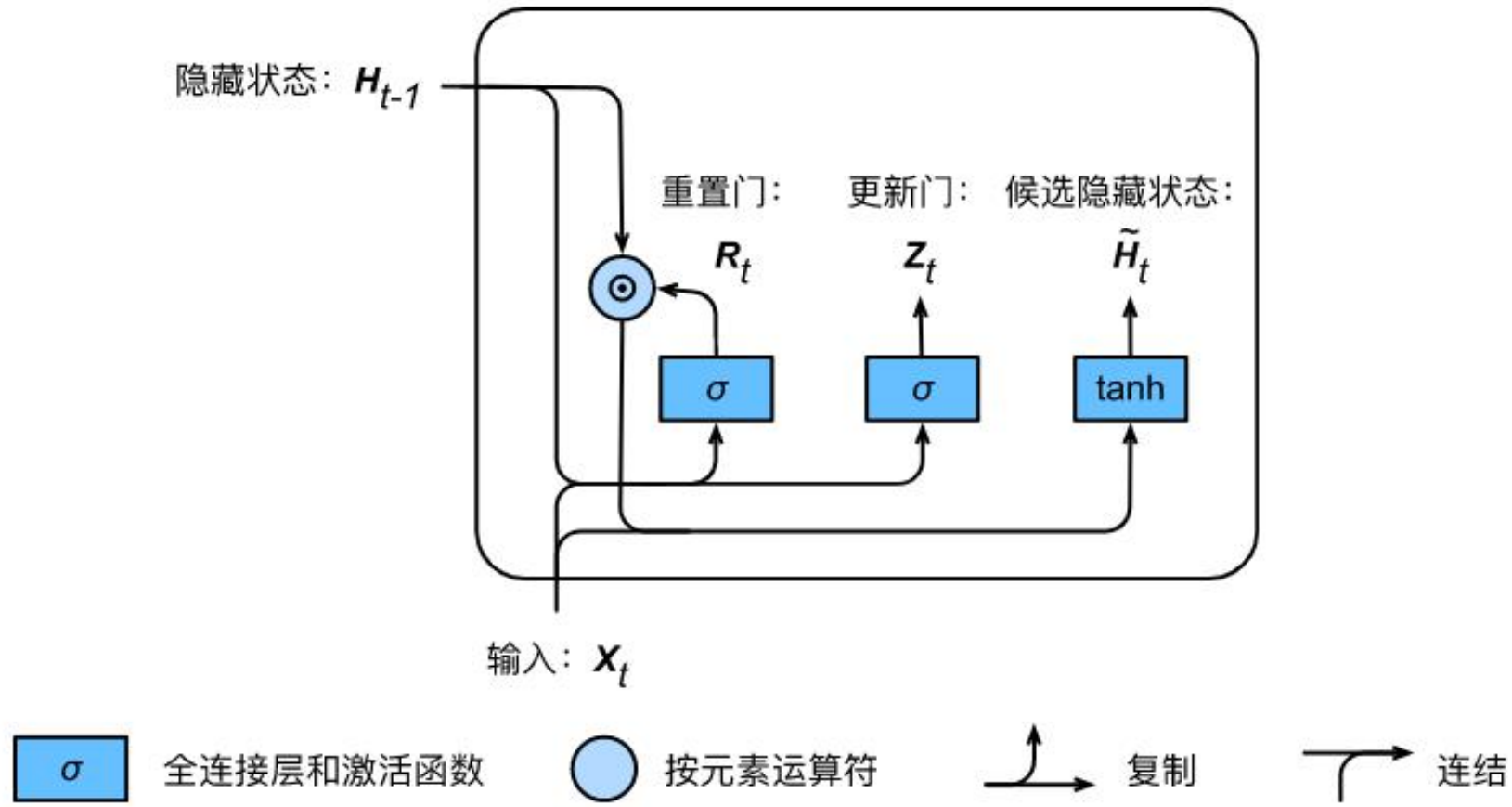


图6.5 门控循环单元中候选隐藏状态的计算

候选隐藏状态

具体来说，时间步t的候选隐藏状态$\tilde{\boldsymbol{H}}_t \in \mathbb{R}^{n \times h}$的计算为

$$\tilde{\boldsymbol{H}}_t = \tanh(\boldsymbol{X}_t \boldsymbol{W}_{xh} + (\boldsymbol{R}_t \odot \boldsymbol{H}_{t-1}) \boldsymbol{W}_{hh} + \boldsymbol{b}_h),$$

其中$\boldsymbol{W}_{xh} \in \mathbb{R}^{d \times h}$和$\boldsymbol{W}_{hh} \in \mathbb{R}^{h \times h}$是权重参数，$\boldsymbol{b}_h \in \mathbb{R}^{1 \times h}$是偏差参数。从上面这个公式可以看出，重置门控制了上一时间步的隐藏状态如何流入当前时间步的候选隐藏状态。而上一时间步的隐藏状态可能包含了时间序列截至上一时间步的全部历史信息。因此，重置门可以用来丢弃与预测无关的历史信息。

最后，时间步t的隐藏状态$\boldsymbol{H}_t \in \mathbb{R}^{n \times h}$的计算使用当前时间步的更新门$\boldsymbol{Z}_t$来对上一时间步的隐藏状态$\boldsymbol{H}_{t-1}$和当前时间步的候选隐藏状态$\tilde{\boldsymbol{H}}_t$做组合：

$$\boldsymbol{H}_t = \boldsymbol{Z}_t \odot \boldsymbol{H}_{t-1} + (1 - \boldsymbol{Z}_t) \odot \tilde{\boldsymbol{H}}_t.$$

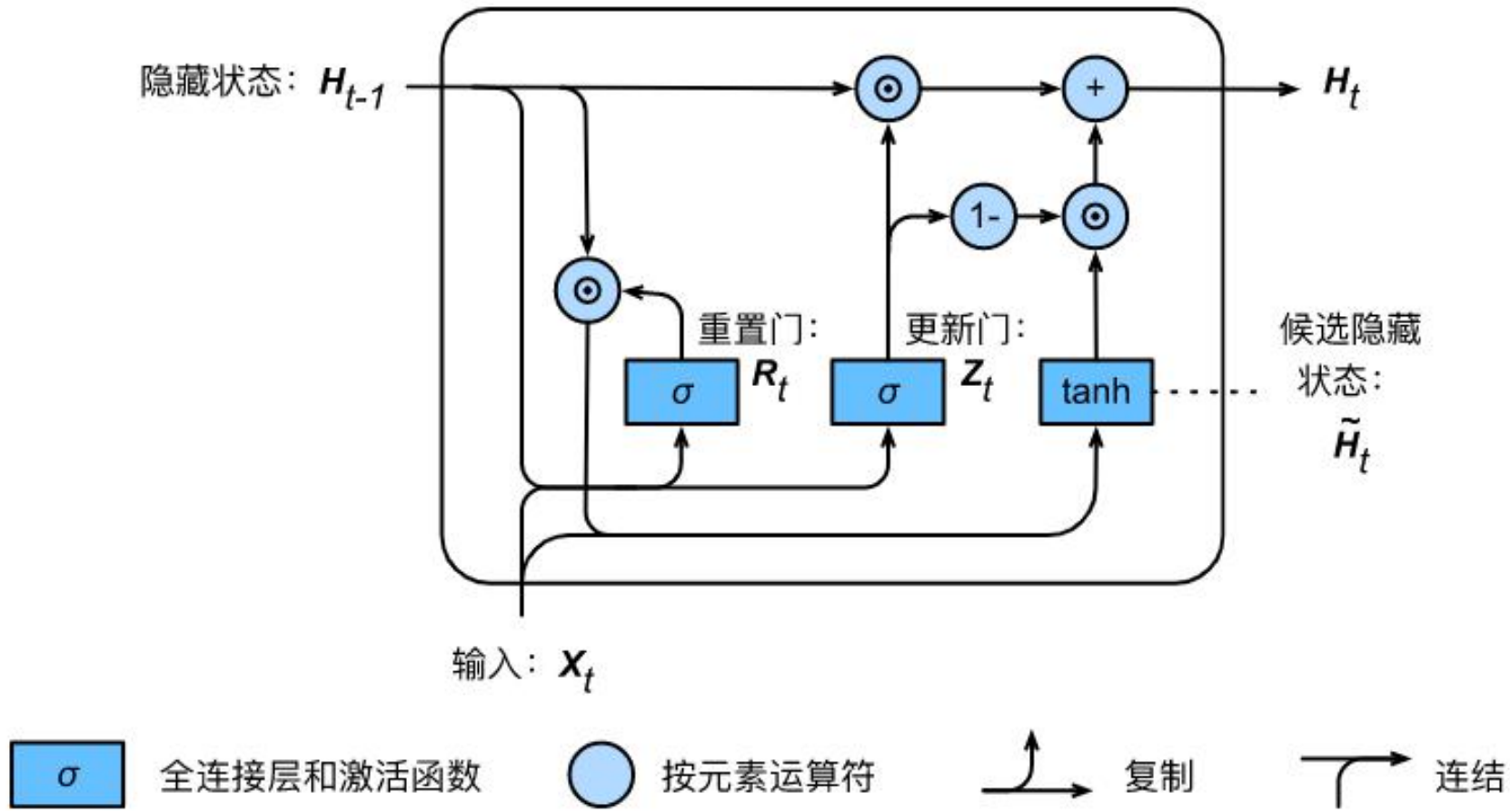


图6.6 门控循环单元中隐藏状态的计算

隐藏状态

- 值得注意的是，更新门可以控制隐藏状态应该如何被包含当前时间步信息的候选隐藏状态所更新，假设更新门在时间步 t′ 到 t（t′<t）之间一直近似1。那么，在时间步 t′ 到 t 之间的输入信息几乎没有流入时间步 t 的隐藏状态 $\boldsymbol{H}_t$。实际上，这可以看作是较早时刻的隐藏状态 $\boldsymbol{H}_{t'-1}$ 一直通过时间保存并传递至当前时间步 t 。这个设计可以应对循环神经网络中的梯度衰减问题，并更好地捕捉时间序列中时间步距离较大的依赖关系。

我们对门控循环单元的设计稍作总结：

- 重置门有助于捕捉时间序列里短期的依赖关系；
- 更新门有助于捕捉时间序列里长期的依赖关系。

简洁实现

在PyTorch中我们直接调用nn模块中的GRU类即可。

```
lr = 1e-2 # 注意调整学习率
gru_layer = nn.GRU(input_size=vocab_size, hidden_size=num_hiddens)
model = d2l.RNNModel(gru_layer, vocab_size).to(device)
d2l.train_and_predict_rnn_pytorch(model, num_hiddens, vocab_size, device, corpus_indices, idx_to_char,
    char_to_idx, num_epochs, num_steps, lr, clipping_theta, batch_size, pred_period, pred_len, prefixes)
```

输出：

```
epoch 40, perplexity 1.022157, time 1.02 sec
 - 分开手牵手 一步两步三步四步望著天 看星星 一颗两颗三颗四颗 连成线背著背默默许下心愿 看远方的星是否听
 - 不分开暴风圈来不及逃 我不能再想 我不能再想 我不 我不 我不能 爱情走的太快就像龙卷风 不能承受我已无处
epoch 80, perplexity 1.014535, time 1.04 sec
 - 分开始想像 爸和妈当年的模样 说著一口吴侬软语的姑娘缓缓走过外滩 消失的 旧时光 一九四三 在回忆 的路
 - 不分开始爱像  不知不觉 你已经离开我 不知不觉 我跟了这节奏 后知后觉 又过了一个秋 后知后觉 我该好好
```

长短期记忆（LSTM）

- 介绍另一种常用的门控循环神经网络：长短期记忆（long short-term memory，LSTM）。它比门控循环单元的结构稍微复杂一点。LSTM 中引入了3个门，即输入门（input gate）、遗忘门（forget gate）和输出门（output gate），以及与隐藏状态形状相同的记忆细胞（某些文献把记忆细胞当成一种特殊的隐藏状态），从而记录额外的信息。

输入门、遗忘门和输出门

与门控循环单元中的重置门和更新门一样，如图6.7所示，长短期记忆的门的输入均为当前时间步输入$\boldsymbol{X}_t$与上一时间步隐藏状态$\boldsymbol{H}_{t-1}$，输出由激活函数为sigmoid函数的全连接层计算得到。如此一来，这3个门元素的值域均为$[0,1]$。

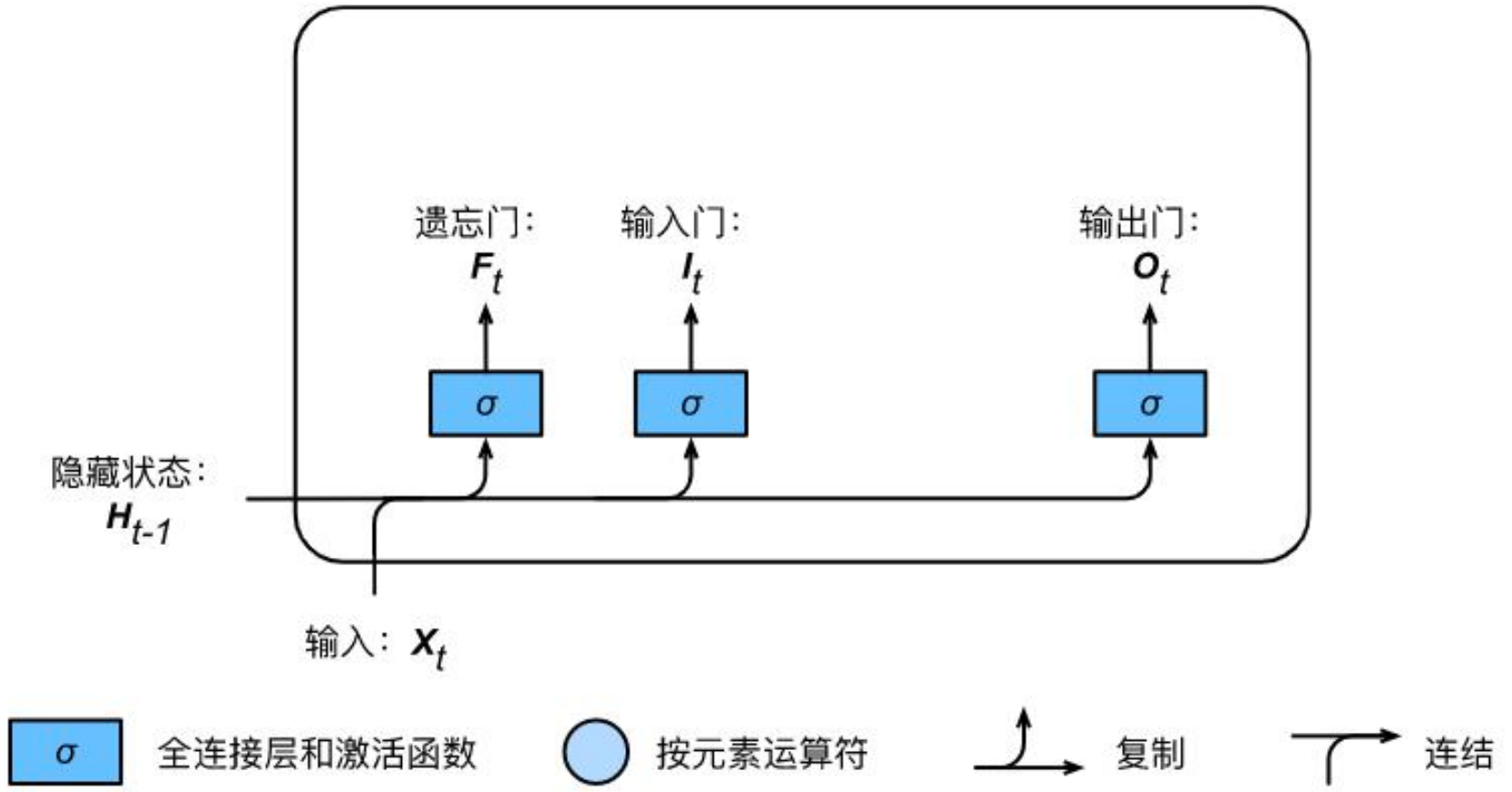


图6.7 长短期记忆中输入门、遗忘门和输出门的计算

输入门、遗忘门和输出门

具体来说，假设隐藏单元个数为h，给定时间步t的小批量输入$\boldsymbol{X}_t \in \mathbb{R}^{n\times d}$（样本数为$n$，输入个数为$d$）和上一时间步隐藏状态$\boldsymbol{H}_{t-1} \in \mathbb{R}^{n\times h}$。时间步$t$的输入门$\boldsymbol{I}_t \in \mathbb{R}^{n\times h}$、遗忘门$\boldsymbol{F}_t \in \mathbb{R}^{n\times h}$和输出门$\boldsymbol{O}_t \in \mathbb{R}^{n\times h}$分别计算如下：

$$
\begin{aligned}
\boldsymbol{I}_t &= \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xi} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hi} + \boldsymbol{b}_i),\\
\boldsymbol{F}_t &= \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xf} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hf} + \boldsymbol{b}_f),\\
\boldsymbol{O}_t &= \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xo} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{ho} + \boldsymbol{b}_o),
\end{aligned}
$$

其中的$\boldsymbol{W}_{xi}, \boldsymbol{W}_{xf}, \boldsymbol{W}_{xo} \in \mathbb{R}^{d\times h}$和$\boldsymbol{W}_{hi}, \boldsymbol{W}_{hf}, \boldsymbol{W}_{ho} \in \mathbb{R}^{h\times h}$是权重参数，$\boldsymbol{b}_i, \boldsymbol{b}_f, \boldsymbol{b}_o \in \mathbb{R}^{1\times h}$是偏差参数。

候选记忆细胞

接下来，长短期记忆需要计算候选记忆细胞$\tilde{C}_t$。它的计算与上面介绍的3个门类似，但使用了值域在$[-1,1]$的tanh函数作为激活函数，如图6.8所示。

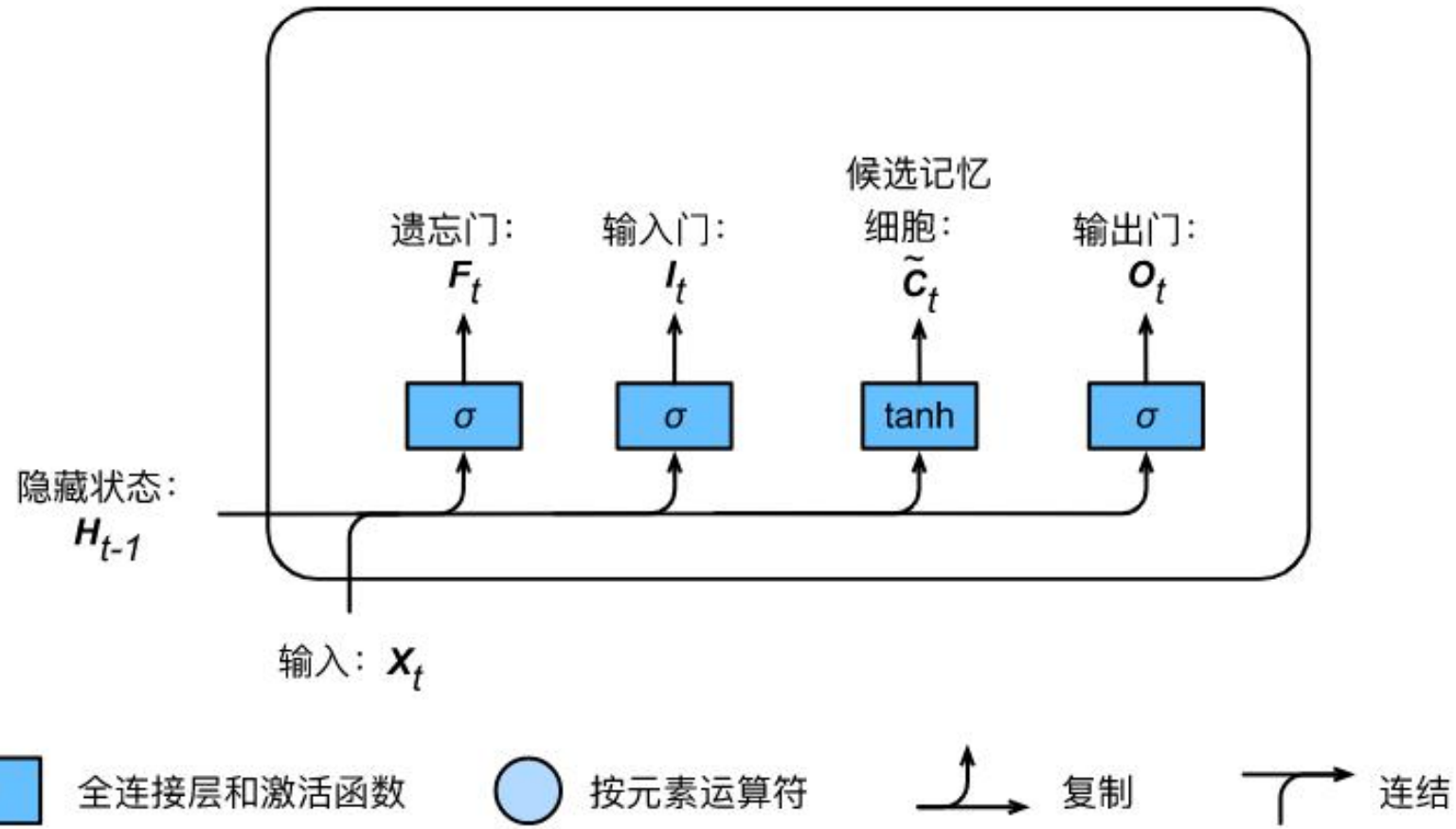


图6.8 长短期记忆中候选记忆细胞的计算

候选记忆细胞

具体来说，时间步t的候选记忆细胞$\tilde{\boldsymbol{C}}_t \in \mathbb{R}^{n \times h}$的计算为

$$\tilde{\boldsymbol{C}}_t = \tanh(\boldsymbol{X}_t \boldsymbol{W}_{xc} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hc} + \boldsymbol{b}_c),$$

其中$\boldsymbol{W}_{xc} \in \mathbb{R}^{d \times h}$和$\boldsymbol{W}_{hc} \in \mathbb{R}^{h \times h}$是权重参数，$\boldsymbol{b}_c \in \mathbb{R}^{1 \times h}$是偏差参数。

记忆细胞

我们可以通过元素值域在$[0,1]$的输入门、遗忘门和输出门来控制隐藏状态中信息的流动，这一般也是通过使用按元素乘法（符号为$\odot$）来实现的。当前时间步记忆细胞$\boldsymbol{C}_t \in \mathbb{R}^{n \times h}$的计算组合了上一时间步记忆细胞和当前时间步候选记忆细胞的信息，并通过遗忘门和输入门来控制信息的流动：

$$\boldsymbol{C}_t = \boldsymbol{F}_t \odot \boldsymbol{C}_{t-1} + \boldsymbol{I}_t \odot \tilde{\boldsymbol{C}}_t.$$

如图6.9所示，遗忘门控制上一时间步的记忆细胞$\boldsymbol{C}_{t-1}$中的信息是否传递到当前时间步，而输入门则控制当前时间步的输入$\boldsymbol{X}_t$通过候选记忆细胞$\tilde{\boldsymbol{C}}_t$如何流入当前时间步的记忆细胞。如果遗忘门一直近似1且输入门一直近似0，过去的记忆细胞将一直通过时间保存并传递至当前时间步。这个设计可以应对循环神经网络中的梯度衰减问题，并更好地捕捉时间序列中时间步距离较大的依赖关系。

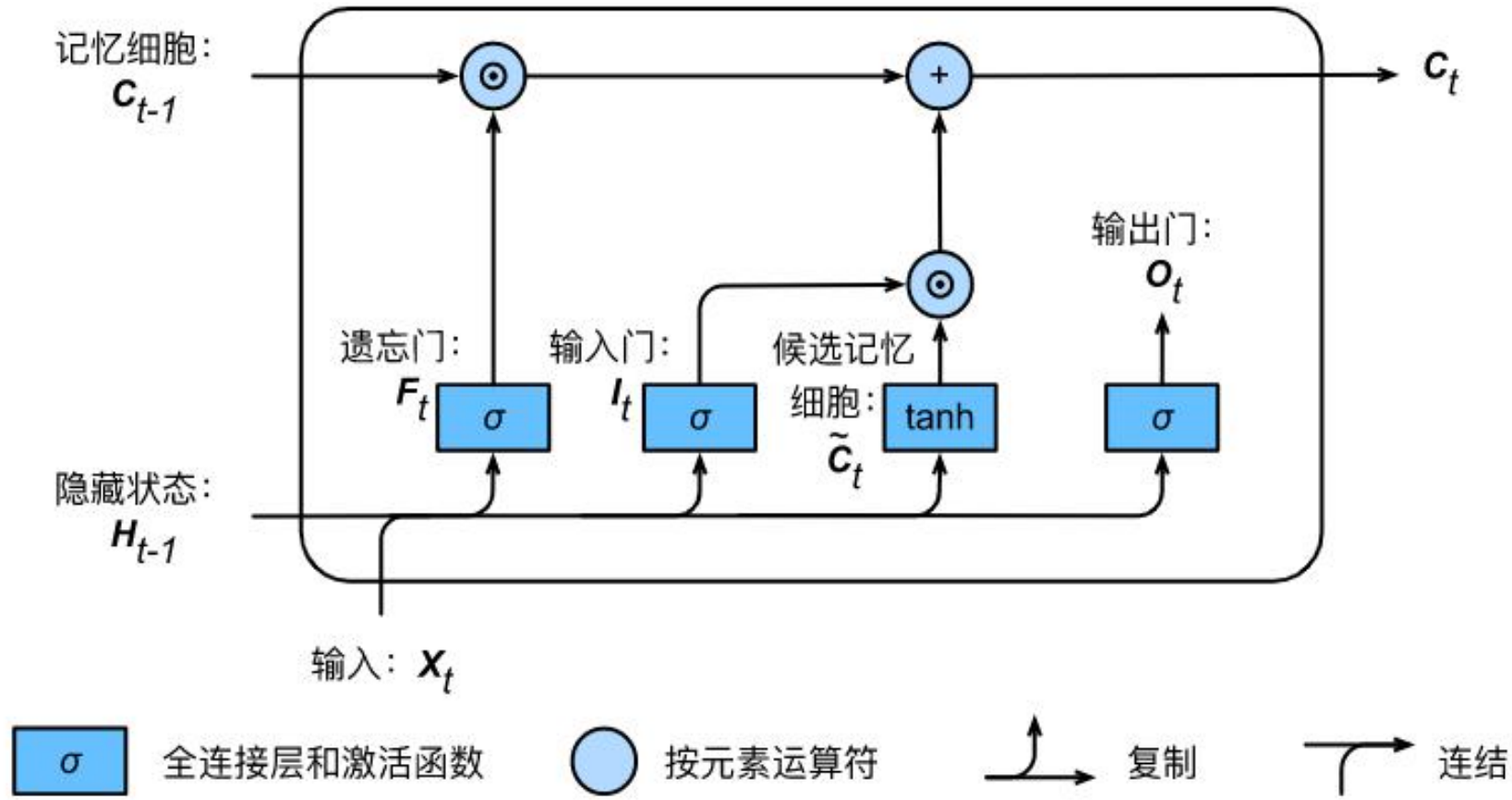
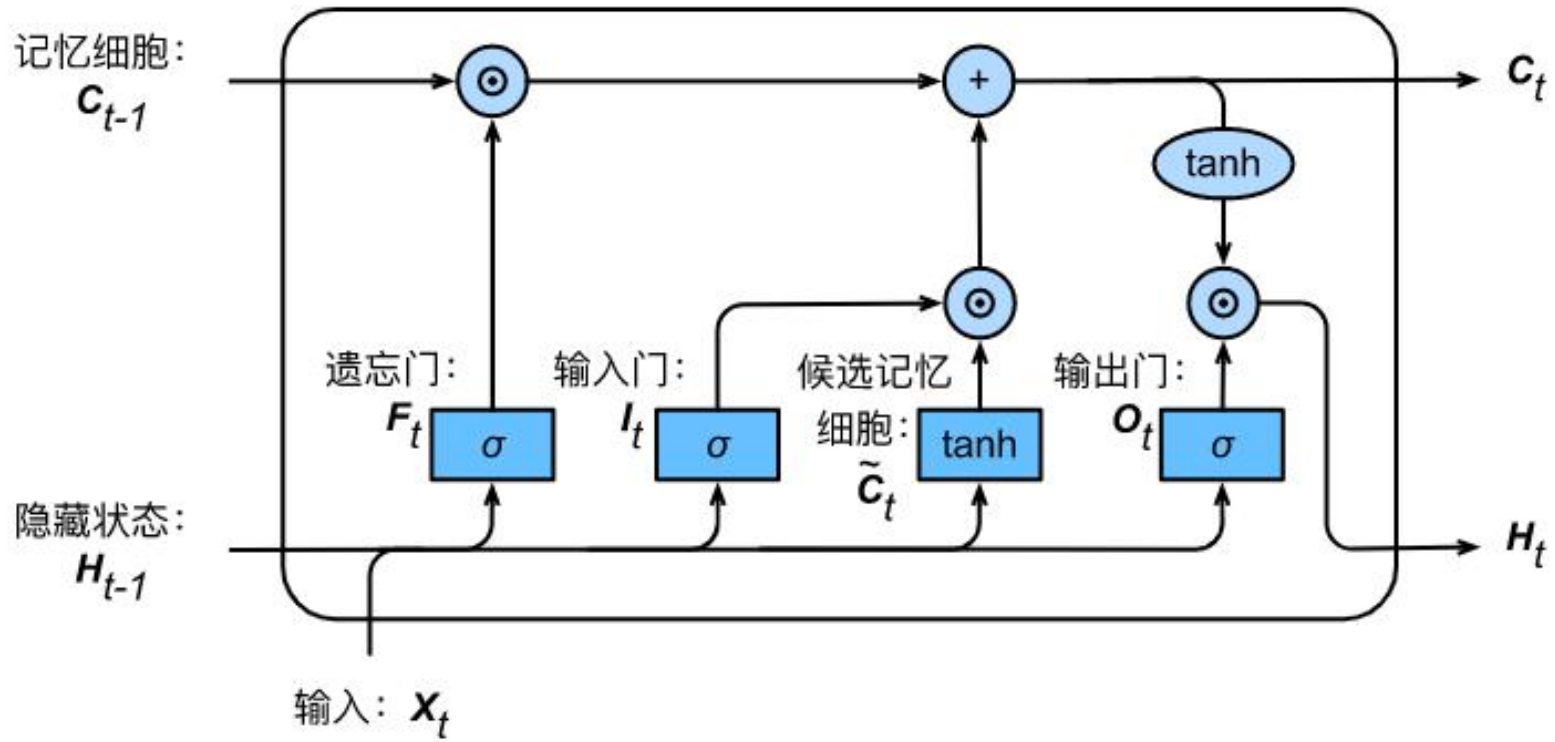


图6.9 长短期记忆中记忆细胞的计算

有了记忆细胞以后，接下来我们还可以通过输出门来控制从记忆细胞到隐藏状态$\boldsymbol{H}_t \in \mathbb{R}^{n \times h}$的信息的流动：

$$\boldsymbol{H}_t = \boldsymbol{O}_t \odot \tanh(\boldsymbol{C}_t).$$

这里的tanh函数确保隐藏状态元素值在-1到1之间。需要注意的是，当输出门近似1时，记忆细胞信息将传递到隐藏状态供输出层使用；当输出门近似0时，记忆细胞信息只自己保留。图6.10展示了长短期记忆中隐藏状态的计算。



简洁实现

在Pytorch中我们可以直接调用rnn模块中的LSTM类。

```
lr = 1e-2 # 注意调整学习率
lstm_layer = nn.LSTM(input_size=vocab_size, hidden_size=num_hiddens)
model = d2l.RNNModel(lstm_layer, vocab_size)
d2l.train_and_predict_rnn_pytorch(model, num_hiddens, vocab_size, device, corpus_indices, idx_to_char, char_to_idx,
                                  num_epochs, num_steps, lr, clipping_theta, batch_size, pred_period, pred_len, prefixes)
```

输出：

```
epoch 40, perplexity 1.020401, time 1.54 sec
 - 分开始想担 妈跟我 一定是我妈在 因为分手前那句抱歉 在感动 穿梭时间的画面的钟 从反方向开始移动 回到
 - 不分开始想像 妈跟我 我将我的寂寞封闭 然后在这里 不限日期 然后将过去 慢慢温习 让我爱上你 那场悲剧
epoch 80, perplexity 1.011164, time 1.34 sec
 - 分开始想担 你的 从前的可爱女人 温柔的让我心疼的可爱女人 透明的让我感动的可爱女人 坏坏的让我疯狂的可
 - 不分开 我满了 让我疯狂的可爱女人 漂亮的让我面红的可爱女人 温柔的让我心疼的可爱女人 透明的让我感动的可
```

深度循环神经网络

- 本章到目前为止介绍的循环神经网络只有一个单向的隐藏层，在深度学习应用里，我们通常会用到含有多个隐藏层的循环神经网络，也称作深度循环神经网络。图6.11演示了一个有L个隐藏层的深度循环神经网络，每个隐藏状态不断传递至当前层的下一时间步和当前时间步的下一层。

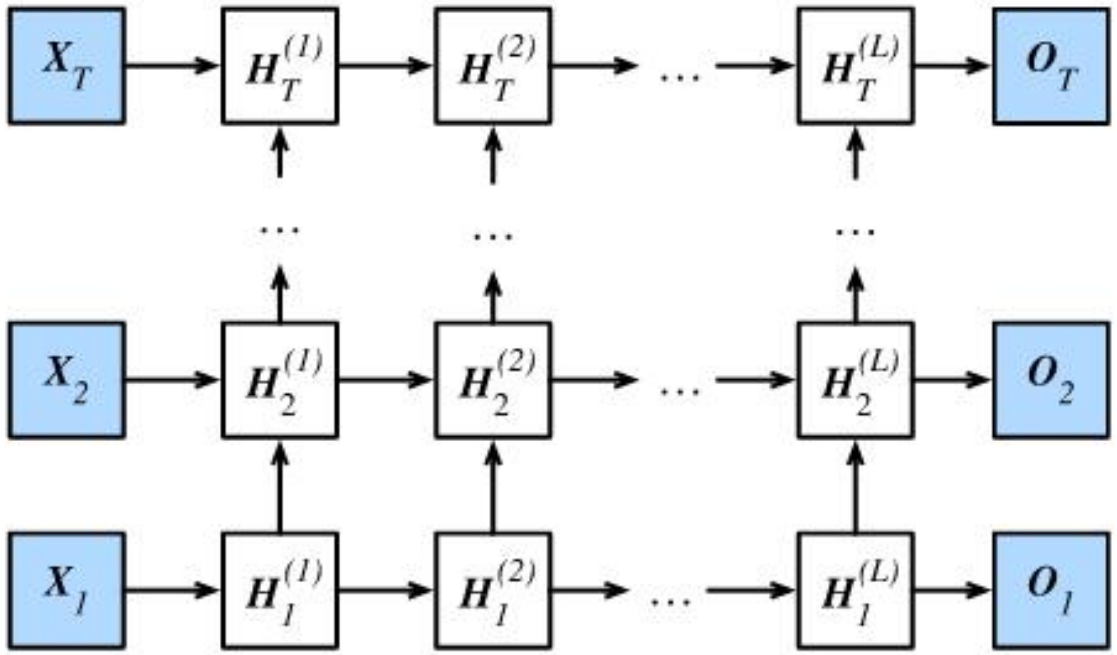


图6.11 深度循环神经网络的架构

深度循环神经网络

具体来说，在时间步t里，设小批量输入$\boldsymbol{X}_t \in \mathbb{R}^{n\times d}$（样本数为$n$，输入个数为$d$），第$\ell$隐藏层（$\ell = 1, \ldots, L$）的隐藏状态为$\boldsymbol{H}_t^{(\ell)} \in \mathbb{R}^{n\times h}$（隐藏单元个数为$h$），输出层变量为$\boldsymbol{O}_t \in \mathbb{R}^{n\times q}$（输出个数为$q$），且隐藏层的激活函数为$\phi$。第1隐藏层的隐藏状态和之前的计算一样：

$$\boldsymbol{H}_t^{(1)} = \phi(\boldsymbol{X}_t \boldsymbol{W}_{xh}^{(1)} + \boldsymbol{H}_{t-1}^{(1)} \boldsymbol{W}_{hh}^{(1)} + \boldsymbol{b}_h^{(1)}),$$

其中权重$\boldsymbol{W}_{xh}^{(1)} \in \mathbb{R}^{d\times h}$、$\boldsymbol{W}_{hh}^{(1)} \in \mathbb{R}^{h\times h}$和偏差$\boldsymbol{b}_h^{(1)} \in \mathbb{R}^{1\times h}$分别为第1隐藏层的模型参数。

深度循环神经网络

当$1 < \ell \leq L$时，第ℓ隐藏层的隐藏状态的表达式为

$$\boldsymbol{H}_t^{(\ell)} = \phi(\boldsymbol{H}_t^{(\ell-1)} \boldsymbol{W}_{xh}^{(\ell)} + \boldsymbol{H}_{t-1}^{(\ell)} \boldsymbol{W}_{hh}^{(\ell)} + \boldsymbol{b}_h^{(\ell)}),$$

其中权重$\boldsymbol{W}_{xh}^{(\ell)} \in \mathbb{R}^{h \times h}$、$\boldsymbol{W}_{hh}^{(\ell)} \in \mathbb{R}^{h \times h}$和偏差$\boldsymbol{b}_h^{(\ell)} \in \mathbb{R}^{1 \times h}$分别为第$\ell$隐藏层的模型参数。

最终，输出层的输出只需基于第L隐藏层的隐藏状态：

$$\boldsymbol{O}_t = \boldsymbol{H}_t^{(L)} \boldsymbol{W}_{hq} + \boldsymbol{b}_q,$$

其中权重$\boldsymbol{W}_{hq} \in \mathbb{R}^{h \times q}$和偏差$\boldsymbol{b}_q \in \mathbb{R}^{1 \times q}$为输出层的模型参数。

双向循环神经网络

- 之前介绍的循环神经网络模型都是假设当前时间步是由前面的较早时间步的序列决定的，因此它们都将信息通过隐藏状态从前往后传递。有时候，当前时间步也可能由后面时间步决定。例如，当我们写下一个句子时，可能会根据句子后面的词来修改句子前面的用词。双向循环神经网络通过增加从后往前传递信息的隐藏层来更灵活地处理这类信息。图6.12演示了一个含单隐藏层的双向循环神经网络的架构。

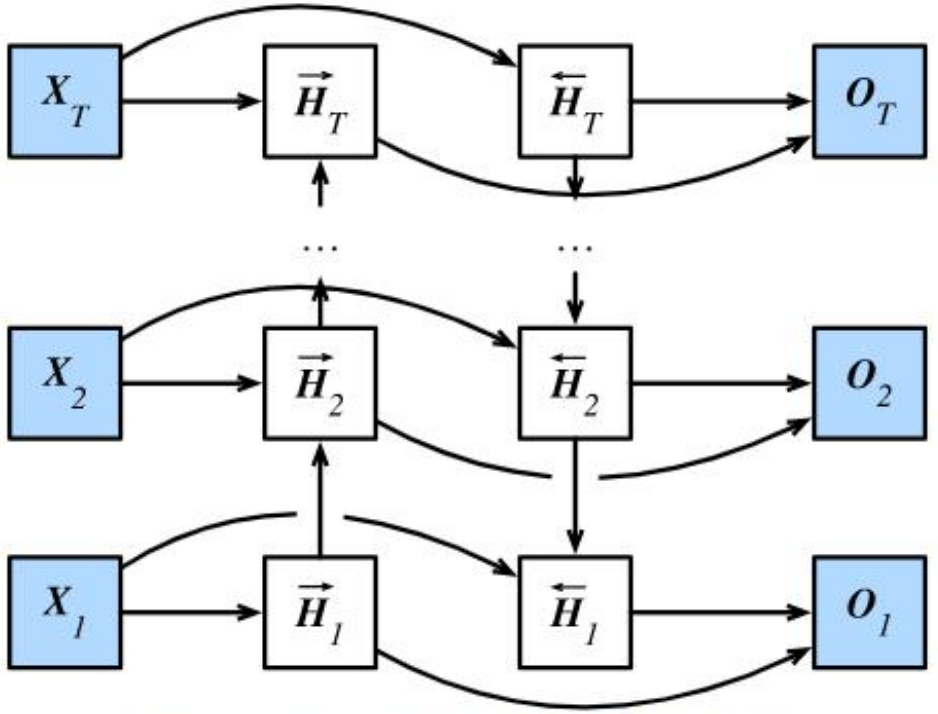


图6.12 双向循环神经网络的架构

双向循环神经网络

下面我们来介绍具体的定义。 给定时间步t的小批量输入$\boldsymbol{X}_t \in \mathbb{R}^{n\times d}$（样本数为$n$，输入个数为$d$）和隐藏层激活函数为$\phi$。在双向循环神经网络的架构中， 设该时间步正向隐藏状态为$\overrightarrow{\boldsymbol{H}}_t \in \mathbb{R}^{n\times h}$（正向隐藏单元个数为$h$）， 反向隐藏状态为$\overleftarrow{\boldsymbol{H}}_t \in \mathbb{R}^{n\times h}$（反向隐藏单元个数为$h$）。我们可以分别计算正向隐藏状态和反向隐藏状态：

$$
\begin{aligned}
\overrightarrow{\boldsymbol{H}}_t &= \phi(\boldsymbol{X}_t \boldsymbol{W}_{xh}^{(f)} + \overrightarrow{\boldsymbol{H}}_{t-1} \boldsymbol{W}_{hh}^{(f)} + \boldsymbol{b}_h^{(f)}),\\
\overleftarrow{\boldsymbol{H}}_t &= \phi(\boldsymbol{X}_t \boldsymbol{W}_{xh}^{(b)} + \overleftarrow{\boldsymbol{H}}_{t+1} \boldsymbol{W}_{hh}^{(b)} + \boldsymbol{b}_h^{(b)}),
\end{aligned}
$$

其中权重$\boldsymbol{W}_{xh}^{(f)} \in \mathbb{R}^{d\times h}$、$\boldsymbol{W}_{hh}^{(f)} \in \mathbb{R}^{h\times h}$、$\boldsymbol{W}_{xh}^{(b)} \in \mathbb{R}^{d\times h}$、$\boldsymbol{W}_{hh}^{(b)} \in \mathbb{R}^{h\times h}$和偏差$\boldsymbol{b}_h^{(f)} \in \mathbb{R}^{1\times h}$、$\boldsymbol{b}_h^{(b)} \in \mathbb{R}^{1\times h}$均为模型参数。

双向循环神经网络

然后我们连结两个方向的隐藏状态$\overrightarrow{\boldsymbol{H}}_t$和$\overleftarrow{\boldsymbol{H}}_t$来得到隐藏状态$\boldsymbol{H}_t \in \mathbb{R}^{n \times 2h}$，并将其输入到输出层。输出层计算输出$\boldsymbol{O}_t \in \mathbb{R}^{n \times q}$（输出个数为$q$）：

$$\boldsymbol{O}_t = \boldsymbol{H}_t \boldsymbol{W}_{hq} + \boldsymbol{b}_q,$$

其中权重$\boldsymbol{W}_{hq} \in \mathbb{R}^{2h \times q}$和偏差$\boldsymbol{b}_q \in \mathbb{R}^{1 \times q}$为输出层的模型参数。不同方向上的隐藏单元个数也可以不同。