**PyTorch小白也能看懂的循环神经网络实现**

上次文章我们学习了循环神经网络（RNN）的梯度消失的问题。具体的文章可以从历史的文章合集中查看。今天我们来学习一下怎么使用Pytorch来实现循环神经网络。

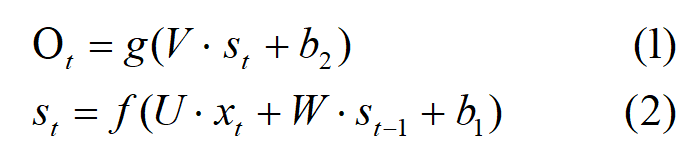
1. **RNN的结构**

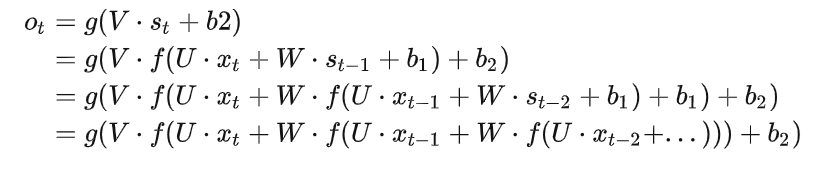
首先，我们还是先来回忆一下RNN的结果：

我们已经知道，RNN网络基于实践中展开的结果如下所示：

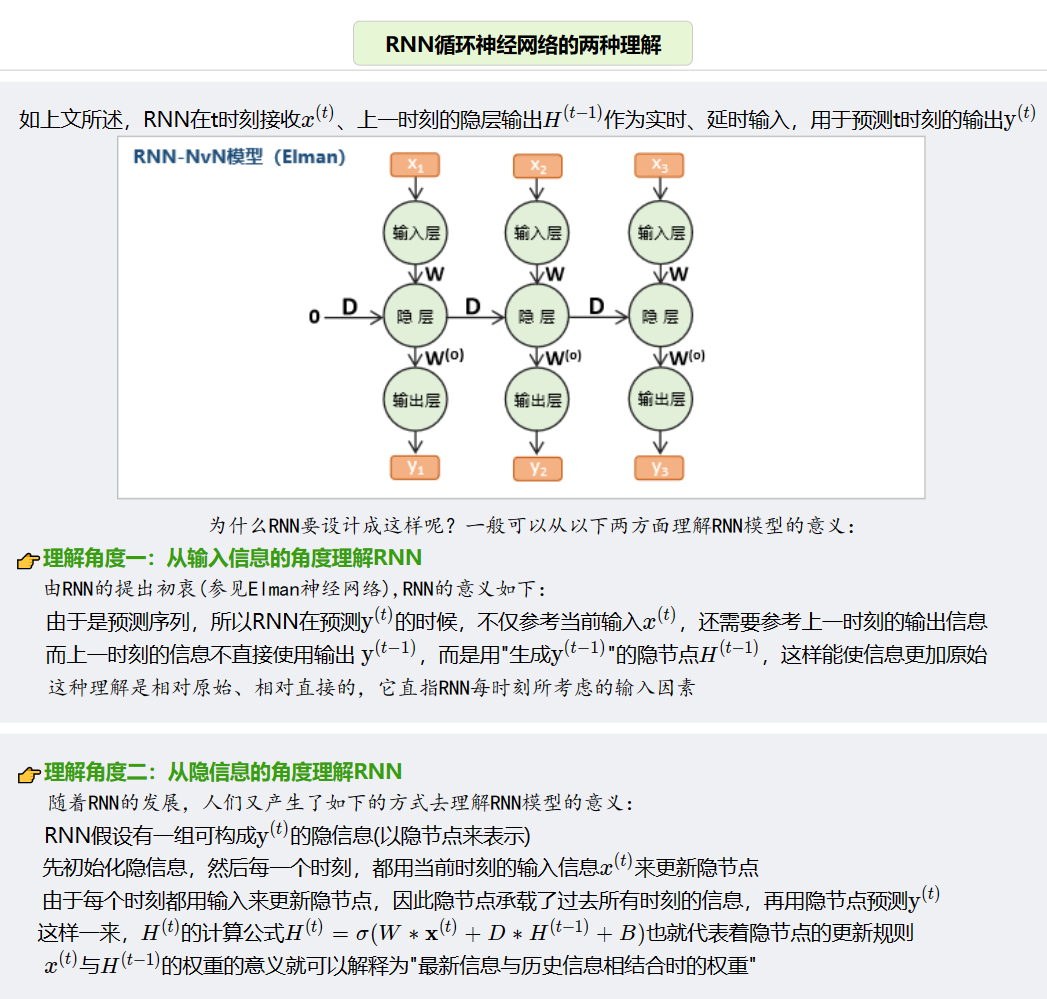


且对于t时刻，RNN的输出可以由以下公式描述：





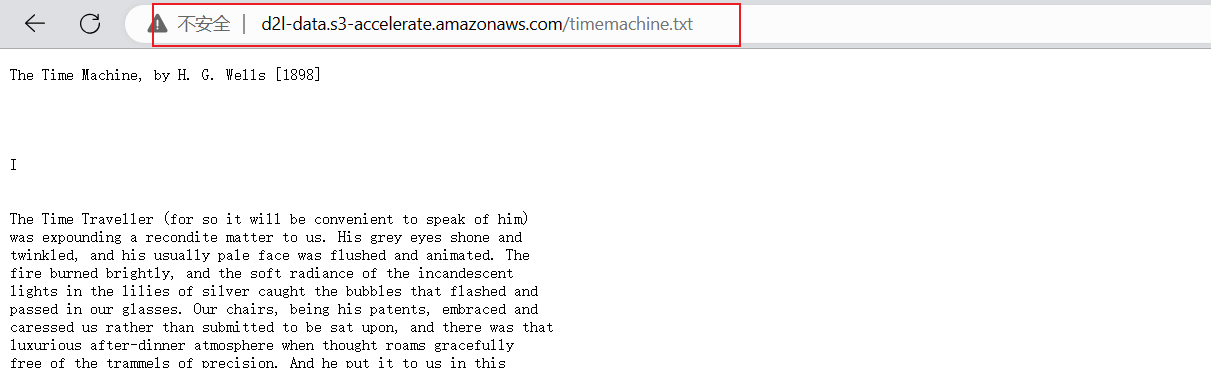
我们进一步来理解RNN的，参考饼老师的文章：



接下来我们就开始使用实际的数据来实现我们的RNN网络。

1. **加载文本数据**

首选是加载数据集，我们的数据是以下网站的文本，即《时间机器》这本书数据集，其是一个英文数据即：



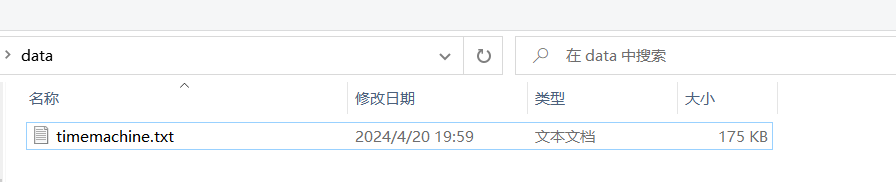
数据集的访问链接如下：

[d2l-data.s3-accelerate.amazonaws.com/timemachine.txt](http://d2l-data.s3-accelerate.amazonaws.com/timemachine.txt)

我们可以编写以下的代码完成上述《时间机器》数据的下载：

*# 数据加载脚本*  
import os  
import re  
import hashlib  
import requests  
  
DATA\_HUB = dict()  
DATA\_URL = 'http://d2l-data.s3-accelerate.amazonaws.com/'  
DATA\_HUB['time\_machine'] = (DATA\_URL + 'timemachine.txt','090b5e7e70c295757f55df93cb0a180b9691891a')  
  
def download(name, cache\_dir):  
    *# 判断*  
    assert name in DATA\_HUB, f"{name} does not exist in {DATA\_HUB}."  
    url, sha1\_hash = DATA\_HUB[name]  
    os.makedirs(cache\_dir, exist\_ok=True)  
    fname = os.path.join(cache\_dir, url.split('/')[-1])  
    if os.path.exists(fname):  
        sha1 = hashlib.sha1()  
        with open(fname, 'rb') as f:  
            while True:  
                *# 1048576表示读取1M的数据*  
                data = f.read(1048576)  
                if not data:  
                    break  
                sha1.update(data)  
        if sha1.hexdigest() == sha1\_hash:  
            return fname  *# Hit cache*  
    print(f'Downloading {fname} from {url}...')  
    r = requests.get(url, stream=True, verify=True)  
    with open(fname, 'wb') as f:  
        f.write(r.content)  
    return fname  
  
def read\_time\_machine(name,cache\_dir):  
    fname = download('time\_machine',cache\_dir=cache\_dir)  
    with open(fname, 'r') as f:  
        lines = f.readlines()  
    *# 稍微清洗并返回列表形式的数据*  
    return [re.sub('[^A-Za-z]+', ' ', line).strip().lower() for line in lines]  
  
*# 调用*  
filename = 'timemachine'  
cache\_dir = os.path.join('.', 'data')  
time\_machine= read\_time\_machine(name=filename,cache\_dir=cache\_dir)

我们就可以在相应的目录下得到需要的数据集，接下来我们需要进行文本的处理。



1. **文本数据处理**

接下来就是数据的处理了，我们需要将数据处理为相应的词汇预料，我们先写出公共的函数和类：

import collections  
def count\_corpus(tokens):  
    """Count token frequencies.  
    Defined in :numref:`sec\_text\_preprocessing`"""  
    *# Here `tokens` is a 1D list or 2D list*  
    if len(tokens) == 0 or isinstance(tokens[0], list):  
        *# Flatten a list of token lists into a list of tokens*  
        tokens = [token for line in tokens for token in line]  
    return collections.Counter(tokens)

上述函数用于计算tokens中每个词汇的出现频率。

# tokenize化

def tokenize(lines, token='word'):  
 if token == 'word':  
 return [line.split() for line in lines]  
 elif token == 'char':  
 return [list(line) for line in lines]  
 else:  
 print('ERROR: unknown token type: ' + token)

接下来，创建一个Vacab类用于处理文本数据中的词汇，它将文本中的每个唯一词汇映射到一个唯一的索引：

class Vocab:  
    """Vocabulary for text."""  
    def \_\_init\_\_(self, tokens=None, min\_freq=0, reserved\_tokens=None):  
        """Defined in :numref:`sec\_text\_preprocessing`"""  
        if tokens is None:  
            tokens = []  
        if reserved\_tokens is None:  
            reserved\_tokens = []  
        *# 计算 tokens 中每个词汇的出现频率*  
        counter = count\_corpus(tokens)  
        *# 对词汇及其频率进行排序，按照频率降序排列*  
        self.\_token\_freqs = sorted(counter.items(), key=lambda x: x[1],reverse=True)  
        *# 创建一个列表，包含一个未知词汇标记 <unk> 和所有保留词汇*  
        self.idx\_to\_token = ['<unk>'] + reserved\_tokens  
        *# 创建一个字典，将每个词汇映射到它在 idx\_to\_token 列表中的索引*  
        self.token\_to\_idx = {token: idx for idx, token in enumerate(self.idx\_to\_token)}  
        for token, freq in self.\_token\_freqs:  
            if freq < min\_freq:  
                break  
            if token not in self.token\_to\_idx:  
                self.idx\_to\_token.append(token)  
                self.token\_to\_idx[token] = len(self.idx\_to\_token) - 1  
  
    def \_\_len\_\_(self):  
        return len(self.idx\_to\_token)  
  
    def \_\_getitem\_\_(self, tokens):  
        if not isinstance(tokens, (list, tuple)):  
            return self.token\_to\_idx.get(tokens, self.unk)  
        return [self.\_\_getitem\_\_(token) for token in tokens]  
    *# 用于将索引转换为对应的词汇*  
    def to\_tokens(self, indices):  
        if not isinstance(indices, (list, tuple)):  
            return self.idx\_to\_token[indices]  
        return [self.idx\_to\_token[index] for index in indices]  
  
    @property  
    def unk(self):  *# Index for the unknown token*  
        return 0  
  
    @property  
    def token\_freqs(self):  *# Index for the unknown token*  
        return self.\_token\_freqs

对其中的一些代码进行解释：

（1）\_\_init\_\_，它接受三个参数：

tokens：一个包含词汇的列表，默认为 None。

Min\_freq：一个整数，表示词汇最小出现频率，默认为 0。

Reserved\_tokens：一个包含保留词汇的列表，默认为 None。

1. counter = count\_corpus(tokens) 来计算 tokens 中每个词汇的出现频率。
2. self.\_token\_freqs = sorted(counter.items(), key=lambda x: x[1], reverse=True) 对词汇及其频率进行排序，按照频率降序排列。
3. self.token\_to\_idx = {token: idx for idx, token in enumerate(self.idx\_to\_token)} 创建一个字典，将每个词汇映射到它在 idx\_to\_token 列表中的索引。
4. self.token\_to\_idx.get(tokens, self.unk) 如果 tokens 不是列表或元组，尝试获取它的索引，如果不存在则返回 self.unk（未知词汇的索引）
5. return [self.\_\_getitem\_\_(token) for token in tokens] 如果 tokens 是列表或元组，对每个词汇调用 \_\_getitem\_\_ 方法，并返回它们的索引列表。

使用 vocab = Vocab(tokens) 创建实例后，您可以通过 vocab对象的方法来访问词汇和索引之间的映射:

（1）len(vocab)：返回词汇表的大小，即idx\_to\_token列表的长度。

（2）vocab[token]：返回词汇 token 对应的索引。

（3）vocab[indices]：如果indices是一个索引列表，返回这些索引对应的词汇列表。

（4）vocab.to\_tokens(indices)：将索引列表indices转换为对应的词汇列表。

第三个函数加载我们的数据集，并返回相应的corpus和vocab：

在Python中，使用方括号[]来访问对象的属性或方法是一种常见的做法。对于类来说，可以通过在方括号中指定一个键（key）来调用一个方法，尤其是当这个方法被设计为类似字典的行为时。在我们的Vocab类中，\_\_getitem\_\_ 方法被重载了，\_\_getitem\_\_ 是一个特殊方法，它允许对象使用方括号来访问。当使用 vocab[token] 这样的表达式时，实际上是在调用 vocab.\_\_getitem\_\_(token)。

def load\_corpus\_time\_machine(max\_tokens=-1):  
    *# 加载预料*  
    lines = read\_time\_machine(name=filename,cache\_dir=cache\_dir)  
    tokens = tokenize(lines, 'char')  
    vocab = Vocab(tokens)  
    *# Since each text line in the time machine dataset is not necessarily a*  
    *# sentence or a paragraph, flatten all the text lines into a single list*  
    corpus = [vocab[token] for line in tokens for token in line]  
    if max\_tokens > 0:  
        corpus = corpus[:max\_tokens]  
    return corpus, vocab

*# 调用*

load\_corpus\_time\_machine()

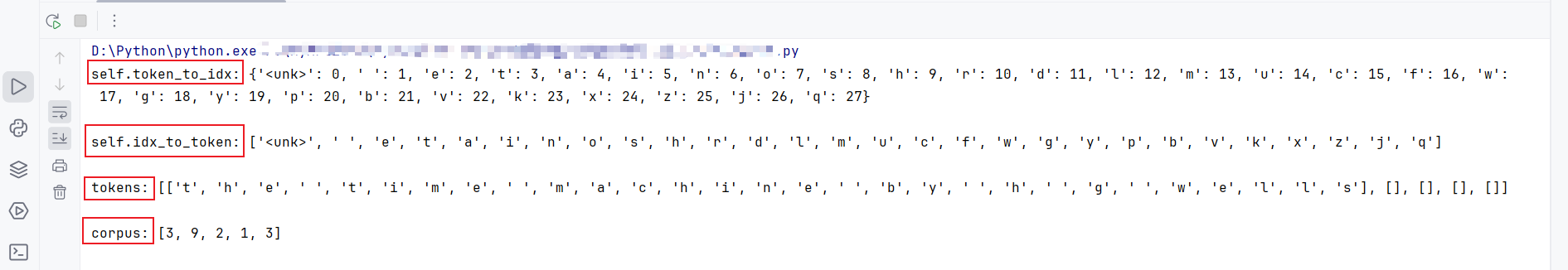
如果打印其中的一些中间结果：

print('self.token\_to\_idx:', self.token\_to\_idx)  
print('self.idx\_to\_token:',self.idx\_to\_token)

print('tokens:',tokens[:5])

print('corpus:', corpus[:5])

则有：



从打印的结果中可以看出每一个对象中的元素。至此就完成了文本到数字的转换。

接下里我们进行序列的数据的加载，加载数据的时候有随机加载和顺序抽样的加载两种方式，他们的目的都是通过随机采样生成minibatch的子序列：

def seq\_data\_iter\_random(corpus, batch\_size, num\_steps):  
    corpus = corpus[random.randint(0, num\_steps - 1):]  
    num\_subseqs = (len(corpus) - 1) // num\_steps  
    initial\_indices = list(range(0, num\_subseqs \* num\_steps, num\_steps))  
    random.shuffle(initial\_indices)  
   
    def data(pos):  
        return corpus[pos: pos + num\_steps]  
  
    num\_batches = num\_subseqs // batch\_size  
    for i in range(0, batch\_size \* num\_batches, batch\_size):  
        initial\_indices\_per\_batch = initial\_indices[i: i + batch\_size]  
        X = [data(j) for j in initial\_indices\_per\_batch]  
        Y = [data(j + 1) for j in initial\_indices\_per\_batch]  
        yield torch.tensor(X),torch.tensor(Y)

我们来稍微解释上述的代码：

（1）corpus[random.randint(0, num\_steps-1):]：这行代码通过随机选择一个起始索引，从 corpus 中裁剪出一个子序列。这样做可以减少序列之间的相关性，因为每次迭代都可能从语料库的不同位置开始采样。

（2）num\_subseqs = (len(corpus)-1)//num\_steps：计算裁剪后的语料库可以分成多少个长度为 num\_steps 的序列。这里减1是为了保证最后一个序列也能有 num\_steps 个元素。

（3）list(range(0, num\_subseqs\*num\_steps,num\_steps))：创建一个初始索引列表，包含所有可能的起始索引，用于生成长度为 num\_steps 的序列。

（3）shuffle(initial\_indices)：对初始索引列表进行随机洗牌，确保相邻的minibatch不是连续的。

（4）X = [data(j) for j in initial\_indices\_per\_batch]，Y = [data(j + 1) for j in initial\_indices\_per\_batch]：同样使用列表推导式和 data 函数，生成当前批次的标签序列Y，它是输入序列X中每个词的下一个词。

上述的加载数据的方式的是随机的，接下来我们看看顺序加载的方式：

def seq\_data\_iter\_sequential(corpus, batch\_size, num\_steps):  
    offset = random.randint(0, num\_steps)  
    num\_tokens = ((len(corpus) - offset - 1) // batch\_size) \* batch\_size  
    Xs = torch.tensor(corpus[offset: offset + num\_tokens])  
    Ys = torch.tensor(corpus[offset + 1: offset + 1 + num\_tokens])  
    Xs, Ys = Xs.reshape(batch\_size, -1), Ys.reshape(batch\_size, -1)  
    num\_batches = Xs.shape[1] // num\_steps  
    for i in range(0, num\_steps \* num\_batches, num\_steps):  
        X = Xs[:, i: i + num\_steps]  
        Y = Ys[:, i: i + num\_steps]  
        yield X, Y

这个按照顺序加载数据的函数使用了顺序划分来生成序列数据的minibatch，这与随机抽样不同，它保证了序列数据的连续性。这对于某些模型可能是有益的，尤其是当模型需要捕捉长期依赖关系时。

之后我们来创建一个SeqDataLoader类：

class SeqDataLoader:  
    """加载顺序数据的迭代器"""  
    def \_\_init\_\_(self, batch\_size, num\_steps, use\_random\_iter, max\_tokens):  
        """两种加载数据的形式`"""  
        if use\_random\_iter:  
            self.data\_iter\_fn = seq\_data\_iter\_random  
        else:  
            self.data\_iter\_fn = seq\_data\_iter\_sequential  
        self.corpus, self.vocab = load\_corpus\_time\_machine(max\_tokens)  
        self.batch\_size, self.num\_steps = batch\_size, num\_steps  
  
    def \_\_iter\_\_(self):  
        return self.data\_iter\_fn(self.corpus, self.batch\_size, self.num\_steps)

这个代码简单，这里稍微解释一下其中的一句：

self.corpus, self.vocab = load\_corpus\_time\_machine(max\_tokens)：调用 load\_corpus\_time\_machine 函数来加载语料库和词汇表，其中 max\_tokens 指定了词汇表的最大词汇数。self.corpus 存储加载的文本数据，self.vocab 存储词汇表对象。

这个类的主要用途是提供一个统一的接口来加载序列数据，无论数据是随机采样还是顺序划分。用户只需要创建 SeqDataLoader 的一个实例并指定加载方式，然后就可以通过迭代器协议来访问数据。

最后我们创建一个函数来加载最终的数据：

def load\_data\_time\_machine(batch\_size, num\_steps, use\_random\_iter=False, max\_tokens=10000):  
    data\_iter = SeqDataLoader(batch\_size, num\_steps, use\_random\_iter, max\_tokens)  
    return data\_iter, data\_iter.vocab

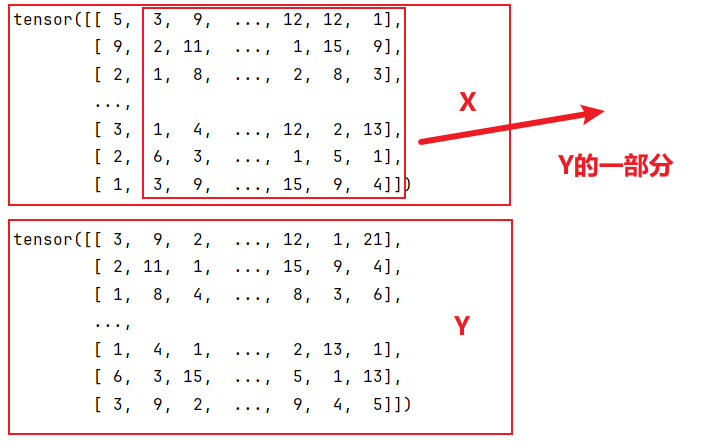
*# 定义加载的batch和时间步骤*  
batch\_size, num\_steps = 32, 35  
# 之前定义的函数  
train\_iter, vocab = load\_data\_time\_machine(batch\_size, num\_steps)  
for X, Y in train\_iter:  
    print(X,'\n')  
    print(Y)  
    break

上述的代码是加载我们的样本的数据的代码其中：

batch\_size：批量大小

num\_steps：时间步（窗口大小，即根据前面多少个词做预测）

我们在最后打印了一个样本的数据，输出如下：



可以看出，张量Y和张量X在shape上是一致的，元素上来看是往前一个step的形式。大家从加载代码形式上也看的出来。

1. **独热编码**

在上一步骤中train\_iter中，每个词元都表示为一个数字索引，将这些索引直接输入神经网络可能会使学习变得困难。那为什么变得困难呢？这里解释一下：

（1）高维空间：自然语言的词汇量是巨大的，如果每个词元都直接映射到一个唯一的数字索引，那么这个索引空间将会非常庞大。这会导致模型需要处理的输入空间维度非常高，增加了模型的复杂度和计算成本。

（2）稀疏表示：使用数字索引作为词元的表示，会得到一个非常稀疏的向量表示。在这种表示中，大部分元素都是0，只有少数几个元素是1，表示当前词元的索引。这种稀疏性不利于模型捕捉词元之间的相关性。

（3）语义信息缺失：直接使用数字索引作为词元的表示，忽略了词元的内在语义信息。神经网络需要从这些数字中学习到词元的语义，这通常是非常困难的。

（4）难以捕捉上下文信息：数字索引本身不包含任何关于词元在句子中位置的信息，而这对于理解词元的上下文含义是非常重要的。

（5）可扩展性问题：如果使用数字索引，每当词汇表增加新词时，模型就需要重新训练以适应新的索引空间，这在实践中是不现实的。

1. 无法处理未知词元：如果一个词元在训练数据中没有出现过，使用数字索引的模型可能无法有效地处理这个词元。

因此在实际工程中我们需要将每一个词语表示为更具表现力的特征向量。最简单的表示称为独热编码（one‐hot encoding），也就是将每个索引映射为相互不同的单位向量。因为之前我们设置了：

batch\_size, num\_steps = 32, 35

即我们每次采样的小批量数据形状是二维张量：（批量大小，时间步数）。如果我们打印一下tensor X的形状则为：

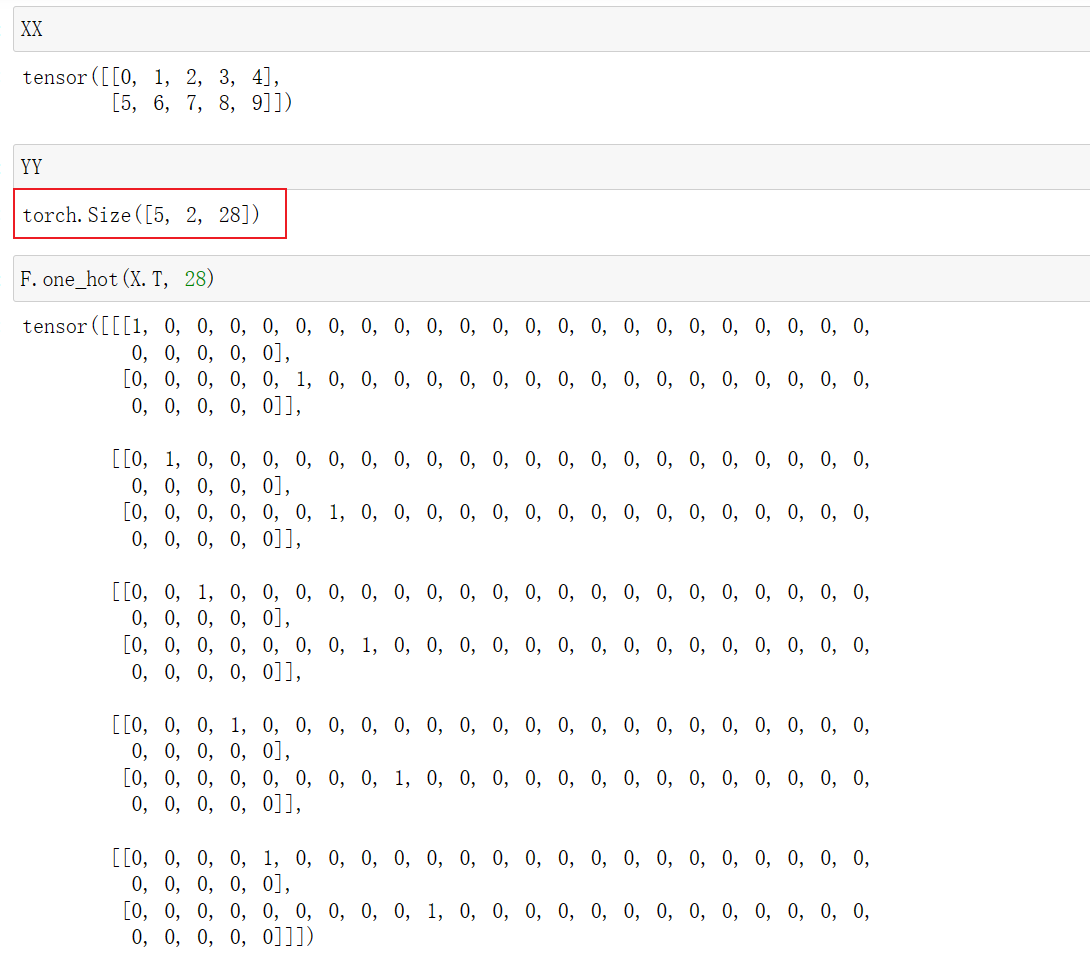
print(X.shape)  
# torch.Size([32, 35])

也就是X中存储的是32个样本，每个样本有35个词。我们这里使用one-hot编码，就是把任意向量表示成长度为len(vocab)的编码，因为使用的是char构建的[词典](https://so.csdn.net/so/search?q=%E8%AF%8D%E5%85%B8&spm=1001.2101.3001.7020)，长度为28（idx\_to\_token列表的长度，每一个词使用长度为28的向量来表示）。

如：

import torch  
from torch.nn import functional as F  
XX = torch.arange(10).reshape((2, 5))  
YY = F.one\_hot(X.T, 28).shape

则有：



也就是说通过one-hot编码再加上len(vocab)的维度我们可以对Tensor的输入维度进行转换。转置后模型的输入格式为：

[seq\_len, batch\_size, input\_size]

即：

[时间步,批次大小,样本特征长度]

后续我们在编写RNN模型的结构的时候会用到One-Hot编码。

Q&A:这里为什么要对张量X进行转置的操作？以下原因可供参考：

1. 将 X 转换为列向量意味着每一列代表一个时间步，这与RNN处理序列数据的方式相匹配。
2. X.T 操作通常意味着将词元索引的批量数据从 [batch\_size, seq\_len] 转换为 [seq\_len, batch\_size]，这样每个时间步上的所有样本可以并行处理。也就是说转置后方便之后从时间维度来遍历Tensor（seq\_len顺序遍历就是时间维度）。
3. **初始化RNN的模型参数**

接下来我们来构建RNN模型，在构建RNN模型的时候我们需要先进行RNN模型参数的初始化：

# vocab\_size 即词典的长度，也就是之前idx\_to\_token的长度

def get\_params(vocab\_size, num\_hiddens, device):

num\_inputs = num\_outputs = vocab\_size

def normal(shape):

return torch.randn(size=shape, device=device) \* 0.01

# 隐藏层参数

W\_xh = normal((num\_inputs, num\_hiddens))

W\_hh = normal((num\_hiddens, num\_hiddens))

b\_h = torch.zeros(num\_hiddens, device=device)

# 输出层参数

W\_hq = normal((num\_hiddens, num\_outputs))

b\_q = torch.zeros(num\_outputs, device=device)

params = [W\_xh, W\_hh, b\_h, W\_hq, b\_q]

for param in params:

param.requires\_grad\_(True)

return params

这里我们稍微解释一下上述的代码：

初始化循环神经网络（RNN）参数的函数 get\_params。这个函数接收三个参数：vocab\_size（词汇表的大小），num\_hiddens（隐藏层的神经元数量），和 device（用于指定计算设备，如CPU或GPU）。同时在训练语言模型时，输入和输出来自相同的词表。因此，它们具有相同的维度，即词表的大小。

1. 函数内部首先定义了num\_inputs和num\_outputs，它们都等于 vocab\_size。这是因为在RNN中，输入和输出通常都是词元的one-hot编码形式，所以它们的维度是相同的。
2. 之后的normal方法用于创建具有正态分布初始化的权重矩阵。它接收一个形状参数 shape，并返回一个随机初始化的张量，其元素值乘以了一个很小的数（在这里是0.01），这有助于模型训练的稳定性。
3. 之后对权重矩阵和偏置向量进行初始化：

W\_xh：输入层到隐藏层的权重矩阵。

W\_hh：隐藏层到隐藏层的权重矩阵，RNN特有的，负责捕捉时序信息。

b\_h：隐藏层的偏置向量。

W\_hq：隐藏层到输出层的权重矩阵。

b\_q：输出层的偏置向量。

1. 使用 param.requires\_grad\_(True)保证这些参数的梯度将被计算和存储，这是训练过程中进行梯度下降的基础。
2. **循环神经网络模型**

定义循环神经网络模型，我们首先需要一个init\_rnn\_state函数在初始化时返回隐层状态。这个函数的返回是一个张量，张量全用0填充，形状为（批量大小，隐藏单元数）。也就是在时间t=0的时候是没有上一个隐藏状态h的，在后面的章节中我们将会遇到隐状态包含多个变量的情况，而使用元组可以更容易地处理些。（动手学深度学习原话）：

def init\_rnn\_state(batch\_size, num\_hiddens, device):  
return (torch.zeros((batch\_size, num\_hiddens), device=device), )

接下来定义如何在一个时间步内计算隐状态和输出。循环神经网络模型通过inputs最外层的维度（也就是上述的seq\_len维度）实现循环，以便逐时间步更新小批量数据的隐状态H，我们来看一下代码：

def rnn(inputs, state, params):  
 *# inputs 的形状(时间步数，批量大小，词典长度)* W\_xh, W\_hh, b\_h, W\_hq, b\_q = params  
 *# 前一个时刻的隐藏状态，也先进行初始化* H, = state  
 outputs = []  
 *# X的形状：(批量大小，词典长度)  
 # 之前的转置的的作用将时间步调整到最前，接着使用循环来计算每一个时间步* for X in inputs:  
 *# 计算当前的H,mm(H, W\_hh)中的H是上一个时间步的隐藏状态* H = torch.tanh(torch.mm(X, W\_xh) + torch.mm(H, W\_hh) + b\_h)  
 *# 使用当前的H来进行预测,Y的形状：(批量大小，词典长度)* Y = torch.mm(H, W\_hq) + b\_q  
 *# 把结果矩阵加到列表outputs中，之后outputs的形状为 (时间步数，批量大小，词典长度)* outputs.append(Y)  
 *# cat之后返回的shape为（时间步数 \* 批量大小，词典长度）的矩阵  
 # 隐藏层H的形状为 （批量大小，隐藏单元数）* return torch.cat(outputs, dim=0), (H,)

我们也来解释一下上述代码：

定义的rnn 的函数，它实现了一个简单的循环神经网络（RNN）的前向传播过程。这个函数接收三个参数：inputs（输入序列），state（初始隐藏状态），和 params（模型参数）。

1. state是一个包含初始隐藏状态的元组，其中H是初始隐藏状态的变量
2. outputs是一个空列表，用于存储每个时间步的输出。
3. for X in inputs: 循环遍历输入序列中的每个时间步。在这个循环中，X 是当前时间步的输入，其形状为 (batch\_size, vocab\_size)。
4. 使用torch.cat函数将 outputs 列表中的所有输出张量按维度dim=0（时间步维度）连接起来，形成最终的输出序列。同时，将最后一个隐藏状态H作为新的隐藏状态返回。
5. 函数返回两个值，第一个是连接后的输出序列，第二个是包含最新隐藏状态的元组,隐藏层H的形状为（批量大小，隐藏单元数）。

与《动手学深度学习》书中的章节一致，在定义了所有需要的函数之后，接下来我们创建一个类来包装这些函数，并存储从零开始实现的循环神经网络模型的参数。

class RNNModelScratch:  
 *'''从零开始实现的循环神经网络模型'''* def \_\_init\_\_(self, vocab\_size, num\_hiddens, device, get\_params,init\_state, forward\_fn):  
 self.vocab\_size, self.num\_hiddens = vocab\_size, num\_hiddens  
 self.params = get\_params(vocab\_size, num\_hiddens, device)  
 self.init\_state, self.forward\_fn = init\_state, forward\_fn  
  
 def \_\_call\_\_(self, X, state):  
 *# 转换为浮点数据， X.T 表示转置 X（将时间步和批次维度转置）  
 # 转置之后X的shape为（时间步数 \* 批量大小，词典长度）* X = F.one\_hot(X.T, self.vocab\_size).type(torch.float32)  
 *# print('X.shape:',X.shape) # torch.Size([35, 32, 28])* return self.forward\_fn(X, state, self.params)  
 *# 初始化RNN的状态* def begin\_state(self, batch\_size, device):  
 return self.init\_state(batch\_size, self.num\_hiddens, device)

有的小伙伴肯定会问，我怎么知道我设置的tensor的形状是否正确呢，不着急，接下来我们打印其中的一些数据的形状：

def try\_gpu(i=0):  
 if torch.cuda.device\_count() >= i + 1:  
 return torch.device(f'cuda:{i}')  
 return torch.device('cpu')  
  
for X, Y in train\_iter:  
 num\_hiddens = 512

*# 前向传播函数为forward\_fn=rnn*  
 net = RNNModelScratch(len(vocab), num\_hiddens, try\_gpu(), get\_params,init\_rnn\_state, rnn)  
 state = net.begin\_state(X.shape[0], try\_gpu())  
 Y, new\_state = net(X.to(try\_gpu()), state)  
 print('Y.shape',Y.shape)

print('len(new\_state):',len(new\_state)) print('new\_state[0].shape:',new\_state[0].shape) break

输出的结果如下：

Y.shape torch.Size([1120, 28])

len(new\_state): 1

new\_state[0].shape: torch.Size([32, 512])

Y的shape中的1120(35\*32)即对应35步32个batch的1120个词，28代表在vocab中的28个词。new\_state是一个元组，里面的元素就是下一个状态的h其shape为（批量大小，隐层单元数量）。

1. **预测**

首先定义预测函数来生成prefix之后的新字符，其中的prefix是一个用户提供的包含多个字符的字符串，在循环遍历prefix中的开始字符时，需要不断地将隐状态传递到下一个时间步，但是不生成任何输出，这被称为预热（warm-up）期。因为在此期间模型会自我更新（例如，更新隐状态），但不会进行预测。预热期结束后，隐状态的值通常比刚开始的初始值更适合预测，从而预测字符并输出它们。因此我们有必要进行“预热”的操作。

def predict\_ch8(prefix, num\_preds, net, vocab, device):  
 *'''函数目的是在给定的前缀 prefix 后面生成新的字符序列'''* state = net.begin\_state(batch\_size=1, device=device)   
 outputs= [vocab[prefix[0]]]  
 get\_input = lambda: torch.tensor([outputs[-1]], device=device).reshape((1,1))  
 for y in prefix[1:]:  
 \_, state = net(get\_input(), state)  
 outputs.append(vocab[y])   
 for \_ in range(num\_preds):   
 y, state = net(get\_input(), state)  
 *# y是一个 1 x vocab 的tensor，取最大值的下标进行后续输出* outputs.append(int(y.argmax(dim=1).reshape(1)))  
 return ''.join([vocab.idx\_to\_token[i] for i in outputs])

我们来解释一下上述的代码：

（1）predict\_ch8()函数接受以下参数：

prefix: 一个字符串，作为生成新字符序列的起点。

num\_preds: 一个整数，表示要生成的预测字符的数量。

net: RNNModelScratch 类实例，用于生成预测。

vocab: 一个词汇表对象，与上文一致。

device: 指定模型运行的设备（CPU 或 GPU）。

（2）state = net.begin\_state(batch\_size=1, device=device) 调用net（即 RNNModelScratch实例）的begin\_state 方法来初始化 RNN 的状态。由于是对单个字符串进行预测，所以batch\_size设置为 1。

（3）outputs = [vocab[prefix[0]]] 初始化outputs列表，其中包含第一个词即的索引，也可以认为第一个词是没有办法预测的，因此先放入。vocab[prefix[0]]将前缀的第一个字符映射到其在词汇表中的索引。

（4）lambda: torch.tensor([outputs[-1]], device=device).reshape((1,1)) 定义了一个lambda函数get\_input，是从outputs中获取最新预测的值作为下一次的输入。它返回当前outputs列表中最后一个元素的张量，即最近预测的字符的索引。这个lambda函数用于获取模型的下一个输入。

（5）for y in prefix[1:]: 循环遍历前缀中的剩余字符（从第二个字符开始），以“预热”模型状态，但不关心输出。outputs.append(vocab[y]) 在预热期间，将前缀中的每个字符的真实的索引添加到outputs列表中。实际上我们是在模拟模型在处理真实数据时的行为，即模型在每一步都会接收到正确的输入。

（6）for \_ in range(num\_preds): 预测num\_preds步骤，生成num\_preds个预测字符。y, state = net(get\_input(), state) 使用模型net和当前状态 state 进行预测，并将预测结果和更新后的状态返回

（7）outputs.append(int(y.argmax(dim=1).reshape(1))) 将预测的字符索引（y的最大值的索引）添加到 outputs 列表中。reshape(1)将其转换为一个标量。

（8）return ''.join([vocab.idx\_to\_token[i] for i in outputs]) 将outputs列表中的索引转换回对应的字符，并将它们连接起来形成最终的预测字符串。

假设我们现在的调用函数写成如下方式:

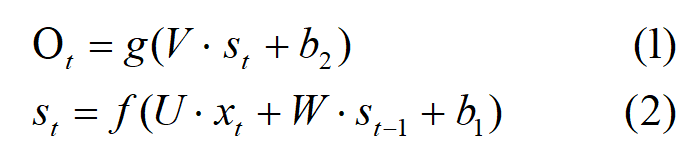
num\_hiddens = 512  
net = RNNModelScratch(len(vocab), num\_hiddens, try\_gpu(), get\_params,init\_rnn\_state, rnn)  
res = predict\_ch8('time traveller ', 10, net, vocab, d2l.try\_gpu())  
print(res)

*# 则输出为：  
# time traveller dkhxscuats*

能看到对于一个没有经过训练的网络，它输出了“dkhxscuats”，字符串的长度为10。注意，每一次运行上述的调用结果也是不一样的。

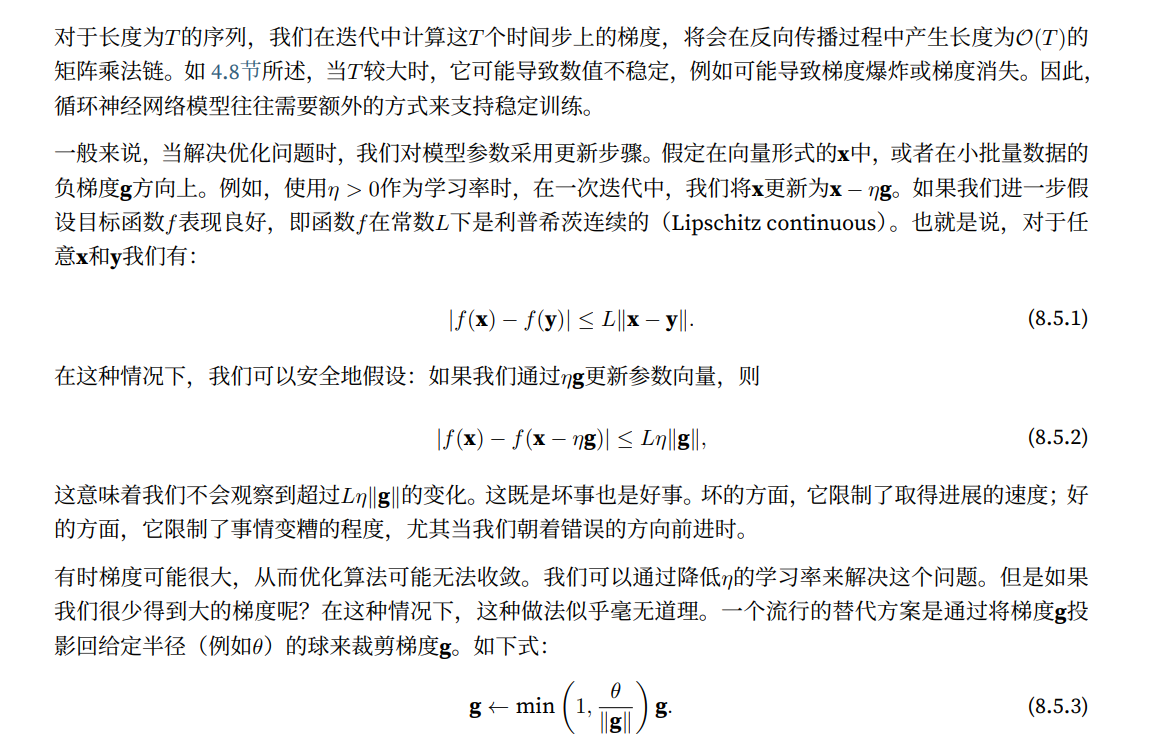
1. **梯度裁剪**

我们回忆一下上述RNN的结构：



为什么需要进行梯度裁剪，梯度裁剪（Gradient Clipping）是一种常用的技术，用于防止梯度爆炸（Gradient Explosion）问题，同时也能在一定程度上减轻梯度消失（Gradient Vanishing）的问题。

梯度裁剪的目的是将梯度的更新限制在一个最大值内，超过这个值的梯度将被缩放，详细的解释我这里参考《动手学深度学习》章节内容：



上述梯度裁剪的方案代码可以写成如下方式：

def grad\_clipping(net, theta):   
 *"""裁剪梯度"""* if isinstance(net, nn.Module):  
 params = [p for p in net.parameters() if p.requires\_grad]  
 else:  
 params = net.params  
 norm = torch.sqrt(sum(torch.sum((p.grad \*\* 2)) for p in params))  
 if norm > theta:  
 for param in params:  
 param.grad[:] \*= theta / norm

解释一下代码：

（1）上述代码先检查传入的net参数是否是 PyTorch 的 nn.Module 类型。params = [p for p in net.parameters() if p.requires\_grad] 如果 net是 nn.Module类型，通过遍历模块的参数并使用列表推导式，创建一个包含所有需要梯度的参数（即requires\_grad=True）的列表。

（2）如果net不是 nn.Module 类型，那么假设 net对象有一个名为 params 的属性，它包含了所有的参数。接着计算每一个梯度的平方合成L2范数并赋值给norm变量（所有层的所有梯度的平方求和再开根号得到L2范数。之后进行梯度的范围限定。

1. **训练**

接下来就可以开始训练了，之前已经定义了一个batch怎么计算，也就是forward，接下来定义函数在一个epoch内训练模型

class Accumulator:  
 def \_\_init\_\_(self, n):  
 self.data = [0.0] \* n  
  
 *# 进行累加* def add(self, \*args):  
 self.data = [a + float(b) for a, b in zip(self.data, args)]  
  
 def reset(self):  
 self.data = [0.0] \* len(self.data)  
  
 def \_\_getitem\_\_(self, idx):  
 return self.data[idx]  
  
  
class Timer:  
 *"""Record multiple running times."""* def \_\_init\_\_(self):  
 *"""Defined in :numref:`subsec\_linear\_model`"""* self.times = []  
 self.start()  
  
 def start(self):  
 *"""Start the timer."""* self.tik = time.time()  
  
 def stop(self):  
 *"""Stop the timer and record the time in a list."""* self.times.append(time.time() - self.tik)  
 return self.times[-1]  
  
 def avg(self):  
 *"""Return the average time."""* return sum(self.times) / len(self.times)  
  
 def sum(self):  
 *"""Return the sum of time."""* return sum(self.times)  
  
 def cumsum(self):  
 *"""Return the accumulated time."""* return np.array(self.times).cumsum().tolist()  
  
  
def train\_epoch\_ch8(net, train\_iter, loss, updater, device, use\_random\_iter):  
 *"""训练网络一个迭代周期"""* state, timer = None, Timer()  
 metric = Accumulator(2)  
 for X, Y in train\_iter:  
 if state is None or use\_random\_iter:  
 *# 在第一次迭代或使用随机抽样时初始化state* state = net.begin\_state(batch\_size=X.shape[0], device=device)  
 *# 如果是连续抽样的，就不用重新做初始化* else:  
 if isinstance(net, nn.Module) and not isinstance(state, tuple):  
 *# state对于nn.GRU是个张量* state.detach\_()  
 else:  
 *# state对于nn.LSTM或对于我们从零开始实现的模型是个张量* for s in state:  
 s.detach\_()  
 *# 转置，时间维度放在最前* y = Y.T.reshape(-1)  
 X, y = X.to(device), y.to(device)  
 y\_hat, state = net(X, state)  
 *# 计算损失* l = loss(y\_hat, y.long()).mean()  
 *# 如果使用的是torch.optim.Optimizer* if isinstance(updater, torch.optim.Optimizer):  
 updater.zero\_grad()  
 l.backward()  
 *# 梯度剪裁* grad\_clipping(net, 1)  
 updater.step()  
 *# 自定义的* else:  
 l.backward()  
 grad\_clipping(net, 1)  
 updater(batch\_size=1)  
 metric.add(l \* y.numel(), y.numel())  
 return math.exp(metric[0] / metric[1]), metric[1] / timer.stop()

稍微介绍一下上述的一些代码：

（1）Accumulator(2) 创建一个 Accumulator 对象，用于累积训练过程中的损失和样本数量。

（2）如果 updater 是 torch.optim.Optimizer 类型，则先清零梯度，然后执行反向传播，接着执行梯度裁剪，最后更新模型参数。如果 updater 不是优化器类型，则直接执行反向传播，梯度裁剪，然后调用 updater 进行更新。

（3）metric.add(l \* y.numel(), y.numel()) 将损失和样本数量添加到metric中。

（4）计算并返回平均损失的指数（即 exp(average loss)）。计算并返回每秒处理的样本数量，即 metric[1]（总样本数量）除以 timer.stop()（总时间）。

这里简单的注意一点就是：use\_random\_iter：会导致隐藏状态更新的方式是不一样：

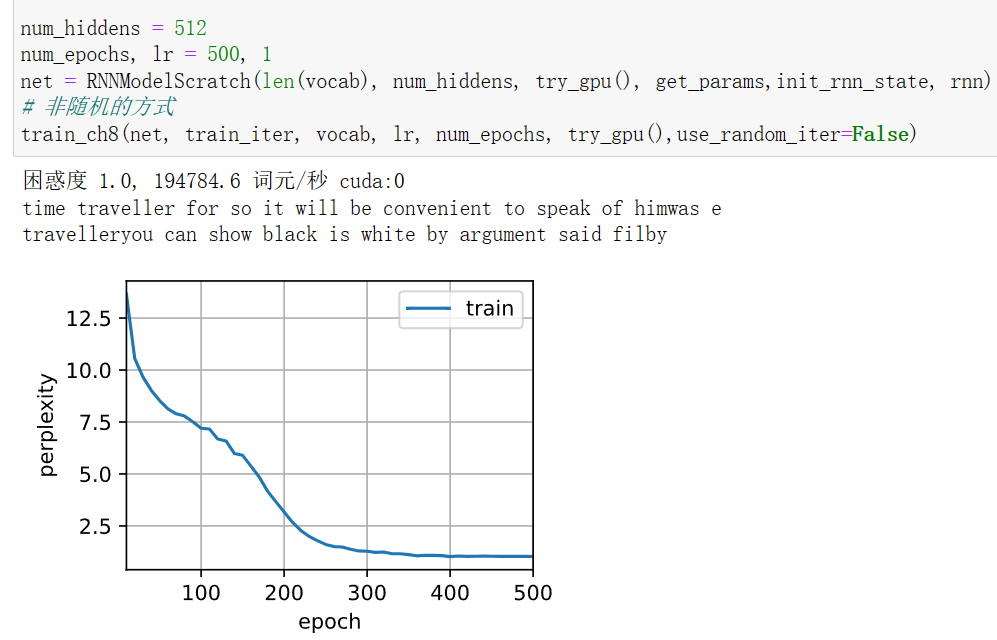
（1）如果是随机的，则下一个batc和上一个batch的第i个输入没有关系

（2）如果非随机的，则下一个batch和上一个batch的第i个输入是连续的

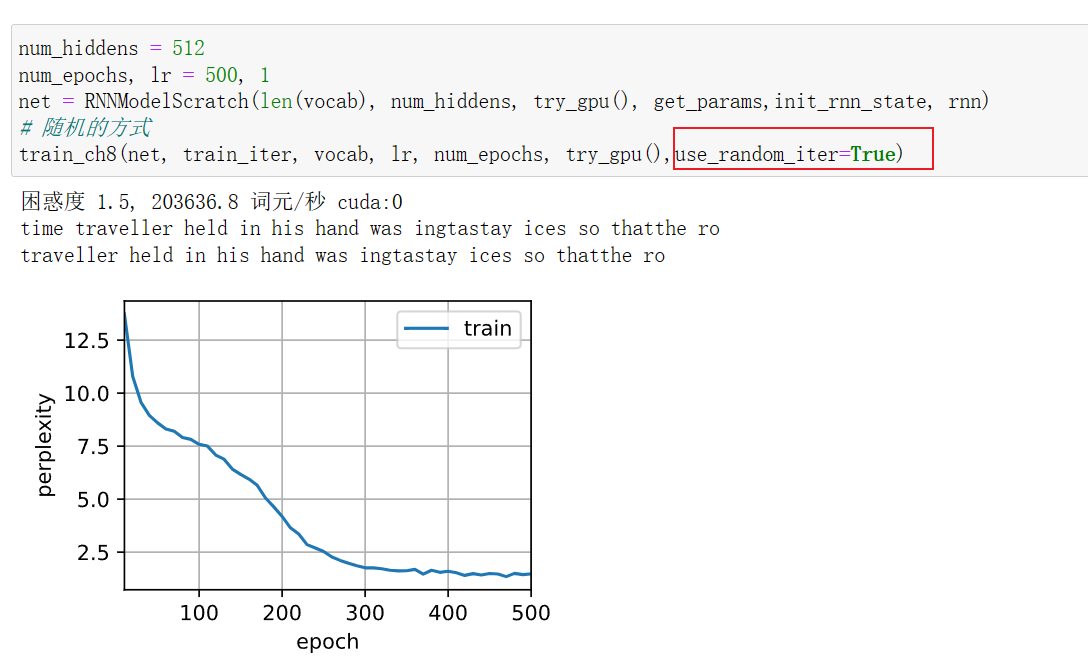
接下来就可以定义训练函数了：

from matplotlib\_inline import backend\_inline  
def use\_svg\_display():  
 backend\_inline.set\_matplotlib\_formats('svg')  
  
def set\_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend):  
 axes.set\_xlabel(xlabel)  
 axes.set\_ylabel(ylabel)  
 axes.set\_xscale(xscale)  
 axes.set\_yscale(yscale)  
 axes.set\_xlim(xlim)  
 axes.set\_ylim(ylim)  
 if legend:  
 axes.legend(legend)  
 axes.grid()  
  
  
  
from IPython import display  
from matplotlib import pyplot as plt  
  
class Animator:  
 *"""For plotting data in animation."""* def \_\_init\_\_(self, xlabel=None, ylabel=None, legend=None, xlim=None,  
 ylim=None, xscale='linear', yscale='linear',  
 fmts=('-', 'm--', 'g-.', 'r:'), nrows=1, ncols=1,  
 figsize=(3.5, 2.5)):  
 *"""Defined in :numref:`sec\_softmax\_scratch`"""  
 # Incrementally plot multiple lines* if legend is None:  
 legend = []  
 use\_svg\_display()  
 self.fig, self.axes = plt.subplots(nrows, ncols, figsize=figsize)  
 if nrows \* ncols == 1:  
 self.axes = [self.axes, ]  
 *# Use a lambda function to capture arguments* self.config\_axes = lambda: set\_axes(  
 self.axes[0], xlabel, ylabel, xlim, ylim, xscale, yscale, legend)  
 self.X, self.Y, self.fmts = None, None, fmts  
  
 def add(self, x, y):  
 *# Add multiple data points into the figure* if not hasattr(y, "\_\_len\_\_"):  
 y = [y]  
 n = len(y)  
 if not hasattr(x, "\_\_len\_\_"):  
 x = [x] \* n  
 if not self.X:  
 self.X = [[] for \_ in range(n)]  
 if not self.Y:  
 self.Y = [[] for \_ in range(n)]  
 for i, (a, b) in enumerate(zip(x, y)):  
 if a is not None and b is not None:  
 self.X[i].append(a)  
 self.Y[i].append(b)  
 self.axes[0].cla()  
 for x, y, fmt in zip(self.X, self.Y, self.fmts):  
 self.axes[0].plot(x, y, fmt)  
 self.config\_axes()  
 display.display(self.fig)  
 display.clear\_output(wait=True)  
  
  
def sgd(params, lr, batch\_size):  
 with torch.no\_grad():  
 for param in params:  
 param -= lr \* param.grad / batch\_size  
 param.grad.zero\_()  
  
def train\_ch8(net, train\_iter, vocab, lr, num\_epochs, device,use\_random\_iter=False):  
 *"""训练模型"""  
 # 虽说是语言模型，但实际上是多分类问题，预测下一个字符的可能性是最大的* loss = nn.CrossEntropyLoss()  
 animator = Animator(xlabel='epoch', ylabel='perplexity',legend=['train'], xlim=[10, num\_epochs])  
 if isinstance(net, nn.Module):  
 updater = torch.optim.SGD(net.parameters(), lr)  
 else:  
 *# 不然就调用咱自己之前搞的* updater = lambda batch\_size: sgd(net.params, lr, batch\_size)  
 *# 输出50个字符* predict = lambda prefix: predict\_ch8(prefix, 50, net, vocab, device)  
 *# 训练和预测* for epoch in range(num\_epochs):  
 *# 困惑度和每秒计算样本个数* ppl, speed = train\_epoch\_ch8(net, train\_iter, loss, updater, device, use\_random\_iter)  
 *# 每10个epoch看一看效果~* if (epoch + 1) % 10 == 0:  
 print(predict('time traveller'))  
 animator.add(epoch + 1, [ppl])  
 print(f'困惑度 {ppl:.1f}, {speed:.1f} 词元/秒 {str(device)}')  
 print(predict('time traveller'))  
 print(predict('traveller'))

接下来我们来调用来看具体的训练输出结果，先使用非随机的方式：



然后我们再次使用随机的方式：



至此我们就从零开始实现上述循环神经网络模型，一步一步看来实现这个RNN还是很麻烦的，不过可以了解整个建模的过程。

1. **参考文献**
2. PyTorch官方文档：https://pytorch.org/docs/stable/index.html
3. 《动手学深度学习》-李沐
4. torchtext官方文档：https://torchtext.readthedocs.io/en/latest/
5. IMDB电影评论数据集：https://ai.stanford.edu/~amaas/data/sentiment/
6. [老饼讲解机器学习 (www.bbbdata.com)](https://www.bbbdata.com/)

<https://zybuluo.com/hanbingtao/note/541458>

<https://dennybritz.com/posts/wildml/recurrent-neural-networks-tutorial-part-3/>

http://proceedings.mlr.press/v28/pascanu13.pdf

1、[Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs](http://link.zhihu.com/?target=http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/)

2、[Recurrent Neural Networks Tutorial, Part 2 – Implementing a RNN with Python, Numpy and Theano](http://link.zhihu.com/?target=http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-2-implementing-a-language-model-rnn-with-python-numpy-and-theano/)

3、[Recurrent Neural Networks Tutorial, Part 3 – Backpropagation Through Time and Vanishing Gradients](http://link.zhihu.com/?target=http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/)

4、[Recurrent Neural Network Tutorial, Part 4 – Implementing a GRU/LSTM RNN with Python and Theano](http://link.zhihu.com/?target=http://www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-grulstm-rnn-with-python-and-theano/)

https://dennybritz.com/posts/wildml/recurrent-neural-networks-tutorial-part-3/

http://colah.github.io/posts/2015-08-Understanding-LSTMs/