实验8 使用 RNN 生成简单序列

一、 实验内容

实验内容：首先让神经网络模型学习形如0^n1^n0n1n 形式的上下文无关语法。然后再让模型尝试去生成这样的字符串。在实验的流程中将演示RNN 及LSTM 相关函数的使用方法。探究什么是上下文无关文法、使用RNN 或LSTM 模型生成简单序列的方法、探究RNN 记忆功能的内部原理

实验环境：Python-3.11;PyTorch-cu121;CUDA-12.1;VS code/JupyterNotebook

二、 实验过程和结果

1. 引入相关包

```python
import torch
import torch.nn as nn
import torch.optim
from torch.autograd import Variable
from collections import Counter
import matplotlib
import matplotlib.pyplot as plt
from matplotlib import rc
import numpy as np
%matplotlib inline
```

Python

2. 生成训练数据集

```python
samples = 2000
sz = 10
probability = 1.0 * np.array([10,6,4,3,1,1,1,1,1,1])
probability = probability[ : sz]
probability = probability/sum(probability)
```

这段代码首先定义一些超参数，代码中sample代表训练集数据的数量，sz代表训练集中01字符的最大长度为10个，针对01个数的取值n设定其取值从1-10的可能性权重数组possibility，经过系列处理成为n取值各值得可能性，被用于np.random.choice的参数中。

```python
train_set = []

for m in range(samples):
    n = np.random.choice(range(1,sz+1),p = probability)
    inputs =[0]* n + [1]*n
    inputs.insert(0,3)
    inputs.append(2)
    train_set.append(inputs)
```

```python
valid_set = []

for m in range(samples//10):
    n = np.random.choice(range(1,sz+1),p = probability)
    inputs =[0]* n + [1]*n
    inputs.insert(0,3)
    inputs.append(2)
    valid_set.append(inputs)
```

这两段代码结构相同，分别用于生成训练数据集和校验数据集，不同处为校验集大小只有训练集的约1/10，首先根据概率选取01个数n，运用列表操作在01串前插入3，后追加2，形成一条数据再加入到数据集中。

```
for m in range(2):
    n = sz+m
    inputs =[0]* n + [1]*n
    inputs.insert(0,3)
    inputs.append(2)
    valid_set.append(inputs)

np.random.shuffle(valid_set)
```

这段代码在先前校验集基础上追加了两条01长度为11和12的数据，并将校验集顺序打乱。

3. 定义 RNN 模型

```
class SimpleRNN(nn.Module):
    def __init__(self,input_size,hidden_size,output_size,num_layers=1):
        super(SimpleRNN,self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.embedding = nn.Embedding(input_size,hidden_size)
        self.rnn = nn.RNN(hidden_size,hidden_size,num_layers,batch_first=True)
        self.fc = nn.Linear(hidden_size,output_size)
        self.softmax = nn.LogSoftmax(dim = 1)

    def forward(self,input,hidden):
        x = self.embedding(input)
        output,hidden =self.rnn(x,hidden)
        output= output[:,-1,:]
        output =self.fc(output)
        output= self.softmax(output)
        return output,hidden
```

```
  def initHidden(self):
      hidden = Variable(torch.zeros(self.num_layers,1,self.hidden_size))
      return hidden
rnn = SimpleRNN(input_size=4,hidden_size=2,output_size=3)
criterion = torch.nn.NLLLoss()
optimizer = torch.optim.Adam(rnn.parameters(),lr = 0.001)
```

Python

定义RNN类的方法与前面的实验大致相同，按层的顺序首先是embedding层将一串字符映射为一个n维向量，将数据变为hidden_size的长度，代替了onehot编码，embedding层存在可学习参数，在训练的过程中学习到最佳的权重；其次是经过封装的RNN层，经过嵌入后rnn输入尺寸和隐层尺寸相同，rnn的层数为1，最后经过一个全连接层，全连接层的输出神经元个数在此例中为3，代表012，2代表的是字符串的结束，输出经过softmax处理后变为三种输出的预测概率。

然后对SimpleRNN进行实例化，定义输入隐层输出层的大小。损失函数采用NLLloss负对数似然损失函数，经过查阅，NLLLoss函数配合softmax使用和交叉熵损失函数是等价的。RNN类中还有initHidden函数，其作用是对隐含单元进行初始化。

前向forward()函数传入的参数既有input，也有hidden表示上一时刻或者初始时刻的隐含单元输出值。其中x的尺寸是(length_seq,batch_size,

input_size），此时第一个维度大于1时RNN就会多步运行，output包含每一步运行的结果，hidden只包含最后一步运行隐含单元的输出。对output取[:,-1,:]代表获取最后一个时刻的隐含层输出作为这一层的输出

```
train_loss = 0

def trainRNN(epoch):
    global train_loss
    train_loss =0
    np.random.shuffle(train_set)

    for i,seq in enumerate(train_set):
        loss = 0
        hidden = rnn.initHidden()
        for t in range(len(seq)-1):
            x=Variable(torch.LongTensor([seq[t]]).unsqueeze(0))
            y =Variable(torch.LongTensor([seq[t+1]]))
            hidden = hidden[0] if isinstance(hidden, tuple) else hidden
            output, hidden = rnn(x, hidden)
            loss+=criterion(output,y)

        loss=1.0*loss/len(seq)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        train_loss+=loss
        if 1>0 and i%500 ==0:
            print('第{}轮，第{}个,训练Loss:{:.2f}'.format(epoch,
                                                    i,
                                                    train_loss.data.numpy()/(i+1)
                                                    ))
```

```
valid_loss = 0
errors = 0
show_out = ''

def evaluateRNN():
    global valid_loss
    global errors
    global show_out
    valid_loss = 0
    errors = 0
    show_out = ''
    for i ,seq in enumerate(valid_set):
        loss =0
        outstring = ''
        targets = ''
        diff = 0
        hidden = rnn.initHidden()
        for t in range(len(seq)-1):
            x = Variable(torch.LongTensor([seq[t]]).unsqueeze(0))
            y =Variable(torch.LongTensor([seq[t+1]]))
            hidden = hidden[0] if isinstance(hidden, tuple) else hidden
            output,hidden=rnn(x,hidden)
            mm = torch.max(output,1)[1][0]
            outstring +=str(mm.data.numpy())
            targets+=str(y.data.numpy()[0])
            loss+=criterion(output,y)
            diff +=1-mm.eq(y).data.numpy()[0]
        loss = 1.0*loss/len(seq)
        valid_loss+=loss
        errors+=diff
        if np.random.rand()<0.1:
            show_out= outstring+'\n'+targets
    print(output[0][2].data.numpy())
```

Python

4. 训练 RNN 模型

```python
num_epoch = 20
results = []
for epoch in range(num_epoch):
    trainRNN(epoch)
    evaluateRNN()
    print('第{}轮,训练Loss:{:.2f},错误率:{:.2f}'.format(epoch,
                                                    train_loss.data.numpy()/len(train_set),
                                                    valid_loss.data.numpy()/len(valid_set),
                                                    1.0*errors/len(valid_set)
                                                    ))
    print(show_out)
    results.append([train_loss.data.numpy()/len(train_set),
                    valid_loss.data.numpy()/len(train_set),
                    1.0*errors/len(valid_set)
                    ])
```

Python

进行RNN 模型的训练。在下面的训练代码中实际上进行了三重循环，Epoch 作为第一重循环，然后在trainRNN 中对每个train_set 中的字符串做第二重循环，最后是对每一个字符串中的每一个字符做循环。

5. 观察 RNN 模型的学习结果

```python
torch.save(rnn,'rnn.mdl')
rnn=torch.load('rnn.mdl')
```

Python

```python
for n in range(20):
    inputs = [0]*n+[1]*n
    inputs.insert(0,3)
    inputs.append(2)
    outstring = ''
    targets = ''
    diff =0
    hiddens = []
    hidden = rnn.initHidden()
    for t in range(len(inputs)-1):
        x = Variable(torch.LongTensor([inputs[t]]).unsqueeze(0))
        y =Variable(torch.LongTensor([inputs[t+1]]))
        hidden = hidden[0] if isinstance(hidden, tuple) else hidden
        output,hidden=rnn(x,hidden)
        mm = torch.max(output,1)[1][0]
        outstring +=str(mm.data.numpy())
        targets+=str(y.data.numpy()[0])

        diff +=1-mm.eq(y).data.numpy()[0]
    print(n)
    print(outstring)
    print(targets)
    print('Diff:{}'.format(diff))
```

Python

```
第0轮，第0个,训练Loss:1.53
第0轮，第500个,训练Loss:0.96
第0轮，第1000个,训练Loss:0.86
第0轮，第1500个,训练Loss:0.78
-1.2439084
第0轮,训练Loss:0.73,错误率:0.57
001112111
000011112
第1轮，第0个,训练Loss:0.71
第1轮，第500个,训练Loss:0.55
第1轮，第1000个,训练Loss:0.54
第1轮，第1500个,训练Loss:0.53
-0.9069356
第1轮,训练Loss:0.52,错误率:0.49
0000211
0001112
第2轮，第0个,训练Loss:0.54
第2轮，第500个,训练Loss:0.49
第2轮，第1000个,训练Loss:0.49
第2轮，第1500个,训练Loss:0.49
-0.76589227
第2轮,训练Loss:0.48,错误率:0.48
010002111
000011112
第3轮，第0个,训练Loss:0.59
...
-0.009064121
第49轮,训练Loss:0.25,错误率:0.24
01000011112
00000111112
```

```
0100000000011111111112
0000000000111111111112
Diff:2
12
01000000000011111111112
00000000000111111111112
Diff:2
13
010000000000011111111112
000000000000111111111112
Diff:2
14
0100000000000011111111111112
0000000000000111111111111112
Diff:2
15
01000000000000011111111111212
00000000000000111111111111112
Diff:3
16
010000000000000011111111111112
000000000000000111111111111112
Diff:2
17
0100000000000000011111111111111212
0000000000000000111111111111111112
Diff:3
18
01000000000000000011111111111111111
00000000000000000111111111111111112
Diff:3
19
010000000000000000011111111111111212
000000000000000000111111111111111112
Diff:3
```

起初训练轮次设置为20，训练没有达到理想效果，且每次训练的效果差异很大。于是将训练轮次改为50，训练正确率得到提高。但可以发现训练出来的模型也不是最理想的效果，经过多次尝试，训练出的结果如上图所示，但总是在第二个符号处出错，可能是陷入了某个局部最优解中。

对于RNN来说，通过观察学习结果可以发现当n比较小时RNN一般只在0变为1时犯错，在n=15时开始出现其他错误，因此可以得到结论该大小的RNN的记忆容量约为14。

6. 实现 LSTM 模型

```python
class SimpleLSTM(nn.Module):
    def __init__(self,input_size,hidden_size,output_size,num_layers=1):
        super(SimpleLSTM,self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.embedding = nn.Embedding(input_size,hidden_size)
        self.lstm = nn.LSTM(hidden_size,hidden_size,num_layers,batch_first=True)
        self.fc = nn.Linear(hidden_size,output_size)
        self.softmax = nn.LogSoftmax(dim = 1)

    def forward(self,input,hidden):
        x = self.embedding(input)
        output,hidden =self.lstm(x,hidden)
        output= output[:,-1,:]
        output =self.fc(output)
        output= self.softmax(output)
        return output,hidden

    def initHidden(self):
        hidden = Variable(torch.zeros(self.num_layers,1,self.hidden_size))
        cell = Variable(torch.zeros(self.num_layers,1,self.hidden_size))
        return (hidden,cell)
```

Python

```python
lstm = SimpleLSTM(input_size=4,hidden_size=2,num_layers=1,output_size=3)
criterion = torch.nn.NLLLoss()
optimizer = torch.optim.Adam(lstm.parameters(),lr = 0.001)
```

Python

```python
train_loss = 0

def trainLSTM(epoch):
    global train_loss
    train_loss =0
    np.random.shuffle(train_set)

    for i,seq in enumerate(train_set):
        loss = 0
        hidden = lstm.initHidden()
        for t in range(len(seq)-1):
            x=Variable(torch.LongTensor([seq[t]]).unsqueeze(0))
            y =Variable(torch.LongTensor([seq[t+1]]))
            output, hidden = lstm(x, hidden)
            loss+=criterion(output,y)
        loss=1.0*loss/len(seq)
        optimizer.zero_grad()
        loss.backward(retain_graph=True)
        optimizer.step()
        train_loss+=loss
        if 1>0 and i%500 ==0:
            print('第{}轮，第{}个,训练Loss:{:.2f}'.format(epoch,
                                                    i,
                                                    train_loss.data.numpy()/(i+1)
                                                    ))
```

Python

```
valid_loss = 0
errors = 0
show_out = ''

def evaluateLSTM():
    global valid_loss
    global errors
    global show_out
    valid_loss = 0
    errors = 0
    show_out = ''
    for i ,seq in enumerate(valid_set):
        loss =0
        outstring = ''
        targets = ''
        diff = 0
        hidden = lstm.initHidden()
        for t in range(len(seq)-1):
            x = Variable(torch.LongTensor([seq[t]]).unsqueeze(0))
            y =Variable(torch.LongTensor([seq[t+1]]))
            output,hidden=lstm(x,hidden)
            mm = torch.max(output,1)[1][0]
            outstring +=str(mm.data.numpy())
            targets+=str(y.data.numpy()[0])
            loss+=criterion(output,y)
            diff +=1-mm.eq(y).data.numpy()[0]
        loss = 1.0*loss/len(seq)
        valid_loss+=loss
        errors+=diff
        if np.random.rand()<0.1:
            show_out= outstring+'\n'+targets
    print(output[0][2].data.numpy())
```

```
num_epoch = 20
results = []
for epoch in range(num_epoch):
    trainLSTM(epoch)

    evaluateLSTM()
    print('第{}轮,训练Loss:{:.2f},错误率:{:.2f}'.format(epoch,
                                                    train_loss.data.numpy()/len(train_set),
                                                    valid_loss.data.numpy()/len(valid_set),
                                                    1.0*errors/len(valid_set)
                                                    ))
    print(show_out)
    results.append([train_loss.data.numpy()/len(train_set),
                    valid_loss.data.numpy()/len(train_set),
                    1.0*errors/len(valid_set)
                    ])
```

Python

```python
torch.save(lstm,'lstm.mdl')
lstm=torch.load('lstm.mdl')
```

Python

```python
for n in range(20):
    inputs = [0]*n+[1]*n
    inputs.insert(0,3)
    inputs.append(2)
    outstring = ''
    targets = ''
    diff =0
    hiddens = []
    hidden = lstm.initHidden()
    for t in range(len(inputs)-1):
        x = Variable(torch.LongTensor([inputs[t]]).unsqueeze(0))
        y =Variable(torch.LongTensor([inputs[t+1]]))
        output,hidden=lstm(x,hidden)
        mm = torch.max(output,1)[1][0]
        outstring +=str(mm.data.numpy())
        targets+=str(y.data.numpy()[0])

        diff +=1-mm.eq(y).data.numpy()[0]
    print(n)
    print(outstring)
    print(targets)
    print('Diff:{}'.format(diff))
```

Python

```
第0轮，第0个,训练Loss:1.34
第0轮，第500个,训练Loss:0.97
第0轮，第1000个,训练Loss:0.91
第0轮，第1500个,训练Loss:0.86
-0.8512011
第0轮,训练Loss:0.80,错误率:0.59
00111111111111102
00000000111111112
第1轮，第0个,训练Loss:0.66
第1轮，第500个,训练Loss:0.54
第1轮，第1000个,训练Loss:0.50
第1轮，第1500个,训练Loss:0.47
-0.22808187
第1轮,训练Loss:0.45,错误率:0.37
002
012
第2轮，第0个,训练Loss:0.38
第2轮，第500个,训练Loss:0.36
第2轮，第1000个,训练Loss:0.35
第2轮，第1500个,训练Loss:0.34
-0.107153565
第2轮,训练Loss:0.33,错误率:0.30
002
012
第3轮，第0个,训练Loss:0.33
...
-0.001335206
第19轮,训练Loss:0.24,错误率:0.23
01012
00112
0
0
2
Diff:1
1
012
012
Diff:0
2
01012
00112
Diff:2
3
0100112
0001112
Diff:2
4
010001112
000011112
Diff:2
5
01000011112
00000111112
Diff:2
6
...
19
01000000000000000000111111111111111111112
00000000000000000000111111111111111111112
Diff:2
```

LSTM 模型的代码与RNN 几乎相同，只有在初始化隐藏层状态的时候，LSTM除了初始化隐藏层的状态，还初始化了隐含层内部细胞的状态，也就是各个“门控单元”的状态。通过观察相同尺寸的RNN和LSTM模型的训练效果，LSTM具有更长的记忆能力。通过观察结果LSTM模型即使n达到了19依然能够保持较好的错误率，验证了LSTM具有更长的记忆能力。而LSTM只训练了20个轮次。

三、　心得体会

本次实验通过生成了简单的序列练习使用了 RNN 和 LSTM。两者间进行对比，两者的基本代码相同，LSTM 具有更长的记忆能力。RNN 结构在前馈神经网络的基础上令隐层的输出和前一次的输出值相关，增加了隐层内部的连接赋予了 RNN 一定的记忆能力。

LSTM 相比 RNN 增加了几个内部的门结构，通过控制门结构使遗忘门被输入信号控制，加入内部储存提高了 LSTM 的长期记忆能力。

实验过程中 RNN 训练函数中出现报错，rnn 中不能传入元组类型的变量，变量的大小不匹配，有关 hidden 的维度出现错误，经过如下的判断语句后再输入到模型中问题解决。

```
hidden = hidden[0] if isinstance(hidden, tuple) else hidden
```

改正：通过复习过程中仔细阅读代码，读懂每一句的作用和含义帮助我更好地记忆各种网络模型地结构，也在这个过程中找到了先前没有发现的错误。RNN 和 LSTM 两个模型代码的主要区别在于 initHidden 函数中初始化的隐层不同，在实验过程中没有阅读教材，而是按照实验指导书中有关 LSTM 的代码直接修改，两者使用了相同的类定义代码，但 RNN 中没有 cell 结构，因此 initHidden 的返回值 hidden 作为参数传入 rnn 中时出现报错，删除 RNN 中对 cell 的初始化和返回值后就不需要上述判断语句了。