课程编号： A3705060010

自然语言处理

姓　　名	王硕	学　　号	20182338
班　　级	软信 1802	学　　院	软件学院
项 目 名 称	搭建 LSTM 语言模型		
开 设 学 期	2021-2022 秋季学期		
开 设 时 间	第 1 周——第 8 周		
报 告 日 期	2021 年 11 月 13 日		
仓 库 地 址	https://github.com/Aa-bN/NLP_work		
报 告 内 容	1.设计思路 2.代码说明 3.模型测试 4.问题总结		
评 定 成 绩		评　定　人	肖桐
		评 定 日 期	

1. 设计思路

该部分为构建模型的具体思路，分为三部分：模型功能，数据形状，核心算法。

1.1 模型功能

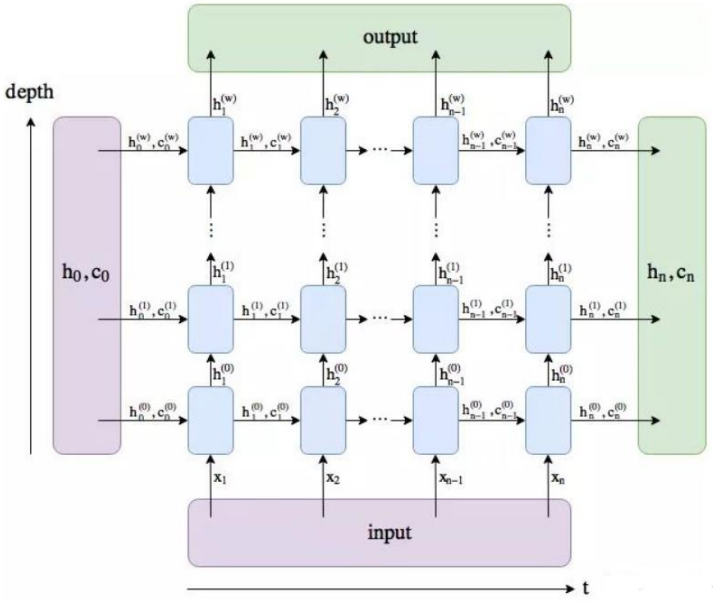
① 完成基本的 LSTM 过程；

② 可以选择 LSTM 的层数，通过传入参数，实现 1 层或多层 LSTM 的使用；

③ 可以选择性地传入 hidden state、cell state、batch first 等参数；

④ 输出每层最终的 hidden state，cell state 和 outputs。

可用下图表示：



图源：https://zhuanlan.zhihu.com/p/79064602

1.2 数据形状

虽然项目只要求完成模型的搭建过程，但是对整个程序中数据类型或形状的分析，有助于更好地了解 LSTM 模型的输入和输出，也有利于为我们自己的函数提供良好的接口。通过对示例程序的 debug，我提取并总结了程序中比较重要的变量，以及它们的形状。如下图：

```
Main:
n_class = 7615      emb_size = 256
n_hidden = 128      n_step = 5          batch_size = 128

all_input_batch     torch.Size([603, 128, 5])     T Tensor 5      data Tensor 603
all_target_batch    torch.Size([603, 128])        T Tensor 128    data Tensor 603

-----------------------------------------------------------------------------------------------------------------------------------------------
Model:

__init__:
        C = Embedding(7615, 256)
        LSTM = LSTM(256, 128)
        W = Linear(128, 7615)
        b = Parameter(torch.ones([n_class]))
forward:
        (self, X)                       X       torch.Size([128, 5])        T 5     data 128        len(X) 128
        X = C(X)                        X       torch.Size([128, 5, 256])   T 256   data 128        len(X) 128
        X = X.transpose(0, 1)           X       torch.Size([5, 128, 256])   T 256   data 5          len(X) 5
        outputs, (_, _) = LSTM(X)       output  torch.Size([5, 128, 128])   T 128   data 5
                                        _       torch.Size([1, 128, 128])   T 128   data 1
        outputs = outputs[-1]           output  torch.Size([128, 128])      T 128   data 128
        model = W(outputs) + b          model   torch.Size([128, 7615])     T 7615  data 128
        return model

-----------------------------------------------------------------------------------------------------------------------------------------------
Train:
        ......
        for input_batch, target_batch in zip(all_input_batch, all_target_batch):        input_batch     torch.Size([128, 5])
                ......                                                                  target_batch    torch.Size([128])
                output = model(input_batch)                                             output          torch.Size([128, 7615])
                loss = crit(output, target_batch)
                ......
```

图源：项目过程中的分析记录

1.3 核心算法

查阅官方文档，可以得到 LSTM 的核心算法：

CLASS `torch.nn.LSTM(*args, **kwargs)` [SOURCE]

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

where h_t is the hidden state at time t, c_t is the cell state at time t, x_t is the input at time t, h_{t-1} is the hidden state of the

图源：pytorch 官方文档

在实现时，可以把同一个表达式中的两个偏移量合并成一个。并且需要提前在 init()函数中定义好权重和偏移量。在这些算法中，用到了两种不同的 tensor 数据类型的乘法，在实现过程中，需要加以区分。

2. 代码说明

模型在 myLSTM.py 的 72-227 行，下面**结合源码注释**进行代码说明。

2.1 首先来看 init()函数：(73-106 行)

```
73      def __init__(self):
74          super(TextLSTM, self).__init__()
75          self.C = nn.Embedding(n_class, embedding_dim=emb_size)
76          # self.LSTM = nn.LSTM(input_size=emb_size, hidden_size=n_hidden)
77          self.W = nn.Linear(n_hidden, n_class, bias=False)
78          self.b = nn.Parameter(torch.ones([n_class]))
79
80          # 定义LSTM中的需要的参数，参考pytorch官方文档中的实现方式
81          # 同一个表达式中的两个bias合并为一个
82          # 遗忘门 ft
83          self.Wif = nn.Parameter(torch.Tensor(emb_size, n_hidden))
84          self.Whf = nn.Parameter(torch.Tensor(n_hidden, n_hidden))
85          self.bf = nn.Parameter(torch.Tensor(batch_size, n_hidden))
86          # 输入门 it
87          self.Wii = nn.Parameter(torch.Tensor(emb_size, n_hidden))
88          self.Whi = nn.Parameter(torch.Tensor(n_hidden, n_hidden))
89          self.bi = nn.Parameter(torch.Tensor(batch_size, n_hidden))
90          # 输入门 gt
91          self.Wig = nn.Parameter(torch.Tensor(emb_size, n_hidden))
92          self.Whg = nn.Parameter(torch.Tensor(n_hidden, n_hidden))
93          self.bg = nn.Parameter(torch.Tensor(batch_size, n_hidden))
94          # 输出门 Ot
95          self.Wio = nn.Parameter(torch.Tensor(emb_size, n_hidden))
96          self.Who = nn.Parameter(torch.Tensor(n_hidden, n_hidden))
97          self.bo = nn.Parameter(torch.Tensor(batch_size, n_hidden))
98          # 激活函数
99          self.sig = nn.Sigmoid()
100         self.tah = nn.Tanh()
```

```
101
102         # 第二层LSTM及以后层，需要使用的矩阵变量
103         self.Wif2 = nn.Parameter(torch.Tensor(batch_size, n_hidden))    # 遗忘门
104         self.Wii2 = nn.Parameter(torch.Tensor(batch_size, n_hidden))    # 输入门
105         self.Wig2 = nn.Parameter(torch.Tensor(batch_size, n_hidden))    # 输入门
106         self.Wio2 = nn.Parameter(torch.Tensor(batch_size, n_hidden))    # 输出门
107
```

在 75-78 行，定义了嵌入层、线性层和一个偏移量（不用于 LSTM 内部)。

在 80-100 行，定义了 LSTM 内部的权重与偏移量，激活函数。

在 102-106 行，定义了第二层及以后层的 LSTM，需要使用的权重和偏移量。

这些变量的 size 可以根据 1.3 核心算法推导出来。

2.2 再来看我们建立的 myLSTM 函数：（122-227 行）

第 122-143 行

```
122    def myLSTM(self, X, hidden_size, num_layers=1, hidden_state=None, cell_state=None, batch_first=False):
123        # input_size = emb_size
124        # hidden_size = n_hidden
125        # input_size没有作为函数参数，因为可由数据X得到：input_size = X.shape[2]
126
127        # 初始化 hidden_state 和 cell_state
128        # 首先，通过 batch_first 参数，默认False。
129        # False 确定 X 的 shape 为 (L, N, Hin) 即 (sequence_length, batch_size, input_size)
130        # True 确定 X 的 shape 为 (N, L, Hin)
131        # 获取X的 batch_size, 最终将X转化为True的形式 (batch_size = 128, sequence_length = 5, input_size = 256)
132        # 每次传入128个句子，每个句子的长度/单词数量为5，每个单词向量的长度为256
133        if batch_first is True:     # X [128, 5, 256]
134            lenX = len(X)
135        if batch_first is False:    # X [5, 128, 256]
136            X = X.transpose(0, 1)   # X [128, 5, 256]
137            lenX = len(X)           # batch_size 128
138
139        # 确定h和c
140        if hidden_state is None:
141            hidden_state = torch.zeros(num_layers, lenX, hidden_size).to(device)    # 只考虑单向LSTM，故num_directions = 1
142        if cell_state is None:
143            cell_state = torch.zeros(num_layers, lenX, hidden_size).to(device)      # [1, 128, 128]
```

第 122 行，可以得到函数的六个参数。

在写 myLSTM 函数的过程中，参考了 pytorch 的官方文档，确定了函数的参数，输入数据的形状，以及函数的返回值。具体如下列表：

表 2-1 参数列表

参数名称	参数说明	形状
X	数据	[sequence length, batch size, input size] 或者 [batch size, sequence length, input size]
hidden_size	隐含层大小	int（256）
num_layers	LSTM 的层数	int（≥1，default=1）
hidden_state	隐层状态	[num layers, batch size, hidden size]
cell_state	细胞状态	[num layers, batch size, hidden size]
batch_first	batch 是否在首位	bool，若为 False，则 X 为第一种形状；若为 True，则 X 为第二种形状

第 123-132 行，注释对数据 X 的 shape 和参数 batch_first 进行了详细的说明。

第 133-137 行，通过对 batch_first 参数的判断，调整 X 的 shape，并获取 batch size，即程序中的 len(X)。

第 139-143 行，先判断有无初始的 hidden_state 和 cell_state，若无，则创建一个默认值。

第 145-157 行

```
145         # 后续操作的总体思路，LSTM cell -> RNN -> LSTM，循环 LSTM 实现多层的LSTM
146
147         # 生成xt
148         # 首先获取数据的 sequence_length，由于已经把数据转换成batch_first，即(N, L, Hin)的形式
149         # 所以第二个参数 L 即为sequence_length，也就是我们主程序中的n_step
150         sequence_length = X.shape[1]    # 5
151
152         # 存储每层LSTM最后一个cell生成的 hidden_state 和 cell_state，即每层的hn，ctn
153         hidden_state_layer_final = []
154         cell_state_layer_final = []
155
156         # 存储每层LSTM生成的 hidden_state，即[h1， h2，... ，hn]， 用来向下一层传递
157         hidden_state_one_layer = []
158
```

第 150 行，得到了序列长度，也可以直接使用主函数中的 n_step。

第 153-154 行，两列表分别存储每层 LSTM 最后一个 cell 生成的 hidden_state 和 cell_state。

第 157 行，存储每层 LSTM 生成的 hidden_state，即[h1， h2，... ，hn]， 用来向下一层传递。

第 161-190 行

该部分代码，构建了第一层 LSTM。同时，完成了核心算法的执行和相关数据的存储，为权重的传播做准备。在其中（第 165-186 行），以 for 循环的形式，完成了 LSTM cell 的实现。

```
161         # 第一层 LSTM  (index=0)
162         # 取出第一个值，作为第一层LSTM中，h0和c0的初始值
163         h0 = hidden_state[0]    # [128, 128]    [batch_size, hidden_size]
164         c0 = cell_state[0]      # [128, 128]
165         for i in range(sequence_length):
166             # batch_size个句子，每个句子的第i个单词的词向量
167             xt = X[:, i, :]             # [128, 256]    [batch_size, input_size]
168 
169             # LSTM cell
170             # 一下变量的格式 [batch_size, hidden_size] [128, 128]
171             # 遗忘门
172             ft = self.sig(xt @ self.Wif + h0 @ self.Whf + self.bf)
173             # ft = torch.sigmoid(xt @ self.Wif + h0 @ self.Whf + self.bf)
174             # ft = nn.functional.sigmoid(xt @ self.Wif + h0 @ self.Whf + self.bf)
175             # 输入门
176             it = self.sig(xt @ self.Wii + h0 @ self.Whi + self.bi)
177             gt = self.tah(xt @ self.Wig + h0 @ self.Whg + self.bg)
178             # 输出门
179             ot = self.sig(xt @ self.Wio + h0 @ self.Who + self.bo)
180             # 记忆更新  这里的乘法是 Hadamard Product
181             ct = ft * c0 + it * gt
182             # 输出门   这里的乘法是 Hadamard Product
183             ht = ot * self.tah(ct)
184 
185             # 保存本层LSTM产生的hidden_state，用以传入下一层
186             hidden_state_one_layer.append(ht)       # [128, 128]，列表中最终会有sequence_length=5个这样的tensor
187 
188         # 到这里，第一层LSTM结束了，保存该层最终的hn和cn
189         hidden_state_layer_final.append(ht.unsqueeze(0))    # [1, 128, 128]     [num_layers, batch_size, hidden_size]
190         cell_state_layer_final.append(ct.unsqueeze(0))      # [1, 128, 128]
191 
```

第 163-164 行，获取了第一层 LSTM，h0 和 c0 的初始值。

第 165 行，以 for 循环的形式实现 LSTM cell。

第 167 行，完成数据的变形，即 batch_size 个句子，分别取每个句子的第 i 个单词的词向量，带入 LSTM cell 中。

第 169-183 行，完成了三个门的实现和记忆更新。

第 186 行，保存本层 LSTM 中，每一个 LSTM cell 产生的 hidden_state，用以传入下一层。

第 189-190 行，第一层的 LSTM 结束，将最终的 hn 和 cn 添加到列表，其 shape 由[128, 128]变为[1, 128, 128]。

第 192-215 行

判断 num_layers 是否大于等于 2，若是，则意味着用户指定了多层 LSTM，第二层及以后层的 LSTM 在这段代码中实现。代码思路与前面基本一致。

第 217-227 行

```
217         # 到这里，所有的层的LSTM均执行完毕，此时hidden_state_one_layer为最后一层的输出，len=5
218         outputs_temp = []
219         for i in range(len(hidden_state_one_layer)):
220             last_layer_output_each_cell = hidden_state_one_layer[i].unsqueeze(0)    # [128, 128] -> [1, 128, 128]
221             outputs_temp.append(last_layer_output_each_cell)                        # 最终中有5个tensor，形状为[1,128,128]
222 
223         outputs_return = torch.cat(outputs_temp, dim=0)                   # [sequence_length, batch_size, hidden_size]
224         hidden_state_return = torch.cat(hidden_state_layer_final, dim=0)    # [num_layers, batch_size, hidden_size]
225         cell_state_return = torch.cat(cell_state_layer_final, dim=0)        # [num_layers, batch_size, hidden_size]
226 
227         return outputs_return, hidden_state_return, cell_state_return
```

到这里，所有的 LSTM 层执行完毕，返回 outputs，shape 为[sequence_length, batch_size, hidden_size]；返回 hidden_state 和 cell_state，shape 均为[num_layers, batch_size, hidden_size]。

2.3 最后看 forward()函数：（108-120 行）

```
def forward(self, X):
    X = self.C(X)       # [128, 5, 256]
    # 确定参数
    num_layers = 2
    hidden_state = torch.rand(num_layers, batch_size, n_hidden).to(device)
    cell_state = torch.rand(num_layers, batch_size, n_hidden).to(device)

    outputs, ht, ct = self.myLSTM(X=X, hidden_size=n_hidden, num_layers=num_layers, hidden_state=hidden_state,
                                  cell_state=cell_state, batch_first=True)
    # outputs, ht, ct = self.myLSTM(X=X, hidden_size=n_hidden, num_layers=num_layers, batch_first=True)
    outputs = outputs[-1]
    model = self.W(outputs) + self.b  # model : [batch_size, n_class]
    return model
```

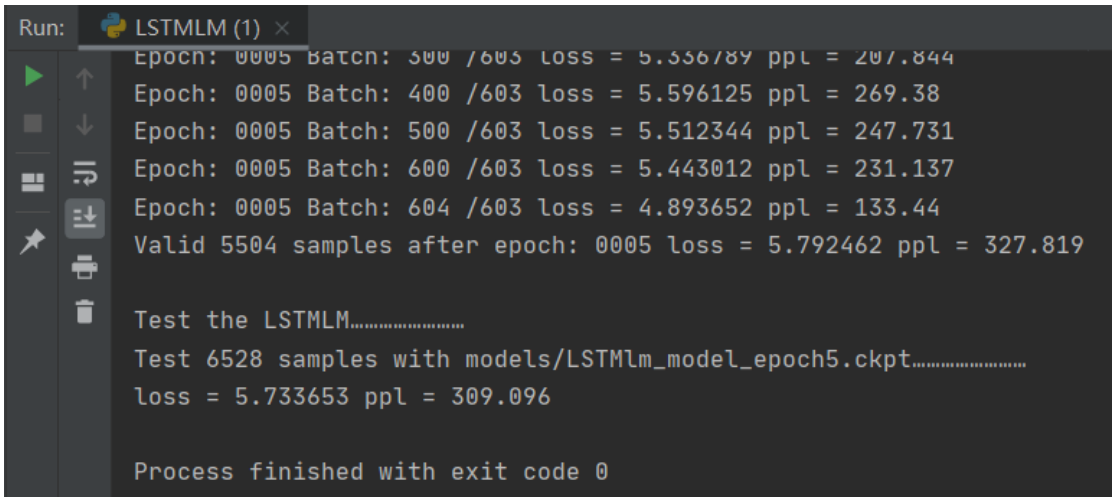
这部分主要完成了模型的前向传播过程，其中 num_layers, hidden_state, cell_state, batch_first 等参数，都是用户可选择的。

在最后，调整了数据的形状，从 outputs 中获取了我们需要的 ht，经过后续线性层，完成了模型。

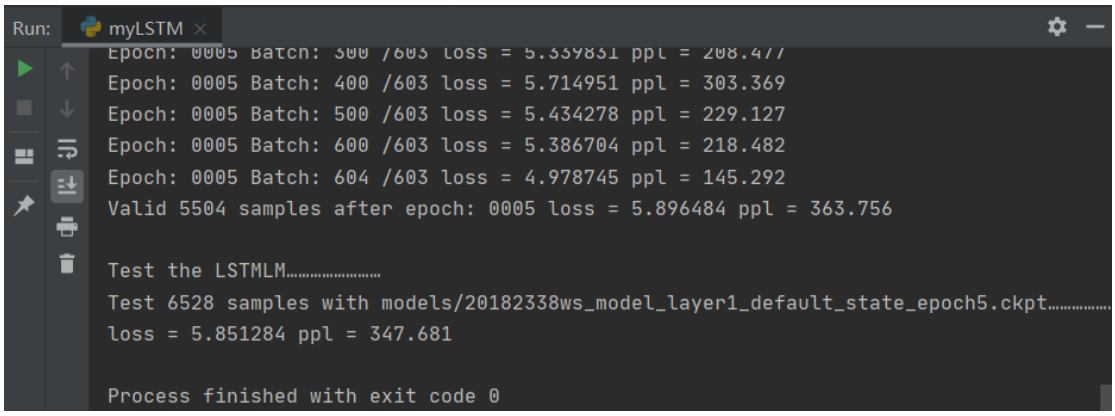
这里，我们的模型不仅完成了单层和双层的 LSTM 网络，理论上也可以实现任意 n(n≥1)层的 LSTM 网络。

3. 模型测试

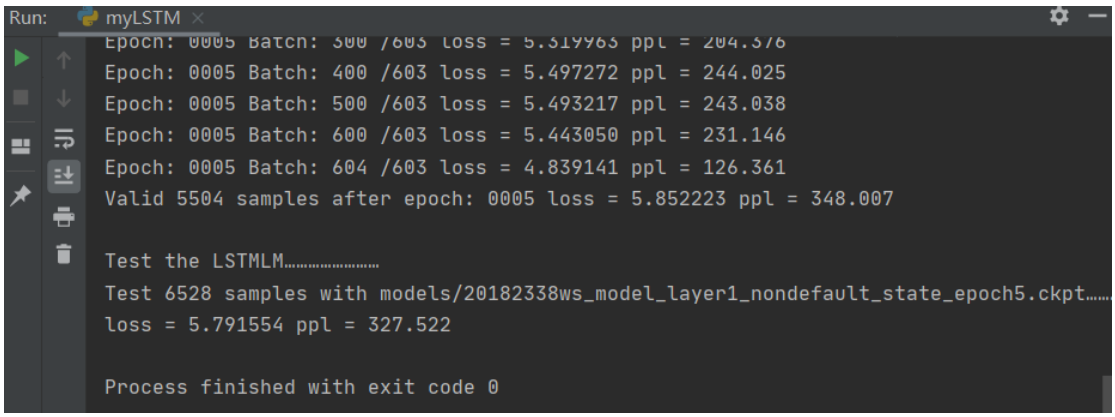
① 示例程序运行结果（层数=1， 默认 hidden_state, cell_state）:



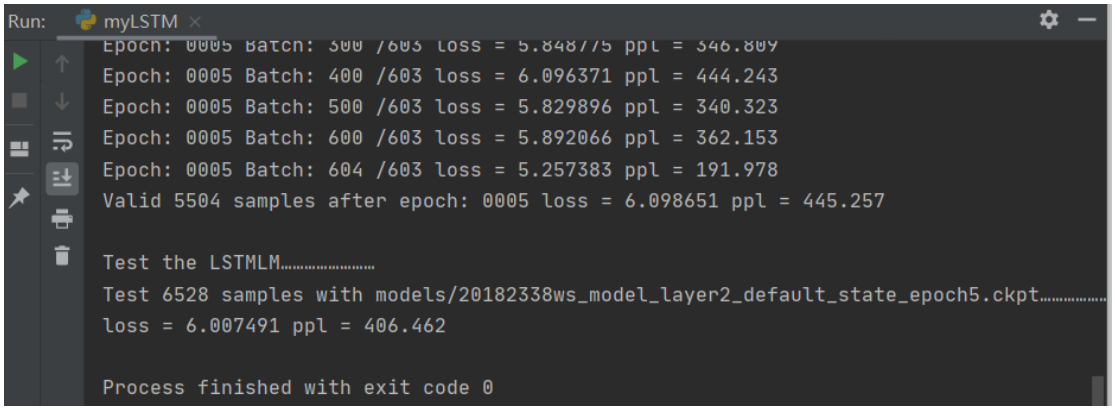
② 项目模型（层数=1，默认 hidden_state, cell_state）：



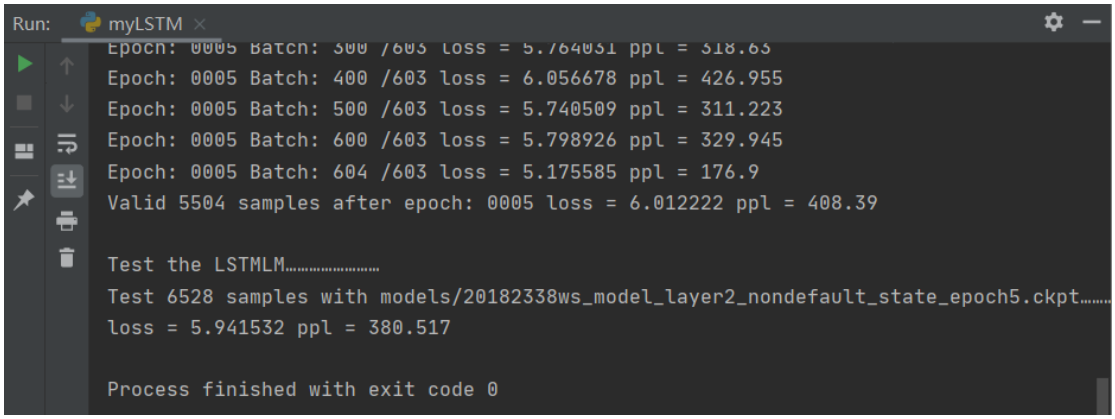
③ 项目模型（层数=1，随机 hidden_state, cell_state）：



④ 项目模型（层数=2， 默认 hidden_state, cell_state）：



⑤ 项目模型（层数=2， 随机 hidden_state, cell_state）：



⑥ 项目模型（层数=3，默认 hidden_state, cell_state）：略

⑦ 项目模型（层数=3，随机 hidden_state, cell_state）：略

由于代码底层原因，使用项目模型时，当层数≥3 时，计算速度过慢。我的 GPU 好像已经不允许我这么测试了。最终结果如下表：

表 3-1 测试结果

序号	模型	LSTM 层数	hidden\cell state	loss	ppl
1	示例模型	1	default	5.733653	309.096
2	项目模型	1	default	5.851284	347.681
3	项目模型	1	random	5.791554	327.522
4	项目模型	2	default	6.007491	406.462
5	项目模型	2	random	5.941532	380.517

4. 问题总结

Q1：注释中有的 tensor 数据的 shape 标注的不太准确，如示例程序 LSTMLM.py 的第 106 行。

Q2：如果使用 GPU 的话，构建或使用模型过程中，产生的新 tensor 要及时加载到 GPU 中，否则会报错。

Q3：自己构建的 LSTM 函数的接口不是很完善。

Q4：没有对传入的参数进行检查，存在安全性问题。

Q5：在程序执行的过程中，有可能产生 loss 和 ppl 为 nan 的问题。原因或解决方案：① 减小学习率；② 数据归一化；③ 加入 gradient clipping；④ 数据出了问题。（为了与示例程序做对比，没有加入上述操作。）

Q6：由于底层原因，随着层数增多，运行速度变慢。