2021.11

自然语言处理

实验报告

黄元通

20195063

目录

基础内容 搭建 LSTM 网络

一、题目

尝试自己搭建 LSTM 网络

a) 不能调用 nn.LSTM、nn.LSTMCell，可以使用 nn.Linear、nn.Parameter 等等搭建网络

b) 可以参考 torch.nn.LSTM 的计算公式、可以仿照其输入输出，官方文档：https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html#torch.nn.LSTM

c) 数据加载、模型训练的代码都是现成的，只需要完成模型搭建

二、网络结构设计

1、计算公式

参考 torch.nn.LSTM，其计算公式如下：

$$f_t = \sigma\left(W_{if}X_t + b_{if} + W_{hf}h_{t-1} + b_{hf}\right)$$

$$i_t = \sigma(W_{ii}X_t + b_{ii} + W_{hi}h_{t-1} + b_{hi})$$

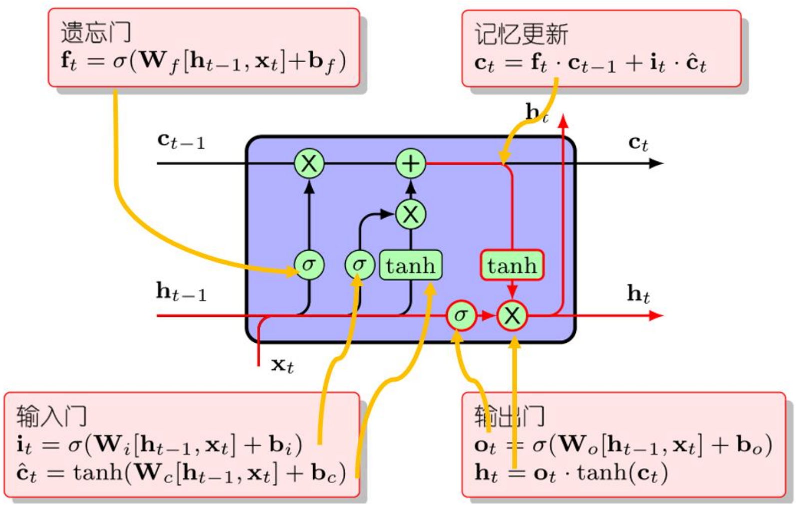
$$g_t = tanh\left(W_{ig}X_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}\right)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

$$o_t = \sigma(W_{io}X_t + b_{io} + W_{ho}h_{t-1} + b_{ho})$$

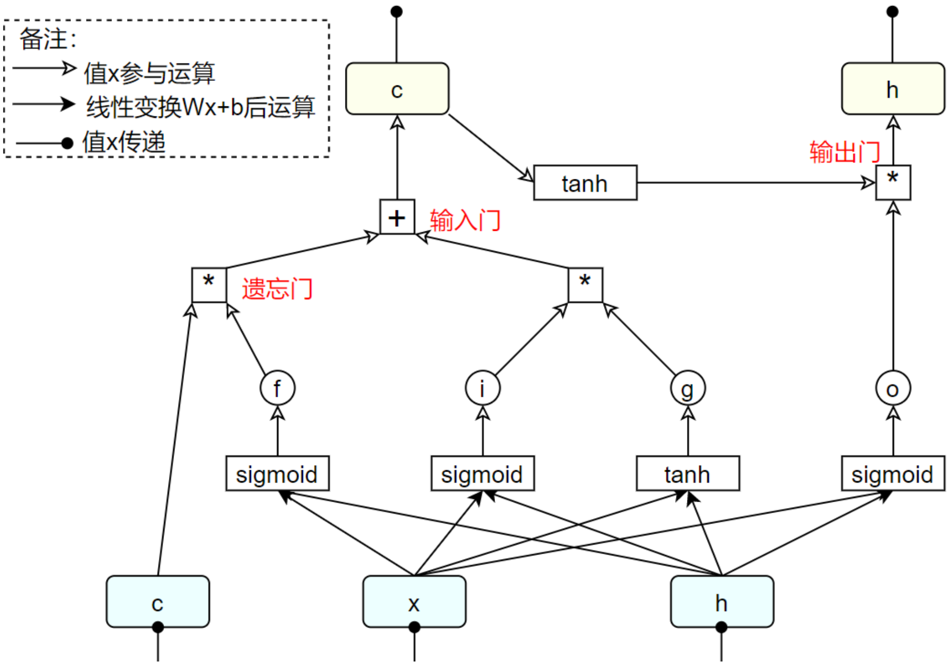
$$h_t = o_t \odot tanh(c_t)$$

其中，f_t为遗忘门，i_t（新记忆占比）和g_t（新记忆）共同组成输入门，o_t为输出门，X_t为输入（或上一层输出），c_t为记忆。其方程与 LSTM 结构图对应如下：



2、程序实现

通过 torch.nn 中的 Linear、Sigmoid、Tanh 函数实现上述功能，则其网络结构图如下：



因此网络非线性，无法使用 nn.Sequential( )进行封装，因此选择在自实现类 LSTM 时，通过在 forward( )函数中手动指定计算流程。

所有运算X_t的线性层：输入输出维度分别为：input_size、hidden_size；
所有运算h_{t-1}的线性层：输入输出维度都别为：hidden_size。类的代码实现如下：

a) 在__init__函数中添加如下图所示的函数组件：

```python
# 遗忘门（旧记忆的占比）
# f
self.linear_if = nn.Linear(input_size, hidden_size)
self.linear_hf = nn.Linear(hidden_size, hidden_size)
self.sigmoid_f = nn.Sigmoid()
# 输入门（新的记忆）
# i
self.linear_ii = nn.Linear(input_size, hidden_size)
self.linear_hi = nn.Linear(hidden_size, hidden_size)
self.sigmoid_i = nn.Sigmoid()
# g
self.linear_ig = nn.Linear(input_size, hidden_size)
self.linear_hg = nn.Linear(hidden_size, hidden_size)
self.tanh_g = nn.Tanh()
# 输出门（由新记忆构造输出）
# o
self.linear_io = nn.Linear(input_size, hidden_size)
self.linear_ho = nn.Linear(hidden_size, hidden_size)
self.sigmoid_o = nn.Sigmoid()
self.tanh_o = nn.Tanh()
```

b) 在 forward( )中，计算过程如下图所示：

```
# 遗忘门（旧记忆的占比）
f = self.sigmoid_f(self.linear_if(x_i) + self.linear_hf(hidden_state_layer))
# 输入门（新的记忆）
i = self.sigmoid_i(self.linear_ii(x_i) + self.linear_hi(hidden_state_layer))
g = self.tanh_g(self.linear_ig(x_i) + self.linear_hg(hidden_state_layer))
cell_state_layer = f * cell_state_layer + i * g
# 输出门
o = self.sigmoid_o(self.linear_io(x_i) + self.linear_ho(hidden_state_layer))
hidden_state_layer = o * self.tanh_o(cell_state_layer)
```

三、传入、传出及其他细节

1、输入参数及维度

在主程序（LSTMLM.py）中，通过以下语句进行 LSTM 调用：

```
outputs, (_, _) = self.LSTM(X, (hidden_state.to(device), cell_state.to(device)))
```

1. 参数含义：
   a) X 为输入数据
   b) hidden_state 为外界定义的 LSTM 胞体初始化 h 值
   c) cell_state 为外界定义的 LSTM 胞体初始化 c 值
2. 各参数的 shape，及在本实验中的值：
   a) X：[n_step=5, batch_size=128, embeding size=256]
   b) hidden_state：[num_layers=1 * num_directions=1, batch_size=128, n_hidden=256]
   c) cell_state：[num_layers=1 * num_directions=1, batch_size=128, n_hidden=256]
3. 各维度含义：
   a) n_step 一层 LSTM 的循环个数，即每次使用的句子长度（单词个数为 5）
   b) embeding size 为外界定义的 LSTM 胞体中间参数的维度，即 LSTM 的 hidden_size
   c) batch_size 为批训练时，一次所用数据个数
   d) num_layers 为 LSTM 层数
   e) num_directions 标识 LSTM 方向，等于 1 时为单向，等于 2 时为双向
4. 数据类型分别为(tensor, tuple(tensor, tensor))，因此，在自实现的 LSTM 类中，其 forward( )函数进行如下处理：

```
hidden_state = state[0]
cell_state = state[1]

for layer in range(len(hidden_state)):
    hidden_state_layer = hidden_state[layer]
    cell_state_layer = cell_state[layer]
    for i in range(len(X)):
        x_i = X[i]
        # 遗忘门（旧记忆的占比）
```

2、运行设备统一

因为在主程序中可能会将网络及各参数运行设备设置为 cpu（即 CPU）或 cuda:0（即 GPU），因此自实现的该 LSTM 网络也应该运行在相同的设备上。

1. pytorch 的选取原则：

首先使用 nn.LSTM 进行测试。

因为在主程序中 hidden_state、cell_state 的运行设备为 CPU，而 X 的运行设备为 GPU（实验所用电脑为支持 cuda 运行的环境，代表更广泛的情景），通过测试发现，若使用如下语句进行运行会发出设备不统一的报错。

```
outputs, (_, _) = self.LSTM(X, (hidden_state, cell_state))
```

也即 pytorch 官方默认使用 X、hidden_state、cell_state 他们各自传入时的设备。

2. 自实现的选取原则：

但考虑到 X、hidden_state、cell_state 若有运行设备不统一的情况无法运行，这样需要编程人员手动显式地在调用前将设备统一。

因此为了降低编程人员的负担，在自实现 LSTM 时决定在胞体内将所有变量都设置到 X 所在的设备上。

forward( )函数更新为下：

```python
def forward(self, X, state: tuple):
    # 根据输入X，确定所运行设备
    device = X.device()
    hidden_state = state[0].to(device)
    cell_state = state[1].to(device)
    outputs_h = torch.zeros(hidden_state.size()).to(device)
    outputs_c = torch.zeros(cell_state.size()).to(device)
    # # 但nn中的LSTM会分别使用X、state的设备，不进行统一
    # hidden_state = state[0]
    # cell_state = state[1]
    # outputs_h = hidden_state.clone()
    # outputs_c = cell_state.clone()
```

3、输出参数及维度

1. 参数含义：

参考 pytorch 官网给出的 LSTM 网络的输出：

Outputs: output, (h_n, c_n)

- **output**: tensor of shape $(L, N, D * H_{out})$ when `batch_first=False` or $(N, L, D * H_{out})$ when `batch_first=True` containing the output features (*h_t*) from the last layer of the LSTM, for each *t*. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence.
- **h_n**: tensor of shape $(D * \text{num\_layers}, N, H_{out})$ containing the final hidden state for each element in the batch.
- **c_n**: tensor of shape $(D * \text{num\_layers}, N, H_{cell})$ containing the final cell state for each element in the batch.

即：

a) output 保存了最后一层，每个 time step 的输出 h，如果是双向 LSTM，每个 time step 的输出 h = [h 正向, h 逆向] (同一个 time step 的正向和逆向的 h 连接起来)。第一维表示序列长度，第二维表示一批的样本数(batch)，第三维是 hidden_size(隐藏层大小) * num_directions。
b) h_n 保存了每一层，最后一个 time step 的输出 h，如果是双向 LSTM，单独保存前向和后向的最后一个 time step 的输出 h。
c) c_n 与 h_n 一致，只是它保存的是 c 的值。

2. 在自定义的 LSTM 结点实现时，使用 outputs、h_n、c_n 分别存储上述参数 output、h_n、c_n。n_step、num_layers、 batch_size、hidden_size 值获取过程，以及 outputs、h_n、c_n 定义如下更新后代码所示：

```
def forward(self, X, state: tuple):
    # 根据输入X，确定所运行设备
    device = X.device
    hidden_state = state[0].to(device)
    cell_state = state[1].to(device)

    # 获取并设置各维长度
    n_step = len(X)
    [num_layers, batch_size, hidden_size] = hidden_state.shape
    outputs = torch.zeros((n_step, batch_size, hidden_size)).to(device)  # 最后一层所有结点的输出
    h_n = torch.zeros((num_layers, batch_size, hidden_size)).to(device)  # 每一层最后一个结点的输出
    c_n = torch.zeros((num_layers, batch_size, hidden_size)).to(device)  # 每一层最后一个结点的记忆
```

3. outputs、h_n、c_n 值的获取，如下更新后代码所示：

```
# 第layer层
for layer in range(num_layers):
    hidden_state_layer = hidden_state[layer]
    cell_state_layer = cell_state[layer]
    # 一层共n_step个结点（句子长度）
    for step in range(n_step):
        x_i = X[step]
        # 遗忘门（旧记忆的占比）


        # 在最后一层时，需要将所有n_step个结点，值保存在outputs中
        if layer == (num_layers - 1):
            outputs[step] = hidden_state_layer
    # 在每一层的最后一个结点，值保存在outputs_h、outputs_c中
    h_n[layer] = hidden_state_layer
    c_n[layer] = cell_state_layer
return outputs, (h_n, c_n)
```

4、未传 state 时的默认初始化

在主程序（LSTMLM.py）中，还可通过以下语句进行 LSTM 调用：

outputs, (_, _) = self.LSTM(X)

1. 因此需要添加在未传入 LSTM 胞体默认状态 state 时，实现默认初始化，因此将 forward() 函数 state 的默认值设为 None
2. 此时 num_layers、hidden_size 因为未传入 state，其值的获取途径也需要做相应改变。可放入初始化函数__init__当中获取
3. num_layers 的值也因此需要添加在__init__的参数列表，默认值为 1

修改后的代码如下：

```python
    # 需要存储一些网络规模有关数据
    self.num_layers = num_layers  # 网络层数
    self.hidden_size = hidden_size  # 中间变量维度（输出维度）

def forward(self, X, state=None):
    # 根据输入X，确定所运行设备
    device = X.device
    # 获取并设置各维长度
    [n_step, batch_size, _] = X.shape
    num_layers = self.num_layers
    hidden_size = self.hidden_size

    # 若state未传入参数，则使用默认初始化
    if state is None:
        hidden_state = torch.zeros((num_layers, batch_size, hidden_size)).to(device)
        cell_state = torch.zeros((num_layers, batch_size, hidden_size)).to(device)
    else:
        hidden_state = state[0].to(device)
        cell_state = state[1].to(device)
```

四、结果

1. 使用自定义的 LSTM 层时，使用以下命令均可成功运行，如图所示：

outputs, (_, _) = self.LSTM(X) （图 1.4.1）

outputs, (_, _) = self.LSTM(X, (hidden_state, cell_state)) （图 1.4.2）

图 1.4.1 使用 self.LSTM(X)，使用默认初始化方法

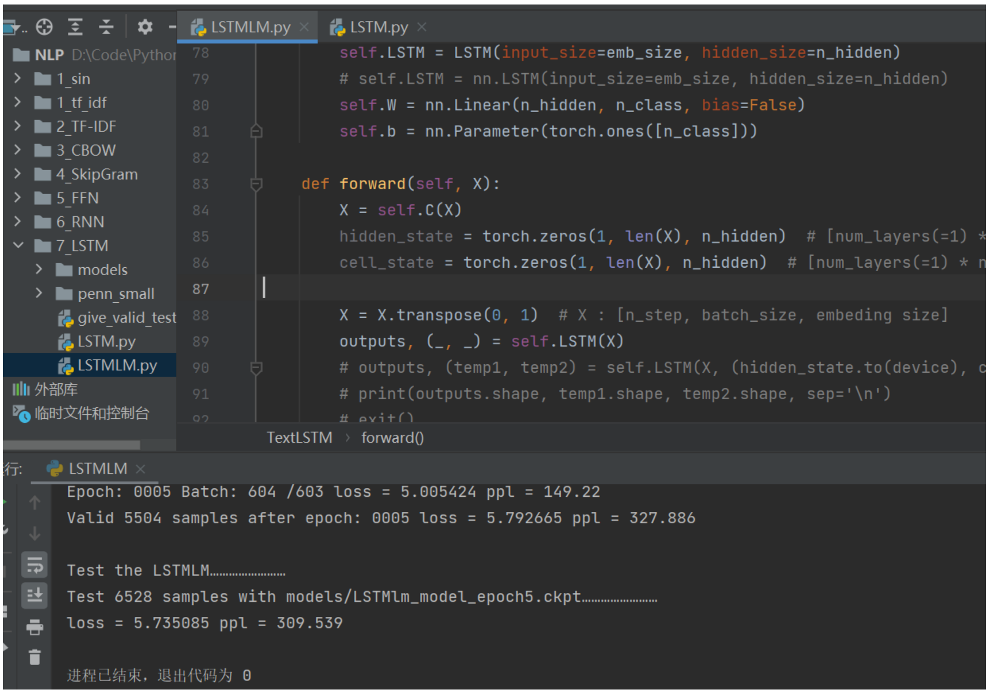
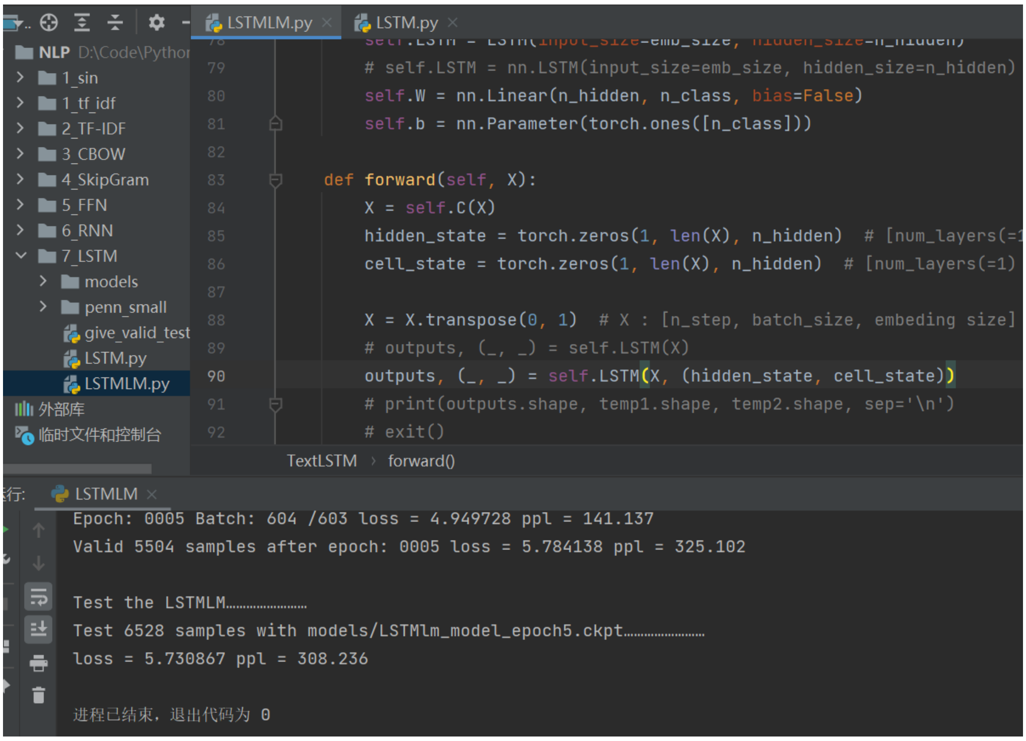


图 1.4.1 使用 self.LSTM(X, (hidden_state, cell_state))，使用外部参数



2. 通过多次运行测试，将使用自实现的 LSTM 和 nn.LSTM 的两种情况进行比较，发现自实现的 LSTM 和 nn.LSTM 在最终（epoch=5 的训练情况下）运行后，考虑到波动性将多次测试所得值进行综合计量，其 ppl 值基本一致。在此训练情况下均在(302,312)大致区间内。

   如下图所示，左侧图 1.4.3 为使用 nn.LSTM 进行测试最终所得结果，右侧图 1.4.4 为使用自定义的 LSTM 进行测试最终所得结果。

图 1.4.3 使用 nn.LSTM 所得结果

```
Epoch: 0002 Batch: 604 /603 loss = 5.431963 ppl = 228.598
Valid 5504 samples after epoch: 0002 loss = 5.984043 ppl = 397.043
Epoch: 0003 Batch: 100 /603 loss = 5.711364 ppl = 302.283
Epoch: 0003 Batch: 200 /603 loss = 5.413890 ppl = 224.503
Epoch: 0003 Batch: 300 /603 loss = 5.786543 ppl = 325.884
Epoch: 0003 Batch: 400 /603 loss = 6.039744 ppl = 419.786
Epoch: 0003 Batch: 500 /603 loss = 5.736348 ppl = 309.931
Epoch: 0003 Batch: 600 /603 loss = 5.787164 ppl = 326.087
Epoch: 0003 Batch: 604 /603 loss = 5.220675 ppl = 185.059
Valid 5504 samples after epoch: 0003 loss = 5.876801 ppl = 356.666
Epoch: 0004 Batch: 100 /603 loss = 5.522790 ppl = 250.333
Epoch: 0004 Batch: 200 /603 loss = 5.167781 ppl = 175.525
Epoch: 0004 Batch: 300 /603 loss = 5.550226 ppl = 257.296
Epoch: 0004 Batch: 400 /603 loss = 5.769549 ppl = 320.393
Epoch: 0004 Batch: 500 /603 loss = 5.572188 ppl = 263.009
Epoch: 0004 Batch: 600 /603 loss = 5.573737 ppl = 263.417
Epoch: 0004 Batch: 604 /603 loss = 5.045685 ppl = 155.351
Valid 5504 samples after epoch: 0004 loss = 5.810508 ppl = 333.789
Epoch: 0005 Batch: 100 /603 loss = 5.342841 ppl = 209.106
Epoch: 0005 Batch: 200 /603 loss = 4.964075 ppl = 143.176
Epoch: 0005 Batch: 300 /603 loss = 5.351686 ppl = 210.964
Epoch: 0005 Batch: 400 /603 loss = 5.530437 ppl = 252.254
Epoch: 0005 Batch: 500 /603 loss = 5.420314 ppl = 225.95
Epoch: 0005 Batch: 600 /603 loss = 5.373782 ppl = 215.677
Epoch: 0005 Batch: 604 /603 loss = 4.886020 ppl = 132.425
Valid 5504 samples after epoch: 0005 loss = 5.770569 ppl = 320.72

Test the LSTMLM.......................
Test 6528 samples with models/LSTMlm_model_epoch5.ckpt.......................
loss = 5.720450 ppl = 305.042

进程已结束，退出代码为 0
```

图 1.4.4 使用自定义的 LSTM 所得结果

```
Epoch: 0002 Batch: 604 /603 loss = 5.502728 ppl = 245.36
Valid 5504 samples after epoch: 0002 loss = 6.016611 ppl = 410.186
Epoch: 0003 Batch: 100 /603 loss = 5.695817 ppl = 297.62
Epoch: 0003 Batch: 200 /603 loss = 5.498709 ppl = 244.376
Epoch: 0003 Batch: 300 /603 loss = 5.849536 ppl = 347.073
Epoch: 0003 Batch: 400 /603 loss = 6.128087 ppl = 458.558
Epoch: 0003 Batch: 500 /603 loss = 5.778356 ppl = 323.227
Epoch: 0003 Batch: 600 /603 loss = 5.802881 ppl = 331.252
Epoch: 0003 Batch: 604 /603 loss = 5.312066 ppl = 202.769
Valid 5504 samples after epoch: 0003 loss = 5.901316 ppl = 365.518
Epoch: 0004 Batch: 100 /603 loss = 5.511003 ppl = 247.399
Epoch: 0004 Batch: 200 /603 loss = 5.247457 ppl = 190.082
Epoch: 0004 Batch: 300 /603 loss = 5.631226 ppl = 279.004
Epoch: 0004 Batch: 400 /603 loss = 5.875920 ppl = 356.352
Epoch: 0004 Batch: 500 /603 loss = 5.615081 ppl = 274.536
Epoch: 0004 Batch: 600 /603 loss = 5.591128 ppl = 268.038
Epoch: 0004 Batch: 604 /603 loss = 5.156785 ppl = 173.606
Valid 5504 samples after epoch: 0004 loss = 5.831859 ppl = 340.992
Epoch: 0005 Batch: 100 /603 loss = 5.339620 ppl = 208.433
Epoch: 0005 Batch: 200 /603 loss = 5.047616 ppl = 155.651
Epoch: 0005 Batch: 300 /603 loss = 5.436223 ppl = 229.573
Epoch: 0005 Batch: 400 /603 loss = 5.645106 ppl = 282.903
Epoch: 0005 Batch: 500 /603 loss = 5.470020 ppl = 237.465
Epoch: 0005 Batch: 600 /603 loss = 5.386901 ppl = 218.525
Epoch: 0005 Batch: 604 /603 loss = 5.006224 ppl = 149.34
Valid 5504 samples after epoch: 0005 loss = 5.790444 ppl = 327.158

Test the LSTMLM.......................
Test 6528 samples with models/LSTMlm_model_epoch5.ckpt.......................
loss = 5.727294 ppl = 307.137

进程已结束，退出代码为 0
```

提高内容 搭建双层 LSTM 网络

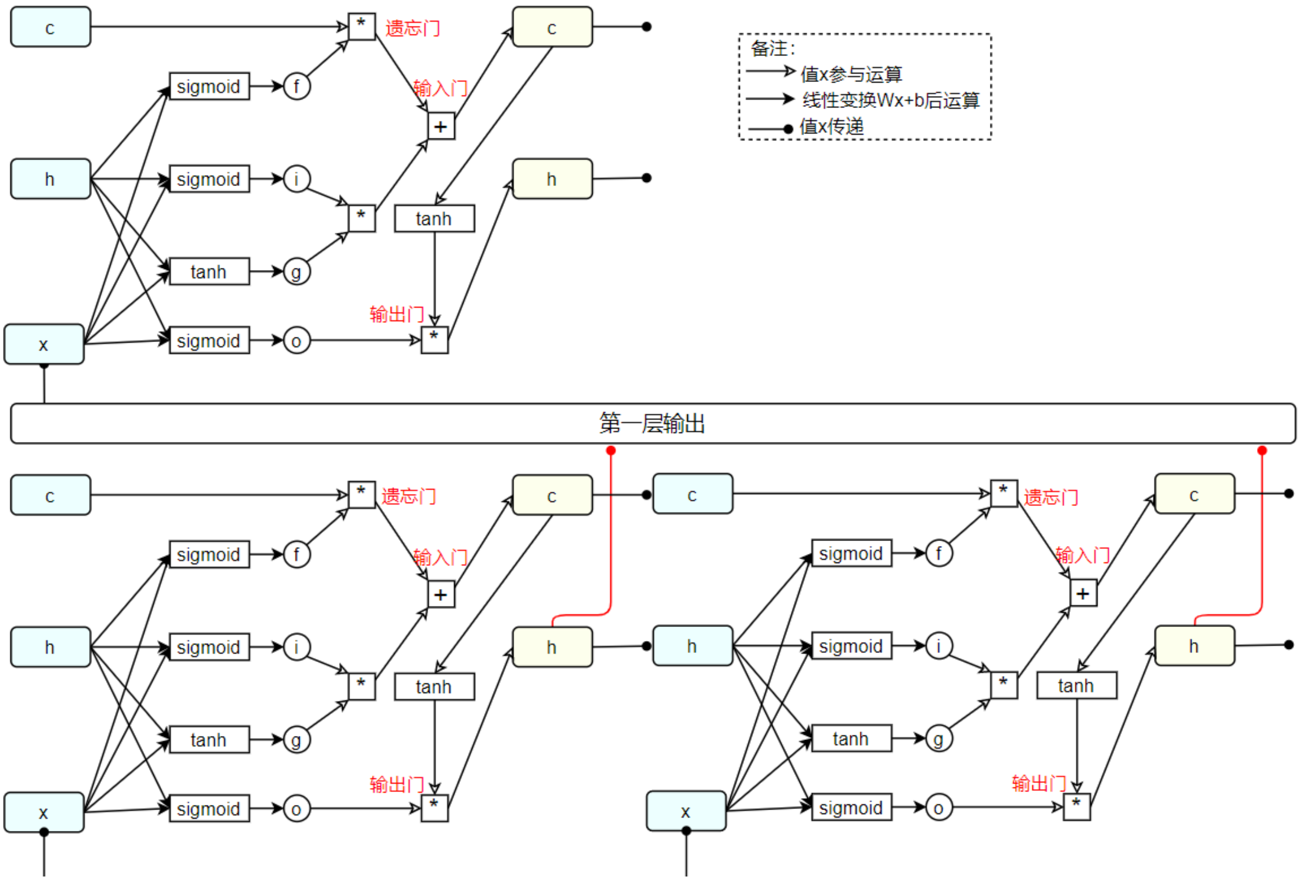
一、题目

在自己搭建出来的 LSTM 网络基础上，实现双层 LSTM 网络。

二、网络结构设计

1、流程图

使用“基础内容 搭建 LSTM 网络”部分设计单层 LSTM 网络，进行如下串联叠加。其中第一层的输入即为传入的 X，第二层的输入为第一层的输出（h_n 部分）。



2、程序实现

1. 将“基础内容 搭建 LSTM 网络”重原有的 LSTM 类改为 LSTM_Base 类
2. 添加 LSTM 类，在__init__中定义包含如下组件：

```
# 第一层
self.LSTM_one = LSTM_Base(input_size=input_size, hidden_size=hidden_size)
# 第二层（若有）
if num_layers == 2: self.LSTM_two = LSTM_Base(input_size=hidden_size, hidden_size=hidden_size)
```

3. 其中第一层的输入即为在主程序中调用时传入的 X，第二层的输入为第一层的输出（h_n 部分），即在 forward( )函数中通过如下过程实现：

```
outputs1, (h_n1, c_n1) = self.LSTM_one(X, (hidden_state[0],
cell_state[0]))
outputs2, (h_n2, c_n2) = self.LSTM_two(outputs1, (hidden_state[1],
cell_state[1]))
```

三、传入、传出及其他细节

1、传入

1. 将运行设备统一、未传 state 时的默认初始化、输入参数维度处理，等工作都从原有的单层网络类中（LSTM_Base）移出至新的 LSTM 类中，如下：

```
def forward(self, X, state=None):
    # 根据输入X，确定所运行设备
    device = X.device
    # 获取并设置各维长度
    [n_step, batch_size, _] = X.shape
    num_layers = self.num_layers
    hidden_size = self.hidden_size
    # 若state未传入参数，则使用默认初始化
    if state is None:
        hidden_state = torch.zeros((num_layers, batch_size, hidden_size)).to(device)
        cell_state = torch.zeros((num_layers, batch_size, hidden_size)).to(device)
    else:
        hidden_state = state[0].to(device)
        cell_state = state[1].to(device)
    # 输出存储
    h_n = torch.zeros((num_layers, batch_size, hidden_size)).to(device)  # 每一层最后一个结点的输出
    c_n = torch.zeros((num_layers, batch_size, hidden_size)).to(device)  # 每一层最后一个结点的记忆
```

2. 对于传入的 state 参数，其最外层的 num_layers 维在 LSTM 类中完成拆分。在本实验中，以第一层为例，在调用 LSTM 内部的 LSTM_Base 的 forward( )时，传入参数的 shape 变为：

   a) X：[n_step=5, batch_size=128, embeding size=256]
   b) hidden_state：[batch_size=128, n_hidden=256]
   c) cell_state：[batch_size=128, n_hidden=256]

在 LSTM_Base 中，其 forward( )函数变为如下：

```
def forward(self, X, state, n_step, batch_size, device):
    # 获取隐层结点初始值
    hidden_state = state[0]
    cell_state = state[1]
    outputs = torch.zeros((n_step, batch_size, self.hidden

    # 一层共n_step个结点（句子长度）
    for step in range(n_step):
        x_i = X[step]
```

2、传出

1. 在 LSTM 类中，两层（num_layers=2 时）网络以串联方式连接，先获取第一层输出 outputs1, (h_n1, c_n1)，再将 output1 作为第二层的输入传入进行计算。
2. 当网络层数为 2 时，对于 LSTM 网络总体，其输出中的 outputs 即为最后一层网络的 outputs2，其中的 h_n、c_n 分别将相应位置取值为各层网络输出中的 h_n1、c_n1，h_n2、c_n2。

如下图所示：

```
# 第一层
outputs1, (h_n1, c_n1) = self.LSTM_one(X, (hidden_state[0], cell_state[0]), n_step, ba
h_n[0], c_n[0] = h_n1, c_n1
# 第二层
if num_layers == 2:
    outputs2, (h_n2, c_n2) = self.LSTM_two(outputs1, (hidden_state[1], cell_state[1]),
    h_n[1], c_n[1] = h_n2, c_n2
    outputs = outputs2
else:
    outputs = outputs1
return outputs, (h_n, c_n)
```

四、结果

1. 使用自定义的 LSTM 层时，使用以下命令均可成功运行，如图所示：

outputs, (_, _) = self.LSTM(X)　（图 1.4.1）

outputs, (_, _) = self.LSTM(X, (hidden_state, cell_state))　（图 1.4.2）

图 1.4.1 使用 self.LSTM(X)，使用默认初始化方法

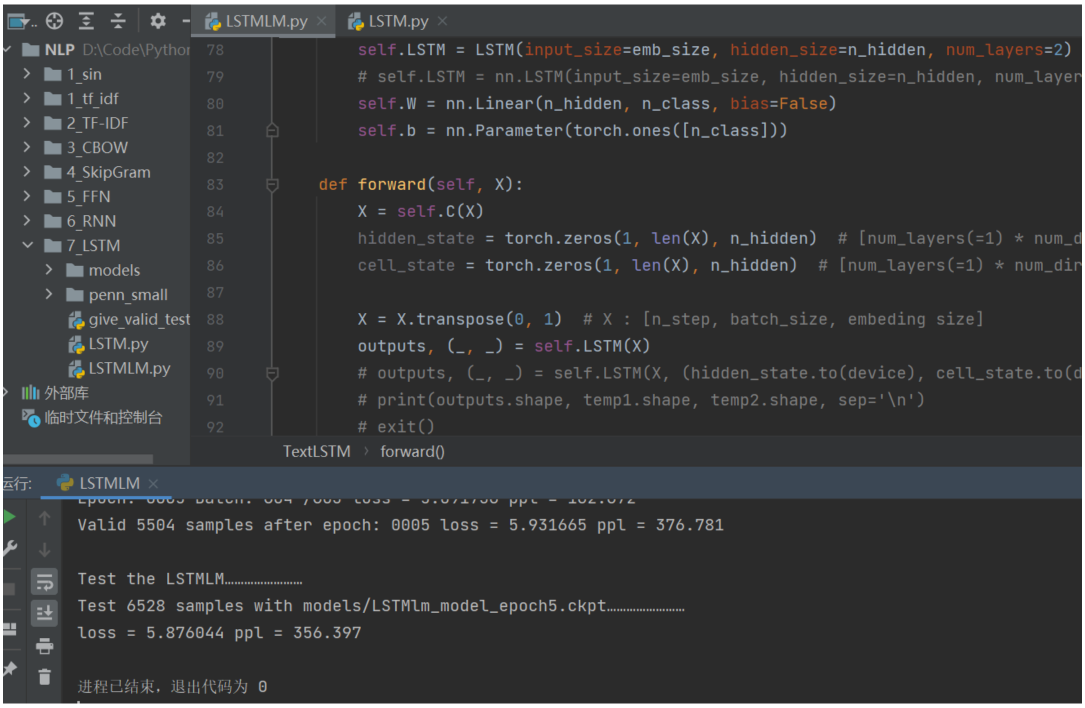
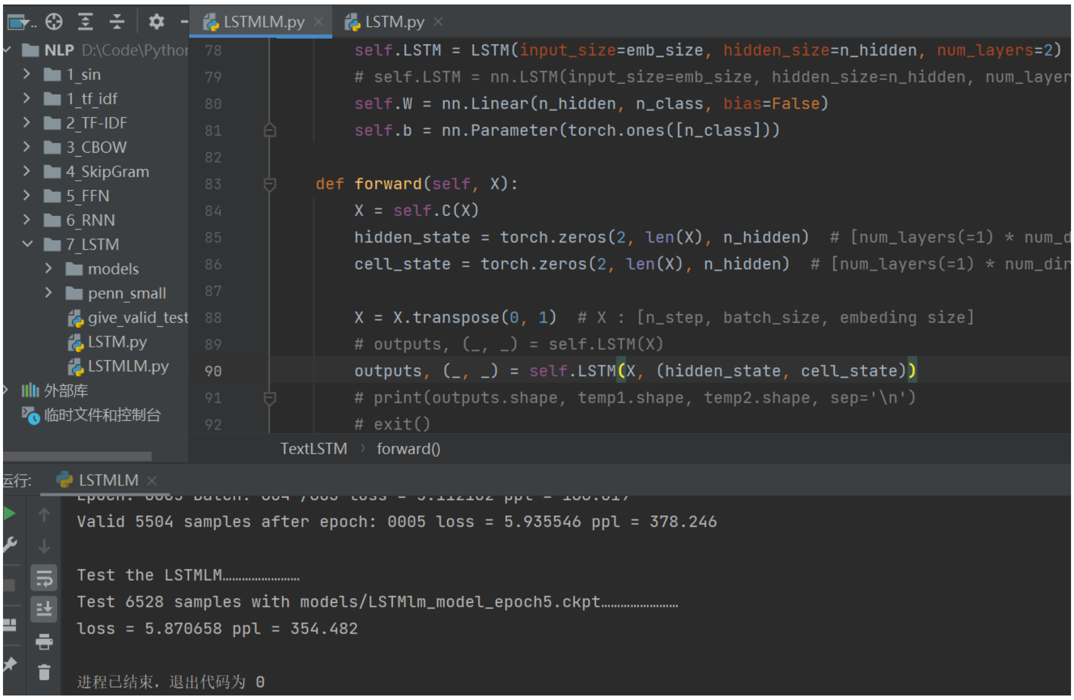


图 1.4.1 使用 self.LSTM(X, (hidden_state, cell_state))，使用外部参数



3. 将使用自实现的 LSTM 和 nn.LSTM 的两种情况进行 10 次运行比较，发现自实现的 LSTM 和 nn.LSTM 在最终（epoch=5 的训练情况下）运行后，考虑到波动性将多次测试所得值进行综合计量，自实现 LSTM 与 nn.LSTM 的 ppl 值大致相同。在此训练情况下自实现 LSTM 在(350, 360)大致区间内；自实现 LSTM 在(345, 355)大致区间内。

   如下图所示，左侧图 1.4.3 为使用 nn.LSTM 进行测试最终所得结果，右侧图 1.4.4 为使用自定义的 LSTM 进行测试最终所得结果。

图 1.4.3 使用 nn.LSTM 所得结果

```
Epoch: 0002 Batch: 604 /603 loss = 5.573294 ppl = 263.3
Valid 5504 samples after epoch: 0002 loss = 6.131390 ppl = 460.075
Epoch: 0003 Batch: 100 /603 loss = 5.854476 ppl = 348.792
Epoch: 0003 Batch: 200 /603 loss = 5.746926 ppl = 313.226
Epoch: 0003 Batch: 300 /603 loss = 5.982981 ppl = 396.621
Epoch: 0003 Batch: 400 /603 loss = 6.246356 ppl = 516.128
Epoch: 0003 Batch: 500 /603 loss = 5.883923 ppl = 359.216
Epoch: 0003 Batch: 600 /603 loss = 5.979486 ppl = 395.237
Epoch: 0003 Batch: 604 /603 loss = 5.371774 ppl = 215.244
Valid 5504 samples after epoch: 0003 loss = 6.020089 ppl = 411.615
Epoch: 0004 Batch: 100 /603 loss = 5.669478 ppl = 289.883
Epoch: 0004 Batch: 200 /603 loss = 5.503080 ppl = 245.447
Epoch: 0004 Batch: 300 /603 loss = 5.754241 ppl = 315.526
Epoch: 0004 Batch: 400 /603 loss = 6.056310 ppl = 426.798
Epoch: 0004 Batch: 500 /603 loss = 5.732818 ppl = 308.838
Epoch: 0004 Batch: 600 /603 loss = 5.795553 ppl = 328.834
Epoch: 0004 Batch: 604 /603 loss = 5.217835 ppl = 184.534
Valid 5504 samples after epoch: 0004 loss = 5.953866 ppl = 385.24
Epoch: 0005 Batch: 100 /603 loss = 5.509496 ppl = 247.027
Epoch: 0005 Batch: 200 /603 loss = 5.281480 ppl = 196.661
Epoch: 0005 Batch: 300 /603 loss = 5.554893 ppl = 258.499
Epoch: 0005 Batch: 400 /603 loss = 5.883299 ppl = 358.992
Epoch: 0005 Batch: 500 /603 loss = 5.604803 ppl = 271.728
Epoch: 0005 Batch: 600 /603 loss = 5.627411 ppl = 277.942
Epoch: 0005 Batch: 604 /603 loss = 5.075421 ppl = 160.04
Valid 5504 samples after epoch: 0005 loss = 5.916004 ppl = 370.927

Test the LSTMLM.......................
Test 6528 samples with models/LSTMlm_model_epoch5.ckpt.......................
loss = 5.847367 ppl = 346.321

进程已结束，退出代码为 0
```

图 1.4.4 使用自定义的 LSTM 所得结果

```
Epoch: 0002 Batch: 604 /603 loss = 5.608694 ppl = 272.788
Valid 5504 samples after epoch: 0002 loss = 6.143945 ppl = 465.888
Epoch: 0003 Batch: 100 /603 loss = 5.923146 ppl = 373.585
Epoch: 0003 Batch: 200 /603 loss = 5.811541 ppl = 334.133
Epoch: 0003 Batch: 300 /603 loss = 6.024852 ppl = 413.581
Epoch: 0003 Batch: 400 /603 loss = 6.319156 ppl = 555.104
Epoch: 0003 Batch: 500 /603 loss = 5.877145 ppl = 356.789
Epoch: 0003 Batch: 600 /603 loss = 5.983741 ppl = 396.923
Epoch: 0003 Batch: 604 /603 loss = 5.405024 ppl = 222.522
Valid 5504 samples after epoch: 0003 loss = 6.031965 ppl = 416.533
Epoch: 0004 Batch: 100 /603 loss = 5.740323 ppl = 311.165
Epoch: 0004 Batch: 200 /603 loss = 5.565217 ppl = 261.182
Epoch: 0004 Batch: 300 /603 loss = 5.829018 ppl = 340.024
Epoch: 0004 Batch: 400 /603 loss = 6.136288 ppl = 462.334
Epoch: 0004 Batch: 500 /603 loss = 5.743849 ppl = 312.264
Epoch: 0004 Batch: 600 /603 loss = 5.795632 ppl = 328.86
Epoch: 0004 Batch: 604 /603 loss = 5.229321 ppl = 186.666
Valid 5504 samples after epoch: 0004 loss = 5.960526 ppl = 387.814
Epoch: 0005 Batch: 100 /603 loss = 5.572866 ppl = 263.187
Epoch: 0005 Batch: 200 /603 loss = 5.346521 ppl = 209.877
Epoch: 0005 Batch: 300 /603 loss = 5.648643 ppl = 283.906
Epoch: 0005 Batch: 400 /603 loss = 5.946282 ppl = 382.329
Epoch: 0005 Batch: 500 /603 loss = 5.621239 ppl = 276.231
Epoch: 0005 Batch: 600 /603 loss = 5.627846 ppl = 278.062
Epoch: 0005 Batch: 604 /603 loss = 5.072190 ppl = 159.523
Valid 5504 samples after epoch: 0005 loss = 5.918902 ppl = 372.003

Test the LSTMLM.......................
Test 6528 samples with models/LSTMlm_model_epoch5.ckpt.......................
loss = 5.862568 ppl = 351.626

进程已结束，退出代码为 0
```

附录

一、主程序修改（使用说明）

1. 需要添加语句“from LSTM import LSTM”以使用自定义 LSTM
2. 使用如下语句进行使用，其中 input_size、hidden_size 为必须传入的形参，num_layers 可不传入，默认值为 1。

   LSTM(input_size=emb_size, hidden_size=n_hidden, num_layers=2)

3. 使用如下语句进行调用，其中从外界指定 state 初始值的调用其 X 与(hidden_state, cell_state)可在不同运行设备上，会自动转为在 X 所在设备。

   使用默认 state：outputs, (h_n, c_n) = self.LSTM(X)
   指定初始 state：outputs, (h_n, c_n) = self.LSTM(X, (hidden_state, cell_state))

二、LSTM. py 源码

```
import torch
from torch import nn


# 自实现 LSTM 结构单层基础
class LSTM_Base(nn.Module):
    # 输入维度、输出维度
    def __init__(self, input_size, hidden_size):
        super(LSTM_Base, self).__init__()
        # 遗忘门（旧记忆的占比）
        # f
        self.linear_if = nn.Linear(input_size, hidden_size)
        self.linear_hf = nn.Linear(hidden_size, hidden_size)
        self.sigmoid_f = nn.Sigmoid()
        # 输入门（新的记忆）
        # i
        self.linear_ii = nn.Linear(input_size, hidden_size)
        self.linear_hi = nn.Linear(hidden_size, hidden_size)
        self.sigmoid_i = nn.Sigmoid()
        # g
        self.linear_ig = nn.Linear(input_size, hidden_size)
        self.linear_hg = nn.Linear(hidden_size, hidden_size)
        self.tanh_g = nn.Tanh()
        # 输出门（由新记忆构造输出）
        # o
        self.linear_io = nn.Linear(input_size, hidden_size)
        self.linear_ho = nn.Linear(hidden_size, hidden_size)
        self.sigmoid_o = nn.Sigmoid()
        self.tanh_o = nn.Tanh()
        # 需要存储一些网络规模有关数据
        self.hidden_size = hidden_size  # 中间变量维度（输出维度）

    def forward(self, X, state, n_step, batch_size, device):
        # 获取隐层结点初始值
```

```
        hidden state = state[0]
        cell state = state[1]
        outputs = torch.zeros((n step, batch size, self.hidden size)).to(device)
# 最后一层所有结点的输出

        # 一层共 n step 个结点（句子长度）
        for step in range(n step):
            x i = X[step]
            # 遗忘门（旧记忆的占比）
            f = self.sigmoid f(self.linear if(x i) + self.linear hf(hidden state))
            # 输入门（新的记忆）
            i = self.sigmoid i(self.linear ii(x i) + self.linear hi(hidden state))
            g = self.tanh g(self.linear ig(x i) + self.linear hg(hidden state))
            cell state = f * cell state + i * g
            # 输出门
            o = self.sigmoid o(self.linear io(x i) + self.linear ho(hidden state))
            hidden state = o * self.tanh o(cell state)

            # 将所有 n step 个结点，值保存在 outputs 中
            outputs[step] = hidden state
        # 在每一层的最后一个结点，值保存在 outputs h、outputs c 中
        h n = hidden state
        c n = cell state
        return outputs, (h n, c n)


# 自实现双层 LSTM 结构（串联）
class LSTM(nn.Module):
    # 输入维度、输出维度
    def __init__(self, input_size, hidden_size, *, num_layers=1):
        super(LSTM, self).__init__()
        # 第一层
        self.LSTM_one = LSTM_Base(input_size=input_size, hidden_size=hidden_size)
        # 第二层（若有）
        if num_layers == 2:
            self.LSTM_two = LSTM_Base(input_size=hidden_size,
hidden_size=hidden_size)
        # 需要存储一些网络规模有关数据
        self.num_layers = num_layers  # 网络层数
        self.hidden_size = hidden_size  # 中间变量维度（输出维度）

    def forward(self, X, state=None):
        # 根据输入 X，确定所运行设备
        device = X.device
        # 获取并设置各维长度
        [n_step, batch_size, _] = X.shape
        num_layers = self.num_layers
        hidden_size = self.hidden_size
        # 若 state 未传入参数，则使用默认初始化
        if state is None:
            hidden state = torch.zeros((num layers, batch size,
hidden size)).to(device)
            cell state = torch.zeros((num layers, batch size,
hidden size)).to(device)
        else:
            hidden state = state[0].to(device)
            cell state = state[1].to(device)
        # 输出存储
        h n = torch.zeros((num layers, batch size, hidden size)).to(device)  # 每一
层最后一个结点的输出
        c n = torch.zeros((num layers, batch size, hidden size)).to(device)  # 每一
层最后一个结点的记忆
```

```
        # 第一层
        outputs1, (h_n1, c_n1) = self.LSTM_one(X, (hidden_state[0],
cell_state[0]), n_step, batch_size, device)
        h_n[0], c_n[0] = h_n1, c_n1
        # 第二层
        if num_layers == 2:
            outputs2, (h_n2, c_n2) = self.LSTM_two(outputs1, (hidden_state[1],
cell_state[1]), n_step, batch_size, device)
            h_n[1], c_n[1] = h_n2, c_n2
            outputs = outputs2
        else:
            outputs = outputs1
        return outputs, (h_n, c_n)
```