自然语言处理课程报告

1.LSTM 语言模型整体流程分析

为了更好地理解 LSTM 语言模型实验的整个流程，首先对实验提供的原文档(LSTMLM.py)进行分析。

1.1 参数定义

对程序中的参数进行说明，以便后续代码的理解。

表格 1 LSTM 语言模型参数说明

参数	值	含义
n_step	5	根据 n_step-1 个词预测下一个词
n_hidden	128	LSTM 隐层维度
batch_size	128	每个 batch 的大小
learn_rate	0.0005	学习率
all_epoch	5	训练总轮数
emb_size	256	词嵌入的维度

1.2 词表构建

通过对 make_dict 函数进行解读，发现本模型词表构建的具体流程是首先取出训练集中所有的单词，然后构建 word2number_dict 和 number2word_dict 两个字典，使得每一个单词可以通过 word2number_dict 映射到一个对应数字，每一个数字可以通过 number2word_dict 映射回一个词。在构建词典时，我们预留出四个位置来添加<pad> ,<unk_word> ,<sos>和<eos>这四个特殊标记。

其中 <PAD>主要用来进行字符补全，<EOS>和<GO>都是用来标识句子的起始与结束，<UNK>则用来替代一些未出现过的词或者低频词。

1.3 数据集构建

通过对 make_batch 函数进行解读，数据集构建流程如下。

第一步，取出训练集中每一行做分词后添加<sos>和<eos>形成每一个句子。

第二步，为长度不足 n_step 的句子添加<pad>，使每个句子长度大于等于 n_step

第三步，定义 input 为每个句子的最后 n-1 个词，定义 target 为每个句子的最后一个词。在这里发现如果一个句子的长度大于 n_step，那么它前 sen_length-n_step 个词将会丢失。

第四步，每构建 128 个（batch_size）个句子就把他们做成一个 batch 进而构建好了数据集。

1.4 模型定义

在构建好数据集后我们对 LSTM 语言模型进行搭建，在这里我一共搭建了 2 种结构的 LSTM，以及一个双层 LSTM。

第一种 LSTM 主要是根据根据 Pytorch 官方文档搭建的，代码详见附录 1，结构如下图：

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

图 1 pytorch 官方 LSTM 实现细节

其中h_t是隐藏状态，h_{t-1}是上一时刻隐藏状态，C_t是细胞状态，i_t是输入门，f_t是遗忘门，O_t是输出门，最终就是要将h_t送入下一个单元。

第二种 LSTM 是根据 PPT 上面的教程来搭建的，代码详见附录 2，结构如下图：

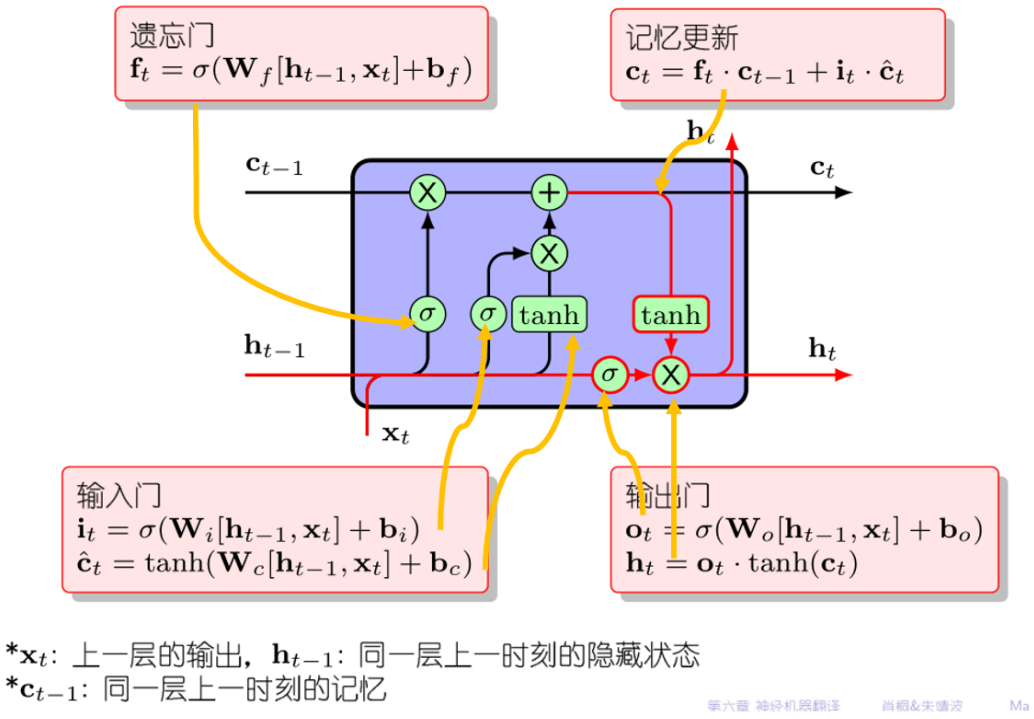

图 2 PPT 中提供的 LSTM 实现细节

通过分析可以看出上述两种 LSTM 的主要区别在于，Pytorch 官方文档给出的 LSTM 是通过将隐层的值乘以记忆门的权重与输入的值乘以输入门的权重求和来实现输入和记忆的混合；而 PPT 中的教程主要是通过将隐层的状态矩阵和输入矩阵进行拼接完成的。

双层 LSTM 主要是基于 pytoch 官方文档给出的结构的基础上进行搭建的，通过将第一次 LSTM 输出的隐层的值当做第二层 LSTM 的输入来完成对于双层 LSTM 的搭建。

这三种 LSTM 的区别和联系会在实验部分详细说明。

1.5 优化方法和评价方式

训练时的损失函数选用交叉熵损失函数(CrossEntropyLoss)，模型评价方法选择困惑度 ppl(perplexity)。

对于 PPL，我们可以这样理解，PPL 越小，一句我们期望的 sentence 出现的概率就越高。Perplexity 可以认为是 average branch factor（平均分支系数），即预测下一个词时可以有多少种选择。别人在作报告时说模型的 PPL 下降到 90，可以直观地理解为，在模型生成一句话时下一个词有 90 个合理选择，可选词数越少，我们大致认为模型越准确。这样也能解释，为什么 PPL 越小，模型越好。

2.实验

2.1 三种 LSTM 的实验结果分析

在本节中，我们分别使用官方实现的 nn.lstm，自己基于 pytorch 文档手动搭建的 lstm 和 PPT 中提供的 LSTM 来进行实验和分析。

通过实验，将结果记录于下表。

表格 2 三种 LSTM 的训练日志

轮次	官方 nn.lstm	基于官方文档手动实现	基于 PPT 实现
1	loss = 5.824519 ppl = 338.498	loss = 6.200636 ppl = 493.062	loss = 6.185050 ppl = 485.437
2	loss = 5.794195 ppl = 328.388	loss = 6.011457 ppl = 408.078	loss = 6.006950 ppl = 406.242
3	loss = 5.773635 ppl = 321.705	loss = 5.887130 ppl = 360.37	loss = 5.890641 ppl = 361.637
4	loss = 5.764179 ppl = 318.677	loss = 5.811646 ppl = 334.169	loss = 5.816112 ppl = 335.665
5	loss = 5.766505 ppl = 319.419	loss = 5.767154 ppl = 319.627	loss = 5.770846 ppl = 320.809
6	loss = 5.787026 ppl = 326.042	loss = 5.745973 ppl = 312.928	loss = 5.746541 ppl = 313.106
7	loss = 5.829112 ppl = 340.057	loss = 5.740398 ppl = 311.188	loss = 5.737530 ppl = 310.297
8	loss = 5.900217 ppl = 365.117	loss = 5.744743 ppl = 312.543	loss = 5.740403 ppl = 311.19
9	loss = 6.014758 ppl = 409.427	loss = 5.758236 ppl = 316.789	loss = 5.752067 ppl = 314.841

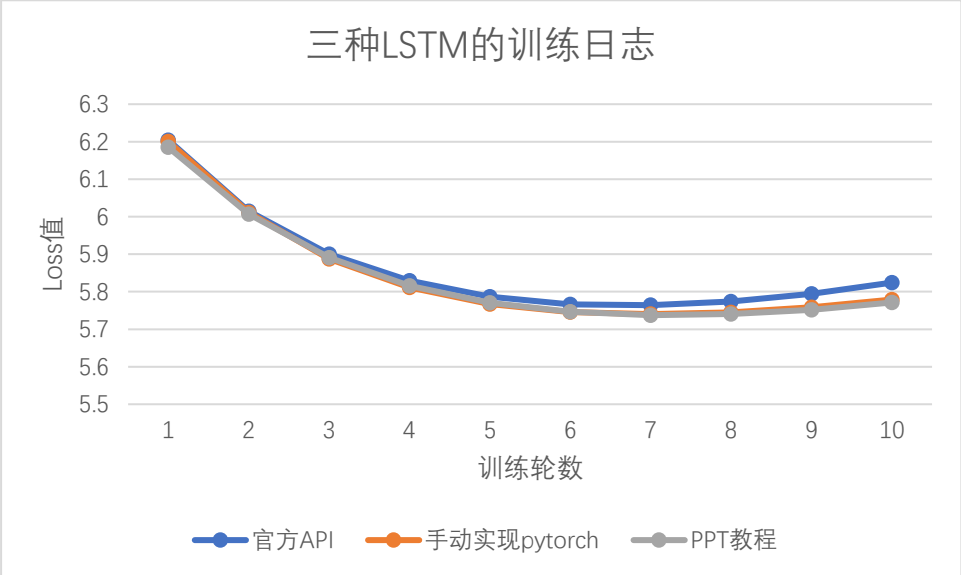

图 3 三种 LSTM 的训练损失

从上图可以看出三种 LSTM 均与第 7 轮收敛，且三种模型相差不大，自己动手实现的两种 LSTM 模型比官方的 API 效果要略微好一点，按照 Pytorch 官方文档给的公式搭出来的 LSTM 和 PPT 中详细介绍的 LSTM 几乎一模一样。

2.2 三种 LSTM 的区别分析

虽然利用官方 API 和手动实现官方文档的差异非常小，但是还是有区别的，感觉可能是官方 API 和自己手动实现的初始化方法不同造成的差异。

除此之外，自己实现的两种 LSTM 结果非常的接近，他们的训练的 loss 曲线几乎重合，但是他们这两种方法在计算的时候却有明显的区别，一种是通过将隐层的值乘以记忆门的权重与输入的值乘以输入门的权重求和来实现输入和记忆的混合；另一种是通过将隐层的状态矩阵和输入矩阵进行拼接完成的。所以这是一个令我感到奇怪的现象。

2.3.n_step 大小与门控单元的关系探究

n_step 是 LSTM 模型中的一个重要参数，在之前对代码的分析中我们发现，当 n_step 值为 n 时，我们是通过前 n-1 个词去预测最后一个词。根据 LSTM 模型的特点，是不是对于越长的 n_step 值，模型需要去记忆的内容就更多呢？n_step 的值是否和会影响记忆门和输入门的权值大小呢？

我分别将 n_step 设置为 3,5,7,9 来测试 n_step 参数大小对记忆门和输出门权值大小的影响，由于记忆门和输出门的权值是一个矩阵，难以直观反映其权值变化情况，因此取记忆门和输出门的权值矩阵的均值来反映其整体情况，实验结果如下表所示。

表格 3 LSTM 窗口大小与门控单元的关系表

n_step	i_t(输入门矩阵均值)	f_t(记忆门矩阵均值)
1	0.5033	0.4967
3	0.5021	0.5042
5	0.5018	0.4983
7	0.5014	0.4987
9	0.5004	0.4999

从上表结果可以看出，当 n_step 值比较小，也就是 LSTM 窗口比较小时，记忆门单元

的权值较低，输入门单元的权值较高，当窗口次数增加时，要记忆的信息也越来越多，所以记忆门的权值有所升高，输入门的权值有所下降。这与我们的预期是相符合的。

2.4.多层 LSTM

在本节中，我们分别尝试了二层、三层 LSTM 结构(代码详见附录 3,4)，并且将他们的效果同单层 LSTM 作比较。多层 LSTM 模型的结构均是基于之前手动搭建 Pytorch 官方文档的单层 LSTM 为基础实现的。

我分别记录了 1、2、3 层 LSTM 的训练日志、收敛轮数以及 ppl 的值，具体见下表。

表格 4 多层 LSTM 训练日志

轮数	单层 LSTM	双层 LSTM	三层 LSTM
1	loss = 6.200636 ppl = 493.062	loss = 6.265022 ppl = 525.853	loss = 6.417784 ppl = 612.644
2	loss = 6.011457 ppl = 408.078	loss = 6.091459 ppl = 442.066	loss = 6.216775 ppl = 501.085
3	loss = 5.887130 ppl = 360.37	loss = 5.966909 ppl = 390.297	loss = 6.095761 ppl = 443.972
4	loss = 5.811646 ppl = 334.169	loss = 5.882458 ppl = 358.69	loss = 6.007362 ppl = 406.41
5	loss = 5.767154 ppl = 319.627	loss = 5.830962 ppl = 340.686	loss = 5.959022 ppl = 387.231
6	loss = 5.745973 ppl = 312.928	loss = 5.806662 ppl = 332.507	loss = 5.935268 ppl = 378.141
7	loss = 5.740398 ppl = 311.188	loss = 5.803683 ppl = 331.518	loss = 5.931839 ppl = 376.847
8	loss = 5.744743 ppl = 312.543	loss = 5.816317 ppl = 335.733	loss = 5.943204 ppl = 381.154
9	loss = 5.758236 ppl = 316.789	loss = 5.841490 ppl = 344.292	loss = 5.967410 ppl = 390.493
10	loss = 5.779451 ppl = 323.582	loss = 5.876274 ppl = 356.478	loss = 6.005028 ppl = 405.462

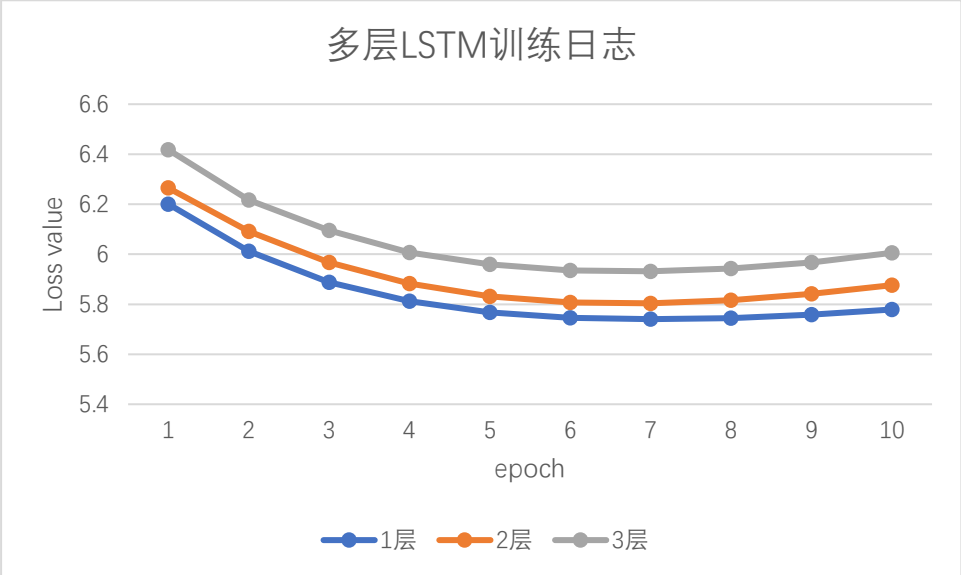

图 4 多层 LSTM 训练损失

从表 4 和图 4 可以看出单层 LSTM 收敛于第 7 轮，ppl 值为 311；双层 LSTM 收敛于第 7 轮，ppl 值为 331；三层 LSTM 收敛于第 7 轮，ppl 值 376。

我们发现单层 LSTM 在该数据集上表现最好，双层次，三层表现最差，所以在本实验这种小数据集下，更深层地模型并不意味着更好地效果，对于该任务单层 LSTM 模型已经拥有足够的表达能力。

3.致谢和建议

这学期的自然语言处理课程眨眼间就要结束了，虽然只有短短八周课程，但是这八周课程的内容非常的充实丰富，足以让一个小白对 NLP 领域有一个入门级别的认识。在这八周里，我在 NLP 方面的基础更加的扎实，对 embedding，self-attention 等概念了解得更加深入，可谓是受益匪浅。不仅在理论方面有了巨大提升，我还在实践能力方面有巨大的进步，通过这次手搭 LSTM 对 pytorch 的掌握更加熟练，虽然任务比较简单，但是整体的探究过程过程中还是收获了不少。

这里特别感谢三位助教学长从暑假开始做的 PPT 和教学的导航网站，感觉非常地用心，当然做的也非常棒，这门课教学的精细程度和方案的设计完全不亚于任何一门课程，可以说是来东大上过的最用心地一门选修课了。也要谢谢肖老师和马老师，有意为本科生开设这门课程，收获真的很大。

最后提一点点建议就是这门课的课时太短了，希望可以改成比较长期的课程，最后我比较期待的预训练模型部分很遗憾没有来得及讲，这种精品课程开设这么短的时间有些可惜了。除此之外还可以继续完善一下实践课的内容，很多次实验课的目的虽然明确，但是实际让我们操作的部分却不是很完善，难以通过实践课去很好地掌握对应的知识点，当然实践课的准备时间更是仓促，这只是一点美中不足罢了。

能选上这门课感觉非常幸运，希望之后能以本课程为基础继续在 NLP 领域一直学习下去。

4.附录

附录 1: LSTMLM_m1.py（手动实现 pytoch 官方文档）

```
class TextLSTM(nn.Module):
    def __init__(self):
        super(TextLSTM, self).__init__()
        self.C = nn.Embedding(n_class, embedding_dim=emb_size)
        self.W_ii = nn.Linear(emb_size, n_hidden, bias=True)
        self.W_hi = nn.Linear(n_hidden, n_hidden, bias=True)
        self.w_if = nn.Linear(emb_size, n_hidden, bias=True)
        self.W_hf = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_ig = nn.Linear(emb_size, n_hidden, bias=True)
        self.W_hg = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_io = nn.Linear(emb_size, n_hidden, bias=True)
        self.W_ho = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W = nn.Linear(n_hidden, n_class, bias=False)
        self.b = nn.Parameter(torch.ones([n_class]))

        self.sigmoid = nn.Sigmoid()
        self.tanh = nn.Tanh()

    def forward(self, X):
        X = self.C(X)
        hidden_state = torch.zeros(1, len(X), n_hidden)      # [num_layers(=1) *
num_directions(=1), batch_size, n_hidden]
        cell_state = torch.zeros(1, len(X), n_hidden)        # [num_layers(=1) *
num_directions(=1), batch_size, n_hidden]
        h_t_1 = hidden_state
        c_t_1 = cell_state
        X = X.transpose(0, 1) # X : [n_step, batch_size, embeding size]

        for x_t in X:
            i_t = self.sigmoid(self.W_ii(x_t)+self.W_hi(h_t_1))
            f_t = self.sigmoid(self.w_if(x_t)+self.W_hf(h_t_1))
            g_t = self.tanh(self.W_ig(x_t)+self.W_hg(h_t_1))
            o_t = self.sigmoid(self.W_io(x_t)+self.W_ho(h_t_1))
            c_t = f_t*c_t_1+i_t*g_t
            h_t = o_t*self.tanh(g_t)
            h_t_1 = h_t
            c_t_1 = c_t
        output = h_t
        output = output[-1]
        model = self.W(output)+self.b
```

```
        return model
```

附录 2: LSTMLM_m2.py（PPT 教程）

```
class TextLSTM(nn.Module):
    def __init__(self):
        super(TextLSTM, self).__init__()
        self.C = nn.Embedding(n_class, embedding_dim=emb_size)
        self.W_i = nn.Linear(emb_size+n_hidden, n_hidden, bias=True)
        self.w_f = nn.Linear(emb_size+n_hidden, n_hidden, bias=True)
        self.W_c = nn.Linear(emb_size+n_hidden, n_hidden, bias=True)
        self.W_o = nn.Linear(emb_size+n_hidden, n_hidden, bias=True)

        self.W = nn.Linear(n_hidden, n_class, bias=False)
        self.b = nn.Parameter(torch.ones([n_class]))

        self.sigmoid = nn.Sigmoid()
        self.tanh = nn.Tanh()

    def forward(self, X):
        X = self.C(X)
        hidden_state = torch.zeros(len(X), n_hidden)  # [num_layers(=1) * num_directions(=1), batch_size, n_hidden]
        cell_state = torch.zeros(len(X), n_hidden)  # [num_layers(=1) * num_directions(=1), batch_size, n_hidden]
        X = X.transpose(0, 1) # X : [n_step, batch_size, embeding size]
        h_t_1 = hidden_state
        c_t_1 = cell_state

        for x_t in X:
            h_x = torch.cat([h_t_1,x_t],1)
            i_t = self.sigmoid(self.W_i(h_x))
            c_t = self.tanh(self.W_c(h_x))
            f_t = self.sigmoid(self.w_f(h_x))
            o_t = self.sigmoid(self.W_o(h_x))
            h_t = o_t * self.tanh(c_t)
            c_t = f_t * c_t_1 + i_t * c_t
            h_t_1 = h_t
            c_t_1 = c_t
        print("i_t:",i_t,i_t.mean())
        print("f_t:",f_t,f_t.mean())
        output = h_t
        model = self.W(output)+self.b

        return model
```

附录 3: LSTMLM_2layers.py(2 层 LSTM 手动实现)

```
class TextLSTM(nn.Module):
    def __init__(self):
        super(TextLSTM, self).__init__()
        self.C = nn.Embedding(n_class, embedding_dim=emb_size)
        self.W_ii = nn.Linear(emb_size, n_hidden, bias=True)
        self.W_ii_2 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_hi = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_hi_2 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.w_if = nn.Linear(emb_size, n_hidden, bias=True)
        self.w_if_2 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_hf = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_hf_2 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_ig = nn.Linear(emb_size, n_hidden, bias=True)
        self.W_ig_2 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_hg = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_hg_2 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_io = nn.Linear(emb_size, n_hidden, bias=True)
        self.W_io_2 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_ho = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_ho_2 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W = nn.Linear(n_hidden, n_class, bias=False)
        self.b = nn.Parameter(torch.ones([n_class]))

        self.sigmoid = nn.Sigmoid()
        self.tanh = nn.Tanh()

    def forward(self, X):
        X = self.C(X)
        hidden_state = torch.zeros(1, len(X), n_hidden)      # [num_layers(=1) *
num_directions(=1), batch_size, n_hidden]
        cell_state = torch.zeros(1, len(X), n_hidden)        # [num_layers(=1) *
num_directions(=1), batch_size, n_hidden]
        layer1_output = torch.zeros(n_step,n_hidden,n_hidden)
        h_t_1 = hidden_state
        c_t_1 = cell_state
        X = X.transpose(0, 1) # X : [n_step, batch_size, embeding size]

        #layer 1 forward
        for id,x_t in enumerate(X):
            i_t = self.sigmoid(self.W_ii(x_t)+self.W_hi(h_t_1))
            f_t = self.sigmoid(self.w_if(x_t)+self.W_hf(h_t_1))
            g_t = self.tanh(self.W_ig(x_t)+self.W_hg(h_t_1))
            o_t = self.sigmoid(self.W_io(x_t)+self.W_ho(h_t_1))
```

```
            c_t = f_t*c_t_1+i_t*g_t
            h_t = o_t*self.tanh(g_t)
            h_t_1 = h_t
            c_t_1 = c_t
            layer1_output[id] = h_t

        #初始化 第二层的 hidden_state,cell state
        h_t_1 = hidden_state
        c_t_1 = cell_state
        #layer 2 forward
        for x_t in layer1_output:
            i_t = self.sigmoid(self.W_ii_2(x_t)+self.W_hi_2(h_t_1))
            f_t = self.sigmoid(self.w_if_2(x_t)+self.W_hf_2(h_t_1))
            g_t = self.tanh(self.W_ig_2(x_t)+self.W_hg_2(h_t_1))
            o_t = self.sigmoid(self.W_io_2(x_t)+self.W_ho_2(h_t_1))
            c_t = f_t*c_t_1+i_t*g_t
            h_t = o_t*self.tanh(g_t)
            h_t_1 = h_t
            c_t_1 = c_t
        output = h_t
        output = output[-1]
        model = self.W(output)+self.b

        return model
```

附录 4: LSTMLM_3layers.py(3 层 LSTM 手动实现)

```
class TextLSTM(nn.Module):
    def __init__(self):
        super(TextLSTM, self).__init__()
        self.C = nn.Embedding(n_class, embedding_dim=emb_size)
        self.W_ii = nn.Linear(emb_size, n_hidden, bias=True)
        self.W_ii_2 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_ii_3 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_hi = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_hi_2 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_hi_3 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_if = nn.Linear(emb_size, n_hidden, bias=True)
        self.W_if_2 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_if_3 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_hf = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_hf_2 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_hf_3 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_ig = nn.Linear(emb_size, n_hidden, bias=True)
        self.W_ig_2 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_ig_3 = nn.Linear(n_hidden, n_hidden, bias=True)
```

```
        self.W_hg = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_hg_2 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_hg_3 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_io = nn.Linear(emb_size, n_hidden, bias=True)
        self.W_io_2 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_io_3 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_ho = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_ho_2 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W_ho_3 = nn.Linear(n_hidden, n_hidden, bias=True)
        self.W = nn.Linear(n_hidden, n_class, bias=False)
        self.b = nn.Parameter(torch.ones([n_class]))

        self.sigmoid = nn.Sigmoid()
        self.tanh = nn.Tanh()

    def forward(self, X):
        X = self.C(X)
        hidden_state = torch.zeros(1, len(X), n_hidden)      # [num_layers(=1) *
num_directions(=1), batch_size, n_hidden]
        cell_state = torch.zeros(1, len(X), n_hidden)        # [num_layers(=1) *
num_directions(=1), batch_size, n_hidden]
        layer1_output = torch.zeros(n_step,n_hidden,n_hidden)
        layer2_output = torch.zeros(n_step,n_hidden,n_hidden)
        h_t_1 = hidden_state
        c_t_1 = cell_state
        X = X.transpose(0, 1) # X : [n_step, batch_size, embeding size]

        #layer 1 forward
        for id,x_t in enumerate(X):
            i_t = self.sigmoid(self.W_ii(x_t)+self.W_hi(h_t_1))
            f_t = self.sigmoid(self.W_if(x_t)+self.W_hf(h_t_1))
            g_t = self.tanh(self.W_ig(x_t)+self.W_hg(h_t_1))
            o_t = self.sigmoid(self.W_io(x_t)+self.W_ho(h_t_1))
            c_t = f_t*c_t_1+i_t*g_t
            h_t = o_t*self.tanh(g_t)
            h_t_1 = h_t
            c_t_1 = c_t
            layer1_output[id] = h_t

        #初始化 第二层的 hidden_state,cell state
        h_t_1 = hidden_state
        c_t_1 = cell_state
        #layer 2 forward
        for id,x_t in enumerate(layer1_output):
```

```
            i_t = self.sigmoid(self.W_ii_2(x_t)+self.W_hi_2(h_t_1))
            f_t = self.sigmoid(self.W_if_2(x_t)+self.W_hf_2(h_t_1))
            g_t = self.tanh(self.W_ig_2(x_t)+self.W_hg_2(h_t_1))
            o_t = self.sigmoid(self.W_io_2(x_t)+self.W_ho_2(h_t_1))
            c_t = f_t*c_t_1+i_t*g_t
            h_t = o_t*self.tanh(g_t)
            h_t_1 = h_t
            c_t_1 = c_t
            layer2_output[id] = h_t

        #初始化 第三层的 hidden_state,cell state
        h_t_1 = hidden_state
        c_t_1 = cell_state
        #layer 3 forward
        for id,x_t in enumerate(layer2_output):
            i_t = self.sigmoid(self.W_ii_3(x_t)+self.W_hi_3(h_t_1))
            f_t = self.sigmoid(self.W_if_3(x_t)+self.W_hf_3(h_t_1))
            g_t = self.tanh(self.W_ig_3(x_t)+self.W_hg_3(h_t_1))
            o_t = self.sigmoid(self.W_io_3(x_t)+self.W_ho_3(h_t_1))
            c_t = f_t*c_t_1+i_t*g_t
            h_t = o_t*self.tanh(g_t)
            h_t_1 = h_t
            c_t_1 = c_t
        output = h_t
        output = output[-1]
        model = self.W(output)+self.b

        return model
```