基于PyTorch的LSTM模型实现

一、 LSTM模型原理分析

1.基于RNN的神经网络

传统前馈神经网络受到结构的制约，其分析和学习的样本范围只能局限于一个较小的常数（即输入层的样本维度），这样的限制对于自然语言处理任务而言是一个很大的缺陷。考虑到语言文字的内在逻辑联系不会仅仅局限在目标词的前后几个词上，因此我们需要一种模型能够拥有一种“记忆”的能力，在对词语进行分析预测时能够将前文所见过的词语都纳入考量范围。由此，我们提出了RNN（循环神经网络）模型，它为传统神经网络加入了“时间”维度。

2.对于“记忆”的进一步优化

RNN模型提出的对前文的”记忆“概念已经向贴近人类语言习惯上迈出了一大步，然而这种对于历史信息的处理方式还是有些粗糙。循环神经网络虽然体现出了记忆的特性，但是其总体的记忆曲线还是呈现出随着”时间“增加而逐渐”遗忘“的趋势。我们都有过做阅读题的经验：通常来说，一篇文章的标题和段首句都会与文章内容有着较强的关联，我们在阅读文章主体时都会时不时回头去看一看这些”关键词“，机器在传统RNN网络下执行NLP任务时却会被最近收到的一些信息冲淡这些关键记忆。如果它们也能拥有这种选择记忆的能力，那将会更加贴近人类对语言的理解模式。因此，我们对传统RNN网络的记忆模式加以改进，让其对信息的记忆更加灵活，赋予网络自主选择”记忆要点“的权利。

3.”门“结构

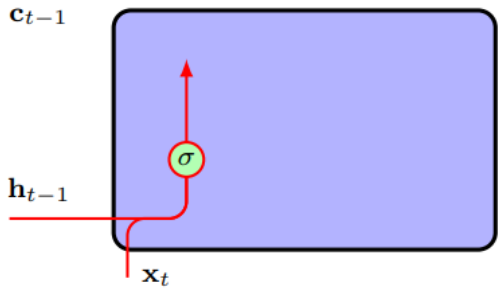
假如我们想改进一个机械臂，目标是使其更加灵活，那么增加活动关节是一个不错的选择。对于循环神经网络的循环单元（cell）来说也是如此，我们想要它实现更加复杂和”智能“的记忆模式，就需要设置更多的参数和一些特殊的结构来供其学习，以便使训练好的模型能够更加灵活地执行任务。

在LSTM（Long Short-Term Memory)中，循环单元的外部结构与RNN比起来十分相似，只是将需要向下一个循环单元传递的参数由一个增加到了两个，其主要的改进在于cell内部新增的三个”门“结构。（此处引用课上ppt的门结构原理图）

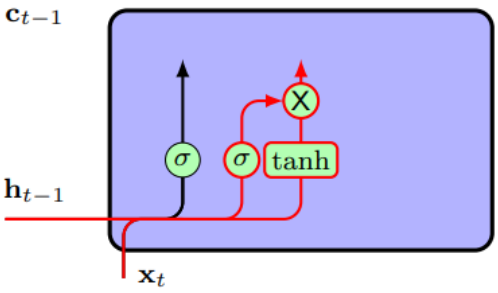
遗忘门：

遗忘门

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f)$$



输入门：

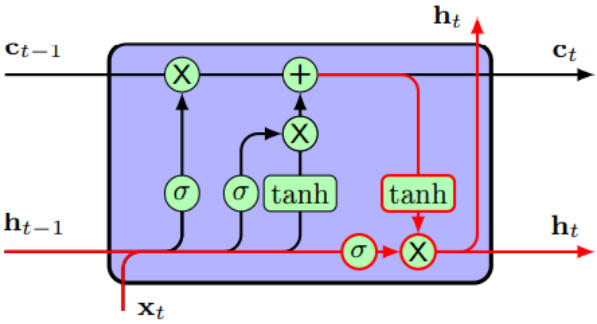


输入门
$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i)$
$\hat{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c)$

输出门：

遗忘门
$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f)$

记忆更新
$\mathbf{c}_t = \mathbf{f}_t \cdot \mathbf{c}_{t-1} + \mathbf{i}_t \cdot \hat{\mathbf{c}}_t$



输入门
$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i)$
$\hat{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c)$

输出门
$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o)$
$\mathbf{h}_t = \mathbf{o}_t \cdot \tanh(\mathbf{c}_t)$

有了这三个门的控制，再结合从训练中获取的理想参数，LSTM模型可以实现对于大篇幅文本的长短选择记忆功能。在此基础上，还可以对模型进行一些适应性改造：搭建多层的循环单元、从双向进行序列分析等。

二、LSTM模型的实现

1.一些准备工作

关于数据：学长发给我们的数据形式为一些英文语段，其中包含了一部分的首尾标注。

按照要求，需要先对语句进行划分，构造词典，确定单词和数字间的映射：

```
def make_dict(train_path):#构建训练数据字典
    text = open(train_path, 'r', encoding='utf-8')  #读文件
    word_list = set()  # a set for making dict
    for line in text:
        line = line.strip().split(" ")
        word_list = word_list.union(set(line))
    word_list = list(sorted(word_list))   #set to list
    word2number_dict = {w: i+2 for i, w in enumerate(word_list)}
    number2word_dict = {i+2: w for i, w in enumerate(word_list)}
    #add the <pad> and <unk_word>
    word2number_dict["<pad>"] = 0
    number2word_dict[0] = "<pad>"
```

```
    word2number_dict["<unk_word>"] = 1
    number2word_dict[1] = "<unk_word>"
    word2number_dict["<sos>"] = 2
    number2word_dict[2] = "<sos>"
    word2number_dict["<eos>"] = 3
    number2word_dict[3] = "<eos>"
    return word2number_dict, number2word_dict
```

拿到单词与数字间的映射后，进行训练样本的构造。在此会基于设定的步长（sequence_len），将读取的英文语句做如下处理：

（1）将每个sentence分割成word（list形式）

（2）为每个word list添加特定的首尾符号，对于长度不足sequence_len的用特殊符号补齐

（3）从头遍历word list，取出sequence_len-1个单词作为输入数据x，sequence_len处的单词作为期望的输出y

（4）当获得的(x,y)达到batch_size时存储为一个batch

2.搭建LSTM单元

完成数据准备工作后，开始使用nn.Embedding、nn.Linear等函数搭建基于LSTM单元的循环网络。

```
class TextLSTM(nn.Module):
    def __init__(self):
        super(TextLSTM, self).__init__()
        self.C = nn.Embedding(n_class, embedding_dim=emb_size)#词嵌入
        #LSTM单元用到的参数和函数
        self.cell_size=n_hidden
        self.gate = nn.Linear(emb_size + n_hidden, self.cell_size)
        self.sigmoid = nn.Sigmoid()
        self.tanh = nn.Tanh()
        #输出层用到的函数
        self.model = nn.Linear(n_hidden, n_class)
        self.softmax = nn.LogSoftmax(dim=1)

    def lstm_cell(self,x,hidden,cell):#LSMTM单元
        combined = torch.cat((hidden, x), 0)# 拼接数据
        f_gate = self.sigmoid(self.gate(combined))  # 遗忘门
        i_gate = self.sigmoid(self.gate(combined))  # 输入门
        c_t = self.tanh(self.gate(combined))
        cell = torch.add(torch.mul(f_gate, cell), torch.mul(i_gate, c_t))# 记忆更
新
        o_gate = self.sigmoid(self.gate(combined))  # 输出门
        hidden = torch.mul(self.tanh(cell), o_gate) # 输出数据
        return hidden, cell

    def forward(self, input):
        input=self.C(input)
        h=torch.zeros(n_hidden).cuda()
        c=torch.zeros(self.cell_size).cuda()
        outputs=torch.zeros(1,n_hidden).cuda()#一些初始化
        for X in input:
            for x in X:
              h,c=self.lstm_cell(x,h,c)
              h_t=torch.unsqueeze(h,0)
            outputs=torch.cat((outputs,h_t),0)
```

```
        outputs=outputs[torch.arange(outputs.size(0))!=0]#输出结果
        model=self.softmax(self.model(outputs))#预测结果
```

在手动搭建的过程中，最关键的问题就是要对应好每一层网络的输入和输出维度，只有将各个传递处的维度做好匹配才能让程序顺利运行。

这里，所有门单元采用的输入输出尺寸为[emb_size+n_hidden,cell_size]，cell_size为循环单元的内部参数，只在其内部的运算中起作用，这里为了运算方便将其设定为与隐藏层维度（n_hidden）相同的尺寸。此外我在调试过程中还遇到了数据一部分在CPU，一部分在GPU上的报错情况，后经分析是由于我自定义的初始化变量没有随着模型加入到cuda中，在其后手动添加命令后问题就得以解决。

3.运行测试

经过调试后，使用自己编写的LSTM单元的程序可以顺利运行，在学长提供的数据中跑出了如下结果：

```
Train the LSTMLM.....
TextLSTM(
  (C): Embedding(7615, 256)
  (gate): Linear(in_features=384, out_features=128, bias=True)
  (sigmoid): Sigmoid()
  (tanh): Tanh()
  (model): Linear(in_features=128, out_features=7615, bias=True)
  (softmax): LogSoftmax(dim=1)
)
Valid 5504 samples after epoch: 0001 loss = 6.236889 ppl = 511.265
Valid 5504 samples after epoch: 0002 loss = 6.072085 ppl = 433.584
Valid 5504 samples after epoch: 0003 loss = 5.963693 ppl = 389.044
Valid 5504 samples after epoch: 0004 loss = 5.892098 ppl = 362.164
Valid 5504 samples after epoch: 0005 loss = 5.846473 ppl = 346.012

Test the LSTMLM.....
Test 6528 samples with models/LSTMlm_model_epoch5.ckpt.....
loss = 5.786897 ppl = 326.0
```

对比使用PyTorch内置LSTM模块的程序，其训练效果和测试成绩都相差无几，达到了预期效果。但是我自己实现的网络在运算速度方面的表现相当“感人”，它完成一个epoch的时间足够使用LSTM包的程序算完五个epoch。为此，我也特地请教了研究生学长，得知Pytorch的LSTM包是经过了专门设计优化过的，另外学长建议我查看一下各部分的运行时间，看看还有没有优化的空间。

在分析了运行时间的占用情况后，我发现影响运算速度的关键应该是出在了前向传播时的双层循环上。在编写代码时，为了最直观地反映原理，我采用的运算策略是依次遍历batch中的每个(x,y)，然后再按顺序对(x,y)里的每个元素执行循环预测。对于每条(x,y)的预测运算是具有时序性的，只有算出前文的参数并向后传递才能预测出最终结果，但是batch中的每条数据之间是可以以矩阵的形式并行运算的。将batch以矩阵而不是行遍历的形式直接送入网络进行运算，这样的改进使程序的运行速度能提升一倍左右，但是需要更加细致地考虑LSTM cell内部的维度关系。

4.双层LSTM

在成功实现LSTM的模型后，我又尝试实现了双层结构的LSTM单元：

```
    def lstm_cell(self,x,hidden_l1,hidden_l2,cell_l1,cell_l2):
        combined_l1 = torch.cat((hidden_l1, x), 0)# 拼接数据
        f_gate_l1 = self.sigmoid(self.gate_l1(combined_l1))  # 遗忘门
        i_gate_l1 = self.sigmoid(self.gate_l1(combined_l1))  # 输入门
        c_t_l1 = self.tanh(self.gate_l1(combined_l1))
```

```
        cell_l1 = torch.add(torch.mul(f_gate_l1, cell_l1), torch.mul(i_gate_l1,
c_t_l1))# 记忆更新
        o_gate_l1 = self.sigmoid(self.gate_l1(combined_l1))  # 输出门
        hidden_l1 = torch.mul(self.tanh(cell_l1), o_gate_l1)#第一层的输出[128]

        combined_l2 = torch.cat((hidden_l2,hidden_l1),0)
        f_gate_l2 = self.sigmoid(self.gate_l2(combined_l2))  # 遗忘门
        i_gate_l2 = self.sigmoid(self.gate_l2(combined_l2))  # 输入门
        c_t_l2 = self.tanh(self.gate_l2(combined_l2))
        cell_l2 = torch.add(torch.mul(f_gate_l2, cell_l2), torch.mul(i_gate_l2,
c_t_l2))  # 记忆更新
        o_gate_l2 = self.sigmoid(self.gate_l2(combined_l2))  # 输出门
        hidden_l2 = torch.mul(self.tanh(cell_l2), o_gate_l2)  # 第二层的输出[128]

        return hidden_l1,hidden_l2,cell_l1,cell_l2
```

经过运行测试后得到如下结果：

```
Train the LSTMLM.....
TextLSTM(
  (C): Embedding(7615, 256)
  (gate_l1): Linear(in_features=384, out_features=128, bias=True)
  (gate_l2): Linear(in_features=256, out_features=128, bias=True)
  (sigmoid): Sigmoid()
  (tanh): Tanh()
  (model): Linear(in_features=128, out_features=7615, bias=True)
  (softmax): LogSoftmax(dim=1)
)
Valid 5504 samples after epoch: 0001 loss = 6.423188 ppl = 615.964
Valid 5504 samples after epoch: 0002 loss = 6.325488 ppl = 558.63
Valid 5504 samples after epoch: 0003 loss = 6.239733 ppl = 512.722
Valid 5504 samples after epoch: 0004 loss = 6.177233 ppl = 481.657
Valid 5504 samples after epoch: 0005 loss = 6.125458 ppl = 457.354

Test the LSTMLM.....
Test 6528 samples with models/LSTMlm_model_epoch5.ckpt.....
loss = 6.058210 ppl = 427.609
```

相比于上文的单层结构，其耗时更长，却没能得到更好的评估结果。

三、课程总结

很开心能和肖老师、马老师以及三位研究生学长共同完成这门课程的学习。半学期的课程虽短，但我也有不少的收获。从对自然语言处理任务略知一二到能亲手训练出来自己的模型，这些都离不开老师和学长的指导和帮助。除了对实际模型的理解，这门课还让我加深了对于nlp发展趋势的认知，看到了几代模型方法之间的迭代和演化。从统计模型，到RNN网络，再到针对"记忆"而改进的循环单元，以及创新的注意力机制和transformer，它们也越来越贴近人们的语言习惯。相信在将来，一些更完备的自然语言处理方法也能更好地服务人们的生活。

最后，祝老师和学长身体健康，工作顺利！