《自然语言处理》大作业

自实现单双层 LSTM 模型

学　　院：计算机科学与工程

班　　级：人工智能 2001

姓　　名：许子强

学　　号：20201111

一、系统设计

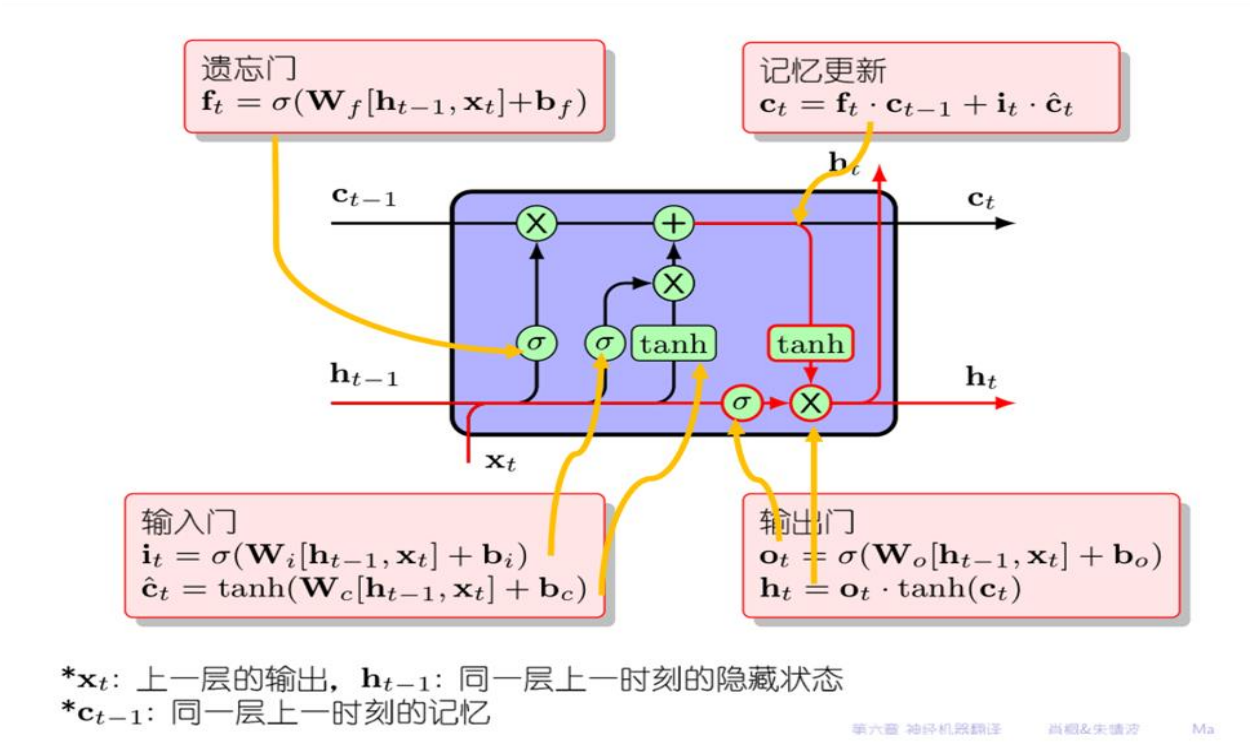
1.1 单层 LSTM

LSTM 的关键是细胞状态（即 cell state），表示为 C_t，用来保存当前 LSTM 的状态信息并传递到下一时刻的 LSTM 中。当前的 LSTM 接收来自上一个时刻的细胞状态 C_{t-1}，并与当前 LSTM 接收的信号输入 x_t 共同作用产生当前 LSTM 的细胞状态 C_t。

LSTM 主要包括三个门结构：遗忘门、记忆门、输出门。这三个门用来控制 LSTM 的信息保留和传递，最终反映到细胞状态 C_t 和输出信号 h_t。

遗忘门决定了细胞状态 C_{t-1} 中的哪些信息将被遗忘。记忆门决定新输入的信息 x_t 和 h_{t-1} 中哪些信息将被保留。

1.1.1 计算公式



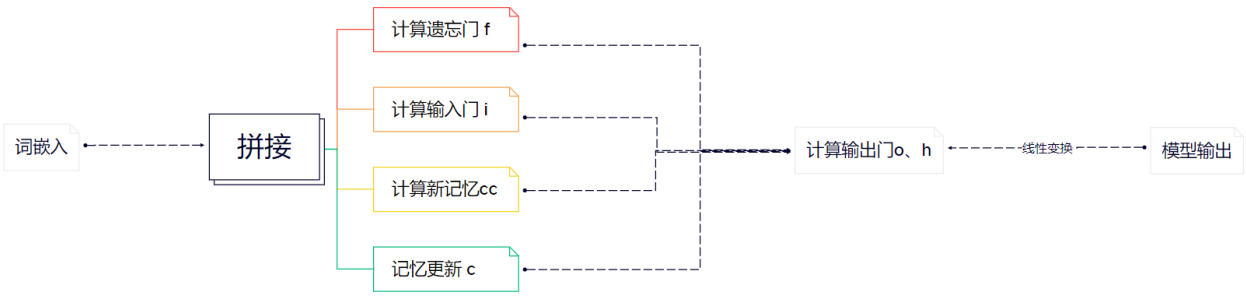
1.1.2 参数设计

```
# 遗忘门(forget)参数
self.W_f = nn.Linear(n_hidden+emb_size, n_hidden, bias=True)
# 输入门(input)参数
self.W_i = nn.Linear(n_hidden+emb_size, n_hidden, bias=True)
self.W_c = nn.Linear(n_hidden+emb_size, n_hidden, bias=True)
# 输出门(output)参数
self.W_o = nn.Linear(n_hidden+emb_size, n_hidden, bias=True)
# 激活函数
self.sigmoid = nn.Sigmoid()
self.tanh = nn.Tanh()
# 最终输出层
self.W = nn.Linear(n_hidden, n_class, bias=True)
```

1.1.3 前向传播

```
c_0 = torch.zeros(sample_size, n_hidden)  # 记忆状态初值，设置为全0
h_0 = torch.zeros(sample_size, n_hidden)  # 隐藏状态初值，设置为全0
c = c_0    # 记忆状态初始化
h = h_0    # 隐藏状态初始化
for x in X:
    # 拼接形成[ h[t-1], x[t] ]
    catenate = torch.cat([h, x], dim=1)    # dim=1表示按行拼接,catenate: [batch_size, n_hidden+emb_size]
    # 遗忘门计算
    f = self.sigmoid(self.W_f(catenate))
    # 输入门计算
    i = self.sigmoid(self.W_i(catenate))
    cc = self.tanh(self.W_c(catenate))
    # 记忆更新
    c = torch.mul(f, c) + torch.mul(i, cc)
    # 输出门计算
    o = self.sigmoid(self.W_o(catenate))
    h = torch.mul(o, self.tanh(c))
# model_output = nn.functional.softmax(self.W(h), dim=1)     # dim=1, 对行作归一化
model_output = self.W(h)
```

1.1.4 流程概要



1.2 双层 LSTM

双层 LSTM 整体思路与单层并无大的差异，只是额外把第一层的输出作为第二层的输入，整体上大致相当于用相同的流程处理了两次，只是第二层参数的输入维度发生了变化——由 [n_hidden+emb_size, n_hidden] 变为 [n_hidden+n_hidden, n_hidden]。

1.2.1 参数设计

```
# 遗忘门(forget)参数
self.W1_f = nn.Linear(n_hidden+emb_size, n_hidden, bias=True)
self.W2_f = nn.Linear(n_hidden+n_hidden, n_hidden, bias=True)
# 输入门(input)参数
self.W1_i = nn.Linear(n_hidden+emb_size, n_hidden, bias=True)
self.W1_c = nn.Linear(n_hidden+emb_size, n_hidden, bias=True)
self.W2_i = nn.Linear(n_hidden+n_hidden, n_hidden, bias=True)
self.W2_c = nn.Linear(n_hidden+n_hidden, n_hidden, bias=True)
# 输出门(output)参数
self.W1_o = nn.Linear(n_hidden+emb_size, n_hidden, bias=True)
self.W2_o = nn.Linear(n_hidden+n_hidden, n_hidden, bias=True)
# 激活函数
self.sigmoid = nn.Sigmoid()
self.tanh = nn.Tanh()
# 最终输出层
self.W = nn.Linear(n_hidden, n_class, bias=True)
```

1.2.2 前向传播

```
c1_0 = torch.zeros(sample_size, n_hidden)  # 第一层记忆状态初值为全0
h1_0 = torch.zeros(sample_size, n_hidden)  # 第一层隐藏状态初值为全0
c2_0 = torch.zeros(sample_size, n_hidden)      # 第二层记忆状态初值为全0
h2_0 = torch.zeros(sample_size, n_hidden)      # 第二层隐藏状态初值为全0
c1 = c1_0     # 第一层记忆状态初始化
h1 = h1_0     # 第一层隐藏状态初始化
c2 = c2_0     # 第二层记忆状态初始化
h2 = h2_0     # 第二层隐藏状态初始化
```

```
for x in X:
    # 第一层拼接形成[ h[t-1], x[t] ]
    catenate1 = torch.cat([h1, x], dim=1)    #dim=1表示按行拼接,catenate1: [batch_size, n_hidden+emb_size]
    # 第一层遗忘门计算
    f1 = self.sigmoid(self.W1_f(catenate1))
    # 第一层输入门计算
    i1 = self.sigmoid(self.W1_i(catenate1))
    cc1 = self.tanh(self.W1_c(catenate1))
    # 第一层记忆更新
    c1 = torch.mul(f1, c1) + torch.mul(i1, cc1)
    # 第一层输出门计算
    o1 = self.sigmoid(self.W1_o(catenate1))
    h1 = torch.mul(o1, self.tanh(c1))
    # 第二层拼接
    catenate2 = torch.cat([h2, h1], dim=1)  # dim=1表示按行拼接,catenate: [batch_size, n_hidden+emb_size]
    # 第二层遗忘门计算
    f2 = self.sigmoid(self.W2_f(catenate2))
    # 第二层输入门计算
    i2 = self.sigmoid(self.W2_i(catenate2))
    cc2 = self.tanh(self.W2_c(catenate2))
    # 第二层记忆更新
    c2 = torch.mul(f2, c2) + torch.mul(i2, cc2)
    # 第二层输出门计算
    o2 = self.sigmoid(self.W2_o(catenate2))
    h2 = torch.mul(o2, self.tanh(c2))
# model_output = nn.functional.softmax(self.W(h2), dim=1)     # dim=1, 对行作归一化
model_output = self.W(h2)
```

二、实验结果

2.1 单层 LSTM

```
epoch: 0001 lost = 8.431622 ppl = 4589.94
epoch: 0002 lost = 7.556726 ppl = 1913.57
epoch: 0003 lost = 7.099878 ppl = 1211.82
epoch: 0004 lost = 6.851283 ppl = 945.092
epoch: 0005 lost = 6.730500 ppl = 837.566
epoch: 0006 lost = 6.679948 ppl = 796.277
epoch: 0007 lost = 6.661658 ppl = 781.846
epoch: 0008 lost = 6.655941 ppl = 777.389
epoch: 0009 lost = 6.653969 ppl = 775.858
epoch: 0010 lost = 6.654221 ppl = 776.054
epoch: 0011 lost = 6.650253 ppl = 772.98
epoch: 0012 lost = 6.649596 ppl = 772.472
epoch: 0013 lost = 6.649964 ppl = 772.757
epoch: 0014 lost = 6.647772 ppl = 771.064
epoch: 0015 lost = 6.647903 ppl = 771.165
```

2.2 双层 LSTM

```
epoch: 0001 lost = 8.251723 ppl = 3834.22
epoch: 0002 lost = 7.438223 ppl = 1699.73
epoch: 0003 lost = 7.036302 ppl = 1137.17
epoch: 0004 lost = 6.808244 ppl = 905.28
epoch: 0005 lost = 6.688860 ppl = 803.406
epoch: 0006 lost = 6.635687 ppl = 761.802
epoch: 0007 lost = 6.616769 ppl = 747.526
epoch: 0008 lost = 6.611986 ppl = 743.959
epoch: 0009 lost = 6.612097 ppl = 744.041
epoch: 0010 lost = 6.614106 ppl = 745.538
epoch: 0011 lost = 6.616861 ppl = 747.594
epoch: 0012 lost = 6.620127 ppl = 750.04
epoch: 0013 lost = 6.624042 ppl = 752.983
epoch: 0014 lost = 6.627780 ppl = 755.802
epoch: 0015 lost = 6.632382 ppl = 759.289
```

三、实验分析

3.1 维度分析

在实践课上听学长说要看懂参数的维度，当时不以为意。然而在我刚开始做实践作业 1 时，就因为维度问题头疼了很久。在本次大作业期间也出现过因为

tensor 的维度不匹配而报错，当即明白了——弄明白参数的维度很重要。

3.2 参数不宜过多

如果每层的参数都较大且层数较多，如将多个隐藏层输出维度设置成 n_class（7613），则训练时显存不够，无法训练。

3.3 模型输出层是否使用 Softmax

在训练时发现个有意思的现象——双层 LSTM 模型中，如果在模型的最终输出层上，先用线性变换拟合输出概率、再用 Softmax 归一化，那么 loss、ppl 会非常非常缓慢地减小；但如果只用线性变换去拟合而不用 Softmax，那么 loss、ppl 会先迅速减小、再缓慢增大。很遗憾目前不清楚原因。

3.4 单、双层 LSTM 结果对比

在不使用 Softmax 的情况下，单层 LSTM 训练时 loss、ppl 相对较大，但一直保持着减小的趋势；而双层 LSTM 虽然 loss、ppl 较小，但有先减小、后增大的趋势。

四、总结感悟

为期八周的 NLP 课程已经结束了，我很高兴能选到这样一门压力相对较小、又能学到理论知识、实践经验的好课。虽然刚开始搭建环境、接触 pytorch 框架的时候很痛苦，但经过实践课的学习，我对编程环境、pytorch 框架都更熟悉了，也对神经网络的参数、前向传播有了一定认识。很感谢细心的老师、助教学长们，考虑到包括我在内的许多学生初次接触 python 编程，而为我们减轻工作量。

总之，通过 NLP 课程我收获了很多，无论是肖老师细致入微的讲解，还是博士生学长们亲自指导的实践，都让我印象深刻。很庆幸能在这个团队的指导下学习 NLP 课程，也希望以后能有更多的机会接触这样的团队。

五、附录

源码 GitHub 地址：https://github.com/Sleepyhead1111/NLPwork

参考资料 1：LSTM-muyongyu.pptx
参考资料 2：Pytorch 官方文档
参考资料 3：深入浅出 LSTM