Summer Study Report

Xie Yuejin

Advanced Class 2201
Huazhong University of Science and Technology
u202210333@hust.edu.cn

2023 年 9 月 4 日

Xie Yuejin, Advanced Class 2201

- Weighted Grades/GPA: 92.17/4.65
- Rank: 1/28
- "Research" Experience:
  - College Student Entrepreneurship Program(Use Transformer to do channel estimation)
  - Study for a summer in Professor Wang Bang's lab.

This part of the study mainly refers to the book *Dive into Deep Learning* by Mu Li.
I learn the basic knowledge and several fundamental models first:

- Data Progress, Basci Maths knowledge for AI...
- GPU and CPU, How to use Pytorch
- Linear Model, Softmax, MLP, SVM, Decision Tree...

Just use the knowledge I learned, and successfully passed the first round of testing.

前言

在最开始的几天里，不太确定自己究竟想选择哪一个方向。在简单了解了几个方向的内容之后，我个人认为NLP和CV两个领域稍微好上手一点，最终选择了NLP方向。
这一部分的理论知识学习主要参考了李沐老师的*Dive into Deep Learning*，最开始的阻力还是比较大的，基本上听不太懂，了解完基本知识之后开始自己动手实现模型，在边做边学中才逐渐有了深一点的了解.

文本预处理 I

不同的模型模型预处理不太相同，调用的库函数:

- LSTM,TextCNN: Transforms库
  - 分词器使用$torchtext.data.get_tokenizer("basic_english")$ 忽略大小写，特殊字符等
  - $vocab = build_vocab_from_iterator(reviews_train, min_freq = 3, specials = [' < pad >', ' < unk >', ' < cls >', ' < sep >'])vocab.set_default_index(vocab[' < unk >'])$构建词汇表,并添加特殊字符
  - $transforms.VocabTransform(vocab = vocab)$将token转化为序号
  - $transforms.Truncate(max_seq_len = max_len)$截断，$transforms.ToTensor(padding_value = vocab[' < pad >'])$转化成tensor，并填充
  - $dataset = TensorDataset(text_transform(reviews), torch.tensor(labels))$返回数据集

文本预处理 II

- BERT, GPT:
  - 专用分词器BertTokenizer, GPT2Tokennizer
  - 添加特殊字符，如$< sep >, < cls >$等
  - 其他类似

LSTM

LSTM相较RNN，有更多可学习参数，模型更大，下面是LSTM模型图:

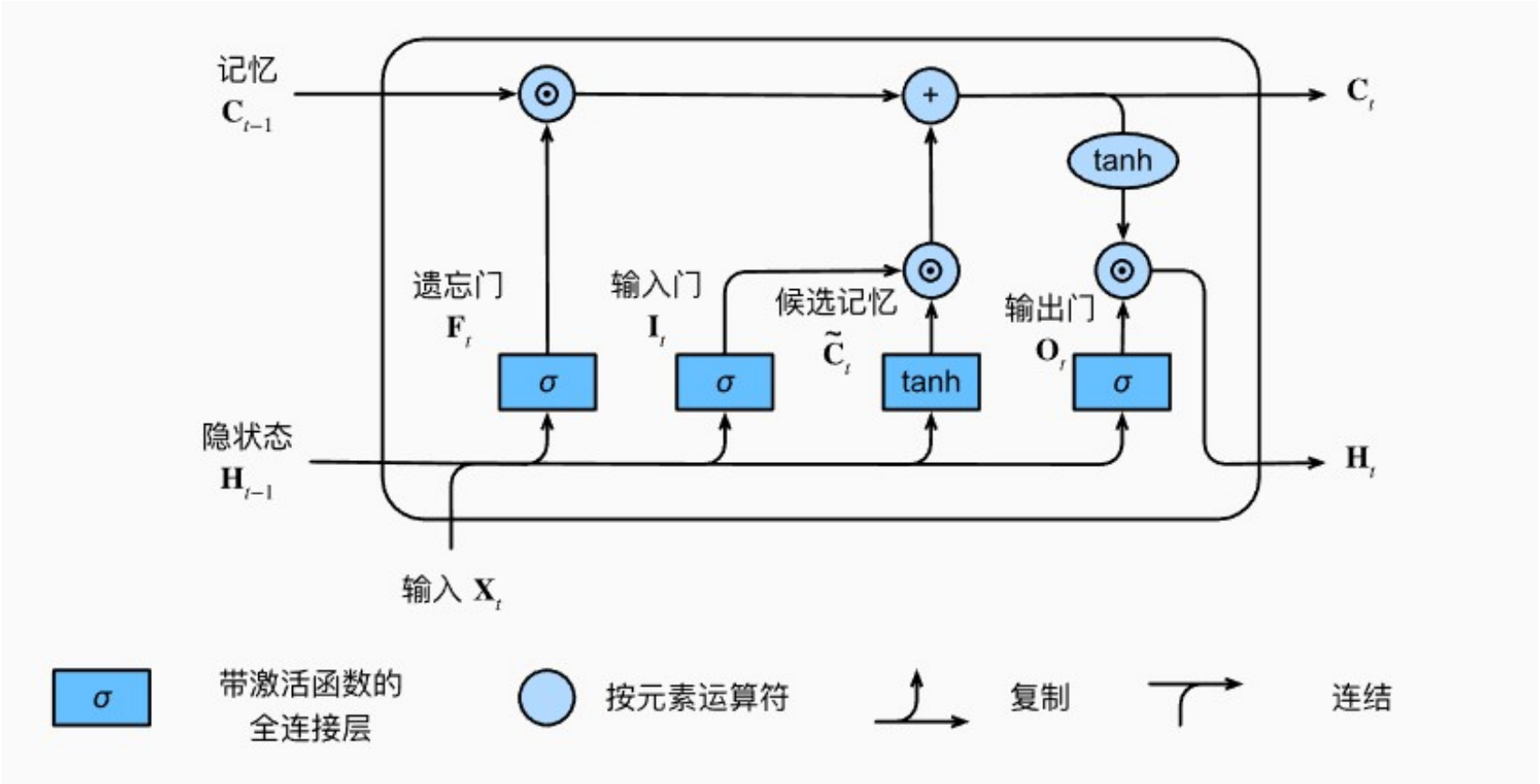

Figure: 1

LSTM I

- 首先是定义了输入门，遗忘门，输出门，以及候选记忆单元

$$
\begin{aligned}
\mathbf{I}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i),\\
\mathbf{F}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f),\\
\mathbf{O}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o),\\
\tilde{\mathbf{C}}_t &= \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c),
\end{aligned}
$$

其中$\sigma(x)$为*sigmoid*激活函数，这些计算实际上和RNN中隐状态的计算比较类似

LSTM II

- 下面我们定义记忆元

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t$$

$\mathbf{F}_t \odot \mathbf{C}_{t-1}$实际上表示了遗忘多少先前的记忆元$\mathbf{C}_{t-1}$，而$\mathbf{I}_t \odot \tilde{\mathbf{C}}_t$ 则代表了当前候选记忆元使用的程度

- *LSTM*中隐状态的定义如下：

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t)$$

这样的定义可以保证每个元素均在$[-1, 1]$之间，防止梯度爆炸

- LSTM层的计算基本是这样，我们取最后一层的隐状态加上Dense输出即可

LSTM III

代码实现如下(仅给出关键代码):

```
class LSTMModel(nn.Module):
    Xie Yuejin
    def __init__(self, vocab_size, num_hiddens, init_state, forward_fn=lstm_calculate,
                 embedded_size=100, *args, **kwargs):
        """Defined in :numref:`sec_rnn_scratch`"""
        super().__init__(*args, **kwargs)
        self.vocab_size, self.num_hiddens = vocab_size, num_hiddens
        self.W_xi, self.W_hi, self.b_i = three(embedded_size, num_hiddens)  # 输入门参数
        self.W_xf, self.W_hf, self.b_f = three(embedded_size, num_hiddens)  # 遗忘门参数
        self.W_xo, self.W_ho, self.b_o = three(embedded_size, num_hiddens)  # 输出门参数
        self.W_xc, self.W_hc, self.b_c = three(embedded_size, num_hiddens)  # 候选记忆元参数
        self.xavier_init()
        self.glove = GloVe(name="6B", dim=100)
        self.embedding = nn.Embedding(vocab_size, embedding_dim=100)
        self.init_state, self.forward_fn = init_state, forward_fn
        self.dense = nn.Linear(num_hiddens, 2)
    Xie Yuejin
    def __call__(self, inputs, state):
        inputs = self.embedding(inputs).transpose(0, 1)
        (H, C) = state
        for X in inputs:
            I = torch.sigmoid(torch.matmul(X, self.W_xi) + torch.matmul(H, self.W_hi) + self.b_i)
            F = torch.sigmoid(torch.matmul(X, self.W_xf) + torch.matmul(H, self.W_hf) + self.b_f)
            O = torch.sigmoid(torch.matmul(X, self.W_xo) + torch.matmul(H, self.W_ho) + self.b_o)
            C_tilda = torch.tanh(torch.matmul(X, self.W_xc) + torch.matmul(H, self.W_hc) + self.b_c)
            C = F * C + I * C_tilda
            H = O * torch.tanh(C)
        return self.dense(H)
```

Figure: 2

LSTM IV

结果：

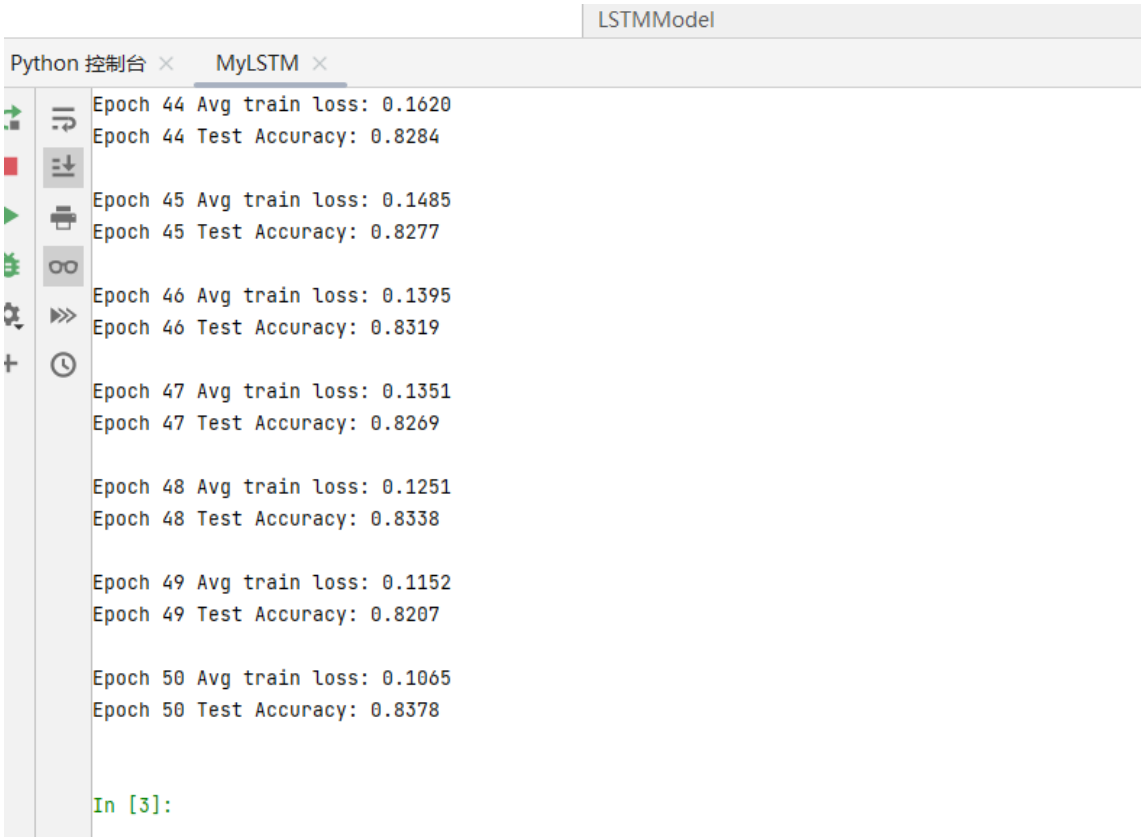

Figure: 3

LSTM V

遇到问题/瑕疵:

- 最开始的时候参数矩阵使用了零初始化，导致了梯度消失，损失一直降不下去(改用xavier初始化解决)
- 没能实现多层LSTM，以及双向LSTM，使得效果不如框架实现的LSTM

TextCNN I

TextCNN的实现总体来说比较轻松，模型图如下：

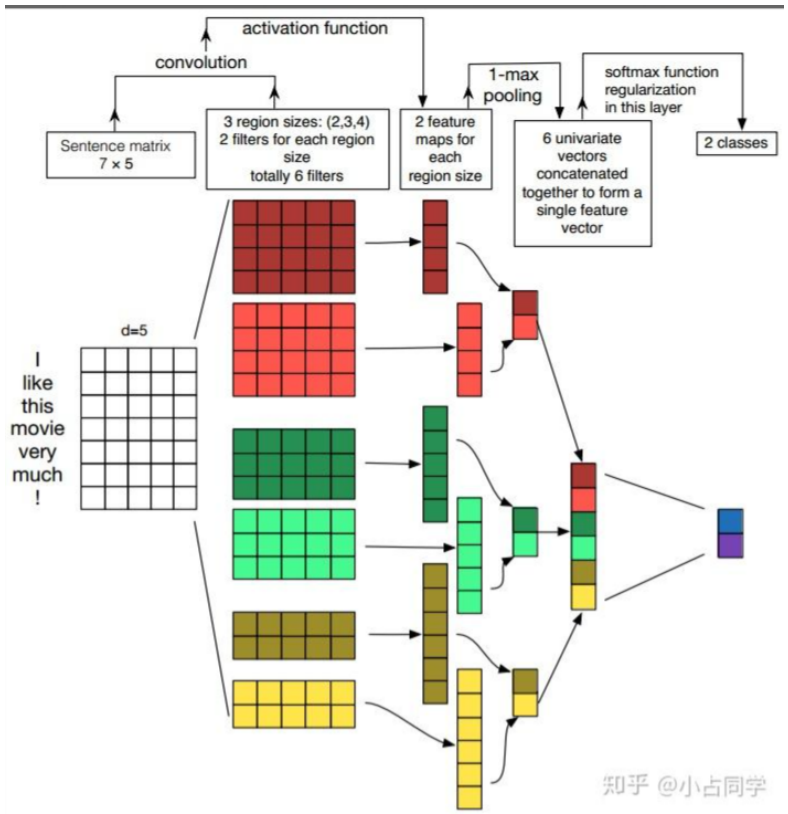

Figure: 4

TextCNN II

- 第一层：输入层
  每个词向量可以是预先在其他语料库中训练好的，也可以作为未知的参数由网络训练得到。预先训练的词嵌入可以利用其他语料库得到更多的先验知识，而由当前网络训练的词向量能够更好地抓住与当前任务相关联的特征。因此，图中的输入层实际采用了双通道的形式，即有两个 $N \times k$ 的输入矩阵，其中一个用预训练好的词嵌入表达，并且在训练过程中不再发生变化；另外一个也由同样的方式初始化，但是会作为参数，随着网络的训练过程发生改变
- 第二层:卷积层
  把$embedded_size$当作通道数，对每个通道进行一维卷积，通道求和得到输出，我们可以得到多个输出通道以提升模型的复杂度，让模型提取不同的语义信息

TextCNN III

- 第三层：池化连接层
  将上一步的输出用最大池化，再进行连接，加上dense层输出二分类的概率即可