

Summer Study Report

Xie Yuejin

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

2023 年 9 月 4 日

- 1 Self Introduction
- 2 First Round of Testing
- 3 Second Round of Testing I
 - 前言
 - 文本预处理
 - LSTM
 - TextCNN
- 4 Second Round of Testing II
 - BERT从0实现
 - Fine-tune:BERT,GPT2

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, Basic 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：

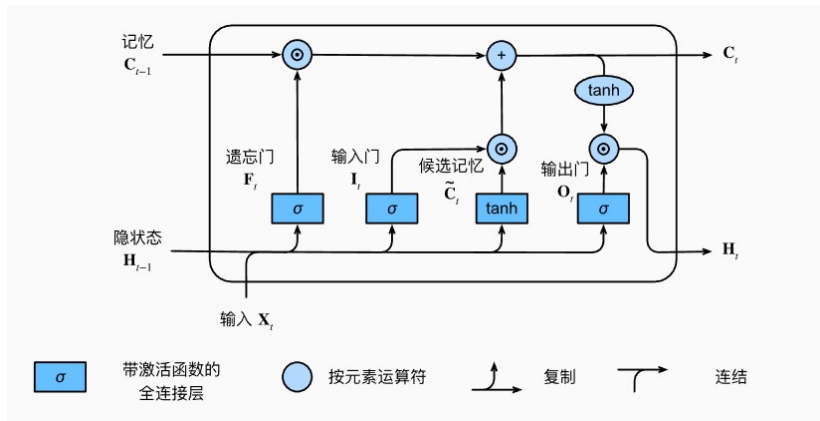


Figure: 1

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

$$\mathbf{I}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i),$$

$$\mathbf{F}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f),$$

$$\mathbf{O}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),$$

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c),$$

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
- embedding 层使用了 100 维的预训练词向量 Glove

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_hidden, 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_hidden = vocab_size, num_hidden
        self.W_xi, self.W_hi, self.b_i = three(embedded_size, num_hidden) # 输入门参数
        self.W_xf, self.W_hf, self.b_f = three(embedded_size, num_hidden) # 遗忘门参数
        self.W_xo, self.W_ho, self.b_o = three(embedded_size, num_hidden) # 输出门参数
        self.W_xc, self.W_hc, self.b_c = three(embedded_size, num_hidden) # 候选记忆元参数
        self.xavier_init()
        self.glove = GloVe(name="68", 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_hidden, 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:

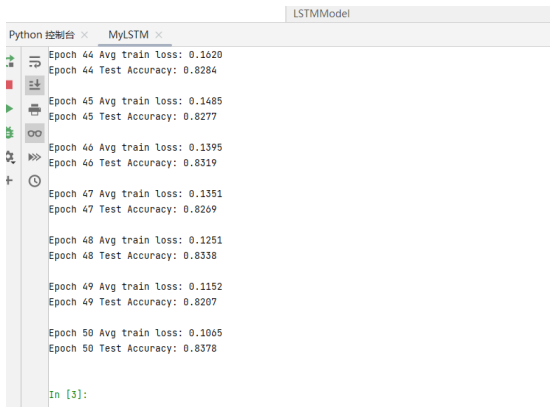


Figure: 3

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：

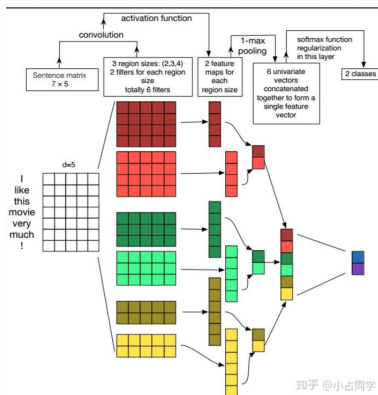


Figure: 4

TextCNN II

- 第一层：输入层输入层采用了双通道的形式，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frozen；另外一个使用nn.Embedding定义，训练过程发生改变

```
self.embedding_constant = nn.Embedding(len(vocab), 100, padding_idx=vocab['<pad>'])  
self.glove = GloVe(name="6B", dim=100)  
self.embedding_changing = nn.Embedding.from_pretrained(self.glove.get_vecs_by_tokens(vocab.get_itos()),  
                                                         padding_idx=vocab['<pad>'],  
                                                         freeze=True)
```

Figure: 5

TextCNN III

- 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,可得到多个输出通道,让模型提取不同信息,这里实际上应该是一维卷积,偷个懒使用二维卷积了,效果一样

```
self.conv_constant = nn.ModuleList()
self.conv_changing = nn.ModuleList()
for out_channels, kernel_size in zip(num_channels, kernel_sizes):
    self.conv_constant.append(
        nn.Conv2d(in_channels=1, out_channels=out_channels, kernel_size=(kernel_size, embed_size)))
    self.conv_changing.append(
        nn.Conv2d(in_channels=1, out_channels=out_channels, kernel_size=(kernel_size, embed_size)))
```

Figure: 6

TextCNN IV

- 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

```
self.pool = nn.AdaptiveMaxPool1d(1)
self.relu = nn.ReLU()
self.dropout = nn.Dropout(0.5)
self.fc = nn.Linear(sum(num_channels) * 2, 2)
self.apply(_init_weights)
```

Figure: 7

TextCNN V

结果:

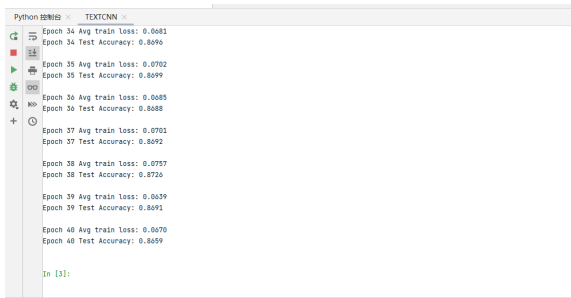


Figure: 8

TextCNN训练速度很快, 效果也还不错, 总体上没有遇到什么问题

BERT从0实现 I

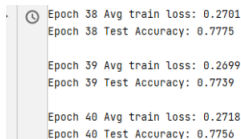
BERT从0自己实现难度很大，所以主要参考了transformers库中关于BERT的实现主要实现了以下类：

- class BertConfig(object) 包含了BERT模型所需要的大部分参数
- class BertEmbeddings(nn.Module) 实际上就是词嵌入+位置编码+*Segment Embedding*
- class BertAttention(nn.Module) 实际上就是Transformer的Encoder的多头注意力机制
- class BertLayer(nn.Module) 整合bertAttention和两个全连接层

BERT从0实现 II

- class BertEncoder(nn.Module) BertLayer层堆积
- class BertPooler(nn.Module) 对最后一层输出的隐状态全局平均池化,再增加一个线性层映射到二分类
- class BertModel(nn.Module) 整合所有部件生成BERT模型类

训练结果:



```
Epoch 38 Avg train loss: 0.2701  
Epoch 38 Test Accuracy: 0.7775  
  
Epoch 39 Avg train loss: 0.2699  
Epoch 39 Test Accuracy: 0.7739  
  
Epoch 40 Avg train loss: 0.2718  
Epoch 40 Test Accuracy: 0.7756
```

Figure: 9

效果确实比较一般

BERT从0实现 III

遇到的问题/瑕疵:

- 最开始时损失完全降不下去, 后使用学习率调度器使得损失有所下降(应该是模型跑飞了)
- 准确率比较差, 调不出来一个比较好的参数

Fine-tune I

这一部分总体上就比较简单了，调库就行了

- BERT:

```
Epoch 1:
Epoch 1 Avg train loss: 0.5392
Epoch 1 Test Accuracy: 0.8350

Epoch 2:
Epoch 2 Avg train loss: 0.3384
Epoch 2 Test Accuracy: 0.7749

Epoch 3:
Epoch 3 Avg train loss: 0.2533
Epoch 3 Test Accuracy: 0.8586

Epoch 4:
Epoch 4 Avg train loss: 0.1995
Epoch 4 Test Accuracy: 0.8575

Epoch 5:
Epoch 5 Avg train loss: 0.1553
Epoch 5 Test Accuracy: 0.8353

Epoch 6:
Epoch 6 Avg train loss: 0.1331
Epoch 6 Test Accuracy: 0.8486

Epoch 7:
Epoch 7 Avg train loss: 0.1059
Epoch 7 Test Accuracy: 0.8427

Epoch 8:
Epoch 8 Avg train loss: 0.0858
Epoch 8 Test Accuracy: 0.8315

Epoch 9:
Epoch 9 Avg train loss: 0.0605
Epoch 9 Test Accuracy: 0.8577

Epoch 10:
Epoch 10 Avg train loss: 0.0512
Epoch 10 Test Accuracy: 0.8542
```

Fine-tune II

- GPT2:

```
Epoch 1:  
Epoch 1 Avg train loss: 0.2928  
Epoch 1 Test Accuracy: 0.9552  
  
Epoch 2:  
Epoch 2 Avg train loss: 0.1787  
Epoch 2 Test Accuracy: 0.9738  
  
Epoch 3:  
Epoch 3 Avg train loss: 0.1247  
Epoch 3 Test Accuracy: 0.9891  
  
Epoch 4:  
Epoch 4 Avg train loss: 0.0794  
Epoch 4 Test Accuracy: 0.9918  
  
Epoch 5:  
Epoch 5 Avg train loss: 0.0515  
Epoch 5 Test Accuracy: 0.9962  
  
Epoch 6:  
Epoch 6 Avg train loss: 0.0388  
Epoch 6 Test Accuracy: 0.9976  
  
Epoch 7:  
Epoch 7 Avg train loss: 0.0276  
Epoch 7 Test Accuracy: 0.9979  
  
Epoch 8:  
Epoch 8 Avg train loss: 0.0223  
Epoch 8 Test Accuracy: 0.9985  
  
Epoch 9:  
Epoch 9 Avg train loss: 0.0150  
Epoch 9 Test Accuracy: 0.9988  
  
Epoch 10:  
Epoch 10 Avg train loss: 0.0098  
Epoch 10 Test Accuracy: 0.9988
```

Figure: 11

GPT2的效果出奇的好，第一个epoch准确率就到了95%，后面准确率甚至达到了99%!?

Fine-tune III

- BERT和GPT的比较
 - BERT是Transformer编码器，GPT是Transformer解码器
 - 预训练差距比较大，BERT是做完形填空，GPT在做预测未来（标准的语言模型），后者显然要更难一些，这可能也是GPT的效果要比BERT差一些的原因之一吧
 - 目标任务不太一样,BERT主要做的还是主要用于抽取特征(?),而GPT主要还是在做文本生成

Thank You!

Xie Yuejin
Advanced Class 2201