Lab 04 Report

实验内容

1. 基于pytorch编写RNN（LSTM）
2. 训练(fine-tune) hugging face的预训练模型BERT.

基于训练好的词向量， 分别用这两个模型实现文本情感分类（Text Sentiment Classification） ： 输入一个句子， 输出是 0（负面） 或 1（正面），并对两个不同模型的实验结果展开对比分析。

实验原理

1. BERT

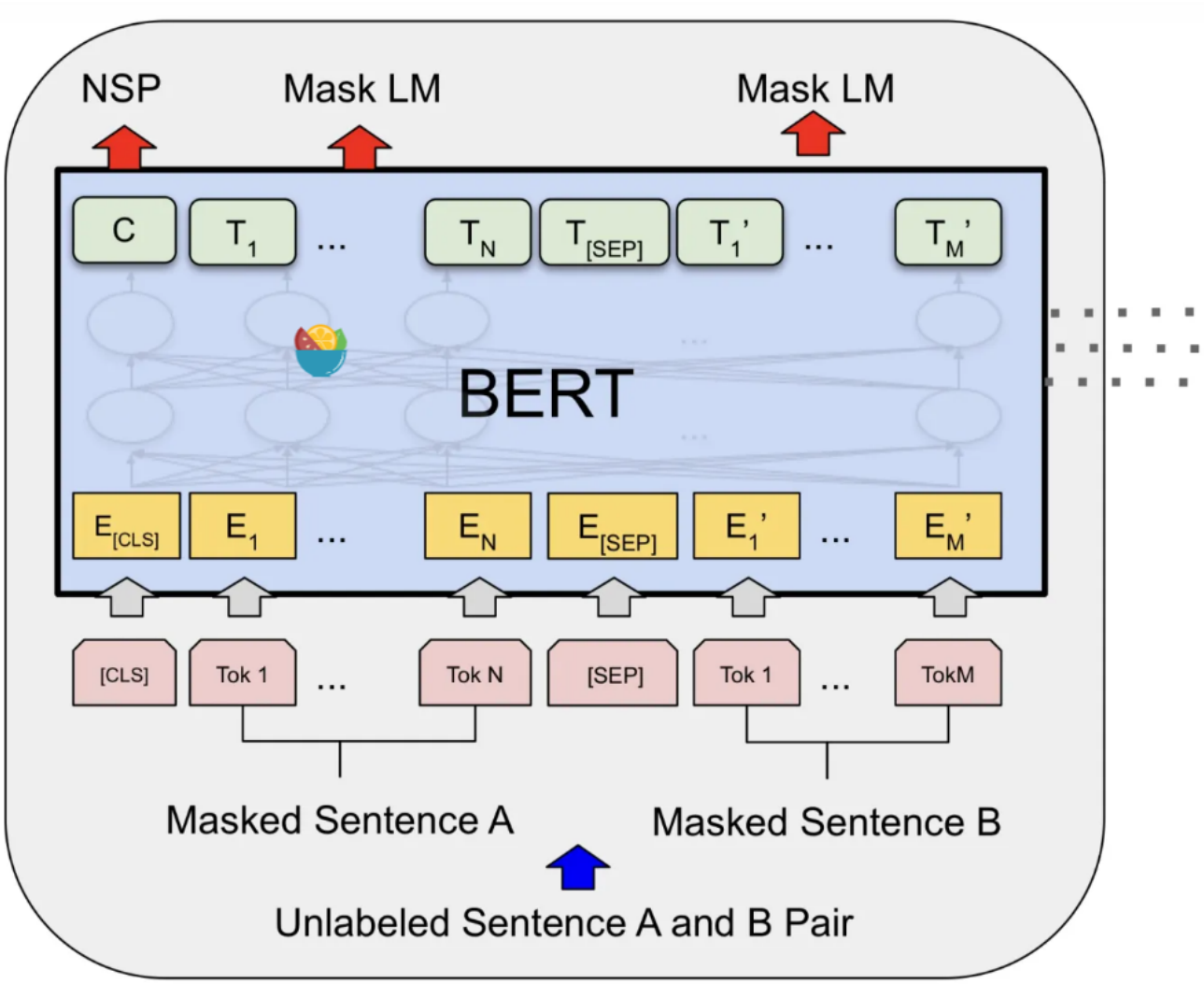
BERT（Bidirectional Encoder Representations from Transformers）是一种预训练语言模型（pretrained language model）, 由Google在2018年提出. 它是基于Transformer模型的, 并且使用了 Transformer模型中的自注意力机制, 通过预训练学习大量数据的上下文信息, 可以用于各种自然语言处理任务, 例如文本分类、命名实体识别、问答系统等.

BERT的预训练过程包含两个阶段. 首先, 使用大型语料库对模型进行无监督的预训练, 让模型学习语言的一般特性和上下文信息. 其次, 通过在特定任务上进行微调, 让模型适应具体任务的特征和需求, 从而提高模型的性能.

在预训练阶段中, 使用了两种技术: Masked Language Model（MLM）和Next Sentence Prediction（NSP）. MLM技术通过在输入的句子中随机遮盖一些单词, 然后让模型预测这些被遮盖的单词是什么, 从而鼓励模型学习句子中单词之间的上下文关系. NSP技术则是让模型预测两个句子是否是连续的, 以帮助模型学习句子之间的关系和逻辑.

预训练后, 只需要添加一个额外的输出层进行fine-tune, 就可以在各种各样的下游任务中取得不错的效果.

bert结构：



2. RNN（LSTM）

长短时记忆网络（Long Short-Term Memory, LSTM）是一种特殊的循环神经网络（Recurrent Neural Network, RNN），用于解决RNN在处理长序列时遇到的梯度消失和梯度爆炸问题。LSTM通过引入多个门（gate）结构和一个记忆单元，使得网络能够有选择地记住或忘记信息，从而有效地处理长序列依赖关系.

LSTM

LSTM单元包括以下几个部分：

1. 输入门（Input gate）：决定如何更新记忆单元的信息。

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

其中，W_{xi}和W_{hi}是权重矩阵，b_i是偏置向量，σ是sigmoid激活函数。

2. 遗忘门（Forget gate）：决定如何遗忘记忆单元的旧信息。

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$

3. 输出门（Output gate）：决定如何根据记忆单元的信息计算输出。

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$

4. 记忆单元（Memory cell）：存储长期信息。

$$\tilde{C}t = \tanh(Wxcx_t + W_{hc}h_{t-1} + b_c)$$

然后，我们可以使用输入门、遗忘门和新的候选状态来更新记忆单元：

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

其中，⊙表示逐元素相乘

5. 隐藏状态更新：

$$h_t = o_t \odot \tanh(C_t)$$

实验步骤

1.数据读取、编码与封装

- 数据读取：读取原始数据并用`pandas`库转换为`dataframe`类型，方便读取
- 数据编码：用`transformer`库`BertTokenizer`函数对数据进行编码。为了更好地对比两种方法，在RNN实现中我们也使用了这种编码方式，不同的是，RNN中我们不需要使用数据的`attention_mask`信息。
- 数据封装：用`torch.utils.data`中`DataLoader`对数据封装。

下面展示了数据编码类`SentimentDataset`：

```
class SentimentDataset(Dataset):
    def __init__(self, dataframe, tokenizer, max_length):
        self.dataframe = dataframe
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.dataframe)

    def __getitem__(self, idx):
        row = self.dataframe.iloc[idx]
        text = row['text']
        label = row['target']
        #编码
        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
            max_length=self.max_length,
            return_token_type_ids=False,
            padding='max_length',
            truncation=True,
            return_attention_mask=True,
            return_tensors='pt',
        )

        #编码
        return {
            'input_ids': encoding['input_ids'].flatten(), #输入序列
            'attention_mask': encoding['attention_mask'].flatten(), #注意力掩码
```

```
            'labels': torch.tensor(label, dtype=torch.long)
        }
```

说明：

1. text：输入文本，需要进行编码的字符串。
2. add_special_tokens：布尔值，表示是否在文本中添加特殊的开始和结束标记（例如，[CLS]和[SEP]）。
3. max_length：整数，表示编码后的序列的最大长度。如果输入文本的长度超过此值，将被截断；如果长度小于此值，将使用填充符号进行填充。
4. return_token_type_ids：布尔值，表示是否返回token_type_ids。在这个例子中，我们不需要token_type_ids，所以设置为False。
5. padding：字符串，表示填充策略。在这个例子中，我们使用'max_length'策略，这意味着所有序列都将被填充或截断到max_length的长度。
6. truncation：布尔值，表示是否对超过max_length的序列进行截断。
7. return_attention_mask：布尔值，表示是否返回注意力掩码。注意力掩码用于区分输入序列中的实际单词和填充符号。
8. return_tensors：字符串，表示返回的张量类型。在这个例子中，我们使用'pt'，表示返回PyTorch张量。

2.BERT：fine-tune和测试

- 加载预训练模型（huggingface）
- 基于数据集的`input_ids`,`attention_mask`,`labels`在训练集上对BERT进行fine-tune
- 在测试集上测试模型。

下面展示了训练过程：

```
num_epochs = 8
device = 'cuda'
model = bert_model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=2e-5)
for epoch in range(num_epochs):
    print("epoch:",epoch)
    count = 0
    for batch in train_dataloader:
        count+=1
        if count%78 == 0:
          print(count/391) #打印训练进度
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)
        outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
        loss = outputs.loss
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

3. 构建RNN模型类

```
class RNN_LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers):
        super(RNN_LSTM, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size, num_layers,
batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        embedded = self.embedding(x)
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)

        out, _ = self.lstm(embedded, (h0, c0))
        out = self.fc(out[:, -1, :])
        return out
```

- embedding：嵌入层，用于将输入数据编码成低维向量。
- lstm：LSTM 层，用于处理序列数据并提取特征。
- fc：全连接层，用于将 LSTM 层的输出映射到输出维度。

4. RNN模型训练、测试

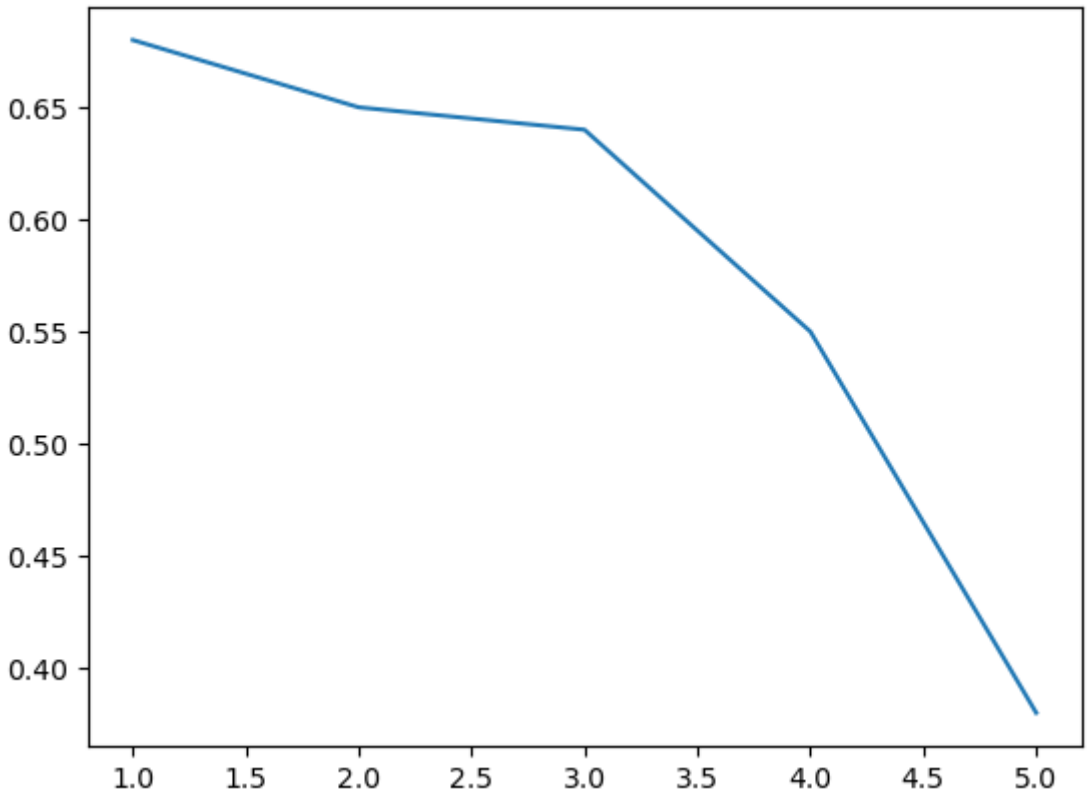
RNN模型的训练、测试和BERT类似，不同之处在于不会将注意力掩码作为输入。

5.参数调节

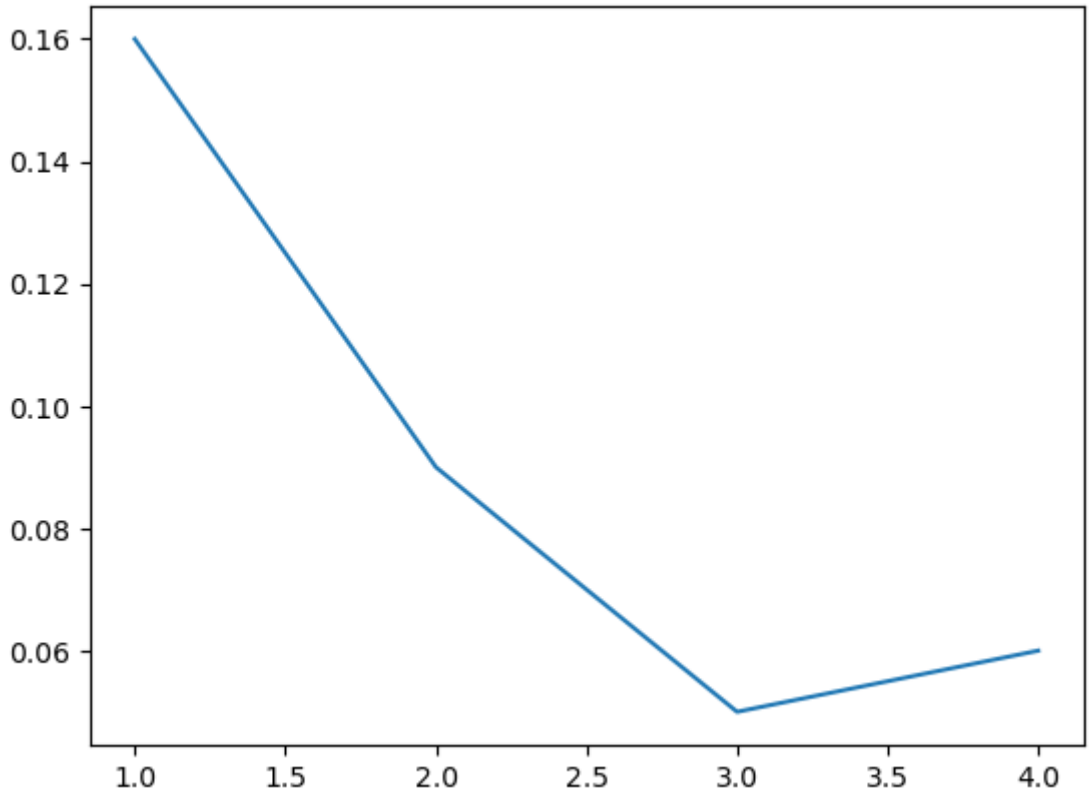
由于BERT是预训练模型，RNN的结构也相对固定，故本实验参数调节较为简单。BERT模型调节了学习率，RNN调节了隐藏层数。

实验结果

1.训练过程损失



- RNN



- BERT

2.测试集正确率

- RNN: 81%
- BERT：91%

结果分析

在结果中我们可以看到，BERT以更少的迭代次数取得了比RNN(LSTM)更好的效果。结合这两个模型的性质，我觉得可能有以下原因：

- BERT对长文本的处理更加高效：在RNN中，由于需要在每一个时间步进行循环计算，所以处理长序列时会面临梯度消失和梯度爆炸的问题，导致模型难以训练和优化。而BERT模型采用了自注意力机制，可以同时考虑整个文本的信息，从而避免了这个问题，使其在处理长文本时更加高效。
- BERT有更好的表示能力：BERT模型使用了双向Transformer结构（注意力机制），使得模型能够同时考虑文本中的上下文信息，从而在表示文本时更加准确和全面。而RNN只能单向处理序列，因此对于某些任务，它可能会丢失一些重要的上下文信息。
- BERT有更好的泛化能力：BERT模型通过在大规模的语料库上进行预训练，使得模型具有更好的泛化能力，可以适应不同的任务和领域。而RNN（LSTM）需要在每个任务上进行训练，因此可能会出现过拟合的情况，导致模型泛化能力弱。
- BERT能够处理全局信息：BERT模型能够同时考虑文本的全局信息，因为它使用了自注意力机制，可以通过学习得到不同位置之间的关联性，从而更好的捕捉文本的语义信息。而RNN（LSTM）只能在局部范围内捕捉信息，这在某些任务上可能会限制其表现。
