lab3：循环神经网络进行文本情感分类

姓名：李晖茜
学号：SA22218131

一、实验要求

使用 pytorch 或者 tensorflow 的相关神经网络库，编写 BERT 的语言模型，并基于训练好的词向量，利用少量的训练数据，微调 BERT 模型用于与实验二相同的文本分类任务，并和实验二的 RNN 模型进行对比分析。
具体来说，在本次实验中，需要通过预训练后的 BERT 模型在数据集上微调后实现文本情感分类(Text Sentiment Classification)：输入一个句子，输出是 0(负面)或 1(正面)。

二、 实验环境

torch 1.10.0+cu111
torchtext 0.6.0
transformers 4.18.0

三、实验过程

3.1 数据

使用斯坦福的IMDb数据集（Stanford's Large Movie Review Dataset）作为文本情感分类的数据集。这个数据集分为训练和测试用的两个数据集，分别包含25,000条从IMDb下载的关于电影的评论。在每个数据集中，标签为"正面"和"负面"的评论数量相等。
BERT是一个预训练模型，它是在大量数据集上进行了预训练后，才被应用到各类NLP任务中。在对BERT模型进行预训练时，才能送入到模型中。而在将文本数据输入到BERT前，会使用到以下3个Embedding层：Token embedding，Segment embedding，Position embedding。在将输入序列经过上述3层embedding处理后，将每层embedding的结果进行相加，即得到了输入数据的最终表示，也就是BERT模型的输入。
首先加载预先训练的bert-base-uncased标记器。

```
# 加载预先训练的bert-base-uncased标记器
tokenizer = BertTokenizer.from_pretrained('/home/lhq/hw/lab3/bert-base-uncased')
# print(len(tokenizer.vocab))

init_token_idx = tokenizer.cls_token_id
eos_token_idx = tokenizer.sep_token_id
pad_token_idx = tokenizer.pad_token_id
unk_token_idx = tokenizer.unk_token_id
# print(init_token_idx, eos_token_idx, pad_token_idx, unk_token_idx)

max_input_length = tokenizer.max_model_input_sizes['bert-base-uncased']
# print(max_input_length)

# 最大长度比实际的最大长度小2。因为需要向每个序列添加两个标记，一个开始一个结束。
def tokenize_and_cut(sentence):
    tokens = tokenizer.tokenize(sentence)
    tokens = tokens[ :max_input_length-2]
    return tokens
```

接下来定义字段：

```
# 定义字段
TEXT = data.Field(
        batch_first=True,
        use_vocab=False,
        tokenize=tokenize_and_cut,
        preprocessing=tokenizer.convert_tokens_to_ids,
        init_token=init_token_idx,
        eos_token=eos_token_idx,
        unk_token=unk_token_idx,
        pad_token=pad_token_idx
)
LABEL = data.LabelField(dtype=torch.float)
```

使用torchtext.datasets加载数据并划分验证集：

```
# 加载数据并创建验证分割
train_data, test_data = datasets.IMDB.splits(
        text_field=TEXT,
        label_field=LABEL
)
train_data, valid_data = train_data.split(random_state = random.seed(SEED))
print(f'Number of training examples: {len(train_data)}')
print(f'Number of validation examples: {len(valid_data)}')
print(f'Number of testing examples: {len(test_data)}')
```

Number of training examples: 17500
Number of validation examples: 7500
Number of testing examples: 25000
虽然已经处理了文本的词汇表，但是仍然需要为标签构建词汇表。然后创建数据迭代器。由于GPU内存限制，设置超参数`BATCH_SIZE = 2`。

```
# 为标签构建词汇表
LABEL.build_vocab(train_data)
# print(LABEL.vocab.stoi)

# 创建数据迭代器
BATCH_SIZE = 2 # 128
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device)
```

3.2 网络搭建

接下来进行网络的搭建，加载预先训练过的模型，确保加载与标记器相同的模型。

```
# 加载预先训练过的模型
bert = BertModel.from_pretrained('/home/lhq/hw/lab3/bert-base-uncased')
```

接下来，定义实际的模型。使用预先训练的transformer模型，然后，这些嵌入信息将被输入GRU，对输入句子的情感进行预测。通过transformer的config属性获得嵌入维度大小，即称为hidden_size。在前向传递中，将transformer封装在no_grad中，以确保模型的这一部分不计算梯度。transformer实际上返回整个序列的嵌入以及一个汇集（pooled）的输出。在最后的时间步中获取隐藏状态，并将其通过线性层获得预测。

```
# 定义实际的模型
class BERTGRUSentiment(nn.Module):
    def __init__(self, bert, hidden_dim, output_dim, n_layers, bidirectional,
dropout):
        super(BERTGRUSentiment, self).__init__()
        self.bert = bert
        embedding_dim = bert.config.to_dict()['hidden_size']
        self.rnn = nn.GRU(embedding_dim, hidden_dim, num_layers=n_layers,
bidirectional=bidirectional,
                          batch_first=True, dropout= 0 if n_layers < 2 else
dropout)

        self.out = nn.Linear(hidden_dim * 2 if bidirectional else  hidden_dim,
output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, text):
        # text = [batch_size, sent_len]
        with torch.no_grad():
            embedded = self.bert(text)[0]
        # embedded = [batch_size, sent_len, emb_dim]
        _, hidden = self.rnn(embedded)
        # hidden = [n_layers * n_directions, batch_size, emb_dim]
        if self.rnn.bidirectional:
            hidden = self.dropout(torch.cat((hidden[-2, :, :], hidden[-1, :,
:]), dim=1))
        else:
            hidden = self.dropout(hidden[-1, :, :])
        # hidden = [batch_size, hidden_dim]
        output = self.out(hidden)
        # output = [batch_size, output_dim]
        return output
```

3.3 网络训练

使用标准超参数创建模型的实例：

```
# 使用标准超参数创建模型的实例
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 2
BIDIRECTIONAL = True
DROPOUT = 0.5

model = BERTGRUSentiment(bert, HIDDEN_DIM, OUTPUT_DIM, N_LAYERS, BIDIRECTIONAL,
DROPOUT)
```

对于bert transformer模型的一部分参数，可以设置`requires_grad = False`来冻结它们。

```
# 冻结参数
for name, param in model.named_parameters():
    if name.startswith('bert'):
        param.requires_grad =  False
```

定义优化器、损失函数和计算准确率函数：

```
# 定义优化器和损失函数
optimizer = optim.Adam(model.parameters())
criterion = nn.BCEWithLogitsLoss()

model = model.to(device)
criterion = criterion.to(device)

# 计算准确率
def binary_accuracy(preds, y):
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float()
    acc = correct.sum() / len(correct)
    return acc
```

模型训练：

```
# 执行训练epoch
def train(model, iterator, optimizer, criterion):
    epoch_loss = 0
    epoch_acc = 0
    model.train()
    for batch in iterator:
        optimizer.zero_grad()
        predictions = model(batch.text).squeeze(1)
        loss = criterion(predictions, batch.label)
        acc = binary_accuracy(predictions, batch.label)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
        epoch_acc += acc.item()
    return epoch_loss / len(iterator), epoch_acc / len(iterator)

# 执行评估epoch
def evaluate(model, iterator, criterion):
    epoch_loss = 0
    epoch_acc = 0
    model.eval()
    with torch.no_grad():
        for batch in iterator:
            predictions = model(batch.text).squeeze(1)
            loss = criterion(predictions, batch.label)
            acc = binary_accuracy(predictions, batch.label)
            epoch_loss += loss.item()
            epoch_acc += acc.item()
    return epoch_loss / len(iterator), epoch_acc / len(iterator)

# 计算训练/评估epoch需要多长时间
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
```

```
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs

# 训练模型
N_EPOCHS = 5
best_valid_loss = float('inf')
for epoch in range(N_EPOCHS):
    start_time = time.time()
    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion)
    end_time = time.time()
    epoch_mins, epoch_secs = epoch_time(start_time, end_time)
    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'tut6-model.pt')

    print(f'Epoch: {epoch + 1:02} | Epoch Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc * 100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc * 100:.2f}%')
```

3.5 测试性能

测试时，加载训练时保存的验证集上损失最小的模型。

```
# 加载最佳验证损失的参数
model.load_state_dict(torch.load('tut6-model.pt'))
test_loss, test_acc = evaluate(model, test_iterator, criterion)
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

四、实验结果

冻结bert模型参数训练结果为：

```
Test Loss: 0.218 | Test Acc: 91.66%
```

不冻结bert模型参数训练结果为：

```
Test Loss: 0.206 | Test Acc: 91.86%
```

五、实验总结

根据实验2，使用训练好的循环神经网络测试的test_acc为0.731，远低于加载预训练的bert测试的结果，而且不冻结bert的部分参数进行训练得到的结果更好。经过此次实验，使用bert基于Attention的方法进行简单的文本分类任务，在提供的数据集上进行微调，取得了远好于RNN的结果，了解学习了bert的网络结构，也感受到了transformer的强大。