1 实验介绍

1.1 实验背景

作家风格是作家在作品中表现出来的独特的审美风貌。通过分析作品的写作风格来识别作者这一研究有很多应用，比如可以帮助人们鉴定某些存在争议的文学作品的作者、判断文章是否剽窃他人作品等。

在本次实验中，我们需要完成基于写作片段识别作者的任务。这其实就是一个文本分类的过程，即在给定的分类体系下，根据文本的内容自动地确定文本所关联的类别。

1.2 实验要求

(1). 建立深度神经网络模型，对一段文本信息进行检测识别出该文本对应的作者。

(2). 绘制深度神经网络模型图、绘制并分析学习曲线。

(3). 用准确率等指标对模型进行评估。

1.3 实验环境

使用基于 Python 分词库进行文本分词处理，使用 Numpy 库进行相关数值运算，使用 pytorch 框架建立深度学习模型。

2 数据预处理

2.1 数据集介绍

该数据集包含了 8438 个经典中国文学作品片段，对应文件分别以作家姓名的首字母大写命名。数据集中的作品片段分别取自 5 位作家的经典作品，分别是：

序号	中文名	英文名	文本片段个数
1	鲁迅	LX	1500 条
2	莫言	MY	2219 条
3	钱钟书	QZS	1419 条
4	王小波	WXB	1300 条
5	张爱玲	ZAL	2000 条

其中截取的片段长度在 100 200 个中文字符不等。数据集路径为“dataset/ + 作者名字首字母缩写”命名。

2.2 预处理

在做文本挖掘的时候，首先要做的预处理就是分词。

英文单词天然有空格隔开容易按照空格分词，但是也有时候需要把多个单词做为一个分词，比如一些名词如"New York"，需要做为一个词看待。而中文由于没有空格，分词就是一个需要专门去解决的问题了。

这里我们使用 jieba 包进行分词，同时使用 GloVe 预训练词向量 (glove.6B.100d) 初始化词向量。

```
def processing_data(data_path, split_ratio=0.7):
    """
    数据处理函数：读取指定路径的数据集，进行分词、标签映射，划分训练集和验证集，并生成数据迭代器。
    :param data_path: 数据集路径，包含多个作者的文本文件。
    :param split_ratio: 验证集划分的比例，默认值为 0.7（70% 训练集，30% 验证集）。
    :return: train_iter, val_iter, TEXT.vocab 分别是训练集迭代器、验证集迭代器和词汇表。
    """
    sentences = [] # 用于存储所有文本句子
    target = [] # 用于存储对应的标签（作者）

    # 定义作者名称（标签）到数值的映射关系
    labels = {'LX': 0, 'MY': 1, 'QZS': 2, 'WXB': 3, 'ZAL': 4}

    # 遍历数据路径下的所有文件
    files = os.listdir(data_path)
    for file in files:
        if not os.path.isdir(file): # 确保处理的是文件，而不是目录
            # 打开每个文件，并逐行读取内容
            f = open(data_path + "/" + file, 'r', encoding='UTF-8')
            for index, line in enumerate(f.readlines()):
                sentences.append(line.strip()) # 读取的句子，去除首尾空格
                target.append(labels[file[:-4]]) # 根据文件名确定对应的作者标签

    # 将句子和标签配对为一个列表，方便后续处理
    mydata = list(zip(sentences, target))

```

```
    # 定义文本字段和标签字段
    # TEXT 字段表示输入的句子，使用结巴分词 (jieba.lcut)，并转为小写
    TEXT = Field(sequential=True, tokenize=lambda x: jb.lcut(x),
                 lower=True, use_vocab=True)
    # LABEL 字段表示标签，直接使用数值，不需要词典化
    LABEL = Field(sequential=False, use_vocab=False)

    # 定义数据字段结构：句子 -> 'text'，标签 -> 'category'
    FIELDS = [('text', TEXT), ('category', LABEL)]

    # 使用从句子和标签构造的 mydata 列表，生成 torchtext 的 Example 对象
    examples = list(map(lambda x: Example.fromlist(list(x), fields=FIELDS
        ),
                        mydata))

    # 创建数据集对象，包含所有样本
    dataset = Dataset(examples, fields=FIELDS)

    # 构建词汇表，并加载预训练的 GloVe 词向量（100 维）
    TEXT.build_vocab(dataset, vectors='glove.6B.100d')

    # 按指定比例划分数据集为训练集和验证集
    train, val = dataset.split(split_ratio=split_ratio)

    # 使用 BucketIterator 创建批次迭代器，可以根据句子长度动态调整批次
    train_iter, val_iter = BucketIterator.splits(
        (train, val), # 传入训练集和验证集
        batch_sizes=(16, 16), # 批量大小设置为 16
        device=device, # 如果使用 GPU，设置为对应 GPU 编号；否则为 -1 (CPU)
        sort_key=lambda x: len(x.text), # 按文本长度排序，优化效率
        sort_within_batch=False, # 不在批次内进行排序
        repeat=False # 禁止重复迭代
    )

    # 返回训练集迭代器、验证集迭代器和构建的词汇表
    return train_iter, val_iter, TEXT.vocab
```

3 模型搭建

3.1 初始网络

3.1.1 网络结构

在本项目中，我们首先尝试了教程中给出的简单的神经网络结构，用于文本分类任务。该网络的核心结构包括一个 LSTM 层以及两层全连接层。其主要设计如下：

(1). **LSTM 层**：网络首先通过一个单层 LSTM 单元进行特征提取，输入维度为 1，隐藏层维度为 64。该层能够捕获输入序列的时间依赖特性，生成高维语义表示。

(2). **全连接层 1**：从 LSTM 输出中提取的特征接入到第一层全连接网络中，隐藏单元数为 128，用于进一步提取全局特征。

(3). **全连接层 2**：最后接入一个输出层，输出维度为 5，对应五个类别的分类任务。

网络结构的详细代码实现如下：

```
1  class Net(nn.Module):
2      def __init__(self):
3          super(Net, self).__init__()
4          self.lstm = torch.nn.LSTM(1, 64) # LSTM层，输入维度1，隐藏层维度64
5          self.fc1 = nn.Linear(64, 128) # 全连接层1，输入维度64，输出维度128
6          self.fc2 = nn.Linear(128, 5) # 全连接层2，输入维度128，输出维度5
7
8      def forward(self, x):
9          """
10         前向传播
11         :param x: 模型输入
12         :return: 模型输出
13         """
14         output, hidden = self.lstm(x.unsqueeze(2).float()) # 添加通道维度
               并传入LSTM
15         h_n = hidden[1] # 获取最终的隐藏状态
16         out = self.fc2(self.fc1(h_n.view(h_n.shape[1], -1))) # 连接两层全
               连接层
17         return out
```

3.1.2 模型特点

- **层次简单**：初始网络仅包含一个 `LSTM` 层和两层全连接层，层次较浅，便于快速训练和验证。
- **高效训练**：网络结构设计紧凑，通过隐藏状态 `hidden` 提取 LSTM 的最后输出，无需保留全部时间步的输出，提高了计算效率。
- **适应小规模数据集**：该模型适合小规模数据集的实验验证，能够快速收敛，为后续优化提供参考基线。

3.2 改进后的网络结构

由于原网络结构的预测效果不佳 (预测效果将在下一节给出)，针对初始网络的不足之处，我们提出了一种基于 **双向 LSTM（BiLSTM）** 和 **注意力机制（Attention）** 的改进网络架构。改进后的网络能够更好地捕获句子全局和局部的语义特征，尤其是在分类任务中具有较强的表现能力。

3.2.1 模型结构设计

改进后的网络主要由以下几部分组成：

(1). **嵌入层（Embedding Layer）**：利用预训练的词向量将输入的词序号映射为词向量表示，维度为 `embedding_dim`。为了增强泛化能力，嵌入层后添加了 Dropout 机制。

(2). **双向 LSTM 层(BiLSTM Layer)**:使用双向 LSTM(隐藏单元维度为 `hidden_dim`)，能够从前向和后向提取文本特征，结合两侧的上下文信息，生成每个时间步的隐状态。

(3). **注意力机制（Attention Mechanism）**：将 BiLSTM 的输出传入注意力网络，通过计算权重矩阵提取重要特征。具体步骤包括：

- 计算注意力分数：利用输入序列与查询向量进行点积计算，结果除以 $\sqrt{d_k}$ 进行归一化。
- 计算注意力分布：对注意力分数使用 Softmax 操作，生成权重分布。
- 提取上下文向量：根据权重分布对输入序列的特征进行加权求和。

通过注意力机制，模型能够更好地关注句子中对分类任务具有重要意义的部分。

(4). **全连接层（Fully Connected Layer）**：将注意力网络输出的特征向量输入全连接层，输出维度为分类任务的类别数（本任务中为 5）。

3.2.2 网络实现代码

改进网络的实现代码如下：

```
class BiLSTM_Attention(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, n_layers):
        super(BiLSTM_Attention, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.rnn = nn.LSTM(embedding_dim, hidden_dim,
                           num_layers=n_layers, bidirectional=True, dropout
                              =0.5)
        self.fc = nn.Linear(hidden_dim * 2, 5) # 输出类别数为5
        self.dropout = nn.Dropout(0.5)

    def attention_net(self, x, query, mask=None):
        d_k = query.size(-1)
        scores = torch.matmul(query, x.transpose(1, 2)) / math.sqrt(d_k) #
            点积注意力
        p_attn = F.softmax(scores, dim=-1) # 注意力权重分布
        context = torch.matmul(p_attn, x).sum(1) # 加权求和生成上下文向量
        return context, p_attn

    def forward(self, x):
        embedding = self.dropout(self.embedding(x)) # 嵌入层 + Dropout
        output, (final_hidden_state, final_cell_state) = self.rnn(
            embedding) # BiLSTM 层
        output = output.permute(1, 0, 2) # 调整维度以适配注意力网络
        query = self.dropout(output) # 查询向量
        attn_output, attention = self.attention_net(output, query) # 注意
            力机制
        logit = self.fc(attn_output) # 全连接层输出
        return logit
```

3.2.3 改进特点

相比初始网络，改进后的 BiLSTM-Attention 网络具有以下优势：

- **双向特征提取**：双向 LSTM 能够捕获句子的全局语义信息和上下文依赖关系。

- **注意力机制**：通过注意力机制重点关注对分类任务重要的词汇和局部信息，提高了模型的表示能力。
- **增强鲁棒性**：通过 Dropout 降低过拟合风险，适应复杂的文本分类任务。

3.2.4 模型输出

模型的输出是一个形状为 $[batch\_size, 5]$ 的张量，其中每一行表示一个样本属于各类别的概率分布。通过对输出概率进行 `argmax` 操作，获得最终的分类结果。

4 实验结果

4.1 初始网络

将学习率设定为 0.001，训练 100 个 epoch，观察训练集上的 Loss 和验证集上的 accuracy 的变化曲线如下：

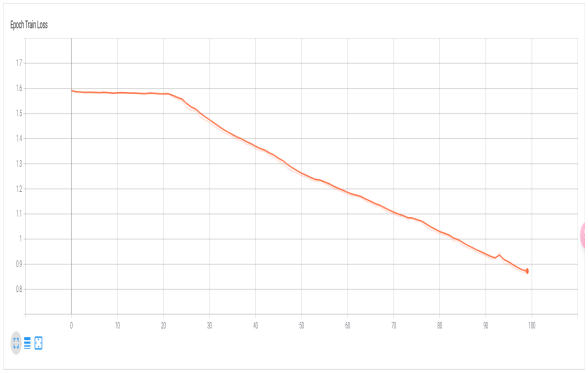

Figure 1: Train Loss

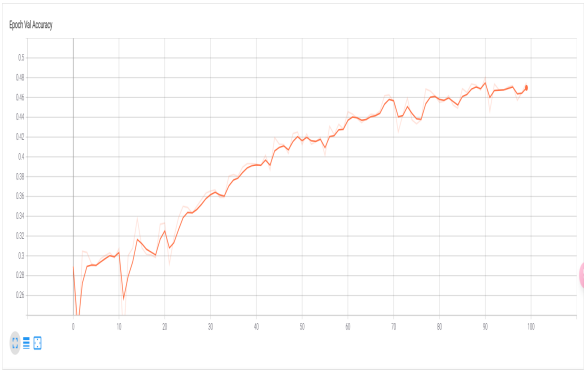

Figure 2: Validation Accuracy

可以看到，随着训练回合的增加，Loss 几乎呈线性下降，Accuray 也逐步提升，但速度逐渐变缓，且 100 个 epoch 后仍未达到 50%，训练效率较低，因此不继续训练而改用新的网络结构。

4.2 改进后的网络

选取 LSTM 的隐藏层维度为 64，学习率设为 0.001，训练 10 个 epcoh：

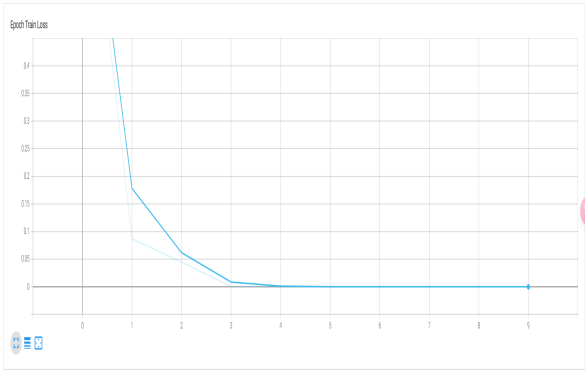

Figure 3: Train Loss

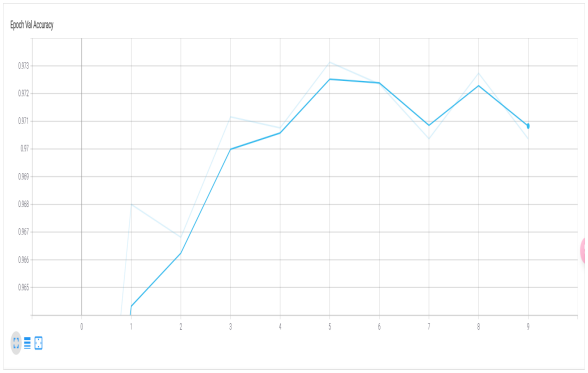

Figure 4: Validation Accuracy

可以看到，在前 3 个 epoch 内 Loss 快速下降，达到了 10^{-4} 的数量级，且 accuracy 最终在 96%-97% 波动。注意到这里我们使用了 `GloVe` 预训练词向量，大大提高了模型的表现。上传到 MO 平台上进行验证，准确率达到 47/50：

4.3 参数优化

由于训练样本较少，这里我们降隐藏层单元由 64 降到 32，且使用学习率调整策略来使训练更加稳定。这里我们设置当模型迭代后，在训练集上的损失没有下降，就调整学习率。

```
scheduler = ReduceLROnPlateau(
 optimizer, mode='min', factor=0.3, patience=1, verbose=True)
 ...
 # 调用学习率调度器
 scheduler.step(train_loss)
```

训练 5 个 epoch, 观察训练集上的 Loss 和验证集上的 accuracy 的变化曲线如下：

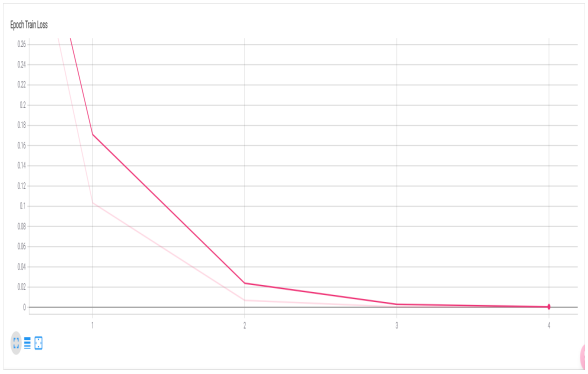

Figure 5: Train Loss

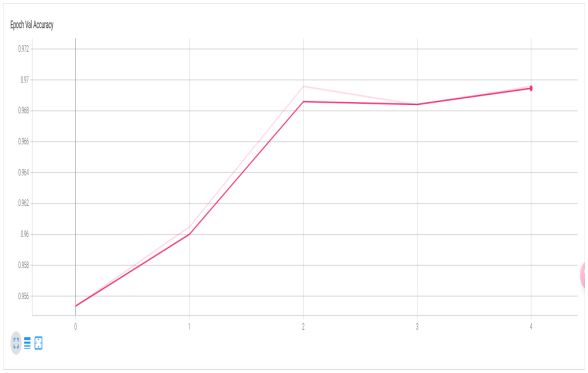

Figure 6: Validation Accuracy

上传到 MO 平台进行验证，准确率为 50/50：

5 总结

通过这次实验我学习了文本序列的分词预处理和神经网络模型，特别是 LSTM 模型的应用，同时通过引入双向 LSTM 与注意力机制，较好地完成了预测任务，是一次很好的实践机会。

6 代码附录

```python
# 导入相关包
import os
import numpy as np
import jieba as jb
import torch
import torch.nn as nn
import torch.nn.functional as F
import math
from torch.optim.lr_scheduler import ReduceLROnPlateau
from torchtext.data import Field, Dataset, Iterator, Example,
   BucketIterator
from torch.utils.tensorboard import SummaryWriter


class Net(nn.Module):
    def __init__(self, vocab_size):
        super(Net, self).__init__()
        pass

    def forward(self, x):
        """
        前向传播
        :param x: 模型输入
        :return: 模型输出
        """
        pass


def processing_data(data_path, split_ratio=0.7):
    """
    数据处理
    :data_path: 数据集路径
    :validation_split: 划分为验证集的比重
    :return: train_iter,val_iter,TEXT.vocab 训练集、验证集和词典
    """

```

```
36      sentences = [] # 片段
37      target = [] # 作者
38
39      # 定义lebel到数字的映射关系
40      labels = {'LX': 0, 'MY': 1, 'QZS': 2, 'WXB': 3, 'ZAL': 4}
41
42      files = os.listdir(data_path)
43      for file in files:
44          if not os.path.isdir(file):
45              f = open(data_path + "/" + file, 'r', encoding='UTF-8') # 打开
                    文件
46              for index, line in enumerate(f.readlines()):
47                  sentences.append(line)
48                  target.append(labels[file[:-4]])
49
50      mydata = list(zip(sentences, target))
51
52      TEXT = Field(sequential=True, tokenize=lambda x: jb.lcut(x),
53                   lower=True, use_vocab=True)
54      LABEL = Field(sequential=False, use_vocab=False)
55
56      FIELDS = [('text', TEXT), ('category', LABEL)]
57
58      examples = list(map(lambda x: Example.fromlist(list(x), fields=FIELDS
            ),
59                          mydata))
60
61      dataset = Dataset(examples, fields=FIELDS)
62
63      TEXT.build_vocab(dataset, vectors='glove.6B.100d')
64
65      train, val = dataset.split(split_ratio=split_ratio)
66
67      # BucketIterator可以针对文本长度产生batch，有利于训练
68      train_iter, val_iter = BucketIterator.splits(
69          (train, val), # 数据集
70          batch_sizes=(16, 16),
71          device=device, # 如果使用gpu，此处将-1更换为GPU的编号
```

```
72          sort_key=lambda x: len(x.text),
73          sort_within_batch=False,
74          repeat=False
75      )
76
77      return train_iter, val_iter, TEXT.vocab
78
79
80  class Net(nn.Module):
81      def __init__(self):
82          super(Net, self).__init__()
83          self.lstm = torch.nn.LSTM(1, 64)
84          self.fc1 = nn.Linear(64, 128)
85          self.fc2 = nn.Linear(128, 5)
86
87      def forward(self, x):
88          """
89          前向传播
90          :param x: 模型输入
91          :return: 模型输出
92          """
93          output, hidden = self.lstm(x.unsqueeze(2).float())
94          h_n = hidden[1]
95          out = self.fc2(self.fc1(h_n.view(h_n.shape[1], -1)))
96          return out
97
98
99  class BiLSTM_Attention(nn.Module):
100     def __init__(self, vocab_size, embedding_dim, hidden_dim, n_layers):
101         super(BiLSTM_Attention, self).__init__()
102         self.hidden_dim = hidden_dim
103         self.n_layers = n_layers
104         self.embedding = nn.Embedding(vocab_size, embedding_dim)
105         self.rnn = nn.LSTM(embedding_dim, hidden_dim,
106                            num_layers=n_layers, bidirectional=True, dropout
                               =0.5)
107         self.fc = nn.Linear(hidden_dim * 2, 5)
108         self.dropout = nn.Dropout(0.5)
```

```
109
110     def attention_net(self, x, query, mask=None):
111         d_k = query.size(-1)
112         scores = torch.matmul(query, x.transpose(1, 2)) / math.sqrt(d_k)
113         p_attn = F.softmax(scores, dim=-1)
114         context = torch.matmul(p_attn, x).sum(1)
115         return context, p_attn
116
117     def forward(self, x):
118         embedding = self.dropout(self.embedding(x))
119         output, (final_hidden_state, final_cell_state) = self.rnn(
                embedding)
120         output = output.permute(1, 0, 2)
121         query = self.dropout(output)
122         attn_output, attention = self.attention_net(output, query)
123         logit = self.fc(attn_output)
124         return logit
125
126
127 data_path = "./dataset" # 数据集路径
128 save_model_path = "results/model.pth" # 保存模型路径和名称
129 train_val_split = 0.7 # 验证集比重
130
131 # 自动选择设备: 如果有 GPU 就用 GPU, 否则使用 CPU
132 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
133 print(device)
134
135 # 获取数据、并进行预处理
136 train_iter, val_iter, Text_vocab = processing_data(
137     data_path, split_ratio=train_val_split)
138
139 vocab_path = "results/text_vocab.pth"
140 torch.save(Text_vocab, vocab_path)
141 print("词典已保存至:", vocab_path)
142
143 # 创建模型实例
144 # model = Net().to(device)
145 EMBEDDING_DIM = 100 # 词向量维度
```

```
len_vocab = len(Text_vocab)
model = BiLSTM_Attention(len_vocab, EMBEDDING_DIM,
                         hidden_dim=32, n_layers=2).to(device)
pretrained_embedding = Text_vocab.vectors
model.embedding.weight.data.copy_(pretrained_embedding)

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
scheduler = ReduceLROnPlateau(
    optimizer, mode='min', factor=0.3, patience=1, verbose=True)

# 创建一个 SummaryWriter 实例，指定日志目录
writer = SummaryWriter('runs/bi_32_5')
for epoch in range(5):
    train_acc, train_loss = 0, 0
    val_acc, val_loss = 0, 0
    for idx, batch in enumerate(train_iter):
        # 移动数据到同一个设备
        text, label = batch.text.to(device), batch.category.to(device)
        optimizer.zero_grad()
        out = model(text)
        loss = loss_fn(out, label.long())
        # loss.backward(retain_graph=True)
        loss.backward()
        optimizer.step()
        accuracy = np.mean((torch.argmax(out, 1) == label).cpu().numpy())
        # 计算每个样本的acc和loss之和
        train_acc += accuracy*len(batch)
        train_loss += loss.item()*len(batch)

        # 在 TensorBoard 中记录训练的 Loss 和 Accuracy
        writer.add_scalar('Train Loss', loss.item(),
                          epoch * len(train_iter) + idx)
        writer.add_scalar('Train Accuracy', accuracy,
                          epoch * len(train_iter) + idx)

    # 在验证集上预测
    with torch.no_grad():
```

```
184         for idx, batch in enumerate(val_iter):
185             text, label = batch.text, batch.category
186             out = model(text)
187             loss = loss_fn(out, label.long())
188             accracy = np.mean((torch.argmax(out, 1) == label).cpu().numpy()
                    )
189             # 计算一个batch内每个样本的acc和loss之和
190             val_acc += accracy*len(batch)
191             val_loss += loss.item()*len(batch)
192 
193     train_acc /= len(train_iter.dataset)
194     train_loss /= len(train_iter.dataset)
195     val_acc /= len(val_iter.dataset)
196     val_loss /= len(val_iter.dataset)
197 
198     scheduler.step(train_loss)
199 
200     # 记录每个 epoch 的 Train 和 Validation 的 Loss 和 Accuracy
201     writer.add_scalar('Epoch Train Loss', train_loss, epoch)
202     writer.add_scalar('Epoch Train Accuracy', train_acc, epoch)
203     writer.add_scalar('Epoch Val Loss', val_loss, epoch)
204     writer.add_scalar('Epoch Val Accuracy', val_acc, epoch)
205 
206     print("epoch:{} loss:{}, val_acc:{}\n".format(
207         epoch, train_loss, val_acc), end=" ")
208 
209 # 保存模型
210 torch.save(model.state_dict(), 'results/temp.pth')
211 
212 # 关闭 TensorBoard writer
213 writer.close()
```