系统文档

小组成员及分工：

姓名	学号	任务分工	任务占比
何欣洋（组长）	202228015329003	主要负责搭建系统整体前端和后端的框架，LSTM 模型、数据预处理等	60%
吕卓锦	202228015329004	主要负责搭建 CNN 模型，对模型进行测试，设计网页样式等	40%

任务定义：

情感分析或观点挖掘是对人们对产品、服务、组织、个人、问题、事件、话题及其属性的观点、情感、情绪、评价和态度的计算研究。近年来随着互联网信息的发展和深度学习的出现，使得越来越多的人投入到情感分析的研究当中。情感分析本质上是一个分类任务，常见的分类方法有：二分类或者进行打分（0 到 10）等等

方法描述：

本次作业主要利用 LSTM 和 CNN 对电影评论数据进行情感分析任务。LSTM 是一种特殊的 RNN，它在 RNN 的基础上引入了 gate 机制，从而使得 LSTM 克服了 RNN “记忆力不好的特点”，克服了简单的 RNN 梯度消失或者梯度爆炸的问题。CNN 是一种具有局部连接、权重共享等特性的深层前馈神经网络，虽然 CNN 广泛地用于图像或视频的分析任务，但如果将文本看作一维图像，也可以将 CNN 用于 nlp 任务中。用卷积层替代全连接层可以极大减少要学习的参数数目。

系统框架：

后端：torch 版本：1.13.0，python 版本：3.19.2，django 版本：4.1.4
前端：bootstrap

实现细节：

数据预处理部分：

通过正则表达式将不是字母或数字的字符替换为空格等

```python
#清理文本，去标点符号，转小写
def clean_str(string):
    string = re.sub(r"[^A-Za-z0-9]", " ", string)
    string = re.sub(r"\'s", " \'s", string)
    string = re.sub(r"\'ve", " \'ve", string)
    string = re.sub(r"n\'t", " n\'t", string)
    string = re.sub(r"\'re", " \'re", string)
    string = re.sub(r"\'d", " \'d", string)
    string = re.sub(r"\'ll", " \'ll", string)
    string = re.sub(r",", " , ", string)
    string = re.sub(r"!", " ! ", string)
    string = re.sub(r"\(", " \( ", string)
    string = re.sub(r"\)", " \) ", string)
    string = re.sub(r"\?", " \? ", string)
    string = re.sub(r"\s{2,}", " ", string)
    string = re.sub(r"\s{2,}", " ", string)
    string = re.sub(r"sssss ", " ", string)
    return string.strip().lower()
```

建立词典：

```python
    word_count_sort = sorted(word_count.items(), key=lambda item : item[1], reverse=True) # 对词进行排序，过滤低频词，只取前MAX_WORD个高频词
    word_number = 1
    for word in word_count_sort:
        if word[0] not in vocab.keys():
            vocab[word[0]] = len(vocab)
            word_number += 1
        if word_number > MAX_WORD:
            break
    return vocab
```

根据词典将句子转换为等长的 tensor

```python
# 根据vocab将句子转为定长MAX_LEN的tensor
def text_transform(sentence_list, vocab):
    sentence_index_list = []
    for sentence in sentence_list:
        sentence_idx = [vocab[token] if token in vocab.keys() else vocab['<UNK>'] for token in tokenizer(sentence)] # 句子分词转为id

        if len(sentence_idx) < MAX_LEN:
            for i in range(MAX_LEN-len(sentence_idx)): # 对长度不够的句子进行PAD填充
                sentence_idx.append(vocab['<PAD>'])

        sentence_idx = sentence_idx[:MAX_LEN] # 取前MAX_LEN长度
        sentence_index_list.append(sentence_idx)
    return torch.LongTensor(sentence_index_list) # 将转为idx的词转为tensor
```

模型训练并保存模型

```python
# 模型训练
def train(model, train_data, loss_fn, optimizer, vocab, epoch=10, method='LSTM'):
    print('train model')
    model = model.to(device)
    #记录每个epoch的损失
    record_loss = []
    record_acc = []
    # 定义损失函数和优化器
    loss_func = loss_fn
    optimizer_func = optimizer

    for epoch in tqdm(range(epoch)):
        model.train()
        avg_loss = 0  # 平均损失
        avg_acc = 0  # 平均准确率
        for i, (text, label) in enumerate(tqdm(train_data)):

            train_x = text_transform(text, vocab).to(device)
            train_y = label.to(device)

            #BP
            optimizer_func.zero_grad()
            pred = model(train_x)
            if (method == 'LSTM'):
                pred = pred.log()
            loss = loss_func(pred, train_y)
            loss.backward()
            optimizer_func.step()
            avg_loss += loss.item()
            avg_acc += accuracy(pred, train_y)
        # 一个epoch结束后，计算平均loss和评平均acc
        avg_loss = avg_loss / len(train_data)
        avg_acc = avg_acc / len(train_data)
        record_loss.append(avg_loss)
        record_acc.append(avg_acc)

        print("avg_loss:", avg_loss, " train_avg_acc:,", avg_acc)

        # 保存训练完成后的模型参数
        torch.save(model.state_dict(), method + '_IMDB_parameter.pkl')
```

构建 CNN 模型：

```python
class TextCNN(nn.Module):
    def __init__(self, vocab_size, embed_sizes, kernel_sizes, num_channels,
                 **kwargs):
        super(TextCNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_sizes)
        # 这个嵌入层不需要训练
        self.constant_embedding = nn.Embedding(vocab_size, embed_sizes)
        self.dropout = nn.Dropout(0.5)
        self.decoder = nn.Linear(sum(num_channels), 2)
        # 最大时间汇聚层没有参数，因此可以共享此实例
        self.pool = nn.AdaptiveAvgPool1d(1)
        self.relu = nn.ReLU()
        # 创建多个一维卷积层
        self.convs = nn.ModuleList()
        for c, k in zip(num_channels, kernel_sizes):
            self.convs.append(nn.Conv1d(2 * embed_sizes, c, k))

    def forward(self, inputs):
        # 沿着向量维度将两个嵌入层连结起来，
        # 每个嵌入层的输出形状都是（批量大小，词元数量，词元向量维度）连结起来
        embeddings = torch.cat((
            self.embedding(inputs), self.constant_embedding(inputs)), dim=2)
        # 根据一维卷积层的输入格式，重新排列张量，以便通道作为第2维
        embeddings = embeddings.permute(0, 2, 1)
        # 每个一维卷积层在最大时间汇聚层合并后，获得的张量形状是（批量大小，通道数，1）
        # 删除最后一个维度并沿通道维度连结
        encoding = torch.cat([
            torch.squeeze(self.relu(self.pool(conv(embeddings))), dim=-1)
            for conv in self.convs], dim=1)
        outputs = self.decoder(self.dropout(encoding))
        return outputs
```

构建 LSTM 模型：

```python
class TextCNN(nn.Module):
    def __init__(self, vocab_size, embed_sizes, kernel_sizes, num_channels,
                 **kwargs):
        super(TextCNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_sizes)
        # 这个嵌入层不需要训练
        self.constant_embedding = nn.Embedding(vocab_size, embed_sizes)
        self.dropout = nn.Dropout(0.5)
        self.decoder = nn.Linear(sum(num_channels), 2)
        # 最大时间汇聚层没有参数，因此可以共享此实例
        self.pool = nn.AdaptiveAvgPool1d(1)
        self.relu = nn.ReLU()
        # 创建多个一维卷积层
        self.convs = nn.ModuleList()
        for c, k in zip(num_channels, kernel_sizes):
            self.convs.append(nn.Conv1d(2 * embed_sizes, c, k))

    def forward(self, inputs):
        # 沿着向量维度将两个嵌入层连结起来，
        # 每个嵌入层的输出形状都是（批量大小，词元数量，词元向量维度）连结起来
        embeddings = torch.cat((
            self.embedding(inputs), self.constant_embedding(inputs)), dim=2)
        # 根据一维卷积层的输入格式，重新排列张量，以便通道作为第2维
        embeddings = embeddings.permute(0, 2, 1)
        # 每个一维卷积层在最大时间汇聚层合并后，获得的张量形状是（批量大小，通道数，1）
        # 删除最后一个维度并沿通道维度连结
        encoding = torch.cat([
            torch.squeeze(self.relu(self.pool(conv(embeddings))), dim=-1)
            for conv in self.convs], dim=1)
        outputs = self.decoder(self.dropout(encoding))
        return outputs
```

数据集：

数据集使用的是 IMDB 网站的电影评论数据，有 25k 训练集和 25k 测试集，训练集和测试集又各包含 12.5k 个 positive 的样本和 negative 的样本。

基准系统：

macos 13.0， m1silicon

实验设置：

基础实验设置为：

对于 LSTM 模型： 设置参数为 **epoch=5**， **learning rate = 0.006**， **batch_size = 256**，**embde_size = 300, num_layers = 2**。

对于 CNN 模型 设置参数为 **epoch=5**, **learning rate = 0.005**, **batch_size = 256，embde_size = 300，kernel size = [3,4,5]，channel_size = [100,100,100]**

实验结果与分析：

1. 当 epoch 越大时，训练的准确率越高，对应的在测试集上的准确率也就越高，但是 epoch 越大， 花费的训练时间也就越多

当 eopch=3 时， 训练准确率为：0.5846， 测试准确率为：0.5403

```
HelloWorld — python3 ‹ python3 manage.py runserver 0.0.0.0:8000 — 80...
System check identified no issues (0 silenced).
December 15, 2022 - 03:12:08
Django version 4.1.4, using settings 'HelloWorld.settings'
Starting development server at http://0.0.0.0:8000/
Quit the server with CONTROL-C.
3
data preprocess
build vocabulary
data preprocess
build vocabulary
train model
100%|██████████████████████████████████| 98/98 [04:12<00:00,  2.58s/it]
avg_loss: 0.6955225990743054  train_avg_acc:, 0.5091753219144801 0:00,  2.42s/it]
100%|██████████████████████████████████| 98/98 [03:57<00:00,  2.42s/it]
avg_loss: 0.6748258161301516  train_avg_acc:, 0.5429137056608357 0:00,  2.16s/it]
100%|██████████████████████████████████| 98/98 [03:52<00:00,  2.38s/it]
avg_loss: 0.6230521305483214  train_avg_acc:, 0.5846999514091351 0:00,  2.15s/it]
100%|███████████████████████████████████| 3/3 [12:03<00:00, 241.04s/it]
test model
100%|████████████████████████████████| 391/391 [02:01<00:00,  3.22it/s]
0.5403852301790282
```

当 eopch=5 时，训练准确率为：0.8758，测试准确率为：0.8313

```
Starting development server at http://0.0.0.0:8000/
Quit the server with CONTROL-C.
5
data preprocess
build vocabulary
data preprocess
build vocabulary
train model
100%|██████████████████████████████████| 98/98 [04:18<00:00,  2.64s/it]
avg_loss: 0.694963308621426  train_avg_acc:, 0.5133966533041788 00:00,  2.15s/it]
100%|██████████████████████████████████| 98/98 [04:03<00:00,  2.49s/it]
avg_loss: 0.6743280443609977  train_avg_acc:, 0.5396869685374149 0:00,  2.19s/it]
100%|██████████████████████████████████| 98/98 [04:07<00:00,  2.53s/it]
avg_loss: 0.6248441971078211  train_avg_acc:, 0.6075566387269193 0:00,  2.11s/it]
100%|██████████████████████████████████| 98/98 [04:00<00:00,  2.45s/it]
avg_loss: 0.42928457138489706  train_avg_acc:, 0.811374438168124 4:00,  2.42s/it]
100%|██████████████████████████████████| 98/98 [08:57<00:00,  5.49s/it]
avg_loss: 0.31104910419303544  train_avg_acc:, 0.875840849732750 2:00,  2.71s/it]
100%|███████████████████████████████████| 5/5 [25:28<00:00, 305.66s/it]
test model
100%|████████████████████████████████| 391/391 [02:38<00:00,  2.47it/s]
0.8313938618925831
[15/Dec/2022 04:02:50] "POST /runoob/ HTTP/1.1" 200 441
```

2. 基于 CNN 的方法：

epoch=5 时，训练准确率可以达到 0.8931。

```
100%|██████████████████████████████████| 98/98 [05:40<00:00,  3.48s/it]
avg_loss: 0.6586544872546682  train_avg_acc:, 0.6115995505344994 | 98/98 [05:40<00:00,  2.92s/it]
100%|██████████████████████████████████| 98/98 [05:25<00:00,  3.32s/it]
avg_loss: 0.49703505513619406  train_avg_acc:, 0.7643513878765792 | 98/98 [05:25<00:00,  2.54s/it]
100%|██████████████████████████████████| 98/98 [04:43<00:00,  2.89s/it]
avg_loss: 0.37114084040632056  train_avg_acc:, 0.8388472576530612 | 98/98 [04:43<00:00,  2.57s/it]
100%|██████████████████████████████████| 98/98 [04:42<00:00,  2.88s/it]
avg_loss: 0.31769947099442386  train_avg_acc:, 0.8649325801749271 | 98/98 [04:42<00:00,  2.57s/it]
100%|██████████████████████████████████| 98/98 [04:42<00:00,  2.88s/it]
avg_loss: 0.2684150502389791  train_avg_acc:, 0.8931608357628765 | 98/98 [04:42<00:00,  2.52s/it]
100%|███████████████████████████████████| 5/5 [25:14<00:00, 302.84s/it]
[W NNPACK.cpp:53] Could not initialize NNPACK! Reason: Unsupported hardware.
positive
```

3. 有时候会出现 loss 为 nan 的情况，可以从学习率、优化方法、损失函数等方面进行考虑，对于不同的模型有不同的合适的优化方法。比如：

```
eopch:  1 learning rate:  0.001 batch size:  256
embed_size:  300 num_hiddens:  3,4,5 num_layers:  100,100,100
train model
100%|██████████████████████████████████| 98/98 [04:42<00:00,  2.88s/it]
avg_loss: nan  train_avg_acc:, 0.50070418792517█| 98/98 [04:42<00:00,  2.50s/it]
100%|███████████████████████████████████| 1/1 [04:42<00:00, 282.61s/it]
record loss:  [nan] record acc:  [0.50070418792517]
```

此时基于 CNN 的训练出现了 loss 为 nan 的情况，原因是因为设置了当前的损失函数为 nllloss，改为交叉熵损失函数可以避免这种情况，如下：

```
embed_size:  300 num_hiddens:  3,4,5 num_layers:  100,100,100
train model
100%|████████████████████████████████████████| 98/98 [04:40<00:00,  2.87s/it]
avg_loss: 0.6808737717112716  train_avg_acc:, 0.57918223092808550:00,  2.48s/it]
100%|████████████████████████████████████████| 98/98 [04:45<00:00,  2.92s/it]
avg_loss: 0.521821597705082  train_avg_acc:, 0.744687272230320700:00,  2.51s/it]
100%|████████████████████████████████████████| 98/98 [04:50<00:00,  2.96s/it]
avg_loss: 0.3879949979636134  train_avg_acc:, 0.83111068087949470:00,  2.74s/it]
100%|█████████████████████████████████████████| 3/3 [14:17<00:00, 285.72s/it]
record loss:  [0.6808737717112716, 0.521821597705082, 0.3879949979636134] record
 acc:  [0.5791822309280855, 0.7446872722303207, 0.8311106808794947]
```

4. 在调用 matplotlib 作图时会出现警告，用 matplotlib 在子线程中作图可能不安全，会导致程序中断：

```
/Users/hxy/Sentiment_project/Sentiment_project/utils.py:121: UserWarning: Starti
ng a Matplotlib GUI outside of the main thread will likely fail.
  plt.clf()
```

解决方法是可以在主线程调用涉及到作图的程序
5. 可以通过手动调参的方式观察训练结果，为了防止过拟合可以设置 dropout 为 0.5 等方式。