

# 实验五——多模态情感分析实验报告

姓名：陆思遥 学号：10215501401







github地址：[Kostium/-hw5: 人工智能第五次实验 \(github.com\)](https://github.com/Kostium/-hw5)  
(<https://github.com/Kostium/-hw5>)

## 一、实验任务

本次实验是实现多模态情感分析，给定配对的文本和图像，实现三分类任务，预测对应的情感标签：positive,neutral,negative。

数据集介绍：

(1) data文件夹：包括所有的训练文本和图片，每个文件按照唯一的guid命名。


|                                                                                           |                |          |
|-------------------------------------------------------------------------------------------|----------------|----------|
|  1.jpg   | 2015/5/14 5:23 | JPG 图片文件 |
|  1.txt   | 2015/5/14 5:23 | 文本文档     |
|  2.jpg   | 2015/5/14 5:23 | JPG 图片文件 |
|  2.txt  | 2015/5/14 5:23 | 文本文档     |
|  3.jpg | 2015/5/14 5:23 | JPG 图片文件 |
|  3.txt | 2015/5/14 5:23 | 文本文档     |

其中一个图片和文本的具体情况如下图所示：

- 图片



- 文本信息

 1.txt - 记事本  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
How I feel today #legday #jelly #aching #gym

(2) train.txt：数据的guid和对应的情感标签。

```
guid,tag
4597,negative
26,neutral
4383,negative
212,positive
2626,positive
```

(3) test\_without\_label.txt：数据的guid和空的情感标签。

```
guid,tag
8,null
1576,null
2320,null
4912,null
3821,null
```

## 二、实验过程

### (1) 数据预处理

1. **读取数据**: 使用pandas库的read\_csv函数从指定的CSV文件中读取训练数据和测试数据。  
delimiter=','指定了CSV文件中的字段分隔符为逗号, header=0表示使用第一行作为列名。

```
train_data_path = 'dataset/train.txt'
test_data_path = 'dataset/test_without_label.txt'
train_data = pd.read_csv(train_data_path, delimiter=',', header=0)
test_data = pd.read_csv(test_data_path, delimiter=',', header=0)
```

2. **划分训练集和验证集**: 使用train\_test\_split函数将训练数据划分为训练集和验证集, 其中test\_size=0.2表示验证集占总数据的20%。

```
train_df, val_df = train_test_split(train_data, test_size=0.2, random_state=42)
```

3. **映射标签为整数**: 创建了一个标签映射字典tag\_mapping, 将字符串标签 ('negative', 'neutral', 'positive') 映射为整数 (0, 1, 2)。然后, 通过map函数, 将tag列中的3种情感标签映射为相应的整数, 将结果存储在新的列tag\_encoded中。

```
tag_mapping = {'negative': 0, 'neutral': 1, 'positive': 2}
train_df['tag_encoded'] = train_df['tag'].map(tag_mapping)
```

4. **定义训练集的标签**: 将训练集的标签转换为PyTorch的张量 (tensor), 并指定数据类型为torch.long, 适用于分类任务的标签。

```
train_labels = torch.tensor(train_df['tag_encoded'].values, dtype=torch.long)
```

5. **数据加载**: 通过定义的load\_data函数, 加载文本和图片数据到相应的列表中, 并通过遍历训练集、验证集和测试集的数据框, 调用load\_data函数完成数据的加载。

- 定义load\_data函数:

- load\_data函数接受两个参数: guid (文件标识符) 和state (表示加载的数据是训练集、验证集还是测试集)。
- 使用os.path.join构建文本文件和图片文件的完整路径。
- 打开文本文件, 读取文本数据, 并将其添加到相应的列表中。同时, 打开对应的图片文件, 将图片数据转换为RGB格式, 然后添加到相应的图片数据列表中。

```
def load_data(guid, state):
    file_path = os.path.join(data_folder, f'{guid}.txt')
    image_file_path = os.path.join(data_folder, f'{guid}.jpg')
    with open(file_path, 'r', encoding='ISO-8859-1') as text_file:
        text_data = text_file.read()
        if state == 'is_train':
            text_data_list_train.append(text_data)
            image_data_list_train.append(Image.open(image_file_path).convert('RGB'))
        elif state == 'is_val':
            text_data_list_val.append(text_data)
            image_data_list_val.append(Image.open(image_file_path).convert('RGB'))
        elif state == 'is_test':
            text_data_list_test.append(text_data)
            image_data_list_test.append(Image.open(image_file_path).convert('RGB'))
```

- 遍历训练集、验证集和测试集，加载数据：对于每一行数据，获取当前行的guid和tag，然后调用load\_data函数加载相应的文本和图片数据到训练集的列表中。

```
# 遍历 train_df(训练集)，加载对应的文本和图片数据
for index, row in train_df.iterrows():
    current_guid, current_tag = row['guid'], row['tag']
    load_data(current_guid, state = 'is_train')
# 遍历 val_df(验证集)，加载对应的文本和图片数据
for index, row in val_df.iterrows():
    current_guid_val, current_tag_val = row['guid'], row['tag']
    load_data(current_guid_val, state = 'is_val')
    true_labels_val.append(tag_mapping[current_tag_val])
# 遍历 test_data(测试集)，加载对应的文本和图片数据
for index, row in test_data.iterrows():
    current_guid_test, current_tag_test = int(row['guid']), row['tag']
    load_data(current_guid_test, state = 'is_test')
print("数据预处理已完成")
```

## (2) 文本特征提取

1. 创建了一个TfidfVectorizer实例，该实例将用于将文本数据转换为TF-IDF特征。
2. 通过 fit\_transform 方法，将训练集中的文本数据 text\_data\_list\_train 转换为 TF-IDF 特征矩阵 text\_vectorizer\_train。
3. 使用 todense() 方法将稀疏矩阵转换为密集矩阵。
4. 将得到的密集矩阵转换为 PyTorch 张量，存储在 text\_features\_train 中。
5. 验证集和测试集同理。

```
# 使用 TfidfVectorizer 进行文本向量化
vectorizer = TfidfVectorizer()
text_vectorizer_train = vectorizer.fit_transform(text_data_list_train)
text_features_train = torch.Tensor(text_vectorizer_train.todense())
text_vectorizer_val = vectorizer.transform(text_data_list_val)
text_features_val = torch.Tensor(text_vectorizer_val.todense())
text_vectorizer_test = vectorizer.transform(text_data_list_test)
text_features_test = torch.Tensor(text_vectorizer_test.todense())
```

## (3) 图像特征提取

1. **定义图像数据的变换：**创建了一个图像数据变换的管道，其中包括将图像调整为大小 (32, 32)，转换为 PyTorch 张量，以及进行归一化操作。

```
image_transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
])
```

2. **将图像数据转换为 PyTorch 张量：**对训练集、验证集和测试集的图像数据应用定义好的图像变换，然后使用 torch.stack 将它们堆叠成一个张量。

```
image_data_tensor = torch.stack([image_transform(image) for image in
    image_data_list_train])
image_data_tensor_val = torch.stack([image_transform(image) for image in
    image_data_list_val])
image_data_tensor_test = torch.stack([image_transform(image) for image in
    image_data_list_test])
```

3. **定义 LeNet 架构：**创建了一个继承自 `nn.Module` 的 LeNet 模型，其中包含两个卷积层、两个最大池化层和三个全连接层。

`__init__`方法用于初始化LeNet模型的各个层。模型包括两个卷积层（`conv1`和`conv2`）、两个最大池化层（`pool1`和`pool2`）以及三个全连接层（`fc1`、`fc2`和`fc3`）。输入图像的通道数为3，因为这里假设使用了RGB图像，所以通道数为3。`kernel_size`表示卷积核的大小。

`forward`方法定义了LeNet模型的前向传播过程。输入 `x` 经过第一个卷积层，然后通过ReLU激活函数和池化层，接着经过第二个卷积层、ReLU激活函数和池化层。`view`方法用于将最终的卷积层输出展平为一维向量。之后，通过两个全连接层，并在每一层应用ReLU激活函数。输出层 `fc3` 输出一个包含3个元素的向量，表示3个类别的概率。

`F.log_softmax` 应用了`log_softmax`作为激活函数，方便后续计算损失。

```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, kernel_size=5)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 3) # 3是类别数量
    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return F.log_softmax(x, dim=1) # 使用log_softmax作为激活函数
lenet_model = LeNet()
```

4. **使用 LeNet 模型提取图像特征：**

使用 LeNet 模型对训练集、验证集和测试集的图像数据进行前向传播，得到相应的图像特征。

`view` 方法用于将特征张量展平，以便与文本特征合并。

```
image_features = lenet_model(image_data_tensor)
image_features = image_features.view(image_features.size(0), -1)
image_features_val = lenet_model(image_data_tensor_val)
image_features_val = image_features_val.view(image_features_val.size(0), -1)
image_features_test = lenet_model(image_data_tensor_test)
image_features_test = image_features_test.view(image_features_test.size(0), -1)
```

## (4) 多模态融合模型

1. **合并训练集、验证集和测试集文本特征和图像特征：**

- 使用 `torch.cat` 将文本特征 `text_features_train` 和图像特征 `image_features` 沿着第二个维度（`dim=1`，列方向）合并成一个张量。
- `merged_features_tensor` 是合并后的特征的 PyTorch 张量表示。

- 对验证集和测试集的文本特征和图像特征进行相同的合并操作。

```
merged_features = torch.cat([text_features_train, image_features], dim=1) # 合并
文本特征和图像特征
merged_features_tensor = torch.Tensor(merged_features) # 将合并后的特征转为
PyTorch 张量
merged_features_val = torch.cat([text_features_val, image_features_val], dim=1)
merged_features_tensor_val = torch.Tensor(merged_features_val)
merged_features_test = torch.cat([text_features_test, image_features_test],
dim=1)
merged_features_tensor_test = torch.Tensor(merged_features_test)
```

## 2. 构造多模态模型 MultimodalModel:

- 创建了一个继承自 nn.Module 的多模态模型 MultimodalModel。
- 模型包括一个输入层 fc1, 一个隐藏层 fc2 和一个输出层 fc3。
- 输入层的大小为 input\_size, 该大小等于合并后的文本特征和图像特征的维度。
- 在前向传播方法 forward 中, 通过ReLU激活函数对每个全连接层的输出进行激活。

```
class MultimodalModel(nn.Module):
    def __init__(self, input_size):
        super(MultimodalModel, self).__init__()
        # 输入层
        self.fc1 = nn.Linear(input_size, 512)
        # 隐藏层
        self.fc2 = nn.Linear(512, 256)
        # 输出层
        self.fc3 = nn.Linear(256, num_classes) # num_classes 是输出类别数量
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

## (5) 训练模型

### 1. 获取输入特征的大小和类别数量:

input\_size 表示模型输入的特征维度, 由合并后的文本和图像特征的形状确定。

num\_classes 表示输出的类别数量, 本次实验中为3类情感标签。

```
input_size = merged_features.shape[1] # 根据合并后的特征的形状确定
num_classes = 3 # 类别数量
```

### 2. 创建多模态模型的实例、定义损失函数和优化器:

```
multimodal_model = MultimodalModel(input_size)
# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(multimodal_model.parameters(), lr=0.0001)
```

### 3. 训练循环:

- 使用嵌套的循环来迭代数据集。外部循环是 epoch 的迭代，内部循环是按批次处理数据。
- 在每个批次内，提取输入和标签，清零梯度，进行前向传播，计算损失，反向传播并更新模型参数。
- 记录每个批次的训练损失到 train\_losses 列表中。

```
num_epochs = 10
# 记录损失值
train_losses = []
for epoch in range(num_epochs):
    for index in range(0, len(merged_features_tensor), 64):
        # 提取输入和标签
        inputs = merged_features_tensor[index:index+64]
        labels = train_labels[index:index+64]
        # 清除之前的梯度
        optimizer.zero_grad()
        # 前向传播
        outputs = multimodal_model(inputs)
        loss = criterion(outputs, labels)
        # 反向传播和优化
        loss.backward(retain_graph=True)
        optimizer.step()
        # 记录训练损失
        train_losses.append(loss.item())
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item()}')
```

## (6) 验证模型并计算在验证集上的准确率

- 使用 torch.max(outputs\_val, 1) 获取每个样本中预测概率最高的类别索引，并将其保存在 predicted\_val 中。
- 通过将预测结果 predicted\_val 与验证集的真实标签 true\_labels\_val 进行比较，计算正确预测的数量。
- 计算验证集的总样本数量，通过正确预测的数量除以总样本数量，计算模型在验证集上的准确率。

```
# 预测标签
with torch.no_grad():
    multimodal_model.eval()
    outputs_val = multimodal_model(merged_features_tensor_val)
    _, predicted_val = torch.max(outputs_val, 1)

# 计算准确率
correct_val = (predicted_val == torch.tensor(true_labels_val)).sum().item()
total_val = len(true_labels_val)
accuracy_val = correct_val / total_val
print(f'Validation Accuracy: {accuracy_val}')
```

## (7) 在测试集上进行预测

使用训练好的多模态模型对测试集进行预测，并输出每个样本的 guid 和模型预测的标签。

**#在测试集上预测结果**

```
with torch.no_grad():
    multimodal_model.eval()
    outputs_test = multimodal_model(merged_features_tensor_test)
    _, predicted_test = torch.max(outputs_test, 1)
for i in range(len(predicted_test)):
    guid_test = test_data.iloc[i]['guid']
    predicted_label_test = list(tag_mapping.keys())[predicted_test[i]]
    print(f"{int(guid_test)}, {predicted_label_test}")
```

## 三、实验bug

1.报错: Traceback (most recent call last):

File "D:\anaconda\Lib\site-packages\pandas\core\indexes\base.py", line 3653, in get\_loc KeyError: 'guid'

错误消息 (KeyError: 'guid') 表明我尝试访问的DataFrame中未包含名称为'guid'的列。

**解决方法:** 使用columns属性检查DataFrame的列，打印train\_data的列名print(train\_data.columns)

得到结果train\_data的列名: Index(['guid,tag'], dtype='object')说明 train\_data DataFrame 中的列名是一个名为 'guid,tag' 的单独字符串，而不是两个独立的列 'guid' 和 'tag'。这可能导致无法正确访问 'guid' 列。所以在读取 CSV 文件时正确指定了分隔符，' '。

2.报错Traceback (most recent call last): File "C:\Users\ls'y\Desktop\10215501401 陆思遥 实验五\代码\hw5.py", line 152, in loss.backward() File "D:\anaconda\Lib\site-packages\torch\tensor.py", line 492, in backward torch.autograd.backward( File "D:\anaconda\Lib\site-packages\torch\autograd\_\_init\_\_.py", line 251, in backward

错误消息: 由于在训练循环中在反向传播之前和之后两次调用了optimizer.zero\_grad(), 导致图的中间值被释放, 从而引发了RuntimeError。

**解决方法:** 我先尝试在反向传播之前删除多余的optimizer.zero\_grad(), 发现问题依然存在, 通过查询资料, 我使用retain\_graph=True: 在loss.backward()中, 添加retain\_graph=True参数, 这会保留计算图, 允许多次调用backward()。

3.报错IndexError: Target -9223372036854775808 is out of bounds.

错误消息表明目标值 (target) 中包含超出范围的索引。在交叉熵损失函数中, 目标值应该是类别的索引, 但出现了一个值 -9223372036854775808, 这是一个非法的索引。

**解决方法:** 检查标签数据类型print(train\_labels\_tensor.dtype)打印结果是torch.int64, 说明训练标签中未包含异常值, 经过检查发现是数据预处理时出现了问题, 'tag' 列包含字符串 (positive、neutral、negative), 而不是数值。在这种情况下, 我需要将这些字符串标签映射到数字标签, 以便在训练中使用, 所以创建了一个标签映射字典tag\_mapping, 将字符串标签 ('negative', 'neutral', 'positive') 映射为整数 (0, 1, 2)。

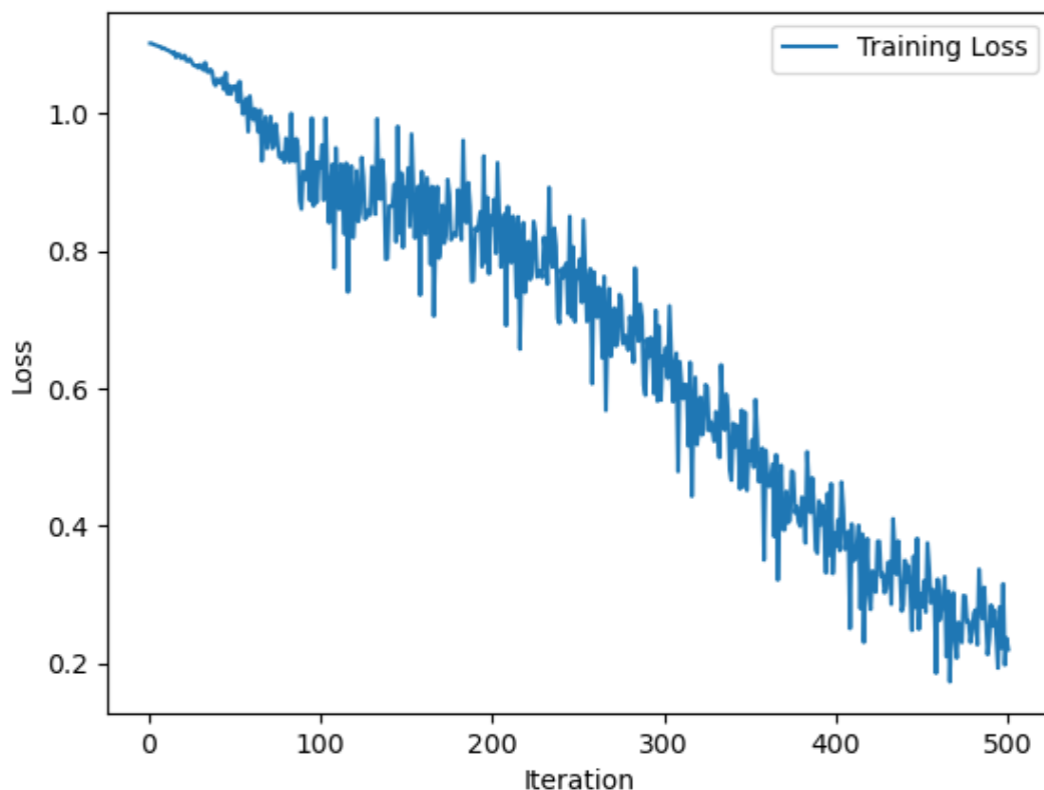


## 四、实验结果

1.多模态融合模型在验证集上的结果为0.645

```
D:\anaconda\python.exe "C:\Users\ls'y\Desktop\10215501401 陆思遥 实验五\代码\hw5.py"
数据预处理已完成
Epoch [1/10], Loss: 1.0367226600646973
Epoch [2/10], Loss: 0.9173918962478638
Epoch [3/10], Loss: 0.8810526132583618
Epoch [4/10], Loss: 0.8431866765022278
Epoch [5/10], Loss: 0.7680968642234802
Epoch [6/10], Loss: 0.6446924805641174
Epoch [7/10], Loss: 0.5005850791931152
Epoch [8/10], Loss: 0.3671616315841675
Epoch [9/10], Loss: 0.27040937542915344
Epoch [10/10], Loss: 0.20187224447727203
Validation Accuracy: 0.645
```

2.训练图像收敛结果如下图所示：



3.消融实验结果

将模型的输入层数据由合并后的特征merged\_features改为文本特征text\_features和图像特征image\_features

- 只输入图像数据



```
D:\anaconda\python.exe "C:\Users\ls'y\Desktop\10215501401 陆思遥 实验五\代码\val.py"
数据预处理已完成
Image-Only Validation Accuracy: 0.44125

Process finished with exit code 0
```

- 只输入文本数据

```
D:\anaconda\python.exe "C:\Users\ls'y\Desktop\10215501401 陆思遥 实验五\代码\val.py"
数据预处理已完成
Text-Only Validation Accuracy: 0.60375
```

4.预测结果：详见提交的results.txt文件

## 五、模型亮点

本次实验我构造的多模态融合模型是使用了较为简单易懂的特征融合方法，将不同模态的特征直接串联在一起。这种方法将不同模态的特征堆叠在一起形成一个更长的特征向量。例如，如果文本特征的维度是 $n$ ，图像特征的维度是 $m$ ，那么串联后的特征向量的维度就是 $n+m$ 。将融合后的特征作为多模态融合模型的输入层，得到最终预测结果。该模型的优势在于简单快捷，我用tf-idf提取文本特征，LeNet提取图像特征，运行效率很高，局限性在于它以一种直接的方式融合了两种模型特征生成高维向量，不能处理复杂的关系，所以准确率还有待提高。

## 六、总结

本次实验实现了一个简单的多模态融合模型，其中融合了文本和图像信息。多模态融合模型的关键目标是充分利用各种模态的信息，以提高模型在特定任务上的性能。模型结构可以根据任务的复杂性和数据特点而变化。