实验5-多模块情感分类

王洪宇 10205501459

一、实验目的

给定配对的文本和图像，预测对应的情感标签，三分类任务：positive, neutral, negative。

二、实验任务

设计一个多模态融合模型，自行从训练集中划分验证集，调整超参数，预测测试集（test_without_label.txt）上的情感标签。

三、实验环境

- win 10
- python 3.9
- numpy 1.25.4
- tensorflow 2.13.0
- pandas 1.3.5
- keras 2.13.1
- nltk 3.8.1
- torch 2.0.1
- torchvision 0.15.2
- sklearn 1.0.1

四、实验过程

4.1 数据预处理

通过 `get_dataset` 函数将提供的数值列表生成一个数据集，在该函数中根据获取到的 guid 得到对应文件的路径，获取其文件编码格式后读取文件内容，加入到 text 当中，最终将获取到的数据样本的 `text`、`label` 以及相关的图像路径作为一个字典添加到 `dataset` 列表中，并返回该列表

```
def get_dataset(values):
    dataset = []
    for i in range(len(values)):
        guid = str(int(values[i][0]))
        label = values[i][1]
        if type(label) != str and math.isnan(label):
            label = None
        file = path_text + guid + '.txt'
        with open(file, 'rb') as f:
            encoding = chardet.detect(f.read())['encoding']
            if encoding == "GB2312":
                encoding = "GBK"
        text = ''
        try:
            with open(file, encoding=encoding) as fp:
                for line in fp.readlines():
                    line = line.strip('\n')
                    text += line
```

```
        except UnicodeDecodeError:
            try:
                with open(file, encoding='ANSI') as fp:
                    for line in fp.readlines():
                        line = line.strip('\n')
                        text += line
            except UnicodeDecodeError:
                print('UnicodeDecodeError')
        dataset.append({
            'text': text,
            'label': label,
            'img': path_text + guid + '.jpg',
        })
    return dataset
```

按照 8：2 的比例划分出训练集、验证集，并将训练集、验证集、测试集转化为所需要的格式后保存为 json 文件

```
train_data, val_data = train_test_split(train_data_val, test_size=0.2)
train_set = get_dataset(train_data.values)
val_set = get_dataset(val_data.values)
test_set = get_dataset(test_data.values)

with open('train.json', 'w', encoding="utf-8") as f:
    json.dump(train_set, f)
with open('val.json', 'w', encoding="utf-8") as f:
    json.dump(val_set, f)
with open('test.json', "w", encoding="utf-8") as f:
    json.dump(test_set, f)
```

4.2 构建词汇表

创建一个 `Counter` 对象 `words`，用于统计每个单词在训练集中出现的次数。然后，遍历训练集中的每一个句子，将句子进行分词，统计每个单词在训练集中出现的频率，并将分词后的结果存储在 `train_text_list` 中

```
words = Counter()
i=0
for text in train_text_list:
    words_list = nltk.word_tokenize(text)
    words.update(words_list)
    train_text_list[i] = words_list
    i+=1
```

将词频小于等于1的单词从列表 `words` 中删除，以避免在构建词汇表时浪费内存和计算资源，将单词按照词频从高到低排序，并在单词列表的开头添加一个特殊的标记'_PAD'，用于填充句子以使它们具有相同的长度。

```
words = {k:v for k,v in words.items() if v>1}
words = sorted(words, key=words.get,reverse=True)
words = ['_PAD'] + words
```

使用 `word2idx` 和 `idx2word` 两个字典，将单词映射为对应的数字和数字映射为对应的单词

```
word2idx = {o:i for i,o in enumerate(words)}
idx2word = {i:o for i,o in enumerate(words)}
```

对训练集的所有句子中的每个单词替换为对应的数字

```
i=0
for text in train_text_list:
    train_text_list[i] = [word2idx[word] if word in word2idx else 0 for word in
text]
    i+=1
```

对验证集执行同样的操作

定义`padding`，用于对数字化后的文本进行填充操作，使得所有文本序列具有相同的长度，调用该函数对输入的序列进行填充

```
def padding(text_list, seq_len):
    features = np.zeros((len(text_list), seq_len),dtype=int)
    i=0
    for text in text_list:
        features[i, -len(text):] = np.array(text)[:seq_len]
        i+=1
    return features

train_text = padding(train_text_list, 200)
val_text = padding(val_text_list, 200)
```

调用`change_label`将输入的情绪标签转化为对应的数字

```
def change_label(data_labels):
  for i in data_labels:
    if(data_labels[i]=='negative'):
      data_labels[i]=0
    elif(data_labels[i]=='positive'):
        data_labels[i]=1
    elif(data_labels[i]=='neutral'):
        data_labels[i]=2
  return data_labels

train_labels = np.array(change_label(train_labels))
val_labels = np.array(change_label(val_labels))
```

4.3 消融实验：仅文本训练

将训练集和验证集的输入和标签组合成一个数据集对象`train_Data`和`val_Data`

```
train_Data = TensorDataset(torch.from_numpy(train_text),
torch.from_numpy(train_labels))
val_Data = TensorDataset(torch.from_numpy(val_text),
torch.from_numpy(val_labels))
```

创建训练集和验证集的数据加载器`train_loader`和`val_loader`，将数据集分成小批量进行训练和验证

```
train_Data = TensorDataset(torch.from_numpy(train_text),
torch.from_numpy(train_labels))
val_Data = TensorDataset(torch.from_numpy(val_text),
torch.from_numpy(val_labels))
```

通过 TextCNN 的卷积神经网络模型，用于仅文本训练的框架

`GlobalMaxPool1d` 类定义了一个全局最大池化操作的模块，将一维输入张量上执行最大池化操作

```
class GlobalMaxPool1d(nn.Module):
    def __init__(self):
        super(GlobalMaxPool1d, self).__init__()
    def forward(self, x):
        return F.max_pool1d(x, kernel_size=x.shape[2])
```

`TextCNN` 类定义了文本分类的卷积神经网络模型，它在文本处理中有一些设计上的优点和亮点：

1. 多尺度卷积：TextCNN通过使用不同大小的卷积核来捕捉不同尺度的局部特征。这些卷积核在不同的窗口大小上滑动，从而能够提取出文本中不同长度的关键特征，增强了模型对于不同长度文本的理解能力。
2. 全局最大池化：TextCNN使用全局最大池化操作来提取特征。这种池化操作能够从每个卷积层输出的特征图中选择最重要的特征，从而保留了文本中最显著的特征。全局最大池化操作还能够使模型对输入文本的长度不敏感，使得模型能够处理不同长度的文本输入。
3. 多通道输入：TextCNN将词向量表示和常数向量表示拼接在一起作为模型的输入。词向量表示捕捉了词语的语义信息，而常数向量表示则可以用来表示词语的位置或其他额外的信息。通过使用多通道输入，模型可以同时考虑词语的语义和位置信息，增强了模型的表达能力。
4. 多通道卷积：TextCNN通过使用多个卷积核，并在每个卷积核上应用多通道输入，可以提取更丰富的特征表示。不同的卷积核可以捕捉不同类型和尺度的特征，从而增强了模型对于文本特征的建模能力。

其中的 `__init__` 方法初始化模型的各个组件。

- `embedding` 是一个词嵌入层，用于将输入的词索引转换为词向量表示
- `constant_embedding` 是另一个词嵌入层，它的权重在训练过程中不会更新，用于保留一些固定的词向量表示
- `pool` 是一个 `GlobalMaxPool1d` 类的实例，用于执行全局最大池化操作
- `convs` 是一个 `nn.ModuleList`，其中每个元素都是一个一维卷积层（`nn.Conv1d`）。每个卷积层有不同的输入通道数、输出通道数和卷积核大小，通过循环遍历 `kernel_sizes` 和 `num_channels` 来创建这些卷积层
- `decoder` 是一个线性层，用于将卷积层的输出映射到类别标签的预测
- `dropout` 是一个丢弃层，用于在训练过程中随机丢弃部分神经元，以减少过拟合风

```
class TextCNN(nn.Module):
    def __init__(self, vocab, embed_size, kernel_sizes, num_channels):
        super(TextCNN, self).__init__()
        self.embedding = nn.Embedding(len(vocab), embed_size)
        self.constant_embedding = nn.Embedding(len(vocab), embed_size)

        self.pool = GlobalMaxPool1d()
        self.convs = nn.ModuleList()
        for c, k in zip(num_channels, kernel_sizes):
            self.convs.append(nn.Conv1d(in_channels = 2*embed_size,
                        out_channels = c,
                        kernel_size = k))
```

```
        self.decoder = nn.Linear(sum(num_channels), 3)
        self.dropout = nn.Dropout(0.3)
```

`forward` 方法定义了模型的前向传播过程。

- 首先，将输入的词索引经过嵌入层和常数嵌入层得到词向量表示，并在通道维度上进行拼接
- 然后，将拼接后的词向量转置，成为卷积层的输入通道维度
- 接下来，通过循环遍历 `self.convs`，对每个卷积层执行卷积操作，并通过 `self.pool` 进行全局最大池化操作，得到每个卷积核特征表示，并将它们在通道维度上进行拼接。
- 将拼接后的特征表示经过丢弃层后，通过线性层进行分类预测，并返回预测结果

```
    def forward(self, inputs):
        embeddings = torch.cat((
            self.embedding(inputs),
            self.constant_embedding(inputs)), dim=2)
        embeddings = embeddings.permute(0, 2, 1)
        encoding = torch.cat([
            self.pool(F.relu(conv(embeddings))).squeeze(-1) for conv in
self.convs], dim=1)
        outputs = self.decoder(self.dropout(encoding))
        return outputs
```

创建一个 `TextCNN` 类的实例，传入相应的参数来构建该模型，并将其赋值给 `net` 变量

```
net = TextCNN(words, embed_size, kernel_sizes, nums_channels)
```

定义训练过程，用于训练模型

- 在每个轮次中，通过循环遍历训练数据迭代器，依次获取输入 `X` 和标签 `y`。
- 使用模型 `net` 对输入 `X` 进行前向传播，得到预测结果 `y_hat`。
- 计算预测结果 `y_hat` 和标签 `y` 之间的损失，使用损失函数 `loss` 进行计算。
- 将优化器的梯度置零，然后进行反向传播计算梯度，并调用优化器的 `step` 方法来更新模型的参数。
- 计算训练损失和训练准确率，并累加到 `train_l_sum` 和 `train_acc_sum` 中，同时记录样本数量 `n`。
- 在每个批次训练结束后，调用 `evaluate_accuracy` 函数对测试数据进行评估

```
def train(train_iter, test_iter, net, loss, optimizer,  num_epochs):
    print("training start")
    batch_count = 0
    for epoch in range(num_epochs):
        train_l_sum, train_acc_sum, n, start = 0.0, 0.0, 0, time.time()
        for X, y in train_iter:
            y_hat = net(X)
            l = loss(y_hat, y.to(torch.int64))
            optimizer.zero_grad()
            l.backward()
            optimizer.step()
            train_l_sum += l.cpu().item()
            train_acc_sum += (y_hat.argmax(dim=1) == y).sum().cpu().item()
            n += y.shape[0]
            batch_count += 1
        test_acc = evaluate_accuracy(test_iter, net)
```

```
        print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f, time %.1f
sec'
              % (epoch + 1, train_l_sum / batch_count, train_acc_sum / n,
test_acc, time.time() - start))
```

创建了一个优化器 `optimizer`，使用了Adam优化算法（`torch.optim.Adam`）来更新模型参数。

- `filter(lambda p: p.requires_grad, net.parameters())` 用于获取所有需要梯度更新的模型参数。
- 通过传入这些参数和学习率 `lr` 给Adam优化器进行初始化

```
optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad,
net.parameters()), lr=lr)
```

定义了损失函数 `loss`，使用了交叉熵损失函数（`nn.CrossEntropyLoss`），该损失函数适用于多类别分类任务

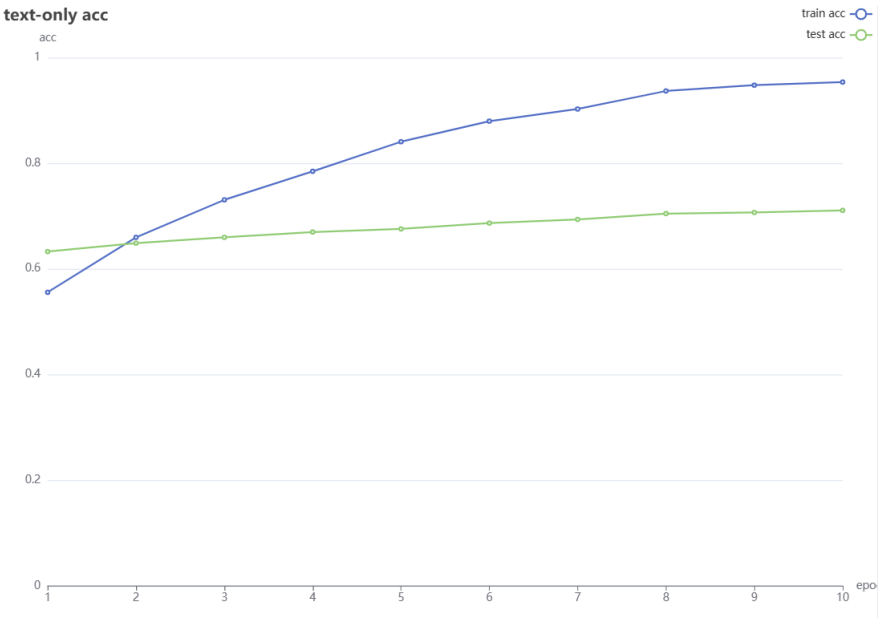
```
loss = nn.CrossEntropyLoss()
```

进行模型的训练，传入训练数据迭代器 `train_loader`、验证数据迭代器 `val_loader`、模型 `net`、损失函数 `loss`、优化器 `optimizer` 和总轮数 `num_epochs`。

```
train(train_loader, val_loader, net, loss, optimizer, num_epochs)
```

训练结果如下图所示，

```
text-only training start
epoch 1, loss 0.9825, train acc 0.556, test acc 0.633, time 26.9 sec
epoch 2, loss 0.3883, train acc 0.660, test acc 0.649, time 39.2 sec
epoch 3, loss 0.2196, train acc 0.731, test acc 0.660, time 38.1 sec
epoch 4, loss 0.1417, train acc 0.785, test acc 0.670, time 26.3 sec
epoch 5, loss 0.0968, train acc 0.841, test acc 0.676, time 25.3 sec
epoch 6, loss 0.0667, train acc 0.880, test acc 0.687, time 28.6 sec
epoch 7, loss 0.0478, train acc 0.903, test acc 0.694, time 44.2 sec
epoch 8, loss 0.0340, train acc 0.937, test acc 0.705, time 34.3 sec
epoch 9, loss 0.0252, train acc 0.948, test acc 0.707, time 30.9 sec
epoch 10, loss 0.0192, train acc 0.954, test acc 0.711, time 42.8 sec
```



可以看到在多轮迭代过程之后，验证集上的 acc 准确率接近稳定，最终的验证集准确率为 0.711

4.4 消融实验：仅图片训练

定义自定义的数据集类 `ToDataset`，用于加载和预处理图像数据集，可以通过实例化对象来加载图像数据集，并在训练或测试过程中使用数据加载器将数据提供给模型。它将图像预处理的逻辑封装在 `data_process` 方法中，该方法对图像转化为张量并进行归一化处理，使得数据处理过程更加方便和可复用

```
    def data_process(self,x):
        return transforms.Compose(
        [
            transforms.Resize((256,256)),
            transforms.ToTensor(),
            transforms.Normalize(
                mean=[0.5,0.5,0.5],
                std=[0.5,0.5,0.5],
            ),
        ]
    )(x)
```

创建数据加载器，用于批量加载训练集、验证集和测试集的数据

```
train_loader = DataLoader(dataset=ToDataset(train_data), batch_size=batch_size,
shuffle=True)
val_loader = DataLoader(dataset=ToDataset(val_data), batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(dataset=ToDataset(test_data), batch_size=batch_size)
```

通过 CNN 的卷积神经网络模型，用于仅图片训练的框架，该卷积神经网络模型具有两个卷积层和一个全连接层，通过卷积操作提取图像特征，并通过全连接层将提取的特征映射到输出类别的预测，该模型有以下的优点：

1. 卷积层和池化层：模型使用了两个卷积层和池化层的序列，这种设计能够有效地从图像中提取特征。卷积层通过卷积操作对图像进行特征提取，而池化层则可以减小特征图的尺寸，保留重要的特征，并且具有平移不变性。
2. ReLU激活函数：在卷积层之后，使用了ReLU激活函数，这可以引入非线性，使得模型能够学习更复杂的特征表示。ReLU激活函数的优点包括计算简单、减少梯度消失问题和增强模型的表达能力。
3. 全连接层：在卷积层之后，模型通过全连接层将提取的特征映射到输出类别的预测。全连接层能够对特征进行线性组合和映射，从而实现分类或回归的任务。
4. 参数数量较少：相对于全连接层，卷积层具有共享权重的特点，这可以大大减少模型中的参数数量。通过卷积和池化的操作，模型能够以较少的参数学习到更具有鲁棒性和泛化能力的特征表示。

其中的 `forward` 方法定义了模型的前向传播过程。

- 输入 `x` 经过 `conv1` 进行卷积、激活函数和池化操作
- 将卷积层的输出再经过 `conv2` 进行卷积、激活函数和池化操作
- 将最后一层池化层的输出展平为一维向量，通过调用 `view` 方法，并指定批次大小 `x.size(0)` 和展平后的特征数量 `-1`
- 将展平后的特征向量传入全连接层 `output`，得到最终的输出结果

```
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)
        output = self.output(x)
        return output
```

创建了一个 Adam 优化器，用于更新模型 cnn 中的参数

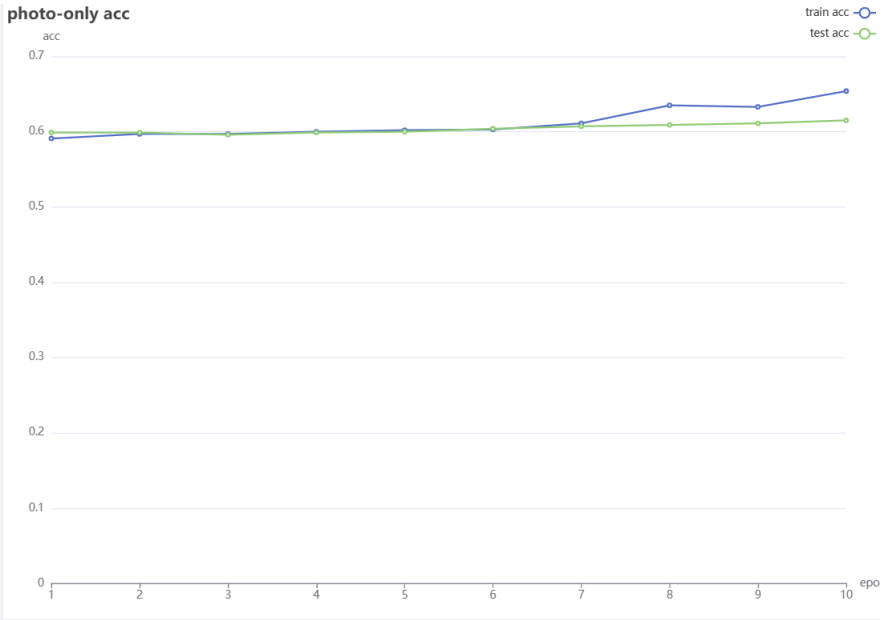
```
optimizer = torch.optim.Adam(cnn.parameters(), lr=lr)
```

进行模型的训练，传入训练数据迭代器 `train_loader`、验证数据迭代器 `val_loader`、模型 `cnn`、损失函数 `loss`、优化器 `optimizer` 和总轮数 `num_epochs`

```
train(train_loader, val_loader, cnn, loss, optimizer, num_epochs)
```

训练结果如下图所示

```
photo-only training start
epoch 1, loss 0.9144, train acc 0.591, test acc 0.599, time 246.8 sec
epoch 2, loss 0.4454, train acc 0.597, test acc 0.599, time 180.9 sec
epoch 3, loss 0.2936, train acc 0.597, test acc 0.596, time 191.8 sec
epoch 4, loss 0.2179, train acc 0.600, test acc 0.599, time 188.0 sec
epoch 5, loss 0.1728, train acc 0.602, test acc 0.600, time 179.0 sec
epoch 6, loss 0.1414, train acc 0.603, test acc 0.604, time 154.6 sec
epoch 7, loss 0.1193, train acc 0.611, test acc 0.607, time 146.9 sec
epoch 8, loss 0.1016, train acc 0.635, test acc 0.609, time 147.3 sec
epoch 9, loss 0.0882, train acc 0.633, test acc 0.611, time 137.0 sec
epoch 10, loss 0.0758, train acc 0.654, test acc 0.615, time 132.9 sec
```



可以看到在多轮迭代过程之后，验证集上的 acc 准确率接近稳定，最终的验证集准确率为 0.615

4.4 多模态融合模型

定义的`multiModel` 函数根据文本预测结果 `predict_text` 和图像预测结果 `predict_pic` 进行判断，得到最终的预测结果，根据文本训练的结果由于图片训练的结果的情况，当二者显示到的预测结果不同时，优先采用文本预测的结果最为最终的预测结果，全部预测完成后返回最终的预测结果，在验证集上计算准确率

```
val_predict = multiModel(val_data,val_loader)
acc_count = 0
for i in range(len(val_data)):
  if val_predict[i] == val_data[i]["label"]:
    acc_count += 1
print(acc_count/len(val_data))
```

✓ 27.0s

```
0.6875
```

得到的验证集准确率为0.6875，介于仅文本训练和仅图片训练的验证集准确率之间

在测试集上进行预测，并将最终的结果保存在 `test_without_label.txt` 中，将该文件重命名为 `results.txt` 后提交

```
test_predict = multiModel(test_data,test_loader)
test_data_file = pd.read_csv("test_without_label.txt")['guid'].values
with open('test_without_label.txt','w') as f:
  f.write('guid,tag\n')
  for i in range(len(test_data_file)):
    f.write(str(test_data_file[i])+','+str(test_predict[i])+'\n')
f.close()
```

4.5 代码中遇到的问题

在消融实验时，如何对仅文本训练的样本和仅图片训练的样本选取模型，在查阅了资料后，最终分别选择了 TextCNN 和 CNN 模型来对两组样本进行训练。对于文本数据，TextCNN模型是一个常用的选择；对于图片数据，CNN模型也是一种常见的选择。

在得到训练好的模型之后，并不能简单地利用 `model.predict()` 方法来进行预测，此处模型的训练方法和预测方法都是编写的函数，而不是简单地调用，最终在查阅了资料的基础上完成了这些函数的编写。总之，在进行预测时，需要根据训练的模型结构和输入方式，以及待预测的数据类型（文本或图片）选择合适的预测方式。