1. 环境要求

本次实验用的 IDE：Pycharm
本次实验的系统：windows11
本次实验利用的云主机：Colab
本次实验用到的库：见 requirements.txt
Github 地址：https://github.com/TheadoraTang/AI-Project5

2. 使用方法

pip install -r requirements.txt
python deal_data.py
python multi.py

3. 实验过程

3.1 实验目的

本次实验需要通过实现一个多模态融合模型，通过输入图片和文字来预测测试集（test_without_label.txt）上的情感标签。同时还需要进行消融实验来检验模型对文字和图片的训练程度和成果。

3.2 实验思路

本次实验需要实现多模态融合模型，多模态是指两个或者两个以上的模态的各种形式的组合。对每一种信息的来源或者形式，都可以称为一种模态（Modality），目前研究领域中主要是对图像，文本，语音三种模态的处理。之所以要对模态进行融合，是因为不同模态的表现方式不一样，看待事物的角度也会不一样，所以存在一些交叉（所以存在信息冗余），互补（所以比单特征更优秀）的现象，甚至模态间可能还存在多种不同的信息交互，如果能合理的处理多模态信息，就能得到丰富特征信息。

首先对于图片和文字的输入需要进行数据预处理，然后对文字和图像分别运用不同的模

型进行处理，最后使用损失函数进行模型训练，采用 Adam 优化器更新模型参数。在每个 epoch 结束后，评估模型在验证集上的性能，并输出损失值和准确率。

3.3 代码实现

首先介绍一下数据预处理的代码 deal_data.py

这段代码的作用是将给定数据集中的文本和图像文件转换为 json 格式，并分别保存为训练、验证和测试文件。具体实现流程如下：

1. 读取文本和图像数据集：

从 `train.txt` 和 `test_without_label.txt` 文件中读取数据，其中包含了每个样本的 id、标签和文本内容。

通过 `train_test_split` 函数将训练集划分为训练集和验证集。

```
train_dev_df = pd.read_csv(train_txt_path)
test_df = pd.read_csv(test_txt_path)
train_df, dev_df = train_test_split(train_dev_df, test_size=arguments.dev_size)
```

这两个函数的作用是将文本文件读取并处理为可用于模型训练的数据格式。具体介绍如下：

2. read_text_file(file, encoding) 函数：

参数 `file` 是要读取的文件路径，`encoding` 是文件的编码方式。该函数通过打开文件，并指定正确的编码方式来读取文件内容。使用 `fp.readlines()` 逐行读取文件内容，并通过 `strip('\n')` 去除每行末尾的换行符。将每行内容拼接到 `text` 字符串中。最后返回拼接的文本内容。

```
def read_text_file(file, encoding):
    text = ''
    with open(file, encoding=encoding) as fp:
        for line in fp.readlines():
            line = line.strip('\n')
            text += line
    return text
```

3. transform_data(data_values) 函数：

参数 `data_values` 是包含数据样本信息的列表或数组。该函数遍历每个数据样本，进行数据转换和处理。将样本的 id 转换为字符串类型，并赋值给 `guid` 变量。检查标签的类型，如果不是字符串且是 NaN 值，则将标签置为 None。根据样本的 id 构建对应的文本文件路径 `file_path`。使用 `chardet` 库检测文本文件的编码方式，并根据需要进行编码方式的修正（将 GB2312 编码改为 GBK）。调用 `read_text_file` 函数读取文本文件的内容，并将结果赋值给 `text` 变量。如果在读取过程中出现 UnicodeDecodeError 错误，尝试用 'ANSI' 编码方式再次读取。将样本的 id、文本内容、标签和图像路径组成一个字典，并添加到 `dataset` 列表中。最后返回包含所有样本信息的 `dataset` 列表。

```python
def transform_data(data_values):
    dataset = []
    for i in range(len(data_values)):
        guid = str(int(data_values[i][0]))
        label = data_values[i][1]
        if type(label) != str and math.isnan(label):
            label = None

        file_path = data_path + guid + '.txt'
        with open(file_path, 'rb') as f:
            encoding = chardet.detect(f.read())['encoding']
            if encoding == "GB2312":
                encoding = "GBK"

        text = ''
        try:
            text = read_text_file(file_path, encoding)
        except UnicodeDecodeError:
            try:
                text = read_text_file(file_path, 'ANSI')
            except UnicodeDecodeError:
                print('UnicodeDecodeError')
        dataset.append({
            'guid': guid,
            'text': text,
            'label': label,
            'image_path': data_path + guid + '.jpg',
        })
    return dataset
```

这两个函数的完成了对文本文件的读取和处理操作，将原始数据转换为模型可以直接使用的格式。`read_text_file` 函数负责具体的文件读取操作，而 `transform_data` 函数则是根据文件中的信息构建样本字典，并进行一些数据清洗和处理操作。这样，就能更好地将原始数据转换为适合模型训练的数据集。

接下来是模型部分的代码

1. 定义数据集：

__init__(self, args, data, transform=None)方法：

`args` 是包含训练/测试参数的命名空间对象。

`data` 是包含数据样本信息的列表或数组。

`transform` 是可选的图像增强方法，用于对样本中的图像进行处理。

该方法在初始化时完成了以下操作：

保存 `args`、`data` 和 `transform` 这三个输入参数。

初始化 `RobertaTokenizer` 对象，用于对文本进行分词和编码。

构建标签和数字之间的转换字典 `label_dict_number` 和 `label_dict_str`。

`__len__(self)` 方法：

返回数据集样本数量。

`__getitem__(self, index)` 方法：

参数 `index` 是样本索引。该方法用于获取指定索引处的数据样本，并将其转换为模型可以接受的格式。首先通过 `tokenize` 方法对样本进行分词和编码，得到文本和图像的 token 表示、样本 id 和标签的数字表示。然后将分词编码后的文本、图像的 tensor、样本 id 和标签数字作为 tuple 返回。

`tokenize(self, item)` 方法：

参数 `item` 是包含数据样本信息的字典。该方法用于对单个样本进行分词和编码，得到模型可以接受的 token 表示。首先从 `item` 字典中提取出样本 id、文本内容、图像路径和标签信息。然后使用 `RobertaTokenizer` 对文本进行分词和编码，得到文本的 token 表示。将文本的 input_ids 和 attention_mask 分别压缩为一维张量，并将它们保存到 `text_token` 字典中。 如果 `transform` 存在，则将图像路径转换为 tensor 格式并进行增强处理。否则将图像路径直接转换为 tensor 格式。将标签转换为数字表示，并返回样本 id、文本的 token 表示、图像 tensor 和标签的数字表示。

```python
class MyDataset(Dataset):
    def __init__(self, args, data, transform=None):
        self.args = args
        self.data = data
        self.transform = transform
        self.tokenizer = RobertaTokenizer.from_pretrained(args.pretrained_model)
        self.label_dict_number = {
            'negative': 0,
            'neutral': 1,
            'positive': 2,
        }
        self.label_dict_str = {
            0: 'negative',
            1: 'neutral',
            2: 'positive',
        }

    def __getitem__(self, index):
        return self.tokenize(self.data[index])

    def __len__(self):
        return len(self.data)

    def tokenize(self, item):
        item_id = item['id']
        text = item['text']
        img = item['img']
        label = item['label']

        text_token = self.tokenizer(text, return_tensors="pt", max_length=self.args.text_size,
                                    padding='max_length', truncation=True)
        text_token['input_ids'] = text_token['input_ids'].squeeze()
        text_token['attention_mask'] = text_token['attention_mask'].squeeze()

        img_token = self.transform(img) if self.transform else torch.tensor(img)

        label_token = self.label_dict_number[label] if label in self.label_dict_number else -1
        return item_id, text_token, img_token, label_token
```

2. 加载 json 数据

创建一个空列表`data_list`，用于存储处理后的数据。然后打开指定路径的 JSON 文件，使用 UTF-8 编码方式读取文件内容并使用 json.load 方法加载 JSON 文件内容，将其转换为

Python 数据结构。加载后的数据存储在 lines 变量中。最后遍历加载的 JSON 数据（每个 line 表示 JSON 中的一行），将每行的图像路径、文本、标签和 ID 组成一个字典（item），并将该字典添加到 data_list 列表中。

```python
def load_json(file):
    data_list = []
    with open(file, 'r', encoding='utf-8') as f:
        lines = json.load(f)
        for line in lines:
            item = {
                'img': np.array(Image.open(line['img'])),
                'text': line['text'],
                'label': line['label'],
                'id': line['guid']
            }
            data_list.append(item)
    return data_list
```

3. 加载数据

img_size = (args.img_size, args.img_size)：设置图像的大小，将其转换为元组。

data_transform = transforms.Compose([...])`：定义了一系列图像转换操作，包括将图像转为张量、调整图像大小、进行归一化等。

data_list = {...}：创建一个字典 data_list 来存储数据集的文件路径或内容，键是数据集的名称（如 'train'、'dev'、'test'），值是相应数据集的文件路径。

data_set = {...}：创建一个字典 data_set 来存储数据集的 MyDataset 对象，键是数据集的名称，值是通过调用 MyDataset 类创建的对象。其中，使用 data_transform 对图像进行转换。

```python
def load_data(args):
    img_size = (args.img_size, args.img_size)
    data_transform = transforms.Compose(
        [transforms.ToTensor(),
         transforms.Resize(img_size),
         transforms.Normalize([0.5], [0.5])]
    )
    data_list = {
        'train': load_json(args.train_file),
        'dev': load_json(args.dev_file),
        'test': args.test_file and load_json(args.test_file),
    }
    data_set = {
        'train': MyDataset(args, data_list['train'], transform=data_transform),
        'dev': MyDataset(args, data_list['dev'], transform=data_transform),
        'test': args.test_file and MyDataset(args, data_list['test'], transform=data_transform),
    }
    return data_set[args.mode], data_set['dev']
```

4. 保存数据

将预测出的测试集的结果输出到 txt 文本中。

```python
def save_data(file, predict_list):
    with open(file, 'w', encoding='utf-8') as f:
        f.write('guid,tag\n')
        for pred in predict_list:
            f.write(f"{pred['guid']},{pred['tag']}\n")
```

5. 仅文本模型

__init__(self, args) 方法：

初始化函数，接受一个参数 args，该参数包含了模型的相关配置。在方法中，首先调用 super() 来继承父类 nn.Module 的属性和方法。创建了一个名为 encoder 的预训练 RoBERTa 模型，使用了 from_pretrained()方法，并通过指定 pretrained_model 参数来指定预训练模型的名称或路径。遍历 encoder 中的参数，并将它们的 requires_grad` 属性设置为 `True`，表示这些参数需要进行梯度计算和更新。创建了一个名为 `transform` 的线性变换层，将 RoBERTa 的输出特征张量转换为一个更低维度的表示。

forward(self, encoded_input)方法：

前向传播函数，接受一个经过编码的文本输入 encoded_input，将编码好的文本输入 encoded_input 传递给 encoder 模型，即通过 RoBERTa 模型对输入进行编码。从 encoder_output 中提取出最后一层的隐藏状态和池化输出，并将池化输出通过 transform 层进行线性变换和非线性激活，得到最终的文本表示。最后将提取得到的隐藏状态张量和经过线性变换得到的文本表示张量返回作为模型的输出。

```python
class TextOnly(nn.Module):
    def __init__(self, args):
        super(TextOnly, self).__init__()
        self.encoder = RobertaModel.from_pretrained(args.pretrained_model)
        for param in self.encoder.parameters():
            param.requires_grad = True
        self.transform = nn.Sequential(
            nn.Linear(768, 1000),
            nn.ReLU(),
        )

    def forward(self, encoded_input):
        encoder_output = self.encoder(**encoded_input)
        hidden_state = encoder_output['last_hidden_state']
        pooler_output = encoder_output['pooler_output']
        output = self.transform(pooler_output)
        return hidden_state, output
```

6. 仅图像模型

__init__(self, args)方法：

初始化函数，接受一个参数 args，该参数包含了模型的相关配置。首先调用 super() 来继承父类 nn.Module 的属性和方法。创建了卷积神经网络模型，使用了 convnext.convnext_base 函数，并通过指定 weights 参数来选择模型的初始化权重。遍历 encoder 中的参数，并将它们的 requires_grad 属性设置为 True，表示这些参数需要进行梯

度计算和更新。

forward(self, x)方法：

前向传播函数，接受一个输入张量 x。将输入张量 x 传递给 encoder 模型，即通过卷积和池化等层对输入进行特征提取和转换。最后将得到的特征张量 x 返回作为模型的输出。

```
class ImgOnly(nn.Module):
    def __init__(self, args):
        super(ImgOnly, self).__init__()
        self.encoder = convnext.convnext_base(weights=convnext.ConvNeXt_Base_Weights.DEFAULT)
        for param in self.encoder.parameters():
            param.requires_grad = True

    def forward(self, x):
        x = self.encoder(x)
        return x
```

7. 多模态模型

__init__(self, args)`方法：

初始化函数，接受一个参数 `args`，该参数包含了模型的相关配置。在方法中，首先调用 `super()` 来继承父类 `nn.Module` 的属性和方法。创建了两个子模块 `TextModule_` 和 `ImgModule_`，分别是前面定义的 `TextOnly` 类和 `ImgOnly` 类的实例，用于处理文本输入和图像输入。定义了一些线性层，用于进行特征变换和分类。创建了一个多头注意力层 `multihead_attn`，用于对多模态特征进行自注意力计算。创建了一个 Transformer 编码器层 `transformer_encoder`，用于对多模态特征进行编码。

forward(self, bach_text=None, bach_img=None)方法：

前向传播函数，接受两个输入参数 `bach_text` 和 `bach_img`，分别表示文本输入和图像输入的批量数据。在方法中，根据输入的情况分别处理文本输入、图像输入和多模态输入。如果只有文本输入，则将文本输入传递给 `TextModule_` 进行编码，并将编码后的特征传递给文本分类器 `classifier_text` 进行分类。如果只有图像输入，则将图像输入传递给 `ImgModule_` 进行编码，并将编码后的特征传递给图像分类器 `classifier_img` 进行分类。如果同时有文本输入和图像输入，则分别将它们传递给对应的模块进行编码，并将编码后的特征拼接起来作为多模态特征。将多模态特征分别传递给文本分类器 `classifier_text`、图像分类器 `classifier_img` 和多模态分类器 `classifier_multi` 进行分类。返回多模态分类器的输出。

Multihead_self_attention(self, text_out, img_out)方法：

用于实现多模态自注意力计算。首先通过线性变换层将文本特征 `text_out` 和图像特征 `img_out` 分别映射到相同的维度。将映射后的特征按照维度进行堆叠，得到键（key）、值（value）和查询（query）。使用多头注意力层 `multihead_attn` 对查询、键和值进行 attention 计算，得到注意力输出。返回注意力输出。

Transformer_Encoder(self, text_out, img_out)` 方法：

用于实现多模态特征的 Transformer 编码。首先将文本特征 `text_out` 和图像特征 `img_out` 按照维度进行堆叠，得到多模态序列。使用 Transformer 编码器层 `transformer_encoder` 对多模态序列进行编码。返回编码后的特征。

```python
class MultiModal(nn.Module):
    def __init__(self, args):
        super(MultiModal, self).__init__()
        self.TextModule_ = TextOnly(args)
        self.ImgModule_ = ImgOnly(args)

        self.multihead_attn = nn.MultiheadAttention(embed_dim=1000, num_heads=2, batch_first=True)
        self.linear_text_k1 = nn.Linear(1000, 1000)
        self.linear_text_v1 = nn.Linear(1000, 1000)
        self.linear_img_k2 = nn.Linear(1000, 1000)
        self.linear_img_v2 = nn.Linear(1000, 1000)

        self.encoder_layer = nn.TransformerEncoderLayer(d_model=1000, nhead=2, batch_first=True)
        self.transformer_encoder = nn.TransformerEncoder(self.encoder_layer, num_layers=2)

        self.classifier_img = nn.Sequential(
            nn.Linear(1000, 500),
            nn.ReLU(),
            nn.Linear(500, 200),
            nn.ReLU(),
            nn.Linear(200, 40),
            nn.ReLU(),
            nn.Linear(40, 3),
        )
        self.classifier_text = nn.Sequential(
            nn.Linear(1000, 500),
            nn.ReLU(),
            nn.Linear(500, 200),
            nn.ReLU(),
            nn.Linear(200, 40),
            nn.ReLU(),
            nn.Linear(40, 3),
        )
```

```
    self.classifier_multi = nn.Sequential(
        nn.Flatten(),
        nn.Linear(2000, 1000),
        nn.ReLU(),
        nn.Dropout(p=args.dropout),
        nn.Linear(1000, 200),
        nn.ReLU(),
        nn.Linear(200, 40),
        nn.ReLU(),
        nn.Linear(40, 3),
    )

def forward(self, bach_text=None, bach_img=None):
    if bach_text is not None and bach_img is None:
        _, text_out = self.TextModule_(bach_text)
        text_out = self.classifier_text(text_out)
        return text_out, None, None

    if bach_text is None and bach_img is not None:
        img_out = self.ImgModule_(bach_img)
        img_out = self.classifier_img(img_out)
        return None, img_out, None

    _, text_out = self.TextModule_(bach_text)
    img_out = self.ImgModule_(bach_img)

    multi_out = torch.cat((text_out, img_out), 1)

    # multi_out = self.Multihead_self_attention(text_out, img_out)

    # multi_out = self.Transformer_Encoder(text_out, img_out)

    text_out = self.classifier_text(text_out)
    img_out = self.classifier_img(img_out)
    multi_out = self.classifier_multi(multi_out)
    return text_out, img_out, multi_out
```

```
    def Multihead_self_attention(self, text_out, img_out):

        text_k1 = self.linear_text_k1(text_out)
        text_v1 = self.linear_text_v1(text_out)
        img_k2 = self.linear_img_k2(img_out)
        img_v2 = self.linear_img_v2(img_out)

        key = torch.stack((text_k1, img_k2), dim=1)
        value = torch.stack((text_v1, img_v2), dim=1)
        query = torch.stack((text_out, img_out), dim=1)

        attn_output, attn_output_weights = self.multihead_attn(query, key, value)
        return attn_output

    def Transformer_Encoder(self, text_out, img_out):
        multimodal_sequence = torch.stack((text_out, img_out), dim=1)
        return self.transformer_encoder(multimodal_sequence)
```

8. 模型训练

在函数中，首先创建了一个 `MultiModal` 类型的模型，并将其移动到指定的设备上。然后定义了优化器，使用 Adam 优化算法来更新模型的参数，学习率为 `args.lr`。接下来定义了损失函数 `loss_func`，使用交叉熵损失函数。然后开始进行循环训练，对每个 epoch 进行迭代。在每个 epoch 中，将模型设置为训练模式，即 `model.train()`。初始化训练损失、训

练正确率和总样本数为 0。通过循环遍历训练数据集中的每个 batch，从中获取文本特征、图像特征和标签，并将它们移动到指定的设备上。使用 `optimizer.zero_grad()` 将模型参数的梯度置零，以免累积梯度影响后续的反向传播。然后调用模型进行前向传播，得到文本输出、图像输出和多模态输出。根据是否有文本特征或图像特征选择相应的输出作为最终的输出。计算损失，并进行反向传播和参数更新。累加训练损失、训练正确数和总样本数。当所有 batch 都被处理完毕后，计算平均训练损失和训练正确率。输出当前 epoch 的训练损失和训练正确率。调用 `evaluate` 函数对当前模型在验证集上进行评估。最后，完成所有 epoch 的训练过程。

```python
def train(args, train_dataloader, dev_dataloader):
    model = MultiModal(args).to(device=args.device)
    optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
    loss_func = nn.CrossEntropyLoss()

    for epoch in range(1, args.epochs + 1):
        model.train()
        train_loss = 0.0
        train_correct = 0
        total_samples = 0

        for step, batch in enumerate(train_dataloader):
            ids, text, image, labels = batch
            text = text.to(device=args.device)
            image = image.to(device=args.device)
            labels = labels.to(device=args.device)
            optimizer.zero_grad()
            # Forward pass
            text_out, img_out, multi_out = model(text=text, image=image)
            # Choose the appropriate output based on the available data
            output = text_out if text is not None else img_out if image is not None else multi_out
            # Calculate loss
            loss = loss_func(output, labels)
            loss.backward()
            optimizer.step()

            train_loss += loss.item() * labels.size(0)
            train_correct += torch.sum(torch.argmax(output, dim=1) == labels).item()
            total_samples += labels.size(0)

        train_loss /= total_samples
        train_accuracy = train_correct / total_samples

        print(f"Epoch {epoch}/{args.epochs}:")
        print(f"  Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.4f}")

        evaluate(args, model, dev_dataloader, epoch)
```

9. 模型评估

这段代码定义了一个评估函数 `evaluate`，接受四个参数：`args`、`model`、`dev_dataloader` 和可选的 `epoch`。其中 `args` 是模型的相关配置，`model` 是要评估的模型，`dev_dataloader` 是验证数据集的数据加载器，`epoch` 表示当前是第几个 epoch。

在函数中，首先将模型设置为评估模式，即 `model.eval()`。然后初始化验证损失、验证正确率和总样本数为 0，并定义了损失函数 `loss_func`，同样使用交叉熵损失函数。

使用 `torch.no_grad()` 上下文管理器，表示在评估阶段不进行梯度计算，以节省内存和计算资源。然后通过循环遍历验证数据集中的每个 batch，从中获取文本特征、图像特征和标签，并将它们移动到指定的设备上。接着，调用 `model` 进行前向传播，得到模型的预测结果。计算损失函数，并计算出验证损失、验证正确率和总样本数。当所有 batch 都被处理完毕

后，计算平均验证损失和验证正确率。输出当前 epoch 的验证损失和验证正确率。最后，返回验证损失和验证正确率作为函数的输出。

```python
def evaluate(args, model, dev_dataloader, epoch=None):
    model.eval()
    dev_loss = 0.0
    dev_correct = 0
    total_samples = 0
    loss_func = nn.CrossEntropyLoss()

    with torch.no_grad():
        for batch in dev_dataloader:
            ids, text, image, labels = batch
            text = text.to(device=args.device)
            image = image.to(device=args.device)
            labels = labels.to(device=args.device)

            outputs = model(text=text, image=image)
            loss = loss_func(outputs, labels)

            dev_loss += loss.item() * labels.size(0)
            dev_correct += torch.sum(torch.argmax(outputs, dim=1) == labels).item()
            total_samples += labels.size(0)

    dev_loss /= total_samples
    dev_accuracy = dev_correct / total_samples

    if epoch:
        print(f"  Dev Loss: {dev_loss:.4f}, Dev Accuracy: {dev_accuracy:.4f} (Epoch {epoch})")
    else:
        print(f"  Dev Loss: {dev_loss:.4f}, Dev Accuracy: {dev_accuracy:.4f}")
```

10. 训练循环

使用 `model.train()` 将模型设置为训练模式。遍历训练数据集（`train_dataloader`）进行多轮训练。计算并打印每轮训练的损失和准确率。并调用 `evaluate` 函数对模型在验证集上进行评估，并打印验证损失和准确率。最后调用 `get_test` 函数对模型在不同场景下进行测试，包括文本输入、图像输入、以及文本和图像同时输入的情况。

```python
def train_and_test(args, train_dataloader, dev_dataloader, test_dataloader):
    model = MultiModal(args).to(device=args.device)
    optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
    loss_func = nn.CrossEntropyLoss()
    for epoch in range(1, args.epochs + 1):
        model.train()
        train_loss = 0.0
        train_correct = 0
        total_samples = 0
        for step, batch in enumerate(train_dataloader):
            ids, text, image, labels = batch
            text = text.to(device=args.device)
            image = image.to(device=args.device)
            labels = labels.to(device=args.device)
            optimizer.zero_grad()
            output = model(text=text, image=image)
            loss = loss_func(output, labels)
            loss.backward()
            optimizer.step()

            train_loss += loss.item() * labels.size(0)
            train_correct += torch.sum(torch.argmax(output, dim=1) == labels).item()
            total_samples += labels.size(0)

        train_loss /= total_samples
        train_accuracy = train_correct / total_samples

        print(f"Epoch {epoch}/{args.epochs}:")
        print(f"  Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.4f}")

        evaluate(args, model, dev_dataloader, epoch)
```

11. 测试场景

使用 `model.eval()` 将模型设置为评估模式，确保在测试阶段不进行梯度更新。遍历测试数据集（`test_dataloader`），获取每个批次的文本和图像数据。使用模型（`model`）对文本和图像进行前向传播，得到输出。通过取预测输出的最大值确定模型对每个样本的预测标签。将每个样本的预测结果（`item_id` 和 `tag`）存储在 `predictions` 列表中。调用 `save_data` 函数，将测试结果保存到指定的文件（`args.test_output_file`）中。

```python
def get_test(args, model, test_dataloader, scenario=""):
    model.eval()
    predictions = []

    with torch.no_grad():
        for batch in test_dataloader:
            ids, text, image, _ = batch
            text = text.to(device=args.device)
            image = image.to(device=args.device)

            outputs = model(text=text, image=image)
            predicted_labels = torch.argmax(outputs, dim=1)

            for i in range(len(ids)):
                item_id = ids[i]
                tag = test_dataloader.dataset.label_dict_str[int(predicted_labels[i])]
                prediction = {
                    'guid': item_id,
                    'tag': tag,
                }
                predictions.append(prediction)

    save_data(args.test_output_file, predictions)

    accuracy = calculate_accuracy(predictions, test_dataloader.dataset.data)
    print(f"  {scenario.capitalize()} Test Accuracy: {accuracy:.4f}")
```

12. 准确率计算

```python
def calculate_accuracy(predictions, data):
    correct_predictions = 0
    total_samples = len(predictions)

    for pred in predictions:
        try:
            guid_int = int(pred['guid'])
            if guid_int < len(data) and pred['tag'] == data[guid_int]['label']:
                correct_predictions += 1
        except (ValueError, IndexError):
            continue

    accuracy = correct_predictions / total_samples if total_samples > 0 else 0.0
    return accuracy
```

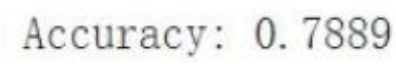
13. 配置与运行

定义了一个配置参数类 `Config`，包含了训练、测试所需的各种参数，如训练文件路径、测试文件路径、预训练模型、学习率等。调用 `load_data` 函数加载训练集和验证集，并创建相应的数据加载器（`DataLoader`）。进行模型的训练和测试。调用 `train_and_test` 函数，传递训练数据加载器、验证数据加载器和测试数据加载器进行模型的训练和测试。

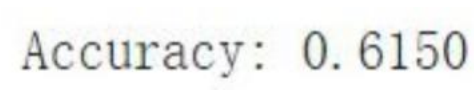
```python
class Config:
    def __init__(self):
        self.do_train = True
        self.do_test = True
        self.test_output_file = './test_with_label.txt'
        self.train_file = './dataset/train.json'
        self.dev_file = './dataset/dev.json'
        self.test_file = './dataset/test.json'
        self.pretrained_model = 'roberta-base'
        self.lr = 1e-5
        self.dropout = 0.0
        self.epochs = 1
        self.batch_size = 4
        self.img_size = 384
        self.text_size = 64
        self.mode = 'train'

args = Config()

args.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f'Device: {args.device}')

train_set, dev_set = load_data(args)
train_dataloader = DataLoader(train_set, shuffle=True, batch_size=args.batch_size)
dev_dataloader = DataLoader(dev_set, shuffle=False, batch_size=args.batch_size)

if args.do_train:
    print('Model training...')
    test_set, _ = load_data(args)
    test_dataloader = DataLoader(test_set, shuffle=False, batch_size=args.batch_size)
    train_and_test(args, train_dataloader, dev_dataloader, test_dataloader)
```

4. 实验结果

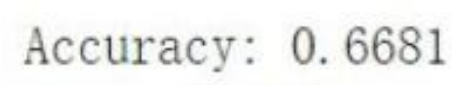
多模态融合模型结果：
正确率：78.89%

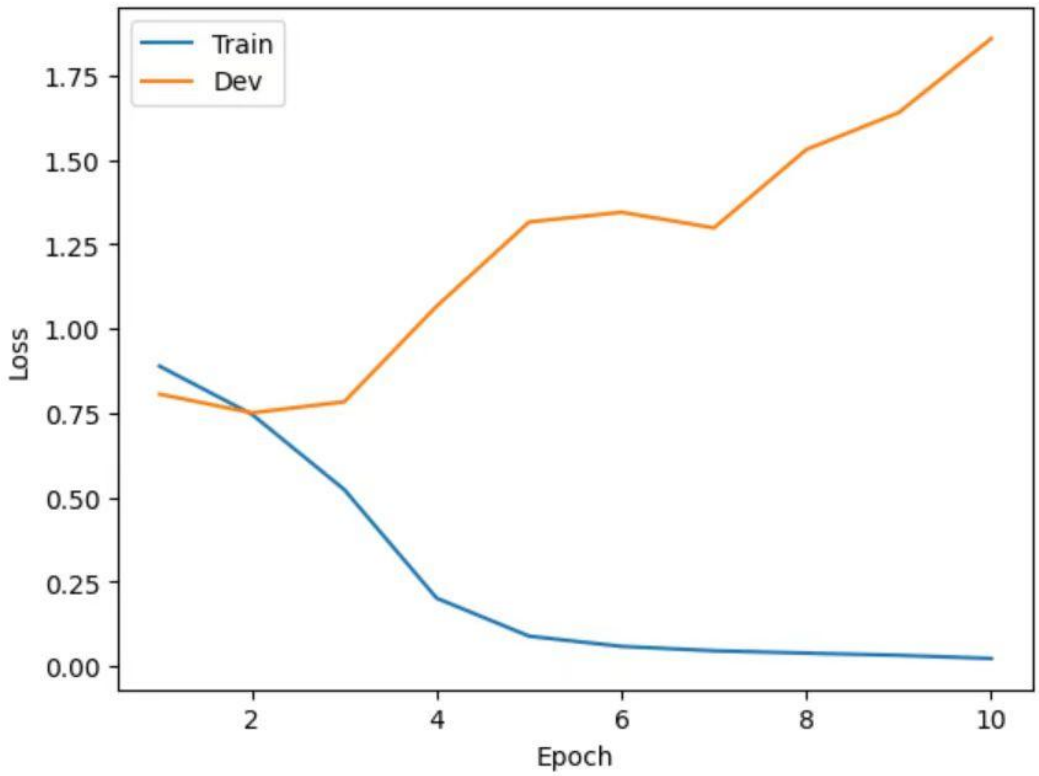


消融实验：
仅图像：61.50%



仅文字：66.81%





5. 总结与思考

5.1 遇到的问题

1. Huggingface 资源连接问题。本次实验中用到的 RoBERTa 模型需要连接到 huggingface，本地 CPU 运行代码会出现访问超时或者其他错误，我找到了两个解决方法：在代码的最上方配置环境到主机的 VPN（我的 VPN 端口是 33210，这个需要根据具体情况调整）

```
import os
os.environ["http_proxy"] = "http://127.0.0.1:33210"
os.environ["https_proxy"] = "http://127.0.0.1:33210"
```

或者也可以使用 colab 运行代码，速度也会更快。CPU 运行一轮 epoch 大约要两个小时，colab 可以让时间减半。

2. 运行中经常出现 turple 长度不匹配的问题，我直接通过 debug 一行一行走来不断调整。

5.2 对本次实验的总结

1. 这个模型的设计是为了处理多模态数据，其中包括文本和图像。它使用了两个单模态模型，一个是基于 RoBERTa 的文本模型（TextOnly），另一个是基于 ConvNeXt 的图像模型（ImgOnly）。这两个模型分别用于处理文本和图像输入，并通过后续的多模态结构进行信

息融合和分类。这两个模型都是我从网络上查找的效果比较好的模型，所以选择了他们。而两个模型也只是简单的进行了拼接。

2. 模型的亮点包括：

① RoBERTa 文本编码：使用了 RoBERTa 预训练模型，这是一种基于 Transformer 的强大的自然语言处理模型。RoBERTa 能够有效地捕捉文本中的语义和上下文信息，为模型提供了丰富的文本表示。

② 多模态融合：通过多头自注意力机制和 Transformer 编码器，模型能够融合来自文本和图像模态的信息。这有助于模型更好地理解文本和图像之间的关系，提高了对多模态输入的处理能力。

③ 灵活的分类器：为每个模态和融合后的信息设计了多层的全连接层分类器，提高了模型学习复杂特征和模式的能力。分类器的层次结构可以适应不同任务和复杂度的数据分布。

总体而言，该模型结合了 RoBERTa 等强大的预训练模型和多模态信息融合的机制，具备处理多模态数据任务的强大能力。