实验五：多模态情感分析

实验任务：给定配对的文本和图像，预测对应的情感标签，有三种分类：positive, neutral, negative。

实验要求：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，并预测测试集（test_without_label.txt）上的情感标签。

github仓库：https://github.com/Nemophilist8/AI_lab5

1 实验概述

本次实验旨在通过使用ResNet50提取图像特征和Bert提取文本特征，探索不同的多模态融合方法。具体而言，我们采用了四种不同的模型结构，包括简单Concat、GatedFusion、单流融合模型和双流融合模型，以比较它们在图像和文本信息融合上的性能。

首先，我们使用ResNet50对图像进行特征提取，Bert对文本进行特征提取。这两个模型都是在大规模数据上预训练的，并在多领域任务中表现出色。通过使用这两个强大的预训练模型，我们能够捕捉到图像和文本领域的丰富信息。

1.1 简单Concat

简单Concat模型直接将图像和文本特征进行拼接，然后输入到一个后续的处理层中。这种方法的优点在于简单明了，不需要复杂的操作，而且参数较少，训练速度相对较快。然而，这种方法忽略了模态之间的交互，可能无法充分挖掘图像和文本之间的语义关系。

1.2 GatedFusion

GatedFusion模型采用门控机制，允许一种模态通过Sigmoid非线性对另一种模态进行"关注"或"控制"。通过引入门控机制，模型可以有选择性地关注图像或文本信息，提高融合效果。这种方法类似于注意力机制，但相比之下，门控机制允许更灵活的模态关注。

门控机制的公式为$o(x_n) = W(\sigma(Ux_{t_n}) \odot Vx_{v_n})$或$o(x_n) = W(Ux_{t_n} \odot \sigma(Vx_{v_n}))$。其中，$U$和$V$是权重矩阵，$\sigma$是 Sigmoid 激活函数，$x_{t_n}$ 和 x_{v_n} 分别表示两种模态的表示，W是另一个权重矩阵。这种方法在概念上类似于(Arevalo等人，2017)中引入的多模态门控单元。

1.3 单流融合模型

单流融合模型直接将两个模态的内容加以拼接，之后输入到transformer中。相当于是将单模态内的交互和多模态之间的交互合到一起来做。比如，vision的q来自于图像，其k和v既来自于图像又来自于文本；文本这边也是如此。其最大的好处是其参数的高效性。并不需要将跨模态的交互和单模态的交互放到一起来做，因此是十分节约参数的。

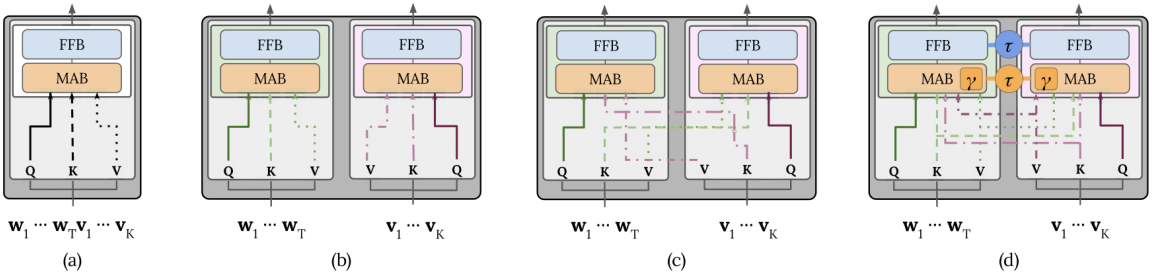


Figure 3: Visualization of the (a) single-stream, (b) dual-stream intra-modal, and (c) dual-stream inter-modal Transformer layers. (d) shows our gated bimodal layer. The inter-modal layer attends across modalities, while the intra-model layer attends within each modality. Ours can attend to either or both.

1.4 双流融合模型

由于单流架构中的单模态特征都是刚刚提取到的，没有学习很好的单模态特征，因此具有一定局限性。为解决这个问题，提出了双流架构的解决方案。双流网络首先在单模态的场景下将模态内的交互做好，然后在多模态场景下，做好多模态的交互。

与单流架构中的自注意操作不同，双流架构采用交叉注意机制来建模V-L交互，其中查询向量来自一种模式，而键和值向量来自另一种模式。交叉注意力层通常包含两个单向交叉注意力子层：一个从语言到视觉，另一个从视觉到语言。它们负责在两种模式之间交换信息和调整语义。

2 实验代码

2.1 数据组织

CustomDataset类

`CustomDataset` 类用于自定义数据集的加载和预处理，继承自 PyTorch 的 `Dataset` 类。

`__init__` 方法用于初始化数据集，接受参数 `datafile`（数据文件夹路径）、`transform`（图像转换函数）、`max_length`（文本的最大长度，默认为 `128`）、和 `hashtags`（是否保留以#开头的话题），创建 `self.imgs`、`self.descriptions`、`self.labels` 分别用于存储图片数据列表、文本数据列表和标签列表，还创建了 `self.tokenizer`，用 BERT 分词器对文本进行编码。

`__len__` 方法用于返回数据集的大小。

```
class CustomDataset(Dataset):
    def __init__(self, datafile, transform=None, max_length=128, hashtags=True):
        # 构造函数
        self.datafile = datafile
        self.transform = transform
        self.hashtags = hashtags
        self.imgs = []
        self.descriptions = []
        self.labels = []
        self.max_length = max_length

        self.readdata()

        self.tokenizer = BertTokenizer.from_pretrained('./bert-base-
multilingual-cased')
    def __len__(self):
        # 返回数据集大小
        return len(self.labels)
```

`readdata()` 方法用于读取数据。首先打开训练数据文件（`train.txt`），读取每行的数据，解析出标签、图像和文本路径。之后加载对应的文本与图片，并对文本进行去除回复信息、去除@某人、去除url、转换成小写等预处理步骤。

```
def readdata(self):
    # 读取数据
    with open(os.path.join(self.datafile, 'train.txt'), 'r', encoding='utf-8')
as fp:
        lines = fp.readlines()
        lines = lines[1:]
```

```
        lines = [i[:-1] for i in lines]
        for line in lines:
            t = line.split(',')
            guid = t[0]
            label = t[1]
            if label == 'positive':
                label = 0
            elif label == 'neutral':
                label = 1
            else:
                label = 2

            img_path = os.path.join(self.datafile, 'data', guid + '.jpg')
            txt_path = os.path.join(self.datafile, 'data', guid + '.txt')

            description = ''
            try:
                with open(txt_path, 'r', encoding='gbk') as fp1:
                    description = fp1.read()[:-1]
            except:
                try:
                    with open(txt_path, 'r', encoding='utf-8') as fp1:
                        description = fp1.read()[:-1]
                except:
                    print(txt_path)
                    continue

            description = self.remove_rt_mentions(description)      # 去除回复信息
            description = self.remove_at(description)               # 去除@某人的信
息
            description = self.remove_urls(description)             # 去除url
            description = description.lower()                       # 转换成小写

            if not self.hashtags:
                description = self.remove_hashtags(description)
            else:
                description = description.replace('#', '')

            img = Image.open(img_path).convert('RGB')
            self.imgs.append(img)
            self.labels.append(label)
            self.descriptions.append(description)

    def remove_hashtags(self, sentence):
        # 去除文本中的hashtags
        cleaned_sentence = re.sub(r'\#\w+', '', sentence)
        return cleaned_sentence.strip()

    def remove_rt_mentions(self, sentence):
        # 去除文本中的RT mentions
        cleaned_sentence = re.sub(r'RT\s?@\w+:\s?', '', sentence)
        return cleaned_sentence.strip()

    def remove_at(self, sentence):
        # 去除文本中的@
```

```
    cleaned_sentence = re.sub(r'@\w+', '', sentence)
    return cleaned_sentence.strip()

def remove_urls(self, sentence):
    # 去除文本中的URLs
    cleaned_sentence = re.sub(r'http[s]?://t\.co\w+', '', sentence)
    return cleaned_sentence.strip()
```

`__getitem__` 方法用于获取指定索引处的样本，返回图像、文本编码和标签。首先，从数据集的存储列表中提取图像、文本描述和标签。按照图像转换函数（`self.transform`）的要求，对图像进行归一化与缩放。接下来，使用 BERT 分词器（`self.tokenizer`）对文本描述进行编码，确保其符合模型的输入格式，并进行了填充和截断处理，以满足模型对文本输入的要求。如果文本编码的长度超过了128，将其截断为前128个标记。

```
def __getitem__(self, idx):
    # 获取数据集中的样本
    img = self.imgs[idx]
    description = self.descriptions[idx]
    label = self.labels[idx]

    if self.transform:
        img = self.transform(img)

    input_ids = self.tokenizer.encode(description, return_tensors='pt',
padding='max_length', max_length=self.max_length).squeeze(0)

    if len(input_ids) > 128:
        input_ids = input_ids[:128]

    return img, input_ids, label
```

定义数据集加载器

- 创建 `CustomDataset` 实例，传递数据文件夹路径和图像转换函数。
- 使用 `SubsetRandomSampler` 进行数据集拆分。`indices` 存储数据集索引，通过计算拆分点 `split` 将数据集划分为训练集和验证集。
- 定义 `DataLoader` 对象 `train_loader` 和 `val_loader`，分别用于训练和验证。

```
dataset = CustomDataset('数据', transform)

validation_split = 0.2
dataset_size = len(dataset)
indices = list(range(dataset_size))
split = int(np.floor(validation_split * dataset_size))
train_indices, val_indices = indices[split:], indices[:split]

# 定义数据加载器
batch_size = 16

train_sampler = SubsetRandomSampler(train_indices)
val_sampler = SubsetRandomSampler(val_indices)

train_loader = DataLoader(dataset, batch_size=batch_size, sampler=train_sampler)
```

```
val_loader = DataLoader(dataset, batch_size=batch_size, sampler=val_sampler)
```

2.2 图片表征提取

图片表征提取使通过ResNet50模型实现的，因此这一步要对预训练的 ResNet50 模型进行微调以进行图像分类任务，在之后需要提取图片表征时，去掉模型最后的全连接层，即可获得2048维的特征向量。

训练ResNet50网络

初始化 ResNet50 模型：

```
model = models.resnet50(pretrained=True)
model.fc = nn.Linear(model.fc.in_features, 3)  # 有3个分类

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print('设备：', device)
model = model.to(device)
```

- 使用 PyTorch 的 `models.resnet50` 函数初始化 ResNet50 模型，同时加载 ImageNet 上预训练的权重（`pretrained=True`）。
- 修改模型的最后一层全连接层（`model.fc`），将其输出特征数调整为 3，以适应具体的分类任务。
- 检测是否有可用的 GPU，并将模型移动到对应的设备上（GPU 或 CPU）。

损失函数和优化器设置：

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0005)
```

- 使用交叉熵损失函数（`nn.CrossEntropyLoss`），适用于多分类问题。
- 使用 Adam 优化器（`optim.Adam`）对模型参数进行优化，设置学习率为 0.0005。

训练循环：

```
num_epochs = 5
best_val_loss = float('inf')  # 初始化为一个较大的数
```

- 设置训练循环的总轮数为 5，并初始化一个较大的值作为最佳验证损失。

```
for epoch in range(num_epochs):
    model.train()
    with tqdm(train_loader, desc=f'Epoch {epoch + 1}/{num_epochs}',
unit='batch') as train_pbar:
        for inputs, labels in train_pbar:
            inputs, labels = inputs.to(device), labels.to(device)  # 将数据移动到
GPU
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            train_pbar.set_postfix({'Loss': loss.item()})
```

- 在每个训练轮次中，将模型设置为训练模式（`model.train()`）。
- 使用 tqdm 进度条迭代训练数据加载器（`train_loader`），对每个批次进行训练。
- 将输入数据和标签移动到指定的设备上，计算损失、反向传播并更新模型参数。

```
# 验证循环
model.eval()
val_loss = 0.0
correct = 0
total = 0

with torch.no_grad():
    for inputs, labels in val_loader:
        inputs, labels = inputs.to(device), labels.to(device)  # 将数据移动到 GPU
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        val_loss += loss.item()

        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()
```

- 在每个训练轮次结束后，将模型设置为评估模式（`model.eval()`）。
- 对验证数据加载器（`val_loader`）进行迭代，计算验证损失和准确率。
- 使用 `torch.no_grad()` 上下文管理器，禁用梯度计算，以加速验证过程。

```
avg_val_loss = val_loss / len(val_loader)
accuracy = 100 * correct / total

print(f'Epoch {epoch + 1}/{num_epochs}, Loss: {avg_val_loss}, Accuracy:
{accuracy}%')
```

- 计算并打印平均验证损失和准确率。

```
# 如果具有最佳的验证损失，则保存模型
if avg_val_loss < best_val_loss:
    best_val_loss = avg_val_loss
    best_model_path = 'resnet50.pth'
    torch.save(model.state_dict(), best_model_path)
    print(f'Saved the best model with validation loss: {best_val_loss} to
{best_model_path}')
```

- 如果当前验证损失低于之前记录的最佳验证损失，则保存当前模型权重。这有助于提前停止训练，以避免过拟合。

提取图片特征

加载预训练权重：

```
img_model = resnet50(pretrained=False)
img_model.fc = nn.Linear(img_model.fc.in_features, 3)

model_weights_path = 'resnet50.pth'
img_model.load_state_dict(torch.load(model_weights_path))
```

从指定路径加载预训练的模型权重文件（`resnet50.pth`）到初始化的 ResNet50 模型。

模型微调设置：

```
img_model.fc = nn.Identity()
img_model = img_model.to(device)
```

将最后一层全连接层（`img_model.fc`）替换为 `nn.Identity()`，相当于这层不做任何操作，将模型移动到GPU上。

提取图片特征：

```
img_model.eval()

for epoch in range(num_epochs):
    with tqdm(train_loader, desc=f'Epoch {epoch + 1}/{num_epochs}',
unit='batch') as train_pbar:
        for imgs, input_ids, labels in train_pbar:
            imgs, input_ids, labels = imgs.to(device), input_ids.to(device),
labels.to(device)
            outputs_img = img_model(imgs)
```

2.3 文本表征提取

文本表征提取使通过Bert模型实现的，因此这一步要对预训练的Bert模型构成的BertForSequenceClassification进行微调以进行文本分类任务，在之后需要提取文本表征时，用BertForSequenceClassification.Bert，即可获得768维的特征向量。

训练BertForSequenceClassification网络

由于BertForSequenceClassification对输入形式的要求，所以要对 `CustomDataset` 类的 `__getitem__` 方法进行一些修改，使之返回编码后的文本（`input_ids`）、注意力掩码（`attention_mask`）和标签。

```
def __getitem__(self, idx):
    img = self.imgs[idx]
    description = self.descriptions[idx]
    label = self.labels[idx]

    # 使用 BERT 分词器对描述进行编码
    inputs = self.tokenizer(description, return_tensors="pt", truncation=True,
padding='max_length', max_length=self.max_length)

    with torch.no_grad():
        outputs = self.bert_model(**inputs)

    # 提取句子表示,使用 [CLS] 令牌的隐藏状态作为整个句子的表示
```

```
    sentence_representation = outputs.last_hidden_state[:, 0, :]

    return inputs['input_ids'].squeeze(), inputs['attention_mask'].squeeze(),
label
```

BERT Tokenizer 和 BERT 模型初始化：

```
tokenizer = BertTokenizer.from_pretrained('bert-base-multilingual-cased')
bert_model = BertForSequenceClassification.from_pretrained('bert-base-
multilingual-cased', num_labels=3)
```

- 通过 `BertTokenizer.from_pretrained` 初始化一个 BERT 分词器，用于将文本数据编码为 BERT 模型可以接受的输入格式。
- 通过 `BertForSequenceClassification.from_pretrained` 初始化一个预训练的 BERT 模型，用于进行序列分类任务，设置类别数为3。

自定义 BERT 分类器模型：

```
model = CustomBERTClassifier(bert_model)
```

```
class CustomBERTClassifier(nn.Module):
    def __init__(self, bert_model):
        super(CustomBERTClassifier, self).__init__()
        self.bert = bert_model

    def forward(self, inputs):
        outputs = self.bert(**inputs)
        return outputs.logits
```

- 创建一个自定义的 BERT 分类器模型 `CustomBERTClassifier`，该模型使用预训练的 BERT 模型作为其组成部分。

优化器和损失函数设置：

```
criterion = nn.CrossEntropyLoss()
optimizer = AdamW(model.parameters(), lr=1e-5)
```

- 使用交叉熵损失函数（`nn.CrossEntropyLoss`）适用于多分类问题。
- 使用 AdamW 优化器（`AdamW`）对模型参数进行优化，设置学习率为 1e-5。

数据获取与模型的前向传播：

```
for batch in train_pbar:
    input_ids, attention_mask, labels = batch
    input_ids = input_ids.to(device)
    attention_mask = attention_mask.to(device)
    labels = labels.to(device)

    optimizer.zero_grad()

    logits = model({'input_ids': input_ids, 'attention_mask': attention_mask})
```

通过`logits = model({'input_ids': input_ids, 'attention_mask': attention_mask})`的方式，利用文本编码和注意力掩码进行前向传播。

整体的训练循环和验证循环结构与上文类似，故不多赘述。

训练函数调用：

```
save_path = 'bert-base-multilingual-cased.pth'
train_model(model, train_loader, val_loader, criterion, optimizer, num_epochs=5,
save_path=save_path)
```

- 指定模型的保存路径，调用`train_model`函数进行模型训练。

提取文本表征

```
bert_model = BertForSequenceClassification.from_pretrained('bert-base-
multilingual-cased', num_labels=3)
loaded_model = CustomBERTClassifier(bert_model)
loaded_model.load_state_dict(torch.load('bert-base-multilingual-cased.pth'))
text_model = loaded_model.bert.bert

text_model = text_model.to(device)
text_model.eval()

for imgs, input_ids, labels in train_pbar:
    imgs, input_ids, labels = imgs.to(device), input_ids.to(device),
labels.to(device)
    outputs_text = text_model(input_ids)

    last_hidden_states = outputs_text.last_hidden_state

    # 提取特征向量（CLS对应的隐藏状态）
    outputs_text = last_hidden_states[:, 0, :]
```

预训练权重由`torch.load`从文件加载，并通过`loaded_model.bert.bert`提取了BERT模型的文本表示部分，即去除了分类层的部分。该文本模型被移动到指定设备上，同时在评估模式下进行操作。最后，在训练过程中，通过迭代数据加载器（`train_pbar`）传递输入文本序列给BERT模型的文本部分，获得文本表示，提取了 BERT 模型输出中的最后一层隐藏状态，进而提取出整个输入文本序列的特征向量表示。

在这部分代码的书写过程中，BertForSequenceClassification模型结构如下，一开始尝试了把它的最后两层删除，出现了如下报错，这是因为CustomBERTClassifier是经过包装的BertForSequenceClassification， BertConfig不再适用，用`text_model = loaded_model.bert.bert`就可以轻松的取出想要的bert模型。

```
CustomBERTClassifier(
  (bert): BertForSequenceClassification(
    (bert): BertModel(...)
    (dropout): Dropout(p=0.1, inplace=False)
    (classifier): Linear(in_features=768, out_features=3, bias=True)
  )
)
```

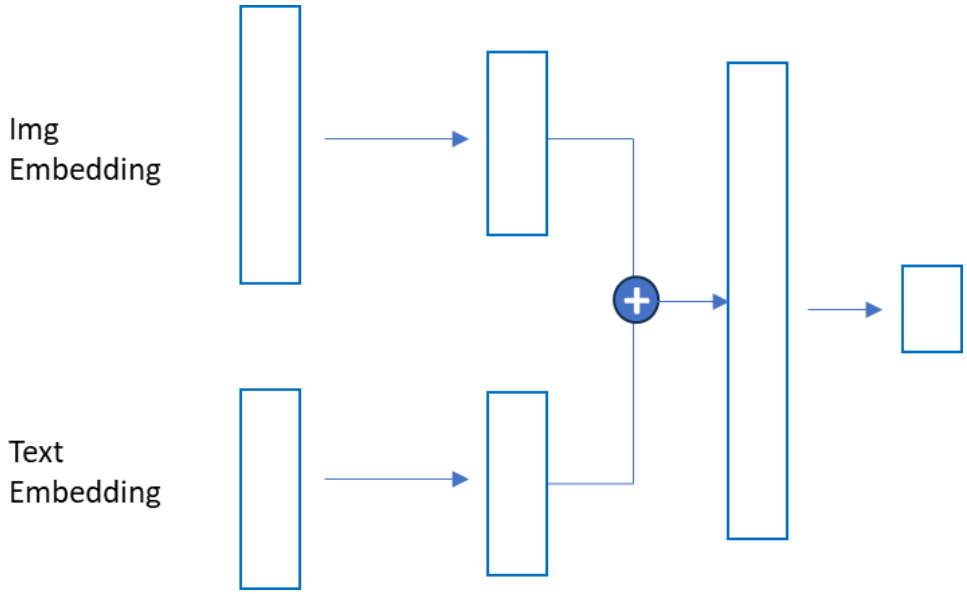
```
bert_model = BertForSequenceClassification.from_pretrained('bert-base-
multilingual-cased', num_labels=3)
text_model = CustomBERTClassifier(bert_model)

text_model.load_state_dict(torch.load('bert-base-multilingual-cased.pth'))

# 获取模型的配置
config = BertConfig.from_pretrained('bert-base-multilingual-cased')

# 删除最后两层
config.num_hidden_layers -= 2

# 根据新配置构建新模型
text_model = text_model.bert(config)
```

```
  File "D:\anaconda3\lib\site-packages\transformers\modeling_utils.py", line 3941, in warn_if_padding_and_no_a
    if self.config.pad_token_id in input_ids[:, [-1, 0]]:
TypeError: 'BertConfig' object is not subscriptable
```

2.4 Fusion Model

简单Concat

ConcatFusion首先将输入张量 `x` 和 `y` 分别通过线性变换层， 连接在一起 (`torch.cat`) 形成融合后的特征向量， 之后通过一个简单的全连接层映射到输出维度。



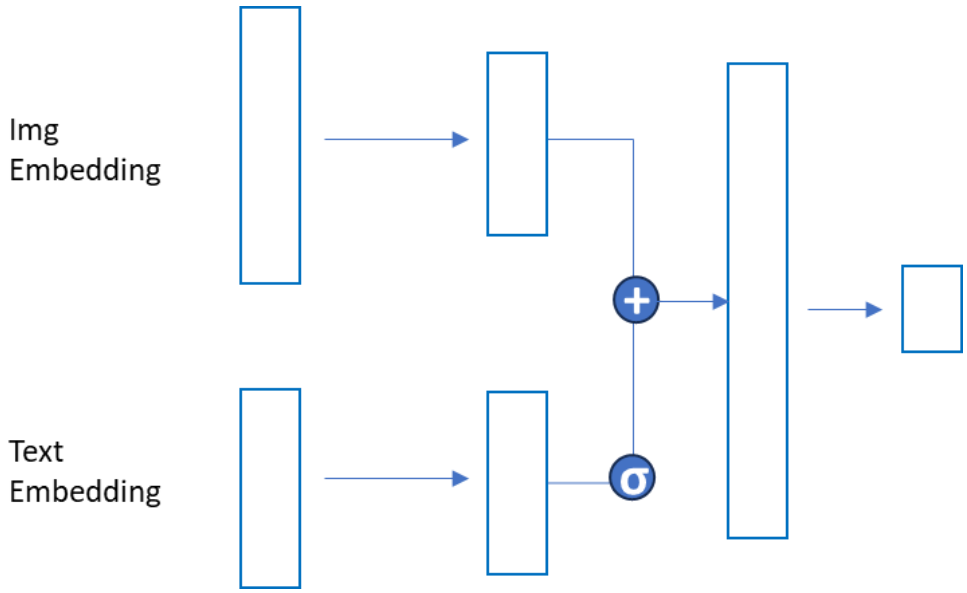

```
class ConcatFusion(nn.Module):
    def __init__(self, x_dim=2048, y_dim=768, hidden_dim=512, output_dim=3,
dropout_prob=0.3):
        """
        :param x_dim: 第一个输入张量 `x` 的特征维度
        :param y_dim: 第二个输入张量 `y` 的特征维度
        :param hidden_dim: 融合过程中的隐藏层维度
        :param output_dim: 输出的维度，通常对应任务的类别数
        :param dropout_prob: Dropout 操作的概率。
        """
        super(ConcatFusion, self).__init__()
        self.fc_x = nn.Linear(x_dim, hidden_dim)
        self.fc_y = nn.Linear(y_dim, hidden_dim)
```

```
        self.dropout1 = nn.Dropout(p=dropout_prob)
        self.fc1 = nn.Linear(hidden_dim*2, output_dim)

    def forward(self, x, y):
        # 前向传播逻辑
        x = self.fc_x(x)
        y = self.fc_y(y)
        # 连接输入张量
        output = torch.cat((x, y), dim=1)

        output = self.dropout1(output)
        output = self.fc1(output)

        return x, y, output
```

在书写代码的过程中发现，如果在模型最后多加几个fc层，准确率反而会下降3%左右，这可能是因为过度的全连接层会引入过拟合或者增加模型的复杂性。

Gated

GatedFusion实现了一个门控融合模型。首先通过两个线性映射层将输入张量映射到相同的维度。然后，通过 Sigmoid 激活函数产生门控权重。在门控权重的作用下，通过元素级别的乘法（`torch.mul`）对两个输入张量进行融合，融合后的结果通过另一个线性映射层映射到最终的输出维度。



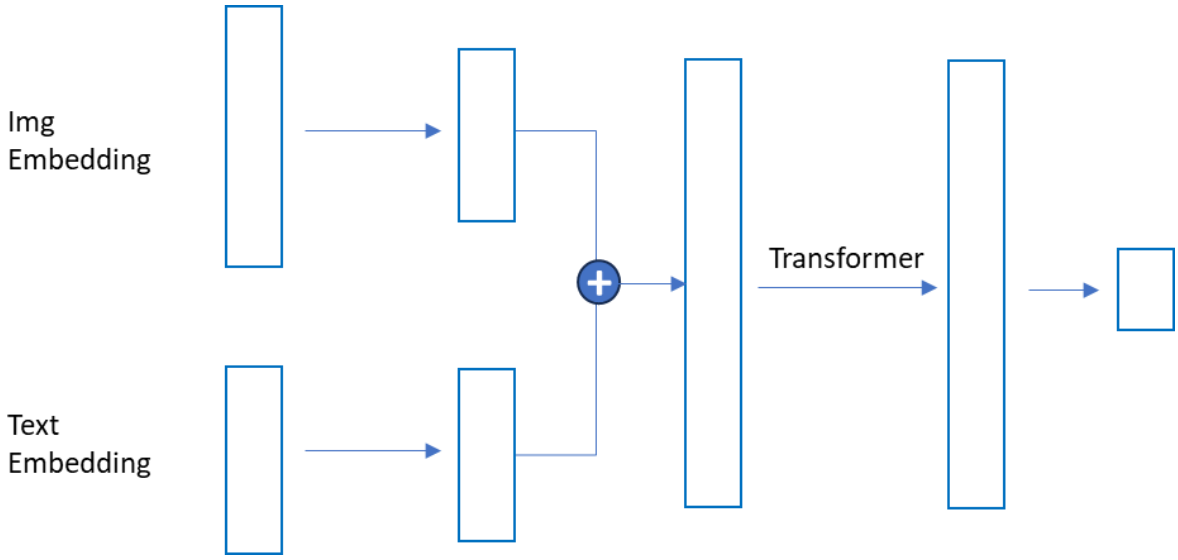

```
class GatedFusion(nn.Module):
    def __init__(self, input_dim_x=512, input_dim_y=512, dim=512, output_dim=3,
x_gate=True):
        super(GatedFusion, self).__init__()
        # 添加线性映射以使输入具有相同的维度
        self.fc_x = nn.Linear(input_dim_x, dim)
        self.fc_y = nn.Linear(input_dim_y, dim)
        self.fc_out = nn.Linear(dim, output_dim)
        self.x_gate = x_gate
        self.sigmoid = nn.Sigmoid()

    def forward(self, x, y):
        # 添加线性映射以使输入具有相同的维度
        out_x = self.fc_x(x)
        out_y = self.fc_y(y)
```

```
        if self.x_gate:
            gate = self.sigmoid(out_x)
            output = self.fc_out(torch.mul(gate, out_y))
        else:
            gate = self.sigmoid(out_y)
            output = self.fc_out(torch.mul(out_x, gate))

        return out_x, out_y, output
```

单流多模态融合模型

MultiModalTransformer实现了多模态Transformer模型，接收图像特征和文本特征，通过Transformer层实现模态融合，允许模型学习两个模态之间的复杂关系。

首先将它们通过全连接层投影到相同的隐藏空间。然后，将两个模态的特征在隐藏空间上进行拼接。接着，通过Transformer编码层对拼接后的特征进行变换。最后，通过全连接层和Dropout进行分类，输出最终的预测结果。




```
class MultiModalTransformer(nn.Module):
    def __init__(self, input_size_img=2048, input_size_text=768, output_size=3,
hidden_size=256, nhead=16):
        super(MultiModalTransformer, self).__init__()

        self.fc_img = nn.Linear(input_size_img, hidden_size)
        self.fc_text = nn.Linear(input_size_text, hidden_size)

        # Transformer layer
        self.transformer_layer = nn.TransformerEncoderLayer(
            d_model= hidden_size*2,
            nhead=nhead,
            # dropout=0.1
        )

        self.dropout = nn.Dropout(p=0.3)
        self.fc = nn.Linear(hidden_size*2, output_size)


    def forward(self, img_features, text_features):
        img_proj = self.fc_img(img_features)
```

```
        text_proj = self.fc_text(text_features)

        combined_features = torch.cat((img_proj, text_proj), dim=1)

        combined_features = self.transformer_layer(combined_features)

        combined_features = torch.squeeze(combined_features, dim=1)

        output = self.dropout(combined_features)
        output = self.fc(output)

        return output
```

在这部分的代码书写过程中，一开始错误的将nn.TransformerEncoderLayer()写成了nn.Transformer()，出现了报错：

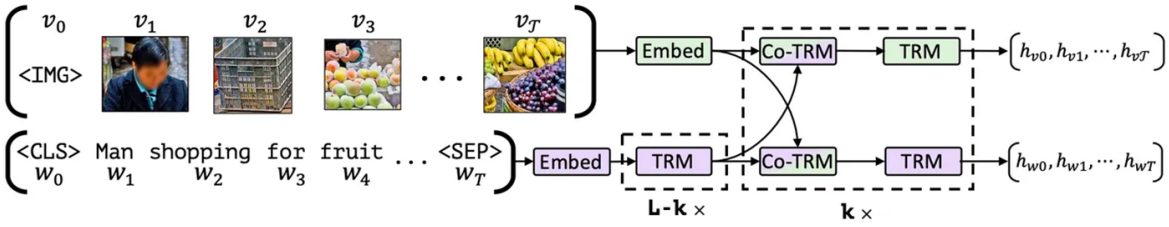
```
Traceback (most recent call last):
  File "D:/schoolworks/人工智能/lab5/transformer能行.py", line 240, in <module>
    outputs = fusion_model(outputs_img.to(device), outputs_text.to(device))
  File "D:\anaconda3\lib\site-packages\torch\nn\modules\module.py", line 1130, in _call_impl
    return forward_call(*input, **kwargs)
  File "D:/schoolworks/人工智能/lab5/transformer能行.py", line 163, in forward
    combined_features = self.transformer_layer(combined_features)
  File "D:\anaconda3\lib\site-packages\torch\nn\modules\module.py", line 1130, in _call_impl
    return forward_call(*input, **kwargs)
TypeError: forward() missing 1 required positional argument: 'tgt'
```

从表面上看是因为Transformer在前向传播的过程中，缺失了tgt参数，从深层次看是因为单独的编码层负责对输入序列进行自注意力机制和前馈神经网络操作，用于捕捉局部和全局的序列关系，提供更高层次的特征表示，而整个Transformer模型（包括多个编码层）则通过堆叠这些层，实现对输入序列的层次化表示学习，有效捕捉序列中的长距离依赖关系。

双流多模态融合模型

MultiModalCoTransformer中实现了多模态协同注意力Transformer模型。其结构包括图像特征和文本特征的投影层、文本特征的Transformer编码层、协同注意力层、两个独立的Transformer编码层和最终的全连接层。

首先通过全连接层将图像特征和文本特征投影到相同的隐藏空间。接着，文本特征通过一个Transformer编码层进行变换。然后，通过协同注意力层对两个模态的特征进行交互和协同处理，协同注意力层用于在两个模态之间建立动态的注意力关系，有助于模型更好地利用不同模态之间的信息。之后，通过两个独立的Transformer编码层分别对图像和文本的特征进行进一步的变换。最终，通过全连接层将两个模态的特征拼接在一起，并输出最终的分类结果。

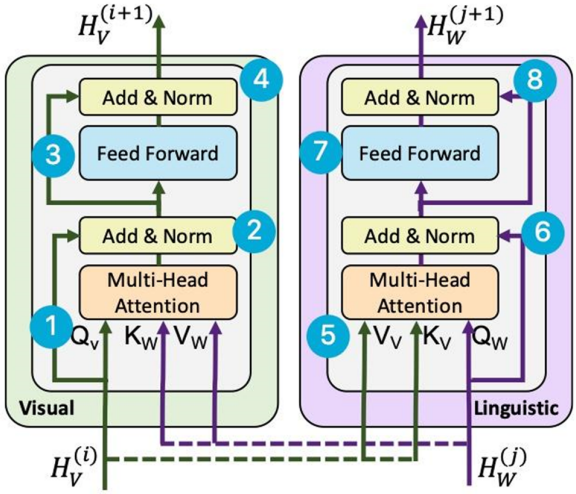



```
import torch
import torch.nn as nn
import torch.nn.functional as F

class MultiModalCoTransformer(nn.Module):
```

```
    def __init__(self, input_size_img=2048, input_size_text=768, output_size=3,
hidden_size=256, nhead=16):
        super(MultiModalCoTransformer, self).__init__()

        # 图像特征和文本特征的投影层
        self.fc_img = nn.Linear(input_size_img, hidden_size)
        self.fc_text = nn.Linear(input_size_text, hidden_size)

        # 文本特征的Transformer编码层
        self.text_encoder = nn.TransformerEncoderLayer(
            d_model=hidden_size,
            nhead=nhead,
            # dropout=0.1
        )

        # 协同注意力层
        self.co_attention =
CoattentionTransformerLayer.CoAttentionEncoderLayer(hidden_size, device)

        # 独立的Transformer编码层
        self.encoder1 = nn.TransformerEncoderLayer(
            d_model=hidden_size,
            nhead=nhead,
            # dropout=0.1
        )
        self.encoder2 = nn.TransformerEncoderLayer(
            d_model=hidden_size,
            nhead=nhead,
            # dropout=0.1
        )

        # 最终的全连接层
        self.fc = nn.Linear(hidden_size*2, output_size)

    def forward(self, img_features, text_features):
        # 图像特征和文本特征的投影
        img_features = F.relu(self.fc_img(img_features))
        text_features = F.relu(self.fc_text(text_features))

        # 对文本特征进行Transformer编码
        text_features = self.text_encoder(text_features)

        # 协同注意力层，交互和协同处理两个模态的特征
        img_features, _, text_features, _ = self.co_attention(img_features,
text_features)

        # 独立的Transformer编码层
        img_features = self.encoder1(img_features)
        text_features = self.encoder2(text_features)

        # 将两个模态的特征拼接在一起
        output = torch.cat((img_features, text_features), dim=1)

        # 最终的全连接层输出
        output = self.fc(output)
```

```
        return output
```

协同注意力层在CoattentionTransformerLayer.py中实现，包含两个注意力子层和两个前馈网络（FeedForwardNet）：



(b) Our co-attention transformer layer

首先，定义了全连接前馈网络的结构（`PoswiseFeedForwardNet`），用于在协同注意力层中进行非线性变换。该前馈网络包含两个线性层和ReLU激活函数，用于处理输入张量，然后通过残差连接和层归一化，得到最终的输出。

```
class PoswiseFeedForwardNet(nn.Module):
    def __init__(self,d_model,device,d_ff=512):
        super(PoswiseFeedForwardNet, self).__init__()
        self.d_model = d_model
        self.device = device
        self.fc = nn.Sequential(
            nn.Linear(d_model, d_ff, bias=False),
            nn.ReLU(),
            nn.Linear(d_ff, d_model, bias=False)
        )

    def forward(self, inputs):
        """
        inputs: [batch_size, seq_len, d_model]
        """
        residual = inputs
        output = self.fc(inputs)
        return nn.LayerNorm(self.d_model).to(self.device)(output + residual)  #
[batch_size, seq_len, d_model]
```

接下来，定义了协同注意力层的主体结构（`CoAttentionEncoderLayer`）。该层包含两个注意力子层，每个子层都由一个多头注意力模块和一个前馈网络模块组成。在每个注意力子层中，通过线性变换（`enc_inputs * W_Q`，`enc_inputs * W_K`，`enc_inputs * W_V`）将输入进行映射，然后使用多头注意力机制处理两个模态的输入。最后，通过前馈网络进行非线性变换，并通过残差连接和层归一化得到输出。

```
class CoAttentionEncoderLayer(nn.Module):
```

```
    def __init__(self, embed_dim, device):
        super(CoAttentionEncoderLayer, self).__init__()
        self.enc_self_attn1 = nn.MultiheadAttention(embed_dim=embed_dim ,
num_heads=8)
        self.pos_ffn1 = PoswiseFeedForwardNet(d_model=embed_dim, device=device)
        self.enc_self_attn2 = nn.MultiheadAttention(embed_dim=embed_dim ,
num_heads=8)
        self.pos_ffn2 = PoswiseFeedForwardNet(d_model=embed_dim, device=device)

    def forward(self, enc_inputs1, enc_inputs2):
        """E
        enc_inputs: [batch_size, src_len, d_model]
        enc_self_attn_mask: [batch_size, src_len, src_len]  mask矩阵(pad mask or
sequence mask)
        """
        # enc_outputs: [batch_size, src_len, d_model], attn: [batch_size,
n_heads, src_len, src_len]
        # 第一个enc_inputs * W_Q = Q
        # 第二个enc_inputs * W_K = K
        # 第三个enc_inputs * W_V = V
        enc_outputs1, attn1 = self.enc_self_attn1(enc_inputs2, enc_inputs1,
enc_inputs1)  # enc_inputs to same Q,K,V（未线性变换前）
        enc_outputs1 = self.pos_ffn1(enc_outputs1)
        # enc_outputs: [batch_size, src_len, d_model]

        enc_outputs2, attn2 = self.enc_self_attn2(enc_inputs1, enc_inputs2,
enc_inputs2)  # enc_inputs to same Q,K,V（未线性变换前）
        enc_outputs2 = self.pos_ffn2(enc_outputs2)
        return enc_outputs1, attn1, enc_outputs2, attn2
```

3 实验结果

【消融实验】

	文本	图像
准确率	70.45%	78.04%

在消融实验中，文本特征的准确率为70.45%，相比之下，图像特征的准确率为78.04%。这表明了在该实验设置下，图像信息对于任务的贡献更为显著。

【四种模型融合效果】

	简单concat	GatedFusion	单流	双流
准确率	77.24%	80.56%	86.71%	83.71%

简单Concat模型，准确率为77.24%，模型直接拼接图像和文本特征，简单而快速，但可能无法充分挖掘模态间的语义关系。

GatedFusion模型，准确率为80.56%，模型通过门控机制，模型可以有选择性地关注图像或文本信息，提高了融合效果。实验还发现 `x_gate` 为真时，即门控机制作用于 `x`（图片特征），而 `y`（文本特征）被加权，分类效果比较好，这与消融实验中图像信息对于结果贡献更大的现象的结果也是吻合的。（`x_gate` 为真时，准确率为80.56%，`x_gate` 为假时，准确率为77.39%）

单流融合模型，准确率为86.71%，模型将两个模态的内容拼接后输入到Transformer中，将单模态内的交互和多模态之间的交互合并。它在参数效率上有优势，可能因为模态内外交互都得到了考虑。

双流融合模型，准确率83.71%，该模型在单模态场景下先进行模态内的交互，然后在多模态场景下进行多模态交互。这种方法可能在处理特定情况下具有优势，但在总体准确率上略低于单流融合模型。可能是因为在单模态场景下学到的特征对多模态场景不够适用。