

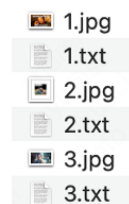
当代人工智能-实验五

github 仓库在 <https://github.com/Bruce-Jay/Multimodal-Sentiment-Analysis/tree/master>

数据预处理

提供的数据文件中，包含一个 data 文件夹，有若干对图片和txt文件，文件名相同的文件一一对应。还有两个 txt 文件，分别是有标签数据和需要被预测的数据，文件中每一条数据对应的 guid 就是 data 文件夹里面文件名所对应的图片和文字的标签。

- 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。
- train.txt: 数据的guid和对应的情感标签。
- test_without_label.txt: 数据的guid和空的情感标签。



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

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

我打算把所有的文字数据都整理到一个文件中，由于data中的txt文件是一个一个句子，其中必然有逗号，所以不能使用 csv 格式保存。于是打算使用以 tab 为分隔符的 tsv 文件格式保存。

读取文件并生成 tsv 的方法在 generate_data.py 文件夹下，这里不过多赘述。把 train.txt 以 80:20 的比例划分为训练集和验证集，分别保存在 train.tsv, valid.tsv.

生成的 tsv 文件格式如下，可以看到，每一行的第一列是 index，第二列是 guid，第三列是将 tag 转换成数字 ({negative: 0, neutral: 1, positive: 2}), 第四列是文字描述。

```
1 0 4597 RT @AmitSwami77: The conspirators have an evil eye & are now set to physically attack Asaram Bapu
2 1 26 Waxwing trills, Chickadees calling "here sweetie", enthusiastic athletes, blue sky & snow at #ualbert
3 0 4383 @NYSE is looking a little despondent today...??? http://t.co/o5xiKyJgT7
4 2 212 FERVENT | S,M,L | 140k free PLASTIC CLIP, keychain rubber AND sticker 085725737197 / 28ae36f3
5 2 2626 Nice day chilling in the park yesterday relieved my mood for a short while. #friends #summer #out
6 1 3042 Ford : F-350 Lariat 6.4L 2008 Lariat Heated Leather Rear Camera 2008 ford f 250 diesel 4 x...
7 2 4713 RT @MOVIE MEMORIES: Furious 7 http://t.co/CEPxKf3QLY
8 0 2073 @MattSmith1230 @ProFlowers The flowers look like a dejected King Tritan:
9 2 2020 #废墟 #廢線 #abandoned #写真撮ってる人と繋がりたい #写真好きな人と繋がりたい
10 0 2688 RT @Pablothemako: UPDATE!Navy discarded illegal fishing after boarding chinese vessels in #Chile's
11 2 4956 Arden B Womens Blue Solid Dress Sz M Medium Short Sleeve Modal Blend Above Knee http://t.co/hHbd7
12 2 3143 RT @dagtotdag: De eerste kilometers in open water met @Kokkie70 @63sjaak @SwimtoFC078
13 2 3624 DVF Diane von Furstenberg Naples Ankle Soft Canvas Blossom Black Tuxedo Pants 4
```

查看 train.tsv, valid.tsv 中的 tag 种类，以及每个种类所占的个数

tags/file	train.tsv	valid.tsv
Negative	941	252
Neutral	332	87
Positive	1926	462

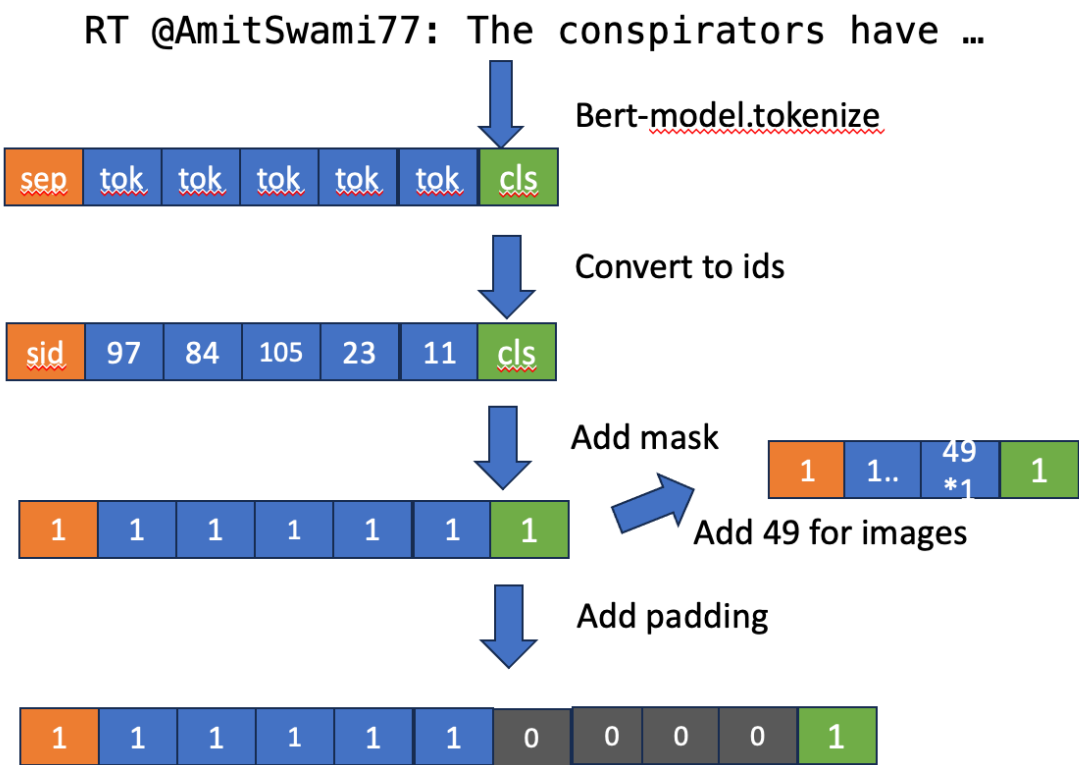
发现 neutral 标签所占的比例略少，如果真正在数据集上进行训练的话，可能会更容易造成过拟合。对于样本较为缺失的标签，一般的做法是在数据集中重复这些较少的数据较多次。尽管听上去没有什么理论支持，但是在实际操作时，这种方法是真实有效的。在许多 kaggle 竞赛上都得到了证实。

后续的计划是将标签为 neutral 的数据条数做复制，再进行模型的训练和拟合，观察对比效果。

把文字图像转化为可以训练的数据

我们读取刚刚得到的 tsv 文件，并且对其中的 text 进行 tokenize, convert_to_ids, add_padding 等一系列操作，最后再加上一条数据，这条数据会留49个空格给图像（图像经过处理后长度都为 49）

对于每一个输入的 text



对于图像数据，我们统一将其先转换为 224*224 的大小，方便模型进行训练

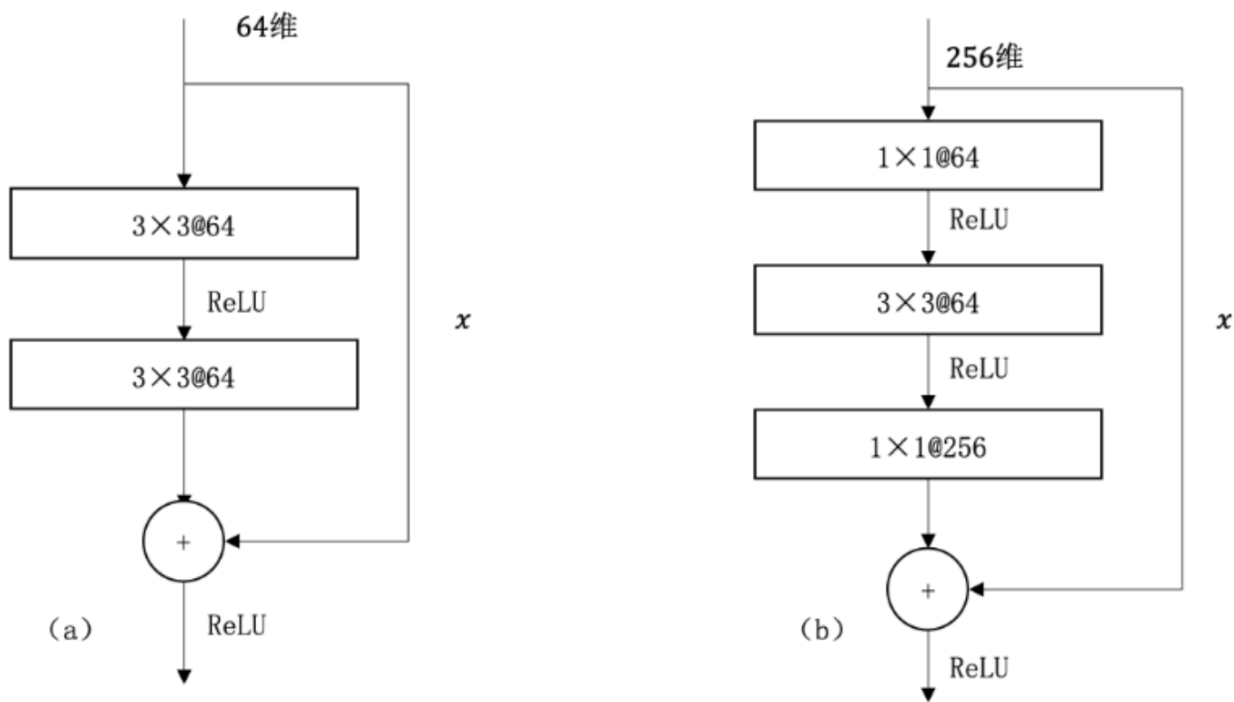
```
transform = transforms.Compose([
    transforms.RandomCrop(crop_size, pad_if_needed=True), # crop_size = 224*224
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.485, 0.456, 0.406),
                          (0.229, 0.224, 0.225)))])
```

对图像使用 RandomCrop 方法统一尺寸到 crop_size 以后，进行随机水平翻转，转换成 torch.tensor 数据格式，并且进行归一化。

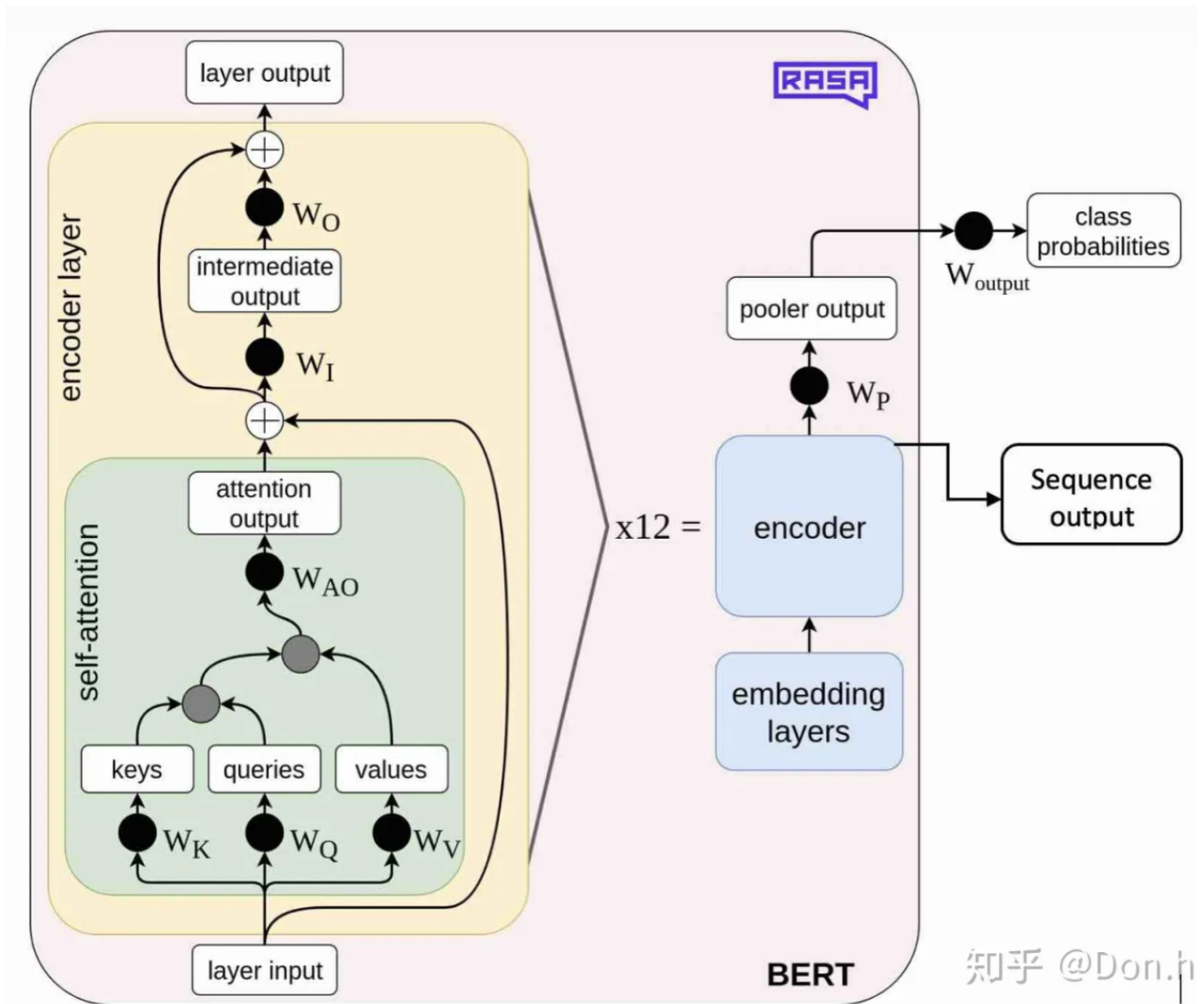
搭建模型

我们在文字部分选取的预训练模型是 bert-base-uncased, 图片部分选取的预训练模型是 resnet-152, 因为在实验三中我所得到的结论是 resnet 是一个准确率很高并且参数相对较小、不容易出现过拟合的模型。

残差神经网络的主要贡献是发现了“退化现象(Degradation)”，并针对退化现象发明了“快捷连接(Shortcut connection)”，极大的消除了深度过大的神经网络训练困难问题。神经网络的“深度”首次突破了 100 层、最大的神经网络甚至超过了 1000 层。非线性转换极大的提高了数据分类能力，但是，随着网络的深度不断的加大，我们在非线性转换方面已经走的太远，竟然无法实现线性转换。显然，在神经网络中增加线性转换分支成为很好的选择，于是，ResNet 团队在 ResNet 模块中增加了快捷连接分支，在线性转换和非线性转换之间寻求一个平衡。



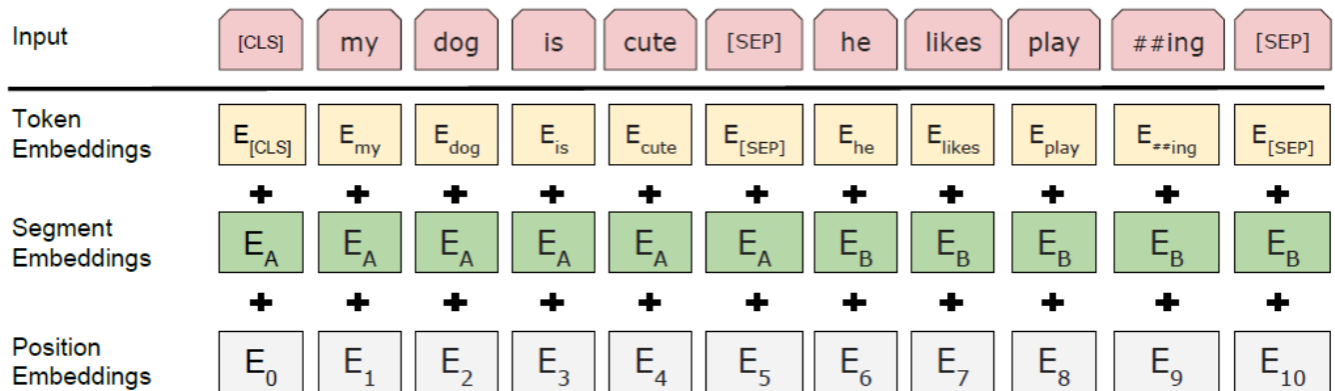
现在开始搭建 bert 模型



分为以下步骤：

把输入先做 embedding，并且将 token embeddings, segmentation embeddings, position embeddings 进行加和。将三种不同的 embedding 进行加和与单独的 embedding 输入相比，使用了不同的角度和方式进行了编码，可以让模型学到更多的特征信息。

- Token Embeddings：用于将输入的单词序列转换为向量表示，作用是将每个输入单词都表示为一个向量，并包含了该单词的语义信息。
- Segment Embeddings：用于区分两个句子中相同位置的单词，作用是将不同句子中相同位置的单词进行区分，避免不同句子中的单词产生干扰。
- Position Embeddings：用于表示单词在输入序列中的位置，作用是将输入序列中单词的位置信息编码成向量，以便模型能够理解单词在序列中的顺序关系。



之后我们添加层归一化和dropout 防止过拟合与防止梯度爆炸或者消失。

之后我们将输入的数据进行应用多头注意力机制。

分别定义 qkv

```
self.query = nn.Linear(config.hidden_size, self.all_head_size)
self.key = nn.Linear(config.hidden_size, self.all_head_size)
self.value = nn.Linear(config.hidden_size, self.all_head_size)
```

将 qkv 进行multihead拆分, 拆成12个头, 拆分后的 qkv 保存在key_layer, query_layer, value_layer 中, 维度为 (1,12,128,64)

```
new_x_shape = x.size()[:-1] + (self.num_attention_heads, self.attention_head_size)
x = x.view(*new_x_shape)
return x.permute(0, 2, 1, 3)
```

attention_scores 矩阵是由 Q 和 K 矩阵相乘得到的, 用于计算注意力分数, 注意力权重经过 Softmax 函数和 Dropout 操作后, 我们可以得到每个单词对其他单词的注意力权重, 并计算上下文向量。最终返回的是上下文向量, 将每个注意力头的向量拼接在一起, 形成一个更大的向量维度。

```
query_layer = self.transpose_for_scores(mixed_query_layer)
key_layer = self.transpose_for_scores(mixed_key_layer)
value_layer = self.transpose_for_scores(mixed_value_layer)

attention_scores = torch.matmul(query_layer, key_layer.transpose(-1, -2))
attention_scores = attention_scores / math.sqrt(self.attention_head_size)
attention_scores = attention_scores + attention_mask

attention_probs = nn.Softmax(dim=-1)(attention_scores)
attention_probs = self.dropout(attention_probs)

context_layer = torch.matmul(attention_probs, value_layer)
context_layer = context_layer.permute(0, 2, 1, 3).contiguous()
new_context_layer_shape = context_layer.size()[:-2] + (self.all_head_size,)
context_layer = context_layer.view(*new_context_layer_shape)
```

intermediate 层，也就是 Feedforward 层，用于对输入的隐藏状态向量进行非线性变换，提取更高层次的特征表示

```
class BertIntermediate(nn.Module):
    def __init__(self, config):
        super(BertIntermediate, self).__init__()
        self.dense = nn.Linear(config.hidden_size, config.intermediate_size)
        self.intermediate_act_fn = ACT2FN[config.hidden_act] \
            if isinstance(config.hidden_act, str) else config.hidden_act

    def forward(self, hidden_states):
        hidden_states = self.dense(hidden_states)
        hidden_states = self.intermediate_act_fn(hidden_states)
        return hidden_states
```

根据以上的代码，可以构建一个 bert encoder. bert encoder 层一层包括 bert input, bert layer 和 bert output, 其中 bert layer 层又包括 self-attention 层和 intermediate 和 output 层。

```
class BertLayer(nn.Module):
    def __init__(self, config):
        super(BertLayer, self).__init__()
        self.attention = BertAttention(config)
        self.intermediate = BertIntermediate(config)
        self.output = BertOutput(config)

    def forward(self, hidden_states, attention_mask):
        attention_output = self.attention(hidden_states, attention_mask)
        intermediate_output = self.intermediate(attention_output)
        layer_output = self.output(intermediate_output, attention_output)
        return layer_output

class BertEncoder(nn.Module):
    def __init__(self, config):
        super(BertEncoder, self).__init__()
        layer = BertLayer(config)
        self.layer = nn.ModuleList([copy.deepcopy(layer) for _ in
range(config.num_hidden_layers)])

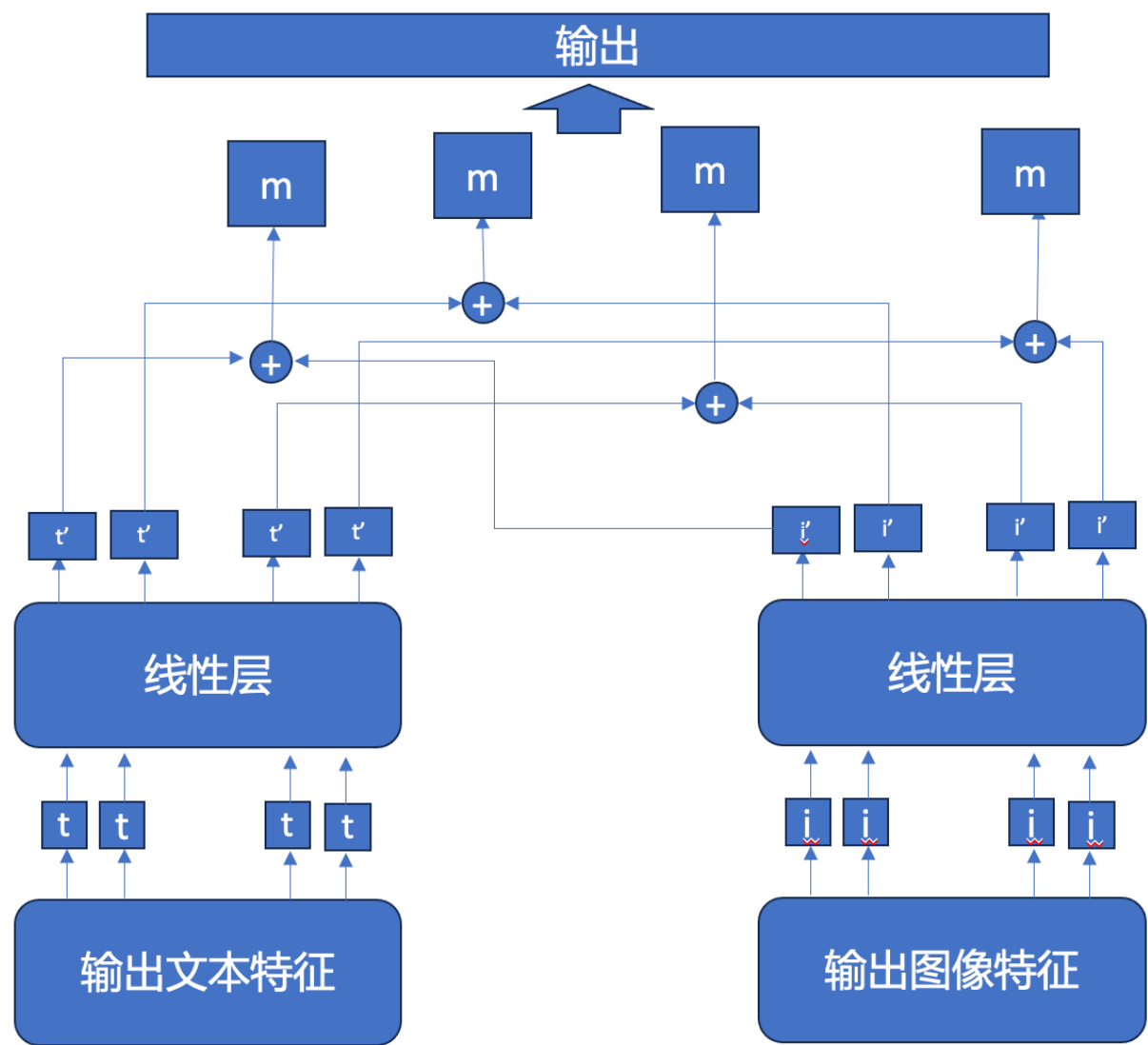
    def forward(self, hidden_states, attention_mask, output_all_encoded_layers=True):
        all_encoder_layers = []
        for layer_module in self.layer:
            hidden_states = layer_module(hidden_states, attention_mask)
            if output_all_encoded_layers:
                all_encoder_layers.append(hidden_states)
        if not output_all_encoded_layers:
            all_encoder_layers.append(hidden_states)
        return all_encoder_layers
```

定义 pooler 层，作用是将模型的最后一层输出向量转换为一个固定长度的向量，用于表示整个输入序列的语义特征，可以用于下游任务的处理，如分类、序列标注等。

```
class BertPooler(nn.Module):
    def __init__(self, config):
        super(BertPooler, self).__init__()
        self.dense = nn.Linear(config.hidden_size, config.hidden_size)
        self.activation = nn.Tanh()

    def forward(self, hidden_states):
        # We "pool" the model by simply taking the hidden state corresponding
        # to the first token.
        first_token_tensor = hidden_states[:, 0]
        pooled_output = self.dense(first_token_tensor)
        pooled_output = self.activation(pooled_output)
        return pooled_output
```

然后我们还需要接受多模态输入。我们可以将 文本+图像的输入也做一个 cross attention。Transformer 解码从完整的输入序列开始，但解码序列为空。 交叉注意力将信息从输入序列引入到解码器的各层，以便它可以预测下一个输出序列标记。 然后解码器将令牌添加到输出序列中，并重复此自回归过程，直到生成 EOS 令牌。



运行程序，消融实验

输入 图像+文本，仅文本，仅图像 的准确率和 f1 值分别如下：

	图像+文本	仅文本	仅图像
Accuracy	71.34	64.67	59.80
F1 score	70.62	63.29	49.89

可以发现，我们的模型在多模态问题上的表现较好，判断正确率在 0.7 左右。并且远高于两个单模态的模型输出结果。推测文本数据表现好于图像数据的可能性是，一个图像可以表示更多范围的情绪，而文本似乎略少一些。

对数据集的标签进行探索，训练

前文提到，neutral 标签所占的比例略少，如果真正在数据集上进行训练的话，可能会更容易造成过拟合。对于样本较为缺失的标签，一般的做法是在数据集中重复这些较少的数据较多次。

定义了 fix_overfit 方法，用于给训练集和验证集进行缺失标签的数据填补。把训练数据和验证数据里面 label 为 1 的数据先添加在末尾，然后将里面的行随机排序生成新文件。

修改以后，训练数据集中，label 为 0,1,2 的数据分别为 941 995 1926 条。验证数据集中，label 为 0,1,2 的数据分别为 252 260 461 条。相比于以前 2:1:3 的数据分布比，现在 1:1:2 的数据分布比似乎更加健壮。

同时，发现数据集中有多条日语、俄语数据，我将其都翻译成了英文，来提升模型的准确率。

进行训练与验证，得到的结果如下：

	图像+文本	仅文本	仅图像
Accuracy	64.74	55.60	47.99
F1 score	63.79	54.54	33.18

发现得到的准确率比原数据集更低，这也比较合理，因为原本的数据集，与其说是一个三分类问题，不如更加靠近一个二分类问题。假设模型有先验知识，知道各种标签所对应的数据比例是 2:1:3, 那么准确率的期望值就是 $0.5*0.5 + 0.33*0.33 + 0.16*0.16 = 0.375$. 而如果知道各种数据的比例是 1:1:2, 准确率的期望值是 $0.25*0.25 + 0.25*0.25 + 0.5*0.5 = 0.365$, 会略低一些。

但是双模态和仅文本的表现稍微还符合这样的比例，但是仅图像就太低了。F1 score 到了 33.18，不如随机猜测。因此判断这个模型的效果不好。

我们来查看预测生成的数据，在 predict 文件夹下。旧数据集和新数据集所对应的预测结果分别是 text_without_label_legacy.txt 和 text_without_label.txt 。发现前者中，标签为 positive 的较多，并且三种标签分布得较为均匀。而后者中，标签为 neutral 的过多，和原始数据的分布不太吻合。舍弃。

所以目前来讲，在这个例子中，使用原数据集表现更好。

总结

在本文中，我们对多模态情感分析任务对于具体数据进行了建模。我们使用 bert 和 resnet 对于文本和图像分别进行了变换并且进行向量拼接，最后喂进模型训练，取得了较好的表现。同时，我们对于数据集进行了探索，发现如果重复标签为 neutral 的数据过多，无论是训练结果还是预测结果都不太理想。我们也进行了消融实验，分别对于仅文本和仅图像的准确率进行了探索，对于多模态模型的表现有了更深刻的了解。

遇到的问题

```
Traceback (most recent call last):
  File "/tmp/pycharm_project_417/run.py", line 92, in <module>
    main()
  File "/tmp/pycharm_project_417/run.py", line 78, in main
    train(args, train_examples, num_train_steps, label_list, optimizer, scheduler, model, encoder, global_step, tokenizer)
  File "/tmp/pycharm_project_417/processors/util.py", line 169, in train
    all_img_feats = torch.tensor([f.img_feat for f in train_features])
ValueError: only one element tensors can be converted to Python scalars
```

使用 torch.tensor 转换时，忘记之前已经通过 torch.transformer 转换成了一系列 torch.tensor 数据类型，list 列表中每一个元素都是一个 tensor 类型的数据。所以应该使用 torch.stack 方法将这些 list 里面的元素拼接起来变成一个 torch.tensor 类型的数据。