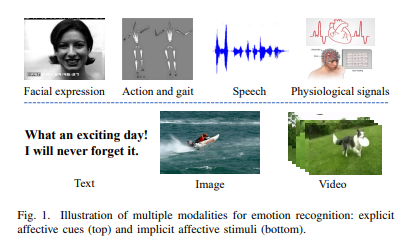
多模态情感分析

## 背景知识

因为我对于多模态情感分析不甚了解，所以我决定去阅读一篇综述来了解这个领域。我读的是Emotion recognition from multiple modalities: Fundamentals and methodologies，链接为：https://arxiv.org/pdf/2108.10152.pdf。由于论文中还包含了很多与本次实验无关的内容，所以我只会记录下一些比较重要且相关的知识点。

### 情感模态的分类

情感模态（affective modality）大体上可被分为显式和隐式两类：



显式指的是可被直接观察到的人类的特定物理上和心理上的反应，比如说面部表情或者手势。隐式指的是常见的电子媒介。通过它们，可以间接地推断人类情绪，比如说在大家在社交平台上上传的照片。本次实验所给的训练资料，即图像与文本，就是隐式的。

文本是记录人类自然语言的一种形式，它隐含着情感信息。文本有不同层次的语言结构，包括单词、句子、段落和文章。目前，已经开发出了很多算法可以把文本分割成小块。然后在一些字典的帮助下，就可以识别每个语言片段的情感属性，并据此推断出文本所唤起的情感。

图像是彩色点在空间上的分布。心理学上已经证明，人类的情感是可以通过图像唤起的。网上爆炸式的图片分享和对场景的强大描述能力使得图像成为关键的情感刺激，已经成为了重要的研究对象。

### 多模态相比单模态的优势

多模态相比起单模态有着以下的优点：

* 不同模态的数据能够互相补充。有时候我们很难单纯从语言判断一个人的情绪，因为有反话这种东西的存在。一个人说“今天天气真好啊”，有可能ta真的是这个意思，也有可能是在说反话。这时候，如果有额外的图片信息就可以帮助我们做出判断了。
* 使得模型的健壮性变强。因为数据收集的过程中可能会出现种种问题，导致某一模态的数据无法获取。这种情况是常常发生的。一旦这种情况发生，多模态的情感识别模型仍然能够运作，因为有其他模态的数据可以使用。
* 性能上更加优越。这点其实和第一点是一脉相承的，数据之间互相补充就能够获得更高的准确率。

### 心理学中的两种代表性模型

在心理学中，CES（categorical emotion states）和DES（dimensional emotion space）是两类比较具有代表性的模型。前者将情感定义为几个基本类别，就比如本次实验的positive, negative和neutral三个类别；后者采用连续的二维、三维或更高维度的笛卡尔空间来表示情感。

CES 模型更符合人类直觉，但心理学家对应该包括多少个离散的情感类别还没有达成共识。此外，情感是复杂而微妙的，有限的离散类别无法精确刻画出情感。理论上来说，DES模型可以将任何情感映射为情感维度空间中的坐标点，但我们人类本身是很难理解这些连续的值的，这些连续值只对机器有着特殊的意义。

本次实验应该是属于CES，因为三个标签positive，negative和neutral是离散的。

### 本次实验所属计算任务

接下来就是计算任务了。经过阅读后，我认为本次实验属于情感分类（ Emotion Classification）任务当中的单标签学习（SLL）问题。情感分类任务中的假设是一个实例只能属于一种或固定数量的情感类别，目的是发现数据空间中的类别边界或类别分布。目前的工作主要集中在人工设计多模态特征编码器和分类器，或以端到端方式应用深度神经网络。

对于单标签学习（SLL）问题，MER 为每个样本分配一个单一的情感标签。然而，相较于单一的代表性情感，情感更可能由来自不同区域或序列的不同情感成分混合而成。而且，不同的人对同一刺激可能有不同的情感反应，这是由性格等多种因素造成的。所以，也就有了多标签学习（MLL）。为了应对MLL出现的问题，又出现了标签分布学习（LDL）。但是这两个任务并非是本次实验的所属任务，因此也就不详细介绍。

### 多模态情感分析的阻碍

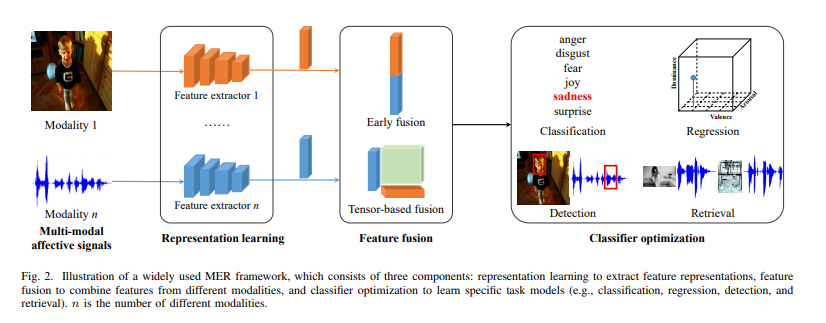
虽然与单模态相比，多模态情感分析有着许多优点，但也不可避免地存在许多缺点：

* 情感差距
  + 这里的定义有点抽象，它用来衡量提取出来特征和感知到的高级情感间的不一致性。
  + 举个例子，盛开的玫瑰和凋谢的玫瑰都包含玫瑰，但却能唤起不同的情感；对于同一个句子，不同的语音语调可能对应完全不同的情感。
  + 提取具有区分性的高级特征，特别是与情感有关的特征，可以帮助弥补情感上的差距。其中存在的主要困难之一是如何评估提取的特征是否与情感有关。
* 认识主观性
  + 由于许多个人、环境和心理因素的影响（如文化背景、个性和社会环境），不同的人对同一刺激可能有不同的情感反应。即使情感相同，生理和心理变化也会有很大差异。
* 数据不完整性
  + 由于数据收集过程中存在许多不可避免的因素（如传感器设备故障），特定模态的信息可能会被破坏，从而导致数据缺失或不完整。
  + 对于前文所说的显式情感模态来说，通过传感器采集到的信号常常会带有噪声；而对于隐式情感模态来说，发朋友圈也可能只带一张图而没有任何文字。当然，这个的影响对于多模态来说并不致命，毕竟缺少了一个模态还有其他的模态。当然，要是所有模态的数据全都缺失，那就比较致命了，但这属于极端情况，不常发生。
* 跨模态不一致性
  + 同一样本的不同模态可能相互冲突，表达不同的情感。
  + 不同模态的数据相互补充只是理想的情况，实际上人在朋友圈发的图像和文本可能是没有任何关联的。有时候和当时的心情无关，就是想配那张图。我本人也有过这样的经历。
* 跨模态不平衡性
  + 不同的模态可能对唤起情感有不同程度的贡献，比如说一篇文章的图片可能非常少，但是文字却非常长。
* 标签噪声与缺失
  + 目前的多模态情感分析方法，尤其是建立在深度学习之上的方法需要大规模的标注数据进行训练。但是在实际应用中，标注情感不仅成本过高、耗时过长，而且标注结果也高度不一致。这就导致了我们可能拥有大量的数据，却只有很少甚至没有情感标签。
  + 而且，随着情感需求的日益多样化和细粒度，也可能某些情感类别有足够的训练数据，其他情感类别却没有。

由于本次实验的数据集是直接给定的，所以上述的六个阻碍当中只有情感差距和跨模态不平衡性是本次实验的真正阻碍。（其他的阻碍并不是本次实验的真正阻碍，因为我认为它们是搜集数据时候的阻碍，而并非是训练时候会出现的问题）

### 计算方法

一般一个多模态情感分析任务的框架是这样的：



由representation learning、feature fusion和classifier optimization三个部分组成。

我在这个板块也同样只会记录与图像和文本有关的内容。

#### representation learning

首先是文本。为了将文本表示为计算机可以理解的形式，需要以下的步骤：

1. 将符号化的词语表示为实数
2. 对语义关系建模
3. 获得整段文本统一的表示方法

在最初的最初，单词通过one-hot编码进行表示。这种表示方式，不仅会让向量的维度随着词汇表大小的增加而增加，也无法体现出单词之间的关系。后来，研究人员利用语言模型预测上下文来训练，得到具有固定维度的词向量。流行的词向量表示模型包括word2vec，GLOVE，BERT，XLNet 等。文本特征提取方法也从简单发展到复杂。而文本特征可以通过简单的平均化词向量来获得。

而对于图像来说，正如在本学期的计算机视觉中所学到的一样，在最初的最初，是通过人为设计出来的特征来表示情感，包括颜色、纹理和形状等。这些低层特征是在心理学和艺术理论的启发下设计的。后来基于视觉概念的中层特征被提出，用以缩小图像中像素和情感标签之间的差距。最具代表性的项目是SentiBank，它由1, 200对形容词-名词组成，在所有的手工特征中展现出卓越而稳健的识别性能。在深度学习领域，CNN 被认为是一个强大的端到端特征提取器。具体来说，为了整合不同层次的各种表示，特征从CNN 的不同层中提取而出。同时，注意力机制被用来学习特定局部情感区域的更好的情感表示。

#### feature fusion

feature fusion通过整合来自多种模态的表示，以预测情感的特定类别或连续值。一般来说有两种策略：model-free和model-based。

##### model-free

不直接依赖特定学习算法的model-free fusion在过去几十年被广泛使用。它分为early fusion、late fusion和hybrid fusion。

early fusion直接将来自不同模态的特征表示拼接为一个表示。这是一种最直观的方法，利用早期各模态间的交互来融合各表示，这种方法只需要训练一个单一的模型。（后面提到了可能失败的情况，我不是很理解后面的时间同步问题是什么）。

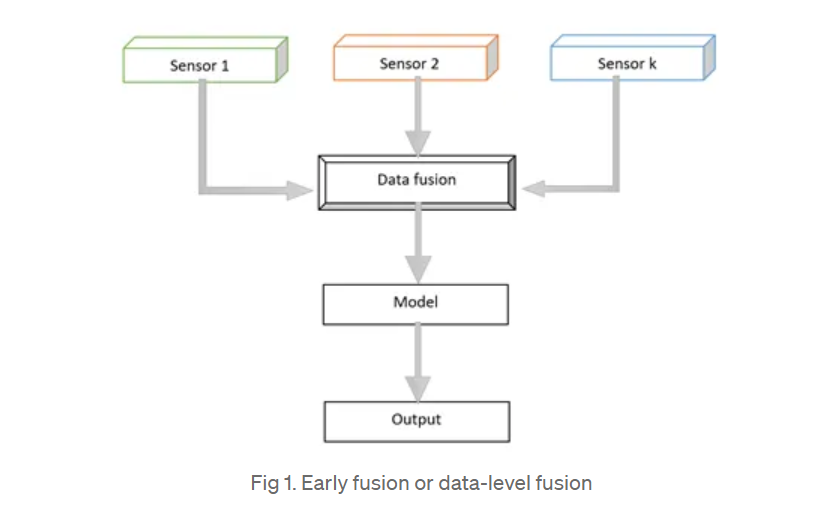
late fusion整合了来自每个单模态的预测结果。流行的机制有平均、投票和信号方差。晚期融合的优点包括：

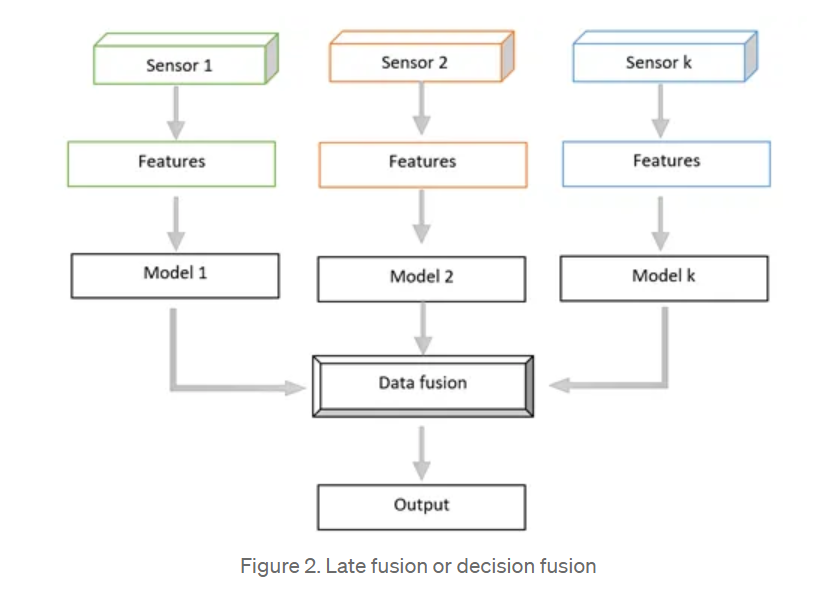
* 灵活性和优越性：可以为不同的模态选择最优分类器
* 健壮性：当某些模态缺失时，晚期融合仍然可以发挥作用。

但late fusion忽略了不同模态做出预测前的关联。

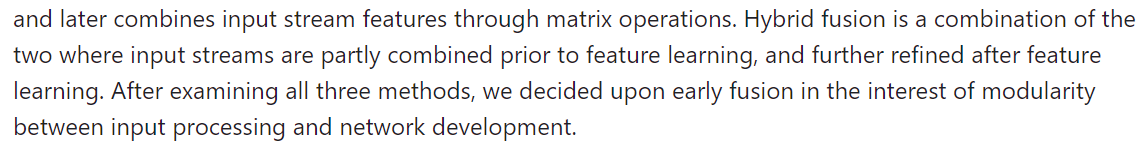
hybrid fusion结合了early fusion和late fusion，在一个统一的框架内利用它们的优势，但计算成本更高。

论文中在讲解这块内容的时候全部都是文字，我个人其实没有完全理解。所以后来我又找到一篇文章https://medium.com/haileleol-tibebu/data-fusion-78e68e65b2d1。这里面的图示更加清楚些：





此外这里的hybrid fusion我不是很能理解是如何将early fusion和late fusion结合起来的。[snoiarao/registration\_trials: 3-Dimensional registration utilities for Sonia Rao's THINC Lab @ UGA research project (github.com)](https://github.com/snoiarao/registration_trials)中提到：

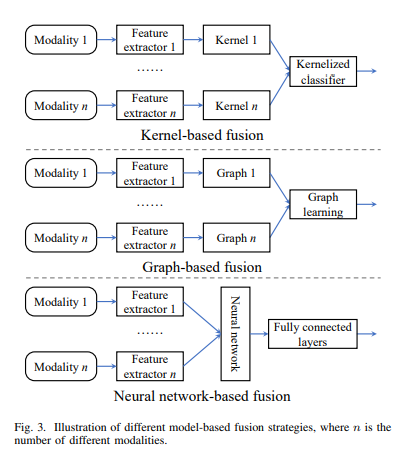


hybrid fusion就是将部分的输入在特征学习之前就结合，并且还要在特征学习结束之后进一步优化。似乎这意味着输入的模态至少需要三个，因为只有两个的话直接结合后，就没有多余的模态信息来帮助优化了。

除此之外我也找不到有关hybrid fusion的有效信息了，所以本次实验设计中并没有考虑hybrid fusion。

##### model-based

model-free fusion是基于一些简单技术，而不是专门针对多模态数据的，所以相较之下，在构建学习模型过程中明确进行融合的model-based的融合受到了更多关注。对于浅层模型，kernel-based和graph-based是两种代表性的方法；对于最近流行的深层模型，则经常使用neural network-based fusion, attention-based fusion, 和tensor-based fusion。



这里的kernel-based fusion和graph-based fusion我读论文的时候其实不怎么看得懂，就略过不谈了，只谈谈第三种。

neural network-based fusion采用直观的策略，通过神经网络融合不同模式的特征表示或预测结果。

attention-based fusion使用注意力机制来获得一组带标量权重的表示的加权和，这些权重由注意力模块动态学习所得。不同的注意力机制用于融合不同的部分。

tensor-based fusion则通过一些特定的张量操作来利用不同表示的相关性。

这些用于深度模型的融合方法能高性能地以端到端方式从大量数据中学习，但存在可解释性低的问题。

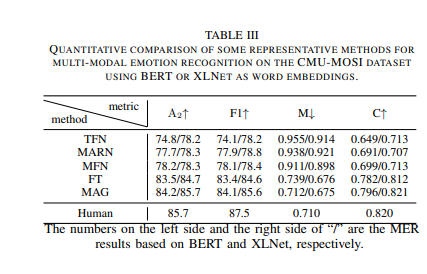
#### classifier optimization

对于用单词嵌入序列表示的文本，最流行的方法是使用RNN和CNN捕捉单词之间的语义关系。LSTM，作为一种经典的RNN，包含一系列具有相同结构的单元。每个单元接受一个词编码和上一个单元的隐藏状态作为输入，计算输出，并为下一个单元更新隐藏状态。隐藏状态记录了之前的词的语义。CNN通过卷积操作计算连续的词之间的局部上下文特征。平均池化层或最大池化层用于进一步整合所提取的特征，以进行后续情感分类。最近，研究人员开始使用基于Transformer的方法，例如BERT 和GPT-3。与RNN相比，Transformer不需要按顺序处理单词，这提高了并行性；而与CNN相比，Transformer可以对距离更远的词之间的关系进行建模。

在早期，与其他模态类似，很多人工设计出来的图像特征被整合并输入到SVM中以训练分类器。随后基于深度学习，分类器和特征提取器被连接起来，并通过相应的损失函数（比如交叉熵）进行端到端的优化。在经历上述的过程之后，每幅图像被预测为一种主要情感类别。后续论文中还提到了上述过程的不足，就是没有考虑到图像能够唤起不同种类的情感的情况。但是解决方法我其实没怎么看懂，论文中说要学习标签的概率分布来预测每个情感类别的具体相关程度，还要用到KL散度作为损失函数。但是KL散度其实就是交叉熵减掉信息熵，彼此之间只差了一个常数，求导之后就没区别了。而且它们都是用来衡量两个分布之间的相似性的，我个人感觉区别不是很大，不太清楚为什么要使用KL散度代替交叉熵。

## 实验设计

基于以上的背景知识，我最终是这样设计的：

* representation learning：
  + 分别使用Bert和XLNet处理文本
    - 因为论文中对比了这两个模型在不同数据集上的表现，所以我也就测试一下这两种究竟哪一个更适合本次的数据集：
    - 
    - 论文中的结论是：在CMU-MOSI数据集上，BERT 和XLNet 能提供比GLOVE更好的词编码，而XLNet 一般比BERT更好。
  + 分别使用ResNet152和MobileNetV3处理图像
    - 选择这两个架构主要是因为第三次实验中学习到的知识。ResNet是一个很经典的结构了，使用了残差结构，使得我们能够训练出更加深的网络，准确率更高。而MobileNet走的是和ResNet不一样的方向，注重的是运算速度（当然也并不是说不注重准确率，准确率离谱的低那也不行）。
    - 两者的具体原理第三次实验中已经详细说明了，我这里就不赘述了。
    - 选择这两个架构主要是因为想看看来自于不同motivation的两个架构在多模态情感识别这个任务上的表现差异。
* feature fusion
  + 使用了model-free中的early fusion和late fusion。其中，early fusion中还实现了一种变体，可以解决先前所提到的跨模态不平衡性的问题，即不同的模态对情感的贡献不同的情况。
    - 本次实验中一开始我对于tensor-based fusion的理解存在问题，以为early fusion的变体就是tensor-based fusion，但经过查阅资料才发现不是。所以后续的名称中带有Torch的其实都是early fusion的变体。
* classifier optimization：这部分因为我并不是很明白论文的内容，所以并没有做什么特殊的处理。

## 实验过程

这板块我就记录了一些值得写下来的注意点，一些比较常规的代码，比如说train函数的书写和比较基础的一些PyTorch定义网络结构的知识我并没有写下来。

### 数据预处理

由于原本的文本，图像，标签都是分别存放的。我希望能够先将guid，tag和text保存在一个csv文件中，方便main.py中直接读取（至于为什么不把image也一并放入这个csv文件中，理由会在之后遇到的问题板块中说明）。

当然，我们还需要对文本内容进行一些预处理。看几个文本的例子吧：

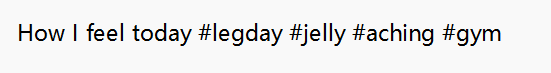
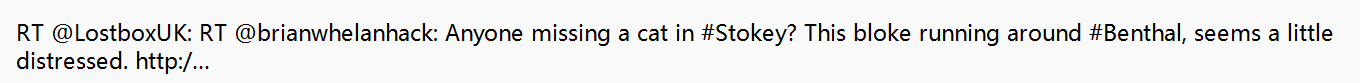


fig:



其实如果大家比较熟悉的话，应该是能够推测出来这是某社交平台上的数据了。

@后面跟随的应该是用户的账号名字，而#后面跟随的就是一些标签了。前者是对我们本次的任务没有什么帮助的，用户的名字并不能帮我们判断情感，而后者显然是有用的。

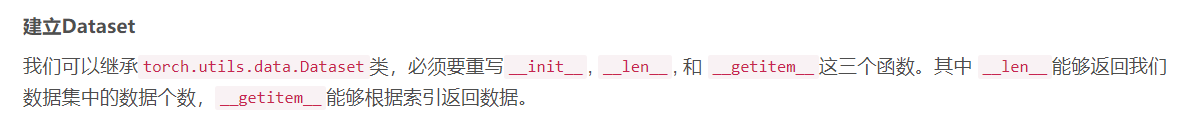
从第三个例子可以看出，某些数据中还抱着以http开头的一些链接，这对于情感识别显然也是没有帮助的。

所以，我们需要做的就是将按照空格后分词的含有@和http的全部都去掉。当然在分词前要记得把#去掉，只留下#后面的标签。

转换后的新数据集就是/data/text\_tag\_train.csv和/data/text\_test.csv。

### 如何生成Dataloader

后面会提到我原本是想使用transformer库提供的Dataset和Trainer组件进行训练的，但因为一些问题，所以只好手动实现训练过程。而在手动实现的过程中，就必然需要将数据转化成Dataloader。如何转化？我参考了这篇博客：[pytorch中使用Dataset和DataLoader创建自定义数据集 入门\_自定义dataset和dataloader统一处理图像数据集和文本数据集-CSDN博客](https://blog.csdn.net/qq_43219379/article/details/123381194)。只需要：



代码如下：

class CustomDataset(Dataset):  
 def \_\_init\_\_(self, image\_data, tags, inputs, masks):  
 self.image\_data = image\_data  
 if type(tags) != type([]):  
 self.tags = tags.tolist()  
 self.input\_ids\_data = inputs  
 self.attention\_mask\_data = masks  
   
 def \_\_len\_\_(self):  
 return len(self.image\_data)  
  
 def \_\_getitem\_\_(self, index):  
 image = self.image\_data[index]  
 tag = self.tags[index]  
 input\_ids = self.input\_ids\_data[index]  
 attention\_mask = self.attention\_mask\_data[index]  
  
 return image, tag, input\_ids, attention\_mask

值得注意的是我并没有存储文本，因为原始文本在巡礼过程中是不被需要的，只需要经过tokenizer之后的input\_ids和attention\_mask即可。

之后把CustomDataset的实例传入Dataloader中即可。

### PIL与PyTorch处理图像的不同

通过Pillow提供的Image工具对图像进行处理后所得到的数据的形状是 (height, width, channels)。而在深度学习中，通常将通道维度放在第一维，即 (channels, height, width)。所以我们需要把数据转化成 (channels, height, width)的形式，这点可以通过transpose方法做到：

image.transpose(2, 0, 1)

transpose(2, 0, 1) 的参数 (2, 0, 1) 指定了轴的顺序，将原始数组的第三维（通道维度）移动到第一维，同时交换了高度和宽度的位置。

### flatten与view使用的注意点

无论你是用x.flatten()还是用x.view(x.shape[0], -1)来将一个张量展平，我们都不能忘记这两个方法都返回一个新的张量，而不会修改原始张量。

所以如果使用flaten()或者view()，一定要记得重新赋值。

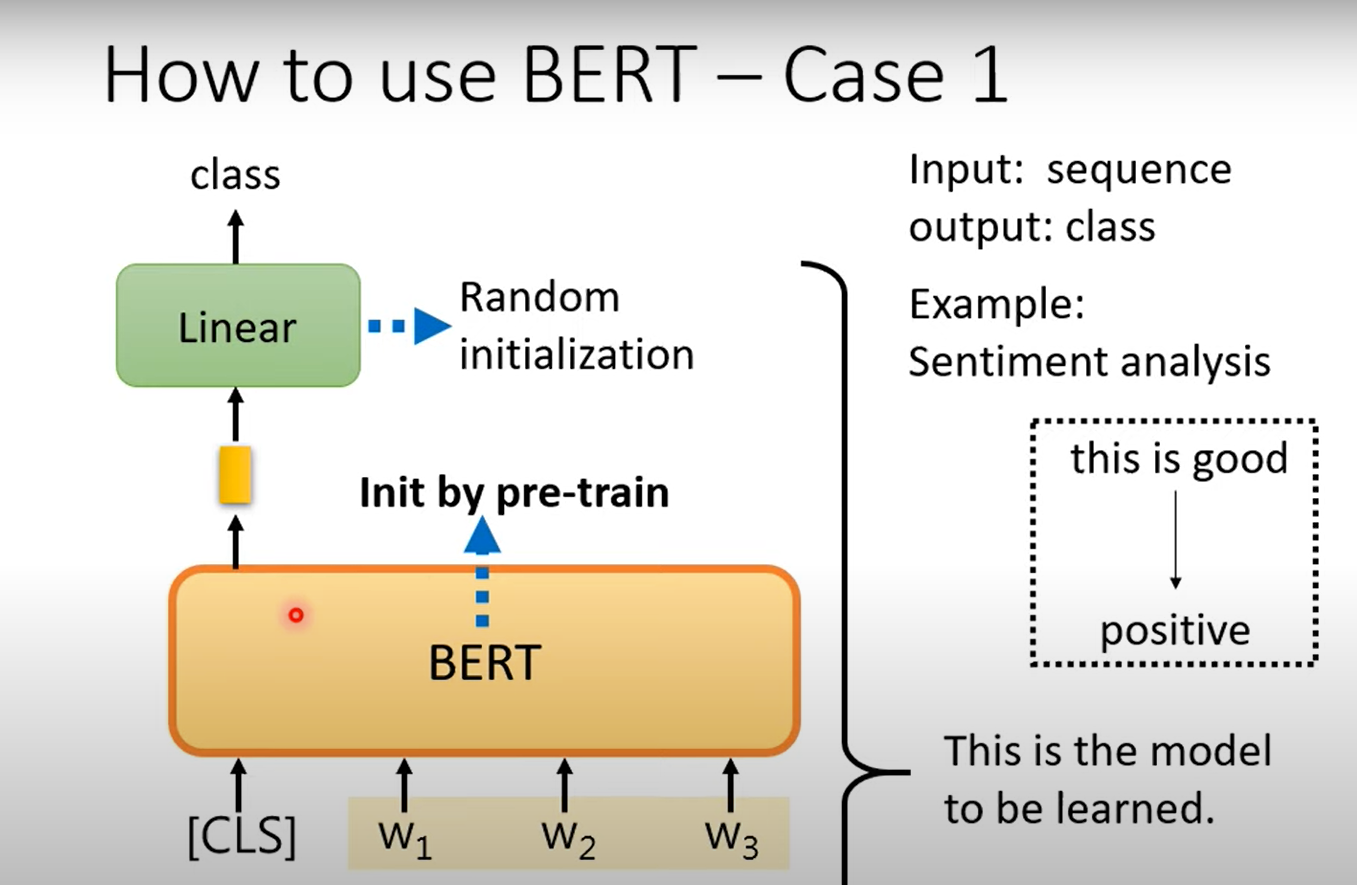
但是呢，这一点在本次实验中其实没有那么重要。理由在于PyTorch中的nn.Linear层在接收多维张量作为输入时，会自动将输入张量展平成一维张量。这是因为nn.Linear层期望的输入是一个包含batch\_size的二维张量，形状为(batch\_size, input\_features)。

如果输入张量是一个多维张量，nn.Linear会将其自动展平。举个例子，假设输入是一个三维张量，形状为 (batch\_size, height, width)，nn.Linear会将其视为 (batch\_size, height \* width)的二维张量，然后进行线性变换。

### Bert中CLS的表示如何获得

我本学期因为周三的通识课的冲突以及发烧了一次，导致老师所讲的self attention机制和transfomer我并没有听到，所以只能自己网上找资源学习。我这里看的是李宏毅所讲的机器学习课程：[【機器學習2021】預測本頻道觀看人數 (上) - 機器學習基本概念簡介 (youtube.com)](https://www.youtube.com/playlist?list=PLJV_el3uVTsMhtt7_Y6sgTHGHp1Vb2P2J)

具体有关Bert的介绍我就不赘述了。



这里老师告诉我们，CLS的表示就是整个文本的表示。它表示“分类”（Classification）。BERT 在预训练阶段的输入中，将一段文本或句子加上 [CLS]标记，并将整个序列的表示用于下游任务，例如文本分类。

那么该如何CLS的表示呢？只需要通过last\_hidden\_state[:, 0, :]即可。Last\_hidden\_state是模型的最后一个隐藏状态，它的形状是 [batch\_size, sequence\_length, hidden\_size]，其中 batch\_size是输入样本的数量，sequence\_length 是序列的长度，hidden\_size是隐藏状态的维度。所以，[:, 0, :]就提取了每个样本的第一个标记（即 [CLS] 标记）的表示。

XLNet也是类似的。

### early fusion与late fusion

这两个其实是很相似的：early fusion是将文本的表示与图像的表示拼接之后放入一个全连接层当中，就能获得三个类别上的分数，取其中分数最大的为预测的类别；late fusion是将文本的表示和图像的表示分别放入全连接层，就能够获得文本和图像各自三个类别的分数，然后我们取一个平均（也可也i采用投票和信号等方式，我这里选取了最简单的取平均），取分数最大的为预测的类别。

# early fusion  
last\_out = self.fc(self.relu(torch.cat((txt\_out, img\_out), dim=-1)))

# late fusion  
# 经过各自的最终线性层  
txt\_out\_final = self.fc\_txt(txt\_out)  
img\_out\_final = self.fc\_img(img\_out)  
  
# 对文本和图像的最终输出进行平均  
avg\_out = (txt\_out\_final + img\_out\_final) / 2.0

### early fusion的改进版本

最简单的early fusion就是直接将文本的表示与图像的表示直接叠加起来，但这显然忽略了一个问题，就是先前所提到的跨模态不平衡性，即不同的模态对于情感的唤起的贡献是不一样的。

所以，我们需要分别给文本的表示和图像的表示一个权重，用于表示其对于情感的唤起做出了多大贡献：

self.image\_gate = nn.Linear(128, 1)   
self.text\_gate = nn.Linear(128, 1)

然后在最后一个全连接层之前将各自的表示乘上各自的权重即可：

image\_weight = self.image\_gate(image\_out)   
text\_weight = self.text\_gate(text\_out)   
fused\_out = image\_weight \* image\_out + text\_weight \* text\_out   
final\_out = self.final\_linear(fused\_out)

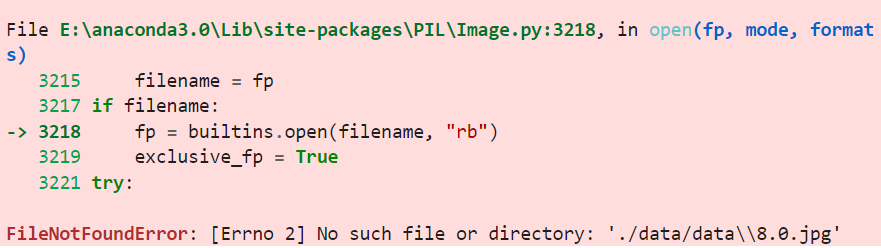
### 碰到的问题

#### 数据转换

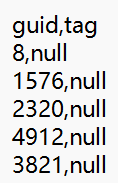
由于本次实验所提供的数据集当中图像、文本和标签是分开的，所以我的第一反应就是把图像、文本和标签合并到同一个csv文件当中，这样可以方便后续的操作。

思路其实也相当简单，就是根据train.txt当中的guid找到对应图像和文本的路径，再将标签映射为数字，将这三个元素组成一个字典存放在列表当中。最后通过pandas提供的to\_csv方法将对应的DataFrame转化为csv文件。

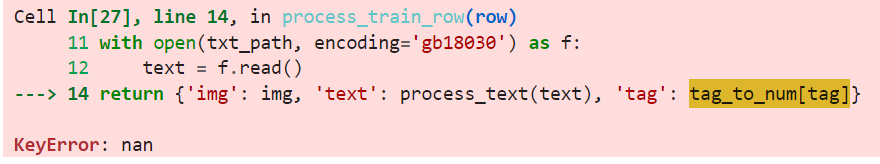
我一开始忘记给test数据进行转化了，只给train数据进行了转化，没有任何异常产生，代码就不放报告当中了。但是当我如法炮制对test数据进行转化的时候，出现了下面的报错：



去看一眼test数据的话：



发现拼凑路径的时候莫名其妙变成了浮点数，但是这在train数据的转换中并没有发生。虽然不明白这背后的原因是什么，但是解决方法很简单，取guid的时候加上int()取整即可。但是随即我又碰到了另外的一个问题：



原因在于train数据当中是给了label的，但是test数据当中的label是需要我们自己预测的。所以，不能用一个统一的函数process\_row来处理train数据和test数据，而是要分拆成precoess\_train\_row和process\_test\_row来分别处理。当然，并在一个函数里面也可以，不过就是要额外增加很多特判，最终的csv文件中还要多存储一列全是null的tag，浪费空间，所以我还是拆分成了两个函数。

#### LateFusion究竟是怎样的

如果我没有理解错论文中的意思的话，late fusion做的事情是将text和image两个类别概率向量通过某种算法综合起来。但是当我想要验证自己的想法，去网上看看是否有没有其他人的实现的时候，我找到了如下两个仓库：

https://github.com/IsaacRodgz/ConcatBERT



可以看到：

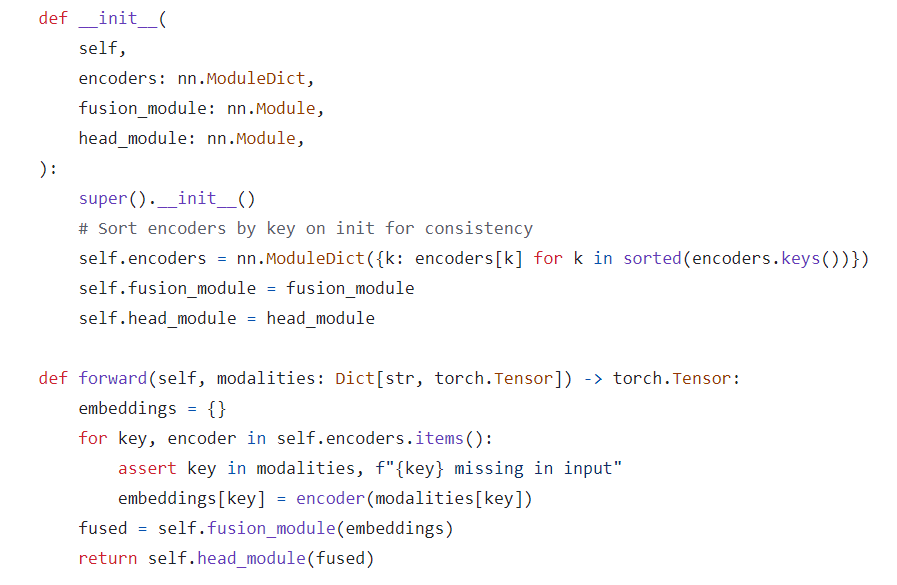
fig:

直接就把图像和文本的表示拼接上了，这个不就是early fusion吗？

而且在后续的具体训练过程中也和正常的一样，没有什么特殊处理：

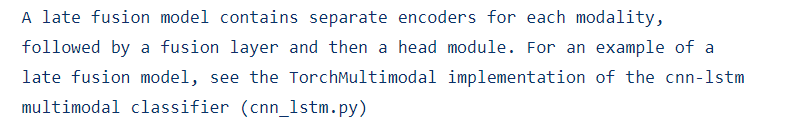


https://github.com/facebookresearch/multimodal/blob/main/torchmultimodal/models/late\_fusion.py

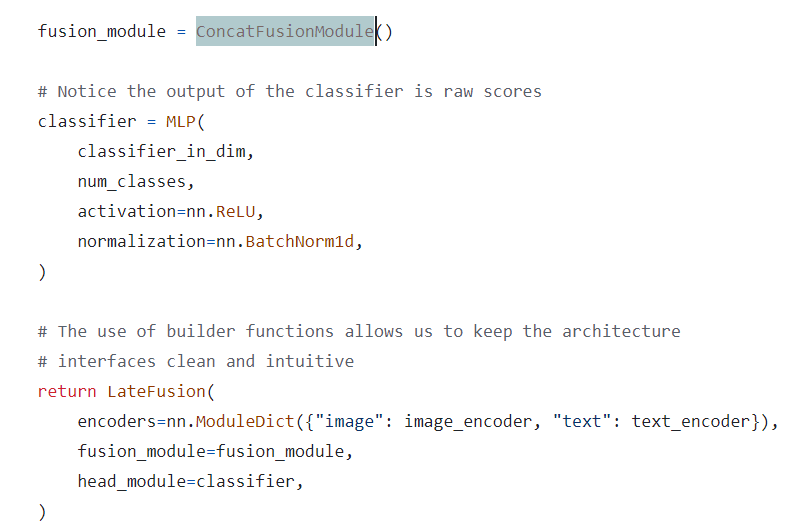


这里感觉也是early fusion。

注释中让我们去看看cnn\_lstm.py，有具体使用的例子：



那就去看看吧：



从变量名字推断出来，这确实就是个early fusion。

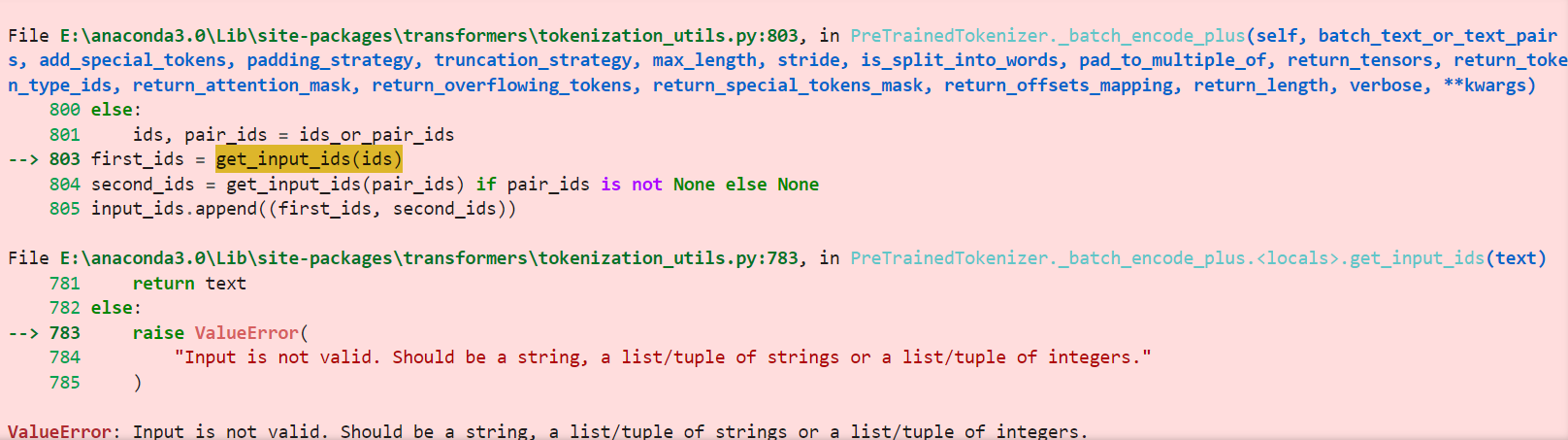
我挺摸不着头脑的，最后依然是根据论文中的描述实现了late fusion。

#### .csv文件Nonetype自动转换

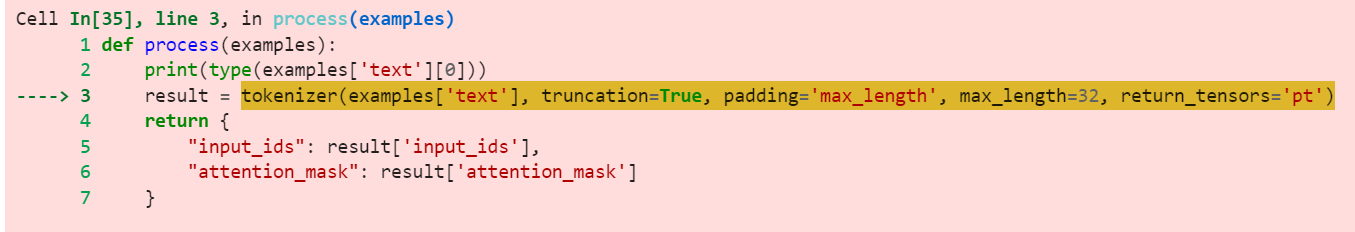
我在按照lab4当中的步骤，对数据集使用tokenizer进行数据预处理的时候碰到了一个匪夷所思的问题。代码是这样的：

def process(examples):  
 result = tokenizer(examples['text'], truncation=True, padding='max\_length', max\_length=32, return\_tensors='pt')  
 return {  
 "input\_ids": result['input\_ids'],  
 "attention\_mask": result['attention\_mask']  
 }  
   
tokenized\_ds = dataset.map(process, batched=True)  
tokenized\_ds

目的也很清楚，就是在dataset中加了input\_ids和attention\_mask两列，但是运行后出现了下面的报错：



问题是出在这里：

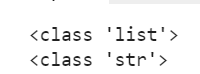


也就是说，examples['text']既不是string，也不是list/tuple of strings，还不是list/tuple of integers。

于是我就在process函数的第一行加入了：

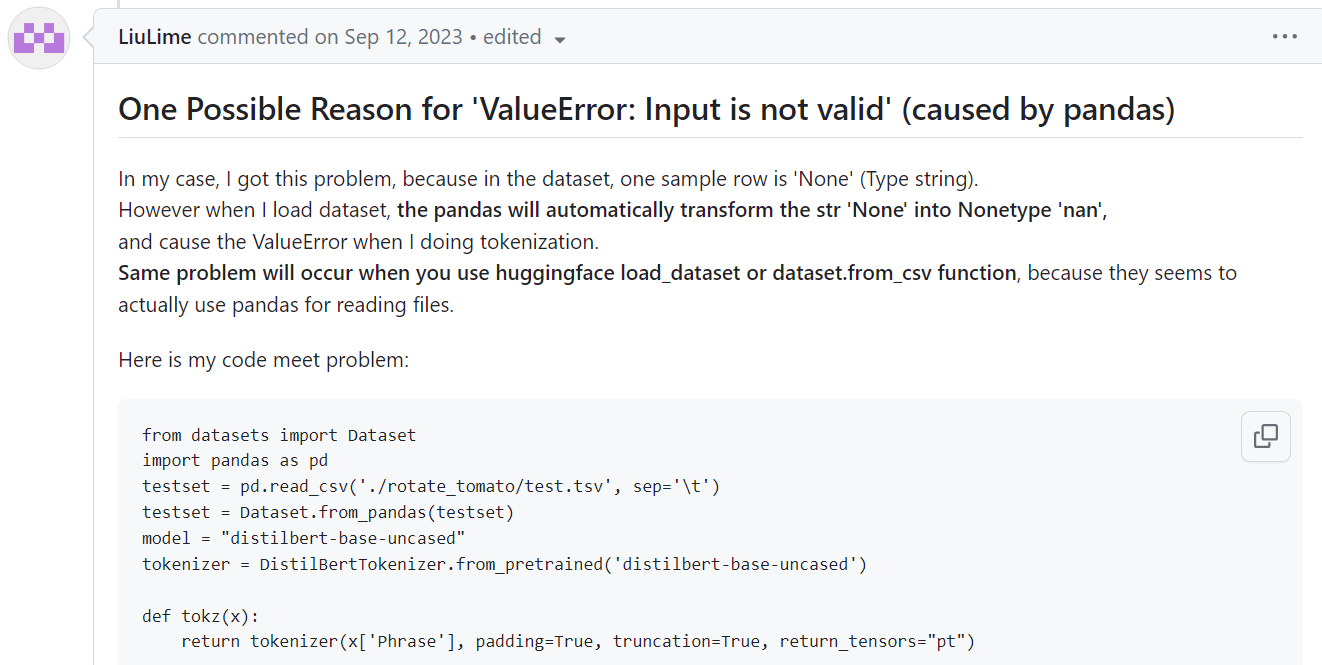
print(type(examples['text']))  
 print(type(examples['text'][0]))

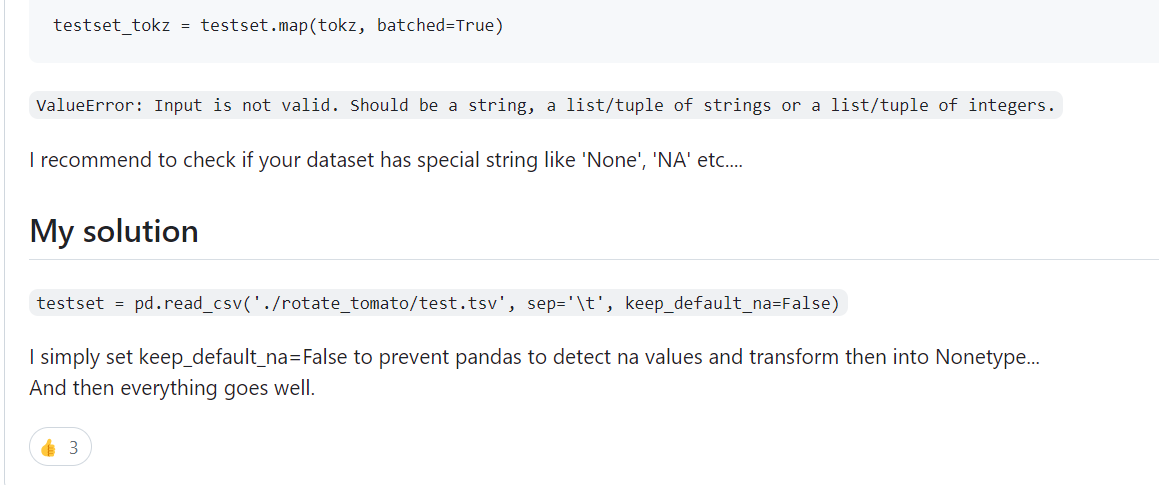
结果显示：



明明是啊。

然后我就开始走歪路了，开始怀疑是不是process函数写的不对，是不是example的结构与我想象的不一样。

在走了一大堆歪路最后依然没什么结果之后，我才找到了https://github.com/huggingface/transformers/issues/4986中的：



也就是说text那一列中可能包含字符串'None','NA'，直接被pandas转化成了Nonetype。我通过下面代码检查了一下：

import pandas as pd  
file\_path = "./data/image\_text\_tag\_train.csv"  
  
df = pd.read\_csv(file\_path)  
  
contains\_nan = df['text'].isnull().values.any()  
  
if contains\_nan:  
 print("包含NaN")  
else:  
 print("不包含NaN")

输出为：

fig:

但是我发现了一个问题，就是我不是用pandas来导入csv文件的，而是通过datasets.Dataset.from\_csv，所以我就又测试了下：

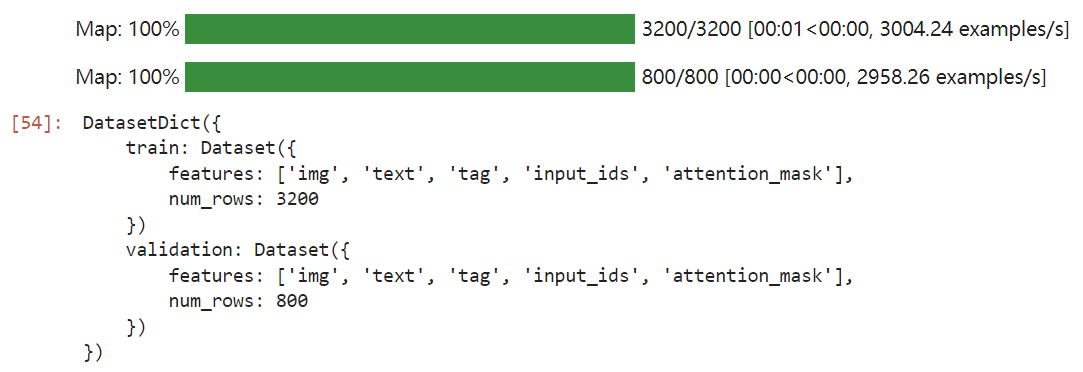
file\_path = "./data/image\_text\_tag\_train.csv"  
  
lst = Dataset.from\_csv(file\_path)  
  
contains\_nan = any(element is None for element in lst['text'])  
  
if contains\_nan:  
 print("包含NaN")  
else:  
 print("不包含NaN")

输出为：

fig:

这下终于是确定了问题所在了，解决方法就是在在from\_csv的时候添加一些额外参数：

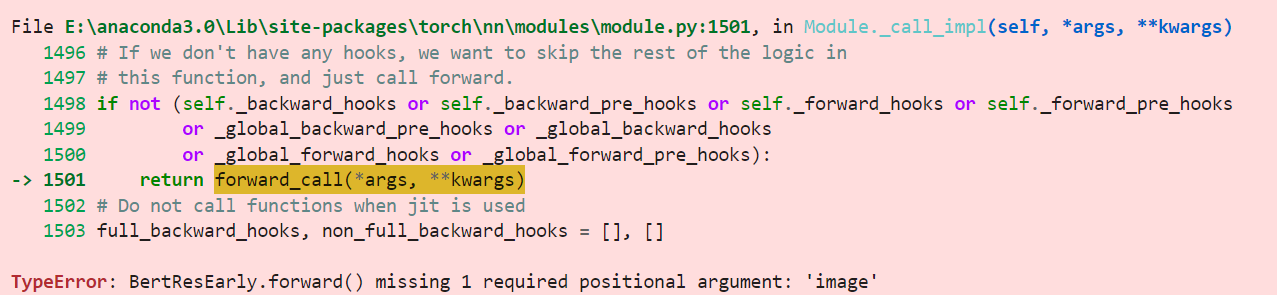
dataset\_train = Dataset.from\_csv("./data/image\_text\_tag\_train.csv", keep\_default\_na=False)



成功！

#### 初次尝试Trainer的时候出现的问题

和lab4一样，我并不打算自己写出train的函数，而是使用transformer库所提供的Trainer。但是我碰到了下面的问题：



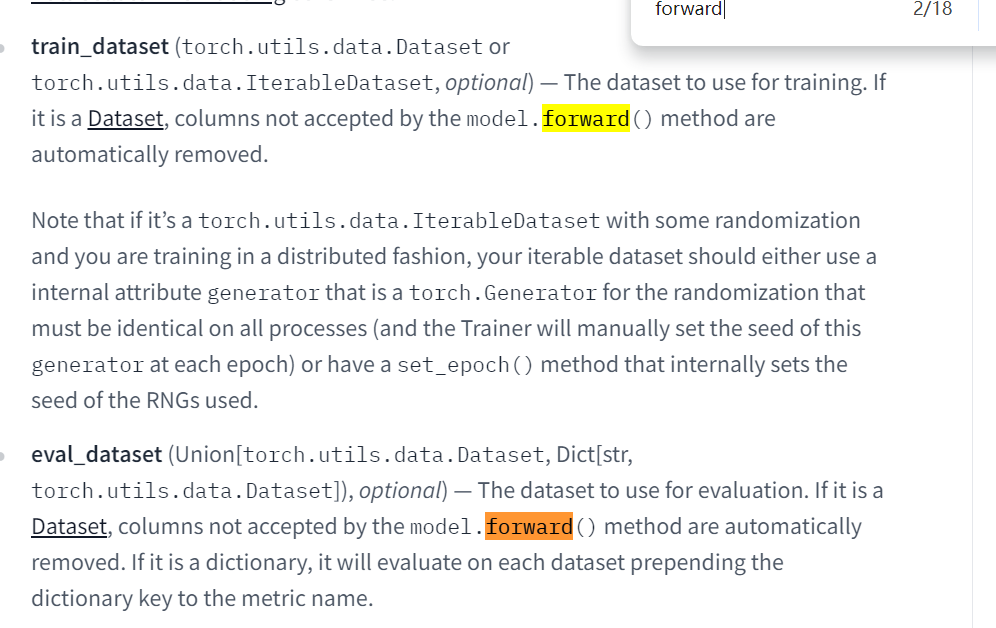
就是说forward中少了一个位置形参image。

我去看了眼模型的.py文件，发现：

fig:

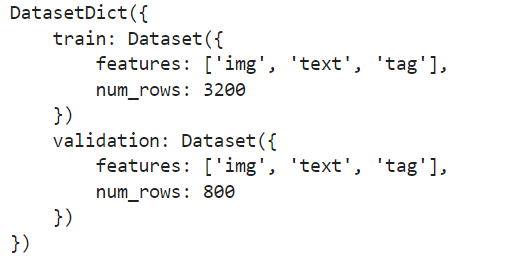
我接着去huggingface的文档中查找有没有解决方法。

我使用argument作为关键词进行网页内搜索，搜索结果实在太多，找不到有用信息。所以我又用forward作为关键词进行搜索：



可以看到出现了很有用的信息：Dataset中列名不在forward形参列表中的会被自动移除掉。

我看向自己先前的csv文件：



属性的名字是img，而不是forward形参列表中的image。

解决方法是比较简单的，要么修改dataset的列名，要么修改模型中的列名。我这里选择了前者，毕竟我的模型中forward的命名基本都是image。

但是：fig:

为什么还是不对？难道是我的思考方向有问题吗？为了验证，我更改了input\_ids的名字：

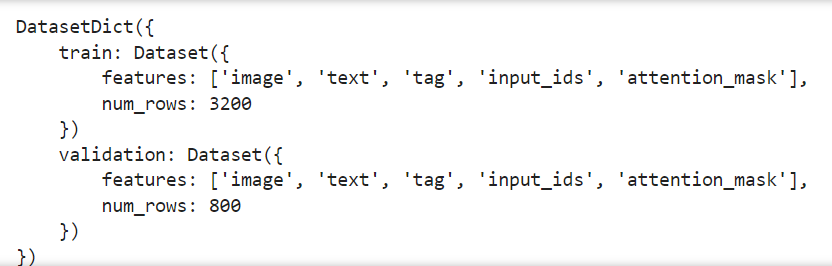
tokenized\_ds['train'] = tokenized\_ds['train'].rename\_column("input\_ids", "input")

报错变成了：

fig:

很好，看来我的思考方向并没有差错。那现在只可能是一个影响因素了，那就是forward形参列表的顺序也是相关的。我的dataset中image是第一个feature，但是形参列表中却是最后一个，所以我做了如下修改：

new\_feature\_order = ['text', 'tag', 'input\_ids', 'attention\_mask', 'image']   
tokenized\_ds['train'].set\_format(type='torch', columns=new\_feature\_order)  
tokenized\_ds['validation'].set\_format(type='torch', columns=new\_feature\_order)

然而：

没有任何的效果。

我后续查资料才发现set\_format方法实际上改变的是Dataset的内部格式，以支持更高效的访问。然而，它并不会直接改变数据的顺序，而是影响了数据加载和预处理的方式。所以，set\_format是无效的。

同时，我也找不到别的方法了。

后来经过漫长时间的搜索，我才想到这可能是Trainer中的一个参数data\_collator的问题。

我找到了两篇文章：https://blog.csdn.net/weixin42105164/article/details/121685283和https://blog.csdn.net/dongliuqi/article/details/114521240。



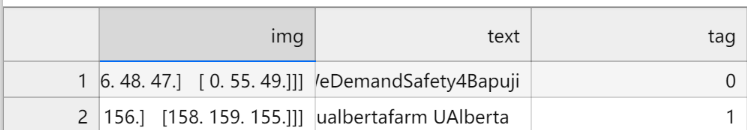
可以看到我目前的情况是基本符合这两个条件的，第二条是因为我正是forward出现的问题。

但是其实这也是一个乌龙，当我自己写了一个collate\_fn后并将其作为参数传递给Trainer之后，出现了新的报错：

Cell In[114], line 31, in collate\_fn(batch)  
 29 def collate\_fn(batch):  
 30 return {  
---> 31 'image': torch.stack([item['image'] for item in batch]),  
 32 'input\_ids': torch.stack([item['input\_ids'] for item in batch]),  
 33 'attention\_mask': torch.stack([item['attention\_mask'] for item in batch]),  
 34 'label': torch.tensor([item['label'] for item in batch])  
 35 }  
  
TypeError: expected Tensor as element 0 in argument 0, but got str

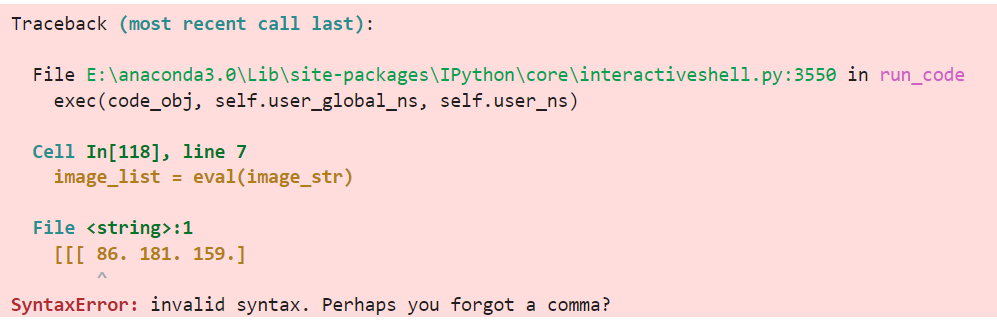
报错显示说：我在尝试调用torch.stack时，存在一个预期是Tensor的地方得到了一个字符串。这时候我才如梦初醒，原来是因为我这里的image是一个字符串，而不是张量，所以才导致无法作为参数传入的。

所以接下来就是要解决将字符串转化成张量的问题了。但是经过不断的尝试之后，发现是做不到这件事情的。因为我的csv是这样的：



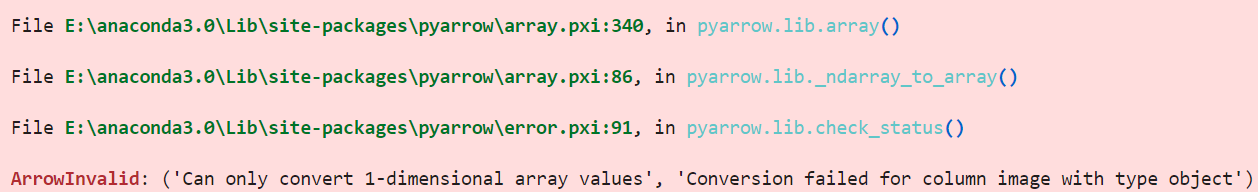
每一个image的张量都是存放在一个格子当中，元素之间并没有用逗号分隔开来。

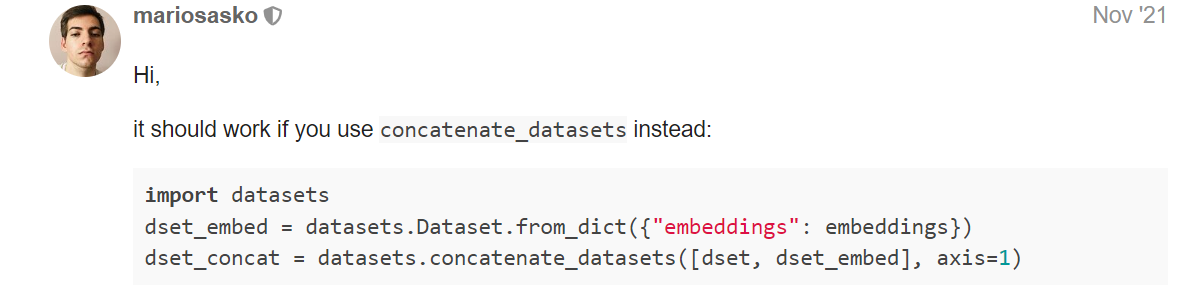
当我尝试着用eval将字符串转化为python对象的时候，会出现下面的报错：



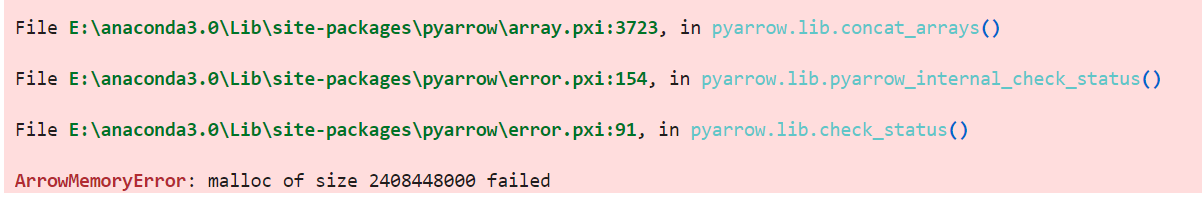
而我尝试了各种办法，包括设置分隔符、改存txt和使用自带csv库等，都不能够将张量带逗号的存入csv文件当中。

那没办法了，不能一次性将数据转换的过程在process\_data.ipynb完成了，只能在main.py里面再做了。而且这意味着我还必须存下guid来寻找图片。

当我想把代表图片的多维数组存入dataset中时，我遇到了下面的报错：

经过搜索，我在https://discuss.huggingface.co/t/add-new-column-to-a-dataset/12149/2这个链接中找到了解决方案：

理论上这个问题应该是解决了，但是由于我的电脑的内存的问题，会出现如下的报错：



为了解决这个问题，我之后应该是要开一个云主机了。

#### 本地连接云主机

我想要在云主机上使用jupyter notebook，因为jupyter notebook单元格运行的方式要远比.py文件调试起来方便。

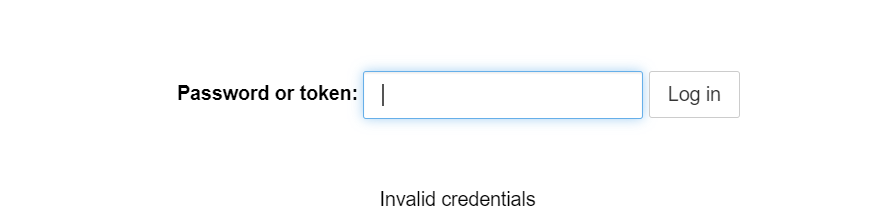
我在网上找到两篇教程：https://www.jianshu.com/p/4dd75f03180b和[在云服务器中搭建Jupyter Notebook环境\_华为云有个服务可以用它的服务器开notebook-CSDN博客](https://blog.csdn.net/qq_33765205/article/details/123311840)。

前面的都很顺利，但是当我在本地主机的浏览器上打算访问的时候：

后来发现是我没有打开端口的原因。打开的方式也很简单，在ucloud当中设置一个防火墙就行了：



终于能够进入登陆页面了，但是此时出现了另外的问题，我发现无论我输入什么都是invalid crediential:



我在这篇文章[服务器上配置jupyter，提示Invalid credentials如何解决\_jupyter invalid credentials-CSDN博客](https://blog.csdn.net/qq_43270444/article/details/134996130)中找到了答案：



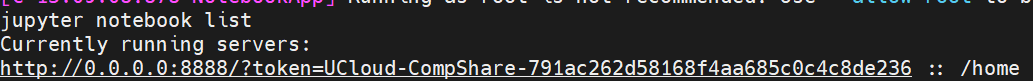
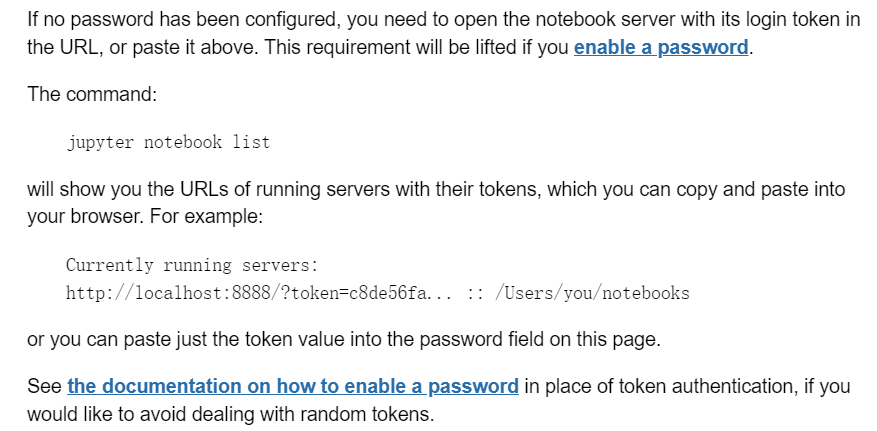
也就是说，只需要在生成密码的时候指定使用sha1即可：

from notebook.auth import passwd  
passwd(algorithm='sha1')

但是我发现问题还是没有得到解决。我输入密码出现的仍然还是invalid credential。

这时候，我才开始阅读起下面的提醒。原来这里的token指的并不是生成的sha1:xxxxx，我之前一直下意识地认为token就是sha1:xxxx。

我按照提示里面说的查看了下当前正在运行的服务器：



token=后面的内容才是token，于是我复制并输入后，终于成功登陆了。

#### 被迫放弃Trainer

但是当我转到云主机上进行操作之后，我发现是无法将所有的image都存放到dataset的image列当中的，内核总是会在运行成功前先一步挂掉。

而且即使有几次我转化了图片的张量表达方式，成功放进dataset中了。但是当我尝试着打印dataset['image'][0]，单元格运行了很长很长的时间才最终输出结果:

fig:

csv\_file\_path = "./data/text\_tag\_train.csv"  
df = pd.read\_csv(csv\_file\_path, keep\_default\_na=False)  
images = []  
for index, row in df.iterrows():  
 pil\_image = Image.open(f"./data/data/{row['guid']}.jpg").convert("RGB")  
 transform = transforms.ToTensor()  
 images.append(transform(pil\_image))  
dset\_embed = Dataset.from\_dict({"image": images})  
dataset\_train = Dataset.from\_csv("./data/text\_tag\_train.csv", keep\_default\_na=False)  
dataset\_train = datasets.concatenate\_datasets([dataset\_train, dset\_embed], axis=1)  
dataset\_train

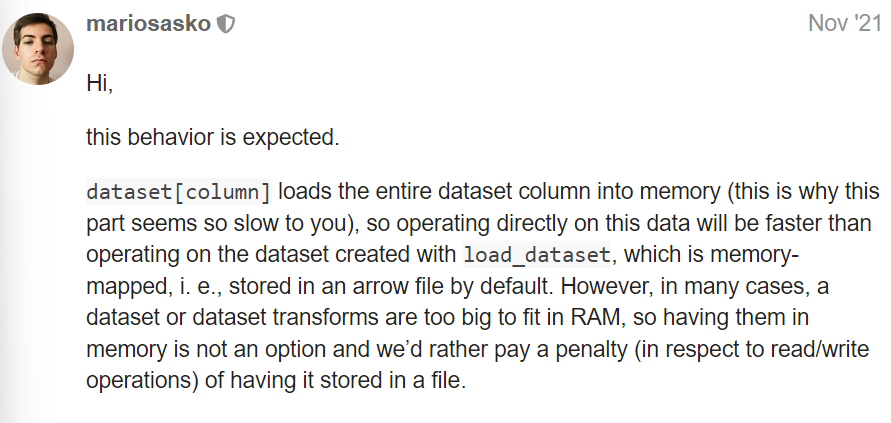
这种方式是根本无法放入dataset当中的。

csv\_file\_path = "./data/text\_tag\_train.csv"  
df = pd.read\_csv(csv\_file\_path, keep\_default\_na=False)  
images = []  
for index, row in df.iterrows():  
 img = Image.open(f"./data/data/{row['guid']}.jpg").resize((224, 224), Image.Resampling.LANCZOS)  
 img = np.asarray(img, dtype='float32')  
 images.append(img.transpose(2, 0, 1))  
dset\_embed = Dataset.from\_dict({"image": images})  
dataset\_train = Dataset.from\_csv("./data/text\_tag\_train.csv", keep\_default\_na=False)  
dataset\_train = datasets.concatenate\_datasets([dataset\_train, dset\_embed], axis=1)  
dataset\_train

这种方式是可以作为一列放入dataset当中的，但是存在我之前说的问题。

但是奇怪的是，如果是直接访问images列表的话，速度就非常快。我觉得原因可能在于dataset内部数据存储的方式有关。

https://discuss.huggingface.co/t/why-is-simply-accessing-dataset-features-so-slow/12041中提到：



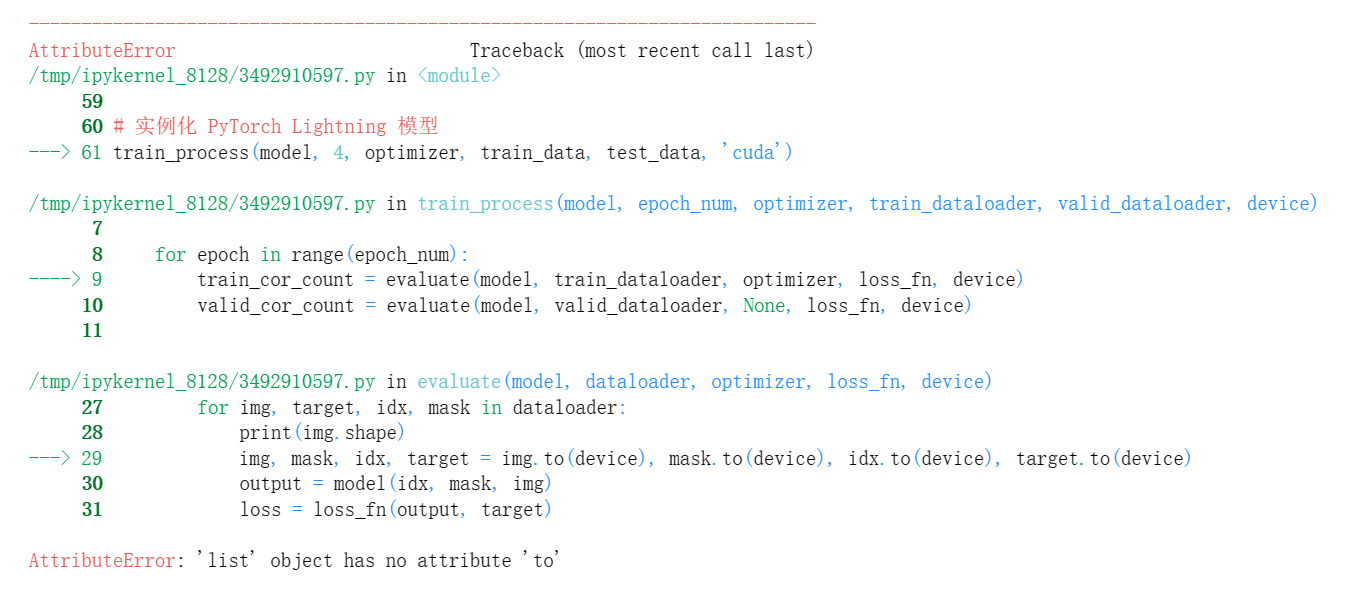
当我们去访问dataset中单独的一列或者是某列的元素的话，那么就会把整个一列都载入内存当中，所以才会很慢。由于已经直接加载入内存了，所以直接在它上面操作会比在dataset上操作要快。然而，在面对大规模数据集的时候（这是很普遍的情况），内存是很难装的下整个数据集的，所以才需要用到dataset（因为dataset的实现原理是基于内存映射的）。如果我们确保有足够的内存空间，就可以用以下的方式直接把数据集载入内存之中：

load\_dataset("squad", split="validation", keep\_in\_memory=True)

但是，我个人还是比较害怕如果我继续使用Trainer的话还会出现一些意料之外的情况，毕竟并不了解内部的具体实现，所以我最终还是决定放弃使用Trainer而是自己实现训练的过程。

#### AttributeError: 'list' object has no attribute 'to'

这里的问题出在当我想把图像，文本，input\_ids和attention\_mask通过to方法转移到gpu上面去的时候，代码出现了如下的错误：



也就是说，图像，文本，input\_ids和attention\_mask至少有一个list而并非是张量。

我在循环过程中加入了print语句，发现input\_ids和attention\_mask不是张量，它们是一个张量的列表，所以只要通过torch.stack()方法将它们转化成张量，再通过to方法转移到gpu上即可。

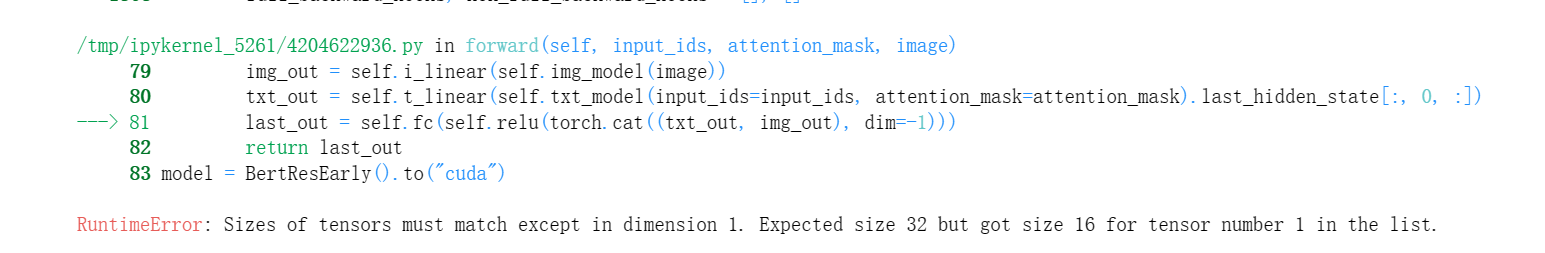
#### 放弃DatasetDict

前面说了我最终依然是打算手动实现训练过程，而不是调用封装好的接口。那就势必需要用到Dataloader，而我原本的数据都在DatasetDict当中，所以我通过下面的代码将DatasetDict转化成了train和test两个dataloader:

class CustomDataset(Dataset):  
 def \_\_init\_\_(self, ds, image\_data):  
 self.image\_data = image\_data  
 self.tags = ds['tag']  
 self.input\_ids\_data = ds['input\_ids']  
 self.attention\_mask\_data = ds['attention\_mask']  
   
 def \_\_len\_\_(self):  
 return len(self.image\_data)  
  
 def \_\_getitem\_\_(self, index):  
 image = self.image\_data[index]  
 tag = self.tags[index]  
 input\_ids = self.input\_ids\_data[index]  
 attention\_mask = self.attention\_mask\_data[index]  
 return image, tag, input\_ids, attention\_mask  
   
batch\_size = 16  
shuffle = True  
custom\_dataset = CustomDataset(ds['train'], images\_train)  
train\_data = DataLoader(custom\_dataset, batch\_size=batch\_size, shuffle=shuffle)  
custom\_dataset = CustomDataset(ds['test'], images\_test)  
test\_data = DataLoader(custom\_dataset, batch\_size=batch\_size, shuffle=shuffle)

但是很离谱的事情发生了，我发现一个batch中的每个属性所包含的元素数量是不一样的。

我这里的batch设定为16，那么每个属性所包含的元素数量就应该是16才对。但是我测试下来我的input\_ids和attention\_mask的元素数量是32个，这也到导致了我训练过程中的频频报错：



我百思不得其解。

我始终解决不了这个bug，只好放弃使用DatasetDict，回归最原始的列表了。

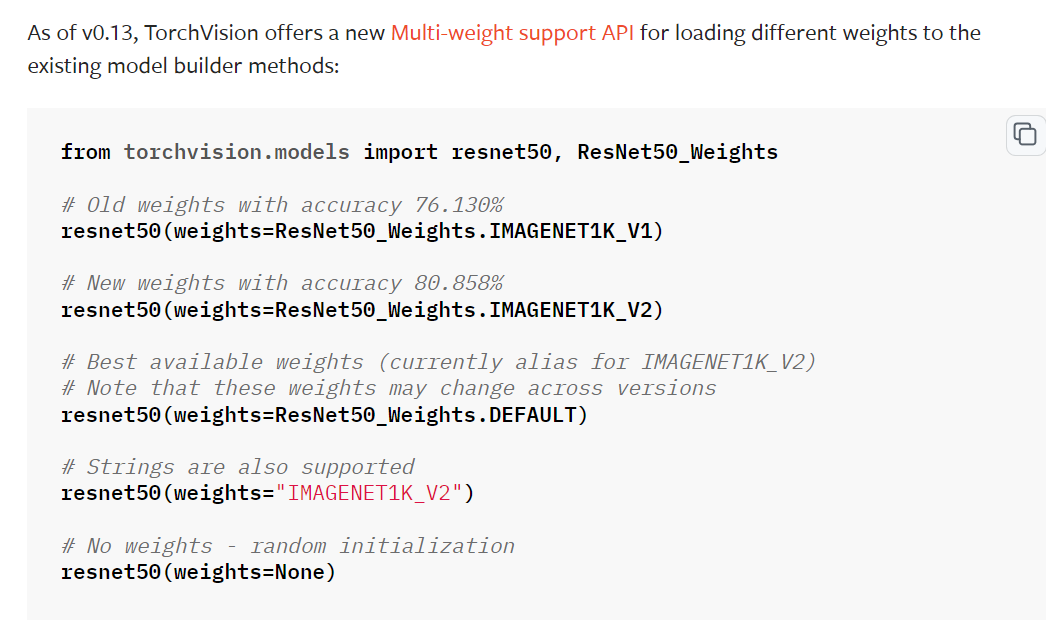
#### UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.

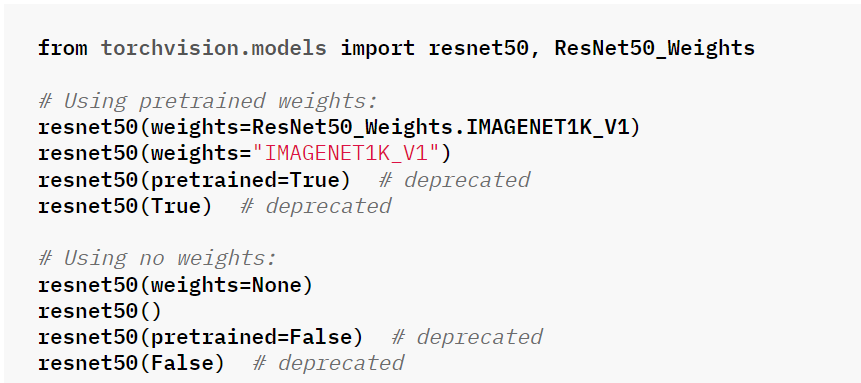
这个wanring是下面的代码产生的：

self.image\_model = torchvision.models.mobilenet\_v3\_large(pretrained=True)

意思就是说从 torchvision 0.13开始，加载预训练模型函数的参数从 pretrained = True改为了weights=预训练模型参数版本。

我们可以从[Models and pre-trained weights — Torchvision 0.16 documentation (pytorch.org)](https://pytorch.org/vision/stable/models.html)获得更加详细的信息：

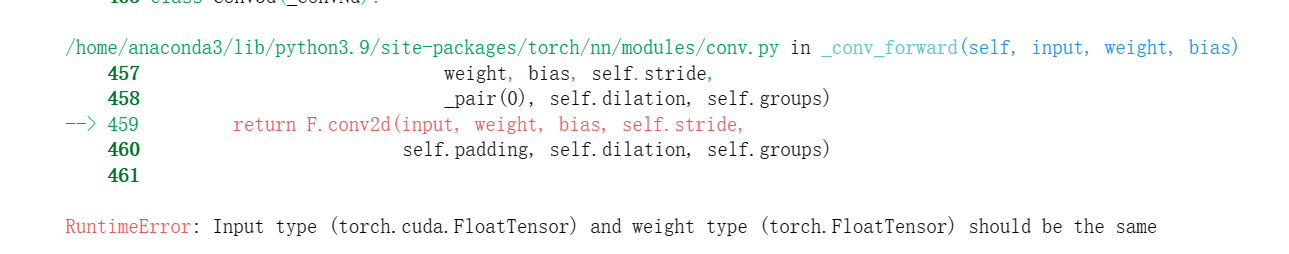




其实就是PyTorch推出了更加优秀的pretrain权重，在ImageNet效果比原先的权重更好。

不过其实这个并不会影响到本次实验，只是个warning，所以我也并没有进行修改，依然采用了pretrained=True的方式。

#### 加载最佳模型时出现的问题



这里是当我想要于测试集进行预测的时候出现的问题。

我想要通过：

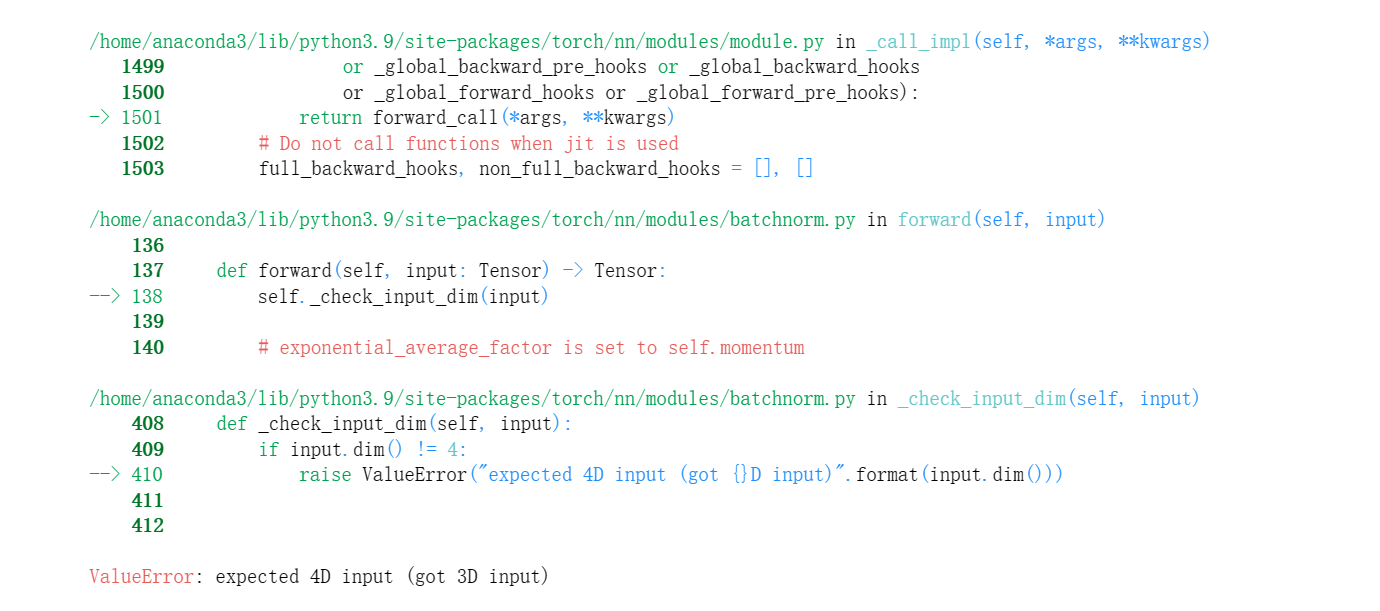
best\_model\_state\_dict = torch.load('best\_model.pth')  
model.load\_state\_dict(best\_model\_state\_dict)

加载最佳模型并进行测试，结果出现了报错。

不过这里的报错信息很明白了：输入类型 torch.cuda.FloatTensor和权重类型torch.FloatTensor应该是一样的。而我的模型最初训练的时候是放到GPU上进行训练的，所以这里加载最佳模型的时候也应该放到gpu上，所以只要在下面再加一行：

model = model.to("cuda")

#### 预测时出现的问题



这个错误的意思是在说torchvision中的ResNet模型期望一个四维的输入张量，形状应为 (batch\_size, channels, height, width)，而我提供的是一个三维张量 (channels, height, width)。

这个问题很好理解，因为测试的时候就是一张张图片，没有batch这个说法了。解决方法相当简单，只需要在将图像张量传递给ResNet模型之前，为其添加一个额外的维度即可，可以使用unsqueeze方法来实现：

for guid in guid\_list:  
 # ...（之前的代码保持不变）  
 image\_tensor = torch.Tensor(image\_tensor).unsqueeze(0).to(device)  
 # ...  
 predictions = model(input\_ids.to(device), attention\_mask.to(device), image\_tensor)  
 # ...

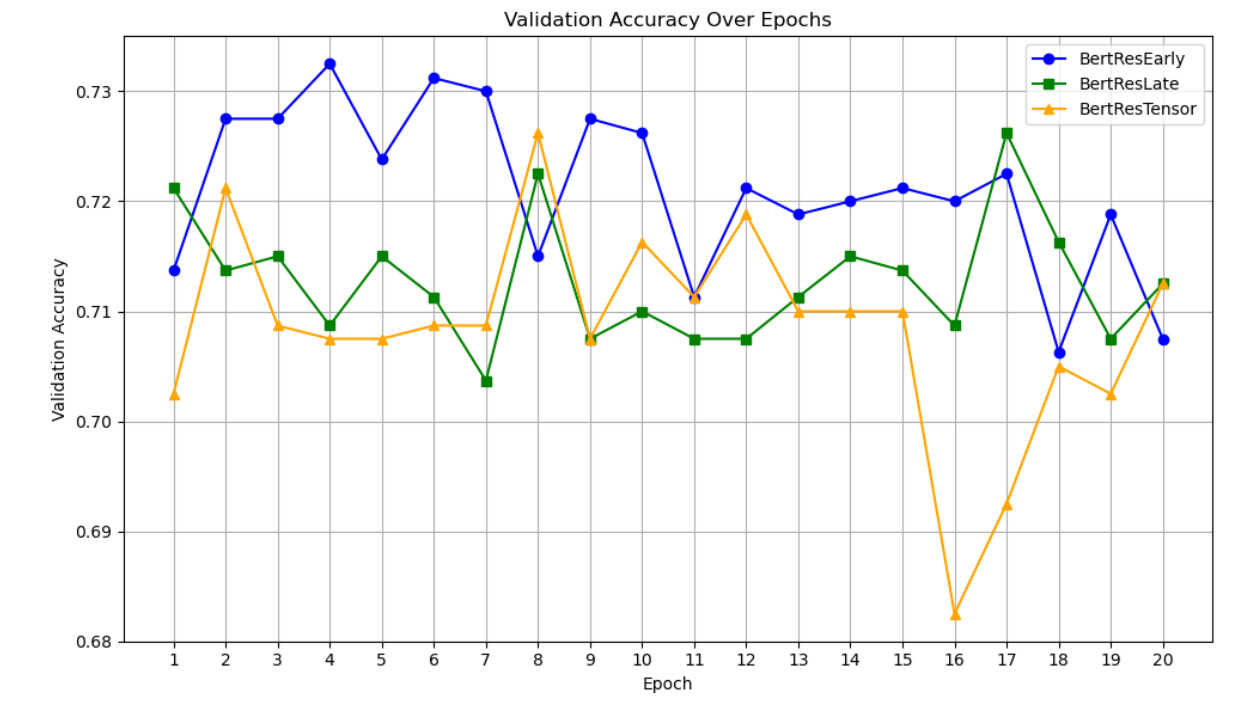
unsqueeze(0)，就是在最前面插入一个维度，而batch\_size这个维度正是在第一个。

## 实验结果

评估模型的时候，我记录下了每个模型每个epoch的验证集准确率，在20个epoch结束后我从20个模型checkpoint中选出了验证集准确率最高的作为最佳模型。

### Bert+ResNet152

BertResEarly  
epoch: 1, loss: 0.0337, train acc: 0.9741, valid acc: 0.7137  
epoch: 2, loss: 0.0406, train acc: 0.9912, valid acc: 0.7275  
epoch: 3, loss: 0.2611, train acc: 0.9919, valid acc: 0.7275  
epoch: 4, loss: 0.0031, train acc: 0.9922, valid acc: 0.7325  
epoch: 5, loss: 0.0104, train acc: 0.9928, valid acc: 0.7238  
epoch: 6, loss: 0.0022, train acc: 0.9928, valid acc: 0.7312  
epoch: 7, loss: 0.0023, train acc: 0.9931, valid acc: 0.7300  
epoch: 8, loss: 0.0874, train acc: 0.9925, valid acc: 0.7150  
epoch: 9, loss: 0.0022, train acc: 0.9925, valid acc: 0.7275  
epoch: 10, loss: 0.0012, train acc: 0.9925, valid acc: 0.7262  
epoch: 11, loss: 0.1456, train acc: 0.9925, valid acc: 0.7113  
epoch: 12, loss: 0.0035, train acc: 0.9928, valid acc: 0.7212  
epoch: 13, loss: 0.0014, train acc: 0.9931, valid acc: 0.7188  
epoch: 14, loss: 0.0012, train acc: 0.9931, valid acc: 0.7200  
epoch: 15, loss: 0.1695, train acc: 0.9931, valid acc: 0.7212  
epoch: 16, loss: 0.0006, train acc: 0.9925, valid acc: 0.7200  
epoch: 17, loss: 0.0009, train acc: 0.9934, valid acc: 0.7225  
epoch: 18, loss: 0.0004, train acc: 0.9934, valid acc: 0.7063  
epoch: 19, loss: 0.0017, train acc: 0.9931, valid acc: 0.7188  
epoch: 20, loss: 0.0765, train acc: 0.9872, valid acc: 0.7075  
BertResLate  
epoch: 1, loss: 0.1953, train acc: 0.8728, valid acc: 0.7212  
epoch: 2, loss: 0.1873, train acc: 0.9653, valid acc: 0.7137  
epoch: 3, loss: 0.0251, train acc: 0.9881, valid acc: 0.7150  
epoch: 4, loss: 0.0139, train acc: 0.9916, valid acc: 0.7087  
epoch: 5, loss: 0.0063, train acc: 0.9928, valid acc: 0.7150  
epoch: 6, loss: 0.0151, train acc: 0.9928, valid acc: 0.7113  
epoch: 7, loss: 0.0043, train acc: 0.9925, valid acc: 0.7037  
epoch: 8, loss: 0.0028, train acc: 0.9922, valid acc: 0.7225  
epoch: 9, loss: 0.0023, train acc: 0.9928, valid acc: 0.7075  
epoch: 10, loss: 0.0625, train acc: 0.9931, valid acc: 0.7100  
epoch: 11, loss: 0.0039, train acc: 0.9922, valid acc: 0.7075  
epoch: 12, loss: 0.0019, train acc: 0.9922, valid acc: 0.7075  
epoch: 13, loss: 0.0029, train acc: 0.9919, valid acc: 0.7113  
epoch: 14, loss: 0.0015, train acc: 0.9919, valid acc: 0.7150  
epoch: 15, loss: 0.0007, train acc: 0.9916, valid acc: 0.7137  
epoch: 16, loss: 0.0917, train acc: 0.9925, valid acc: 0.7087  
epoch: 17, loss: 0.0014, train acc: 0.9925, valid acc: 0.7262  
epoch: 18, loss: 0.0813, train acc: 0.9931, valid acc: 0.7163  
epoch: 19, loss: 0.0013, train acc: 0.9944, valid acc: 0.7075  
epoch: 20, loss: 0.0004, train acc: 0.9928, valid acc: 0.7125  
BertResTensor  
epoch: 1, loss: 0.5507, train acc: 0.8969, valid acc: 0.7025  
epoch: 2, loss: 0.0291, train acc: 0.9778, valid acc: 0.7212  
epoch: 3, loss: 0.0337, train acc: 0.9900, valid acc: 0.7087  
epoch: 4, loss: 0.0291, train acc: 0.9928, valid acc: 0.7075  
epoch: 5, loss: 0.0084, train acc: 0.9938, valid acc: 0.7075  
epoch: 6, loss: 0.0011, train acc: 0.9928, valid acc: 0.7087  
epoch: 7, loss: 0.0143, train acc: 0.9931, valid acc: 0.7087  
epoch: 8, loss: 0.1401, train acc: 0.9941, valid acc: 0.7262  
epoch: 9, loss: 0.0004, train acc: 0.9938, valid acc: 0.7075  
epoch: 10, loss: 0.0008, train acc: 0.9931, valid acc: 0.7163  
epoch: 11, loss: 0.0001, train acc: 0.9931, valid acc: 0.7113  
epoch: 12, loss: 0.0759, train acc: 0.9931, valid acc: 0.7188  
epoch: 13, loss: 0.0001, train acc: 0.9922, valid acc: 0.7100  
epoch: 14, loss: 0.0003, train acc: 0.9928, valid acc: 0.7100  
epoch: 15, loss: 0.0007, train acc: 0.9928, valid acc: 0.7100  
epoch: 16, loss: 0.0323, train acc: 0.9853, valid acc: 0.6825  
epoch: 17, loss: 0.0084, train acc: 0.9650, valid acc: 0.6925  
epoch: 18, loss: 0.0487, train acc: 0.9888, valid acc: 0.7050  
epoch: 19, loss: 0.0005, train acc: 0.9919, valid acc: 0.7025  
epoch: 20, loss: 0.0001, train acc: 0.9934, valid acc: 0.7125





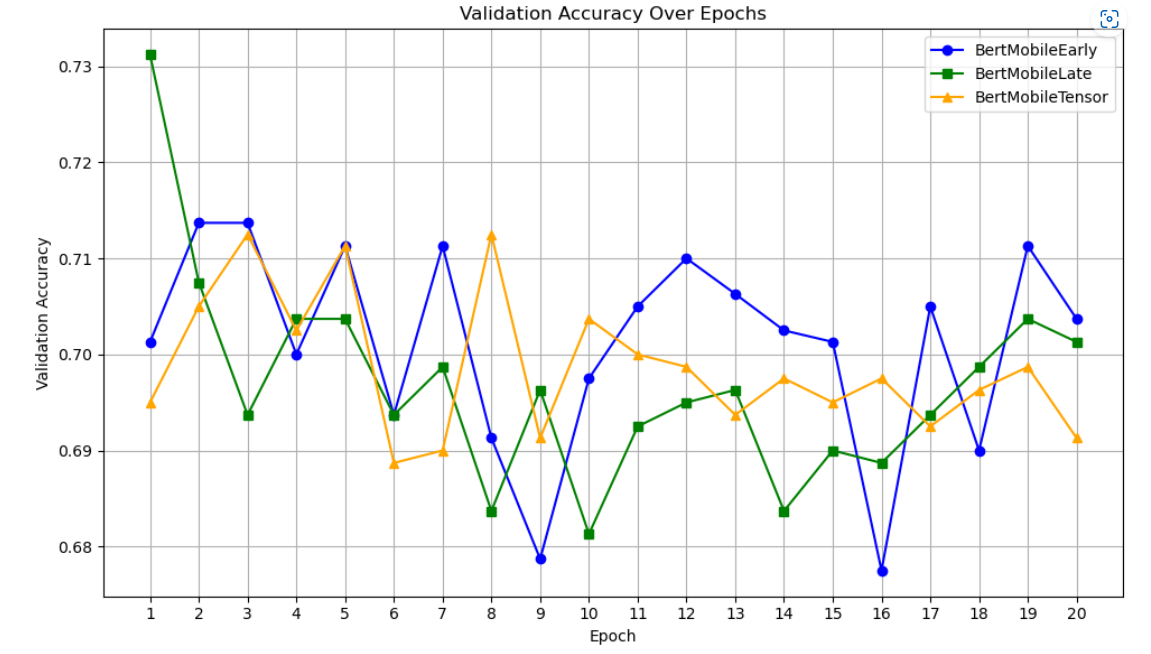
BertResEarly在epoch 4达到了最大的验证集准确率，为0.7325

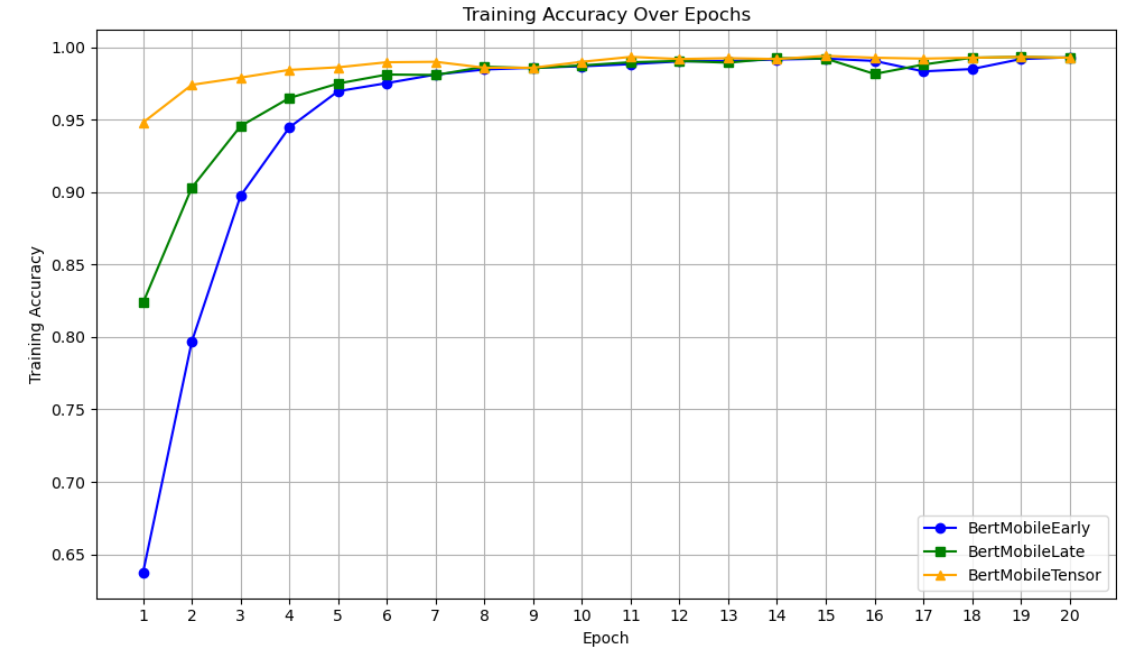
BertResLate在epoch 17达到了最大的验证集准确率，为0.7262

BertResTensor在epoch 8达到了最大的验证集准确率，为0.7262

### Bert+MobileNetV3

BertMobileEarly  
epoch: 1, loss: 0.8617, train acc: 0.6375, valid acc: 0.7013  
epoch: 2, loss: 0.5619, train acc: 0.7969, valid acc: 0.7137  
epoch: 3, loss: 0.2712, train acc: 0.8975, valid acc: 0.7137  
epoch: 4, loss: 0.1040, train acc: 0.9447, valid acc: 0.7000  
epoch: 5, loss: 0.1244, train acc: 0.9697, valid acc: 0.7113  
epoch: 6, loss: 0.0258, train acc: 0.9753, valid acc: 0.6937  
epoch: 7, loss: 0.0470, train acc: 0.9812, valid acc: 0.7113  
epoch: 8, loss: 0.0094, train acc: 0.9847, valid acc: 0.6913  
epoch: 9, loss: 0.0103, train acc: 0.9859, valid acc: 0.6787  
epoch: 10, loss: 0.0109, train acc: 0.9869, valid acc: 0.6975  
epoch: 11, loss: 0.0051, train acc: 0.9884, valid acc: 0.7050  
epoch: 12, loss: 0.0034, train acc: 0.9903, valid acc: 0.7100  
epoch: 13, loss: 0.0052, train acc: 0.9906, valid acc: 0.7063  
epoch: 14, loss: 0.0151, train acc: 0.9916, valid acc: 0.7025  
epoch: 15, loss: 0.0027, train acc: 0.9922, valid acc: 0.7013  
epoch: 16, loss: 0.0105, train acc: 0.9906, valid acc: 0.6775  
epoch: 17, loss: 0.3740, train acc: 0.9834, valid acc: 0.7050  
epoch: 18, loss: 0.0021, train acc: 0.9850, valid acc: 0.6900  
epoch: 19, loss: 0.0061, train acc: 0.9919, valid acc: 0.7113  
epoch: 20, loss: 0.0014, train acc: 0.9931, valid acc: 0.7037  
BertMobileLate  
epoch: 1, loss: 0.6094, train acc: 0.8237, valid acc: 0.7312  
epoch: 2, loss: 0.0854, train acc: 0.9031, valid acc: 0.7075  
epoch: 3, loss: 0.1223, train acc: 0.9456, valid acc: 0.6937  
epoch: 4, loss: 0.1896, train acc: 0.9650, valid acc: 0.7037  
epoch: 5, loss: 0.0299, train acc: 0.9750, valid acc: 0.7037  
epoch: 6, loss: 0.1681, train acc: 0.9812, valid acc: 0.6937  
epoch: 7, loss: 0.0204, train acc: 0.9809, valid acc: 0.6987  
epoch: 8, loss: 0.0097, train acc: 0.9866, valid acc: 0.6837  
epoch: 9, loss: 0.0188, train acc: 0.9856, valid acc: 0.6963  
epoch: 10, loss: 0.0100, train acc: 0.9875, valid acc: 0.6813  
epoch: 11, loss: 0.0142, train acc: 0.9897, valid acc: 0.6925  
epoch: 12, loss: 0.0082, train acc: 0.9906, valid acc: 0.6950  
epoch: 13, loss: 0.0203, train acc: 0.9894, valid acc: 0.6963  
epoch: 14, loss: 0.0045, train acc: 0.9925, valid acc: 0.6837  
epoch: 15, loss: 0.0051, train acc: 0.9922, valid acc: 0.6900  
epoch: 16, loss: 0.0040, train acc: 0.9816, valid acc: 0.6887  
epoch: 17, loss: 0.0018, train acc: 0.9881, valid acc: 0.6937  
epoch: 18, loss: 0.0027, train acc: 0.9928, valid acc: 0.6987  
epoch: 19, loss: 0.0621, train acc: 0.9934, valid acc: 0.7037  
epoch: 20, loss: 0.0158, train acc: 0.9928, valid acc: 0.7013  
BertMobileTensor  
epoch: 1, loss: 0.0594, train acc: 0.9481, valid acc: 0.6950  
epoch: 2, loss: 0.0110, train acc: 0.9741, valid acc: 0.7050  
epoch: 3, loss: 0.0062, train acc: 0.9791, valid acc: 0.7125  
epoch: 4, loss: 0.1686, train acc: 0.9844, valid acc: 0.7025  
epoch: 5, loss: 0.0030, train acc: 0.9862, valid acc: 0.7113  
epoch: 6, loss: 0.0005, train acc: 0.9897, valid acc: 0.6887  
epoch: 7, loss: 0.0004, train acc: 0.9900, valid acc: 0.6900  
epoch: 8, loss: 0.4468, train acc: 0.9859, valid acc: 0.7125  
epoch: 9, loss: 0.0016, train acc: 0.9859, valid acc: 0.6913  
epoch: 10, loss: 0.0920, train acc: 0.9900, valid acc: 0.7037  
epoch: 11, loss: 0.1076, train acc: 0.9934, valid acc: 0.7000  
epoch: 12, loss: 0.0052, train acc: 0.9919, valid acc: 0.6987  
epoch: 13, loss: 0.0002, train acc: 0.9925, valid acc: 0.6937  
epoch: 14, loss: 0.0001, train acc: 0.9919, valid acc: 0.6975  
epoch: 15, loss: 0.0004, train acc: 0.9941, valid acc: 0.6950  
epoch: 16, loss: 0.0000, train acc: 0.9928, valid acc: 0.6975  
epoch: 17, loss: 0.0003, train acc: 0.9922, valid acc: 0.6925  
epoch: 18, loss: 0.0002, train acc: 0.9925, valid acc: 0.6963  
epoch: 19, loss: 0.0001, train acc: 0.9925, valid acc: 0.6987  
epoch: 20, loss: 0.0766, train acc: 0.9925, valid acc: 0.6913





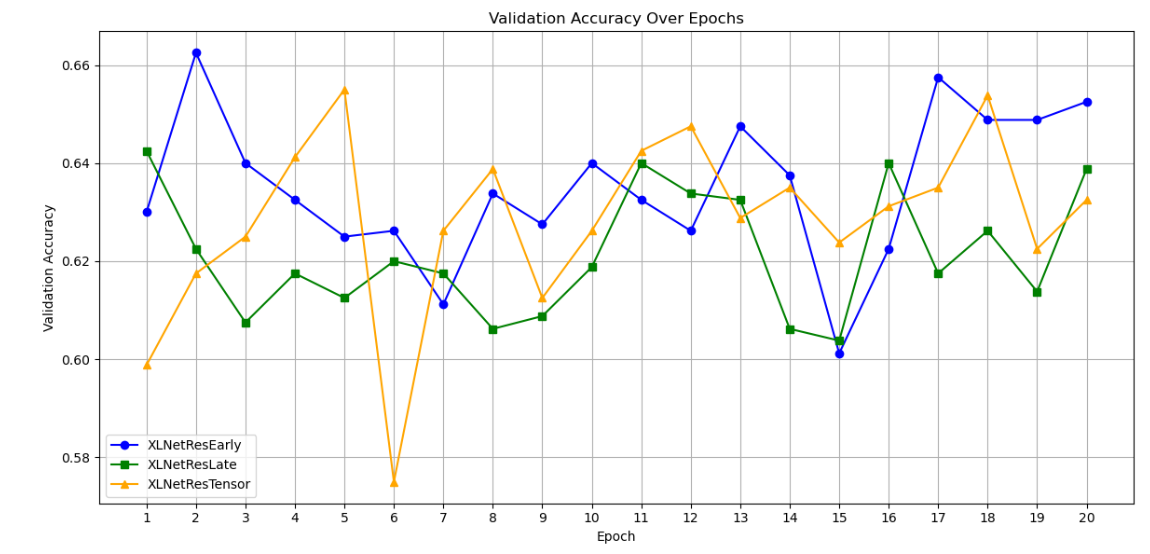
BertMobileEarly在epoch 2/3达到了最大的验证集准确率，为0.7137

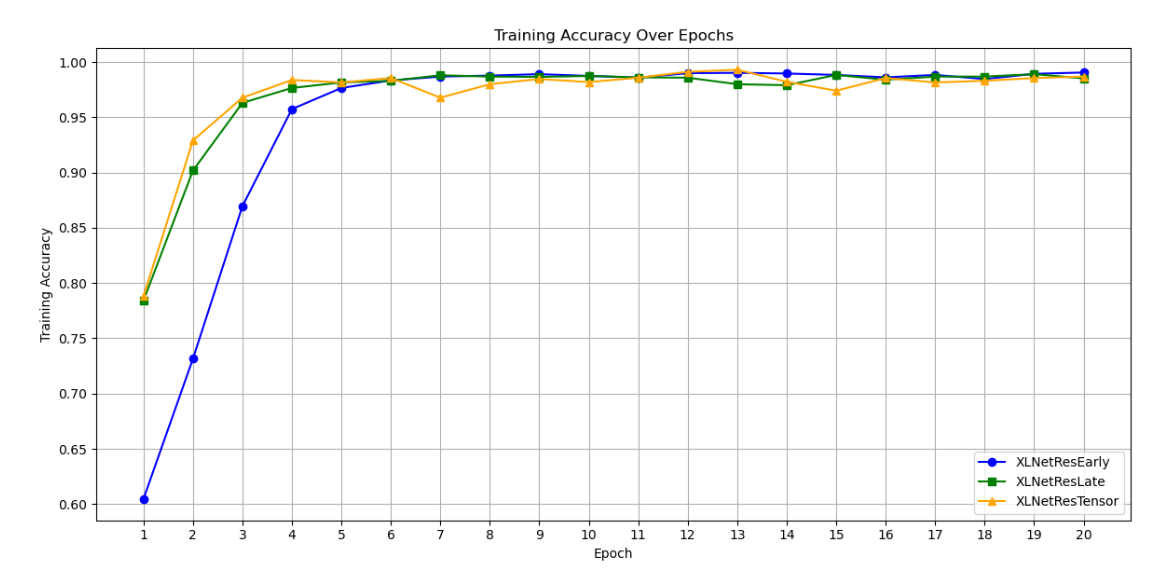
BertMobileLate在epoch 1达到了最大的验证集准确率，为0.7312

BertMobileTensor在epoch 3/8达到了最大的验证集准确率，为0.7125

### XLNet+ResNet152

XLNetResEarly  
epoch: 1, loss: 0.5573, train acc: 0.6047, valid acc: 0.6300  
epoch: 2, loss: 0.4097, train acc: 0.7316, valid acc: 0.6625  
epoch: 3, loss: 0.2849, train acc: 0.8697, valid acc: 0.6400  
epoch: 4, loss: 0.0755, train acc: 0.9575, valid acc: 0.6325  
epoch: 5, loss: 0.1188, train acc: 0.9766, valid acc: 0.6250  
epoch: 6, loss: 0.0168, train acc: 0.9834, valid acc: 0.6262  
epoch: 7, loss: 0.0386, train acc: 0.9869, valid acc: 0.6112  
epoch: 8, loss: 0.0109, train acc: 0.9878, valid acc: 0.6338  
epoch: 9, loss: 0.0212, train acc: 0.9891, valid acc: 0.6275  
epoch: 10, loss: 0.0360, train acc: 0.9875, valid acc: 0.6400  
epoch: 11, loss: 0.0096, train acc: 0.9859, valid acc: 0.6325  
epoch: 12, loss: 0.0100, train acc: 0.9900, valid acc: 0.6262  
epoch: 13, loss: 0.0022, train acc: 0.9903, valid acc: 0.6475  
epoch: 14, loss: 0.1027, train acc: 0.9897, valid acc: 0.6375  
epoch: 15, loss: 0.0047, train acc: 0.9884, valid acc: 0.6012  
epoch: 16, loss: 0.0298, train acc: 0.9862, valid acc: 0.6225  
epoch: 17, loss: 0.0141, train acc: 0.9884, valid acc: 0.6575  
epoch: 18, loss: 0.0276, train acc: 0.9847, valid acc: 0.6488  
epoch: 19, loss: 0.0852, train acc: 0.9894, valid acc: 0.6488  
epoch: 20, loss: 0.0001, train acc: 0.9906, valid acc: 0.6525  
XLNetResLate  
epoch: 1, loss: 0.4339, train acc: 0.7841, valid acc: 0.6425  
epoch: 2, loss: 0.3314, train acc: 0.9019, valid acc: 0.6225  
epoch: 3, loss: 0.1563, train acc: 0.9631, valid acc: 0.6075  
epoch: 4, loss: 0.1310, train acc: 0.9766, valid acc: 0.6175  
epoch: 5, loss: 0.0704, train acc: 0.9816, valid acc: 0.6125  
epoch: 6, loss: 0.0992, train acc: 0.9831, valid acc: 0.6200  
epoch: 7, loss: 0.1661, train acc: 0.9881, valid acc: 0.6175  
epoch: 8, loss: 0.0144, train acc: 0.9869, valid acc: 0.6062  
epoch: 9, loss: 0.0022, train acc: 0.9866, valid acc: 0.6088  
epoch: 10, loss: 0.0024, train acc: 0.9875, valid acc: 0.6188  
epoch: 11, loss: 0.0715, train acc: 0.9862, valid acc: 0.6400  
epoch: 12, loss: 0.0265, train acc: 0.9859, valid acc: 0.6338  
epoch: 13, loss: 0.0121, train acc: 0.9800, valid acc: 0.6325  
epoch: 14, loss: 0.0070, train acc: 0.9791, valid acc: 0.6062  
epoch: 15, loss: 0.0039, train acc: 0.9884, valid acc: 0.6038  
epoch: 16, loss: 0.0011, train acc: 0.9841, valid acc: 0.6400  
epoch: 17, loss: 0.0642, train acc: 0.9869, valid acc: 0.6175  
epoch: 18, loss: 0.0620, train acc: 0.9869, valid acc: 0.6262  
epoch: 19, loss: 0.0676, train acc: 0.9891, valid acc: 0.6138  
epoch: 20, loss: 0.0016, train acc: 0.9853, valid acc: 0.6388  
XLNetResTensor  
epoch: 1, loss: 0.3572, train acc: 0.7881, valid acc: 0.5988  
epoch: 2, loss: 0.2919, train acc: 0.9291, valid acc: 0.6175  
epoch: 3, loss: 0.0266, train acc: 0.9678, valid acc: 0.6250  
epoch: 4, loss: 0.0278, train acc: 0.9838, valid acc: 0.6412  
epoch: 5, loss: 0.0579, train acc: 0.9816, valid acc: 0.6550  
epoch: 6, loss: 0.0056, train acc: 0.9856, valid acc: 0.5750  
epoch: 7, loss: 0.0573, train acc: 0.9678, valid acc: 0.6262  
epoch: 8, loss: 0.0025, train acc: 0.9800, valid acc: 0.6388  
epoch: 9, loss: 0.1120, train acc: 0.9847, valid acc: 0.6125  
epoch: 10, loss: 0.0149, train acc: 0.9819, valid acc: 0.6262  
epoch: 11, loss: 0.0254, train acc: 0.9859, valid acc: 0.6425  
epoch: 12, loss: 0.0003, train acc: 0.9912, valid acc: 0.6475  
epoch: 13, loss: 0.0000, train acc: 0.9931, valid acc: 0.6288  
epoch: 14, loss: 0.0381, train acc: 0.9822, valid acc: 0.6350  
epoch: 15, loss: 0.0003, train acc: 0.9741, valid acc: 0.6238  
epoch: 16, loss: 0.0008, train acc: 0.9856, valid acc: 0.6312  
epoch: 17, loss: 0.0860, train acc: 0.9816, valid acc: 0.6350  
epoch: 18, loss: 0.0274, train acc: 0.9831, valid acc: 0.6538  
epoch: 19, loss: 0.0139, train acc: 0.9856, valid acc: 0.6225  
epoch: 20, loss: 0.0003, train acc: 0.9869, valid acc: 0.6325





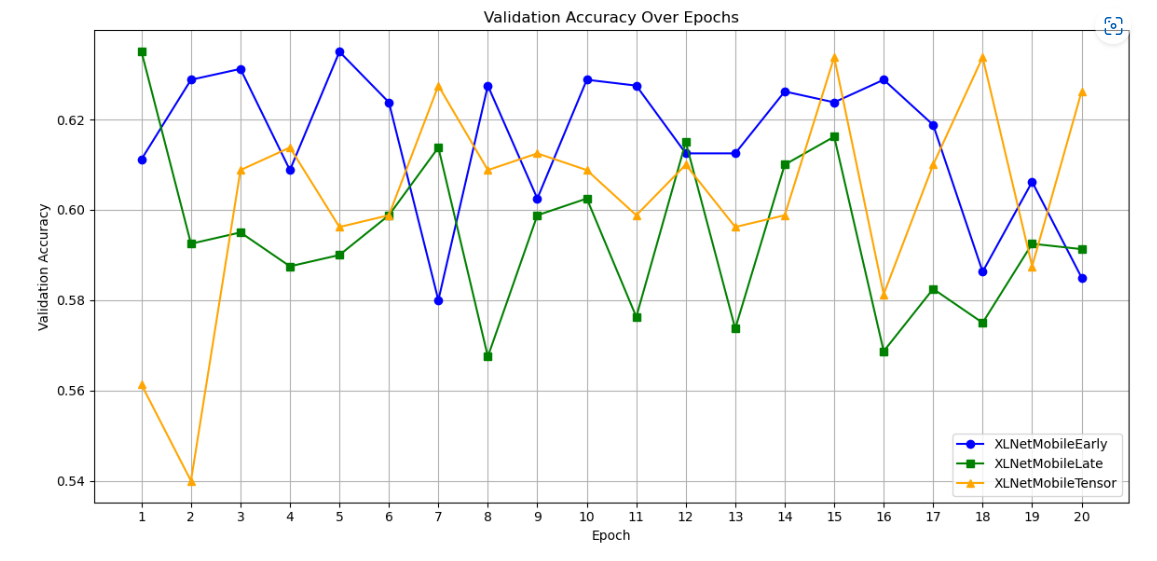
XLNetResEarly在epoch 2达到了最大的验证集准确率，为0.6625

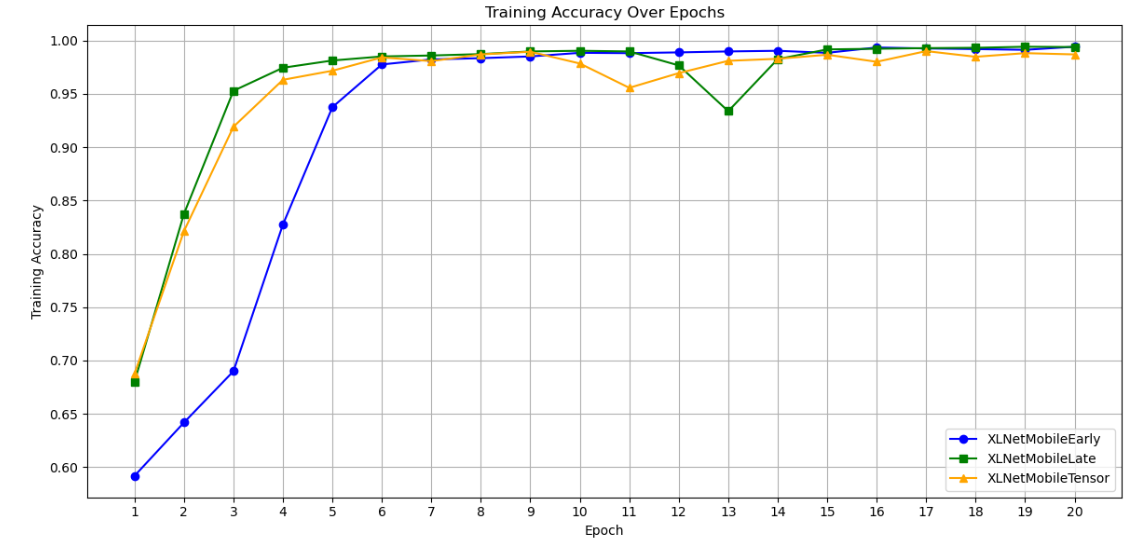
XLNetResLate在epoch 1达到了最大的验证集准确率，为0.6425

XLNetResTensor在epoch 5达到了最大的验证集准确率，为0.6550

### XLNet+MobileNetV3

XLNetMobileEarly  
epoch: 1, loss: 0.8337, train acc: 0.5919, valid acc: 0.6112  
epoch: 2, loss: 0.5602, train acc: 0.6419, valid acc: 0.6288  
epoch: 3, loss: 0.6570, train acc: 0.6900, valid acc: 0.6312  
epoch: 4, loss: 0.7748, train acc: 0.8275, valid acc: 0.6088  
epoch: 5, loss: 0.2074, train acc: 0.9375, valid acc: 0.6350  
epoch: 6, loss: 0.0408, train acc: 0.9775, valid acc: 0.6238  
epoch: 7, loss: 0.0135, train acc: 0.9822, valid acc: 0.5800  
epoch: 8, loss: 0.0292, train acc: 0.9834, valid acc: 0.6275  
epoch: 9, loss: 0.0010, train acc: 0.9850, valid acc: 0.6025  
epoch: 10, loss: 0.0036, train acc: 0.9884, valid acc: 0.6288  
epoch: 11, loss: 0.0163, train acc: 0.9881, valid acc: 0.6275  
epoch: 12, loss: 0.0931, train acc: 0.9888, valid acc: 0.6125  
epoch: 13, loss: 0.0143, train acc: 0.9897, valid acc: 0.6125  
epoch: 14, loss: 0.0025, train acc: 0.9903, valid acc: 0.6262  
epoch: 15, loss: 0.0318, train acc: 0.9884, valid acc: 0.6238  
epoch: 16, loss: 0.0070, train acc: 0.9934, valid acc: 0.6288  
epoch: 17, loss: 0.0012, train acc: 0.9925, valid acc: 0.6188  
epoch: 18, loss: 0.0004, train acc: 0.9919, valid acc: 0.5863  
epoch: 19, loss: 0.0006, train acc: 0.9912, valid acc: 0.6062  
epoch: 20, loss: 0.0004, train acc: 0.9941, valid acc: 0.5850  
XLNetMobileLate  
epoch: 1, loss: 0.5322, train acc: 0.6800, valid acc: 0.5863  
epoch: 2, loss: 0.5434, train acc: 0.8375, valid acc: 0.5925  
epoch: 3, loss: 0.2570, train acc: 0.9528, valid acc: 0.5950  
epoch: 4, loss: 0.0397, train acc: 0.9744, valid acc: 0.5875  
epoch: 5, loss: 0.1651, train acc: 0.9812, valid acc: 0.5900  
epoch: 6, loss: 0.0018, train acc: 0.9850, valid acc: 0.5988  
epoch: 7, loss: 0.0014, train acc: 0.9859, valid acc: 0.6138  
epoch: 8, loss: 0.0018, train acc: 0.9872, valid acc: 0.5675  
epoch: 9, loss: 0.0011, train acc: 0.9897, valid acc: 0.5988  
epoch: 10, loss: 0.0629, train acc: 0.9903, valid acc: 0.6025  
epoch: 11, loss: 0.0020, train acc: 0.9897, valid acc: 0.5763  
epoch: 12, loss: 0.3514, train acc: 0.9766, valid acc: 0.6150  
epoch: 13, loss: 0.1563, train acc: 0.9337, valid acc: 0.5737  
epoch: 14, loss: 0.0672, train acc: 0.9825, valid acc: 0.6100  
epoch: 15, loss: 0.0059, train acc: 0.9916, valid acc: 0.6162  
epoch: 16, loss: 0.0018, train acc: 0.9922, valid acc: 0.5687  
epoch: 17, loss: 0.0010, train acc: 0.9928, valid acc: 0.5825  
epoch: 18, loss: 0.0006, train acc: 0.9931, valid acc: 0.5750  
epoch: 19, loss: 0.0858, train acc: 0.9941, valid acc: 0.5925  
epoch: 20, loss: 0.0007, train acc: 0.9938, valid acc: 0.5913  
XLNetMobileTensor  
epoch: 1, loss: 0.5570, train acc: 0.6875, valid acc: 0.5613  
epoch: 2, loss: 0.5491, train acc: 0.8213, valid acc: 0.5400  
epoch: 3, loss: 0.2436, train acc: 0.9191, valid acc: 0.6088  
epoch: 4, loss: 0.0246, train acc: 0.9631, valid acc: 0.6138  
epoch: 5, loss: 0.2540, train acc: 0.9716, valid acc: 0.5962  
epoch: 6, loss: 0.0007, train acc: 0.9841, valid acc: 0.5988  
epoch: 7, loss: 0.0333, train acc: 0.9806, valid acc: 0.6275  
epoch: 8, loss: 0.0636, train acc: 0.9869, valid acc: 0.6088  
epoch: 9, loss: 0.0060, train acc: 0.9894, valid acc: 0.6125  
epoch: 10, loss: 0.5883, train acc: 0.9784, valid acc: 0.6088  
epoch: 11, loss: 0.1595, train acc: 0.9556, valid acc: 0.6088  
epoch: 12, loss: 0.0476, train acc: 0.9694, valid acc: 0.6100  
epoch: 13, loss: 0.0090, train acc: 0.9809, valid acc: 0.6200  
epoch: 14, loss: 0.0015, train acc: 0.9828, valid acc: 0.6238  
epoch: 15, loss: 0.0002, train acc: 0.9866, valid acc: 0.6338  
epoch: 16, loss: 0.0037, train acc: 0.9800, valid acc: 0.5813  
epoch: 17, loss: 0.0502, train acc: 0.9900, valid acc: 0.6100  
epoch: 18, loss: 0.0013, train acc: 0.9847, valid acc: 0.6338  
epoch: 19, loss: 0.1155, train acc: 0.9881, valid acc: 0.5875  
epoch: 20, loss: 0.0192, train acc: 0.9869, valid acc: 0.6262





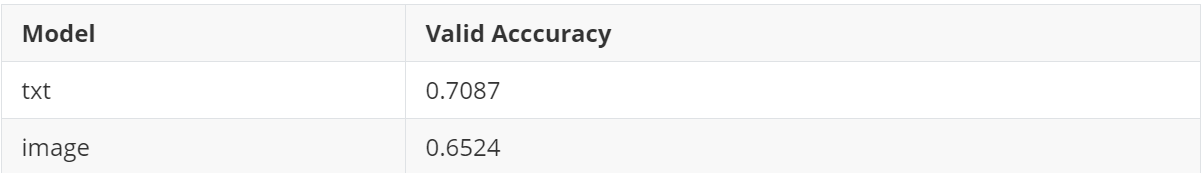
XLNetMobileEarly在epoch 5达到了最大的验证集准确率，为0.6350

XLNetMobileLate在epoch 15达到了最大的验证集准确率，为0.6162

XLNetMobileTensor在epoch 15/18达到了最大的验证集准确率，为0.6338

### 消融实验

# text only  
epoch 1/20 -> train acc: 0.6362, valid acc: 0.6950  
epoch 2/20 -> train acc: 0.7425, valid acc: 0.7075  
epoch 3/20 -> train acc: 0.8087, valid acc: 0.7050  
epoch 4/20 -> train acc: 0.8700, valid acc: 0.6863  
epoch 5/20 -> train acc: 0.9078, valid acc: 0.6850  
epoch 6/20 -> train acc: 0.9378, valid acc: 0.6675  
epoch 7/20 -> train acc: 0.9506, valid acc: 0.6725  
epoch 8/20 -> train acc: 0.9606, valid acc: 0.6775  
epoch 9/20 -> train acc: 0.9697, valid acc: 0.6787  
epoch 10/20 -> train acc: 0.9706, valid acc: 0.6937  
epoch 11/20 -> train acc: 0.9738, valid acc: 0.6875  
epoch 12/20 -> train acc: 0.9759, valid acc: 0.6913  
epoch 13/20 -> train acc: 0.9759, valid acc: 0.6825  
epoch 14/20 -> train acc: 0.9756, valid acc: 0.6925  
epoch 15/20 -> train acc: 0.9756, valid acc: 0.6987  
epoch 16/20 -> train acc: 0.9741, valid acc: 0.7050  
epoch 17/20 -> train acc: 0.9778, valid acc: 0.7037  
epoch 18/20 -> train acc: 0.9762, valid acc: 0.7025  
epoch 19/20 -> train acc: 0.9756, valid acc: 0.7087  
epoch 20/20 -> train acc: 0.9750, valid acc: 0.6900  
# image only  
epoch 1/20 -> train acc: 0.5763, valid acc: 0.6362  
epoch 2/20 -> train acc: 0.7331, valid acc: 0.6525  
epoch 3/20 -> train acc: 0.8559, valid acc: 0.6162  
epoch 4/20 -> train acc: 0.9556, valid acc: 0.5925  
epoch 5/20 -> train acc: 0.9841, valid acc: 0.6150  
epoch 6/20 -> train acc: 0.9850, valid acc: 0.6012  
epoch 7/20 -> train acc: 0.9856, valid acc: 0.6175  
epoch 8/20 -> train acc: 0.9881, valid acc: 0.6125  
epoch 9/20 -> train acc: 0.9856, valid acc: 0.5863  
epoch 10/20 -> train acc: 0.9878, valid acc: 0.6100  
epoch 11/20 -> train acc: 0.9866, valid acc: 0.6150  
epoch 12/20 -> train acc: 0.9875, valid acc: 0.6062  
epoch 13/20 -> train acc: 0.9891, valid acc: 0.6138  
epoch 14/20 -> train acc: 0.9878, valid acc: 0.6162  
epoch 15/20 -> train acc: 0.9856, valid acc: 0.6375  
epoch 16/20 -> train acc: 0.9875, valid acc: 0.6150  
epoch 17/20 -> train acc: 0.9862, valid acc: 0.5962  
epoch 18/20 -> train acc: 0.9891, valid acc: 0.6212  
epoch 19/20 -> train acc: 0.9881, valid acc: 0.6250  
epoch 20/20 -> train acc: 0.9906, valid acc: 0.6325

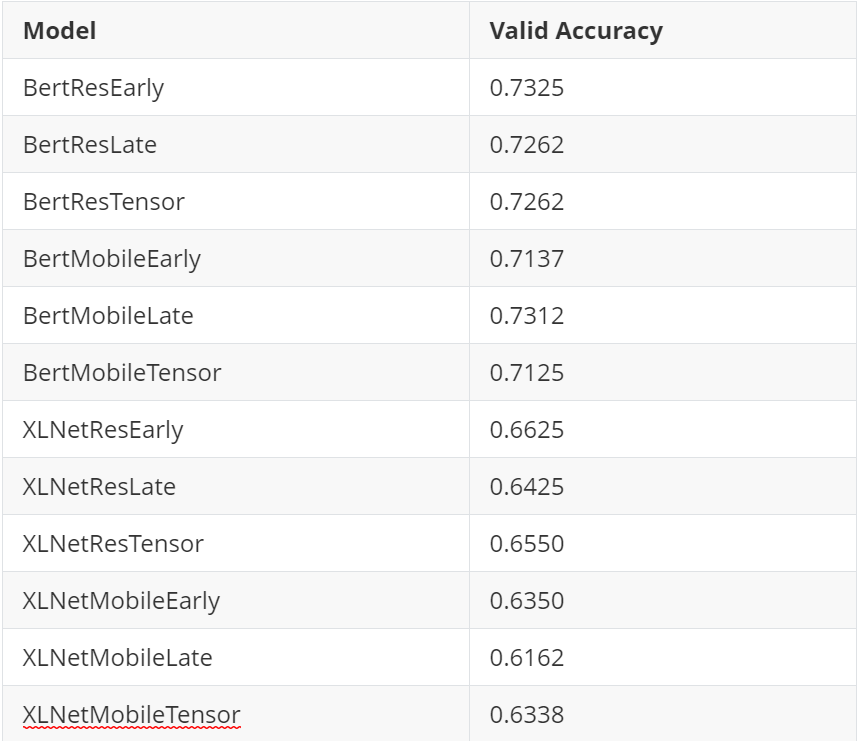


可以看到仅仅从图像来进行情感识别，准确率是远远不如从文本进行情感识别的。说明，文本相比起图像更加能够唤起情感，图像仅能起到辅助的作用而已。

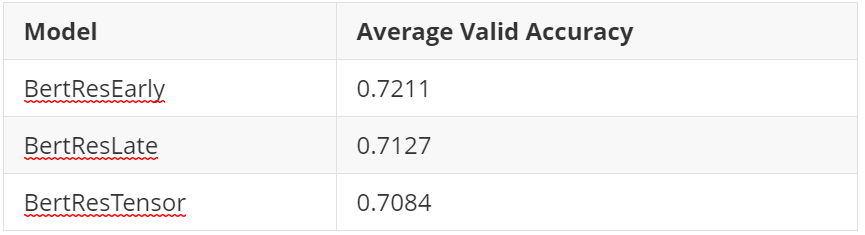
就拿之前举过的反话的例子，文字如果是”天气真好啊“，配图如果天气晴朗万里无云，那就是正话；配图如果是狂风暴雨的话，那就是反话。但是说反话的毕竟是少数情况，仅从文本判断的话也是能够得到不错的准确率了。但是如果仅从图像进行判断的话，那么情况就多的多了。如果是狂风暴雨，那么干旱已久之后的出现狂风暴雨与正要出门玩的时候出现的狂风暴雨肯定不一样，而且这两种情况都很有可能发生。所以，图像的准确率要低于文本的准确率。

但是比较离谱的一点是，当使用XLNet作为text encoder的时候，获得的准确率甚至要低于仅用图片的情况。这似乎表示经过XLNet之后的文本表示似乎还起到了反效果。我目前无法得出原因，希望日后可以继续探索。

### 综合分析



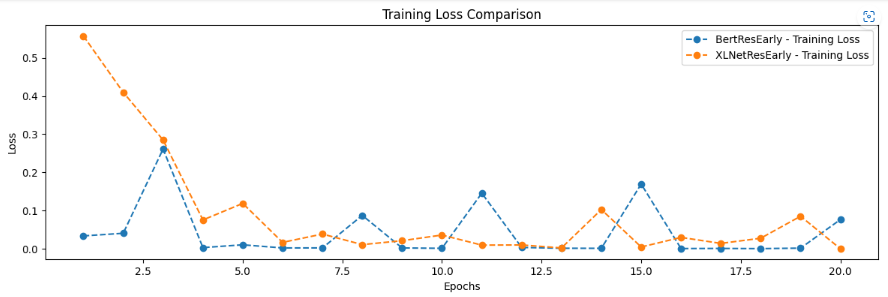
可以看到BertResEarly到达了最好的准确率。这其实挺出乎我的意料的，因为我觉得更加复杂的方法应该能够获得更好的准确率，但现在却是最最简单的方法达到了最高的准确率。不过，我上面记录的仅仅是20个epoch中最高的准确率，而并非是平均准确率。再来看下平均准确率吧：

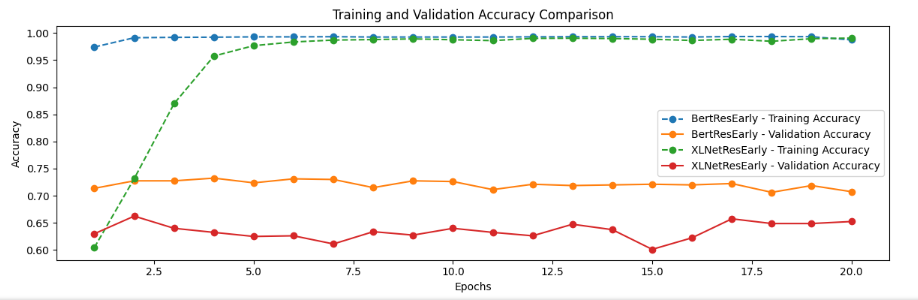
可以看到，最高的依然是BertResEarly。

当然这个”最高“的准确率听起来也并非那么的最，因为它也仅仅比仅文本情况下准确率高出了三个百分比。我觉得这主要说明了图像对于情感识别的帮助真的十分有限。有时候光靠文本就已经能够做出足够可靠的识别，只不过在特殊情况（反话等）下才需要图像的帮助。

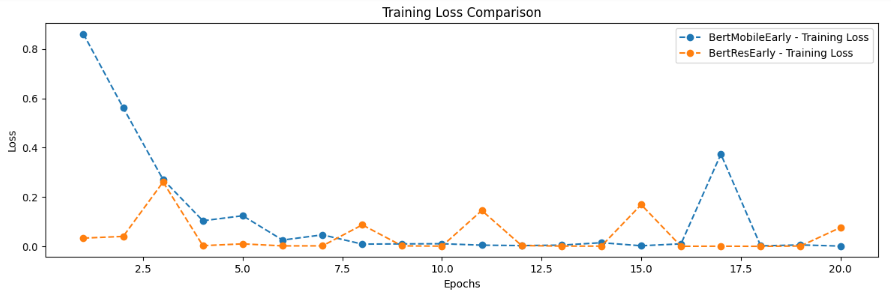
early fusion的改进版本的表现并不如最最直接的early fusion，我认为原因可能就是图像对于情感唤起的作用实在是太有限，所以始终都是文本占据主导地位。无论图像乘上多大的权重，对于总体的影响可以说是微乎其微。

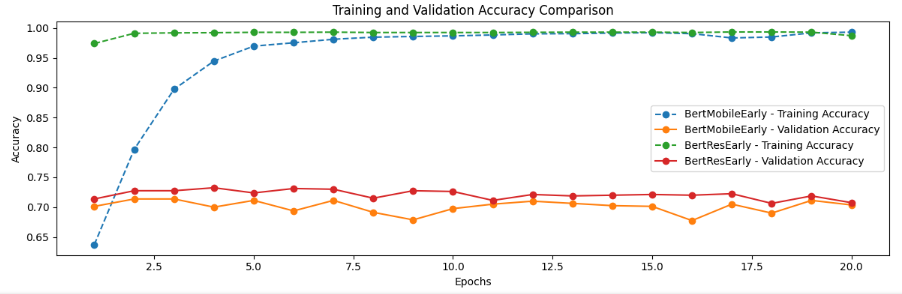
而且训练过程中是出现了比较严重的过拟合现象的，训练集准确率基本都能够达到99%，但是验证集准确率却只能在60%和70%左右。原因还是在于数据集比较小，其实我本来是想用数据增强来扩大数据集规模的。但是马上意识到在多模态情感识别这个任务当中数据增强并没有多少意义，因为图像和文本是配套的，把图像左右翻转或者上下颠倒，图像所唤起的情感也并不会有改变。





可以看到在同样都是ResEarly的情况下，Bert的表现要远远好于XLNet。在其他给定的imageencoder和fusionmethod上也出现了相似的情况。





可以看到在都是BertEarly的情况下，ResNet的结果可以说稍微比MobileNet要好一点，但也只是一点，其实准确率是差不多的。



可以看到在都是BerRes的情况下，三种fusion method是基本没什么差异的。

所以，其实可以说BertResEarly的高准确率是拜Bert所赐，和ResNet152以及Early Fusion的关系虽然也有那么一点，但是并不是很大。不过我这里的模型都是加载的torchivision所提供的预训练好的模型，如果你是要自己训练的话，那么训练MobileNet所得到的结果不会比ResNet差很多，但是速度上会快上很多。

## 感想

感觉通过阅读论文来了解某一个研究领域是一个快速上手的好办法，因为论文好歹是论文，平均质量上肯定是要优于网上能找到的博客之类的。但是想要找到好的论文，尤其是好的综述论文还是需要花一点时间的。我在找到报告中所提到的那篇论文之前还找了许多论文，但是真的就是看不懂，那些论文针对的是已经有了一定基础的读者，而不是像我这样几乎完全不懂的零基础。我也是偶然之间通过csdn上的论文推荐才找到这篇论文的。我在想这类面向新手的综述类论文是不是有些比较共同的关键词，比如说这篇论文标题：Emotion Recognition from Multiple Modalities: Fundamentals and Methodologies中的Fundamentals and Methodologies。感觉熟悉了这些关键词之后就能够通过谷歌学术找到符合我需要的论文。

而且其实本次实验的任务是多模态情感分析（Multimodal Sentiment Analysis）中的一个子任务：多模态情感识别（Multimodal Emotion Recognition）。多模态情感分析还有许多其他的子任务，比如说Multimodal Aspect-Based Sentiment Analysis（多模态方面级情感分析）和多模态反讽识别（Multimodal Irony Detection）。如果没有精确地搜索Multimodal Emotion Recognition的话，可能会被无关信息所淹没。所以，先明确目的也是很重要的。

可以看到我遇到的问题板块我写的很长，因为确实写代码的时候走了相当多的弯路。我感觉自己还是应该对自己所使用的库有一些最基础的了解。就比如本次实验我在处理dataset产生的问题上花了相当多的功夫，结果最后还因为无法解决的问题放弃了使用dataset。放弃Trainer也是同样的原因。如果我能稍微去了解一些Dataset和Trainer的原理，稍微看点源的话，可能上面的问题都能顺利解决。而且感觉写代码的时候，出现的问题中和模型定义相关的还是相当少的，多半都是些库的使用上的问题。

感觉这最后的实验基本上用到了这学期学到的大部分知识，同时涵盖了cv、nlp和多模态等知识。而由于我本学期同时选修了计算机视觉，而且两节课刚好是接着的。经常会出现陆雪松老师刚讲的内容，李翔老师又讲了的情况。难度上计算机视觉是入门，而当代人工智能就是稍微不那么入门，进阶一些的东西了。总体上，我感觉上完这两门课之后，对深度学习就有一定的基础了。希望日后能借由这些知识，继续学习并研究AI。

## 一些补充

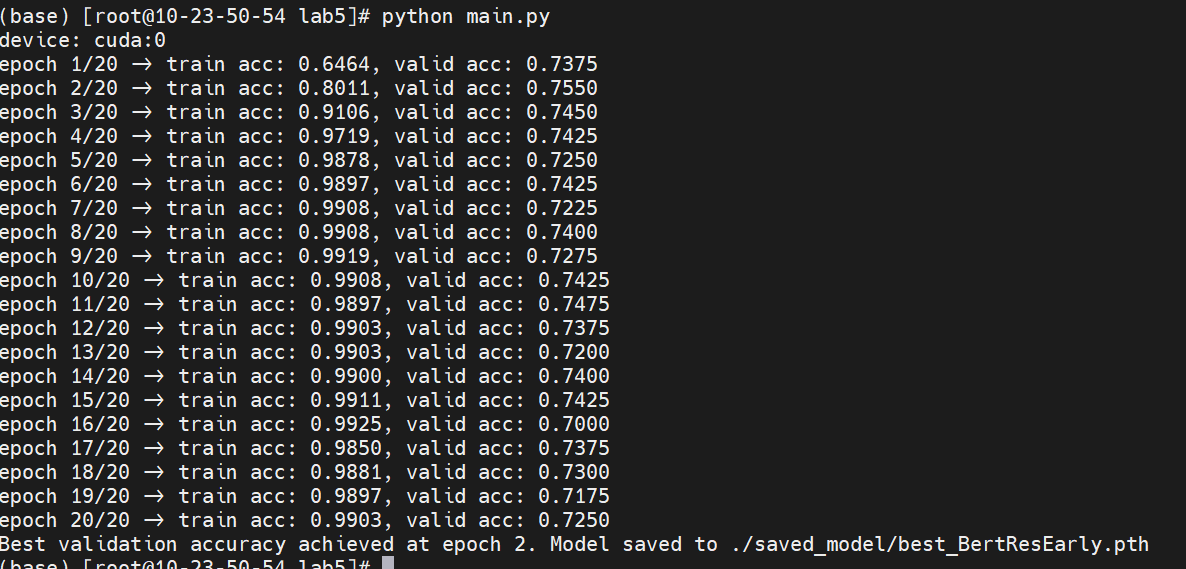
我在交完作业后才意识到一个地方其实是有改进空间的，我就补充在这里吧。

前面我提到训练过程出现了非常严重的过拟合现象，训练集准确率可以到达99%，但是验证集准确率只有70%左右。而众所周知的一个解决过拟合的方式就是扩大数据集，我本来想着是data augmentation，但行不通。其实还有另一个方法，就是在划分训练集和验证集的时候扩大训练集比例，降低验证集比例：

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.1, random\_state=42)

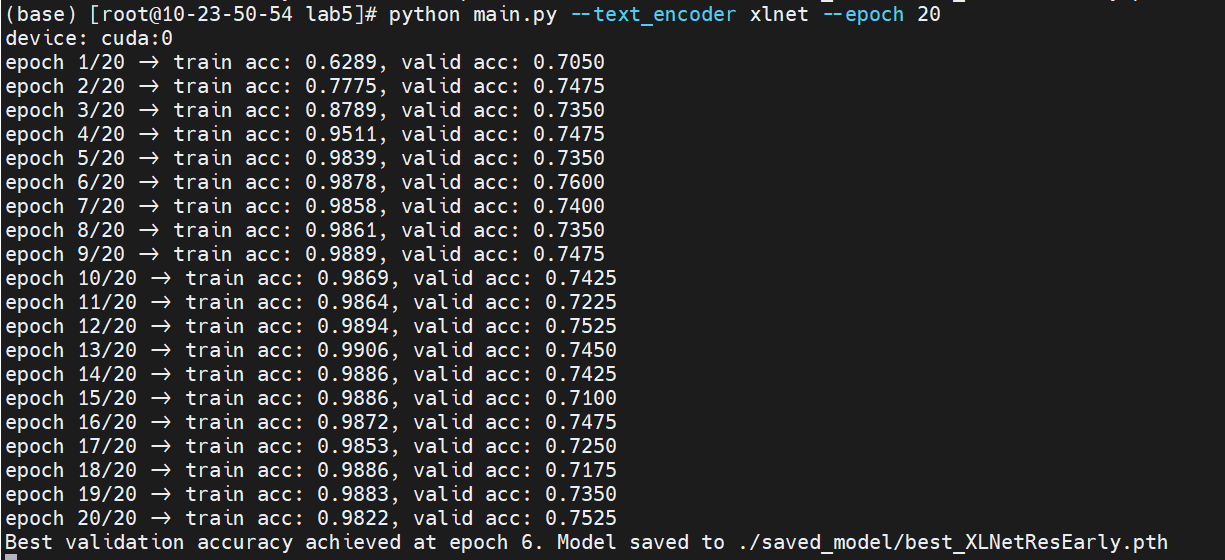
我本来是8:2的，现在调整成9:1。

然后就直接python main.py，使用BertResEarly：

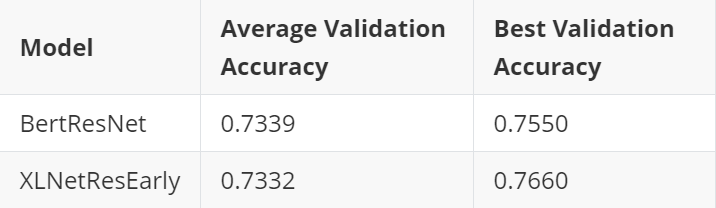


可以看到结果有了非常显著的提升。

而且还有一个惊喜的发现，当我python main.py --text\_encoder xlnet --epoch 20使用XLNetResEarly，效果要远远比之前8:2划分数据集的时候要好：



可以看到XLNetResEarly最高的验证集准确率为0.7600，而BertResNet最高的验证集准确率为0.7550。而且，不光是最高，甚至连平均也超越了：



这样其实就找到先前XLNet的表现不尽如人意的原因了，就是在于训练集实在是太小。

所以本次我最终提交的预测文件是基于9:1划分的数据集的