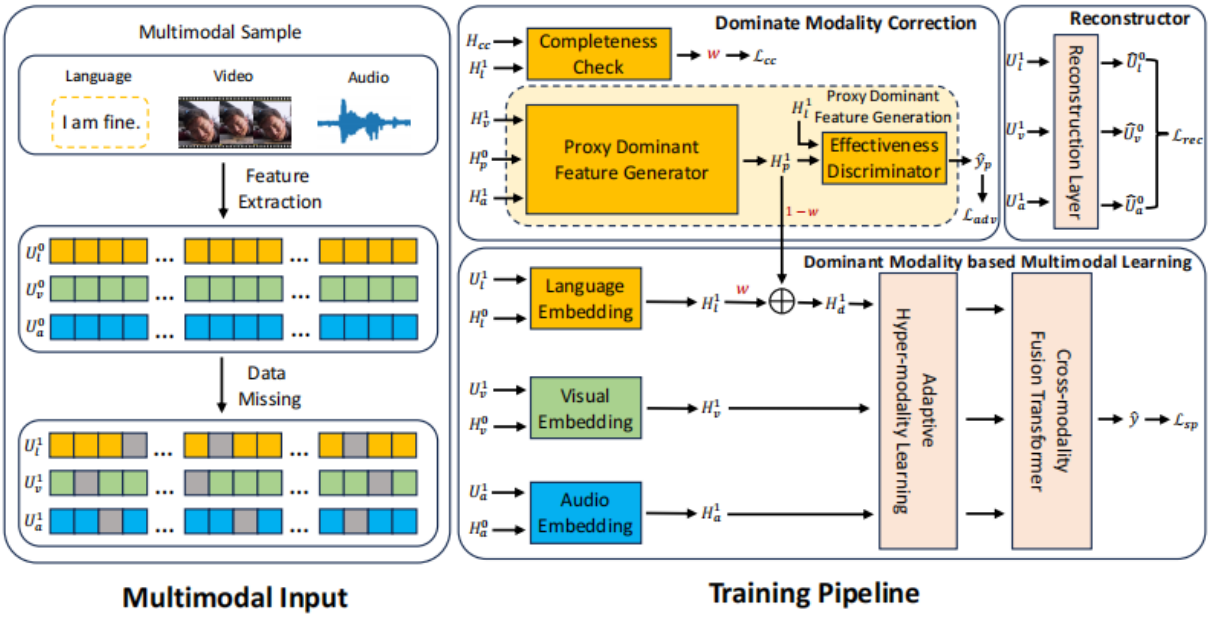
本篇是基于AMLT改良的。



模态嵌入获取：

语言——bert

音频——Librosa

视觉——Openface

获取到U_m^0 ,然后对其应用随机数据缺失——对于每个样本中的每个模态，我们随机删除了不断变化的信息比例（从0%到100%）。具体来说，对于视觉和音频模式，我们用零填充被擦除的信息。对于语言形态，我们用[UNK]填充被删除的信息，它表示BERT中的未知词，得到U_m^1

基于主导模态的多模态学习DMML

受之前工作ALMT的启发，**我们假设，尽管噪声水平不同，但当主导模态的完整性保持不变时，模型的鲁棒性会得到提高。**

模态嵌入:

$$H_m^1 = \mathrm{E}_m^1\left(\mathrm{concat}\left(H_m^0, U_m^1\right)\right)$$

适应性超模态学习:

考虑到随机数据缺失可能对语言模态（即主导模态）造成严重干扰的可能性，我们设计了一个显性模态校正（DMC）模块来生成代理主导特征H_p^1，并构建校正后的主导特征H_d^1

i∈{2,3}表示自适应超模态学习模块(f Adaptive Hyper-modality Learning module)的第i层,Eim（·）是第i个Transformer编码器层:

$$H_d^i = \mathrm{E}_m^i \left(H_d^{i-1} \right)$$

是在不同尺度上校正后的主导特征。

为了学习超模态表示，使用校正后的主导特征和音频/视觉特征分别计算查询（Query）和键值（Key/Value）。简要来说，这个过程可以写成如下形式:

$$H_{\text{hyper}}^i = H_{\text{hyper}}^{i-1} + \text{MHA}(H_d^i, H_a^1) + \text{MHA}(H_d^i, H_v^1)$$

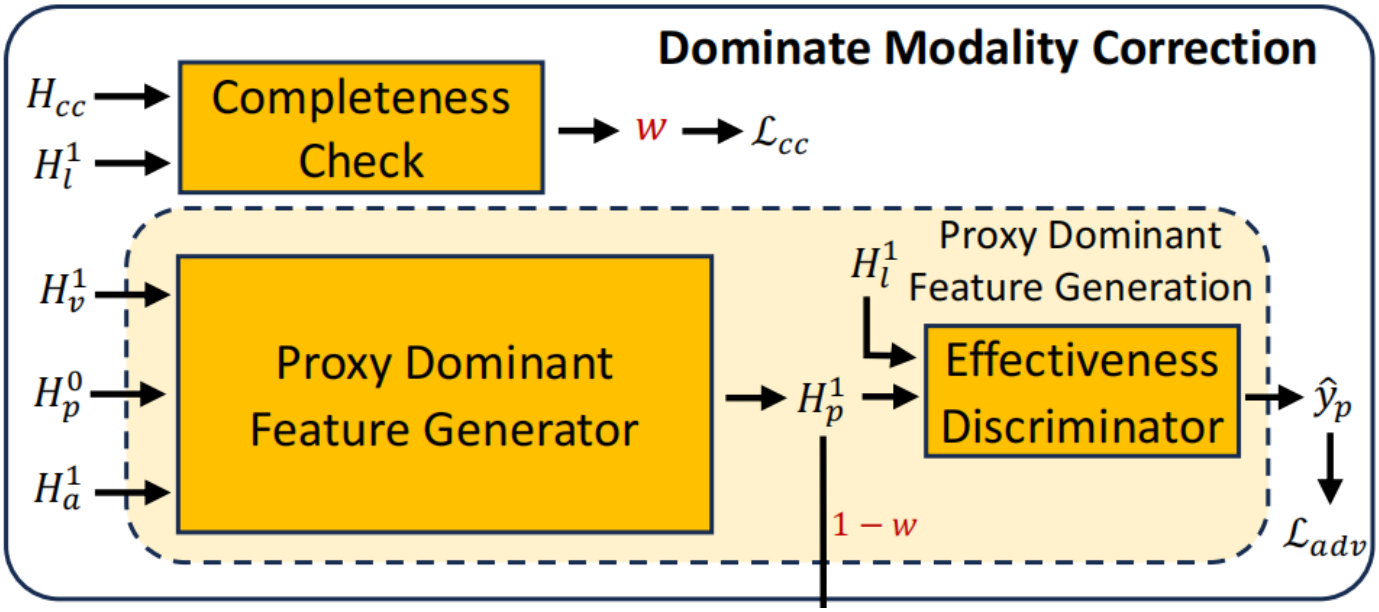
其中 MHA(·) 表示多头注意力机制，Hhyper_i ∈ R^T×d 是超模态特征。

跨模态融合变换器:

利用获得的 Hd_3 和 Hhyper_3，采用一个具有分类器且深度为4层的Transformer编码器进行多模态融合和情感预测。

$$\hat{y} = \text{CrossTransformer}(H_d^3, H_{hyper}^3)$$

优势模态校正DMC:



完全性检查(使用对抗学习):

还能这样呢，单模态的完整性预测。我有一个问题，这个完整性（1-缺失率），为啥不能直接数出来呢？我记得缺失是可以数的吧。

我们应用了一个编码器Ecc，它由一个具有两层深度的变压器编码器和一个分类器组成，以进行完整性检查。例如，如果主导模态的缺失率为0.3，则完整性的标签为0.7。这个完整性预测w可以得到如下结果

$$w = \mathrm{E}_{cc}\left(\mathrm{Concat}\left(H_{cc}, H_l^1\right)\right)$$

这个w是和缺失率相关的一个权重啊，但是我可以借鉴他这样取了个权重的方法好像。我目前在融合阶段想的也是想办法搞一个权重，为了满足u不是盲目增大减小，这个权重一定要归一化。

可以使用一个L2范数来优化这个过程：

$$\mathcal{L}_{cc} = \frac{1}{N_b}\sum_{k=0}^{N_b}\left\|w^k - \hat{w}^k\right\|_2^2$$

采用了一个代理主导特征生成器E_{DFG}生成代理主导特征H_p^1旨在补充和纠正主导模式。通过将H_p^1和语言特征H_L^1相结合，计算出修正后的优势特征H_d^1，并对预测的完整性w进行加权:

$$H_p^1 = \mathrm{E}_{\mathrm{DFG}}\left(\mathrm{Concat}\left(H_p^0, H_a^1, H_v^1\right), \theta_{DFG}\right)$$

$$H_d^1 = (1-w) * H_p^1 + w * H_l^1,$$

为了确保代理特征提供了一个从视觉和音频特征不同的视角，我们使用了一个有效性鉴别器Effectiveness Discriminator,这个鉴别器包括一个二元分类器和一个梯度反向层,任务是识别代理特征的起源

$$\hat{y}_p = \mathrm{D}\left(H_p^1 / H_l^1, \theta_D\right)$$

$\hat{y}_p$表示对输入特征是否来自于语言模态的预测。

在实践中，生成器(Proxy Dominant Feature Generator)和鉴别器(Effectiveness Discriminator)采用了一种对抗性的学习结构。鉴别器的目的是识别这些特征是否来自于语言模态，而生成器的目标是挑战鉴别器做出准确预测的能力。这种动态被封装在对抗性的学习目标中

$$\min_{\theta_D}\max_{\theta_{DFG}}\mathcal{L}_{adv} = -\frac{1}{N_b}\sum_{k=0}^{N_b} y_p^k \cdot \log \hat{y}_p^k$$

暂时对生成器和鉴别器的理解：生成器想让负对数损失最大化，鉴别器想让负对数损失最小化，就是说，生成器想往好的生产，鉴别器想往坏的生成。

** 代理主导特征生成：**

采用了一个代理主导特征生成器E_{DFG}生成代理主导特征H_p^1旨在补充和纠正主导模式。通过将H_p^1和语言特征H_L^1相结合，计算出修正后的优势特征H_d^1，并对预测的完整性w进行加权:

$$H_p^1 = \mathrm{E}_{\mathrm{DFG}}\left(\mathrm{Concat}\left(H_p^0, H_a^1, H_v^1\right), \theta_{DFG}\right)$$

$$H_d^1 = (1-w) * H_p^1 + w * H_l^1,$$

为了确保代理特征提供了一个从视觉和音频特征不同的视角，我们使用了一个有效性鉴别器Effectiveness Discriminator,这个鉴别器包括一个二元分类器和一个梯度反向层,任务是识别代理特征的起源

$$\hat{y}_p = \mathrm{D}\left(H_p^1 / H_l^1, \theta_D\right)$$

$\hat{y}_p$表示对输入特征是否来自于语言模态的预测。

在实践中，生成器(Proxy Dominant Feature Generator)和鉴别器(Effectiveness Discriminator)采用了一种对抗性的学习结构。鉴别器的目的是识别这些特征是否来自于语言模态，而生成器的目标是挑战鉴别器做出准确预测的能力。这种动态被封装在对抗性的学习目标中

$$\min_{\theta_D} \max_{\theta_{DFG}} \mathcal{L}_{adv} = -\frac{1}{N_b} \sum_{k=0}^{N_b} y_p^k \cdot \log \hat{y}_p^k$$

暂时对生成器和鉴别器的理解：生成器想让负对数损失最大化，鉴别器想让负对数损失最小化，就是说，生成器想往好的生产，鉴别器想往坏的生成。

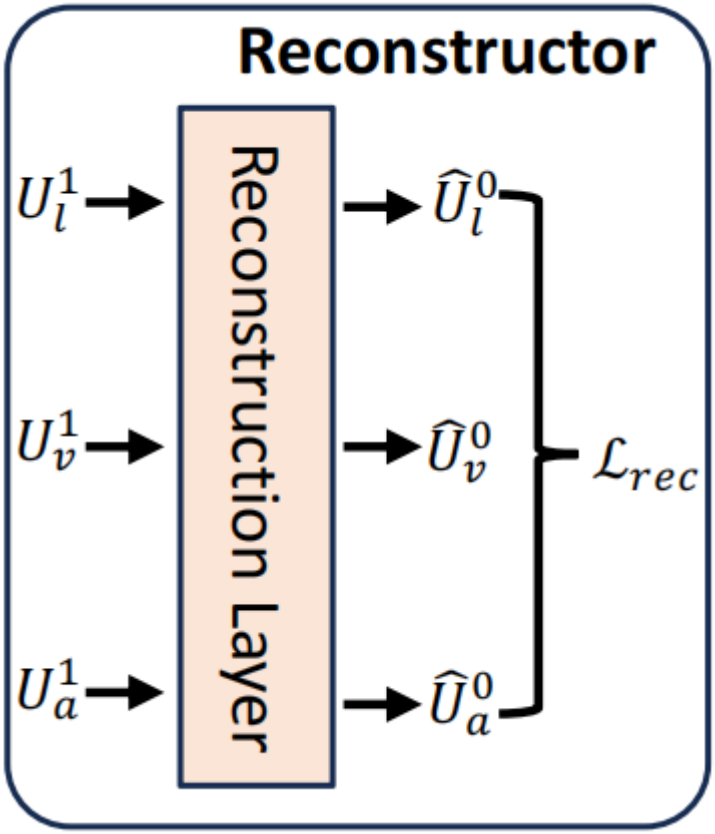
对于生成模型 G，其目标是最小化 V(D,G)，即希望生成尽可能逼真的样本，使得判别模型无法区分真实样本和生成样本。

对于判别模型 D，其目标是最大化 V(D,G)，即希望尽可能准确地判断样本是真实的还是生成的。

== 我有一个问题：你这个y_p^k咋搞的？==

== 对抗学习的困难与挑战：对抗学习的评估通常比传统机器学习更为困难，因为生成的数据样本没有明确的标签或评价标准。==

** 再现器Reconstructor**



一个重构器，表示为Erec，它由两个变压器层组成，旨在有效地重建每个模态的缺失信息。

$$\hat{U}_m^0 = \mathrm{E}_m^{rec}\left(U_m^1\right)$$

$$\mathcal{L}_{rec} = \frac{1}{N_b}\sum_{h=0}^{N_b}\sum_{m}\left\|U_m^{0\,k} - \hat{U}_m^{0\,k}\right\|_2^2$$

这种损失函数有助于最小化原始特征和重建特征之间的差异，从而提高其他成分的准确性，如显性模态校正和最终的情绪预测。

总体学习目标

$$\mathcal{L} = \alpha\mathcal{L}_{cc} + \beta\mathcal{L}_{adv} + \gamma\mathcal{L}_{rec} + \delta\mathcal{L}_{sp}$$

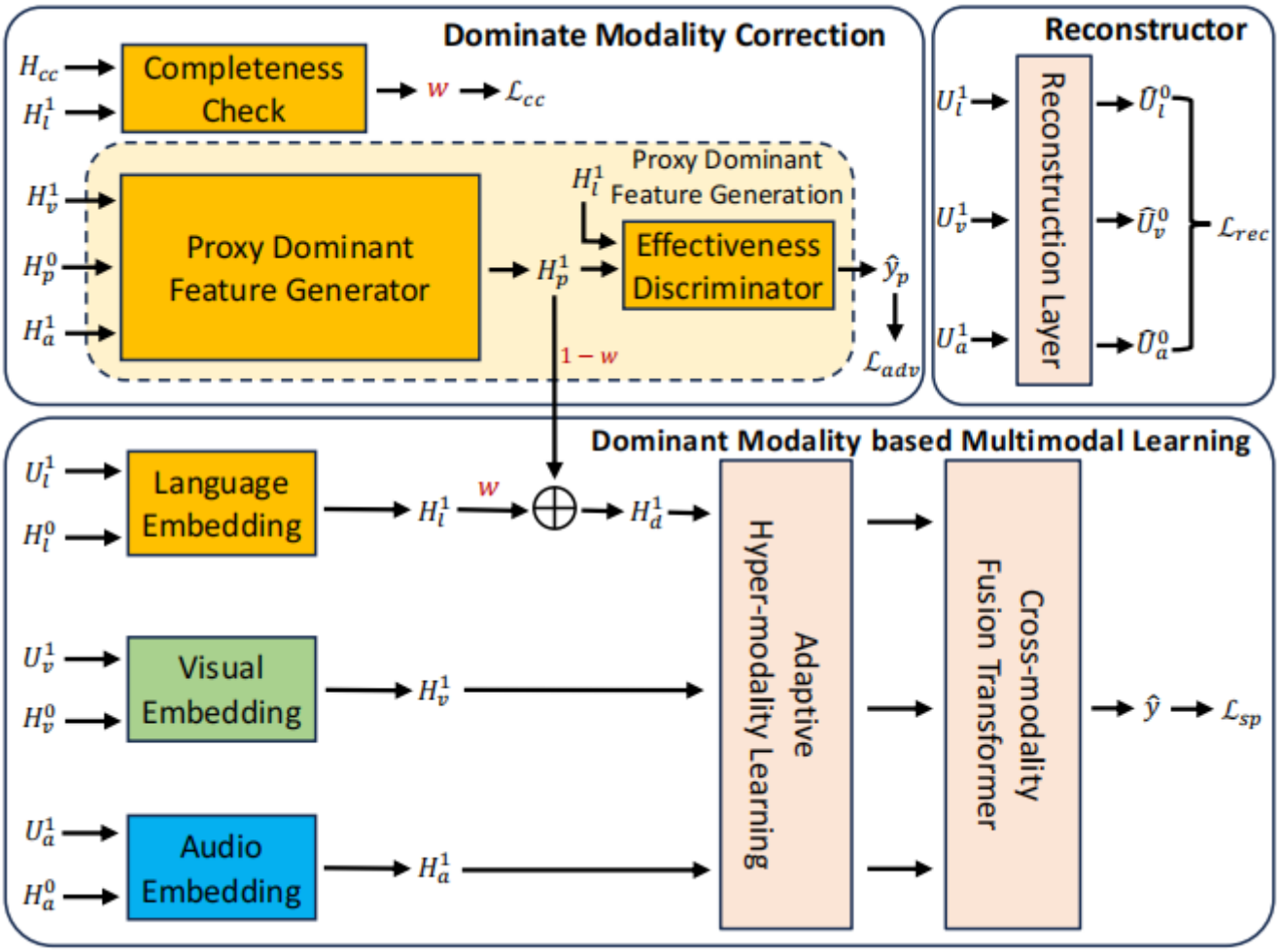
α、β、γ和δ均为超参数。在MOSI和MOSEI数据集上，我们根据经验将它们分别设置为0.9、0.8、0.1和1.0。在SIMS数据集上，我们将它们分别设置为0.9、0.6、0.1和1.0。

结合论文看代码：

```
bert.py
class BertTextEncoder(nn.Module):
    def __init__(self, use_finetune=False, transformers='bert', pretrained='bert-base-
uncased'):
        super().__init__()

        tokenizer_class = TRANSFORMERS_MAP[transformers][1]
        model_class = TRANSFORMERS_MAP[transformers][0]
        self.tokenizer = tokenizer_class.from_pretrained(pretrained)
        self.model = model_class.from_pretrained(pretrained)
        self.use_finetune = use_finetune

    def get_tokenizer(self):
        return self.tokenizer

    def forward(self, text):
        """
        text: (batch_size, 3, seq_len)
        3: input_ids, input_mask, segment_ids
        input_ids: input_ids,
        input_mask: attention_mask,
        segment_ids: token_type_ids
        """
        input_ids, input_mask, segment_ids = text[:,0,:].long(), text[:,1,:].float(),
text[:,2,:].long()
        if self.use_finetune:
            last_hidden_states = self.model(input_ids=input_ids,
                                            attention_mask=input_mask,
                                            token_type_ids=segment_ids)[0]  # Models
outputs are now tuples
        else:
            with torch.no_grad():
                last_hidden_states = self.model(input_ids=input_ids,
                                                attention_mask=input_mask,
                                                token_type_ids=segment_ids)[0]  #
Models outputs are now tuples
        return last_hidden_states
```

forward接受的输入张量text形状为(batch_size, 3, seq_len)，包括input_ids, input_mask, segment_ids
其中：
input_ids 是一个整数张量，表示输入文本的词汇表中的索引。每个元素对应于一个词或子词（token）。
input_mask 是一个浮点数张量，用于指示哪些 tokens 是实际的输入，哪些是填充（padding）的 tokens。通常，实际的 toke
ns 被标记为 1，而填充的 tokens 被标记为 0。
segment_ids 是一个整数张量，用于区分不同的句子片段。这在处理两个或多个句子的任务（如问答、下一句预测等）时特别
有用。

各个模块的参数设置如下 train_sims.py：



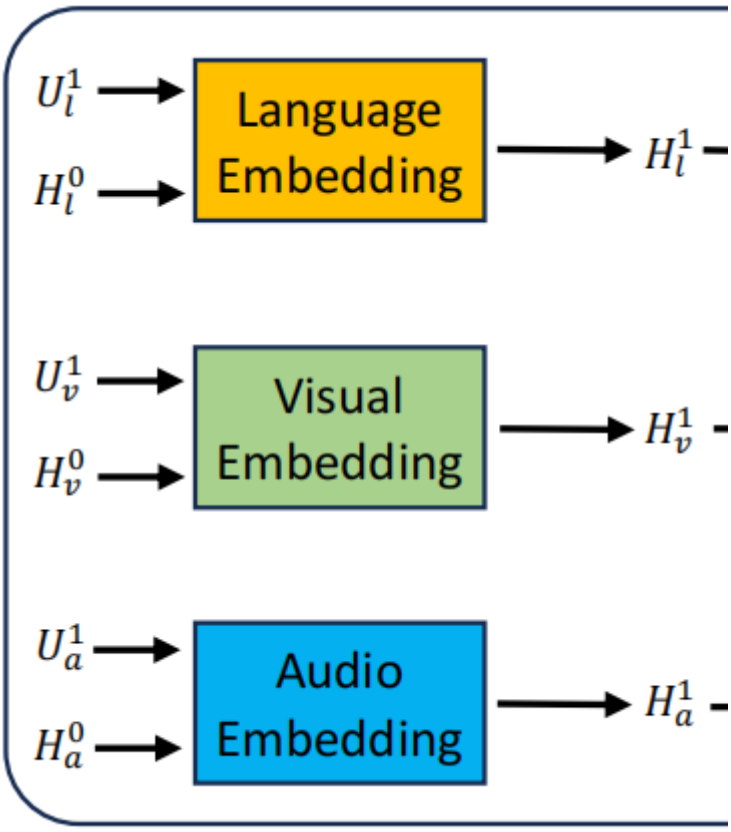

```yaml
model:
  feature_extractor:
    bert_pretrained: 'bert-base-chinese'
    input_length: [39, 55, 400] # language, video, audio
    token_length: [8, 8, 8] # language, video, audio
    heads: 8
    input_dims: [768, 709, 33] # language, video, audio
    hidden_dims: [128, 128, 128] # language, video, audio
    depth: 2

  dmc:
    proxy_dominant_feature_generator:
      input_length: 24
      token_length: 8
      depth: 2
      heads: 8
      input_dim: 128
      hidden_dim: 128

    effectiveness_discriminator:
      input_dim: 128
      hidden_dim: 64
      out_dim: 2
```

```
    completeness_check:
      input_length: 8
      token_length: 1
      depth: 2
      heads: 8
      input_dim: 128
      hidden_dim: 128

  reconstructor:
    input_length: 8
    depth: 2
    heads: 8
    input_dim: 128
    hidden_dim: 128

  dmml:
    language_encoder:
      input_length: 8
      token_length: 8
      depth: 2
      heads: 8
      input_dim: 128
      hidden_dim: 128

    hyper_modality_learning:
      depth: 3
      heads: 8
      input_dim: 128
      hidden_dim: 128

    fuison_transformer:
      source_length: 8
      tgt_length: 8
      depth: 4
      heads: 8
      input_dim: 128
      hidden_dim: 128

    regression:
      input_dim: 128
      out_dim: 1
```

构建模型：

```
train.py
#构建模型
model = build_model(args).to(device)
```

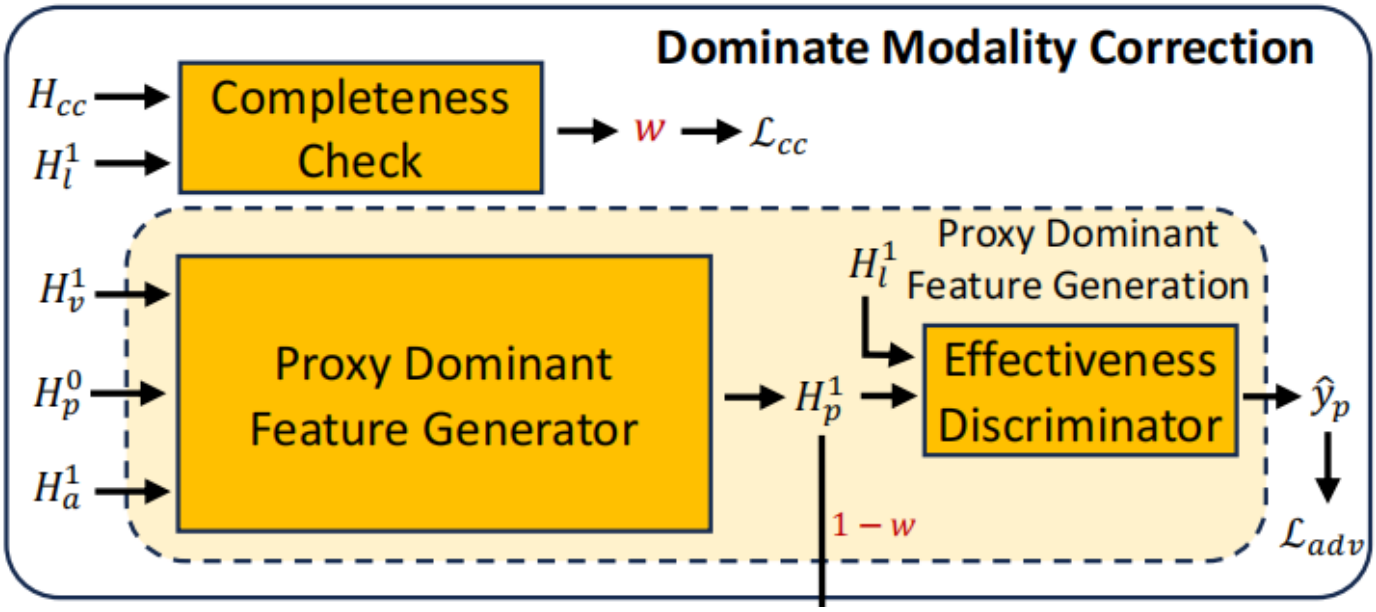
首先是定义三个投影模块：




```
#语言模态
self.proj_l = nn.Sequential(
            nn.Linear(args['model']['feature_extractor']['input_dims'][0],
args['model']['feature_extractor']['hidden_dims'][0]),
            Transformer(num_frames=args['model']['feature_extractor']['input_length']
[0],
                        save_hidden=False,
                        token_len=args['model']['feature_extractor']['token_length']
[0],
                        dim=args['model']['feature_extractor']['hidden_dims'][0],
                        depth=args['model']['feature_extractor']['depth'],
                        heads=args['model']['feature_extractor']['heads'],
                        mlp_dim=args['model']['feature_extractor']['hidden_dims'][0])
        )
#其余俩模态代码差不多
```

然后是定义对抗学习的生成器和鉴别器、以及完备性检测器：

这里有个坑点，他图里没说清楚，看代码才看出来，仅有这里检测器的输入H_l^1不是用设置缺失的U跑出来的，而是无缺失的U跑出来的，图里其他地方的H_l^1是用设置缺失的U跑出来的。




```
#对抗学习的生成器
self.proxy_dominate_modality_generator = Transformer(
        num_frames=args['model']['dmc']['proxy_dominant_feature_generator']
['input_length'],
        save_hidden=False,
        token_len=args['model']['dmc']['proxy_dominant_feature_generator']
['token_length'],
        dim=args['model']['dmc']['proxy_dominant_feature_generator']['input_dim'],
        depth=args['model']['dmc']['proxy_dominant_feature_generator']['depth'],
        heads=args['model']['dmc']['proxy_dominant_feature_generator']['heads'],
        mlp_dim=args['model']['dmc']['proxy_dominant_feature_generator']
['hidden_dim'])

#对抗学习的鉴别器
self.effective_discriminator = nn.Sequential(
        nn.Linear(args['model']['dmc']['effectiveness_discriminator']['input_dim'],
                  args['model']['dmc']['effectiveness_discriminator']
['hidden_dim']),
        nn.LeakyReLU(0.1),
        nn.Linear(args['model']['dmc']['effectiveness_discriminator']
['hidden_dim'],
                  args['model']['dmc']['effectiveness_discriminator']['out_dim']),
    )

#GradientReversalLayer 是一种用于对抗性训练（Adversarial Training）的层，通常在多模态学习或域适应任务中使用。
#其主要作用是反转梯度，使得模型在训练过程中能够区分源域和目标域的数据。
#在反向传播过程中，它会将输入的梯度乘以一个负数 alpha。这样，当梯度从下游网络传递回上游网络时，其方向会被反转。
#个人目前理解就是为了那个max(min(Loss))吧
self.GRL = GradientReversalLayer(alpha=1.0)

#完备性检测器
self.completeness_check = nn.ModuleList([
        Transformer(num_frames=args['model']['dmc']['completeness_check']
['input_length'],
```

```
                    save_hidden=False,
                    token_len=args['model']['dmc']['completeness_check']
['token_length'],
                    dim=args['model']['dmc']['completeness_check']['input_dim'],
                    depth=args['model']['dmc']['completeness_check']['depth'],
                    heads=args['model']['dmc']['completeness_check']['heads'],
                    mlp_dim=args['model']['dmc']['completeness_check']
['hidden_dim']),

            nn.Sequential(
                nn.Linear(args['model']['dmc']['completeness_check']['hidden_dim'],
int(args['model']['dmc']['completeness_check']['hidden_dim']/2)),
                nn.LeakyReLU(0.1),
                nn.Linear(int(args['model']['dmc']['completeness_check']
['hidden_dim']/2), 1),
                nn.Sigmoid()),
            ])
```

加载数据：

```
train.py
#加载数据
dataLoader = MMDataLoader(args)
```

首先是获取初始数据集,然后人工设置缺失率missing_rate,然后根据缺失率设置缺失：

```
def __init_mosi(self):
    with open(self.dataPath, 'rb') as f:
        data = pickle.load(f)

    self.data = data

    self.text = data[self.mode]['text_bert'].astype(np.float32)
    self.vision = data[self.mode]['vision'].astype(np.float32)
    self.audio = data[self.mode]['audio'].astype(np.float32)

    self.rawText = data[self.mode]['raw_text']
    self.ids = data[self.mode]['id']
    self.labels = {
        #这里只取了多模态标签
        'M': data[self.mode][self.train_mode+'_labels'].astype(np.float32),
        'missing_rate_l': np.zeros_like(data[self.mode]
[self.train_mode+'_labels']).astype(np.float32),
        'missing_rate_a': np.zeros_like(data[self.mode]
[self.train_mode+'_labels']).astype(np.float32),
        'missing_rate_v': np.zeros_like(data[self.mode]
[self.train_mode+'_labels']).astype(np.float32),
    }

    #如果是sims则可以取到三个单模态标签
```

```
    if self.datasetName == 'sims':
        for m in "TAV":
            self.labels[m] = data[self.mode][self.train_mode+'_labels_'+m]

    self.audio_lengths = data[self.mode]['audio_lengths']
    self.vision_lengths = data[self.mode]['vision_lengths']
    self.audio[self.audio == -np.inf] = 0

    if self.mode == 'train':
        missing_rate = [np.random.uniform(size=(len(data[self.mode]
[self.train_mode+'_labels']), 1)) for i in range(3)]
        # missing_rate = [
        #     np.array([[0.5], [0.7], [0.2], [0.9]]),  # 文本模态的缺失率
        #     np.array([[0.6], [0.1], [0.8], [0.4]]),  # 音频模态的缺失率
        #     np.array([[0.3], [0.9], [0.5], [0.7]])  # 视频模态的缺失率
        # ]
        for i in range(3):
            sample_idx = random.sample([i for i in range(len(missing_rate[i]))],
int(len(missing_rate[i])/2))
            missing_rate[i][sample_idx] = 0
        # 对于文本模态：sample_idx = [0, 2]
        # 对于音频模态：sample_idx = [1, 3]
        # 对于视频模态：sample_idx = [0, 2]
        # missing_rate = [
        #     np.array([[0.0], [0.7], [0.0], [0.9]]),  # 文本模态的缺失率
        #     np.array([[0.6], [0.0], [0.8], [0.0]]),  # 音频模态的缺失率
        #     np.array([[0.0], [0.9], [0.0], [0.7]])  # 视频模态的缺失率
        # ]
        self.labels['missing_rate_l'] = missing_rate[0]
        self.labels['missing_rate_a'] = missing_rate[1]
        self.labels['missing_rate_v'] = missing_rate[2]
        #这样设置缺失率的话，就是这一批数据里，我有一半是missing_rate=0的，另一半的
missing_rate是一个随机值
        #也就是train里一半的样本是没有缺失的，另一半里是有缺失的，缺失程度missing_rate随机而
定，至于一个样本的missing_rate_i怎样影响样本i还没看到
    else:
        missing_rate = [self.missing_rate_eval_test * np.ones((len(data[self.mode]
[self.train_mode+'_labels']), 1)) for i in range(3)]
        self.labels['missing_rate_l'] = missing_rate[0]
        self.labels['missing_rate_a'] = missing_rate[1]
        self.labels['missing_rate_v'] = missing_rate[2]
        # self.labels['missing_rate_l'] = np.array([[0.5], [0.5], [0.5], [0.5]])
        # self.labels['missing_rate_a'] = np.array([[0.5], [0.5], [0.5], [0.5]])
        # self.labels['missing_rate_v'] = np.array([[0.5], [0.5], [0.5], [0.5]])

    self.text_m, self.text_length, self.text_mask, self.text_missing_mask =
self.generate_m(self.text[:,0,:], self.text[:,1,:], None,
                                                                    
missing_rate[0], self.missing_seed, mode='text')
    Input_ids_m = np.expand_dims(self.text_m, 1)
    Input_mask = np.expand_dims(self.text_mask, 1)
    Segment_ids = np.expand_dims(self.text[:,2,:], 1)
    self.text_m = np.concatenate((Input_ids_m, Input_mask, Segment_ids), axis=1)
```

```
    self.audio_m, self.audio_length, self.audio_mask, self.audio_missing_mask =
self.generate_m(self.audio, None, self.audio_lengths,
                                                                    
missing_rate[1], self.missing_seed, mode='audio')
    self.vision_m, self.vision_length, self.vision_mask, self.vision_missing_mask =
self.generate_m(self.vision, None, self.vision_lengths,
                                                                    
missing_rate[2], self.missing_seed, mode='vision')
```

根据缺失率missing_rate设置缺失的函数：

```
def generate_m(self, modality, input_mask, input_len, missing_rate, missing_seed,
mode='text'):

    if mode == 'text':
        input_len = np.argmin(input_mask, axis=1)
    elif mode == 'audio' or mode == 'vision':
        input_mask = np.array([np.array([1] * length + [0] * (modality.shape[1] -
length)) for length in input_len])
    np.random.seed(missing_seed)
    missing_mask = (np.random.uniform(size=input_mask.shape) >
missing_rate.repeat(input_mask.shape[1], 1)) * input_mask

    assert missing_mask.shape == input_mask.shape

    if mode == 'text':
        # CLS SEG Token unchanged.
        for i, instance in enumerate(missing_mask):
            instance[0] = instance[input_len[i] - 1] = 1

        modality_m = missing_mask * modality + (100 * np.ones_like(modality)) *
(input_mask - missing_mask) # UNK token: 100.
    elif mode == 'audio' or mode == 'vision':
        modality_m = missing_mask.reshape(modality.shape[0], modality.shape[1], 1) *
modality

    return modality_m, input_len, input_mask, missing_mask
```

定义优化器、学习率调度器、损失函数、评估指标：

```
#使用AdamW优化器，根据配置文件中的学习率和权重衰减参数进行初始化。
optimizer = torch.optim.AdamW(model.parameters(),
                              lr=args['base']['lr'],
                              weight_decay=args['base']['weight_decay'])

#根据配置文件中的参数获取学习率调度器。
scheduler_warmup = get_scheduler(optimizer, args)
```

```
#根据配置文件中的参数初始化多模态损失函数。
loss_fn = MultimodalLoss(args)

#定义评估指标
metrics = MetricsTop(train_mode = args['base']['train_mode']).getMetics(args['dataset']
['datasetName'])
```

整个LNLN的前向传播：

```
def forward(self, complete_input, incomplete_input):
    #完整数据
    vision, audio, language = complete_input
    #缺失数据
    vision_m, audio_m, language_m = incomplete_input

    b = vision_m.size(0)

    h_1_v = self.proj_v(vision_m)[:, :8]
    h_1_a = self.proj_a(audio_m)[:, :8]
    h_1_l = self.proj_l(self.bertmodel(language_m))[:, :8]

    #完备性检测器的输出
    feat_tmp = self.completeness_check[0](h_1_l)[:, :1].squeeze()
    w = self.completeness_check[1](feat_tmp) # completeness scores

    h_0_p = repeat(self.h_p, '1 n d -> b n d', b = b)
    #生成器的输出
    h_1_p = self.proxy_dominate_modality_generator(torch.cat([h_0_p, h_1_a, h_1_v],
dim=1))[:, :8]
    h_1_p = self.GRL(h_1_p)

    h_1_d = h_1_p * (1-w.unsqueeze(-1)) + h_1_l * w.unsqueeze(-1)

    h_hyper = repeat(self.h_hyper, '1 n d -> b n d', b = b)
    h_d_list = self.dmml[0](h_1_d)
    h_hyper = self.dmml[1](h_d_list, h_1_a, h_1_v, h_hyper)

    feat = self.dmml[2](h_hyper, h_d_list[-1])
    # Two ways to get the cls_output: using extra cls_token or using mean of all the
features
    # output = self.dmml[3](feat[:, 0])
    output = self.dmml[3](torch.mean(feat[:, 1:], dim=1))

    rec_feats, complete_feats, effectiveness_discriminator_out = None, None, None
    if (vision is not None) and (audio is not None) and (language is not None):
        # Reconstruction
        # for layer in self.reconstructor:
        rec_feat_a = self.reconstructor[0](h_1_a)
        rec_feat_v = self.reconstructor[1](h_1_v)
        rec_feat_l = self.reconstructor[2](h_1_l)
        rec_feats = torch.cat([rec_feat_a, rec_feat_v, rec_feat_l], dim=1)
```

```
        # Compute the complete features as the label of reconstruction 将完整的特征作为再现器的标签进行计算。
        complete_language_feat = self.proj_l(self.bertmodel(language))[:, :8]
        complete_vision_feat = self.proj_v(vision)[:, :8]
        complete_audio_feat = self.proj_a(audio)[:, :8]

        #! ! ! ! ! ! ! 检测器的输入，这里为啥是h_1_d啊，我看图和公式里面是h_1_p ! ! ! ! ! ! ! ! ! ! ! ! !
        #! ! ! ! ! ! ! 检测器的输入，这里为啥是h_1_d啊，我看图和公式里面是h_1_p ! ! ! ! ! ! ! ! ! ! ! ! !
        #! ! ! ! ! ! ! 检测器的输入，这里为啥是h_1_d啊，我看图和公式里面是h_1_p ! ! ! ! ! ! ! ! ! ! ! ! !
        effective_discriminator_input = rearrange(torch.cat([h_1_d, complete_language_feat]), 'b n d -> (b n) d')

        #检测器的输出
        effectiveness_discriminator_out = self.effective_discriminator(effective_discriminator_input)

        complete_feats = torch.cat([complete_audio_feat, complete_vision_feat, complete_language_feat], dim=1) # as the label of reconstruction

    #output是dmml模块的输出，w是完备性检测器的输出，effectiveness_discriminator_out是检测器的输出，rec_feats是再现器的输出结果（特征向量），complete_feats是以无人工设置缺失数据得到的特征向量
    return {'sentiment_preds': output,
            'w': w,
            'effectiveness_discriminator_out': effectiveness_discriminator_out,
            'rec_feats': rec_feats,
            'complete_feats': complete_feats}
```

计算损失函数：

```
    def forward(self, out, label):

        #完备性检测模块
        l_cc = self.MSE_Fn(out['w'], label['completeness_labels']) if out['w'] is not None else 0

        #对抗学习模块，这个地方比较迷惑，effectiveness_labels的设置我不理解
        l_adv = self.CE_Fn(out['effectiveness_discriminator_out'], label['effectiveness_labels']) if out['effectiveness_discriminator_out'] is not None else 0

        #再现器，这个模块我做冲突应该不需要吧,而且他实操和图里面不太一样
        l_rec = self.MSE_Fn(out['rec_feats'], out['complete_feats']) if out['rec_feats'] is not None and out['complete_feats'] is not None else 0

        #dmml模块
        l_sp = self.MSE_Fn(out['sentiment_preds'], label['sentiment_labels'])
```

```
        loss = self.alpha * l_cc + self.beta * l_adv + self.gamma * l_rec + self.sigma
* l_sp

        return {'loss': loss, 'l_sp': l_sp, 'l_cc': l_cc, \
                'l_adv': l_adv, 'l_rec': l_rec}
```

多模态情绪分析（MSA）方法可以根据建模方法分为基于上下文的MSA和基于噪声感知的MSA。

大部分以前的作品(Zadeh等人，2017年；蔡等人，2019年；迈等人，2020年；哈扎里卡等人，2020；梁等，20220；拉赫曼等，2020；余等，2021；韩等，2021；吕等，2021；杨等，2022；郭等，2022；张等，2023年）可分类为基于上下文的MSA。这一工作主要集中于通过分析模态内部或模式之间的上下文关系来学习统一的多模态表示。例如，Zadeh等人（2017）探索了使用笛卡尔乘积计算不同模式之间的关系。Tsai等人（2019）利用成对的变压器来建模不同模式之间的长期依赖关系。Yu等人（2021）提出为每个模态生成伪标签，以进一步挖掘不同模态之间的一致性和差异的信息。尽管有了这些进展，但基于上下文的方法在不同水平的噪声效应（例如，随机数据缺失）下通常是次优的。

最近的几项工作（Mittal等人，2020年；Yuan等人，2021,2024；李等人，2024）已经被提出来解决这个问题。例如，Mittal等人（2020）引入了一个模态检查步骤来区分无效和有效的模态，实现了更高的鲁棒性。Yuan等人（2024）提出在构建的“原始-噪声”实例对之间学习统一的联合表示。虽然在提高模型在噪声场景下的鲁棒性方面取得了一些进展，但目前还没有现有的方法提供全面和深入的比较分析。

相比之下，噪声感知MSA更注重感知和消除数据中存在的噪声。例如，Mittal等人（2020）设计了一个基于度量学习和规范相关分析（CCA）的模态检查模块，以识别具有更大噪声的模态。Yuan等人（2021）设计了一个特征重构网络，以预测序列中缺失信息的位置并进行重构。Yuan等人（2024）引入了对抗性学习（Goodfell等人，2014）来感知和生成更清晰的表征。

相比之下，噪声感知MSA更注重感知和消除数据中存在的噪声。例如，Mittal等人（2020）设计了一个基于度量学习和规范相关分析（CCA）的模态检查模块，以识别具有更大噪声的模态。Yuan等人（2021）设计了一个特征重构网络，以预测序列中缺失信息的位置并进行重构。Yuan等人（2024）引入了对抗性学习（Goodfell等人，2014）来感知和生成更清晰的表征。

麻了，一直报错内存不够啊，我试试数据集选小点，不行试试整个服务器跑：

```
Traceback (most recent call last):
  File "E:\yanyi\code of paper\LNLN-main\LNLN-main\train.py", line 180, in <module>
    main()
  File "E:\yanyi\code of paper\LNLN-main\LNLN-main\train.py", line 69, in main
    train(model, dataLoader['train'], optimizer, loss_fn, epoch, metrics)
  File "E:\yanyi\code of paper\LNLN-main\LNLN-main\train.py", line 103, in train
    out = model(complete_input, incomplete_input)
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
  File "E:\anaconda3\envs\envir3\Lib\site-packages\torch\nn\modules\module.py", line
1501, in _call_impl
    return forward_call(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "E:\yanyi\code of paper\LNLN-main\LNLN-main\models\lnln.py", line 140, in
forward
    print('self.bertmodel(language_m)',self.bertmodel(language_m))
                                        ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "E:\anaconda3\envs\envir3\Lib\site-packages\torch\nn\modules\module.py", line
1501, in _call_impl
    return forward_call(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "E:\yanyi\code of paper\LNLN-main\LNLN-main\models\bert.py", line 38, in forward
    last_hidden_states = self.model(input_ids=input_ids,
                         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "E:\anaconda3\envs\envir3\Lib\site-packages\torch\nn\modules\module.py", line
1501, in _call_impl
    return forward_call(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "E:\anaconda3\envs\envir3\Lib\site-
packages\transformers\models\bert\modeling_bert.py", line 1142, in forward
    encoder_outputs = self.encoder(
                      ^^^^^^^^^^^^^
  File "E:\anaconda3\envs\envir3\Lib\site-packages\torch\nn\modules\module.py", line
1501, in _call_impl
    return forward_call(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "E:\anaconda3\envs\envir3\Lib\site-
packages\transformers\models\bert\modeling_bert.py", line 695, in forward
    layer_outputs = layer_module(
                    ^^^^^^^^^^^^
  File "E:\anaconda3\envs\envir3\Lib\site-packages\torch\nn\modules\module.py", line
1501, in _call_impl
    return forward_call(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "E:\anaconda3\envs\envir3\Lib\site-
packages\transformers\models\bert\modeling_bert.py", line 627, in forward
    layer_output = apply_chunking_to_forward(
                   ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "E:\anaconda3\envs\envir3\Lib\site-packages\transformers\pytorch_utils.py", line
248, in apply_chunking_to_forward
    return forward_fn(*input_tensors)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "E:\anaconda3\envs\envir3\Lib\site-
packages\transformers\models\bert\modeling_bert.py", line 639, in feed_forward_chunk
    intermediate_output = self.intermediate(attention_output)
                          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "E:\anaconda3\envs\envir3\Lib\site-packages\torch\nn\modules\module.py", line
1501, in _call_impl
    return forward_call(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "E:\anaconda3\envs\envir3\Lib\site-
packages\transformers\models\bert\modeling_bert.py", line 539, in forward
```

```
    hidden_states = self.dense(hidden_states)
                    ^^^^^^^^^^^^^^^^^^^^^^^^^
  File "E:\anaconda3\envs\envir3\Lib\site-packages\torch\nn\modules\module.py", line
1501, in _call_impl
    return forward_call(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "E:\anaconda3\envs\envir3\Lib\site-packages\torch\nn\modules\linear.py", line
114, in forward
    return F.linear(input, self.weight, self.bias)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
torch.cuda.OutOfMemoryError: CUDA out of memory. Tried to allocate 20.00 MiB (GPU 0;
12.00 GiB total capacity; 1.27 GiB already allocated; 9.64 GiB free; 1.28 GiB reserved
in total by PyTorch) If reserved memory is >> allocated memory try setting
max_split_size_mb to avoid fragmentation.  See documentation for Memory Management and
PYTORCH_CUDA_ALLOC_CONF

进程已结束,退出代码1
```

调成batch_size=16 终于能运行了，复现实验结果跑了2h，复现结果和论文里差不多。