BLIP

👫 **论文：BLIP: Bootstrapping Language-Image Pre-training for Unified Vision-Language Understanding and Generation**

论文链接：https://arxiv.org/pdf/2201.12086

可以参考的博客：https://blog.csdn.net/m0_51976564/article/details/134356373，https://zhuanlan.zhihu.com/p/28392731664，https://zhuanlan.zhihu.com/p/640887802，https://blog.csdn.net/wl1780852311/article/details/148871284

可以参考的视频：https://www.bilibili.com/video/BV1fA411Z772/?spm_id_from=333.337.search-card.all.click

1. BLIP 简介

🐞 **BLIP（Bootstrapping Language-Image Pre-training）** 是一种统一的**视觉-语言预训练框架**，旨在解决现有的**视觉-语言预训练（VLP）**模型在**理解任务与生成任务**上的局限性，以及网页**图像-文本对噪声问题**

实验表明，BLIP 在图像-文本检索（平均召回率 @1 提升 2.7%）、图像 captioning（CIDEr 提升 2.8%）、VQA（分数提升 1.6%） 等任务上达到 SOTA，并在零样本迁移至视频-语言任务中表现优异

1.1 BLIP 的背景意义与动机

🛫 模型能力分化问题

- 传统 VLP 模型**要么擅长理解（Encoder 模型），要么擅长生成（如 Encoder-Decoder 结构）**，但两者未能统一
- Encoder 模型难迁移至生成任务（如图像 captioning），Encoder-Decoder 模型在多模态检索任务中表现不佳

🛫 噪声网络语料影响效率

- 网页爬取的**图像-文本对存在大量噪声（文本与图像内容不匹配）**，虽通过扩大数据量提升性能，但噪声仍然降低学习效率，影响训练质量及下游表现

- **BLIP 的目标**：在融合理解与生成能力的同时，通过 **Captioner + Filter（CapFilt）** 机制清洗和增强噪声数据，提高预训练样本质量

1.2 BLIP 的核心创新点

🍽️ 多模态混合编码器-解码器架构（MED）

- 支持单模态编码、图像接地文本编码和图像接地文本解码三种功能，通过图像-文本对比学习、匹配和条件语言建模三种目标联合预训练

🙌 CapFilt（Captioning and Filtering） 数据增强方法

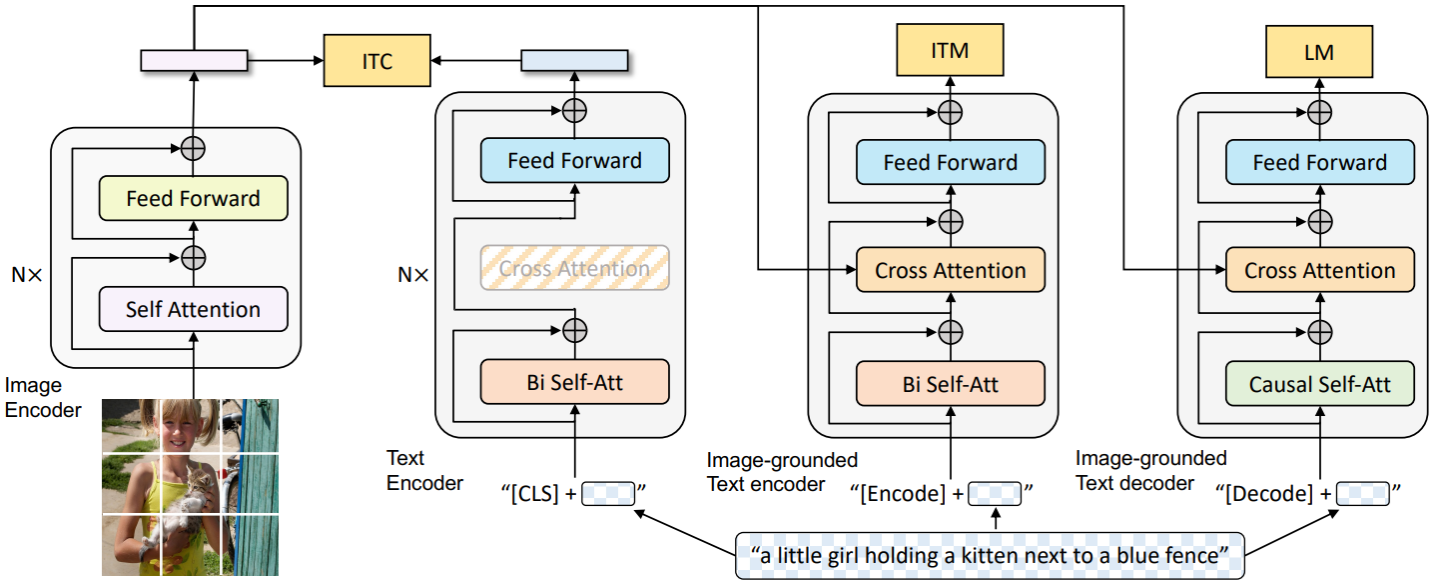
- 通过生成器生成合成字幕并结合过滤器移除噪声，提升数据质量
- Captioner 生成图像的文本标注，Filter去除标注中的噪声，提升数据质量通过不断迭代生成和过滤，BLIP 能够从有限的标注数据中扩展出更多高质量的训练数据。

📚 BLIP 的核心思想是**通过 Bootstrapping 方法，利用 Captioner-Filter 机制生成高质量的文本标注**，从而提高数据的质量和数量。在 BLIP 中，Bootstrapping 体现在 Captioner-Filter 机制中

> Bootstrapping 是一种统计估计方法，通过对观测数据进行再抽样，进而对总体的分布特性进行统计推断。在机器学习中，Bootstrapping 常用于小数据集场景，通过有放回地抽样生成多个训练集，从而引入随机性，增加模型的多样性，提高泛化能力

2. BLIP 方法细节

2.1 BLIP 模型架构

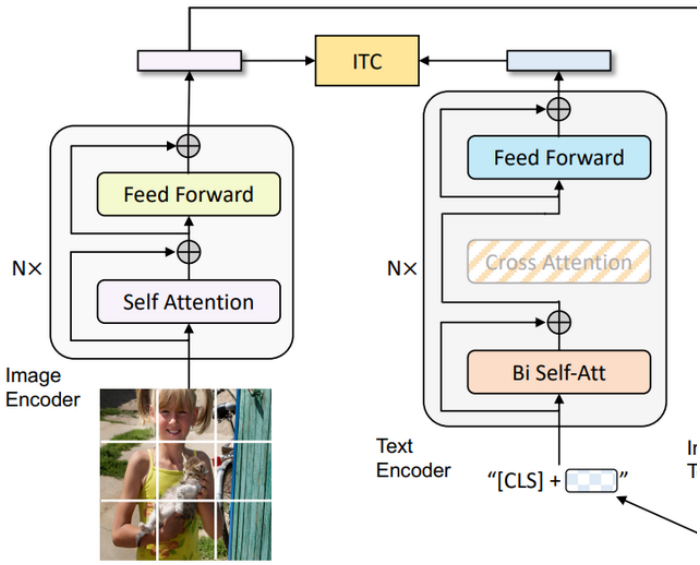


BLIP 核心是多模态混合编解码器结构（Multi-modal mixture of Encoder-Decoder, MED），包括三个模块：Unimodal encoder，Image-grounded text encoder，Image-grounded text decoder

2.1.1 Unimodal Encoder

Image Encoder

- 使用**基于 Transformer 的 ViT架构**
- 将输入图像分割为多个 patch，编码为一系列 Image Embedding
- 使用 [CLS] token 表示全局图像特征
- 目标：**提取图像特征，用于对比学习（类似于 CLIP 中的 Image Encoder）**

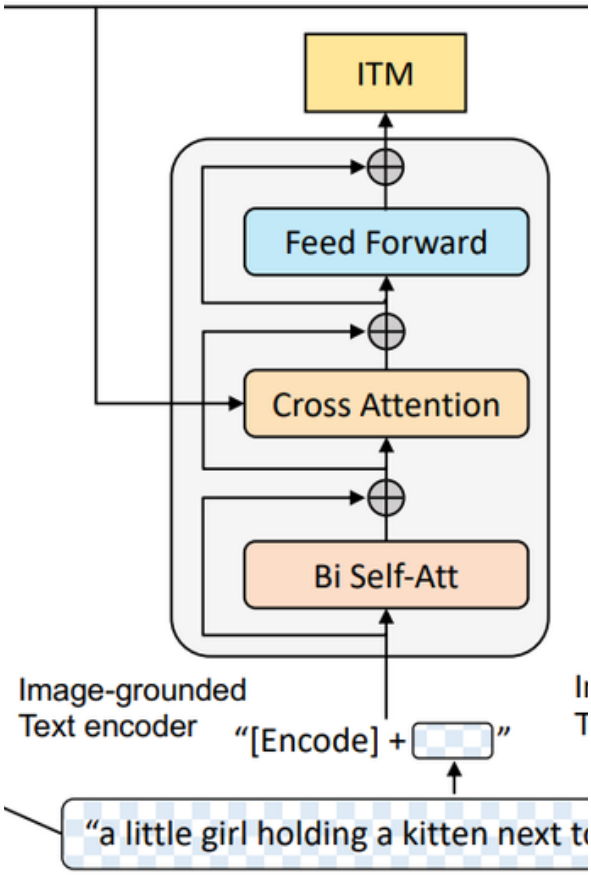


Text Encoder

- **基于 BERT 架构**
- **输入：**文本开头添加 [CLS] token 以表示整个句子
- 目标：**提取文本特征，用于对比学习（类似于 CLIP 中的 Text Encoder）**

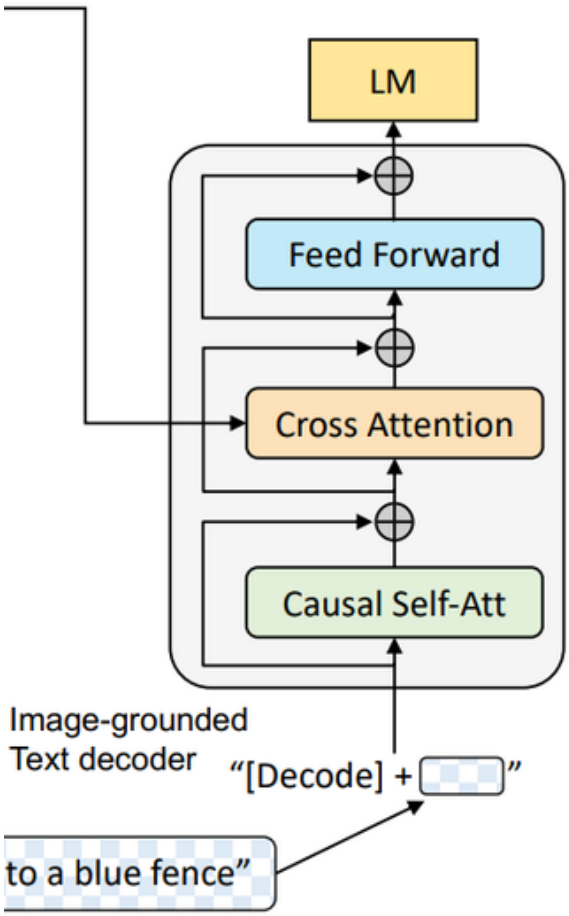
这一阶段的训练目标是对齐图像和文本的特征空间，通过将 **Image Encoder 和 Text Encoder 输出的 embedding 做对比学习（Image Text Contrastive learning，ITC）**

2.1.2 Image-grounded Text Encoder



- **结构：**在 Text Encoder 的 **双向 self-attention 层和前馈网络之间添加交叉注意（Cross-Attention, CA）层，用于注入视觉信息**
- **输入：文本开头添加 [Encode] token 以标识特定任务**，然后通过**双向自注意力**之后，与图像 embedding 在**交叉注意力**中进行跨模态学习，最后经过 FFN 得到输出
- **输出：**取 [Encode] token 对应的输出 embedding 用作图像-文本对的多模态表示
- **目标：提取文本特征并与图像特征对齐**（比 CLIP 更精细化的 Text-Image 对齐）。对文本加入**跨模注意力，实现图文匹配（Image Text Matchong，ITM）**，区分正负图像-文本对
- **核心：**插入交叉注意力层融合视觉信息，用于匹配任务

2.1.3 Image-grounded Text Encoder



- **结构：**将 Image-grounded Text Encoder 的 **双向 self-attention 层替换为因果自注意力（Causal Self-Attention）层**
- **输入：**文本**开头和结尾分别添加 `[Decode]` token 和 `[EOS]` token**，标识序列的开始和结束。这里将图像 **embedding 与当前时刻之前的文本的表征做交叉注意力，然后预测下一个 token**，生成图像描述
- 因果注意力的作用便是防止 Decoder 获得当前 token 之后的信息，一般使用 Casul Mask 实现

- **目标：**生成符合图像和文本特征的文本描述（CLIP 不具备此功能）。此阶段主要遵循自回归的**语言建模（Language Modeling，LM）**，进行 Next Token Prediction，以自回归方式生成图像描述
- **核心：**采用因果自注意力，用于文本生成任务

2.1.4 模块对比

模块名称	架构	功能	类比
Image Encoder	基于 ViT（Vision Transformer）	提取图像特征，用于对比学习	类似于 CLIP 的 Image Encoder
Text Encoder	基于 BERT	提取文本特征，用于对比学习	类似于 CLIP 的 Text Encoder
Image-grounded Text Encoder	在 BERT 基础上添加交叉注意力层	提取文本特征并与图像特征对齐	更精细化的 Text-Image 对齐
Image-grounded Text Decoder	self-attention 替换为因果自注意力层	生成符合图像和文本特征的文本描述	CLIP 不具备此功能

🀄 值得注意的是，上图的**三大模块颜色相同的部分之间共享参数**，训练 ITC 任务的时候也完成了 ITM 的部分训练。同时每个图像-文本对只需通过计算量较大的**视觉 Transformer 进行一次前向传播**，再通过**文本 Transformer 进行三次前向传播**

2.2 BLIP 预训练方法

👨‍🍳 BLIP在预训练过程中，联合优化三个目标函数，其中包括**两个基于理解的目标**和一个**基于生成的目标**

2.2.1 图文对比损失

🦛 图文对比损失（Image-Text Contrastive Loss, ITC），基于理解

- **目标：**对齐图像和文本的特征空间
- **方法：**

- 最大化正样本图像-文本对的相似度
- 最小化负样本图像-文本对的相似度
- 使用动量编码器（Momentum Encoder）生成伪标签以辅助训练

> Momentum Encoder 出自论文：Vision and language representation learning with momentum distillation，是一种用于稳定训练、提升特征一致性的技术，尤其在对比学习（如图像-文本对比损失 ITC）中广泛应用。动量编码器本质上是一个 “缓慢更新” 的模型副本，与主编码器（Main Encoder）结构完全相同，但参数更新方式不同，不直接通过梯度更新，而是通过主编码器的参数 “平滑过渡” 更新，公式大致为：
> `动量编码器参数 = 动量系数 × 动量编码器旧参数 + (1−动量系数) × 主编码器新参数`

- **作用：**用于**训练 Image Encoder 和 Text Encoder**

2.2.2 图文匹配损失

图文匹配损失（Image-Text Matching Loss, ITM），基于理解

- **目标：**实现**视觉和语言之间的细粒度对齐**
- **方法：**
  - 二分类任务，利用一个图像-文本匹配头（一个线性层），根据图像-文本对的多模态特征预测图像-文本对是正样本还是负样本
  - 使用 **hard negative mining** 技术更好地捕捉负样本信息

> 难负样本（Hard Negatives）：负样本中与正样本 “非常相似” 的样本，模型很难将其与正样本区分开。例如：描述 “猫爬树” 的文本，对应的负样本图像是 “猫卧在地上”（语义接近，比 “汽车” 这类负样本更难区分）；在检索任务中，与查询结果排名较靠前但实际不匹配的样本

- **作用：**用于**训练 Image-grounded Text Encoder 和 Image Encoder（图像特征来源）**

2.2.3 语言建模损失

语言建模损失（Language Modeling Loss, LM），基于生成

- **目标：**生成**图像的文本描述**
- **方法：**
  - 通过优化交叉熵损失函数，训练模型以自回归的方式最大化下一个输出文本 token 的概率，输出的文本为图像的描述，即 caption 信息

- 使用 0.1 的标签平滑计算损失
- 与视觉-语言预训练（VLP）中广泛使用的掩码语言建模（MLM）损失相比，语言建模损失（LM）使模型具备将视觉信息转化为连贯字幕的泛化能力。（从 BERT 到 GPT 的转变）

- **作用：**用于训练 **Image-grounded Text Decoder 和 Image Encoder（图像特征来源）**

损失函数对比			
损失函数	目标	方法	训练模块
图文对比损失（ITC）	对齐图像和文本的特征空间	最大化正样本相似度，最小化负样本相似度，使用动量编码器生成伪标签	Image Encoder 和 Text Encoder
图文匹配损失（ITM）	实现视觉和语言之间的细粒度对齐	通过二分类任务预测正负样本，使用 hard negative mining 技术捕捉负样本信息	Image-grounded Text Encoder
语言建模损失（LM）	生成图像的文本描述	通过交叉熵损失函数，以自回归方式最大化文本概率，使用标签平滑计算损失	Image-grounded Text Decoder

2.3 BLIP CapFilt 机制

2.3.1 CapFilt 的背景

🤦

- 由于**标注成本过高，高质量的人工标注图像-文本对 `{(I_h, T_h)}` 数量有限**（例如 COCO 数据集）
- 近年许多研究利用了数量多得多的，从网页中自动收集的图像与替代文本对 `{(I_w, T_w)}`。但是，**这些替代文本（alt-text）往往无法准确描述图像的视觉内容，使其成为一种噪声信号**，对于学习视觉-语言对齐来说并非最优选择

2.3.2 CapFilt 的方法

🤹

- **核心目的：从噪声网页数据中提取高质量图像-文本对**
- **步骤：Caption and Filtering**

a. **基于预训练的 MED 微调两个模块：**

- **生成器（Captioner）**：为网页图像生成合成字幕 caption（用 COCO 数据集微调），通过模型生成描述，增加样本多样性
- **过滤器（Filter）**：判断文本与图像是否匹配（用 COCO 数据集微调），通过模型评估去除噪声 caption，确保质量

b. **整合数据：**过滤原始网页文本和合成文本中的噪声，结合**人工标注数据（如 COCO）**形成新训练集。该机制使训练集在原始网络语料基础上，更加干净且多样，而非盲目扩增

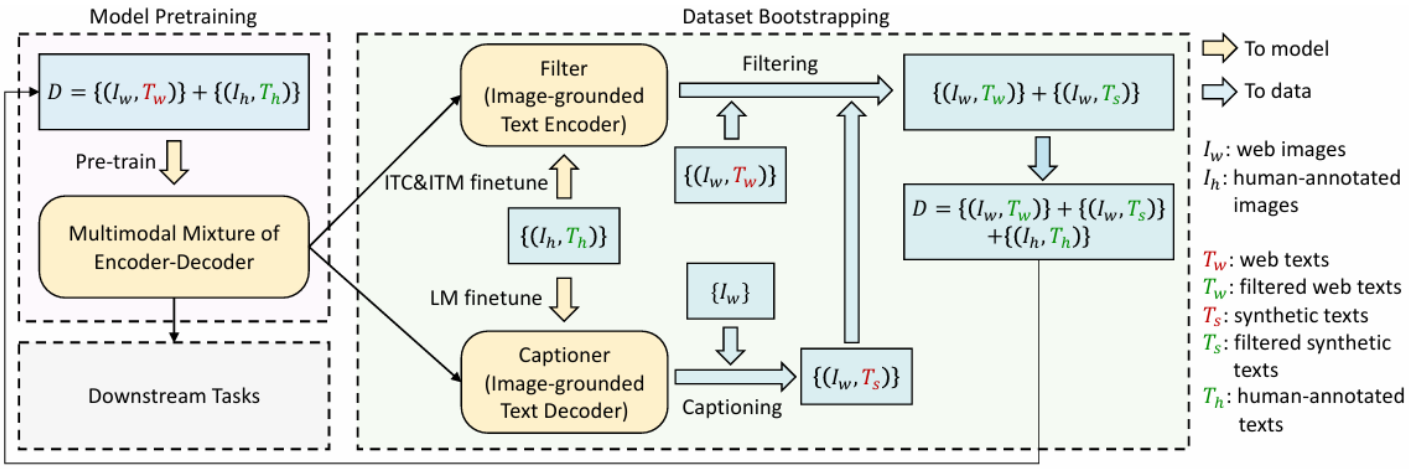


Figure 3. Learning framework of BLIP. We introduce a captioner to produce synthetic captions for web images, and a filter to remove noisy image-text pairs. The captioner and filter are initialized from the same pre-trained model and finetuned individually on a small-scale human-annotated dataset. The bootstrapped dataset is used to pre-train a new model.

2.3.3 CapFilt 方法详解

🎦 字幕器（Captioner）

- **功能：**基于 **Image-grounded Text Decoder**，生成给定图像的文本描述
- **训练：**在 COCO 数据集上使用 **LM 损失函数进行微调**
- **输出：**给定网络图片 I_w，生成字幕 T_w

🧹 过滤器（Filter）

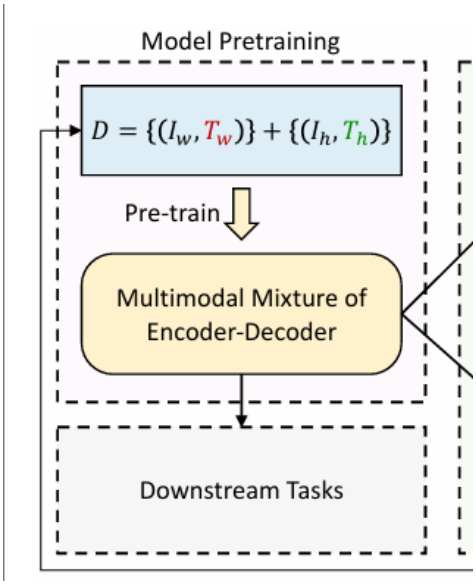
- **功能：**基于 **Image-grounded Text Encoder**，去除文本噪声
- **训练：**在 COCO 数据集上**使用 ITC 和 ITM 损失函数进行微调**
- **方法：**通过比对文本和图像的匹配情况，**删除原始 Web 文本 T_w 和合成文本 T_s 中的噪声**

CapFilt 详细流程

1️⃣ 模型预训练

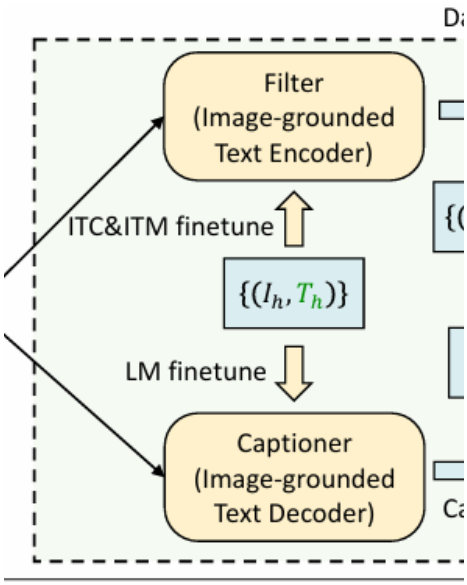
基于 MED 框架训练 BLIP，数据包含**含有噪声的网络数据** `{(I_w, T_w)}`

以及**人工标注的高质量数据** `{(I_h, T_h)}`



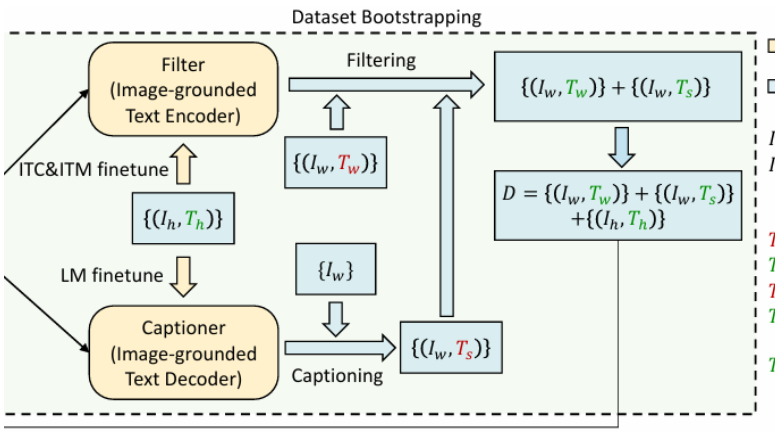
2️⃣ 模型微调

Captioner 和 Filter 都是**在预训练完成后的 MED 模型上初始化**的，然后在 COCO 数据集上微调 Captioner 和 Filter，分别使用 LM 目标和 ITC+ITM 目标训练



3️⃣ 数据过滤

使用 Captioner 对输入的图像 `{I_w}` 生成字幕 caption `{T_s}`，然后使用 Filter 判断网络数据 `{(I_w, T_w)}` 和生成数据 `{(I_w, T_s)}` 是否匹配，**留下匹配的并与人工标注的合并，形成新的高质量数据集** `{(I_h, T_h)+(I_w, T_w)+(I_w, T_s)}`，送给MED继续预训练，使用新的数据量更大且更干净的数据，实现 Bootstrap训练



3. BLIP 实验效果

3.1 CapFilt 的有效性

预训练数据	生成器（C）	过滤器（F）	图像-文本检索（COCO，TR@1）	图像 captioning（COCO，CIDEr）
14M 图像	无	无	78.4	127.8
14M 图像	有	有	80.6	129.7

- 生成器与过滤器结合时效果最佳，14M 图像上检索 TR@1 提升 2.2%，CIDEr 提升 1.9%
- 更大数据集（129M）和模型（ViT-L）进一步提升性能，验证可扩展性

3.2 各任务 SOTA 表现

- **图像-文本检索：**14M 图像上，COCO 数据集平均 recall@1 较 ALBEF 提升 2.7%；零样本迁移至 Flickr30K，TR@1 达 94.8%
- **图像 captioning：**COCO 数据集 CIDEr 达 129.7（14M 图像），NoCaps 零样本 CIDEr 达 105.1
- **VQA：**14M 图像上测试集分数达 77.62，较 ALBEF 提升 1.6%
- **零样本视频-语言任务：**文本-视频检索 R@1 达 43.3（超微调模型 12.4%），视频 QA 准确率达 19.2（MSRVTT）

4. BLIP 代码

官方源码：https://github.com/salesforce/BLIP，主要看models文件夹下的 med.py，blip_pretrain.py 两个文件

4.1 med.py

- `BertSelfAttention` 类支持跨模态自注意力机制，通过 `is_cross_attention` 判断

BertSelfAttention代码

```
class BertSelfAttention(nn.Module):
    def __init__(self, config, is_cross_attention):
        super().__init__()
        self.config = config
        # 检查 hidden_size 是否能被 head 数整除
        if config.hidden_size % config.num_attention_heads != 0 and not hasattr(config, "embedding_size"):
            raise ValueError("hidden size 不是 attention head 数的整数倍")
```

```
8
9           # 多头注意力配置
10          self.num_attention_heads = config.num_attention_heads
11          self.attention_head_size = int(config.hidden_size /
    config.num_attention_heads)
12          self.all_head_size = self.num_attention_heads *
    self.attention_head_size
13
14          # 查询向量投影层
15          self.query = nn.Linear(config.hidden_size, self.all_head_size)
16
17          # 判断是 cross-attention（来自视觉）还是自注意力
18          if is_cross_attention:
19              self.key = nn.Linear(config.encoder_width, self.all_head_size)
20              self.value = nn.Linear(config.encoder_width, self.all_head_size)
21          else:
22              self.key = nn.Linear(config.hidden_size, self.all_head_size)
23              self.value = nn.Linear(config.hidden_size, self.all_head_size)
24
25          self.dropout = nn.Dropout(config.attention_probs_dropout_prob)
26          self.position_embedding_type = getattr(config,
    "position_embedding_type", "absolute")
27
28          # 支持相对位置编码
29          if self.position_embedding_type in ["relative_key",
    "relative_key_query"]:
30              self.max_position_embeddings = config.max_position_embeddings
31              self.distance_embedding = nn.Embedding(2 *
    config.max_position_embeddings - 1, self.attention_head_size)
32
33          self.save_attention = False  # 可开启保存注意力图（用于可视化等）
34
35      # 注册保存注意力梯度
36      def save_attn_gradients(self, attn_gradients):
37          self.attn_gradients = attn_gradients
38
39      def get_attn_gradients(self):
40          return self.attn_gradients
41
42      def save_attention_map(self, attention_map):
43          self.attention_map = attention_map
44
45      def get_attention_map(self):
46          return self.attention_map
47
48      # 将张量调整为 [batch, head, seq, head_dim] 格式
49      def transpose_for_scores(self, x):
```

```
        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)

    def forward(
        self, hidden_states, attention_mask=None, head_mask=None,
        encoder_hidden_states=None, encoder_attention_mask=None,
        past_key_value=None, output_attentions=False,
    ):
        # 将输入投影为 query
        mixed_query_layer = self.query(hidden_states)

        is_cross_attention = encoder_hidden_states is not None

        if is_cross_attention:
            # cross-attention 情况下使用来自 encoder 的 key/value
            key_layer = self.transpose_for_scores(self.key(encoder_hidden_states))
            value_layer = self.transpose_for_scores(self.value(encoder_hidden_states))
            attention_mask = encoder_attention_mask
        elif past_key_value is not None:
            # 支持缓存历史 key/value（如生成任务中）
            key_layer = self.transpose_for_scores(self.key(hidden_states))
            value_layer = self.transpose_for_scores(self.value(hidden_states))
            key_layer = torch.cat([past_key_value[0], key_layer], dim=2)
            value_layer = torch.cat([past_key_value[1], value_layer], dim=2)
        else:
            key_layer = self.transpose_for_scores(self.key(hidden_states))
            value_layer = self.transpose_for_scores(self.value(hidden_states))

        query_layer = self.transpose_for_scores(mixed_query_layer)

        past_key_value = (key_layer, value_layer)

        # 计算注意力分数（Q·K^T）
        attention_scores = torch.matmul(query_layer, key_layer.transpose(-1, -2))

        # 相对位置编码加分数
        if self.position_embedding_type in ["relative_key", "relative_key_query"]:
            seq_length = hidden_states.size()[1]
            position_ids_l = torch.arange(seq_length, dtype=torch.long, device=hidden_states.device).view(-1, 1)
```

```
            position_ids_r = torch.arange(seq_length, dtype=torch.long, device=hidden_states.device).view(1, -1)
            distance = position_ids_l - position_ids_r
            positional_embedding = self.distance_embedding(distance + self.max_position_embeddings - 1)
            positional_embedding = positional_embedding.to(dtype=query_layer.dtype)  # 兼容混合精度

            if self.position_embedding_type == "relative_key":
                attention_scores += torch.einsum("bhld,lrd->bhlr", query_layer, positional_embedding)
            elif self.position_embedding_type == "relative_key_query":
                attention_scores += torch.einsum("bhld,lrd->bhlr", query_layer, positional_embedding)
                attention_scores += torch.einsum("bhrd,lrd->bhlr", key_layer, positional_embedding)

        # 缩放注意力分数
        attention_scores = attention_scores / math.sqrt(self.attention_head_size)

        # 应用 attention mask（一般为 -inf 位置，避免关注 pad）
        if attention_mask is not None:
            attention_scores += attention_mask

        # softmax 转换为注意力权重
        attention_probs = nn.Softmax(dim=-1)(attention_scores)

        # 如果设置保存注意力图，就注册 hook 记录梯度
        if is_cross_attention and self.save_attention:
            self.save_attention_map(attention_probs)
            attention_probs.register_hook(self.save_attn_gradients)

        # dropout，随机 mask 掉一些 token
        attention_probs_dropped = self.dropout(attention_probs)

        # 如果提供了 head mask，对每个注意力头加权
        if head_mask is not None:
            attention_probs_dropped = attention_probs_dropped * head_mask

        # 上下文向量：softmax(Q·K^T)·V
        context_layer = torch.matmul(attention_probs_dropped, value_layer)

        # 调整维度回原始格式
        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)

        # 返回值格式：(context, [attention_probs], past_key_value)
        outputs = (context_layer, attention_probs) if output_attentions else (context_layer,)
        outputs = outputs + (past_key_value,)
        return outputs

```

- `BertSelfAttention` 类支持跨模态自注意力机制，通过 `is_cross_attention` 判断

4.2 blip_pretrain.py

- `__init__` 类

初始化代码

```
class BLIP_Pretrain(nn.Module):
    def __init__(self,
                 med_config='configs/bert_config.json',  # 文本编码器和解码器的配置文件
                 image_size=224,                         # 输入图像大小
                 vit='base',                             # 使用的 ViT 模型规模（base 或 large）
                 vit_grad_ckpt=False,                    # 是否启用梯度检查点
                 vit_ckpt_layer=0,                       # ViT 中启用检查点的层数
                 embed_dim=256,                          # 图文共同嵌入空间维度
                 queue_size=57600,                       # 动量队列长度（用于对比学习）
                 momentum=0.995):                        # 动量编码器更新比例
        super().__init__()

        # 初始化视觉编码器（ViT）及其输出维度
        self.visual_encoder, vision_width = create_vit(vit, image_size, vit_grad_ckpt, vit_ckpt_layer, 0)

        # 加载视觉编码器预训练参数（来自 DeiT）
        if vit == 'base':
            checkpoint = torch.hub.load_state_dict_from_url(
                url="https://dl.fbaipublicfiles.com/deit/deit_base_patch16_224-b5f2ef4d.pth",
                map_location="cpu", check_hash=True)
            state_dict = checkpoint["model"]
            msg = self.visual_encoder.load_state_dict(state_dict, strict=False)
        elif vit == 'large':
            from timm.models.helpers import load_custom_pretrained
```

```
            from timm.models.vision_transformer import default_cfgs
            load_custom_pretrained(self.visual_encoder,
default_cfgs['vit_large_patch16_224_in21k'])

        # 初始化 tokenizer（BERT）
        self.tokenizer = init_tokenizer()

        # 构造文本编码器 BertModel（不含 pooling）
        encoder_config = BertConfig.from_json_file(med_config)
        encoder_config.encoder_width = vision_width  # 跨模态融合用
        self.text_encoder = BertModel.from_pretrained('bert-base-uncased',
config=encoder_config, add_pooling_layer=False)
        self.text_encoder.resize_token_embeddings(len(self.tokenizer))  # 动态调
整词表大小

        # 定义文本嵌入维度
        text_width = self.text_encoder.config.hidden_size

        # 图文编码投影到同一嵌入空间
        self.vision_proj = nn.Linear(vision_width, embed_dim)
        self.text_proj = nn.Linear(text_width, embed_dim)

        # 图文匹配分类器（2类）
        self.itm_head = nn.Linear(text_width, 2)

        # 构建动量编码器（结构与主模型相同）
        self.visual_encoder_m, _ = create_vit(vit, image_size)
        self.vision_proj_m = nn.Linear(vision_width, embed_dim)
        self.text_encoder_m = BertModel(config=encoder_config,
add_pooling_layer=False)
        self.text_proj_m = nn.Linear(text_width, embed_dim)

        self.model_pairs = [
            [self.visual_encoder, self.visual_encoder_m],
            [self.vision_proj, self.vision_proj_m],
            [self.text_encoder, self.text_encoder_m],
            [self.text_proj, self.text_proj_m]
        ]
        self.copy_params()  # 同步初始化参数

        # 初始化图文动量队列
        self.register_buffer("image_queue", torch.randn(embed_dim, queue_size))
        self.register_buffer("text_queue", torch.randn(embed_dim, queue_size))
        self.register_buffer("queue_ptr", torch.zeros(1, dtype=torch.long))

        # 归一化嵌入向量
        self.image_queue = F.normalize(self.image_queue, dim=0)
```

```
68          self.text_queue = F.normalize(self.text_queue, dim=0)
69
70          # 设置队列大小与动量参数
71          self.queue_size = queue_size
72          self.momentum = momentum
73          self.temp = nn.Parameter(0.07 * torch.ones([]))   # 对比学习温度
74
75          # 创建文本解码器（用于图文生成）
76          decoder_config = BertConfig.from_json_file(med_config)
77          decoder_config.encoder_width = vision_width
78          self.text_decoder = BertLMHeadModel.from_pretrained('bert-base-
    uncased', config=decoder_config)
79          self.text_decoder.resize_token_embeddings(len(self.tokenizer))
80          tie_encoder_decoder_weights(self.text_encoder, self.text_decoder.bert,
    '', '/attention')  # encoder-decoder 权重绑定
81
```

- `forward` 类，先后计算 ICT、ITM 和 LM loss

ICT、ITM 和 LM loss

```
1       def forward(self, image, caption, alpha):
2           # 限制温度值在 [0.001, 0.5] 之间
3           with torch.no_grad():
4               self.temp.clamp_(0.001, 0.5)
5
6           # 图像编码 → [CLS] 特征
7           image_embeds = self.visual_encoder(image)
8           image_atts = torch.ones(image_embeds.size()[:-1],
    dtype=torch.long).to(image.device)
9           image_feat = F.normalize(self.vision_proj(image_embeds[:, 0, :]),
    dim=-1)
10
11          # 文本编码
12          text = self.tokenizer(caption, padding='max_length', truncation=True,
    max_length=30, return_tensors="pt").to(image.device)
13          text_output = self.text_encoder(text.input_ids,
    attention_mask=text.attention_mask, return_dict=True, mode='text')
14          text_feat =
    F.normalize(self.text_proj(text_output.last_hidden_state[:, 0, :]), dim=-1)
15
16          # -------- 动量编码器提取特征 + 构造对比目标 --------
17          with torch.no_grad():
18              self._momentum_update()
19
20              image_embeds_m = self.visual_encoder_m(image)
```

```
            image_feat_m = F.normalize(self.vision_proj_m(image_embeds_m[:, 0, :]), dim=-1)
            image_feat_all = torch.cat([image_feat_m.t(), self.image_queue.clone().detach()], dim=1)

            text_output_m = self.text_encoder_m(text.input_ids, attention_mask=text.attention_mask, return_dict=True, mode='text')
            text_feat_m = F.normalize(self.text_proj_m(text_output_m.last_hidden_state[:, 0, :]), dim=-1)
            text_feat_all = torch.cat([text_feat_m.t(), self.text_queue.clone().detach()], dim=1)

            sim_i2t_m = image_feat_m @ text_feat_all / self.temp
            sim_t2i_m = text_feat_m @ image_feat_all / self.temp

            # 构造软目标标签
            sim_targets = torch.zeros(sim_i2t_m.size()).to(image.device)
            sim_targets.fill_diagonal_(1)

            sim_i2t_targets = alpha * F.softmax(sim_i2t_m, dim=1) + (1 - alpha) * sim_targets
            sim_t2i_targets = alpha * F.softmax(sim_t2i_m, dim=1) + (1 - alpha) * sim_targets

        # -------- 图文对比损失（ITC） --------
        sim_i2t = image_feat @ text_feat_all / self.temp
        sim_t2i = text_feat @ image_feat_all / self.temp

        loss_i2t = -torch.sum(F.log_softmax(sim_i2t, dim=1) * sim_i2t_targets, dim=1).mean()
        loss_t2i = -torch.sum(F.log_softmax(sim_t2i, dim=1) * sim_t2i_targets, dim=1).mean()
        loss_ita = (loss_i2t + loss_t2i) / 2  # 图文对比学习损失

        self._dequeue_and_enqueue(image_feat_m, text_feat_m)  # 更新队列

        # -------- 图文匹配任务（ITM） --------
        encoder_input_ids = text.input_ids.clone()
        encoder_input_ids[:, 0] = self.tokenizer.enc_token_id  # 替换 [CLS] 为 [ENC]

        # 正样本编码（图文匹配）
        bs = image.size(0)
        output_pos = self.text_encoder(encoder_input_ids, attention_mask=text.attention_mask,
                                       encoder_hidden_states=image_embeds, encoder_attention_mask=image_atts,
```

```
                                        return_dict=True)

        with torch.no_grad():
            weights_t2i = F.softmax(sim_t2i[:, :bs], dim=1) + 1e-4
            weights_t2i.fill_diagonal_(0)
            weights_i2t = F.softmax(sim_i2t[:, :bs], dim=1) + 1e-4
            weights_i2t.fill_diagonal_(0)

        # 采样负样本（图像）
        image_embeds_neg = [image_embeds[torch.multinomial(weights_t2i[b],
1).item()] for b in range(bs)]
        image_embeds_neg = torch.stack(image_embeds_neg, dim=0)

        # 采样负样本（文本）
        text_ids_neg = [encoder_input_ids[torch.multinomial(weights_i2t[b],
1).item()] for b in range(bs)]
        text_atts_neg = [text.attention_mask[torch.multinomial(weights_i2t[b],
1).item()] for b in range(bs)]
        text_ids_neg = torch.stack(text_ids_neg, dim=0)
        text_atts_neg = torch.stack(text_atts_neg, dim=0)

        # 合并正负样本
        text_ids_all = torch.cat([encoder_input_ids, text_ids_neg], dim=0)
        text_atts_all = torch.cat([text.attention_mask, text_atts_neg], dim=0)
        image_embeds_all = torch.cat([image_embeds_neg, image_embeds], dim=0)
        image_atts_all = torch.cat([image_atts, image_atts], dim=0)

        output_neg = self.text_encoder(text_ids_all,
attention_mask=text_atts_all,
                                       encoder_hidden_states=image_embeds_all,
encoder_attention_mask=image_atts_all,
                                       return_dict=True)

        # 匹配判断
        vl_embeddings = torch.cat([output_pos.last_hidden_state[:, 0, :],
output_neg.last_hidden_state[:, 0, :]], dim=0)
        vl_output = self.itm_head(vl_embeddings)
        itm_labels = torch.cat([torch.ones(bs, dtype=torch.long), torch.zeros(2
 * bs, dtype=torch.long)], dim=0).to(image.device)
        loss_itm = F.cross_entropy(vl_output, itm_labels)

        # -------- 图文生成任务（LM） --------
        decoder_input_ids = text.input_ids.clone()
        decoder_input_ids[:, 0] = self.tokenizer.bos_token_id
        decoder_targets = decoder_input_ids.masked_fill(decoder_input_ids ==
self.tokenizer.pad_token_id, -100)

```

```
95          decoder_output = self.text_decoder(decoder_input_ids,
96                                             attention_mask=text.attention_mask,
97                                             encoder_hidden_states=image_embeds,
98                                             encoder_attention_mask=image_atts,
99                                             labels=decoder_targets,
100                                            return_dict=True)
101
102         loss_lm = decoder_output.loss  # 语言建模损失（交叉熵）
103
104         return loss_ita, loss_itm, loss_lm
105
```

5. BLIP 的关键点与关键问题

😜 关键点总结

- **Bootstrapping**：通过 Captioner-Filter 机制生成高质量数据
- **MED 架构**：结合单模态和多模态编码器与解码器，实现视觉与语言的对齐与生成
- **预训练目标**：联合优化 ITC、ITM 和 LM 三个损失函数
- **CapFilt 机制**：通过字幕器和过滤器提升数据质量，扩展训练集

💞 关键问题：

1. **BLIP 如何解决现有 VLP 模型在任务适应性上的局限？**
   BLIP 提出多模态混合编码器-解码器（MED）架构，**支持三种功能：单模态编码（用于检索）、图像接地文本编码（用于匹配）、图像接地文本解码（用于生成）**，并通过三种目标联合预训练，实现理解与生成任务的灵活迁移
2. **CapFilt 方法如何提升噪声网页数据的质量？**
   CapFilt 包含两个模块：（1）生成器基于网页图像生成合成字幕，补充原始文本；（2）过滤器判断文本与图像的匹配度，移除原始网页文本和合成文本中的噪声，两者结合形成高质量训练集
3. **BLIP 在零样本迁移至视频-语言任务中表现优异的原因是什么？**
   BLIP 的图像-语言模型通过统一的视觉-语言表示学习，具备强泛化能力。处理视频时，通过均匀采样帧并拼接特征（忽略时序信息），直接迁移至文本-视频检索和视频 QA 任务，1k 测试集上文本-视频检索 R@1 达 43.3，超现有零样本方法 30% 以上