Day2

在day1调研中，Omni模型架构上面分为：Encoding，Alignment，Interaction，Generation，并且其训练和MLLM相似但是又有自己的特殊技巧和痛点。Omni模型发展路线有很多的经典模型或者是变体。接下来，我将通过文献或者是repo的形式，来进一步了解Omni的架构相关的**部分**具体操作和实践，而在明天会进行训练、后训练和Omni家族经典模型成员的调研。

Multi-modalities Alignment

MLLM和Omni综述中说到，Alignment，或者说Connector，可以是projector或者是Q-Former的操作模式。经过调查：BLIP-2中使用的是Q-Former，Qwen2.5-Omni中使用的是projector结合的方式。

Qwen2.5-Omni

> https://github.com/QwenLM/Qwen2.5-Omni/blob/main/

Qwen2.5-Omni中处理的主要是音频+图像（包含视频）+文字。

首先能够在modeling_qwen2_5_omni_low_VRAM_mode.py这个文件中，找到对应modality的encoder：

```
class Qwen2_5OmniAudioEncoder(Qwen2_5OmniPreTrainedModel):
    """
    Transformer encoder consisting of *config.encoder_layers* self
attention layers. Each layer is a
    [`Qwen2_5OmniAudioEncoderLayer`].

    Args:
        config: Qwen2_5OmniAudioEncoderConfig
    """
class Qwen2_5OmniVisionEncoder(Qwen2_5OmniPreTrainedModel):
    config_class = Qwen2_5OmniVisionEncoderConfig
    _no_split_modules = ["Qwen2_5OmniVisionBlock"]
```

对于音频处理，其中在init中可以看到的重要处理组件：

```
self.conv1 = nn.Conv1d(self.num_mel_bins, embed_dim, kernel_size=3,
padding=1)
self.conv2 = nn.Conv1d(embed_dim, embed_dim, kernel_size=3, stride=2,
padding=1)
self.positional_embedding =
SinusoidsPositionEmbedding(self.max_source_positions, embed_dim)
self.audio_bos_eos_token = nn.Embedding(2, config.output_dim)
self.layers = nn.ModuleList([Qwen2_5OmniAudioEncoderLayer(config) for _ in
range(config.encoder_layers)])
self.ln_post = nn.LayerNorm(config.d_model)
self.avg_pooler = nn.AvgPool1d(2, stride=2)
self.proj = nn.Linear(config.d_model, config.output_dim)
```

两个一维卷积估计使用用来提取音频特征的，然后并且有projection的线性层，投射到想要的维度上面，并且还有池化操作。

对于图像处理：

```
self.patch_embed = Qwen2_5_VisionPatchEmbed(
            patch_size=config.patch_size,
            temporal_patch_size=config.temporal_patch_size,
            in_channels=config.in_channels,
            embed_dim=config.hidden_size,
        )

head_dim = config.hidden_size // config.num_heads
self.rotary_pos_emb = Qwen2_5_VisionRotaryEmbedding(head_dim // 2)
self.blocks = nn.ModuleList([Qwen2_5OmniVisionBlock(config) for _ in
range(config.depth)])
self.merger = Qwen2_5OmniPatchMerger(
    dim=config.out_hidden_size,
    context_dim=config.hidden_size,
    spatial_merge_size=config.spatial_merge_size,
)
```

根据forward中的调用，图片先进行patching (An image is worth 16 words那篇文章的patching思想)然后分别编码，然后使用Conv3D投影维度之后使用merger进行合并。上面两个模态的encoding，分别使用的是Linear and Conv3D来进行的投影和维度转换。而在进行align之后，实际使用的时候，就是把它们当做正常的token放进文本token里面。可见在写的时候，projector就已经写在了encoder里面。

文本中有Image Audio and Video的占位符，然后embedding的时候先给正常的文本token进行embed，然后再个给对应的占位符换成encoder+projector处理好的tensor。

BLIP-2

> https://github.com/wooseungw/blip2/tree/main

在BLIP2模型中，Q-Former使用交叉注意力来对齐不同模态编码后的特征。

```

```

```
def init_Qformer(cls, num_query_token, vision_width,
cross_attention_freq=2):
    encoder_config = BertConfig.from_pretrained("bert-base-uncased")
    encoder_config.encoder_width = vision_width
    # insert cross-attention layer every other block
    encoder_config.add_cross_attention = True
    encoder_config.cross_attention_freq = cross_attention_freq
    encoder_config.query_length = num_query_token
    Qformer = BertLMHeadModel.from_pretrained(
        "bert-base-uncased", config=encoder_config
    )
    query_tokens = nn.Parameter(
        torch.zeros(1, num_query_token, encoder_config.hidden_size)
    )
    query_tokens.data.normal_(mean=0.0,
std=encoder_config.initializer_range)
    return Qformer, query_tokens
```

Q Former在初始化的时候，会初始化Query Token `query_tokens = nn.Parameter`，并且设置encoder的输入维度，以及cross attention的频率（这个组件中也有self attention）。BLIP2中只初始化一个Q Former，它自己对应一套唯一的可学习的query tokens，并根据调用方式在三条路径中被使用：图像路径（query 与 image encoder 交互）、文本路径（仅文本自注意力）、以及多模态路径（query + text + encoder 一起交互）。

```
if input_ids is not None:
    embeddings = self.word_embeddings(input_ids)
    ...
    if query_embeds is not None:
        embeddings = torch.cat((query_embeds, embeddings), dim=1)
else:
    embeddings = query_embeds
```

当同时提供 input_ids（文本）和 query_embeds（query tokens）时，query_embeds 会被拼到嵌入序列之前，形成一个统一序列，确保self attention能够在query and text token之间交互。其中，多出来的是cross attention layer，这些会对query部分额外做一次cross attention，从而接受Image的信息。

```
image_embeds = self.ln_vision(self.visual_encoder(image))
image_atts = torch.ones(image_embeds.size()[:-1],
dtype=torch.long).to(image.device)
query_tokens = self.query_tokens.expand(image_embeds.shape[0], -1, -1)
query_output = self.Qformer.bert(
    query_embeds=query_tokens,
    encoder_hidden_states=image_embeds,
    encoder_attention_mask=image_atts,
    use_cache=True,
    return_dict=True,
)
image_feats = F.normalize(self.vision_proj(query_output.last_hidden_state),
dim=-1)
```

传入 query_embeds和 encoder_hidden_states(也就是image的embedding)进行交叉注意力，然后query的last_hidden_state 并投影到 vision_proj 作为图像表示。

当只有text token的时候，Q Former只是文本编码器，走自注意力。所以总而言之：只有文本时：Q-Former 仅对文本做自注意力（text self-attention）；文本+图像时：把一组可学习的 query tokens 拼到文本前面，整个序列先做 self-attention，同时配置为 cross-attention 的层（由 cross_attention_freq 控制）对 query tokens 额外做 cross-attention，key/value 来自图像 encoder 的输出，从而让 query 从图像特征中抽取信息并实现对齐。

Multi-modalities Interaction

LLM 内部跨模态融合通过 input-level concatenation 或 layer-level cross-attention来进行结合推理。

MiniGPT-4

> https://github.com/Vision-CAIR/MiniGPT-4

MiniGPT-4使用Q-Former输出的对齐的视觉token，然后和文本token在embedding层拼接输入LLM，让LLM内部的attention take care of both tokens.

```
image_embeds = self.ln_vision(self.visual_encoder(image)).to(device)

image_atts = torch.ones(image_embeds.size()[:-1],
dtype=torch.long).to(device)
query_tokens = self.query_tokens.expand(image_embeds.shape[0], -1, -1)

query_output = self.Qformer.bert(
    query_embeds=query_tokens,
    encoder_hidden_states=image_embeds,
    encoder_attention_mask=image_atts,
    return_dict=True,
)

inputs_llama = self.llama_proj(query_output.last_hidden_state)
```

图像先经过 ViT，再经过 Q-Former，之后产生的`query_output.last_hidden_state`被输入对齐层，映射到LLaMA的维度，作为图像token。

之后文本的图像token在embedding里面进行拼接。MiniGPT-4 使用 `<ImageHere>` 作为图像占位符，然后对占位符之外的每一段文本做 tokenizer + embedding，然后文本embedding和图像embedding进行拼接：

```
p_tokens = self.llama_tokenizer(seg, return_tensors="pt",
add_special_tokens=False)
p_embed = self.embed_tokens(p_tokens.input_ids)
...
interleave_emb.append(
    torch.cat([p_embed, each_img_embed[None][:, idx * pn:(idx + 1) * pn]],
dim=1)
)
...
wrapped_emb = torch.cat([wrapped_emb, p_embed], dim=1)
```

形成的效果大致是：

```
[text_before_<ImageHere> embeddings]
[visual embeddings from Q-Former + llama_proj]
[text_after_<ImageHere> embeddings]
```

Flamingo

> https://github.com/mlfoundations/open_flamingo

Flamingo在 LLM decoder layer 中插入 gated cross-attention，让文本 hidden states 主动 cross-attend 到视觉 token。

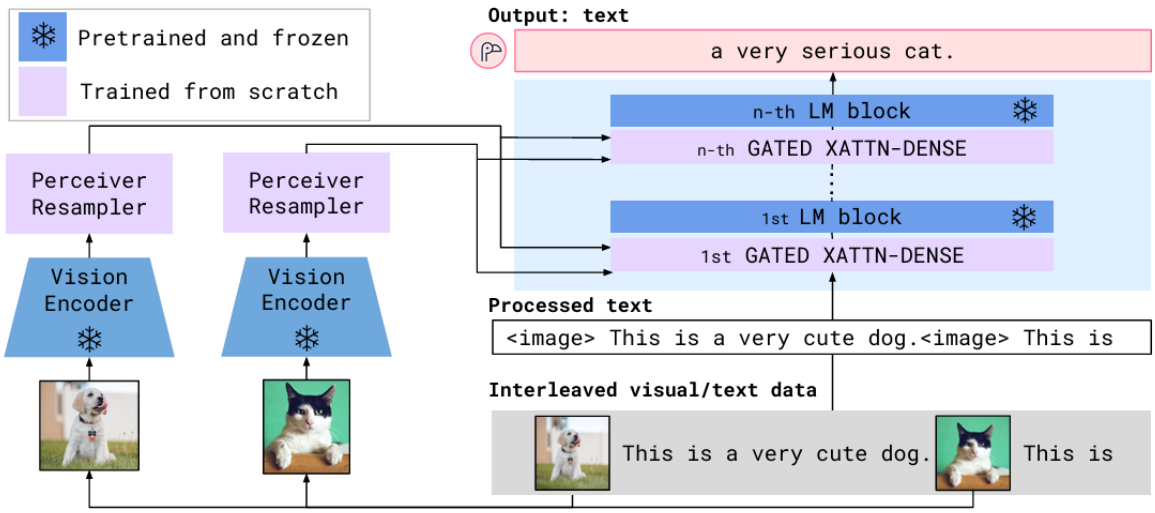


Figure 3: **Flamingo architecture overview.** Flamingo is a family of visual language models (VLMs) that take as input visual data interleaved with text and produce free-form text as output.

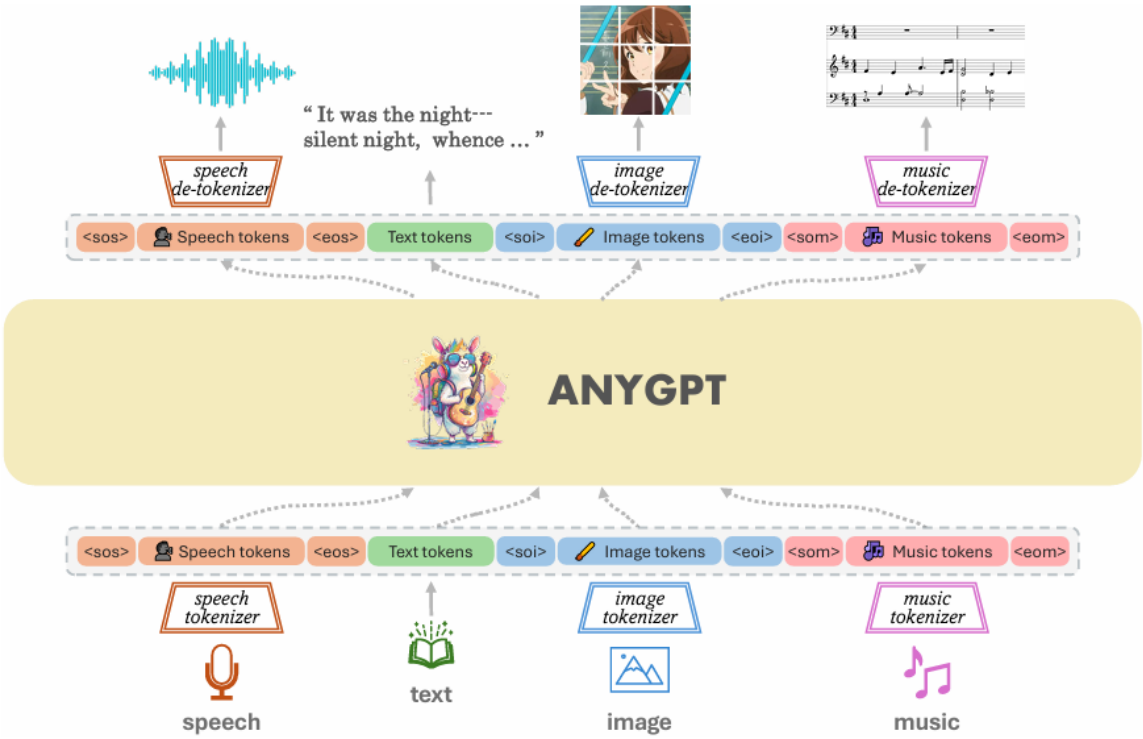
Flamingo使用的是CLIP encoder，并且输出的是patch level visual feature token, 而不是一个全局向量。tokenizer中加入了两个特殊token：`<|endofchunk|>` 和 `<image>`。其中 `<image>` 用来在文本序列中标记图像位置。文本中这些image占位符是为了告诉 cross-attention：当前文本 token 应该 attend 到哪一张图像的embedding。

`FlamingoLMMixin` 会把原来的 LLM decoder blocks 包装起来，在每隔若干层插入一个 `GatedCrossAttentionBlock`。参数 `cross_attn_every_n_layers` 控制插入频率，默认可以每层插一次。如果当前层有 `gated_cross_attn_layer`，就先执行跨模态 cross-attention；然后再执行原本的 decoder layer。cross attention中，文本对应的张量为query，视觉的embedding为key and value。

`GatedCrossAttentionBlock` 不是简单把 cross-attention 输出直接加到文本 hidden states 上，而是乘了一个可学习 gate，并且后面也有gated feed-forward，这意味着视觉信息注入强度是可学习的。

AnyGPT

> https://github.com/OpenMOSS/AnyGPT



AnyGPT将不同模态（图像、音频、视频、文本）token 化成统一序列，然后LLM进行 next-token prediction，支持任意模态组合输入和任意模态输出。模型结构和训练目标基本不改，主要靠数据预处理和后处理。

在anything2token的文件中，有：

```
def modality_tokens_to_string(tokens, modality="image"):
    ...
    tokens_str = [f"<{prefix}{token}>" for token in tokens]
    return start + "".join(tokens_str) + end
```

可见，对于模态输出的token来说，还会配上类似于`<soim>`和`<eoim>`这种模态起始和终止符，这里的notation是start / end of Image的意思。

图像使用 SEED tokenizer；语音使用 SpeechTokenizer；音乐使用 Encodec。README 也说明：SpeechTokenizer 用于 speech tokenization/reconstruction，SoundStorm 补全语音的副语言信息，SEED tokenizer 用于图像 tokenization；unCLIP SD-UNet 用于图像重建，Encodec-32k 用于音乐 tokenization/reconstruction。论文中也给出了 token 粒度：图像是每张图 32 个 token，image vocab size 是 8192；speech vocab size 是 1024；music 使用 Encodec RVQ，最终 flatten 成音乐 token 序列。

其他模态的内容被tokenize后，是直接变成token string，插入到prompt里面，来构建数据集的。实例如下：

```
[Human]: <sosp><1><1><1><eosp> Please acknowledge the user's vocal input...
[MMGPT]: <-Ins-> hello, how are you
         <-Res-> I am fine, thank you <sosp><2><2><2><eosp>
```

然后，使用LLaMA tokenizer 把整个 multimodal prompt 转成 `input_ids`，然后让标准 causal LLM 生成后续 token。当然，这意味着需要扩展tokenizer的vocab和对应的embedding matrix。当然，这些新增的embedding row是可学习参数，并且是用AnyGPT 的 multimodal pretraining / SFT 中，通过 next-token prediction 学出来的。

Multi-modalities Generation

如Omni综述所说，有text/token/representation based model产出的。text-based非常直观，就是输出的文字是caption，然后输入到对应模态的generator里面进行生成。

Text Based - GPT-4o and Gemini(1.5 / 2.5)

在 GPT-4o 和 Gemini 系列中，模型本身主要处理理解和推理任务，包括文本、图像或语音输入，但实际的生成输出多依赖外部模块:

- GPT-4o：用户输入可能包含图像和文本指令，LLM 对输入进行理解并生成自然语言描述或指令提示，然后调用 TTS(Text to Speach)、Stable Diffusion 或 Video Diffusion 等外部生成器完成最终输出。
- Gemini 1.5/2.5：类似方式，LLM 输出文本或控制信息，再由专门生成模型完成图像或音频生成。

Token Based - AnyGPT

AnyGPT生成了对应的离散token之后，会被送到对应的detokenizer进行还原。AnyGPT输入文本、图像、音频等模态均被 tokenizer 转化为离散 token 序列，然后LLM 直接预测下一模态 token，保持统一的 token 序列自回归机制。token 序列经过相应 detokenizer 解码为原始模态，例如图像、语音或视频信号。在工程上面，输出中的标识符依然是很重要的。

Representation Based - VITA系列

LLM 生成连续 latent representation，再由 decoder 或 diffusion 网络将其转换为最终模态输出。VITA系列使用的是这个方法。其中，VITA的多模态对齐、LLM内部interact的主要方式是：encoder 输出通过 projector 或connector变成LLM接受的token，然后interact方式是token concat。如：

```
[视觉 token1, 视觉 token2, …, 音频 token1, 音频 token2, …, 文本 token1, …]
```

而对于生成来说，LLM 生成 latent representation，生成高层语义向量表示，表示模态间的语义信息和上下文，然后使用decoder或者是diffusion来将latent representatoin转化为对应的模态。

总结

今天通过一些经典工作中的实现，来了解了Omni or MLLM中architecture的选择对应的做法和实践，让我对于这些design choice背后的动机有了更深刻的理解。比如说Q Former的设计初心，其实是为了能够让query token能和text token通过self attention进行信息交互，而cross attention缓解让query token感知到image embedding的信息。这样的设计其实是挺巧妙的，既可以防止过多的计算（如果是text和其他模态直接cross attention），又query token 可以被训练成同时适配不同模态，且Q-Former 可以作为统一对齐模块来处理图像、视频、音频输入。

我倒是认为：模型训练和微调，是非常关键的一部分。而同时，训练数据也是十分关键的。因此我打算明天调研经典Omni模型，并且着重探究其是如何进行训练的。在之后，再调研一下现有的benchmark，以及关于dataset构建相关技巧和结论相关的文献。