Day6

在Day6的调研中，我将补充泛读昨天发现的值得一看的文献，然后调研Any-to-any典范AnyGPT的代码实现。

补充文献泛读

Omni-DuplexEval

> Omni-DuplexEval: Evaluating Real-time Duplex Omni-modal Interaction

摘要中部分内容：Real-time duplex interaction is essential for multimodal AI systems operating in real-world scenarios, where models must continuously process streaming inputs and respond at appropriate moments. The benchmark consists of two complementary scenarios: (1) **Real-Time Description**, which evaluates the ability to generate continuous, time-aligned responses that track evolving multimodal inputs, and (2) **Proactive Reminder**, which evaluates the ability to identify salient events and respond at appropriate moments.

实验结论来说：当前Full Duplex模型离真实可用的交互助手还很远；它们能部分感知视频流，但很难同时做到“及时说、说得对、持续说”，尤其不擅长判断什么时候应该主动回应。

论文认为，传统视频理解 benchmark 多数是离线评测：模型先看完整视频，再给最终答案（如 Video-MME）；这与真实交互中边感知、边输出的要求不一致。

Real-Time Description(RTD) 要求模型持续输出与当前视频变化对齐的描述包含 6 个子任务：Counting、Interaction Relation、Omni、World Knowledge、OCR、Fine-grained Movement。另外一个Proactive Reminder(PR)，要求模型根据用户指令监控视频，在事件发生时主动提醒或纠正；它包含 3 个子任务：Event Reminder、Post-Event Reminder、Correction。论文中说所有视频都短于 1 分钟、平均约 34 秒，说明这对于长token处理或者是Memory/Attention的要求没有过于严苛，能够更好地纯粹检测模型的实时交互能力。

> 关于时间不长这一点，论文中claim是：Omni-DuplexEval 目前主要是短时流式交互，尚不能覆盖需要长期记忆、持续规划的复杂对话

文章有一些有意思的结论和依据：

- 很多已有视频/音视频 benchmark 要么不是 open-ended，要么不支持 streaming，要么不评估 proactive 行为，要么没有 temporal alignment。但是这个benchmark理论上如果想要有很好的表现需要四个都能支持

> Streaming 输入和普通 MLLM 输入的核心区别在于: 普通 MLLM 是一次性看完再回答，Streaming 是边接收边理解边回答

- 当前模型整体远低于人类实时水平（我记得Video-MME-v2那个benchmark的结论也是如此）
- 最大短板是主动响应时机
- 模型经常不回应，或回应错事件
- 感知类任务好于结构化推理类任务

论文不像(Video-)MME那样，使用acc作为指标，而是不同人物有不同的评判维度。RTD是两个维度：Content Consistency 和 Temporal Sensitivity。前者看整段回答是否与视频、音频和指令一致，惩罚事实错误、幻觉和遗漏；后者按带时间戳的句子评估回答是否对齐当前视频片段。这里评判标准特意把时间戳引进来，及其强调了Omni模型对于时间戳的感知能力。PR 的评测是时间出发测试：比如A说时间B发生的时候提醒他，然后随着视频的播放，发生了事件B，然后模型理想情况下应该输出提醒A关于B时间发生的内容，这个检测的窗口时期为10秒，并且让LLM-as-judge结合输出和我们设定好的gt来认为，是否成功有效提醒。

MLLMs are Deeply Affected by Modality Bias

> MLLMs are Deeply Affected by Modality Bias　2025年5月

摘要核心内容：This position paper argues that MLLMs are deeply affected by modality bias

文章定义Modality Bias为：Modality bias arises when certain modalities dominate the learning process, while others are under utilized or contribute less effectively。论文还将其后果总结为三类：对主导模态过度依赖、对弱势模态利用不足，以及 missing modality 场景下性能下降。这个现象为什么会发生？论文把原因归纳为三个主要因素，加上两个次要但重要的因素：

- 数据特性与数据集不平衡（dataset imbalance）：文本数据通常更紧凑、更抽象、更语义密集；图像、视频、音频则更冗余、更复杂、更难直接压缩成明确语义。因此在训练中，模型更容易从文本里找到 shortcut，而不是从视觉信号中学习细粒度语义。

> 论文明确指出，训练数据组成会显著影响模态利用；文本在很多任务中更容易用作推理依据，因此模型倾向于优先使用文本，把图像当成辅助。

- Backbone 能力不对称：LLM 太强，但是视觉 encoder、audio encoder 或 projector 通常只是接入模块。论文认为语言模型受益于成熟的 Transformer 架构、大规模语料和工业级训练；相比之下，视觉、音频等模态通常需要更复杂、更专门的 backbone，也未必有同等规模和质量的预训练资源。否则的话，很有可能MLLM推理核心仍然是LLM中继承的 language prior
- 很多既有训练目标都是本质上都容易把语言作为语义锚点。论文认为，这些目标虽然能学到图文对齐，但并不一定能促使模型真正做到 robust cross-modal fusion。反而可能这会让模型把文本当成主导语义空间，其他模态只是向文本对齐。这样就不是“模态和语言共同形成semantics”，而是“模态被训练为靠近text semantics”
- 不同模态的收敛速度不同，有些模态更容易拟合 label，有些模态更难训练，因此训练过程中容易形成不均衡贡献
- 如果模型没有显式学习模态之间的关系，就会倾向于使用更容易处理的模态，比如文本

实验使用了一个case study来验证一些结论：作者构造如下的输入条件：

```
1. Image-Text：正常图像 + 正常文本
2. Text-only with white image：文本 + 全白图
3. Text-only with black image：文本 + 全黑图
4. Text-only with noise image：文本 + 噪声图
5. Image-only：只给图像，用基础 prompt 要求根据图像回答
```

其中 white/black/noise 是为了模拟视觉模态缺失或不可用；image-only 则用于观察模型只依赖视觉时的能力。结果发现：只给文本 + 空白图，模型仍有相当表现；空白图、黑图、噪声图下结果相近；完整输入与 text-only 的预测一致性远高于与 image-only 的一致性；图像和文本单独输入时差异很大，说明融合机制不稳定。

文章提出了三种解决方向：

- 增强数据集中视觉模态的贡献。例如 MMStar 通过筛选视觉依赖样本来避免数据集中的语言捷径；MMMU-Pro 则通过过滤视觉无关样本、把问题嵌入图像等方式。
- 让模型在推理或训练中更关注视觉信息。一些方法会在 decoding、attention、prompt 或训练策略上强制模型更多利用图像，减少纯文本先验主导。
- preference optimization。可以构造modality bias 的正负偏好数据，让模型偏好那些真正依据多模态证据的回答，而不是依据单模态 shortcut 的回答。但文章也认为现有偏好数据构造方式仍有限，真实 modality bias 的来源更复杂。

A Survey of Multimodal Hallucination Evaluation and Detection

摘要关键内容：Specifically, we first propose a **taxonomy of hallucination** based on faithfulness and factuality, incorporating the common types of hallucinations observed in practice. Then we provide an **overview of existing hallucination evaluation benchmarks** for both T2I and I2T tasks, highlighting their construction process, evaluation objectives, and employed metrics. Furthermore, we **summarize recent advances in hallucination detection methods**, which aims to identify hallucinated content at the instance level and serve as a practical complement of benchmark-based evaluation. Finally, we **highlight key limitations in current benchmarks and detection methods**, and outline potential directions for future research.

文章关注两个主要方向：Image-to-Text, I2T，例如图像描述、视觉问答、图像理解；以及 Text-to-Image, T2I，例如根据文本生成图像。作者强调，MLLM 的幻觉不是单纯“回答错”，而是模型生成的内容看起来合理，但与视觉输入、文本提示或世界知识相矛盾。

Taxonomy: 文章最重要的分类是：Faithfulness and Factuality hallucination，前者指的是模型输出与输入不一致，后者指的是模型输出与世界知识、常识或领域知识不一致。这篇综述的一个核心观点是：I2T 和 T2I 虽然方向相反，但幻觉本质上都是跨模态对齐失败。

hallucination detection methods: 文章分成 black-box 和 white-box 两类。黑盒不需要访问模型内部，只看输入和最终的输出。典型方法包括 LLM-as-judge、VQA-based verification、caption-based verification等，判断这些输出事实是否和图像一致。白盒需要访问内部信号，如 attention、feature、logits等，优点是有可能定位幻觉从哪一层、哪个视觉区域产生，缺点是依赖模型结构。文章指出，I2T 中已有一些 white-box 方法，但 T2I 的检测目前主要还是依赖外部工具。

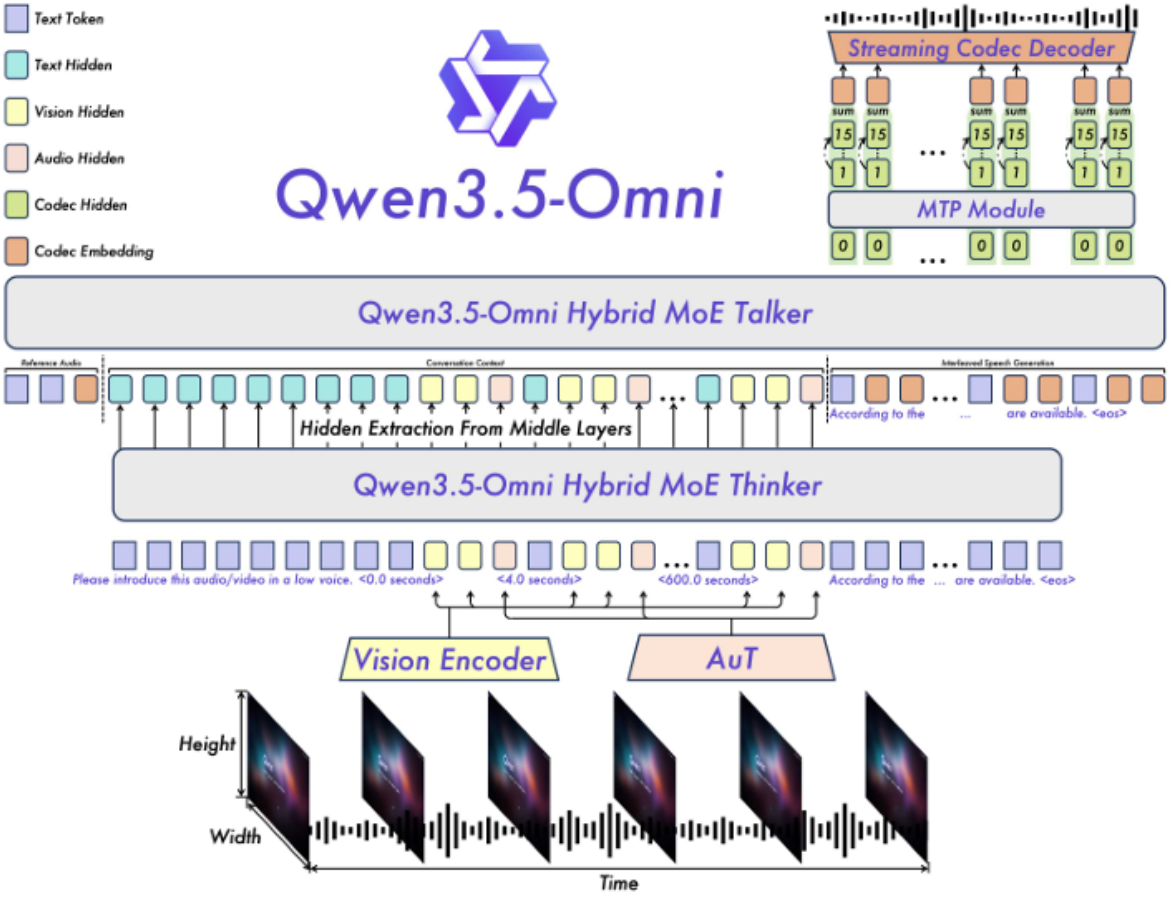
benchmark: 作者认为，很多 benchmark 只看最终答案是否正确，或者用粗粒度 accuracy 评估幻觉，但没有分析模型为什么幻觉、幻觉发生在推理链哪个阶段、训练数据和结构设计如何影响幻觉。文章明确提出，未来需要从 static outcome-based evaluation 转向 dynamic process-oriented evaluation。并且existing benchmark还有可解释性不足、真实场景评测不足等问题。

Qwen3.5-Omni

> Qwen3.5-Omni Technical Report

一个大规模端到端 Omni 模型系统：能够处理 text、image、audio、video/audio-visual 输入，并生成 text 或 streaming speech 输出。Qwen3.5-Omni 扩展到数百 billion 参数规模，支持 256k context，并基于异构图文数据和超过 1 亿小时音视频内容训练。

> 摘要中：Qwen3.5-Omni scales to hundreds of billions of parameters and supports a 256k context length. Qwen3.5-Omni-Plus achieves SOTA results across 215 audio and audio-visual understanding, reasoning, and interaction subtasks and benchmarks, surpassing Gemini 3.1 Pro in key audio tasks and matching it in comprehensive audio-visual understanding.



Qwen3.5-Omni 延续了 Qwen2.5-Omni / Qwen3-Omni 的 Thinker–Talker 架构。Thinker 负责多模态理解和文本生成。它接收 text、audio、image、video/audio-visual 信息，把它们转成统一序列表示，然后进行理解、推理和文本输出。audio 和 video 会被 interleave 到统一序列中，并插入显式 **timestamp**，以增强长视频和音视频上下文中的时间感知。Talker 负责语音生成。它接收 Thinker 的高层表示和当前文本输出，然后生成 streaming speech tokens。为了实现低延迟，Talker autoregressively 预测 multi-codebook sequence

Qwen3.5-Omni 的 Thinker 和 Talker 都采用 Hybrid Attention MoE。报告强调，这一设计用于扩展模型容量，同时兼顾推理效率和长序列建模。它还包含 Gated Delta Net，用于降低长音视频序列推理中的 KV-cache I/O overhead，提高吞吐和并发能力。

显式 timestamp 与长上下文音视频建模也很有意思。Qwen3.5-Omni 使用了更显式的时间编码策略：在 video 或 audio-video temporal patch 前插入 formatted timestamp，例如用秒数表示时间；对于 audio sequence，还会随机插入 timestamps，以提升跨模态时间对齐。作者认为，这比直接用绝对 temporal position IDs 更适合长视频/长音频，因为后者会产生稀疏且很大的 position IDs，削弱长程时间建模。

Qwen3.5-Omni 的 pretraining 分成三阶段。第一阶段是 Encoder Alignment Stage：LLM 初始化自 Qwen3.5，vision encoder 也来自 Qwen3.5，audio encoder 初始化为 AuT；训练时固定 LLM，分别训练 vision/audio encoder 及其 adapters，使不同模态能接入 LLM 语义空间。第二阶段是 General Stage：使用约 4T tokens 的多模态数据， unfreeze all parameters，用更广泛的多模态数据训练模型的通用理解与交互能力。第三阶段是 Long Context Stage：最大长度从 32,768 增加到 262,144，相比于Qwen2.5-Omni是巨大提升，并提高 long audio / long video 数据比例，以增强长序列理解能力。

Post-training 方面，Thinker 使用三阶段策略。第一阶段通过 specialist distillation，把 text、vision、audio 等专门 teacher models 的能力蒸馏进统一模型；第二阶段做 on-policy distillation，把文本输入下更强的 response capability 迁移到 audio-input setting，缩小音频查询和文本查询之间的质量差距。

> 本质上：前者是专家模型生成高质量答案，后者是把文字输入下生成的高质量答案拿来监督“语音输入”下的回答

FutureOmni

> FutureOmni: Evaluating Future Forecasting from Omni-Modal Context for Multimodal LLMs

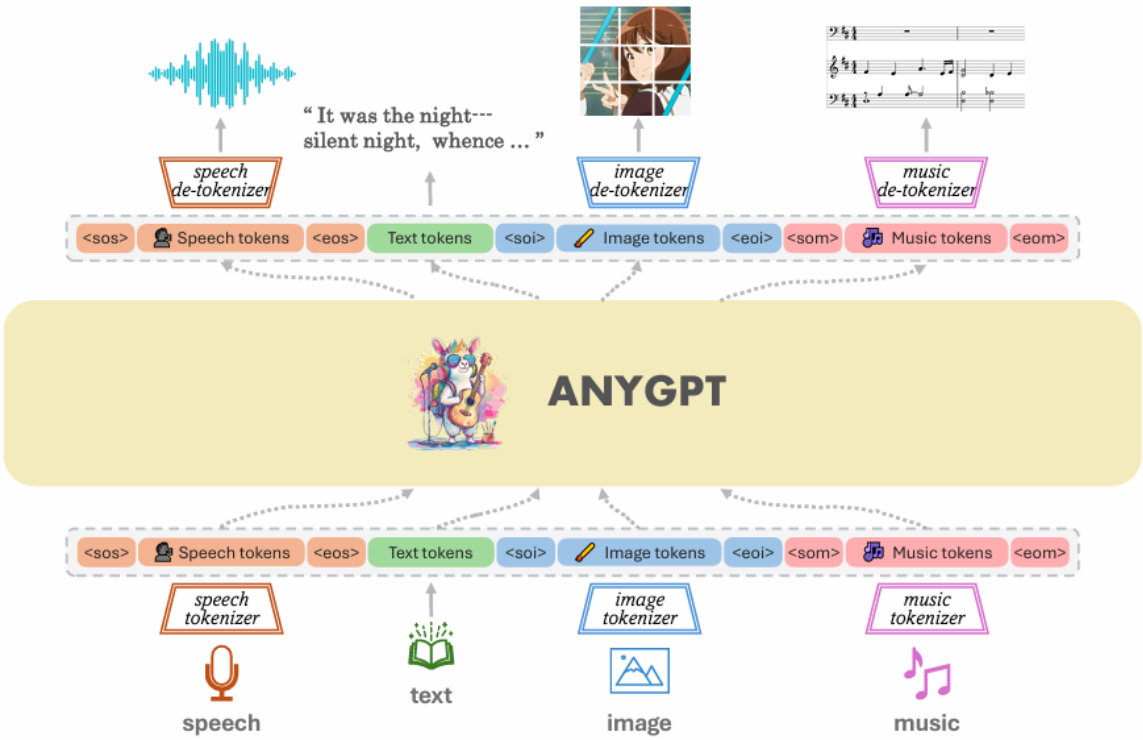
摘要重要内容：Although Multimodal Large Language Models (MLLMs) demonstrate strong omni-modal perception, **their ability to forecast future events from audio–visual cues remains largely unexplored**, as existing benchmarks focus mainly on retrospective understanding. To bridge this gap, we introduce Future Omni,the first benchmark designed to evaluate **omni-modal future forecasting from audio–visual environments.**

真实智能体需要的是 prospective forecasting，也就是基于当前视觉、声音、语音、常识和因果线索预测未来。文章因此提出 FutureOmni，称其为第一个专门评估 audio-visual future forecasting 的 Omni-modal benchmark。样本是 multiple-choice QA。每个样本给出一个 premise event，然后问which event is its most direct conclusion?模型需要从多个候选未来事件中选择最合理的一个。

FutureOmni 包含 919 个视频 和 1,034 个 multiple-choice QA pairs，覆盖 8 个主要领域。GitHub README 还强调所有视频都经过收集以保证 zero contamination，即 100% Original Video Rate。

文章还提出 Omni-Modal Future Forecasting （OFF） training strategy，并构造了 7K-sample instruction-tuning dataset 来缓解模型的未来预测能力不足。OFF 用 instruction tuning 的方式，来促使模型能够学会使用当前多模态环境线索推理出未来相关内容候选选项中最有可能选项的能力。

AnyGPT 代码实现



首先是词表相关的设计。通过上面这张图，可以发现有Speech, Image and Music tokens以及特殊的占位符start/end of X. 在`anything2token.py`中，先定义了token相关的内容：

```
image_prefix = "👀"
speech_prefix = "🗣️"
music_prefix = "🎶"
audio_prefix = "👂"
start_of_image, end_of_image = '<soim>', '<eoim>'
start_of_speech, end_of_speech = '<sosp>', '<eosp>'
start_of_music, end_of_music = '<somu>', '<eomu>'
start_of_audio, end_of_audio = '<soau>', '<eoau>'
image_vocab_size=8192
speech_vocab_size=1024
music_codebook_size=2048
music_codebook_num=4
music_vocab_size=music_codebook_size * music_codebook_num
audio_codebook_size=1024
audio_codebook_num=4
audio_vocab_size=audio_codebook_size * audio_codebook_num

modal_special_str = {
    "image":{
        "prefix": image_prefix,
        "sos": start_of_image,
        "eos": end_of_image,
        "vocab_size": image_vocab_size
    },
......
```

可以看到modal_special_str 是一个字典，记录每个模态的 (prefix, sos, eos, vocab_size)，供训练脚本统一注册新
token。

之后是把离散索引拼成字符串`modality_tokens_to_string`：image / speech（1-D 序列）：直接 <👀{idx}> / <🗣️{idx}> 拼接，再外包 … 或 ….。对于music / audio，通过`tokens[layer_idx][idx] + codebook_size * layer_idx`把不同层的 codebook 拼成一个大词典，然后按帧交错铺平（第 1 帧的 4 层 → 第 2 帧的 4 层 → …）。这样 4 个 2048 维 codebook 就变成 1 个 8192 维大词典，每个 LLaMA token 都对应明确的「层 + 索引」。这个是和soundstorm speechtokenizer相关联的。下面是处理之后的数据例子：


```
{"text": "A flat design background featuring an artistic design.", "image": "<soim><👀7357><👀680><👀5037><👀7841><👀680><👀430><👀5037><👀2118><👀5649><👀2157><👀2157><👀5037><👀5037><👀5963><👀3707><👀3707><👀2157><👀430><👀2157><👀2118><👀5037><👀5873><👀5037><👀2157><👀2118><👀2275><👀2118><👀430><👀5649><👀5963><👀7319><👀4519><eoim>"}
```

```
{"text": "\"Kitty History\" is a comedic song by Trevor Moore, from his album \"High In Church\". The song, originally performed on Comedy Central, is a satirical exploration of a parallel universe where cunning felines commit adorable war crimes. The piece combines elements of stand-up comedy, music videos, and history in a hilarious and unique way.", "music": "<somu><🎶166><🎶3979><🎶6115><🎶7914><🎶166><🎶3979><🎶6115><🎶7914><🎶166><🎶3979><🎶6002>.....<eomu>"}
```


在推理的时候，即`cli_infer_base_model.py`中，对于一个case，会调用相关的encoder，然后调用`modality_tokens_to_string`函数，这个会进一步和prompt结合构成instruction来用于推理。

token的定义和转化为string处理完了，那么论文中提到的一个重要操作是扩展LLaMA的词表。对于stage1来说，在`prompter.py`中：

```
tokenizer = LlamaTokenizer.from_pretrained(
        model_args.model_name_or_path,
        model_max_length=training_args.model_max_length,
        padding_side="right",
        use_fast=False,
    )
tokenizer.pad_token_id = (
    0  # unk. we want this to be different from the eos token
)
tokenizer.padding_side = "left"  # Allow batched inference
for token in [user_name, chatbot_name, user_end, chatbot_end]:
    if token not in tokenizer.get_vocab():
        logger.info(f"Add special unit tokens {token} to tokenizer.vocab")
        tokenizer.add_tokens([token])

# 开始给tokenizer加入其他模态的字表
for modality in modal_special_str.keys():
    prefix=modal_special_str[modality]["prefix"]
```

```
    start=modal_special_str[modality]["sos"]
    end=modal_special_str[modality]["eos"]
    modality_vocab_size = modal_special_str[modality]["vocab_size"]
    if start not in tokenizer.get_vocab():
        logger.info(f"Add {modality} tokens <{prefix}0>-<{prefix}
{modality_vocab_size-1}> to tokenizer.vocab")
        tokens = [f"<{prefix}{x}>" for x in range(modality_vocab_size)] +
[start, end]
        tokenizer.add_tokens(tokens)

model = LlamaForCausalLM.from_pretrained(
    model_args.model_name_or_path
)
# resize embedding
embedding_size = model.get_input_embeddings().weight.shape[0]
if len(tokenizer) > embedding_size:
    model.resize_token_embeddings(len(tokenizer))
```

加完之后词表扩展了：image: 8192 + 2 (sos/eos); speech: 1024 + 2; music: 8192 + 2。根据huggingface的规则，这些token对应的embedding是平均初始化的，之后会进行学习。

token处理完和加入词表之后，要构造用于训练的数据。这里首先介绍模板的构造，之后介绍数据是如何获得和处理的。对于stage1的pretrain来说：

```
def generate_x2t_template(
    self,
    modality_str: str,
    text: Union[None, str],
    modality: str
) -> str:
    meta_template = user_name+": {instruction} {input}"+f"{user_end}
{chatbot_name}: "+"{output}"+f"{chatbot_end}"
    instructions = other2text_instructions[modality]
    res = meta_template.format(
        instruction=random.choice(instructions),
        input=modality_str,
        output=text
    )
    return res

def generate_t2x_template(
    self,
    modality_str: str,
    text: Union[None, str],
    modality: str
) -> str:
    meta_template = user_name+": {instruction} This is input: {input}"+f"
{user_end} {chatbot_name}: "+"{output}"+f"{chatbot_end}"
    instructions = text2other_instructions[modality]
    res = meta_template.format(
        instruction=random.choice(instructions),
        input=text,
```

```
        output=modality_str
    )
    return res

def generate_template(
    self,
    modality_str: str,
    text: Union[None, str],
    modality: str,
    x2text_prob: float = 0.5
) -> str:
    # options = ["other2text", "text2other"]
    # 按照概率随机选择
    if random.random() < x2text_prob:
        res = self.generate_x2t_template(modality_str, text, modality)
    else:
        res = self.generate_t2x_template(modality_str, text, modality)
    if self._verbose:
        print(res)
    return res
```

前两个函数是定义X->Text and Text->X的模板，而后使用.format进行替换；
`generate_template`里面会随机选一个模板，然后抽一模态对应的指令。这样就有了两个方向的用于训练的数据。

而在stage2中，AnyGPT使用多轮对话进行SFT，在`conversation.py`中：

```
def get_prompt(self, force_image_generation=False,
force_speech_generation=False, force_music_generation=False,
force_res_prefix=None) -> str:
    """Get the prompt for generation."""
    system_prompt =
self.system_template.format(system_message=self.system_message)
    ret = system_prompt + "\n"
    for i in range(len(self.messages)):
        role, message = self.messages[i]
        if message and i % 2 == 0:
            ret += role + ": " + message + self.sep + "\n"
        elif message and i % 2 == 1:
            ret += role + ": " + message + self.sep2
            if i != len(self.messages) - 1:
                ret += "\n"
        else:
            ret += role + ":"
    if self.messages[-1][0] == self.roles[0]:
        ret += self.roles[1] + ":"
        if force_res_prefix != None:
            ret += " "+force_res_prefix
        if force_image_generation:
            ret+=" "+start_of_image
        elif force_speech_generation:
```

```
            ret+=" "+start_of_speech
        elif force_music_generation:
            ret+=" "+start_of_music
    return ret
```

最后得到的是这样的模板：

```
{"id": 1, "conversation": [[{"role": "user", "message": "<sosp><🗣1><🗣1><
🗣1><eosp> Please acknowledge the user's vocal input, create a textual
response"}, {"role": "assistant", "message": "<-Ins-> hello, how are you\n
<-Res-> I am fine, thank you <sosp><🗣2><🗣2><🗣2><eosp> "}]]}
```

上述是指令的改造。那么对于数据集原料的构造，repo中有大量的相关代码。对于stage1的raw data，首先是`load_and_process`：

```
def load_and_preprocess(data_path, modality, x2text_prob=0.5):
    raw_datasets = load_dataset("json", data_files=data_path)
    # 从后缀名前面抽取出数据集名称
    dataset_name = data_path.split("/")[-1].split(".")[0]
    print(dataset_name)
    tokenized_cache_file_names = {
        "train":os.path.join(data_args.cache_dir, modality, dataset_name,
'tokenized', 'train', 'processed_train.arrow'),
        "test":os.path.join(data_args.cache_dir, modality, dataset_name,
'tokenized', 'test', 'processed_valid.arrow'),
    }
    logger.info(f"loading {dataset_name} tokenized data")

    with training_args.main_process_first(desc="dataset map tokenization"):
        tokenized_datasets = raw_datasets.map(
            lambda x: template_and_tokenize(x, x2text_prob),
            batched=False,
            remove_columns=["text", modality],
            num_proc=data_args.preprocessing_num_workers,
            load_from_cache_file=True,
            desc="Running tokenizer on dataset",
            cache_file_names=tokenized_cache_file_names
    )
```

datasets 直接读 jsonl，这里的样本只有text and modality两种字段。然后是模板填空和文本tokenize(`template_and_tokenize`):

```
def template_and_tokenize(example, add_eos_token=True, x2text_prob=0.5):
    prompt = generate_template(example, x2text_prob)
    return tokenize_func(prompt, add_eos_token)

def tokenize_func(sentence, add_eos_token=True):
    result = tokenizer(
        sentence,
        truncation=True,
```

```
            max_length=tokenizer.model_max_length,
            padding=False,
            return_tensors=None,
        )
        if (
            result["input_ids"][-1] != tokenizer.eos_token_id
            and len(result["input_ids"]) < tokenizer.model_max_length
            and add_eos_token
        ):
            result["input_ids"].append(tokenizer.eos_token_id)
            result["attention_mask"].append(1)
        result["labels"] = result["input_ids"].copy()
        return result
```

这里面对任何位置进行mask，这说明instruction等所有内容都在loss里面。之后进行train and val切分之后，使用`group_texts`来packing batch到block_size以充分利用显存：

```
# Main data processing function that will concatenate all texts from our
dataset and generate chunks of block_size.
def group_texts(examples):
    result = {}
    for k in examples.keys():
        v = examples[k]
        if k == "input_ids":
            grouped_input_ids = []
            current_chunk = []
            current_length = 0
            for ids in v:
                if current_length + len(ids) > block_size:
                    grouped_input_ids.append(current_chunk)
                    current_chunk = []
                    current_length = 0
                current_chunk.extend(ids)
                current_length += len(ids)
            # Make sure the last chunk is added
            if current_length > 0:
                grouped_input_ids.append(current_chunk)
            result[k] = grouped_input_ids
    result["attention_mask"] = [[1 for _ in row] for row in
result["input_ids"]]
    result["labels"] = result["input_ids"].copy()
    return result
```

依次把短样本拼接到一起，凑到 block_size=4500就切一刀。最后也是最重要的是：多模态数据的interleave。`stage1_pretrain.py`是一堆 elif，根据传入了哪些数据路径选择 probabilities 比例。`interleave_datasets` 按概率从各数据集里抽样，生成一个流式混合数据集，并且也最终控制着每种模态在最终训练batch中出现的频率。

对于stage2的数据集构造，主要的preprocess就是套用stage2对话模板，但是重要的是这里有label masking了：只在AnyGPT输出部分的算loss。

```
targets = copy.deepcopy(input_ids)
sep = conv.sep + '\n' + conv.roles[1] + ": "   # "<eoh>\n[AnyGPT]: "
for conversation, target in zip(conversations, targets):
  total_len = len(target)
  turns = conversation.split(conv.sep2)       # 用 <eom> 切多轮
  cur_len = 1
  target[0] = IGNORE_TOKEN_ID                 # 跳过 BOS
  for i, turn in enumerate(turns):
      if turn == "": break
      turn_len = len(tokenizer(turn).input_ids)
      parts = turn.split(sep)                 # parts[0]=用户提问段，
parts[1]=助手回答
      if len(parts) != 2: break
      parts[0] += sep
      instruction_len = len(tokenizer(parts[0]).input_ids) - 2
      if i != 0 and not tokenizer.legacy:
          instruction_len -= 1
      for k in range(cur_len, cur_len + instruction_len):
          target[k] = IGNORE_TOKEN_ID         # 把 user prompt 全部 mask 为
-100
      cur_len += turn_len
      if i != 0 and not tokenizer.legacy:
          cur_len -= 1
  excepted_len = total_len - 1 if add_eos_token else total_len
  for k in range(cur_len, excepted_len):
      target[k] = IGNORE_TOKEN_ID             # 末尾不完整内容也 mask
```

那么数据整理好了之后，就是进行分布式训练的常规深度学习训练代码，并且使用的是Trainer这个高度封装好的内容来托管训练内部发生的事情。

总结来说，训练相关的代码并不是最核心的，而如何构造数据集并构造指令instruction是最重要的。