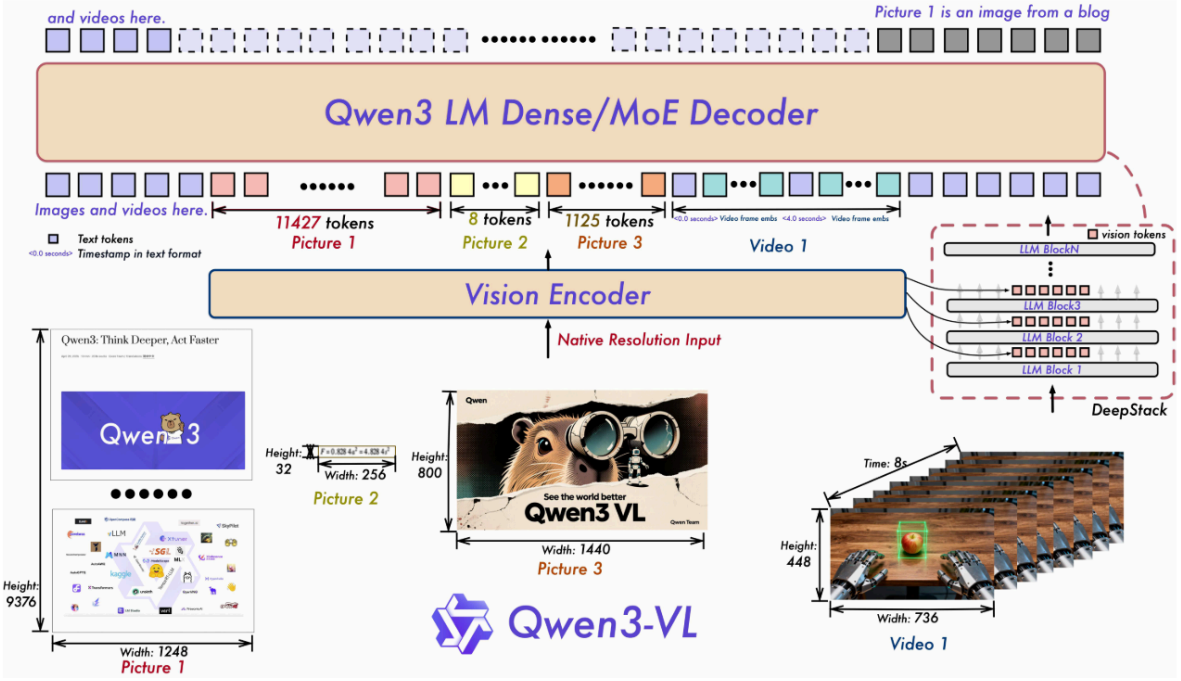
Qwen3VL

1. 前置知识准备

- ROPE系列：https://zhuanlan.zhihu.com/p/1948048954689295110
- RoPE以及MRoPE（Qwen2VL和Qwen2.5VL）视频讲解：旋转位置编码-详细解析RoPE和MRoPE*哔哩哔哩*bilibili
- LVLM架构图（QwenVL系列遵循该架构，Qwen2.5VL的connector属于mlp merger）：https://arxiv.org/pdf/2306.13549

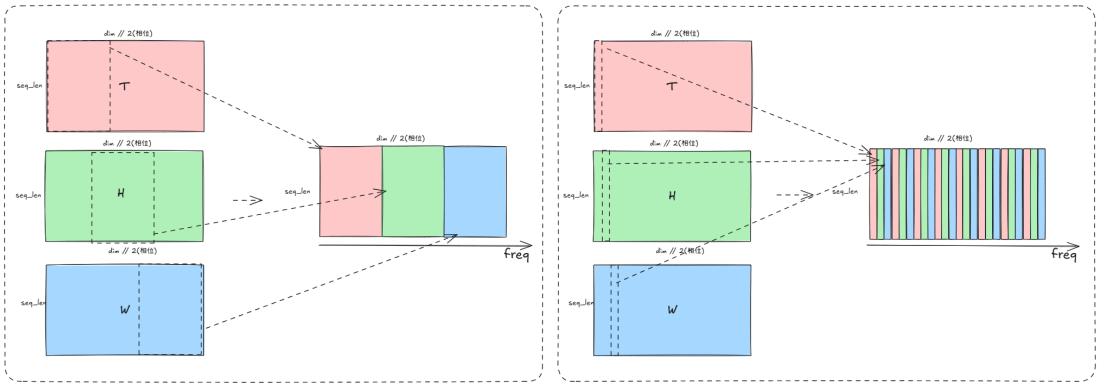
2. 架构创新点(总览)



2.1 paper主要创新点

1. Interleaved MRoPE

   Qwen2.5VL采用的MRoPE中，t, h, w是按照freq的dim顺序分配，如下图左侧部分：



   这导致高频和低频范围只有某一种模态，造成不平衡的问题。改进是交错排布，如上图右侧部分。

2. DeepStack

   收集特定层输出的ViT编码器特征，经过merger将维度和LLM对齐，随后直接添加到LLm decoder的前三个block中，与LLM的hidden_states 相加。

3. TimeStamps

   在preprocessor的过程中，对video的时间直接使用timestamp去tokenize，添加到input_ids中，不再对多个fps进行sample。

2.2 其余架构上的不同点（相较于Qwen2.5VL）

ViT的改动（paper中说采用了SigLIP2-arch）：

1. 取消了window attention，改回sdpa
2. 默认hidden_act的激活函数从silu改成了gelu
3. PatchEmbed启动bias
4. patch_size从`14 * 14`变成了`16 * 16`
5. ViT的位置编码同时使用了绝对位置编码和相对位置编码
6. RMSNorm -> LayerNorm （`self.norm = nn.LayerNorm`）（包括PatchMerger）

支持256k token上下文。

参考资料：Qwen3-VL源码解读-创新点速览*哔哩哔哩*bilibili

3. 环境配置

👂 environment

搜索文档，找别人配置好的包，然后写到requirements.txt，uv add -r。

```
# Core dependencies
gradio==5.46.1
gradio_client==1.13.1
```

```
transformers-stream-generator==0.0.5
torch==2.8.0
torchvision==0.23.0
accelerate
transformers==4.57.0
vllm==0.11.0
qwen-vl-utils
openai
decord[gpu]
# Optional dependency
# Uncomment the following line if you need flash-attn
# flash-attn
```

🍸测试demo.py **(from Huggingface)**

```
# Debugging setup
import debugpy
try:
    # 5678 is the default attach port in the VS Code debug configurations. Unless
a host and port are specified, host defaults to 127.0.0.1
    debugpy.listen(("localhost", 9501))
    print("Waiting for debugger attach")
    debugpy.wait_for_client()
except Exception as e:
    pass

import torch

from transformers import Qwen3VLForConditionalGeneration, AutoProcessor

# default: Load the model on the available device(s)
model = Qwen3VLForConditionalGeneration.from_pretrained(
    "model_dir", dtype="auto", device_map="cpu"
)
device = "cuda" if torch.cuda.is_available() else "cpu"
# We recommend enabling flash_attention_2 for better acceleration and memory
saving, especially in multi-image and video scenarios.
# model = Qwen3VLForConditionalGeneration.from_pretrained(
#     "Qwen/Qwen3-VL-8B-Thinking",
#     dtype=torch.bfloat16,
#     attn_implementation="flash_attention_2",
#     device_map="auto",
# )s
processor = AutoProcessor.from_pretrained("model_dir")

# messages = [
#     {
#         "role": "user",
#         "content": [
#             {
#                 "type": "image",
#                 "image": "https://qianwen-res.oss-cn-beijing.aliyuncs.com/Qwen-
VL/assets/demo.jpeg",
```

```
#             },
#             {"type": "text", "text": "Describe this image."},
#         ],
#     }
# ]
messages = [
    {
        "role": "user",
        "content": [
            {
                "type": "video",
                "video": "https://qianwen-res.oss-cn-beijing.aliyuncs.com/Qwen2-
VL/space_woaudio.mp4"
            },
            {
                "type": "text",
                "text": "How long is this video?"
            }
        ]
    }
]
# Preparation for inference
inputs = processor.apply_chat_template(
    messages,
    tokenize=True,
    add_generation_prompt=True,
    return_dict=True,
    return_tensors="pt"
)
inputs = inputs.to(device)

# Inference: Generation of the output
generated_ids = model.generate(**inputs, max_new_tokens=128)
generated_ids_trimmed = [
    out_ids[len(in_ids) :] for in_ids, out_ids in zip(inputs.input_ids,
generated_ids)
]
output_text = processor.batch_decode(
    generated_ids_trimmed, skip_special_tokens=True,
clean_up_tokenization_spaces=False
)
print(output_text)
```

（可选）vscode debug 方法推荐

参考github: https://github.com/yuanzhoulvpi2017/vscode_debug_transformers

步骤：

1. wsl remote打开vscode
2. install debugpy package （uv，如果是conda就用conda的方式）
3. uv run 入口文件(.py)（uv，如果是conda就用conda的方式）
4. 点击debug

如果是在一台服务器看代码，在另一台服务器运行，代码里要做对应更改。 代码：import debugpy try: # 监听所有网络接口，允许远程调试连接 debugpy.listen(("0.0.0.0", 9501)) print("Waiting for debugger attach on 0.0.0.0:9501") debugpy.wait_for_client() except Exception as e: pass launch.json 中要把localhost改为对应服务器的网址

4. 模型调试details

4.1 Image模态

> **输入**形式类似于Agent的输入形式，传入一个messages，包含user content:

```
messages = [
    {
        "role": "user",
        "content": [
            {
                "type": "image",
                "image": "https://qianwen-res.oss-cn-beijing.aliyuncs.com/Qwen-VL/assets/demo.jpeg",
            },
            {"type": "text", "text": "Describe this image."},
        ],
    }
]
```

4.1.1 预处理阶段

> 总体目标：将输入的 messages 全部 tokenize，得到LLM形式的输入序列id： **input ids**。

第一步将用户的输入插入到模板中。例如，对于**image + text输入**，会得到以下promt template：

`'<|im_start|>user\n<|vision_start|><|image_pad|><|vision_end|>Describe this image.<|im_end|>\n<|im_start|>assistant\n<think>\n'`

主要关注prompt的视觉部分：

```
<|vision_start|><|image_pad|><|vision_end|>
```

image模态的token暂时先使用`<|image_pad|>`占用，但是整个图像不会只有一个token，所以目前只用一个pad是不对的，因此要弄清楚image token的个数是多少，然后填充对应个数的pad。所以，需要**计算出图像所占用的token的具体长度。**

> 思路：加载图像->图像预处理->计算出所需要的token数->替换<|image_pad|>

下面的函数包含了对图像数据的预处理过程。

Path : .venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/processing_qwen3_vl.py (line 163)

```
image_inputs = self.image_processor(images=images,
**output_kwargs["images_kwargs"])
```

最终跳转到下面的代码块，

Path: .venv/lib/python3.13/site-packages/transformers/models/qwen2_vl/image_processing_qwen2_vl_fast.py的 **_preprocess_image_like_inputs** 方法 (line 143)

```
if images is not None:
    images = self._prepare_image_like_inputs(
        images=images, do_convert_rgb=do_convert_rgb,
input_data_format=input_data_format, device=device
    )# 读取并转成tensor
    batch_feature = self._preprocess(images, **kwargs)# 预处理，kwargs包含了所有定义的预处理参数，包括patch size。
```

接下来解释上面两行函数具体做了什么。

1. 第一行**读取：**PIL 对image读取，传入进来的size是（2048，1365），第一步转成tensor：torch.Size([3, 1365, 2048])。
2. 第二行预处理之**resize：**目的是将原始图像resize变成能打patch的size，输出的resized分辨率大小为：**（1376， 2048）**，resize的代码如下。

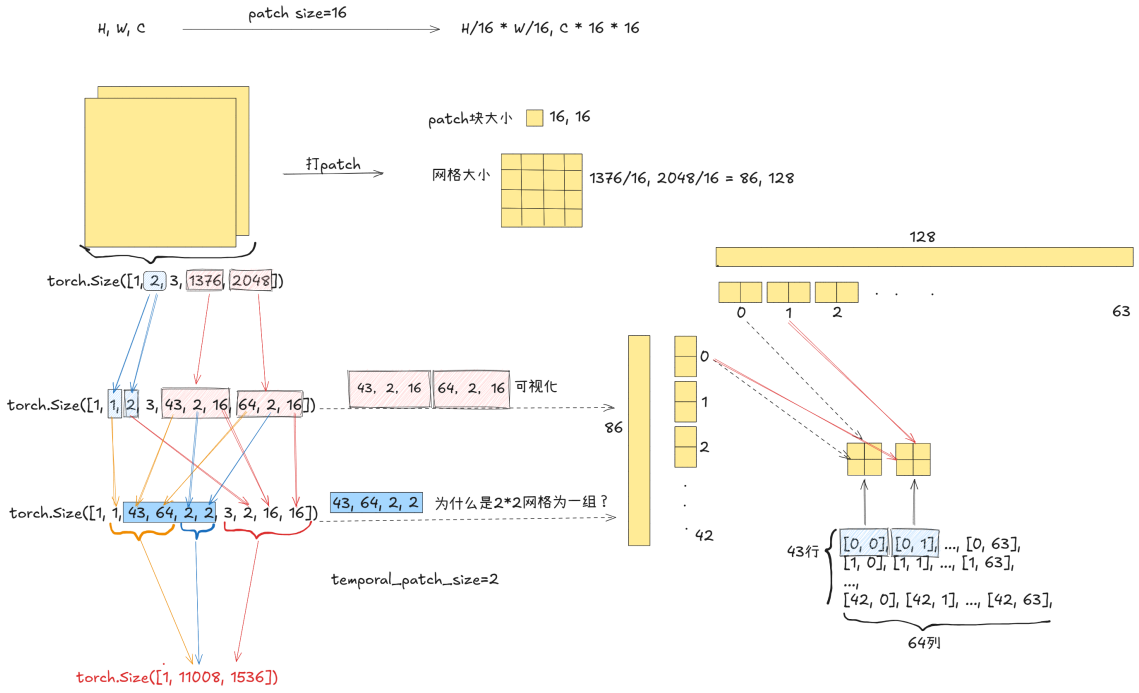
   ```
   resized_height, resized_width = smart_resize(
       height,
       width,
       factor=patch_size * merge_size,
       min_pixels=size["shortest_edge"],
       max_pixels=size["longest_edge"],
   )

   # smart_resize函数关键处理如下：
   h_bar = round(height / factor) * factor
   w_bar = round(width / factor) * factor
   ```

3. 第二行预处理之**重组**：

   这一步涉及以下几个关键点：

   - 图像怎么打patch，打完patch一共有多少个网格？
   - 怎么排布成四个网格为一组？
   - 为了和视频兼容处理（2帧为一组，详细可以查看4.2.1节）复制完全一样的图像模拟出2帧的概念。

   > 特别注意：处理image 模态的时候，需要对原图完全复制一份，模拟出时间的维度，能和视频兼容处理。但是video模态不需要这样的复制。



现在可以计算出最终 **visual modality** 有多少个token了，从而生成对应数量的pad token填充到下面的prompt中（把<|vision_start|><|image_pad|><|vision_end|>替换掉），作为LLM的输入。

`'<|im_start|>user\n<|vision_start|><|image_pad|><|vision_end|>Describe this image.<|im_end|>\n<|im_start|>assistant\n<think>\n'`

注意，对齐到LLM的时候，visual embedding会经过Merger层，该层会将上图中2*2的block压缩成一个block，因此，

$$\text{最后填充到}prompt\text{中的视觉}token\text{数量是：}\frac{43 \times 64 \times 2 \times 2}{2 \times 2} = 43 \times 64 = 2752$$

将promt中出现的单个 `'<|image_pad|>'` 变成2752个 `'<|image_pad|>'` 。

Path: .venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/processing_qwen3_vl.py（line 192）

```
 num_image_tokens = image_grid_thw[index].prod() // merge_length
 text[i] = text[i].replace(self.image_token, "<|placeholder|>" *
num_image_tokens, 1)
```

最后，将text tokenize，得到id。

id以及visual data会以prefill的方式输入。（**Path:** .venv/lib/python3.13/site-packages/transformers/generation/utils.py（_sample方法））

4.1.2 了解Qwen3VL模型结构

> 总体目标：快速了解Qwen3VL class的组成。

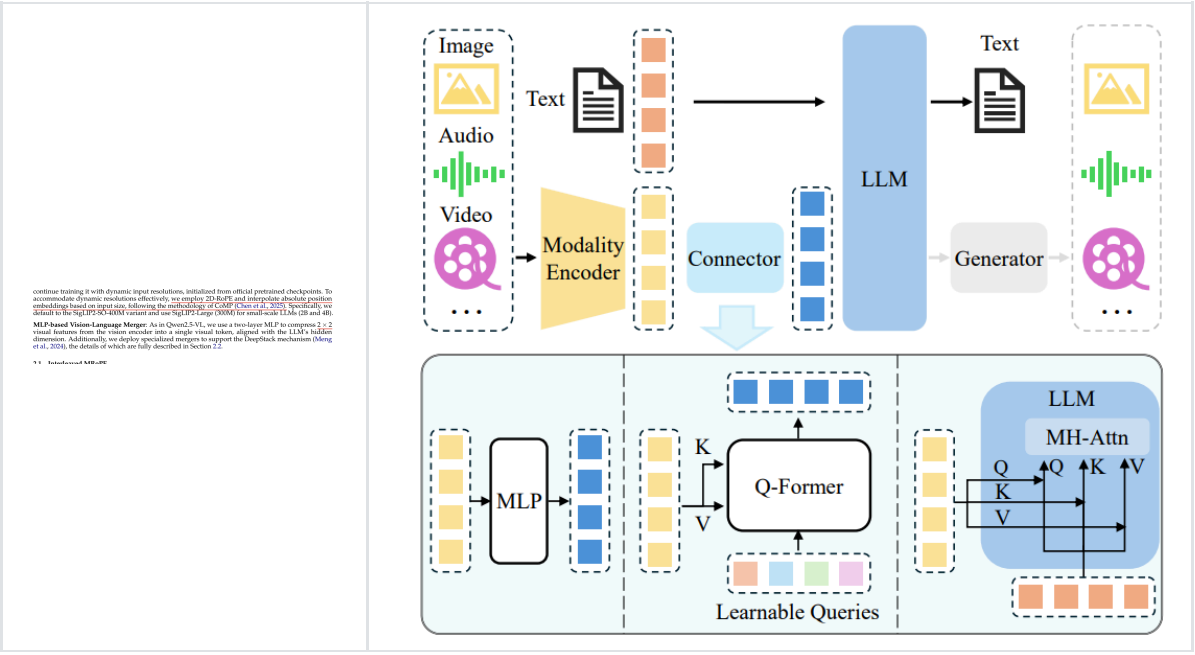
路径：**~/projects/Qwen3-VL/.venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/modeling_qwen3_vl.py**

VLModel 结构如下：

```python
class Qwen3VLModel(Qwen3VLPreTrainedModel):
    base_model_prefix = ""
    _checkpoint_conversion_mapping = {}
    # Reference: fix gemma3 grad acc #37208
    accepts_loss_kwargs = False
    config: Qwen3VLConfig
    _no_split_modules = ["Qwen3VLTextDecoderLayer", "Qwen3VLVisionBlock"]

    def __init__(self, config):
        super().__init__(config)
        self.visual = Qwen3VLVisionModel._from_config(config.vision_config)
        self.language_model = Qwen3VLTextModel._from_config(config.text_config)
        self.rope_deltas = None  # cache rope_deltas here

        # Initialize weights and apply final processing
        self.post_init()
```

Qwen3VLModel class 包含**visual model和language_model。**

根据Qwen3Vl paper和综述`A Survey on Multimodal Large Language Models`(如下图)



需要注意的是：Merger部分没有在init单独作为一个模块，而是包含在visual 内部，作为一个projection module。

4.1.3 前向传播

> 总体目标：理解architecture。

Path: .venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/modeling_qwen3_vl.py（Qwen3VLForConditionalGeneration class forward method）

```
outputs = self.model(
    input_ids=input_ids,
    pixel_values=pixel_values,
    pixel_values_videos=pixel_values_videos,
    image_grid_thw=image_grid_thw,
    video_grid_thw=video_grid_thw,
    position_ids=position_ids,
    attention_mask=attention_mask,
    past_key_values=past_key_values,
    inputs_embeds=inputs_embeds,
    cache_position=cache_position,
    **kwargs,
)
```

目前准备好的：input_ids，visual data

顺序：token_embedding -> visual encoder -> LLM decoder。

下面是Qwen3VLModel foward过程。

该forward的源码位置如下：

Path：.venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/modeling_qwen3_vl.py（**Qwen3VLModel的forward**）

首先，序列id被**token embedding**，每个token的维度是**4096**，得到**（1， 2768， 4096）**，具体代码如下。

```
if inputs_embeds is None:
    inputs_embeds = self.get_input_embeddings()(input_ids)
```

> 注意：每个id 被编码的维度是4096，包括了visual 填充的token id。这个填充token最终是需要被真实的visual embedding给替换的。因此，visual embedding最后会对齐到4096这个维度（Merger干的事情），然后把原本pad的位置给替换掉。

其次是对图像特征提取，即视觉encoder部分：

image encoder相比于2.5VL来说，不再是window+full，而是SigLIP2 arch，其实就是ViT。代码如下，

Path：Qwen3VLModel forward (line 1137)

```
if pixel_values is not None:
    image_embeds, deepstack_image_embeds = self.get_image_features(pixel_values,
image_grid_thw)
    image_embeds = torch.cat(image_embeds, dim=0).to(inputs_embeds.device,
inputs_embeds.dtype)
    image_mask, _ = self.get_placeholder_mask(
        input_ids, inputs_embeds=inputs_embeds, image_features=image_embeds
    )
    inputs_embeds = inputs_embeds.masked_scatter(image_mask, image_embeds)
```

encoder的过程在 self.get_image_features(pixel_values, image_grid_thw) 中。代码如下，

Path: Qwen3VLModel forward (line 1050)

```
def get_image_features(self, pixel_values: torch.FloatTensor, image_grid_thw:
Optional[torch.LongTensor] = None):
    ...
    image_embeds, deepstack_image_embeds = self.visual(pixel_values,
grid_thw=image_grid_thw)
    split_sizes = (image_grid_thw.prod(-1) //
self.visual.spatial_merge_size**2).tolist()
    image_embeds = torch.split(image_embeds, split_sizes)
    return image_embeds, deepstack_image_embeds
```

可以看到，调用了`self.visual`的forward，在4.1.2节中可知，`self.visual`是Qwen3VLVisionModel。

> 提醒一下：视觉输入pixel_value（11008, 1536）输入到Qwen3VLVisionModel中。（like ViT）

接下来进入到QwenQwen3VLVisionModel forward，查看**image encoder**的具体过程。

Path: .venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/modeling_qwen3_vl.py Qwen3VLVisionModel forward (line 714)

> 思路是：对输入patch embedding->添加绝对位置编码->2d RoPE-> Encoder Blocks forward pass
>
> 下面代码块中有一些变量的解释如下：
>
> - pos_embeds 是**可学习的绝对位置嵌入**（通过 fast_pos_embed_interpolate 插值出来），直接加到 hidden_states 上，让视觉 token 带上全局位置偏置。
> - rotary_pos_emb 是**旋转位置编码（RoPE）**，不是加到 hidden states 上，而是作为 **注意力里的相对位置相位** 使用（position_embeddings = (cos, sin)），注入到 Q/K 的旋转操作里。
> - RoPE：这个函数要为视觉 token 生成二维 RoPE 位置编码。做法是先构造每个 patch 在图像上的 (row, col) 坐标，再用这两个坐标去查 rotary_pos_emb 生成的频率表，得到每个 token 的二维旋转位置向量。

```
def forward(self, hidden_states: torch.Tensor, grid_thw: torch.Tensor, **kwargs)
-> torch.Tensor:
        # part 1
        hidden_states = self.patch_embed(hidden_states)
        # part 2
        pos_embeds = self.fast_pos_embed_interpolate(grid_thw)
```

```
        hidden_states = hidden_states + pos_embeds

        rotary_pos_emb = self.rot_pos_emb(grid_thw)

        seq_len, _ = hidden_states.size()
        hidden_states = hidden_states.reshape(seq_len, -1)
        rotary_pos_emb = rotary_pos_emb.reshape(seq_len, -1)
        emb = torch.cat((rotary_pos_emb, rotary_pos_emb), dim=-1)
        position_embeddings = (emb.cos(), emb.sin())

        cu_seqlens = torch.repeat_interleave(grid_thw[:, 1] * grid_thw[:, 2],
grid_thw[:, 0]).cumsum(
            dim=0,
            # Select dtype based on the following factors:
            #  - FA2 requires that cu_seqlens_q must have dtype int32
            #  - torch.onnx.export requires that cu_seqlens_q must have same
dtype as grid_thw
            # See https://github.com/huggingface/transformers/pull/34852 for more
information
            dtype=grid_thw.dtype if torch.jit.is_tracing() else torch.int32,
        )
        cu_seqlens = F.pad(cu_seqlens, (1, 0), value=0)

        # part 3
        deepstack_feature_lists = []
        for layer_num, blk in enumerate(self.blocks):
            hidden_states = blk(
                hidden_states,
                cu_seqlens=cu_seqlens,
                position_embeddings=position_embeddings,
                **kwargs,
            )
            if layer_num in self.deepstack_visual_indexes:
                deepstack_feature =
self.deepstack_merger_list[self.deepstack_visual_indexes.index(layer_num)](
                    hidden_states
                )
                deepstack_feature_lists.append(deepstack_feature)
        # part 4
        hidden_states = self.merger(hidden_states)

        return hidden_states, deepstack_feature_lists
```

关键步骤先总结为以下几点，然后逐一详细解释：

1. patch_embedding: 打patch还停留在RGB的input level，like ViT，先对其进行一次编码，称为patch embedding。
2. 分别计算绝对位置编码和相对位置编码
3. for loop foward pass，收集deepstack feature（需要merger到LLM的特征维度）。
4. 最后的visual hidden_state被merger到LLM的维度。

---

part 1

首先，对于patch_embedding，实际上就是通过了3D卷积实现，dim从1536变成1152。

不同点：bias=True

```
nn.Conv3d(3, 1152, kernel_size=(2,16,16), stride=(2,16,16), bias=True)
```

part 2

其次，关于位置编码：

对于绝对位置编码，

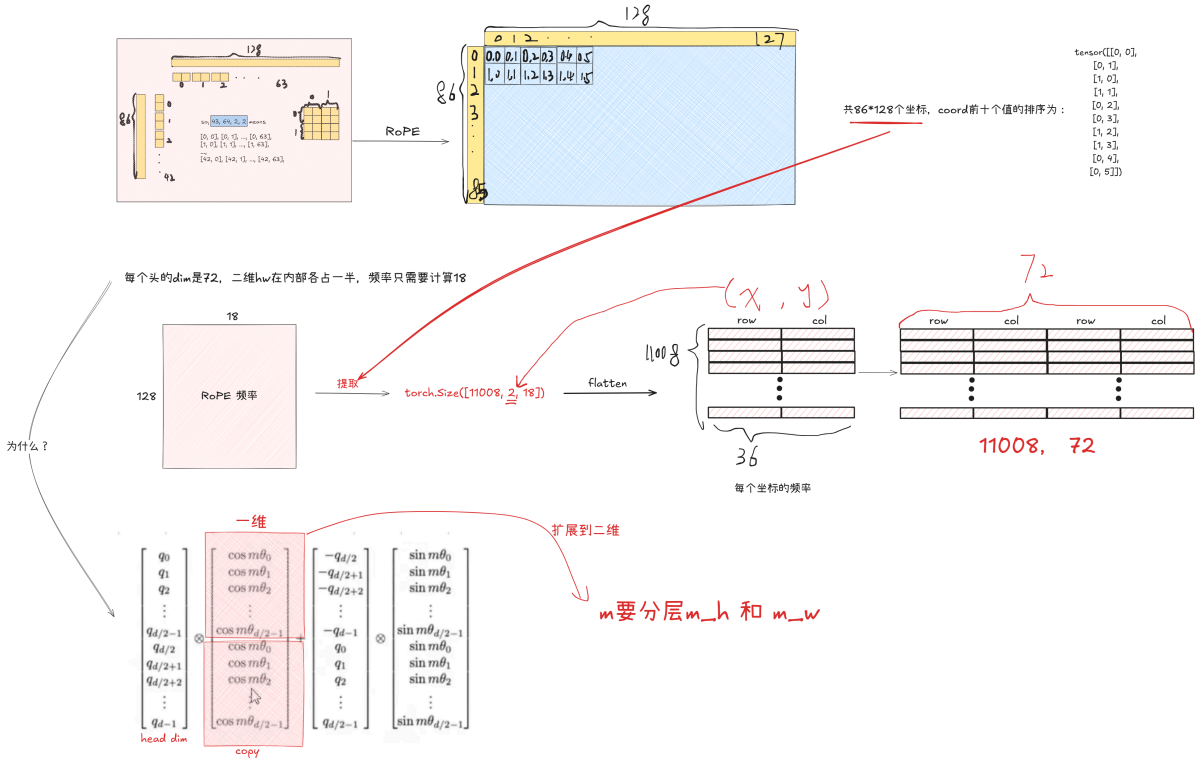
pos_embeds 是**可学习的绝对位置嵌入**（通过 fast_pos_embed_interpolate 插值出来），直接加到 hidden_states 上，让视觉 token 带上全局位置偏置。

对于图像的 2D-RoPE (not MRoPE)，

重点在于：

- freq表的大小是torch.Size([128, 18])（128是h，w的最大值，18是dim）
- coord的频率如何获取

可通过下图理解：



part 3

block forward pass（比较常规）

Path: .venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/modeling_qwen3_vl.py (line 260)

> 注意：Qwen3VLVisionBlock 类中的norm layer 使用的是layernorm
>
> self.norm1 = nn.LayerNorm(config.hidden_size, eps=1e-6)
>
> self.norm2 = nn.LayerNorm(config.hidden_size, eps=1e-6)

改进一：deepstack（在index等于8、16、24的时候收集）

```
if layer_num in self.deepstack_visual_indexes:
    deepstack_feature =
self.deepstack_merger_list[self.deepstack_visual_indexes.index(layer_num)](
        hidden_states
    )
    deepstack_feature_lists.append(deepstack_feature)
```

hidden_states传递的shape是(11008, 1152), **Merger类的代码以及论文原文**描述如下：

Path: .venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/modeling_qwen3_vl.py (line 93)

```
路径：merger class
class Qwen3VLVisionPatchMerger(nn.Module):
    def __init__(self, config: Qwen3VLVisionConfig, use_postshuffle_norm=False) -
> None:
        super().__init__()
        self.hidden_size = config.hidden_size * (config.spatial_merge_size**2)
        self.use_postshuffle_norm = use_postshuffle_norm
        self.norm = nn.LayerNorm(self.hidden_size if use_postshuffle_norm else
config.hidden_size, eps=1e-6)
        self.linear_fc1 = nn.Linear(self.hidden_size, self.hidden_size)
        self.act_fn = nn.GELU()
        self.linear_fc2 = nn.Linear(self.hidden_size, config.out_hidden_size)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.norm(x.view(-1, self.hidden_size) if self.use_postshuffle_norm
else x).view(-1, self.hidden_size)
        x = self.linear_fc2(self.act_fn(self.linear_fc1(x)))
        return x
```

MLP-based Vision-Language Merger: As in Qwen2.5-VL, we use a two-layer MLP to compress 2×2 visual features from the vision encoder into a single visual token, aligned with the LLM's hidden dimension. Additionally, we deploy specialized mergers to support the DeepStack mechanism (Meng et al., 2024), the details of which are fully described in Section 2.2.

将**2*2**的网格合并来压缩visual token，对齐到LLM dim。

具体维度变换如下，

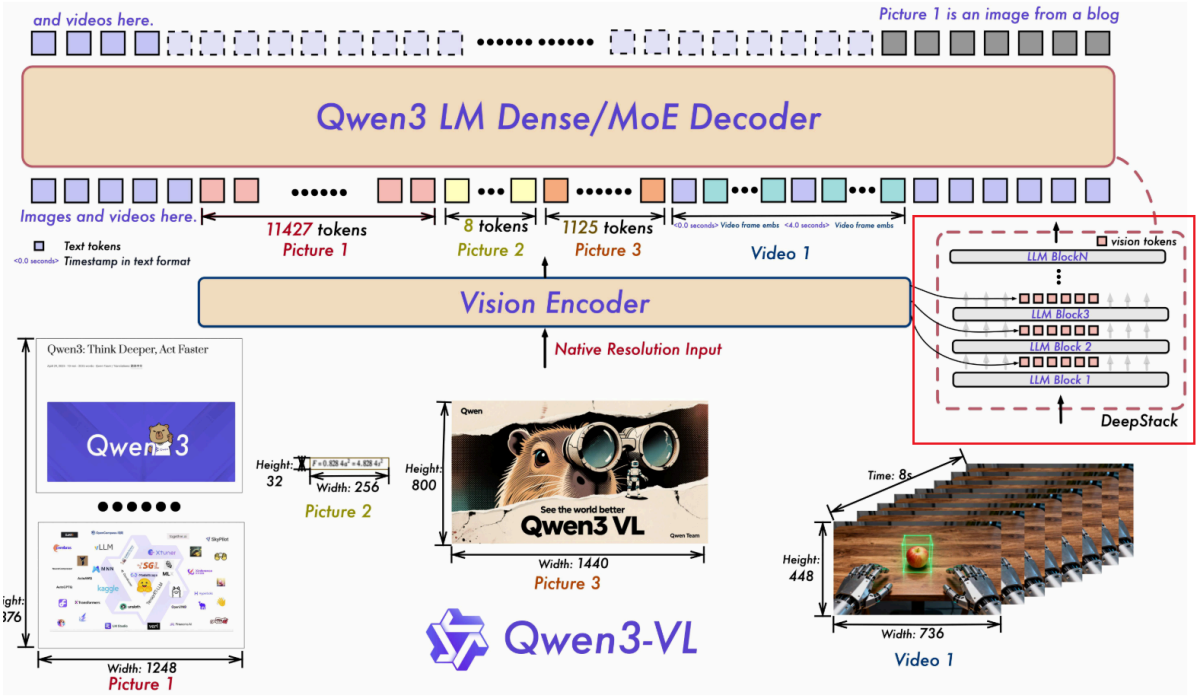
$$(11008, 1152) \rightarrow (\frac{11008}{2*2}, 1152 * (2*2)) = torch.Size([2752, 4608]),$$

然后使用MLP 模块$proj$到LLm的$dim = 4096$。

最后输出的维度是：$(2752, 4096)$

part 4

最后用一个列表deepstack_feature_lists去保存，然后最后一层输出的hidden_states也会经过一个merger（Qwen3VLVisionPatchMerger）将维度对齐到LLM



视觉encoder输出后，现在就可以实现将视觉嵌入填充到文本序列中，把原本pad 的部分给替换掉了。

> 前面提到：输出的visual feature (image_embeds) 的维度是(2752, 4096)。而在输入的文本embedding (inputs_embeds) 中对于visual 的pad 个数也一定是2752（总长是2768）

Path: .venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/modeling_qwen3_vl.py (line 1143)

```
inputs_embeds = inputs_embeds.masked_scatter(image_mask, image_embeds)
```

到此为止，visual encoder 和 Merger部分的处理就结束了。

接下来是LLM decoder部分，我们需要将整个序列（包含多模态信息）扔进decoder。

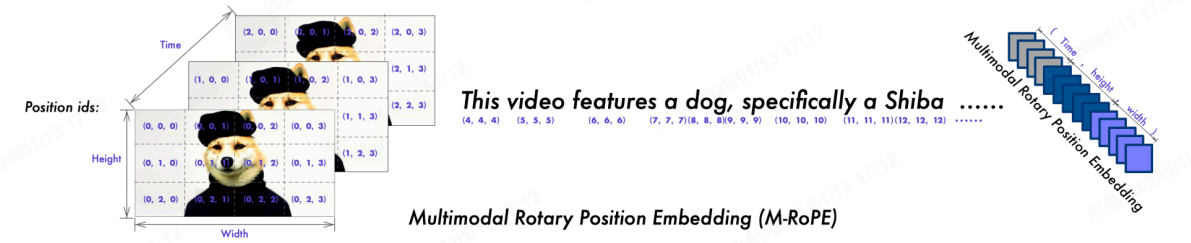
入口函数如下：

Path: .venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/modeling_qwen3_vl.py (line 1223)

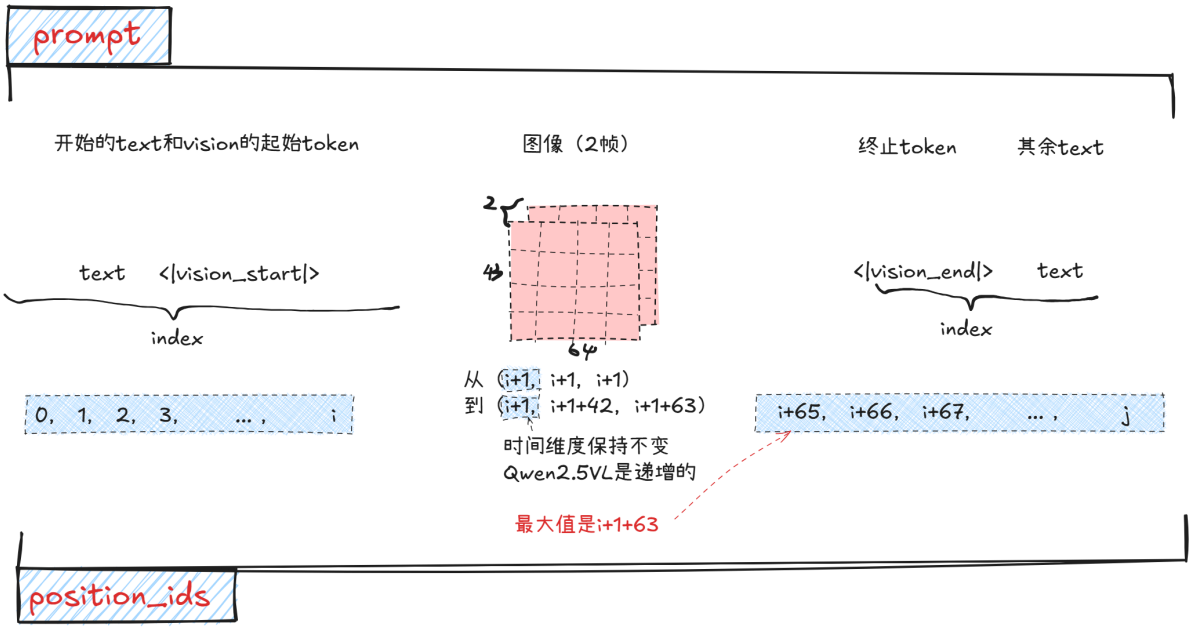
```
outputs = self.language_model(
    input_ids=None,
    position_ids=position_ids,
    attention_mask=attention_mask,
    past_key_values=past_key_values,
    inputs_embeds=inputs_embeds,
    cache_position=cache_position,
    visual_pos_masks=visual_pos_masks,
    deepstack_visual_embeds=deepstack_visual_embeds,
    **kwargs,
)
```

进入该函数之前，**position_ids**还没有得到，该参数指的是整个序列在LLM decoder中使用**interleaved MRoPE**时需要的位置idx。

所以需要先获取到这些id（就像下图的id）。（Qwen3VL和下图Qwen2VL相比，id获取是一样的，只不过加了timestamp。更直观的图解请直接看4.2.2节，不放在这里是因为image比较简单，不容易深刻理解区别在哪）



对图像来说网格大小是tensor([[1，86, 128]])，但是要4个网格为一组压缩，因此对齐到LLM的网格是（43，64）。具体流程如下图：



Path: .venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/modeling_qwen3_vl.py (line 1201)

```
position_ids, rope_deltas = self.get_rope_index(
        input_ids,
        image_grid_thw,
        video_grid_thw,
        attention_mask=attention_mask_tensor,
 )
# 最终得到torch.Size([3, 1, 2768]),取position_ids[:,0]就得到第一个位置的idx（t,h,w）.
```

最后进入language_model。

> 主要就两点：interleaved MRoPE和decoder block。

重点解释**interleaved MRoPE**。

对整个序列使用**interleaved MRoPE**编码，输入为2个：

- 序列的embedding（总长度2768，其中视觉占用了2752）（实际上只需要其dtype）
- 序列的位置id（input_ids）。

先通过下面的函数进入：

Path: .venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/modeling_qwen3_vl.py (line 846)

```
position_embeddings = self.rotary_emb(inputs_embeds, position_ids)
```

跳转到如下方法：

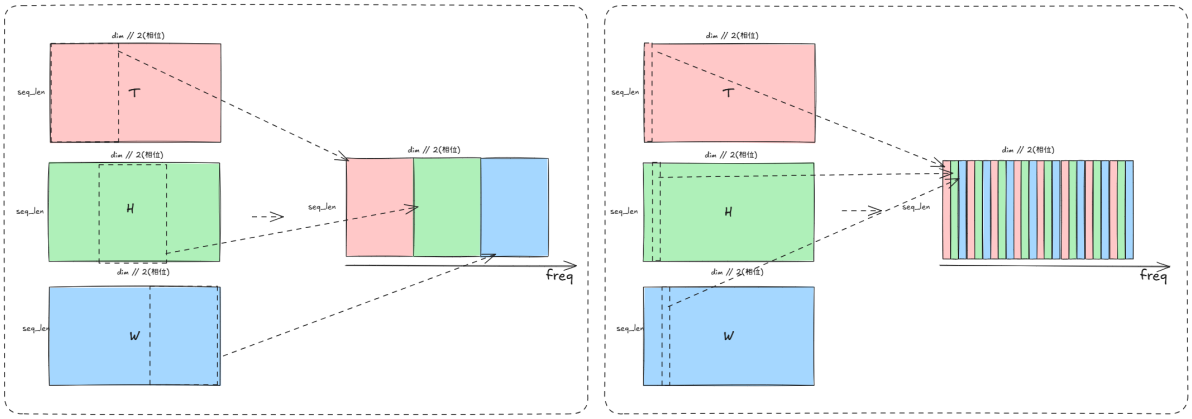
Path: .venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/modeling_qwen3_vl.py (line 846)

```
def forward(self, x, position_ids):
        # In contrast to other models, Qwen3VL has different position ids for the grids
        # So we expand the inv_freq to shape (3, ...)
        if position_ids.ndim == 2:
            position_ids = position_ids[None, ...].expand(3, position_ids.shape[0], -1)
        inv_freq_expanded = self.inv_freq[None, None, :, None].float().expand(3, position_ids.shape[1], -1, 1)# torch.Size([3, 1, 64, 1])
        position_ids_expanded = position_ids[:, :, None, :].float()  # shape (3, bs, 1, positions)

        device_type = x.device.type if isinstance(x.device.type, str) and x.device.type != "mps" else "cpu"
        with torch.autocast(device_type=device_type, enabled=False):  # Force float32
            freqs = (inv_freq_expanded.float() @ position_ids_expanded.float()).transpose(2, 3)
            freqs = self.apply_interleaved_mrope(freqs, self.mrope_section)
            emb = torch.cat((freqs, freqs), dim=-1)
            cos = emb.cos() * self.attention_scaling
            sin = emb.sin() * self.attention_scaling

        return cos.to(dtype=x.dtype), sin.to(dtype=x.dtype)
```

inv_freq_expanded是指频率表（decoder head_dim=128, 而序列是一维，因此编码一半=64，复制一份变成128即可）

freqs的维度：torch.Size([3, 1, 2768, 64])。 现在的**3代表了(t, h, w)。**

求出三个模态的频率值之后就可以拼接了，interleave的关键思想如下：



参考资料：https://zhuanlan.zhihu.com/p/1978253449704543933

最后，进入LLM decoder

```
for layer_idx, decoder_layer in enumerate(self.layers):
    layer_outputs = decoder_layer(
        hidden_states,
        attention_mask=attention_mask,
        position_ids=text_position_ids,
        past_key_values=past_key_values,
        cache_position=cache_position,
        position_embeddings=position_embeddings,
        **kwargs,
    )
    hidden_states = layer_outputs

    # add visual features to the hidden states of first several layers
    if deepstack_visual_embeds is not None and layer_idx in
range(len(deepstack_visual_embeds)):
        hidden_states = self._deepstack_process(
            hidden_states,
            visual_pos_masks,
            deepstack_visual_embeds[layer_idx],
        )

hidden_states = self.norm(hidden_states)

return BaseModelOutputWithPast(
    last_hidden_state=hidden_states,
    past_key_values=past_key_values,
)
```

里面QK的head_dim=128,和RoPE 编码后计算attn。

然后在计算的前三层，会对hidden_states进行deepstack融合

```
    路径：接着上面textmodel
    def _deepstack_process(
        self, hidden_states: torch.Tensor, visual_pos_masks: torch.Tensor,
visual_embeds: torch.Tensor
    ):
        visual_pos_masks = visual_pos_masks.to(hidden_states.device)
        visual_embeds = visual_embeds.to(hidden_states.device,
hidden_states.dtype)
        local_this = hidden_states[visual_pos_masks, :].clone() + visual_embeds
        hidden_states[visual_pos_masks, :] = local_this
        return hidden_states
```

> decoder中的norm采用RMS，而不是layernorm

4.2 video 模态

video模态和image模态的不同点在于**时间维度**的处理，包括：

1. 预处理阶段怎么对video加载、怎么对video预处理。（**重点：timestamp形式的tokenize**）
2. 前向传播时对video的时间维度怎么处理。（**重点：interleaved MRoPE**）

4.2.1 预处理

> 总体目标：将输入的 messages 全部 tokenize，得到LLM形式的输入序列id： **input ids**。

当处理video模态时，输入的messages需要修改为如下格式（**type从image改成video**）：

```
messages = [
    {
        "role": "user",
        "content": [
            {
                "type": "video",
                "video": "video_wechat.mp4"
            },
            {
                "type": "text",
                "text": "How long is this video?"
            }
        ]
    }
]
```

根据输入的video路径，会按照路径加载并预处理 video。（代码如下）

> 需要注意：加载的时候推荐指定backend == "torchcodec"，但该包和torch版本存在兼容问题，需要选择合适的版本，否则加载的时候会出现下面的问题：
>
> terminate called after throwing an instance of 'std::bad_alloc' what(): std::bad_alloc

Path: .venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/processing_qwen3_vl.py (line 170)

```
videos_inputs = self.video_processor(videos=videos,
**output_kwargs["videos_kwargs"])
```

为了方便理解，假设加载的视频的维度为：torch.Size([4, 3, 1280, 720])，

其中4表示帧数，也是模型要求的视频帧数的最小值，每一帧的图像大小为（3，1280，720）。

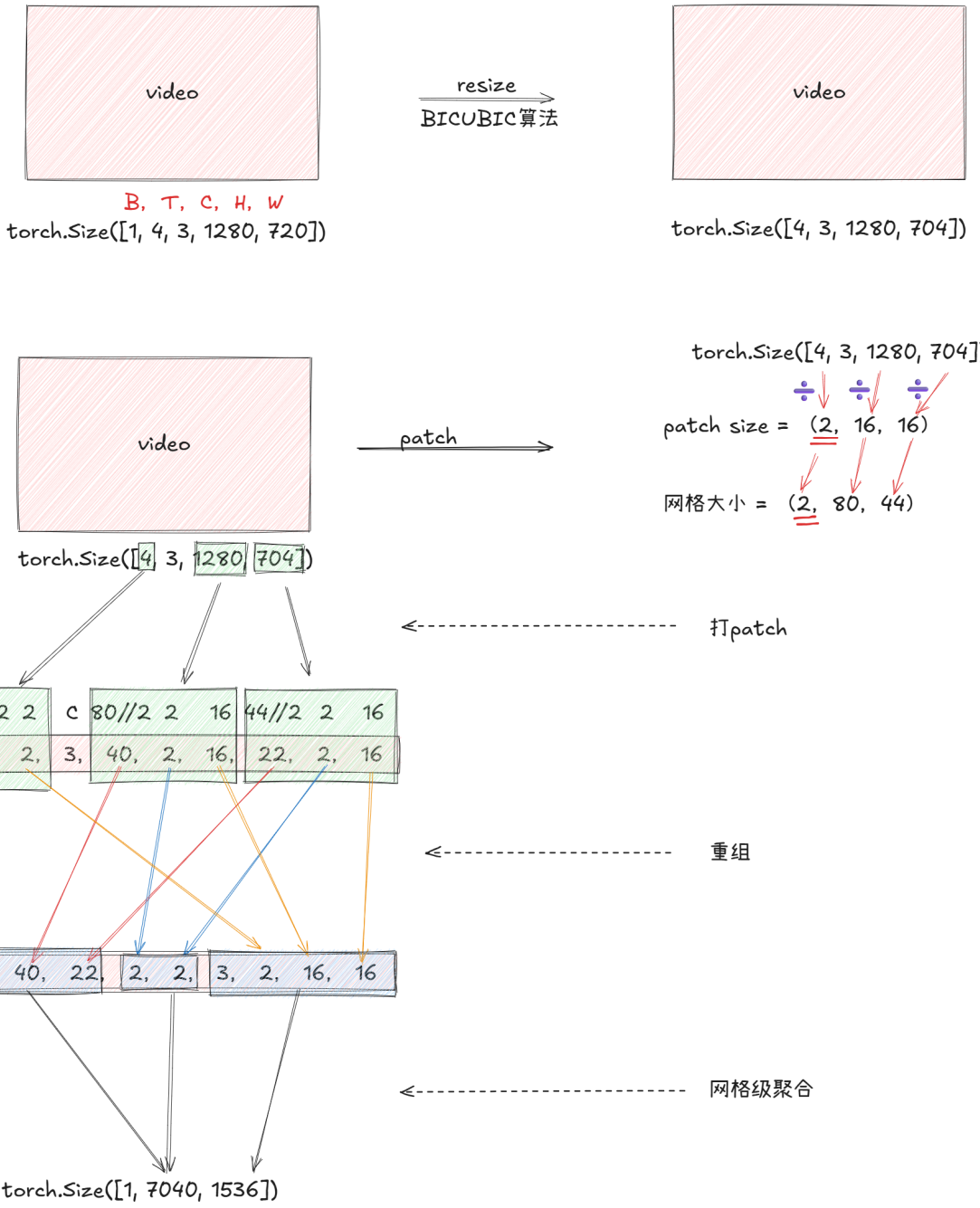
加载完成后，和image模态一样，也是通过`_preprocess`方法进行预处理。

Path: .venv/lib/python3.13/site-packages/transformers/video_processing_utils.py (line 387)

```
preprocessed_videos = self._preprocess(videos=videos, **kwargs)
```

预处理的流程也和image模态相同，都是：**resize -> norm -> 重组。**

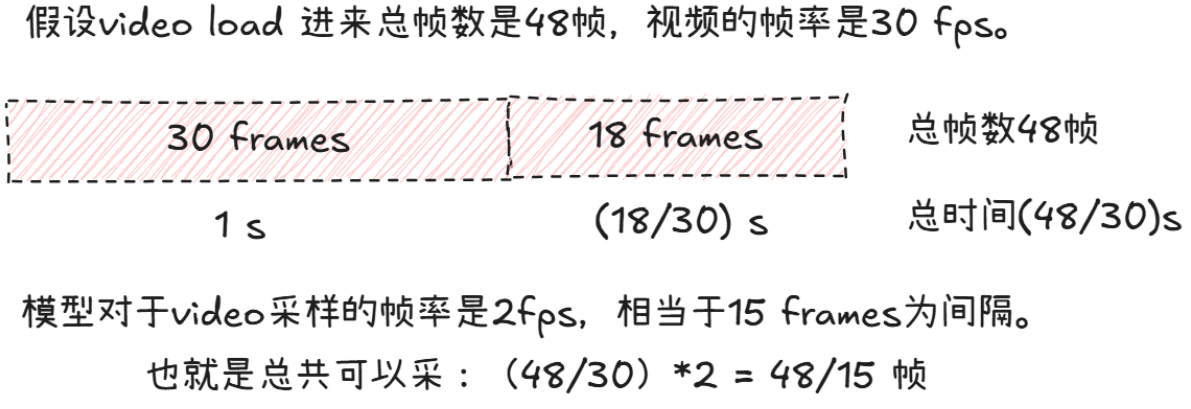
下图给出video的resize和重组过程，norm不会改变维度所以没有绘制。



重组完的video维度为:(7040, 1536)。其中**7040是指每四个网格合并为一组，一共有2 * 40 * 22组。**

重点：接下来详细解释Qwen3VL的创新：timestamp形式的tokenizer （替代了Qwen2.5VL的位置编码方式），以及video是如何填充pad token 的。

首先给出视频帧是如何采样的，如下图所示：



接下来需要根据以上信息计算出2个视频帧组的**timestamp。**(代码如下)

timestamp计算**实际上是计算每一组的平均时间戳**,

比如说，上图中0和16为一组，31和47为一组，

那么0和16这一组的平均时间戳就是: (0/30 + 16/30) / 2（30帧是一秒，俩时间的中间值）

为了方便理解，直接给出2个视频帧组的timestamp结果分别是：[0.3, 1.3]，时间上具有先后顺序。

Path: .venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/processing_qwen3_vl.py (line 211)

```
# if timestamps are not provided, calculate them
curr_timestamp = self._calculate_timestamps(
    metadata.frames_indices,
    metadata.fps,
    self.video_processor.merge_size,
)
```

然后，最关键的一步如下，做替换：

> 原本的prompt是：'<|im_start|>user\n<|vision_start|><|video_pad|><|vision_end|>How long is this video?<|im_end|>\n<|im_start|>assistant\n\n'

```
<|vision_start|><|video_pad|><|vision_end|>

替换成

f"<0.3 seconds>" + <|vision_start|> + 第一组视频帧的pad token个数（40 * 22）*
'<|video_pad|>' + <|vision_end|>
+
f"<1.3 seconds>" + <|vision_start|> + 第一组视频帧的pad token个数（40 * 22）*
'<|video_pad|>' + <|vision_end|>
```

作为两组各自的prompt。

代码如下：

Path: .venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/processing_qwen3_vl.py (line 217)

```
video_placeholder = ""
frame_seqlen = video_grid_thw[index][1:].prod() // merge_length
for frame_idx in range(video_grid_thw[index][0]):
    curr_time = curr_timestamp[frame_idx]
    video_placeholder += f"<{curr_time:.1f} seconds>"
    video_placeholder += (
        self.vision_start_token + "<|placeholder|>" * frame_seqlen +
self.vision_end_token
    )
```

然后和image模态一样将prompt映射到id就完成了。

4.2.2 前向传播

该部分只列出和image模态处理**不同**的地方，未列出的地方表示和image模态处理一致。

video提取编码器特征的函数就是image模态的函数(因为和image模拟出时间维度后可以兼容处理)：

Path: .venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/modeling_qwen3_vl.py (line 1035)

```
 def get_video_features(
        self, pixel_values_videos: torch.FloatTensor, video_grid_thw:
Optional[torch.LongTensor] = None
    ):
    return self.get_image_features(pixel_values_videos, video_grid_thw)
```

这个函数内部的流程是：

- patch embedding：把原本RGB的通道值编码成torch.Size([7040, 1152])。
- 添加绝对位置编码：可学习的embedding，插值到原图大小（时间维度直接复制），然后相加。
- 计算2dRoPE：因为本质上是**图像的编码器**，所以对于时间维度**全部都是复制。**

  Path: .venv/lib/python3.13/site-packages/transformers/models/qwen3_vl/modeling_qwen3_vl.py (line 631)

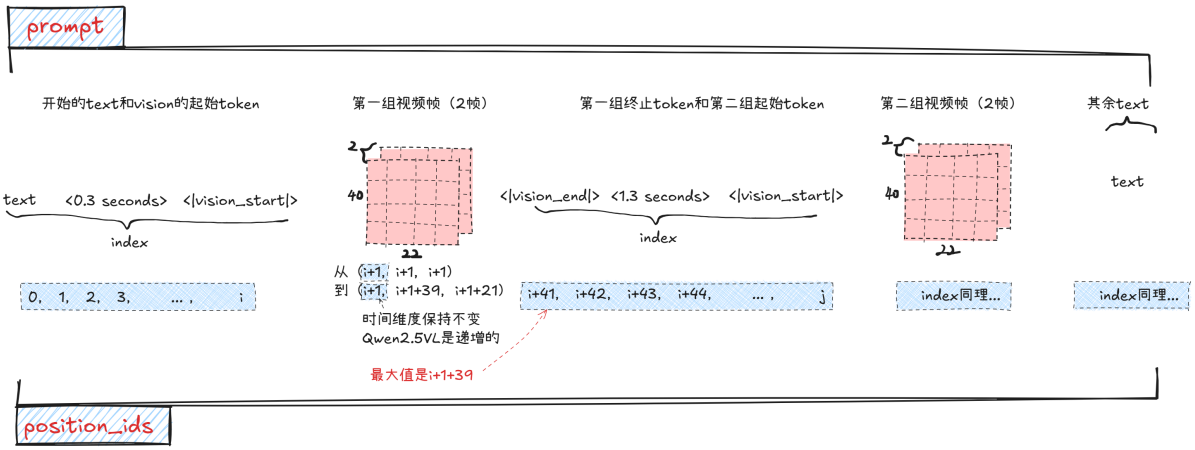
  ```
  if num_frames > 1:
      coords = coords.repeat(num_frames, 1)
      # coords是2d-rope需要用到的位置坐标。
      # 从(3520, 2) -> (7040, 2)，时间维度上没有特殊处理，直接复制。
  ```

  然后同样freq只需要18维，高度h给18维，宽度w给18维，再因为两帧一组，复制一份 -> 得到72，72就是encoder的head_dim。
- block前向传播+deepstack，这一步和image模态没有差别。
- 得到编码器特征之后，需要将特征替换掉当时用 '<|video_pad|>' 填充的位置。

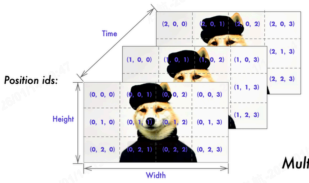
最后是LLM decoder部分。

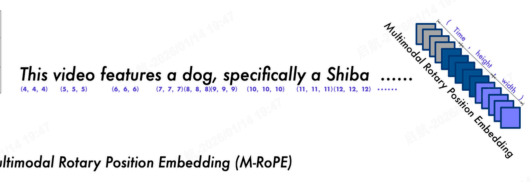
在此之前，为了计算MRoPE，和image模态一样，需要计算出video的**position_ids**。计算position_ids的方式如下：

- 对于video，前面打patch的时候计算出来网格的维度是**（2，80，44）**。这里面的2代表了2组，每一组里面有两个视频帧（例如之前提到的0和16，31和47）。而Qwen3-VL的观点是：使用timestamp（即平均时间戳）的方式对两个视频帧进行时间编码，无需Qwen2.5-VL那样（不同时间维度的t逐步递增），只需要采用相同值。**具体可以看下图：**



弄清楚上面这一点之后，再给出从Qwen2VL到Qwen3VL的历史过程（从image模态求position_ids跳转过来的看这里）：





Multimodal Rotary Position Embedding (M-RoPE)

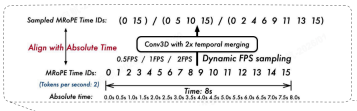
Qwen2-VL

在Qwen2 - VL中，MRoPE中的时间位置ID与输入帧数绑定在一起，并没有考虑视频中内容变化的速度或事件的绝对时序。

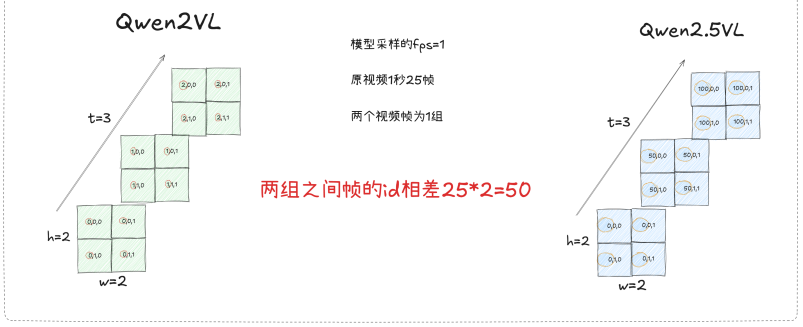
采样的帧之间时间差值通常不平均，且几乎不会固定为1,

0->1->2->...应该改成和绝对时间对齐的形式！

Qwen2.5-VL



看不懂没关系，看下面这个例子👇



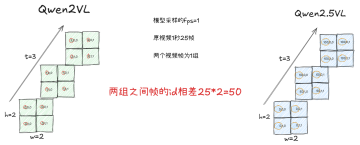
两组之间帧的id相差25*2=50

这个id太过于分散，长视频的大ID有长尾特性。

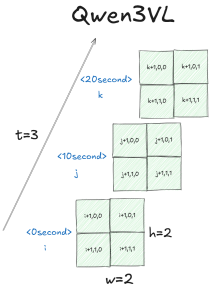
采样的fps不同，id一直在变，不利于训练！

用时间戳来表示每一组视频！

Qwen3VL



那么👉



假设三组视频的时间戳分别是：
0second，10second，20second

是的你没有看错，我和Qwen2VL很像。

- 计算完position_ids之后，计算interleaved MroPE的方式和image相同。
- LLM forward过程和image一致。