Qwen2-VL 模型结构和万字源码解析

xinxin

接上一篇Qwen-VL模型介绍

xinxin：Qwen-VL、Qwen2-VL论文阅读记录
36 赞同 · 0 评论 文章

本文主要对Qwen2-VL进行解析模型结构。下面以7B模型进行举例。

欢迎来点点赞哈~~

1. Qwen2-VL模型的组成部分

Qwen2-VL模型主要包含以下几个部分，结合官方infer代码的流程可以对应:

```
1. chat_template: 用于将输入转化为模型所需要的标准格式，如ChatML格式；
2. processor:
    image_process:对图像进行预处理，将图像转化为模型所需要的格式，如切分patch操作；
    tokenizer: 文本prompt处理和tokens预定义；
    数据整合
3. prepare_inputs: 准备model_inputs,用于输入到model中
4. model:
    vision model: vision提取特征信息；
    Embedding: prompt Embedding;
    Scatter: 将visiion embedding tokens嵌入到prompt tokens中
    LLM: Qwen大语言模型
```

2. Qwen2-VL模型结构

本文沿着上面组成部分顺序进行讲解。

2.1 chat_template处理

Qwen2-VL采用ChatML格式template。首先加载好MODEL_PATH, 执行processor.chat_template即可查看Qwen2-VL的模版形式。本文通过下面举一个例子进行:

```
# 加载模型
processor = AutoProcessor.from_pretrained(MODEL_PATH, min_pixels=min_pixels, max_pixel
print(processor.chat_template)
# 设置conversation
prompt = "请描述这两张图片"
conversation = [
        {
            "role": "user",
            "content": [
                {"type": "image", "image": "./0001.png"},
                {"type": "image", "image": "./0002.png"},
                {"type": "text", "text": prompt},
            ],
        }
    ]
# 假设我们设置上面的conversation, 转化为template形式如下, 注意换行符也是一个token
print(processor.apply_chat_template(conversation))
```

```
<|im_start|>user\n
<|vision_start|><|image_pad|><|vision_end|><|vision_start|><|image_pad|><|vision_end|>
'''
print(processor.apply_chat_template(conversation, add_generation_prompt=True)) # 添加推
'''<|im_start|>system\n
You are a helpful assistant.<|im_end|>\n
<|im_start|>user\n
<|vision_start|><|image_pad|><|vision_end|><|vision_start|><|image_pad|><|vision_end|>
<|im_start|>assistant\n
'''
```

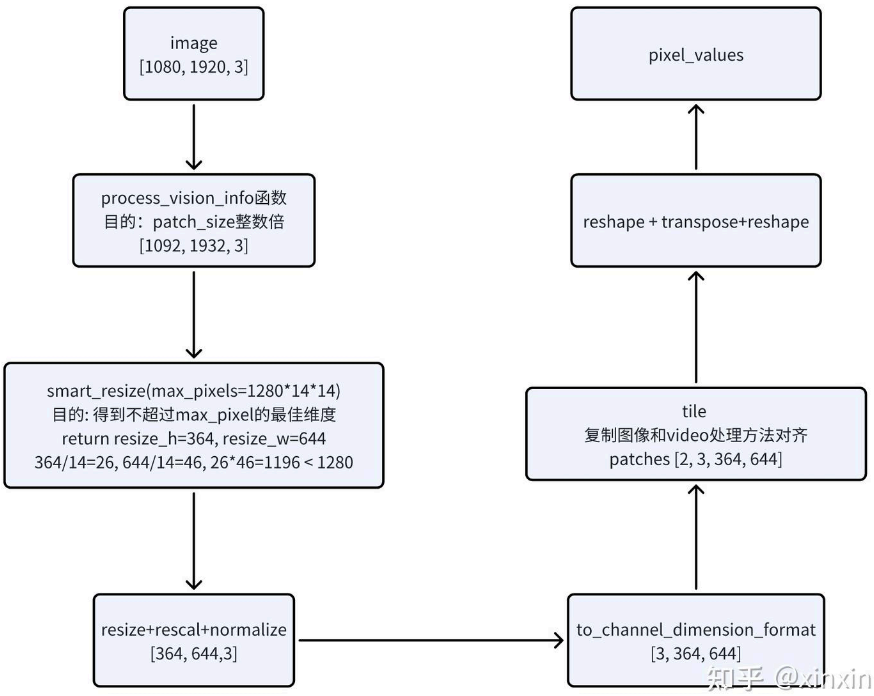
可以看到Qwen2-VL将图片编码为<|vision_start|><|image_pad|><|vision_end|>形式。

2.2 process 处理

process包含imge_processor图像处理和tokens的预定义处理；

2.2.1 image_processor

image_process位于image_processing_qwen2_vl.py中Qwen2VLImageProcessor类。以1080*1920图为例，处理流程：



Qwen2-VL在图像预处理主要两部分：

```
1. smart_resize：
   将图像的宽高reszie到patch_size的整数倍，比如(1080, 1920)的图像变为(1092, 1932);
   目的：patch_size的整数倍，patch数量不要超过ViT的上限。
2. 图片flatten到(patch_num_h * patch_num_w) 个 patch  (patch_size * patch_size)
```

2.2.1.1 smart_resize

第一个细节点：为什么要设置 `factor=14*2` 里面包含2，不直接等于28，`max_pixels=14*14*4*1280` 包含4，不直接设置14*14*5120?
smart_resize函数可以看到 `factor=14*2`, `max_pixels=14*14*4*1280` ,这个和论文Naive Dynamic Resolution方法(先按patch_size=14切分，然后在通过MLP对相邻的2x2的tokens进行特征合

```
# 生成patch_size正式倍的宽高，且位于min_pixels和max_pixels之间
resized_height, resized_width = smart_resize(
                    height, #原始图像宽高
                    width,
                    factor=self.patch_size * self.merge_size, # 14 * 2
                    min_pixels=self.min_pixels,
                    max_pixels=self.max_pixels,
                )
image = resize(
                    image, size=(resized_height, resized_width), resample=resample, in
                )

def smart_resize(
    height: int, width: int, factor: int = 28, min_pixels: int = 56 * 56, max_pixels:
):
    h_bar = round(height / factor) * factor
    w_bar = round(width / factor) * factor
    if h_bar * w_bar > max_pixels:
        beta = math.sqrt((height * width) / max_pixels)
        h_bar = math.floor(height / beta / factor) * factor
        w_bar = math.floor(width / beta / factor) * factor
    elif h_bar * w_bar < min_pixels:
        beta = math.sqrt(min_pixels / (height * width))
        h_bar = math.ceil(height * beta / factor) * factor
        w_bar = math.ceil(width * beta / factor) * factor
    return h_bar, w_bar
```

2.2.1.2 patch切分细节

第2个细节点： patch切分和reshape。为什么reshape成这个那么长串的维度，再transpose再reshape呢？为何不直接reshape成(1196, 1176)?
是为了后面MLP(2x2相邻token进行准备), 不让相邻的2x2patches分开, 如果直接reshape(grid_t * grid_h * grid_w, ...) 那么是按行进行切分，相邻的2x2的patch flatten后就不相连了；这样的切分方式可以保证2x2patch flatten后是相连的, 方便后面MLP的时候切分。

```
# self.temporal_patch_size = 2
# self.merge_size = 2
# patches.shape = (2, 3, H, W), 以h=364, w=644为例
grid_t = patches.shape[0] // self.temporal_patch_size # 1
# 宽高按patch_size进行切分数量, grid_h=26, grid_w=46
grid_h, grid_w = resized_height // self.patch_size, resized_width // self.patch_size
# patch shape: [1,2,3,13,2,23,2,14]
patches = patches.reshape(
    grid_t,
    self.temporal_patch_size,
    channel,
    grid_h // self.merge_size, # 注意：为什么再除self.merge_size，为了后面MLP(2x2相邻toke
    self.merge_size,
    self.patch_size,
    grid_w // self.merge_size,
    self.merge_size,
    self.patch_size,
    )
# 维度变换(1, 13, 23, 2, 2, 3, 2, 14, 14)
patches = patches.transpose(0, 3, 6, 4, 7, 2, 1, 5, 8)
# flatten_patches.shape = (1196, 1176)
flatten_patches = patches.reshape(
    grid_t * grid_h * grid_w, channel * self.temporal_patch_size * self.patch_size * s
)
```

```
# pixel_values.shape = (2392, 1176) 传了两幅图像
# vision_grid_thws = array([[ 1, 26, 46], [ 1, 26, 46]])
image_inputs = {"pixel_values": pixel_values, "image_grid_thw": vision_grid_thws}
```

2.2.2 tokens预定义

通过计算横纵patch的数量，将text中的 `<image_pad>` 替换为 `image_grid_thw[index].prod() // merge_length`个`<|placeholder|>` ,为什么需要除merge_length? 就是和MLP(2x2相邻patch)变成一个vision token有关。
然后再将所有的 `<|placeholder|>` 变成 `<image_pad>` .

```
# merge_length = 4
merge_length = self.image_processor.merge_size**2
index = 0
for i in range(len(text)):
    while self.image_token in text[i]:
        text[i] = text[i].replace(
            self.image_token, "<|placeholder|>" * (image_grid_thw[index].prod() // mer
        )
        index += 1
    '''
    text = <|im_start|>system\n
            You are a helpful assistant.<|im_end|>\n
            <|im_start|>user\n
            <|vision_start|><|placeholder|> * 26*46/4 <|vision_end|><|vision_start|><|
            <|im_start|>assistant\n
    '''
    text[i] = text[i].replace("<|placeholder|>", self.image_token)
    '''
    text = <|im_start|>system\n
            You are a helpful assistant.<|im_end|>\n
            <|im_start|>user\n
            <|vision_start|><image_pad> * 26*46/4 <|vision_end|><|vision_start|><image
            <|im_start|>assistant\n
    '''
# 将text的文本tokenizer编码为id的形式
# text_inputs.kyes = (['input_ids', 'input_ids'])
# input_ids=[[151664,...,198], attention_mask=[[1,..,1]]
text_inputs = self.tokenizer(text, **output_kwargs["text_kwargs"])
```

2.2.3 processor

经过前面image_processor得到image_inputs, text_prompt tokenizer得到text_inputs。最后processor将这两部分的信息合并起来得到inputs。

```
uts = {"pixel_values": pixel_values, "image_grid_thw": vision_grid_thws}
ts = {"input_ids": input_ids, "attention_mask": attention_mask}
{"input_ids": input_ids, "attention_mask": attention_mask, "pixel_values": pixel_values
```

2.3 model input数据准备

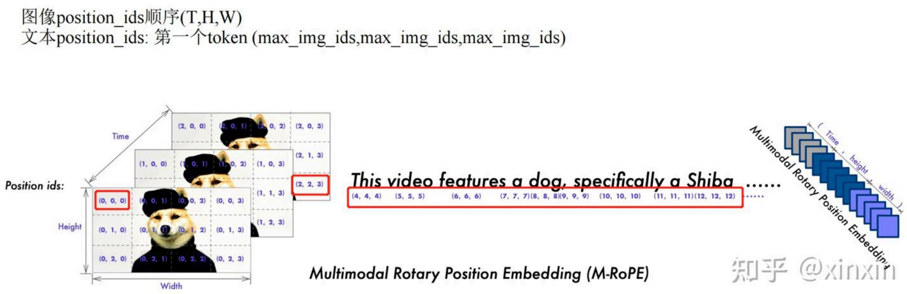
2.3.1 model_inputs变量

model_inputs：Qwen2VL模型的输入。
数据准备位于Qwen2VLForConditionalGeneration.prepare_inputs_for_generation()，得到model_inputs。

```
            "position_ids": position_ids, # 3D RoPE的位置编码index
            "past_key_values": past_key_values, # DynamicCache()用于储存KVCache的信息
            "use_cache": use_cache, # 是否采用cache
            "attention_mask": attention_mask, # 推理的attention mask
            "pixel_values": pixel_values, # images的patch 原始像素信息
            "pixel_values_videos": pixel_values_videos, #video的patch 原始像素信息
            "image_grid_thw": image_grid_thw, # images的patch数量信息
            "video_grid_thw": video_grid_thw, # video的patch数量信息
            "rope_deltas": rope_deltas, # rope的惩罚系数
        }
```

2.3.3 position_ids变量生成

作用：3D RoPE位置编码index，包含temprαol,height和width。
图像包含3个维度的index是不同的,而文本3个维度的index是一样的。如下图所示:

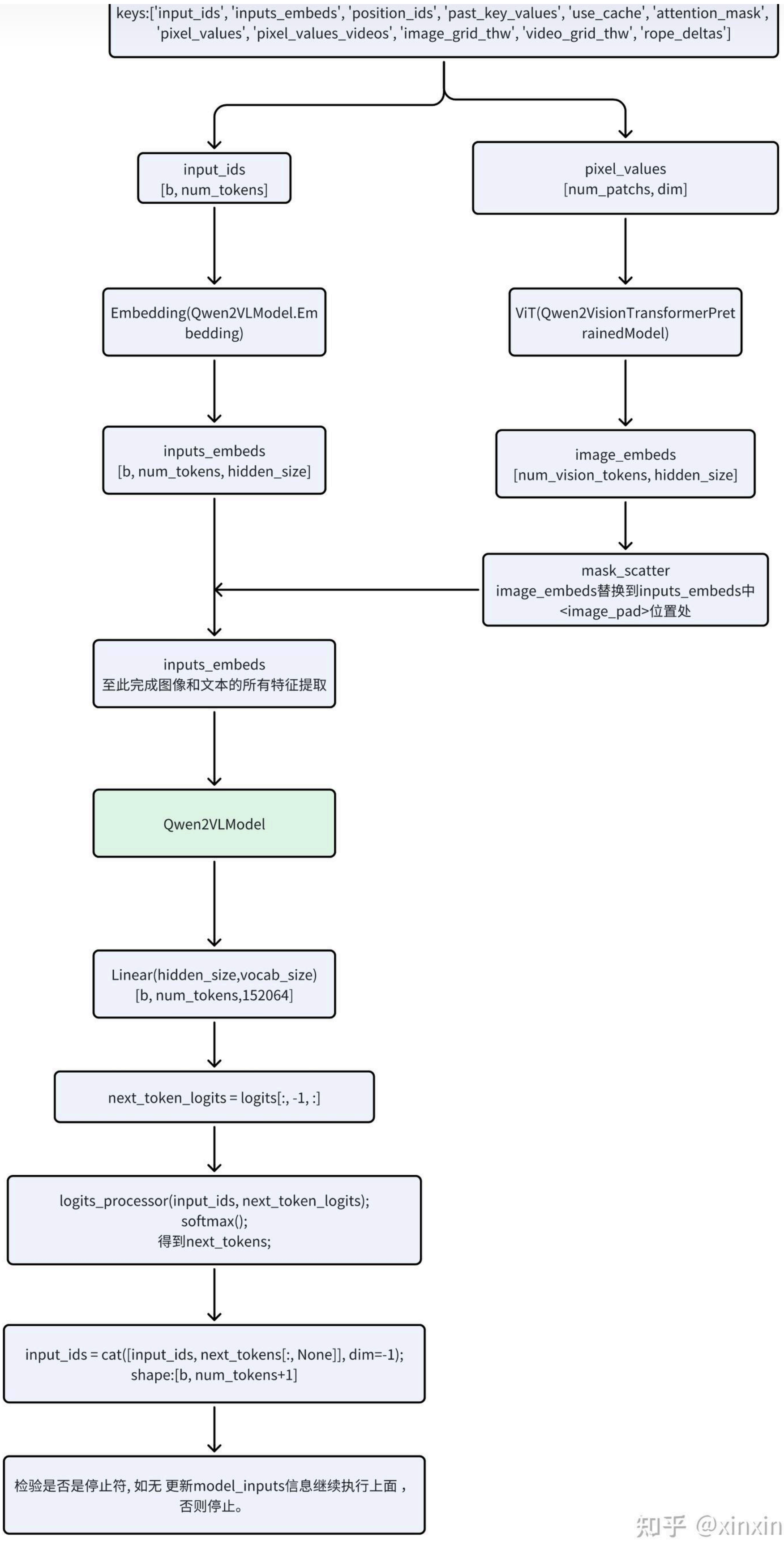



```
# 代码位于：Qwen2VLForConditionalGeneration.get_rope_index()
# 假设input_ids:[V V V V V V V V V V V V T T T T T], V表示vision的token <image_pad>, T表
# 计算图像和文本的 temproal, height和width的位置编码index
vision temporal position_ids: [0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2]
vision height position_ids: [0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1]
vision width position_ids: [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
text temporal position_ids: [3, 4, 5, 6, 7]
text height position_ids: [3, 4, 5, 6, 7]
text width position_ids: [3, 4, 5, 6, 7]
# 文本开始的position_idx是vision position_idx的最大值+1
# 最后将不同维度的图和文本的position_ids进行拼接，输出最终的position_ids，shape:[3, 1, num_t
```

2.4 model推理过程

2.4.1 Qwen2-VL主干

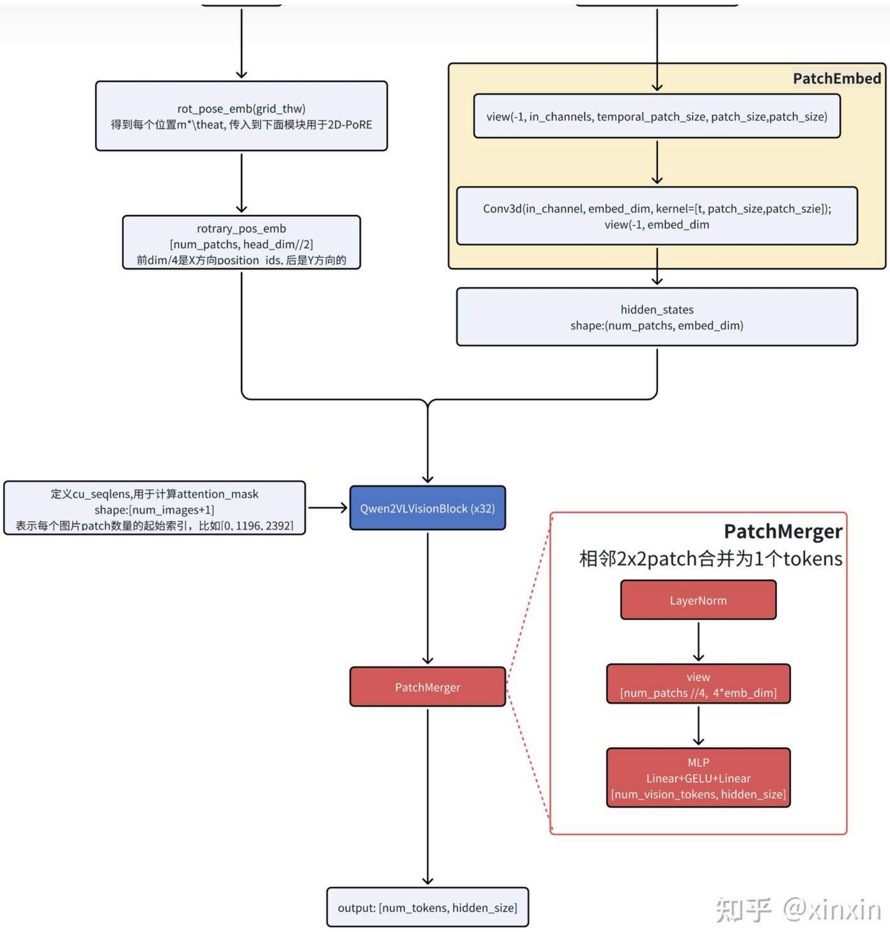
Qwen2-VL模型推理位于modeling_qwen2_vl.py中Qwen2VLForConditionalGeneration类。通过将上述的processorj将model_inputs输入到Qwen2VLForConditionalGeneration.forward中进行推理。
整体架构图如下：



Qwen2-VL模型整体结构

2.4.2 ViT(Qwen2VisionTransformerPretrainedModel)

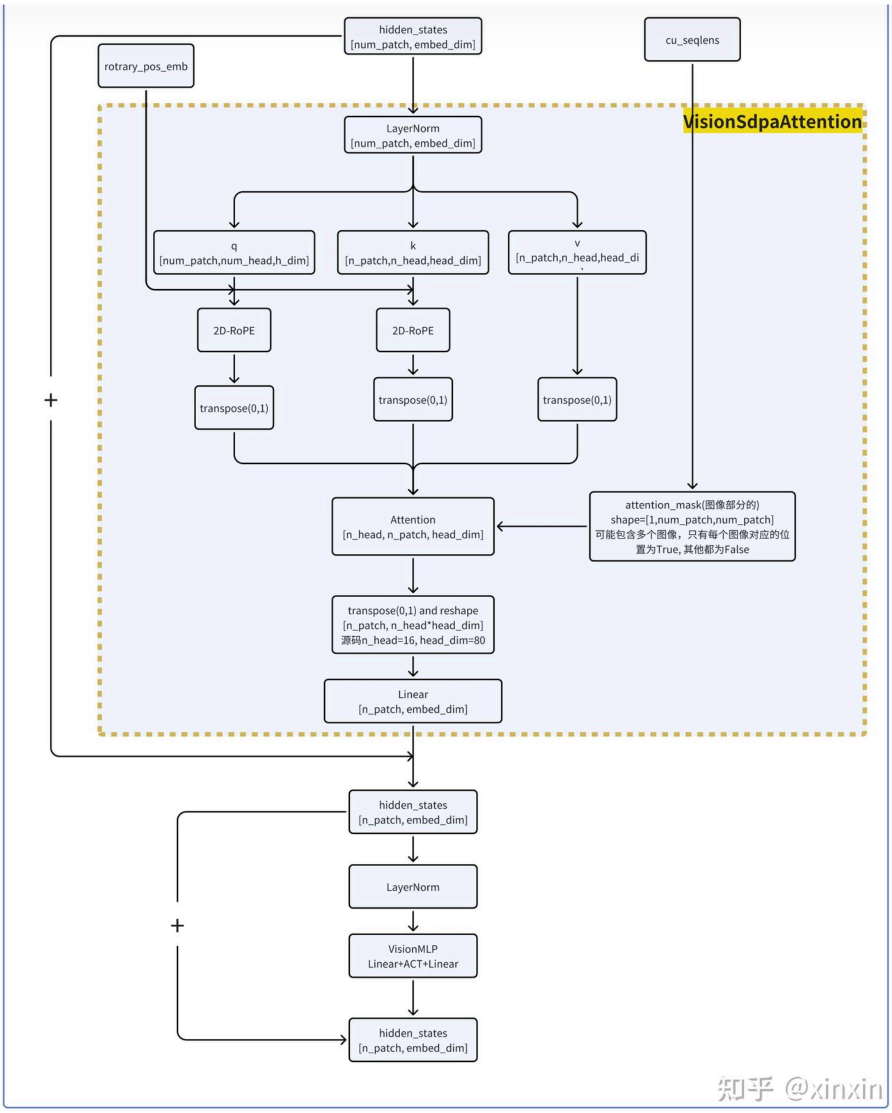
ViT是对image和video进行特征提取，其中包含2D旋转位置编码生成、PatchEmbed(时序Conv3D, 特征提取)、Qwen2VLVisionBlock(ViT tokens特征提取)、PatchMerger(降低vision token数量)。



ViT网络结构

2.4.2.1 Qwen2VLVisionBlock

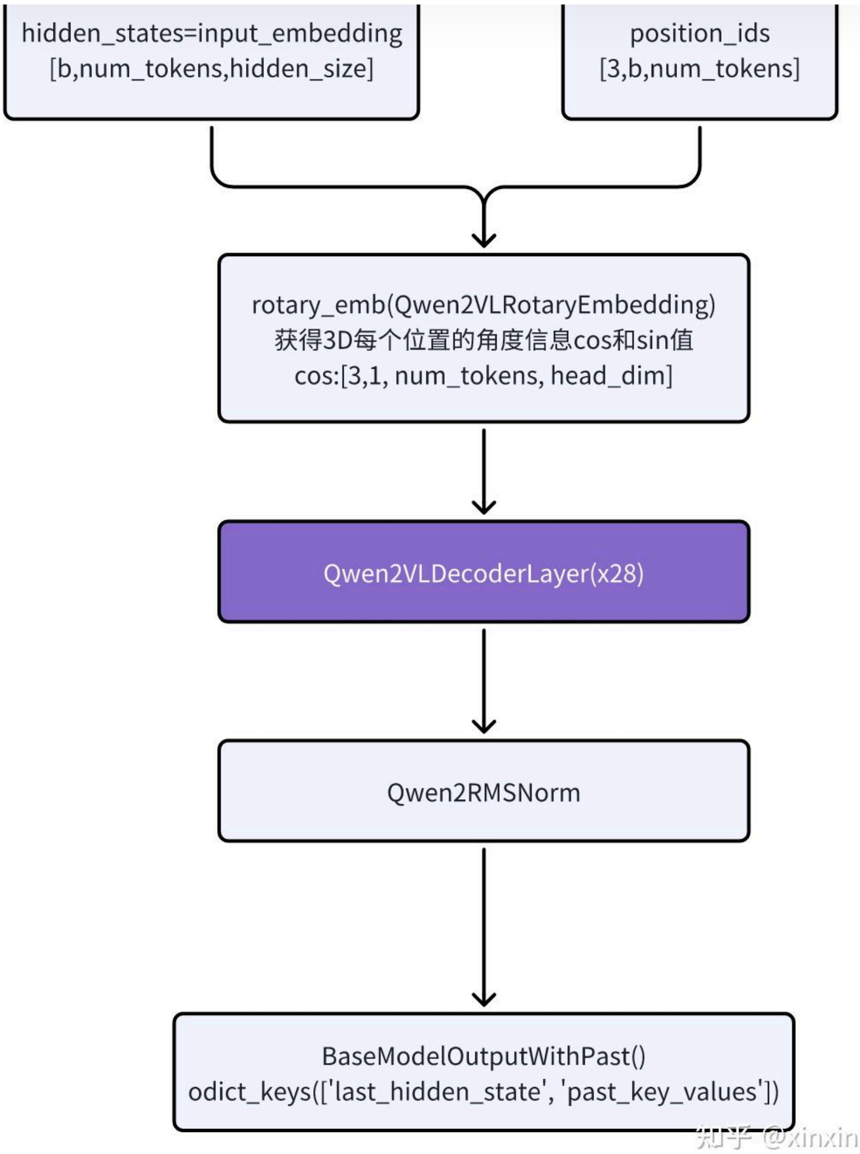
ViT中的Qwen2VLVisionBlock主要是VisionSdpaAttention构成，其中涉及2D-RoPE。



Qwen2VLVisionBlock

2.4.3 Qwen2VLModel

Qwen2VLModel生成模块的主干结构，主要包含3D位置编码的生成、DecoderLayer。结构和变量维度如下所示：

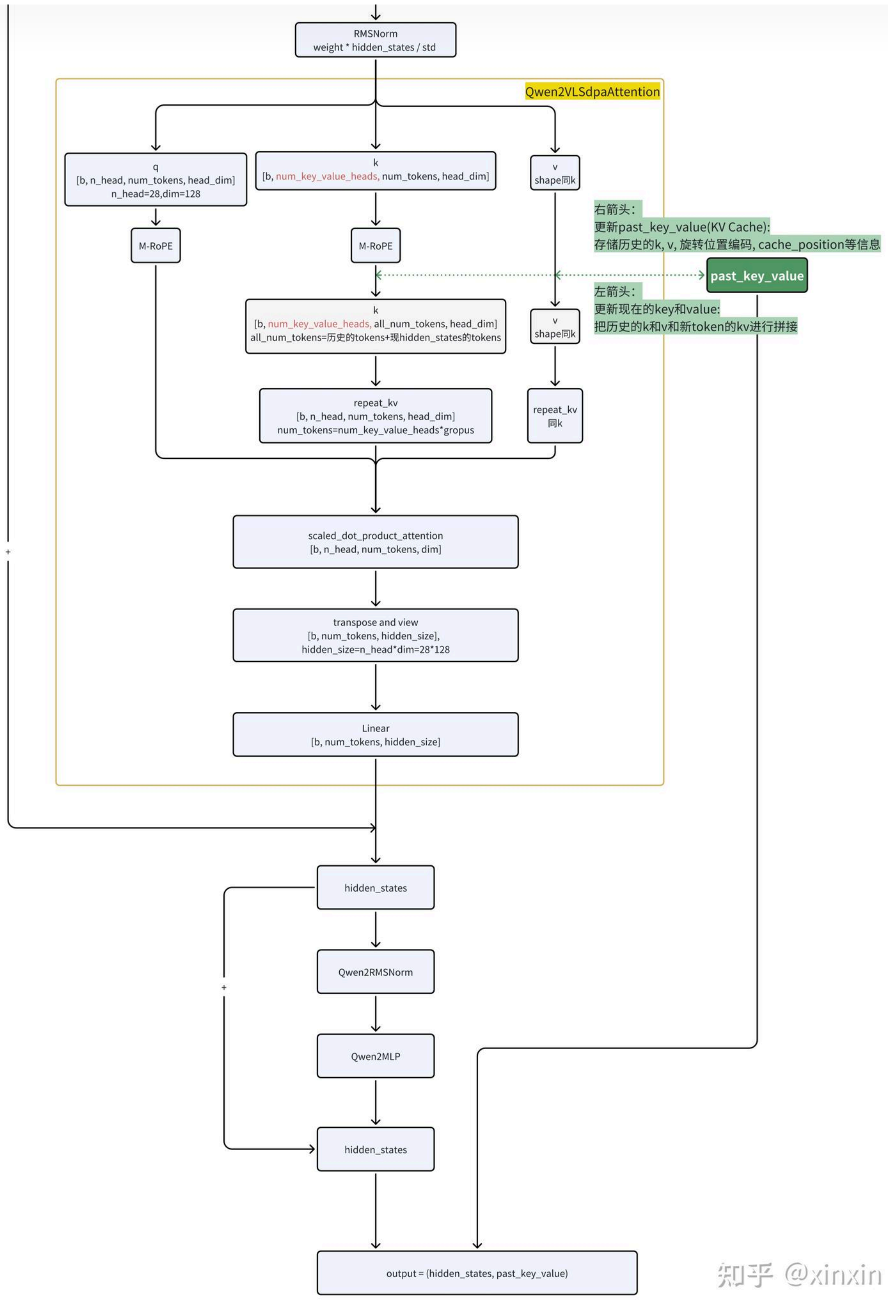


Qwen2VL Decoder主结构

2.4.3.1 Qwen2VLDecoderLayer

Qwen2VDecoderLayer主要包含Qwen2VLSdpaAttention和KV Cache等模块。

```
# 输入数据:shape
hidden_states:[b,num_tokens,hidden_size]
attention_mask:[b, num_tokens]
position_ids:[3,b,num_tokens]
past_key_value:DynamicCache()
output_attentions:False
use_cache:True
cache_position:[num_tokens]
position_embeddings:(cos:[3,1,num_tokens, head_dim], sin:[3,1,num_tokens, head_dim])
```



DecoderLayer

2.4.3.2 Qwen2VLSdpaAttention

在Qwen2VLSdapaAttention中添加了KV Cache。其中生成query和key、value的Linear的维度是不同的, 然后将每一层的key和value等信息更新到past_key_value用于KVCache，例如decoder循环28次，则len(past_key_values.key_cache)=28，past_key_values.key_cache[0].shape=[b, num_key_value_heads, num_tokens, head_dim]。Qwen2VLSdapaAttention输出attention后特征hidden_states和present_key_value(更新后的past_key_value)。

```
# [b, num_head, num_tokens, head_dim]
query_states = query_states.view(bsz, q_len, self.num_heads, self.head_dim).transpose(
# [b, num_key_value_heads, num_tokens, head_dim]
key_states = key_states.view(bsz, q_len, self.num_key_value_heads, self.head_dim).tran
value_states = value_states.view(bsz, q_len, self.num_key_value_heads, self.head_dim).

# 将key_states和value_states更新到past_key_value中, 旋转位置编码和cache_position
cache_kwargs = {"sin": sin, "cos": cos, "cache_position": cache_position}  # Specific
key_states, value_states = past_key_value.update(key_states, value_states, self.layer_
```

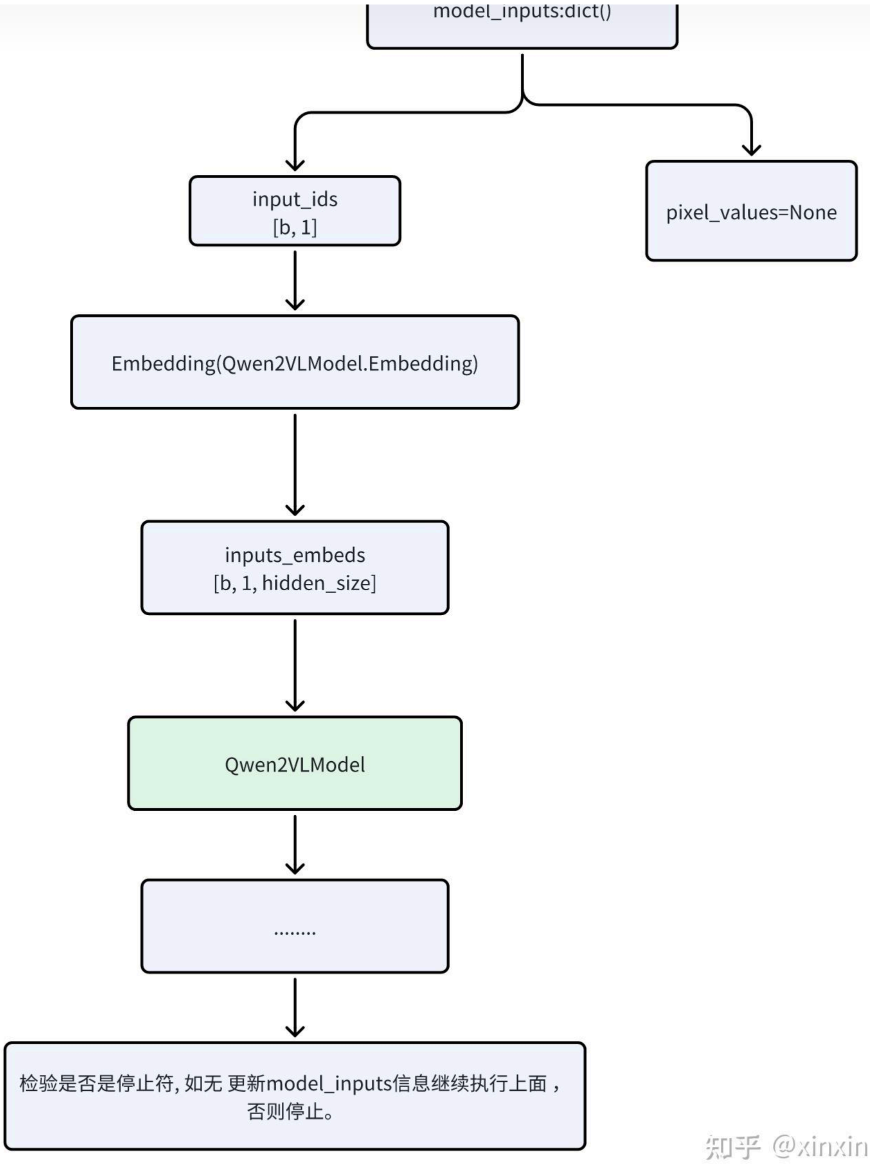
```
value_states = repeat_kv(value_states, self.num_key_value_groups)
```

2.4.3.3 KV Cache的作用

past_key_value用于储存KV Cache所需要的信息。可以发现在数据准备的时候，需要提前把这些信息准备好。在预测第一个token时，input_ids是[1, num_tokens]，而在进行预测第二个甚至更往后的时候使用KV Cache，model_inputs数据如下所示；

```
# 当使用KV Cache时, 预测T+1后更新model_inputs:
model_inputs=
    {   "input_ids": input_ids, # T+1的token的index, shape=[1,1], T表示原始输入的num_tok
        "inputs_embeds":inputs_embeds, #None
        "position_ids": position_ids, # 3D RoPE的位置编码index, shape:[3, 1, 1]
        "past_key_values": past_key_values, # DynamicCache()用于储存KVCache的信息
        "use_cache": use_cache, # 是否采用cache
        "attention_mask": attention_mask, # 推理的attention mask
        "pixel_values": pixel_values, # None
        "pixel_values_videos": pixel_values_videos, # None
        "image_grid_thw": image_grid_thw, # images的patch数量信息
        "video_grid_thw": video_grid_thw, # video的patch数量信息
        "rope_deltas": rope_deltas, # 输入的最后一个token的position_ids与num_tokens的差值
    }
```

此时pixel_values和pixel_values_videos都是None。图像特征就不需要经过ViT，只对新的token进行embedding，此时进行Qwen2VLSdpaAttention进行attention时，num_tokens=1。因此在输入到Qwen2VL前发生了一点点变化。



2.4.4 ViT-2D多维RoPE

1D-RoPE的实现方法：

$$R_{\Theta,m}^{d}x=\begin{pmatrix}x_1\\x_2\\x_3\\x_4\\\vdots\\x_{d-1}\\x_d\end{pmatrix}\otimes\begin{pmatrix}\cos m\theta_1\\\cos m\theta_1\\\cos m\theta_2\\\cos m\theta_2\\\vdots\\\cos m\theta_{d/2}\\\cos m\theta_{d/2}\end{pmatrix}+\begin{pmatrix}-x_2\\x_1\\-x_4\\x_3\\\vdots\\-x_d\\x_{d-1}\end{pmatrix}\otimes\begin{pmatrix}\sin m\theta_1\\\sin m\theta_1\\\sin m\theta_2\\\sin m\theta_2\\\vdots\\\sin m\theta_{d/2}\\\sin m\theta_{d/2}\end{pmatrix}$$

2D-RoPE的实现方法，可以对比发现， θ 的序列编码最大到d/4，两个特征为X方向的编码ids进行旋转，然后再两个特征为Y方向的编码ids进行旋转，依次类推。

$$R^d_{\Theta,ids_x,ids_y}x = \begin{pmatrix} \vdots \\ x_{d/2-1} \\ x_{d/2} \\ x_{d/2+1} \\ x_{d/2+2} \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \vdots \\ \cos ids_x\theta_{d/4} \\ \cos ids_y\theta_{d/4} \\ \cos ids_x\theta_1 \\ \cos ids_y\theta_1 \\ \vdots \\ \cos ids_x\theta_{d/4} \\ \cos ids_y\theta_{d/4} \end{pmatrix} + \begin{pmatrix} \vdots \\ -x_{d/2-1} \\ -x_{d/2} \\ x_1 \\ x_2 \\ \vdots \\ x_{d/2-1} \\ x_{d/2} \end{pmatrix} \otimes \begin{pmatrix} \vdots \\ \sin ids_x\theta_{d/4} \\ \sin ids_y\theta_{d/4} \\ \sin ids_x\theta_1 \\ \sin ids_y\theta_1 \\ \vdots \\ \sin ids_x\theta_{d/4} \\ \sin ids_y\theta_{d/4} \end{pmatrix}$$

代码实现如下：

1. 计算2D旋转位置编码的角度信息 rotary_pos_emb

```
def rot_pos_emb(self, grid_thw):
    '''
    得到每个patch位置的2D-位置编码的正余弦()内的角度信息, 然后再对xy方向进行flatten
    '''
    pos_ids = []
    for t, h, w in grid_thw:
        # [h, w]
        hpos_ids = torch.arange(h).unsqueeze(1).expand(-1, w)
        # [h//spatial_merge_size, spatial_merge_size, w//spatial_merge_size, spatial_m
        hpos_ids = hpos_ids.reshape(
            h // self.spatial_merge_size,
            self.spatial_merge_size,
            w // self.spatial_merge_size,
            self.spatial_merge_size,
        )
        # [h//spatial_merge_size,  w//spatial_merge_size, spatial_merge_size, spatial_
        hpos_ids = hpos_ids.permute(0, 2, 1, 3)
        # [h*w]
        hpos_ids = hpos_ids.flatten()

        wpos_ids = torch.arange(w).unsqueeze(0).expand(h, -1)
        wpos_ids = wpos_ids.reshape(
            h // self.spatial_merge_size,
            self.spatial_merge_size,
            w // self.spatial_merge_size,
            self.spatial_merge_size,
        )
        wpos_ids = wpos_ids.permute(0, 2, 1, 3)
        wpos_ids = wpos_ids.flatten()
        pos_ids.append(torch.stack([hpos_ids, wpos_ids], dim=-1).repeat(t, 1)) # [t, h
    # [n*h*w, 2], 每个patch的(x,y)位置, 比如这个例子就是[2392,2]
    pos_ids = torch.cat(pos_ids, dim=0)
    max_grid_size = grid_thw[:, 1:].max()
    # [max_grid_size, head_dim//4], x和y每个方向最大的就是\theta_{d//4}
    rotary_pos_emb_full = self.rotary_pos_emb(max_grid_size)
    # 提取每个patch的x,y的embedding角度信息m*\theta_{i}, [nhw, 2, head_dim//4] -> [nhw,
    rotary_pos_emb = rotary_pos_emb_full[pos_ids].flatten(1)
    return rotary_pos_emb
  # 其中rotary_pos_emb定义如下:
  class VisionRotaryEmbedding(nn.Module):
    def __init__(self, dim: int, theta: float = 10000.0) -> None:
        super().__init__()
        # 就是 1 / (10000^{2i/d})
        inv_freq = 1.0 / (theta ** (torch.arange(0, dim, 2, dtype=torch.float) / dim))
        self.register_buffer("inv_freq", inv_freq, persistent=False)

    def forward(self, seqlen: int) -> torch.Tensor:
```

```
        # [seqlen, dim//2], 对应每m个对应的位置编码正余弦里面的数 m/(10000^{2i/dim})
        freqs = torch.outer(seq, self.inv_freq) # 外积
        return freqs
```

通过上面我们得到每个patch对应的2D旋转位置编码的角度信息 $ids * \theta$ ，然后看看怎么将这个运用高效计算添加到Q和K中：

```
def apply_rotary_pos_emb_vision(tensor: torch.Tensor, freqs: torch.Tensor) -> torch.Te
    # tensor:query或者key, freqs: 2D旋转位置编码的角度信息
    orig_dtype = tensor.dtype
    tensor = tensor.float() # [b, seq_len, num_head, dim]
    cos = freqs.cos() # [seq_len, dim//2]
    sin = freqs.sin() # [seq_len, dim//2]
    # repeat(1,1,2)中的2 就是 公式中的两个相同的 m\theta_i
    cos = cos.unsqueeze(1).repeat(1, 1, 2).unsqueeze(0).float() # 维度对齐 [b, seq_len,
    sin = sin.unsqueeze(1).repeat(1, 1, 2).unsqueeze(0).float()
    output = (tensor * cos) + (rotate_half(tensor) * sin) # rope 2D高效计算 [b, seq_len
    output = output.to(orig_dtype)
    return output

 # 其中rotate_half是将[x1,..,x_{d/2},x_{d/2+1},..,x_d]变为[-x_{d/2+1},..,-x_d,x1,..,x_{
 def rotate_half(x):
    """Rotates half the hidden dims of the input."""
    x1 = x[..., : x.shape[-1] // 2]
    x2 = x[..., x.shape[-1] // 2 :]
    return torch.cat((-x2, x1), dim=-1)
```

2.4.5 Modal-RoPE

上面是2D-RoPE直接将每个patch的xy的角度信息embed_dim拼接为2*embed_dim，然后再repeat，也就是dim中一半用x_id的位置编码，一半用y_id的位置编码。
而Modal-RoPE是dim中不同区域取不同信息(temporal, height and width)的位置编码。
疑问：这种dim中不同区域取不一致的，能满足RoPE原理中的相对位置关系证明吗?
首先先看一下rotary_embed如何实现M-RoPE。首先计算head_dim//2个 θ_i 角度信息, 然后与position_id进行相乘得到freqs，最后两个相同的freqs进行拼接得到emb, 以第ids位置token的freqs进行举例 $[ids * \theta_1, ids * \theta_2, \ldots, ids * \theta_{d/2}, ids * \theta_1, ids * \theta_2, \ldots, ids * \theta_{d/2}]$

```
class Qwen2VLRotaryEmbedding(nn.Module):
 def forward(self, x, position_ids):
        if "dynamic" in self.rope_type:
            self._dynamic_frequency_update(position_ids, device=x.device)

        # Core RoPE block. In contrast to other models, Qwen2_VL has different positio
        # So we expand the inv_freq to shape (3, ...)
        # 首先计算head_dim//2个$$\theta_i$$角度信息，并expand到[3, 1, 64, 1]，其中head_di
        inv_freq_expanded = self.inv_freq[None, None, :, None].float().expand(3, posit
        position_ids_expanded = position_ids[:, :, None, :].float()  # shape (3, bs, 1
        # Force float32 (see https://github.com/huggingface/transformers/pull/29285)
        device_type = x.device.type
        device_type = device_type if isinstance(device_type, str) and device_type != "
        with torch.autocast(device_type=device_type, enabled=False):
            # $$\theta_i$$与position_ids进行相乘，得到ids * $$\theta_i$$
            freqs = (inv_freq_expanded.float() @ position_ids_expanded.float()).transp
            # 相同的freqs进行拼接
            emb = torch.cat((freqs, freqs), dim=-1) # [3, bs, positions, 128]
            cos = emb.cos()
            sin = emb.sin()

        # Advanced RoPE types (e.g. yarn) apply a post-processing scaling factor, equi
        cos = cos * self.attention_scaling
        sin = sin * self.attention_scaling
```

上面通过freqs得到每个维度的cos和sin信息。那么在多维怎么做RoPE呢？是每个维度切取一部分的位置编码，然后进行拼接得到最终的旋转位置编码，这样就包含了不同维度的位置信息。以head_dim=128举例，公式和code实现如下，例如 x_1 和 x_{65} 特征值进行了旋转。

$$R^d_{\Theta,ids_t,ids_x,ids_y}x = \begin{pmatrix} x_1 \\ \vdots \\ x_{16} \\ x_{17} \\ \vdots \\ x_{40} \\ x_{41} \\ \vdots \\ x_{64} \\ x_{65} \\ \vdots \\ x_{80} \\ x_{81} \\ \vdots \\ x_{104} \\ x_{105} \\ \vdots \\ x_{128} \end{pmatrix} \otimes \begin{pmatrix} \cos ids_t\theta_1 \\ \vdots \\ \cos ids_t\theta_{16} \\ \cos ids_x\theta_1 \\ \vdots \\ \cos ids_x\theta_{24} \\ \cos ids_y\theta_1 \\ \vdots \\ \cos ids_y\theta_{24} \\ \cos ids_t\theta_1 \\ \vdots \\ \cos ids_t\theta_{16} \\ \cos ids_x\theta_1 \\ \vdots \\ \cos ids_x\theta_{24} \\ \cos ids_y\theta_1 \\ \vdots \\ \cos ids_y\theta_{24} \end{pmatrix} + \begin{pmatrix} -x_{65} \\ \vdots \\ -x_{80} \\ -x_{81} \\ \vdots \\ -x_{104} \\ -x_{105} \\ \vdots \\ -x_{128} \\ x_1 \\ \vdots \\ x_{16} \\ x_{17} \\ \vdots \\ x_{40} \\ x_{41} \\ \vdots \\ x_{64} \end{pmatrix} \otimes \begin{pmatrix} \sin ids_t\theta_1 \\ \vdots \\ \sin ids_t\theta_{16} \\ \sin ids_x\theta_1 \\ \vdots \\ \sin ids_x\theta_{24} \\ \sin ids_y\theta_1 \\ \vdots \\ \sin ids_y\theta_{24} \\ \sin ids_t\theta_1 \\ \vdots \\ \sin ids_t\theta_{16} \\ \sin ids_x\theta_1 \\ \vdots \\ \sin ids_x\theta_{24} \\ \sin ids_y\theta_1 \\ \vdots \\ \sin ids_y\theta_{24} \end{pmatrix}$$

```
# position_ids是一个[3, b, num_tokens], 每个token有3个方向temporal, height and width的位
# cos.shape = [3,b,num_tokens, 128]
# sin.shape = [3,b,num_tokens, 128]
# q,k.shape = [b,n_head,num_tokens,dim] dim=128
def apply_multimodal_rotary_pos_emb(q, k, cos, sin, mrope_section, unsqueeze_dim=1):
    # mrope_section = [16, 24, 24, 16, 24, 24]
    mrope_section = mrope_section * 2
    # 先将cos对dim维度划分为6组数据, [3, 1, num_tokens, 16], [3, 1, num_tokens, 24], ...
    # (1) 第1组数据slice[0,...]即(temporal)出来[1,num_tokens,16]。
    # (2) 第2组数据slice[1,...]即(height)出来[1,num_tokens,24]
    # (3) 第3组数据slice[2,...]即(width)出来[1,num_tokens,24]
    # (4) 第4组数据slice[0,...]即(temporal)出来[1,num_tokens,16], 由于freqs(128)前一半(64
    # (5) 第5组数据slice[1,...]即(height)出来[1,num_tokens,24]
    # (6) 第6组数据slice[2,...]即(width)出来[1,num_tokens,24]
    # cat拼接为[1,1,num_tokens,128]
    cos = torch.cat([m[i % 3] for i, m in enumerate(cos.split(mrope_section, dim=-1))]
        unsqueeze_dim
    )
    # sin处理和cos一样的
    sin = torch.cat([m[i % 3] for i, m in enumerate(sin.split(mrope_section, dim=-1))]
        unsqueeze_dim
    )
    # 希望0-15特征元素使用temporal类型的位置编码, 16-39使用height类型的位置编码, ...
    q_embed = (q * cos) + (rotate_half(q) * sin)
    k_embed = (k * cos) + (rotate_half(k) * sin)
    return q_embed, k_embed
```

编辑于 2025-01-16 10:19・IP 属地广东

内容所属专栏

LLaVA

订阅专栏

知乎 首发于 LLaVA

理性发言，友善互动

17 条评论 默认 最新

川川
写的真好，非常详细，谢谢分享！！😘
03-24 · 四川 回复 喜欢

temp
请问这种模型结构，输入文本和图片，如何提取综合的多模态表征?
03-20 · 北京 回复 喜欢

xinxin 作者
文本是embedding，图像是vit
03-22 · 广东 回复 喜欢

樨潮
2d rope的公式x下标打错了吧
03-17 · 北京 回复 喜欢

abc
有一个小问题，文章中提到的3D-rope和2D-rope分别是什么时候使用呢
02-26 · 上海 回复 喜欢

xinxin 作者
2D-rope是在vit中，3D-rope是在llm网络中
02-27 · 广东 回复 喜欢

泡泡肥猫
请问Qwen内部的Qwen2VL model是什么结构啊？如何控制是否冻结ViT呢🥺
02-07 · 广东 回复 喜欢

xinxin 作者
Qwen2VLModel其实就是qwen的LLM模型。冻结vit的话，直接把vit模块参数requires_grad设置为false就好啦
02-07 · 广东 回复 喜欢

Letme
大佬，我想请教一下qwen2vl的每一帧的token怎么计算呀，似乎和我预想的不一样。我把min_pixel和max_pixel都设为336*336，理论上一帧占token数为（336/28）^{2} = 144，然后我输入了一个长视频，并把最小帧数和最大帧数都设为了300，理论上这个视频占用的token数应该为43200，但实测发现大概就21600左右（约1/2），是我哪里理解错了嘛🥺
01-17 · 广东 回复 喜欢

xinxin 作者
输入原始图像的shape是多大的
01-17 · 广东 回复 喜欢

想去海边吹吹风
2.2.1 image_processor 流程图里面的 size 错了吧
01-15 · 上海 回复 喜欢

xinxin 作者
已修改
01-16 · 广东 回复 喜欢

世界上那你
牛逼
01-15 · 中国香港 回复 喜欢

阿阿阿是阿翼啊

2024-12-28 · 中国香港 回复 喜欢

xinxin 作者 ▸ Leiy

是的

2024-12-31 · 广东 回复 喜欢

阿阿阿是阿翼啊 ▸ Leiy

感谢!

2024-12-31 · 中国香港 回复 喜欢

展开其他 1 条回复 ›