# 1.视觉模块

1.写config

维度信息，注意力头信息，eps，处理图像信息（size,token等等）

2.写大类

配置 + 模型类

模型类：嵌入 + 编码 + 归一化（层归一化）

3.写小类

嵌入：nn.embedding转, numpy转tensor，卷积提取特征，加位置编码（相对位置编码）

编码： encoder层， 每一层过多头+add&norm，MLP+ADD_NORM

归一化：层归一化

4.小类组件

位置编码：

```
self.position_embedding = nn.Embedding(self.num_positions, self.embed_dim)
self.register_buffer(
    "position_ids",
    torch.arange(self.num_positions).expand((1, -1)),
    persistent=False,
)
embeddings = embeddings + self.position_embedding(self.position_ids)
```

encoder:

多头：

QKVO

分维度，算QKV，然后过O

MLP：

激活函数：

```
nn.functional.gelu(hidden_states, approximate="tanh")#允许有一定负值
```

# 2.语言模块

1.写大config

大config = 视觉+语言config

2.写大类PaliGemmaForConditionalGeneration

模型+准备输入部分 + tie_weights

准备输入部分：token_embeding + image_imbedding 替换填充，将attention_mask转换为casual_mask，注意Kv_cache

模型：视觉+projector+语言

3.写中类

projector：nn.Linear对齐维度

语言：GemmaForCausalLM

nn.embeding + lauguage_model+lm_head

lm_head和nn.embeding进行tie_weights

语言模型：lauguage_model

标准transformer

多头 + add&norm

MLP + ADD%NORM

norm均为RMS

4.写小类：

多头

q,k,v + 分组查询（local memory比较小，瓶颈不在计算而在这个）

q和k用旋转位置编码

旋转位置编码实现：

```
class GemmaRotaryEmbedding(nn.Module): # NO.25
    """
    For each position to generate the rotational position encoding, understand
the formula of rotational position encoding can understand this part,
    mainly some formulas and dimensional transformation.
    """
    def __init__(self, dim, max_position_embeddings=2048, base=10000,
device=None):
        super().__init__()

        self.dim = dim # it is set to the head_dim
        self.max_position_embeddings = max_position_embeddings
        self.base = base

        # Calculate the theta according to the formula theta_i = base^(2i/dim)
where i = 0, 1, 2, ..., dim // 2
        inv_freq = 1.0 / (self.base ** (torch.arange(0, self.dim, 2,
dtype=torch.int64).float() / self.dim))
        self.register_buffer("inv_freq", tensor=inv_freq, persistent=False) #
Registration parameters, same as the visual model

    @torch.no_grad()
    def forward(self, x, position_ids, seq_len=None):
        # x: [bs, num_attention_heads, seq_len, head_size]
        self.inv_freq.to(x.device)
```

```
        # Copy the inv_freq tensor for batch in the sequence
        # inv_freq_expanded: [Batch_Size, Head_Dim // 2, 1]
        inv_freq_expanded = self.inv_freq[None, :,
None].float().expand(position_ids.shape[0], -1, 1)
        # position_ids_expanded: [Batch_Size, 1, Seq_Len]
        position_ids_expanded = position_ids[:, None, :].float()
        device_type = x.device.type
        device_type = device_type if isinstance(device_type, str) and
device_type != "mps" else "cpu"
        with torch.autocast(device_type=device_type, enabled=False):
            # Multiply each theta by the position (which is the argument of the
sin and cos functions)
            # freqs: [Batch_Size, Head_Dim // 2, 1] @ [Batch_Size, 1, Seq_Len] -
-> [Batch_Size, Seq_Len, Head_Dim // 2]
            freqs = (inv_freq_expanded.float() @
position_ids_expanded.float()).transpose(1, 2)
            # emb: [Batch_Size, Seq_Len, Head_Dim]
            emb = torch.cat((freqs, freqs), dim=-1)
            # cos, sin: [Batch_Size, Seq_Len, Head_Dim]
            cos = emb.cos()
            sin = emb.sin()
        return cos.to(dtype=x.dtype), sin.to(dtype=x.dtype)

def rotate_half(x): # NO.26
    """
    Here huggingface did some processing, but the calculation is still rotational
position encoding, before someone mentioned issue, do not care about this part.
    """
    # Build the [-x2, x1, -x4, x3, ...] tensor for the sin part of the positional
encoding.
    x1 = x[..., : x.shape[-1] // 2] # Takes the first half of the last dimension
    x2 = x[..., x.shape[-1] // 2 :] # Takes the second half of the last
dimension
    return torch.cat((-x2, x1), dim=-1)


def apply_rotary_pos_emb(q, k, cos, sin, unsqueeze_dim=1): # NO.27
    """
    Calculation of the rotational position code
    """
    cos = cos.unsqueeze(unsqueeze_dim) # Add the head dimension
    sin = sin.unsqueeze(unsqueeze_dim) # Add the head dimension
    # Apply the formula (34) of the Rotary Positional Encoding paper.
    q_embed = (q * cos) + (rotate_half(q) * sin)
    k_embed = (k * cos) + (rotate_half(k) * sin)
    return q_embed, k_embed

self.rotary_emb = GemmaRotaryEmbedding( # Used to calculate the rotation angle
of each position
    self.head_dim,
    max_position_embeddings=self.max_position_embeddings,
    base=self.rope_theta,
)
# [Batch_Size, Seq_Len, Head_Dim], [Batch_Size, Seq_Len, Head_Dim]
cos, sin = self.rotary_emb(value_states, position_ids, seq_len=None)
```

```
# [Batch_Size, Num_Heads_Q, Seq_Len, Head_Dim], [Batch_Size, Num_Heads_KV,
Seq_Len, Head_Dim]
query_states, key_states = apply_rotary_pos_emb(query_states, key_states, cos,
sin)
```

k和v在计算之前根据q和Kv_cache的情况来进行kv_cache处理

RMS_NORM

```
class GemmaRMSNorm(nn.Module): # NO.17
    """
    RMSNorm is a special normalisation method, called Root Mean Square
Normalization, which, unlike standard LayerNorm,
    does not involve a mean-subtracting operation on the input dimensions, but
instead normalises based only on the root-mean-square value of the input.
    eps=config.rms_norm_ eps is a small constant used to prevent division-by-zero
errors and ensure numerical stability of the normalisation.
    Compared to LayerNorm, RMSNorm is typically less computationally intensive
and can reduce the dependency on certain input dimensions.
    """
    def __init__(self, dim: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.zeros(dim))

    def _norm(self, x):
        return x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)

    def forward(self, x):
        output = self._norm(x.float())
        output = output * (1.0 + self.weight.float())
        return output.type_as(x)
```

## 3.processor模块

1.写大类

图像处理 + tokenizer

tokenizer加image_token_id

图像处理->pixelvalues：

缩放

转NUMPY

转[0,1]

减去均值除以方差

transpose转维度,channel在前  [Height, Width, Channel] -> [Channel, Height, Width]

tokenizer：

autotokenizer读取

设置模板（添加imagetoken）

过tokenizer出input_ids

做attention_mask

## 4.inference

模型：

初始化模型config，读取Json

根据config初始化模型

safeopen读取模型权重

model.load_statedict(tensors)最终完成模型

准备输入

Image读一下图像，和text一起输入processor

输入和模型转同设备

输入：kv_cache如果有，input_ids,attention_mask，pixel_values

输入之前记得配置好配置文件：

例子：

```
generetion_config = {
  "bos_token_id": lq_tokenizer.tokenizer.bos_token_id,
  "do_sample": True,
  "eos_token_id": [128009],
   pad_token_id =  lq_tokenizer.tokenizer.pad_token_id,
  "repetition_penalty": 1.15,
  "temperature": 1.0,
  "top_p": 0.001,
  "top_k": 5
 }
```