llama 2代码详解

LLM(Large Language Model)应该是今年深度学习领域一项具有革命性的技术突破，因为ChatGPT3.5/4没有开源，所以本文选择Meta AI半开源的LLM 模型 Llama 2，该模型也是Hugging Face open_llm_leaderboard的榜首模型

> 所谓半开源即只有inference过程没有train过程

- paper： https://arxiv.org/abs/2307.09288
- code：https://github.com/facebookresearch/llama
- 笔者逐行注释的code： https://github.com/sunkx109/llama

1.处理流程

首先在了解Llama 2模型结构细节之前，先来看一看大语言模型通常的处理流程：

1.1 常见大模型处理流程

（1）输入数据

LLM的输入数据是一段文本，可以是一个句子或一段话。文本通常被表示成单词或字符的序列。

```
[君不见黄河之水天上来，奔流到海不复回。君不见高堂明镜悲白发，朝如青丝暮成雪。...五花马、千金裘，呼儿将出换美酒，与尔同销万古愁]
```

（2）Tokenization

之后需要将文本进行Tokenization，**将其切分成单词或字符，形成Token序列**。之后再将文本映射成模型可理解的输入形式，将文本序列转换为整数索引序列(这个索引就是单词或字符在语料库中的index)，这个过程通常由一些开源的文本Tokenzier工具，如sentencepiece等来处理

```
序列化->
['BOS','君','不','见','黄','河','之','水','天','上','来',', '
,'奔','流','到'...'与','尔','同','销','万','古','愁','EOS']

假设语料库索引化->
['BOS','10','3','67','89','21','45','55','61','4','324','565'
,'789','6567','786'...'7869','9','3452','563','56','66','77','EOS']
```

（3）Embedding

文本信息经过Tokenization之后变成了token序列，而Embedding则继续**将每个Token映射为一个实数向量**，为Embeding Vector

```
'BOS'-> [p_{00},p_{01},p_{02},...,p_{0d-1}]
'10' -> [p_{10},p_{11},p_{12},...,p_{1d-1}]
'3'  -> [p_{20},p_{21},p_{22},...,p_{2d-1}]
...
'EOS'-> [p_{n0},p_{n1},p_{n2},...,p_{nd-1}]
```

（4）位置编码

对于Token序列中的每个位置，添加位置编码（Positional Encoding）向量，以提供关于Token在序列中位置的信息。位置编码是为了**区分不同位置的Token，并为模型提供上下文关系的信息。**

```
[p_{00},p_{01},p_{02},...,p_{0d-1}]       [pe_{00},pe_{01},pe_{02},...,pe_{0d-1}]
[p_{10},p_{11},p_{12},...,p_{1d-1}]       [pe_{10},pe_{11},pe_{12},...,pe_{1d-1}]
[p_{20},p_{21},p_{22},...,p_{2d-1}]    +  [pe_{20},pe_{21},pe_{22},...,pe_{2d-1}]
...                                       ...
[p_{n0},p_{n1},p_{n2},...,p_{nd-1}]       [pe_{n0},pe_{n1},pe_{n2} ,...,pe_{nd-1}]
```

（5）Transformer

在生成任务中，模型只需要用到Transformer 的decoder阶段，即Decoder-Only，比如GPT、LLaMA 都是。

（6）自回归生成

在生成任务中，使用自回归（Autoregressive）方式，即**逐个生成输出序列中的每个Token**。在解码过程中，每次生成一个Token时，使用前面已生成的内容作为上下文，来帮助预测下一个Token。

```
model = LLaMA2()
def generate(inputs, n_tokens_to_generate):
    for _ in range(n_tokens_to_generate): # auto-regressive decode loop
        output = model(inputs) # model forward pass
        next = np.argmax(output[-1]) # greedy sampling
        inputs.append(next) # append prediction to input
    return inputs[len(inputs) - n_tokens_to_generate :]  # only return generated
tokens

input = [p0, p1,p2]  #对应['BOS','君','不']
output_ids = generate(input, 3) # 假设生成 ['p3','p4','p5']
output_ids = decode(output_ids) # 通过Tokenization解码
output_tokens = [vocab[i] for i in output_ids] # "见" "黄" "河"
```

（7）输出处理

生成的Token序列通过一个输出层，通常是线性变换加上Softmax函数，将每个位置的概率分布转换为对应Token的概率。根据概率，选择概率最高的Token或者作为模型的预测结果。或者其他的的方法生成next token ,比如:

```
def sample_top_p(probs, p):
    #从给定的概率分布中采样一个token，采样的方式是先对概率进行排序，然后计算累积概率，
    #然后选择累积概率小于p的部分，最后在这部分中随机选择一个token。
    probs_sort, probs_idx = torch.sort(probs, dim=-1, descending=True) #给定的概率降序排序
    probs_sum = torch.cumsum(probs_sort, dim=-1) #从第一个元素开始，依次将序列中的每个元素与前面所有元素的和相加得到的
    mask = probs_sum - probs_sort > p
    probs_sort[mask] = 0.0 #将累计和减去当前值>p的地方全部置0,留下来的就是概率较大的
    probs_sort.div_(probs_sort.sum(dim=-1, keepdim=True)) #归一化下
    next_token = torch.multinomial(probs_sort, num_samples=1) # 从归一化之后的样本抽取一个样本
    next_token = torch.gather(probs_idx, -1, next_token) #从原始probs_idx找到next_token所对应的index
    return next_token
```

1. 2 Code

本段代码在`llama/generation.py`中的generate函数，为了便于梳理逻辑笔者这里做了一些裁剪

```
@torch.inference_mode()
def generate(prompt_tokens: List[List[int]], #提示的tokens
    max_gen_len: int, #最大生成长度
    temperature: float = 0.6,
    top_p: float = 0.9,
    logprobs: bool = False,
    echo: bool = False,
) -> Tuple[List[List[int]], Optional[List[List[float]]]]:
    ...
    min_prompt_len = min(len(t) for t in prompt_tokens) # 提示句子中最短的提示长度
    max_prompt_len = max(len(t) for t in prompt_tokens) # 提示句子中最长的提示长度
    ...
    total_len = min(params.max_seq_len, max_gen_len + max_prompt_len) #最终要生成字总长度
    pad_id = self.tokenizer.pad_id #填充字，在tokenizer中定义的填充字
    # 生成一个shape 为(提示token的组数,total_len) 初始字符为pad_id的tokens
    tokens = torch.full((bsz, total_len), pad_id, dtype=torch.long, device="cuda")
    ...# 接着将prompt_tokens填充至tokens
    prev_pos = 0 #初始位置为0
    eos_reached = torch.tensor([False] * bsz, device="cuda") # 用于判断prompt中的每个句子是否已经处理完成
    input_text_mask = tokens != pad_id #mask 标记那些不是填充字的地方
    for cur_pos in range(min_prompt_len, total_len):
        #初始时加载prompt部分进行预测第一个生成的token
        logits = self.model.forward(tokens[:, prev_pos:cur_pos], prev_pos) # 以每个句子中的[prev_pos:cur_pos]部分作为输入去推理
        if logprobs:
            # 如果开启了计算概率，就会把当前输出的序列logits，与原始提示中的序列右移一位之后
            token_logprobs[:, prev_pos + 1 : cur_pos + 1] = -F.cross_entropy(
                input=logits.transpose(1, 2),
                target=tokens[:, prev_pos + 1 : cur_pos + 1], #shape=(bst,cur_pos-prev_pos)
                reduction="none",
                ignore_index=pad_id, #这里需要注意一下，ignore_index参数的作用是忽略target中为pad_id所对应的logits分量
```

```
                                        #也就说当target右移到了pad_id，那么他与logits计算的
loss不对整体loss产生影响，也就是你预测的是啥就是啥
                                        #target也不知道正确答案了
            )
        if temperature > 0:
            probs = torch.softmax(logits[:, -1] / temperature, dim=-1) #带温度系数的
softmax
            next_token = sample_top_p(probs, top_p) #按sample_top_p的方式取next_token
        else:
            next_token = torch.argmax(logits[:, -1], dim=-1) #之间取概率最大的
next_token
        # only replace token if prompt has already been generated
        ...#再将生成的next_token填入cur_pos位置
        tokens[:, cur_pos] = next_token
        prev_pos = cur_pos
        ... #更改eos_reached的值，但所有句子全部生成完毕时退出

#最后按照生成的tokens的顺序返回即可
```

2.模型结构

可以说目前主流的LLM处理模型都是基于Transformer而进行构建的，Llama 2也不例外，而LLM这种生成式的任务是根据给定输入文本序列的上下文信息预测下一个单词或token，所以LLM模型通常只需要使用到Transformer Decoder部分，而所谓Decoder相对于Encoder就是在计算$Q*K$时引入了Mask以确保当前位置只能关注前面已经生成的内容。

Llama 2的模型结构与标准的Transformer Decoder结构基本一致，主要由32个 Transformer Block 组成，不同之处主要包括以下几点：

1. 前置的**RMSNorm**层
2. Q在与K相乘之前，先使用**RoPE**进行位置编码
3. **K V Cache**，并采用**Group Query Attention**
4. FeedForward层

2.1 RMSNorm

Transformer中的Normalization层一般都是采用LayerNorm来对Tensor进行归一化，LayerNorm的公式如下：

LayerNorm:

$$y=\frac{x-E[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}}*\gamma+\beta$$

$$E[x]=\frac{1}{N}\sum_{i=1}^{N}x_i$$

$$\mathrm{Var}[x]=\frac{1}{N}\sum_{i=1}^{N}(x_i-E[x])^2$$

而RMSNorm就是LayerNorm的变体，**RMSNorm省去了求均值的过程，也没有了偏置β**，即

RMSNorm:

$$y=\frac{x}{\sqrt{\mathrm{Mean}(x^2)+\epsilon}}*\gamma$$

$$\text{Mean}(x^2) = \frac{1}{N}\sum_{i=1}^{N} x_i^2$$

其中 γ 和 β 为可学习的参数。

```
# RMSNorm
class RMSNorm(torch.nn.Module):
    def __init__(self, dim: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps # ϵ
        self.weight = nn.Parameter(torch.ones(dim)) #可学习参数γ

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

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

2.2 RoPE

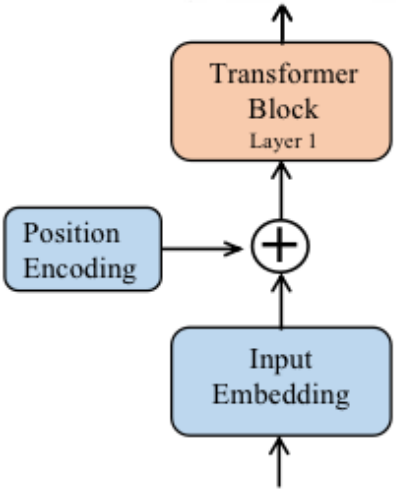
Llama 2 在对序列进行位置编码时，也与标准Transformer不一样，Llama 2的位置编码在每个Attention层中分别对Q K 进行RoPE位置编码，而不是在Transformer Block之前进行一次位置编码，也就是说每次计算Attention时都分别要对Q K做位置编码（llama 2 官方代码中是这么干的）。

一次输入数据经过tokenization之后，会得到一组单词索引序列 $\{w_0, w_1, w_2, \ldots, w_n\}$，之后经过embedding处理后也就变成了 $\{x_0, x_1, x_2, \ldots, x_n\}$，embedding后的序列通过Linear层将输入数据 x_i 转换为对应的 q_i, k_i, v_i，之后便会对 q_i, k_i 两者做RoPE位置编码，之后便计算Attention。

其中 x_i 为第 i 个单词索引序列所对应的 d 维词嵌入向量 $\{x_i^0, x_i^1, x_i^2, \ldots, x_i^{d-1}\}$

（1）绝对位置编码

在标准的Transformer中通常是在整个网络进入Transformer Block之前做一个位置编码，如下图所示



比较经典的位置编码用公式表达就是，其中 $p_{i,2t}$ 表示第 i 个嵌入向量 x_i 的第 2t 个位置的位置编码：

$$f_{\{q,k,v\}}(x_i, i) = W_{\{q,k,v\}}(x_i + p_i)$$

$$p_{i,2t} = \sin\left(\frac{i}{10000^{\frac{2t}{d}}}\right)$$

$$p_{i,2t+1} = \cos\left(\frac{i}{10000^{\frac{2t}{d}}}\right)$$

（2）旋转位置编码

首先，在介绍RoPE时，先抛出一个问题：RoPE解决了一个什么问题？

在位置编码上，使用旋转位置嵌入（Rotary Positional Embeddings，RoPE）代替原有的绝对位置编码。RoPE 借助了复数的思想，出发点是通过绝对位置编码的方式实现相对位置编码。其目标是通过下述运算来给 q, k添加绝对位置信息：

$$\tilde{q}_m = f(q, m), \quad \tilde{k}_n = f(k, n)$$

经过上述操作后，$\tilde{q}_m$ 和 $\tilde{k}_n$ 就带有位置 m 和 n 的绝对位置信息。

最终可以得到二维情况下用复数表示的 RoPE：

$$f(q, m) = R_f(q, m)e^{i\Theta_f(q,m)} = \|q\|e^{i(\Theta(q)+m\theta)} = qe^{im\theta}$$

根据复数乘法的几何意义，上述变换实际上是对应向量旋转，所以位置向量称为“旋转式位置编码”。还可以使用矩阵形式表示：

$$f(q, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} q_0 \\ q_1 \end{pmatrix}$$

根据内积满足线性叠加的性质，任意偶数维的 RoPE，都可以表示为二维情形的拼接，即：

$$f(q, m) = \underbrace{\begin{pmatrix} \cos m\theta_0 & -\sin m\theta_0 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_0 & \cos m\theta_0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_1 & -\sin m\theta_1 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_1 & \cos m\theta_1 & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \ddots & \cdots & \cdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2-1} & -\sin m\theta_{d/2-1} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2-1} & \cos m\theta_{d/2-1} \end{pmatrix}}_{R_d} \begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ \cdots \\ q_{d-2} \\ q_{d-1} \end{pmatrix}$$

（3）RoPE Code

```
def precompute_freqs_cis(dim: int, end: int, theta: float = 10000.0):
    # 计算词向量元素两两分组以后，每组元素对应的旋转角度
    # arange生成[0,2,4...126]
    freqs = 1.0 / (theta ** (torch.arange(0, dim, 2)[: (dim // 2)].float() / dim))
    # t = [0,....end]
    t = torch.arange(end, device=freqs.device)  # type: ignore
    # t为列向量 freqs为行向量做外积
    # freqs.shape = (t.len(),freqs.len()) #shape (end,dim//2)
    freqs = torch.outer(t, freqs).float()  # type: ignore
    # 生成复数
    # torch.polar(abs,angle) -> abs*cos(angle) + abs*sin(angle)*j
    freqs_cis = torch.polar(torch.ones_like(freqs), freqs)  # complex64
    # freqs_cis.shape  = (end,dim//2)
    return freqs_cis

def reshape_for_broadcast(freqs_cis: torch.Tensor, x: torch.Tensor):
```

```
    # ndim为x的维度数 ,此时应该为4
    ndim = x.ndim
    assert 0 <= 1 < ndim
    assert freqs_cis.shape == (x.shape[1], x.shape[-1])
    shape = [d if i == 1 or i == ndim - 1 else 1 for i, d in enumerate(x.shape)]
    # (1,x.shape[1],1,x.shape[-1])
    return freqs_cis.view(*shape)

def apply_rotary_emb(
    xq: torch.Tensor,
    xk: torch.Tensor,
    freqs_cis: torch.Tensor,
) -> Tuple[torch.Tensor, torch.Tensor]:
    # xq.shape = [bsz, seqlen, self.n_local_heads, self.head_dim]
    # xq_.shape = [bsz, seqlen, self.n_local_heads, self.head_dim//2 , 2]
    # torch.view_as_complex用于将二维向量转换为复数域 torch.view_as_complex即([x,y]) ->
(x+yj)
    # 所以经过view_as_complex变换后xq_.shape = [bsz, seqlen, self.n_local_heads,
self.head_dim//2]
    xq_ = torch.view_as_complex(xq.float().reshape(*xq.shape[:-1], -1, 2))
    xk_ = torch.view_as_complex(xk.float().reshape(*xk.shape[:-1], -1, 2))


    freqs_cis = reshape_for_broadcast(freqs_cis, xq_) # freqs_cis.shape =
(1,x.shape[1],1,x.shape[-1])

    # xq_ 与freqs_cis广播哈达玛积
    # [bsz, seqlen, self.n_local_heads, self.head_dim//2] *
[1,seqlen,1,self.head_dim//2]
    # torch.view_as_real用于将复数再转换回实数向量, 再经过flatten展平第4个维度
    # [bsz, seqlen, self.n_local_heads, self.head_dim//2] ->[bsz, seqlen,
self.n_local_heads, self.head_dim//2,2 ] ->[bsz, seqlen, self.n_local_heads,
self.head_dim]
    xq_out = torch.view_as_real(xq_ * freqs_cis).flatten(3)
    xk_out = torch.view_as_real(xk_ * freqs_cis).flatten(3)
    return xq_out.type_as(xq), xk_out.type_as(xk)
# 精简版Attention
class Attention(nn.Module):
    def __init__(self, args: ModelArgs):
        super().__init__()
        self.wq = Linear(...)
        self.wk = Linear(...)
        self.wv = Linear(...)

        self.freqs_cis = precompute_freqs_cis(dim, max_seq_len * 2)

    def forward(self, x: torch.Tensor):
        bsz, seqlen, _ = x.shape
        xq, xk, xv = self.wq(x), self.wk(x), self.wv(x)
        xq = xq.view(bsz, seqlen, self.n_local_heads, self.head_dim)
        xk = xk.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)
        xv = xv.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)
         # attention 操作之前，应用旋转位置编码
        xq, xk = apply_rotary_emb(xq, xk, freqs_cis=freqs_cis)
        #...
        # 进行后续Attention计算
        scores = torch.matmul(xq, xk.transpose(1, 2)) / math.sqrt(dim)
```

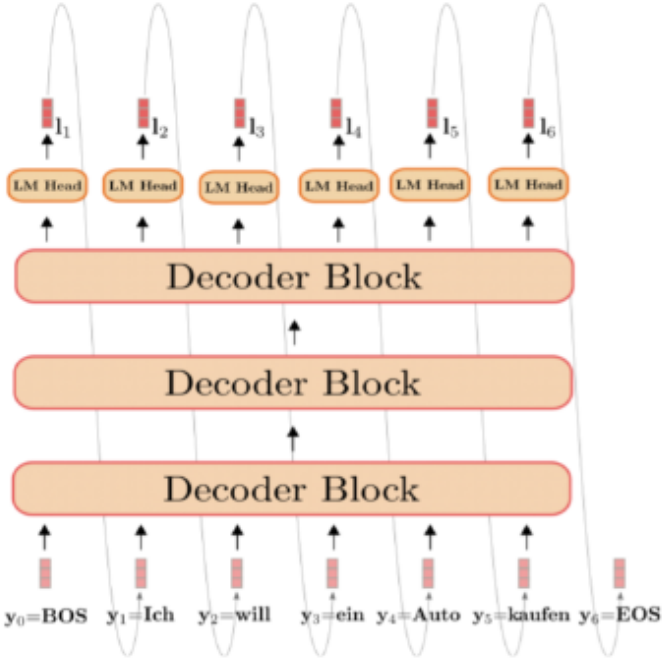
```
        scores = F.softmax(scores.float(), dim=-1)
        output = torch.matmul(scores, xv)  # (batch_size, seq_len, dim)
# ......
```

2.3.KV Cache & GQA

（1）KV Cache

大模型推理性能优化的一个常用技术是KV Cache，那么什么是K V Cache呢？首先这里的K V 值得分别是Attention计算时的KV，而非哈希存储引擎中的Key和Value，这里的Cache也不是那个会发生Cache Missing的Cache，这里的K V Cache就是将Attention 中的KV缓存下来，通过空间换时间的方式来加速计算Attention。

从第一节处理流程中可以知道，在**LLama 2模型的推理阶段是采用自回归的方式来进行推理，即每一个Token的生成都是由之前所有生成的所有token作为输入而得到的。**

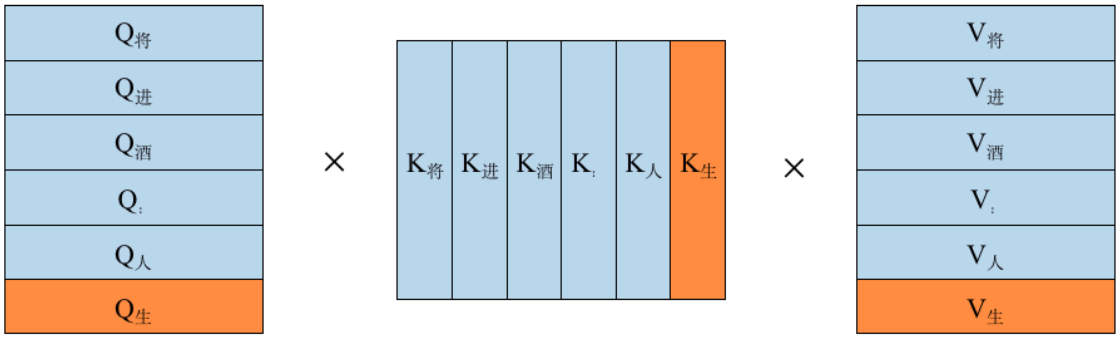


举个例子，假设有这样一个生成任务:

```
In  [1]: {prompt:"将进酒：}
Out [1]: 将进酒：人

In  [2]: 将进酒：人
Out [2]: 将进酒：人生

In  [3]: 将进酒：人生
Out [3]: 将进酒：人生得

In  [4]: 将进酒：人生得
Out [4]: 将进酒：人生得意

In  [5]: 将进酒：人生得意
Out [5]: 将进酒：人生得意需


In  [6]: 将进酒：人生得意需
Out [6]: 将进酒：人生得意需尽
```

```
In  [7]: 将进酒：人生得意需尽
Out [7]: 将进酒：人生得意需尽欢
```

而第三次的处理过程是用"将进酒：人生" 来预测下一个"得"字，所以需要把 **"将进酒：人生"** 进行token化后再进行Attention计算，即$Softmax(Q * K^T) * V$,如下图所示：



不难发现在第二次处理的时候，就已经把 **"将进酒：人"** 所对应的Q,K,V进行过相关的运算，所以没必要在对他们进行Attention计算，这样就能节省大部分算力，由此K V Cache便是来解决这个问题的：**通过将每次计算的K和V缓存下来，之后新的序列进来时只需要从KV Cache中读取之前的KV值即可，就不需要再去重复计算之前的KV了。**此外，对于Q也不用将序列对应的所有 Q_i都计算出来，只需要计算最新的 $Q_{newtoken}$, (即此时句子长度为1), K V同理，所以用简易代码描述一下这个过程就是

```
def mha(x, c_attn, c_proj, n_head, kvcache=None):  # [n_seq, n_embd] -> [n_seq, n_embd]
    # qkv projection
    # when we pass kvcache, n_seq = 1. so we will compute new_q, new_k and new_v
    x = linear(x, **c_attn)  # [n_seq, n_embd] -> [n_seq, 3*n_embd]
    # split into qkv
    qkv = np.split(x, 3, axis=-1)  # [n_seq, 3*n_embd] -> [3, n_seq, n_embd]
    if kvcache:
        # qkv
        new_q, new_k, new_v = qkv  # new_q, new_k, new_v = [1, n_embd]
        old_k, old_v = kvcache
        k = np.vstack([old_k, new_k]) # k = [n_seq, n_embd], where n_seq = prev_n_seq + 1
        v = np.vstack([old_v, new_v]) # v = [n_seq, n_embd], where n_seq = prev_n_seq + 1
        qkv = [new_q, k, v]
```

> **Note**
>
> 至于为什么不用缓存Q?
> **因为是单向注意力机制，Q 只用于和过去的 K/V 做 attention，而过去的 Q 不会被再次用到，所以无需缓存 Q。**

另外，利用KV Cache技术能节省多少计算量呢？有兴趣可以看看分析transformer模型的参数量、计算量、中间激活、KV cache

(2) MQA & GQA

在自回归语言模型中，为了加速推理过程，通常会对 Key（K）和 Value（V）进行缓存，以便在生成下一个 token 时无需重复计算。然而，这种缓存机制会带来显著的内存开销，尤其是在处理长序列或多序列任务时。

以 LLaMA-7B 模型为例，其隐藏层维度（hidden_size）为 4096。每个 Transformer Block 中包含一个 Key 和一个 Value 向量，每个向量长度为 4096，使用 float16 半精度浮点数表示。
因此，单个 Block 的 K/V 缓存大小为：4096×2（K 和 V）$\times 2$（bytes per float16）$= 16\text{KB}$。
LLaMA-2 共有 32 个 Transformer Block，因此单序列推理所需 K/V 缓存总量为：$16\text{KB} \times 32 = 512\text{KB}$

当处理长度为 1024 的句子时，若同时处理多个序列，则总缓存需求将线性增长。例如，处理 1000 条序列时，所需缓存空间将达到约 512MB。

然而，当前主流 GPU 的高速缓存容量有限。例如，NVIDIA H100 的 SRAM 容量约为 50MB，A100 则为 40MB。显然，KV Cache 的存储需求远远超出了片上高速缓存的承载能力。

若将 KV Cache 存储于 DRAM（即 GPU 显存）中，虽然技术上可行，但会显著影响推理性能。由于全局内存（DRAM）的访问速度远低于寄存器或共享内存，处理器的计算单元（ALU）往往需要等待数据搬运完成，从而造成“内存墙”现象。**所谓内存墙，是指处理器的计算速度远高于内存的数据读写速度，导致计算资源无法充分利用，形成性能瓶颈。**

解决该问题可以从硬件和软件两个层面入手：

- **硬件层面**：
  - 使用高带宽内存（HBM）提升内存访问速度；
  - 探索非冯·诺依曼架构，如存内计算（Processing-in-Memory），通过将计算单元与存储单元融合来减少数据搬移开销，例如忆阻器（Memristor）等新型器件。
- **软件层面**：
  - 优化注意力机制，减少对 KV Cache 的依赖，从而降低内存占用并提升推理效率。Llama 2 所采用的 Grouped Query Attention（GQA）正是此类优化方法之一。

为了简单明了说明 MQA GQA 这里用 GQA 原论文的一个图来表示：

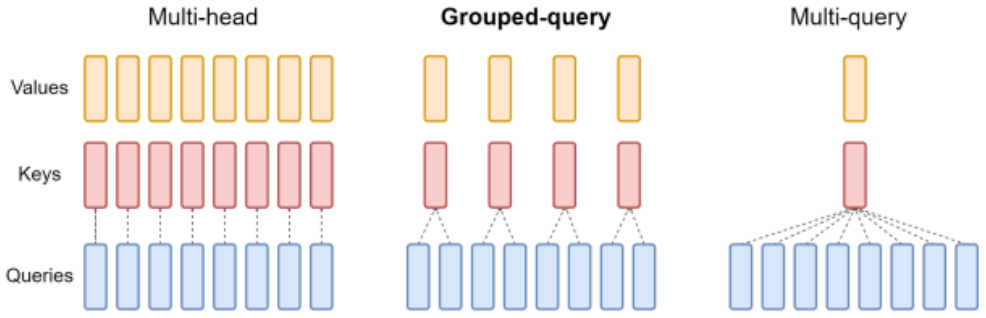


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.

就如图例所言，多头注意力机制（MHA）就是多个头各自拥有自己的 Q, K, V 来算各自的 Self-Attention，而 MQA（Multi Query Attention）就是 Q 依然保持多头，但是 K, V 只有一个，所有多头的 Q 共享一个 K, V，这样做虽然能最大程度减少 KV Cache 所需的缓存空间，但是可想而知参数的减少意味着精度的下降，所以为了在精度和计算之间做一个 trade-off，GQA（Group Query Attention）孕育而生，即 Q 依然是多头，但是分组共享 K, V，即减少了 K, V 缓存所需的缓存空间，也暴露了大部分参数不至于精度损失严重。

(3) Code

这一部分最后结合Llama 2的代码来看看他们的具体实现(为了篇幅做了一些简化)

```
def repeat_kv(x: torch.Tensor, n_rep: int) -> torch.Tensor:
    """torch.repeat_interleave(x, dim=2, repeats=n_rep)"""
    bs, slen, n_kv_heads, head_dim = x.shape
    # 根据n_rep，拓展KV
    if n_rep == 1:
        return x
    return (x[:, :, :, None, :].expand(bs, slen, n_kv_heads, n_rep,
head_dim).reshape(bs, slen, n_kv_heads * n_rep, head_dim))
class Attention(nn.Module):
    def __init__(self, args: ModelArgs):
        super().__init__()
        ...
        self.n_local_heads = args.n_heads // model_parallel_size #Q的头数
        self.n_local_kv_heads = self.n_kv_heads // model_parallel_size  #KV的头数
        self.n_rep = self.n_local_heads // self.n_local_kv_heads
        ...
        self.wq = ColumnParallelLinear(args.dim,args.n_heads * self.head_dim, # Q的
头数* head_dim
                                       ...)
        self.wk = ColumnParallelLinear(args.dim,self.n_kv_heads * self.head_dim, #
K的头数* head_dim
                                       ...)
        self.wv = ColumnParallelLinear(args.dim,self.n_kv_heads * self.head_dim,# V
的头数* head_dim
                                       ...)
        self.wo = RowParallelLinear(args.n_heads * self.head_dim,args.dim,... )

        self.cache_k =
torch.zeros((args.max_batch_size,args.max_seq_len,self.n_local_kv_heads, #KV的头数
                self.head_dim,)).cuda()
        self.cache_v =
torch.zeros((args.max_batch_size,args.max_seq_len,self.n_local_kv_heads,#KV的头数

                                       self.head_dim,)).cuda()
    def forward(
        self,
        x: torch.Tensor,
        start_pos: int,
        freqs_cis: torch.Tensor,
        mask: Optional[torch.Tensor],
    ):
        bsz, seqlen, _ = x.shape
        xq, xk, xv = self.wq(x), self.wk(x), self.wv(x)

        xq = xq.view(bsz, seqlen, self.n_local_heads, self.head_dim)
        xk = xk.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)
        xv = xv.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)

        xq, xk = apply_rotary_emb(xq, xk, freqs_cis=freqs_cis) #嵌入RoPE位置编码
        ...
        # 按此时序列的句子长度把kv添加到cache中
        # 初始在prompt阶段seqlen>=1, 后续生成过程中seqlen==1
```

```
        self.cache_k[:bsz, start_pos : start_pos + seqlen] = xk
        self.cache_v[:bsz, start_pos : start_pos + seqlen] = xv
        # 读取新进来的token所计算得到的k和v
        keys = self.cache_k[:bsz, : start_pos + seqlen]
        values = self.cache_v[:bsz, : start_pos + seqlen]

        # repeat k/v heads if n_kv_heads < n_heads
        keys = repeat_kv(keys, self.n_rep)  # (bs, seqlen, n_local_heads, head_dim)
        values = repeat_kv(values, self.n_rep)  # (bs, seqlen, n_local_heads,
head_dim)

        xq = xq.transpose(1, 2)  # (bs, n_local_heads, seqlen, head_dim)
        keys = keys.transpose(1, 2)
        values = values.transpose(1, 2)
        #计算q*k
        scores = torch.matmul(xq, keys.transpose(2, 3)) / math.sqrt(self.head_dim)
        if mask is not None:
            #加入mask，使得前面的token在于后面的token计算attention时得分为0，mask掉
            scores = scores + mask  # (bs, n_local_heads, seqlen, cache_len +
seqlen)
        scores = F.softmax(scores.float(), dim=-1).type_as(xq)
        output = torch.matmul(scores, values)  # (bs, n_local_heads, seqlen,
head_dim)
        output = output.transpose(1, 2).contiguous().view(bsz, seqlen, -1)
        return self.wo(output)
```

2.4 FeedForward 层

在 LLaMA 2 模型中，FeedForward 层（FFN）紧接在 Attention 层之后，用于对注意力机制输出的信息进行非线性变换。与标准 Transformer 的 FFN 层相比，LLaMA 2 的实现有一些细微但重要的改动。

标准的 Transformer 使用 ReLU 或 GeLU 作为激活函数，其结构通常为两层线性变换加激活：

$$\text{FFN}(x) = W_2 \cdot \text{ReLU}(W_1 \cdot x + b_1) + b_2$$

而 LLaMA 2 中采用了 SwiGLU 结构，这是一种引入门控机制的改进形式，公式如下：

$$\text{FFN}(x) = W_2 \cdot (\text{SiLU}(W_1 \cdot x) \otimes (W_3 \cdot x))$$

其中 W_1, W_2, W_3 是可学习参数，$\otimes$ 表示逐元素相乘，而 SiLU 是该结构的核心激活函数。
SiLU（Sigmoid Linear Unit），也被称为 Swish，定义如下：

$$\text{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}}$$

它结合了输入值和 Sigmoid 函数的输出，具有平滑、非单调的特点。相比传统的 ReLU，SiLU 在负值区域也有响应，能够缓解神经元死亡问题，并有助于梯度传播。

LLaMA 系列模型借鉴了 GLU 类激活函数的设计思想，通过 SwiGLU 引入门控机制来控制信息流。这种结构相比传统 FFN 更具灵活性，能更好地建模复杂函数关系，提升模型表达能力。

总结来看，LLaMA 2 的 FeedForward 层主要特点包括：

- 使用 SiLU 激活函数替代 ReLU / GeLU；
- 使用 SwiGLU 形式，引入门控机制；
- 提升了模型的非线性表达能力和训练稳定性。

```
class FeedForward(nn.Module):
```

```
    def __init__(
        self,
        dim: int,
        hidden_dim: int,
        multiple_of: int,
        ffn_dim_multiplier: Optional[float],
    ):
        super().__init__()
        hidden_dim = int(2 * hidden_dim / 3)
        # custom dim factor multiplier
        if ffn_dim_multiplier is not None:
            hidden_dim = int(ffn_dim_multiplier * hidden_dim)
        hidden_dim = multiple_of * ((hidden_dim + multiple_of - 1) // multiple_of)
        # Linear 1
        self.w1 = ColumnParallelLinear(...)
        # Linear 2
        self.w2 = RowParallelLinear(...)
        # Linear 3
        self.w3 = ColumnParallelLinear(...)
    def forward(self, x):
        return self.w2(F.silu(self.w1(x)) * self.w3(x))
```

参考资料

[1] 一文看懂 LLaMA 中的旋转式位置编码

[2] Transformer升级之路：2、博采众长的旋转式位置编码

[3] 大模型推理性能优化之KV Cache解读

[4] 分析transformer模型的参数量、计算量、中间激活、KV cache

[5] 为什么现在大家都在用 MQA 和 GQA?