

轻量级大语言模型MiniMind源码解读（五）：魔改的注意力机制，细数当代LLM的效率优化手段

Original 凡希 南极Python 2025年10月27日 02:00

前文回顾：

轻量级大语言模型MiniMind源码解读（一）：如何从头训练tokenizer？

轻量级大语言模型MiniMind源码解读（二）：为什么RMSNorm更适合大模型推理？

轻量级大语言模型MiniMind源码解读（三）：原始Transformer的位置编码及其缺陷

轻量级大语言模型MiniMind源码解读（四）：旋转位置编码原理与应用全解析

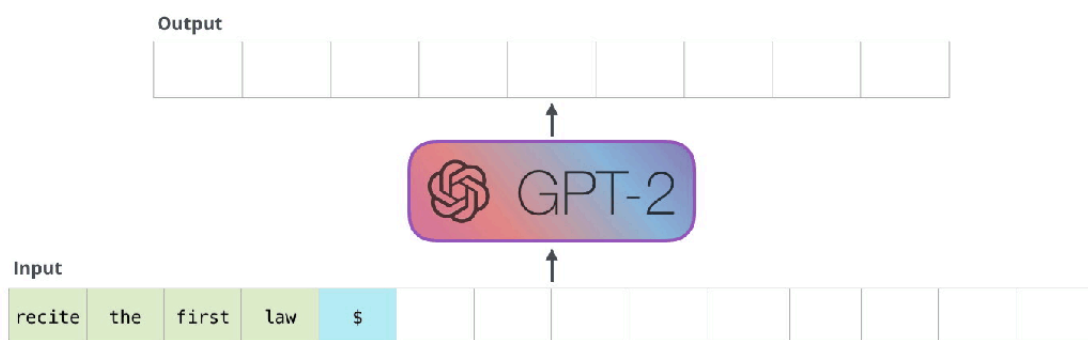
MiniMind中的注意力机制包含了KV cache，GQA，Flash Attention等技术。

一、KV Cache

KV Cache（Key-Value Cache）是在Transformer自回归模型（如GPT）推理阶段中，为了加速推理、减少重复计算而引入的缓存机制。

具体来说，在推理时，为每个生成的token计算K和V矩阵，KV Cache将这些矩阵存储在内存中，以便在生成后续token时，我们只需为新的token计算K和V，而不是重新计算所有当前已生成token的K和V。

解码器以自回归的方式工作，如下面的GPT-2文本生成示例所示：



在解码器的自回归生成中，给定一个输入，模型预测下一个token，然后在下一步将组合输入进行下一次预测。

这种自回归行为重复了一些操作，可以通过放大解码器中计算的掩码缩放点积注意力计算来更好地理解这一点：

这里对是否使用KV Cache的QK计算过程进行对比：

上图中，紫色是从缓存中获取的，绿色是计算得到的，灰色是根据causal机制（当前token只能看到自己以及之前的信息）被mask掉的（因此无需计算）。通过这些动图，可以很清晰的观察到使用KV Cache可以减少许多tokens的K和V向量的重复计算。

举个例子，假设在执行推理时，`batch_size=1`，当前已经累计生成的token数量为`seq_len=511`，`hidden_size=512`，`num_query_heads = 8`，`num_key_value_heads = 2`，

则`head_dim = hidden_size / num_query_heads = 64`，已经缓存的KV的shape均为`[batch_size, seq_len, num_key_value_heads, head_dim] = [1, 511, 2, 64]`，并且将刚预测的新的token与已经预测的511个拼接在一起，总共512个，根据KV Cache的原理，在预测下一个token时，只需要计算新加入的第512个token（记作x）与所有512个token之间的注意力。

x 经过embedding层得到的shape为 `[batch_size, seq_len, hidden_size] = [1, 1, 512]`，然后通过线性层投影出xk和xv，shape都是

[batch_size,seq_len,num_key_value_heads,head_dim]=[1,1,264], 然后reshape成[batch_size,seq_len,num_key_value_heads,head_dim]=[1,1,2,64], 这就是最新生成的tokens的q和v, 将其与缓存的前面512个tokens的q和v进行拼接, 就得到了本次预测需要的KV矩阵, shape为concat([1,511,2,64],[1,1,2,64]) --> [1,512,2,64].

另外, 这里快速介绍一下Transformer中的qkv, 以加深理解:

在Transformer的翻译任务中, encoder接收[bsz,src_seq_len,hidden_size]的token embedding, 经过attention等层输出[bsz,src_seq_len,hidden_size]作为kv(只生成一次), 供q多轮查询, q来自decoder, shape为[bsz,total_seq_len,hidden_size], total_seq_len是decoder当前处理(已经生成)的序列长度。

二、MHA, MQA和GQA

在Transformer中, 每一层的注意力通常是基于多头注意力(Multi-Head Attention, MHA)实现的。这个模块的核心操作是将模型的总隐藏维度 d_{model} (MiniMind中也称之为hidden_dim) 拆分为多个“头”(heads), 每个头独立地进行注意力计算, 然后再将它们的结果拼接起来, 投影回原始空间。

通过"多头"操作, 每个头(head)可以专注不同的子空间, 模型可以更好的捕捉多种不同的语义/结构关系。多个头并行, 相当于多个子空间并行抽取特征, 最后拼接后再映射, 增强了模型的非线性表达能力。

既然"多头"操作有这些优势, 那为什么还会出现GQA、MQA这种注意力机制呢?

实际上, MHA的并行多头机制虽然表达能力强, 但它存在计算和内存上的开销问题。具体表现为:

- 在标准MHA中, 每个头都需要独立的Q、K、V, 共3个线性层, 存储开销是成倍的。
- 在推理时, 尤其是大模型部署中, 每个token都要执行所有头的K/V计算并缓存, 非常耗显存。

为了减少存储占用, 出现了MQA, 其核心思想是: 每个注意力头依然保留独立的Query (Q) 向量, 但所有注意力头共享一组Key (K) 和Value (V) 向量。这样大大减少了计算和缓存中K/V的冗余部分。尤其在解码器结构中(如 GPT 系列), 推理阶段需要缓存每个token的key/value, 如果每个头都独立缓存, 会造成巨大的显存压力, MQA 可以显著降低这种开销。

然而, 由于K和V只有1个头, 所有头的Q都共享这1个K和V, 导致每个Query头无法获取独立的上下文信息, 只能从同一组Key/Value中抽取信息。这在一定程度上限制了模型捕捉多样化注意力模式的能力, 尤其是在建模复杂依赖关系或多语义对齐时可能效果不如标准MHA。

因此, 虽然MQA在计算和显存占用方面具有显著优势, 但也带来了表达能力的折损。这正是GQA出现的原因: 通过让多个Q头共享部分(而不是全部)K/V子空间,

GQA在保持效率的同时，部分恢复了MHA的灵活性和多样性，是一种性能与效率的折中设计。

这张图清晰的展示了MHA，MQA和GQA的区别：

注意，在计算注意力时，虽然MQA/GQA头数少，但通常会通过repeat/broadcast扩展回与Q相同的头数来做注意力计算，因此：从注意力矩阵计算（即 $Q @ K^T$ 和 $attention @ V$ ）的FLOPs来上看，三者几乎是一样的。

但计算量不等于性能损耗，区别主要在：

- KV的投影次数（线性层）少了：MHA每头独立，KV线性投影各做num_heads次,MQA只做一次，GQA做M次（M是一个大于1 & 小于num_heads，且能够被num_heads整除的数）。这样一来，KV cache大幅减少，因为缓存的head 数是1或M，不是num_heads。
- KV的repeat操作可以高效实现：比如FlashAttention可以使用broadcast或index select避免真的物理复制(repeat)。

现在来举个例子，对比三者的存储开销，这里我们只比较注意力中K/V缓存的显存占用（推理时需保留），忽略 Q。

假设总隐藏维度 $d_{model}=4096$ ，num_heads=32，seq_len=1024，batch_size=1，有head_dim= $d_{model} // \text{num_heads}=4096 // 32=128$ ，在GQA中将总共num_heads=32个Q向量分成8组。

每个float32占用4字节，每种方法的总存储占用（Byte）= $\text{batch_size} \times \text{num_kv_heads} \times \text{seq_len} \times \text{head_dim} \times \text{float_size}$

类型	num_kv_heads	计算公式	显存占用 (Byte)	显存占用 (MB)
MHA	32	$1 \times 32 \times 1024 \times 128 \times 4 = 16,777,216$	16,777,216	16 MB
GQA (8组)	8	$1 \times 8 \times 1024 \times 128 \times 4 = 4,194,304$	4,194,304	4 MB
MQA	1	$1 \times 1 \times 1024 \times 128 \times 4 = 524,288$	524,288	0.5 MB

实际部署中，节省的是decoder每个token的KV缓存量，对于长序列推理影响显著。

如果真的希望减少计算量（而不仅仅是缓存量），就不能repeat kv，而是要修改attention的实现方式，让多个Q共享一个K/V。

三、Flash Attention

Flash Attention是一种显存优化 + 更快计算的注意力实现，数学本质不变，只是在实现细节上做了大量工程优化，尤其适用于长序列Transformer模型（如LLM）。

它通过采用流式分块计算，在softmax前后都不存中间结果，避免显存瓶颈。提供更高吞吐、更长上下文能力而不会爆显存。

在PyTorch中，可以简单的通过 `torch.nn.functional.scaled_dot_product_attention` 直接使用Flash Attention。

以上介绍的KV cache，GQA，Flash Attention均已在MiniMind的Attention模块进行了实现，如下：

```
from typing import Optional, Tuple, List, Union
import torch.nn.functional as F

def repeat_kv(x: torch.Tensor, n_rep: int) -> torch.Tensor:
    """torch.repeat_interleave(x, dim=2, repeats=n_rep)"""
    bs, slen, num_key_value_heads, head_dim = x.shape
    if n_rep == 1:
        return x
    return (
        x[:, :, :, None, :]
        .expand(bs, slen, num_key_value_heads, n_rep, head_dim)
        .reshape(bs, slen, num_key_value_heads * n_rep, head_dim)
    )

class Attention(nn.Module):
    def __init__(self, args):
        super().__init__()
        # 支持多query-head共享同一个key/value-head
        self.num_key_value_heads = args.num_attention_heads if args.num_key_value_heads is None else args.num_key_value_heads
        assert args.num_attention_heads % self.num_key_value_heads == 0

        self.n_local_heads = args.num_attention_heads # 总的 attention head 数
        self.n_local_kv_heads = self.num_key_value_heads # k/v head 数
        self.n_rep = self.n_local_heads // self.n_local_kv_heads # 每个 k/v head 被多少个 query-head 共享

        self.head_dim = args.hidden_size // args.num_attention_heads # 每个 head 的维度

        # QKV线性映射层，不使用 bias
        self.q_proj = nn.Linear(args.hidden_size, args.num_attention_heads * self.head_dim)
        self.k_proj = nn.Linear(args.hidden_size, self.num_key_value_heads * self.head_dim)
        self.v_proj = nn.Linear(args.hidden_size, self.num_key_value_heads * self.head_dim)
```

```

self.o_proj = nn.Linear(args.num_attention_heads * self.head_dim, args.hidden_size)

# dropout
self.attn_dropout = nn.Dropout(args.dropout)
self.resid_dropout = nn.Dropout(args.dropout)
self.dropout = args.dropout

# 是否启用 Flash Attention (PyTorch >= 2.0)
self.flash = hasattr(torch.nn.functional, 'scaled_dot_product_attention') and arg

def forward(self,
            x: torch.Tensor, # 输入特征: [bsz, seq_len, hidden_size]
            position_embeddings: Tuple[torch.Tensor, torch.Tensor], # rotary pos emb
            past_key_value: Optional[Tuple[torch.Tensor, torch.Tensor]] = None, # kv
            use_cache=False, # 是否返回 kv_cache
            attention_mask: Optional[torch.Tensor] = None): # attention 掩码 (padding的

    bsz, seq_len, _ = x.shape

    # 线性变换 -> Q, K, V
    xq = self.q_proj(x) # [bsz, seq_len, n_heads * head_dim]
    xk = self.k_proj(x) # [bsz, seq_len, n_kv_heads * head_dim]
    xv = self.v_proj(x) # [bsz, seq_len, n_kv_heads * head_dim]

    # reshape 为多头格式
    xq = xq.view(bsz, seq_len, self.n_local_heads, self.head_dim)
    xk = xk.view(bsz, seq_len, self.n_local_kv_heads, self.head_dim)
    xv = xv.view(bsz, seq_len, self.n_local_kv_heads, self.head_dim)

    # 应用 rotary position embedding
    cos, sin = position_embeddings
    # cos和sin是预计算的很长的一段, 这里只截取前seq_len, 因为最大pos就是seq_len-1
    xq, xk = apply_rotary_pos_emb(xq, xk, cos[:seq_len], sin[:seq_len])

    # 处理 KV 缓存: 将历史的 key/value 与当前拼接(训练时不需要)
    if past_key_value is not None:
        xk = torch.cat([past_key_value[0], xk], dim=1) # time 维度拼接
        xv = torch.cat([past_key_value[1], xv], dim=1)

    past_kv = (xk, xv) if use_cache else None # xk/xv: [bsz, seq_len, self.n_local_kv_h

    # KV head 重复扩展 -> 让所有 Q head 对应到正确的 KV head
    xq = xq.transpose(1, 2) # [bsz, n_heads, seq_len, head_dim]
    xk = repeat_kv(xk, self.n_rep).transpose(1, 2) # [bsz, n_heads, seq_len, head_dim]
    xv = repeat_kv(xv, self.n_rep).transpose(1, 2) # [bsz, n_heads, seq_len, head_dim]

    # Flash Attention 方法 (更快更省内存)
    if self.flash and seq_len != 1:
        dropout_p = self.dropout if self.training else 0.0
        attn_mask = None

```

```

if attention_mask is not None:
    attn_mask = attention_mask.view(bsz, 1, 1, -1).expand(bsz, self.n_local_h
    attn_mask = attn_mask.bool()

# 使用 PyTorch 原生 flash attention (内置 causal)
output = F.scaled_dot_product_attention(
    xq, xk, xv,
    attn_mask=attn_mask,
    dropout_p=dropout_p,
    is_causal=True
)

# 普通 attention 方法
else:
    # 注意力分数计算
    scores = (xq @ xk.transpose(-2, -1)) / math.sqrt(self.head_dim) # [bsz, n_he

    # 添加上三角 mask, 实现 causal attention
    scores = scores + torch.triu(
        torch.full((seq_len, seq_len), float("-inf")), device=scores.device),
        diagonal=1
    ).unsqueeze(0).unsqueeze(0) # scores+mask, 上三角变成-inf, 经过softmax后趋于0, 从

    # attention 掩码 (如 padding mask)
    if attention_mask is not None:
        extended_attention_mask = attention_mask.unsqueeze(1).unsqueeze(2) # [bs
        extended_attention_mask = (1.0 - extended_attention_mask) * -1e9 # 需要掩的
        scores = scores + extended_attention_mask # 掩的位置后面经过softmax就近似为0了

    # softmax + dropout
    scores = F.softmax(scores.float(), dim=-1).type_as(xq)
    scores = self.attn_dropout(scores)

    # 注意力加权求和
    output = scores @ xv # [bsz, n_heads, q_len, head_dim]

# 还原输出形状, 并通过输出线性层
output = output.transpose(1, 2).reshape(bsz, seq_len, -1)
output = self.resid_dropout(self.o_proj(output))

return output, past_kv

```

这里涉及到3个mask, 其作用分别为:

- 屏蔽未来信息: `torch.triu(torch.full((seq_len, seq_len), float("-inf")), device=scores.device), diagonal=1).unsqueeze(0).unsqueeze(0)`
- padding的token不参与注意力计算: `attention_mask`, shape是`[batch_size, seq_len]`。在MiniMind训练时, 默认是None, 可根据实际情况进行调整。

- 训练时, padding的token不参与loss计算: $\text{loss} = (\text{loss} * \text{loss_mask}).\text{sum}() / \text{loss_mask}.\text{sum}()$ [这个来自训练代码]

参考:

- [1] <https://medium.com/@joaolages/kv-caching-explained-276520203249>
- [2] <https://neptune.ai/blog/transformers-key-value-caching>