self Attention与kv cache

公式

$\text{Output} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \times V$ 复杂度是O(n^2)

KV Cache

推理阶段最常用的缓存机制，用空间换时间。

原理：

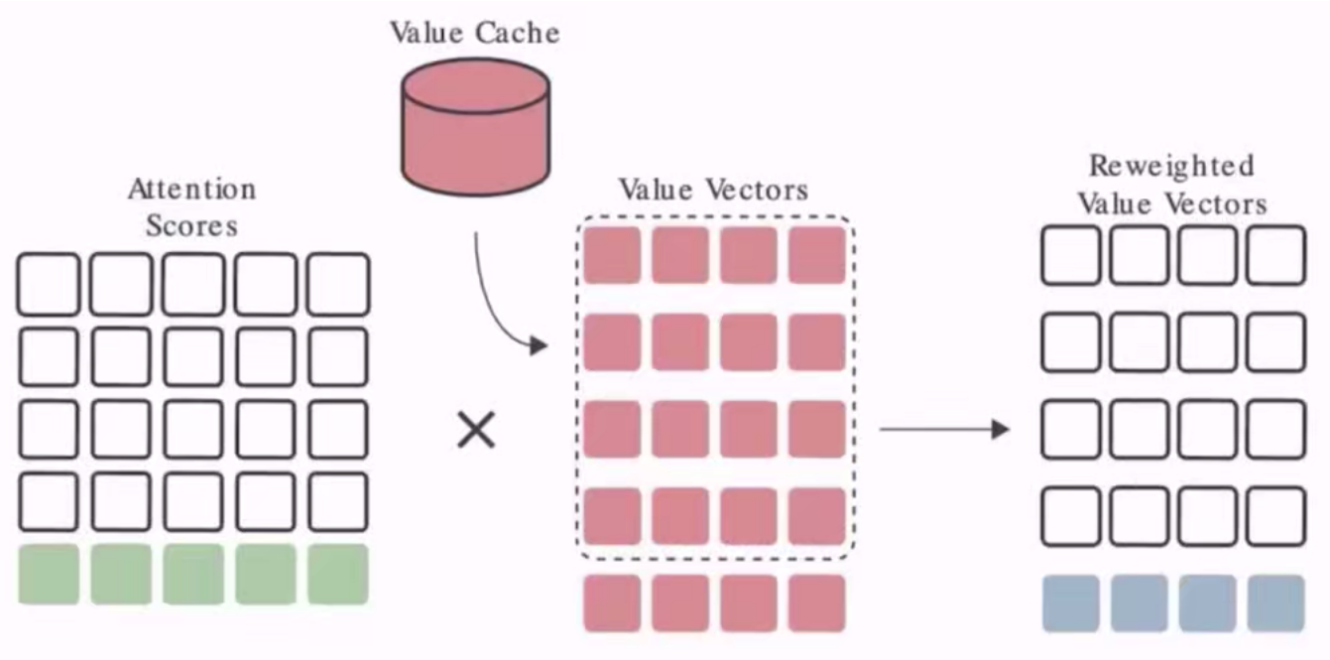
在进行自回归解码的时候，新生成的token会加入序列，一起作为下一次解码的输入。

由于单向注意力的存在，新加入的token并不会影响前面序列的计算，因此可以把已经计算过的每层的kv值保存起来，这样就节省了和本次生成无关的计算量。

通过把kv值存储在速度远快于显存的L2缓存中，可以大大减少kv值的保存和读取，这样就极大加快了模型推理的速度。

分别做一个k cache和一个v cache，把之前计算的k和v存起来

以v cache为例：



存在的问题：存储碎片化

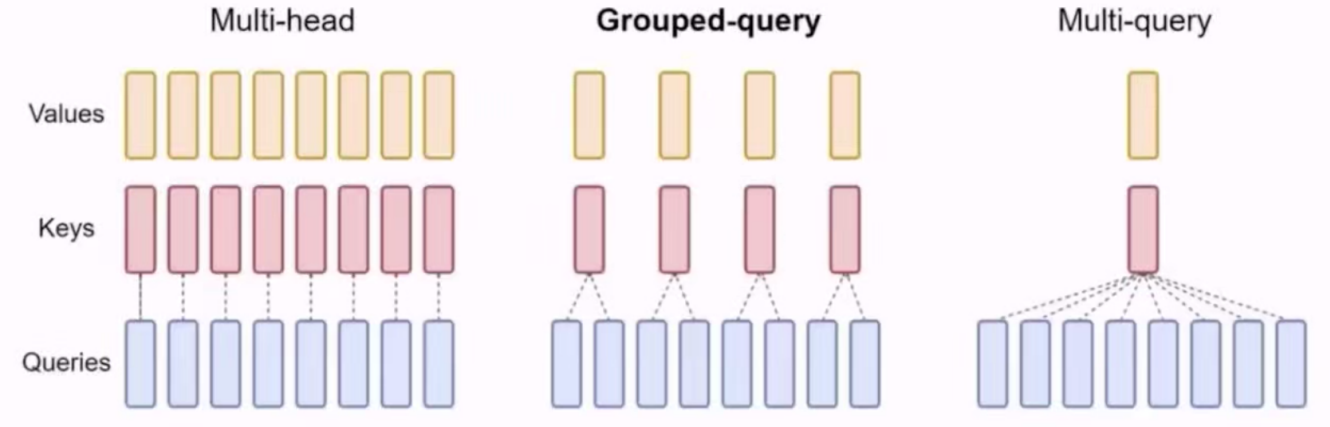
解决方法：page attention（封装在vllm里了）

MHA、MQA、GQA

Multi-Head Attention、Multi-Query Attention、Group-Query Attention

目的：优化KV Cache所需空间大小

原理是共享k和v，但是使用MQA效果会差一些，于是又出现了GQA这种折中的办法



面试题

为什么除以 $\sqrt{d_k}$

压缩softmax输入值，以免输入值过大，进入了softmax的饱和区，导致梯度值太小而难以训练。

Multihead的好处

1、每个head捕获不同的信息，多个头能够分别关注到不同的特征，增强了表达能力。多个头中，会有部分头能够学习到更高级的特征，并减少注意力权重对角线值过大的情况。

比如部分头关注语法信息，部分头关注知识内容，部分头关注近距离文本，部分头关注远距离文本，这样减少信息缺失，提升模型容量。

2、类似集成学习，多个模型做决策，降低误差

decoder-only模型在训练阶段和推理阶段的input有什么不同?

- **训练阶段**：模型一次性处理整个输入序列，输入是完整的序列，掩码矩阵是固定的上三角矩阵。
- **推理阶段**：模型逐步生成序列，输入是一个初始序列，然后逐步添加生成的 token。掩码矩阵需要动态调整，以适应不断增加的序列长度，并考虑缓存机制。

手撕必背-多头注意力

```
import torch.nn as nn
class MultiHeadAttentionScores(nn.Module):
```

```
    def __init__(self, hidden_size, num_attention_heads, attention_head_size):
        super(MultiHeadAttentionScores, self).__init__()
        self.num_attention_heads = num_attention_heads # 8,16, 32, 64

        # Create a query, key, and value projection layer
        # for each attention head.  W^Q, W^K, W^V
        self.query_layers = nn.ModuleList([
            nn.Linear(hidden_size, attention_head_size)
            for _ in range(num_attention_heads)
        ])

        self.key_layers = nn.ModuleList([
            nn.Linear(hidden_size, attention_head_size)
            for _ in range(num_attention_heads)
        ])

        self.value_layers = nn.ModuleList([
            nn.Linear(hidden_size, attention_head_size)
            for _ in range(num_attention_heads)
        ])

    def forward(self, hidden_states):
        # Create a list to store the outputs of each attention head
        all_attention_outputs = []

        for i in range(self.num_attention_heads): # i.e. 8
            query_vectors = self.query_layers[i](hidden_states)
            key_vectors = self.key_layers[i](hidden_states)
            value_vectors = self.value_layers[i](hidden_states)

            # softmax(Q&K^T)*V
            attention_scores = torch.matmul(query_vectors, key_vectors.transpose(-1, -2))
            # attention_scores combined with softmax--> normalized_attention_score
            attention_outputs = torch.matmul(attention_scores, value_vectors)
            all_attention_outputs.append(attention_outputs)

        return all_attention_outputs
```