自制大模型推理框架-第15次课程-KVCache的原理和实现

课程简介

视频

kv cache的原理和实现.mp4

Attention的计算

KV Cache已成为当前大型模型推理框架的标准配置，其主要功能是通过牺牲一定的空间复杂度来换取时间效率的提升，从而加快大型模型的推理预测速度。

由于GPT、LLama等模型采用自回归的方式，逐个步骤进行推理，上一步的预测结果将被纳入下一步的计算中作为输入。假设初始输入序列（input token）的长度（seqlen）为1，其维度为1×dim，在每次自回归的计算过程中，我们都会将前一次预测得到的单词添加到输入序列中。

第1步计算的时候

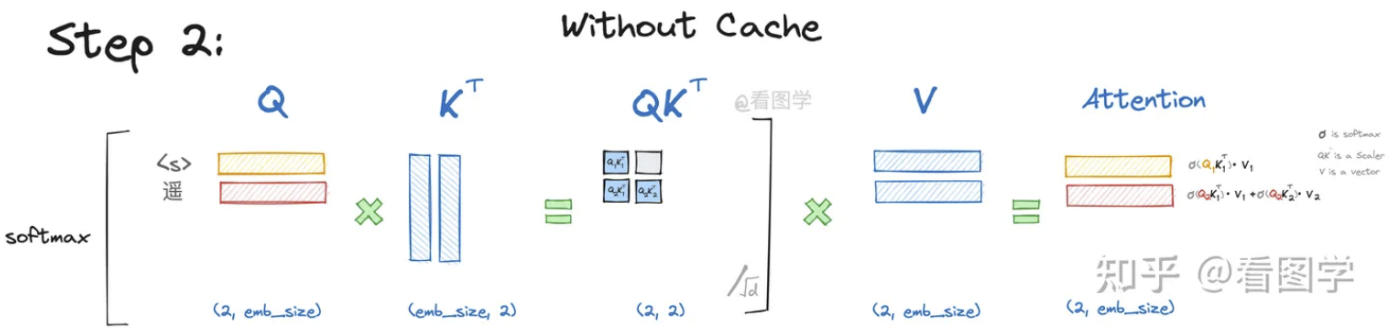
“<s>”->编码-->32，**32去查一个词表**，词表对应32位置记录了一个向量，得到开始词对应的向量，它的维度是1×dim。

当步长等于1的时候，我们有一个维度为*1×dim*的输入Token，1是输入token的个数，通过矩阵wq和矩阵wk（**wq和wk矩阵的维度均为*dim×dim***）将输入序列(input token)映射得到Q和K矩阵，随后就是将Q矩阵和K^T进行矩阵相乘得到V矩阵，在步长等于1的时候Q的维度为1×dim，K的维度同样为1×dim，**当Q矩阵乘以K的转置时得到一个1x1的分数矩阵**，随后该分数矩阵再对1×dim的V矩阵进行加权，得到最终的注意力输出。

input token 矩阵乘 wv矩阵维度同样是dim×dim，是固有的权重，input token×Wv矩阵 = V矩阵。

$$Att_1(Q,K,V) = softmax(Q_1K_1^T)V_1$$

第2步计算的时候



在第二步的计算过程中，我们处理了两个输入Token，维度为2×dim。根据上一步的输入，得到的预测单词为"遥"。

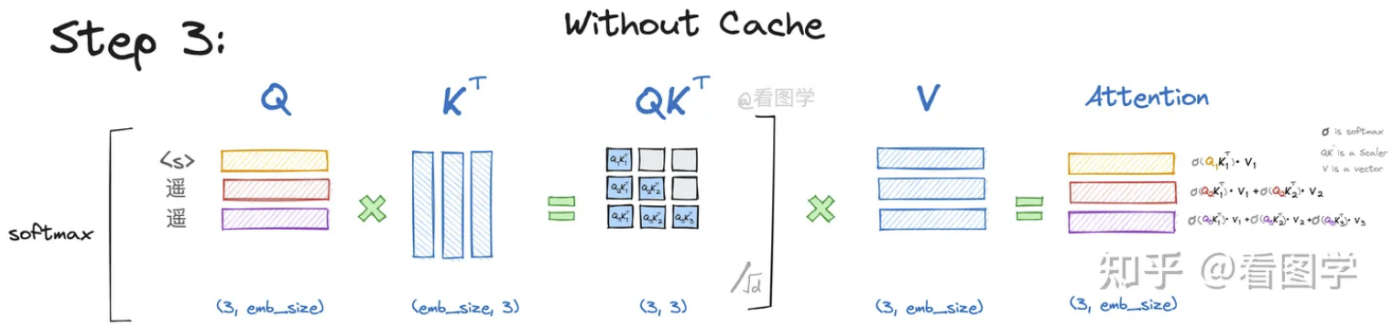
因此，当这些Token与Wq和Wk矩阵相乘进行映射时，我们得到的Q和K矩阵的维度为2×dim。**2×dim，Wq和Wk权重矩阵同样为dim×dim大小，v矩阵也是2×dim**。但是第2步在Q矩阵与K^T矩

阵相乘时，我们必须注意一个关键点：**Q矩阵的第1行与K^T矩阵的第2列是不会相乘的**，这一机制就是Transformer Decoder结构中的Causal Mask机制。

该机制的目的在于，在计算注意力(Attention)的过程中，将这些Token从注意力机制中屏蔽掉，确保模型在预测时仅能关注过去和当前的token，从而使得模型基于每个时间步骤可用的信息进行预测。对于Q矩阵的第一行而言，K矩阵的第二、三列代表的就是未来的数据。

第3步计算的时候

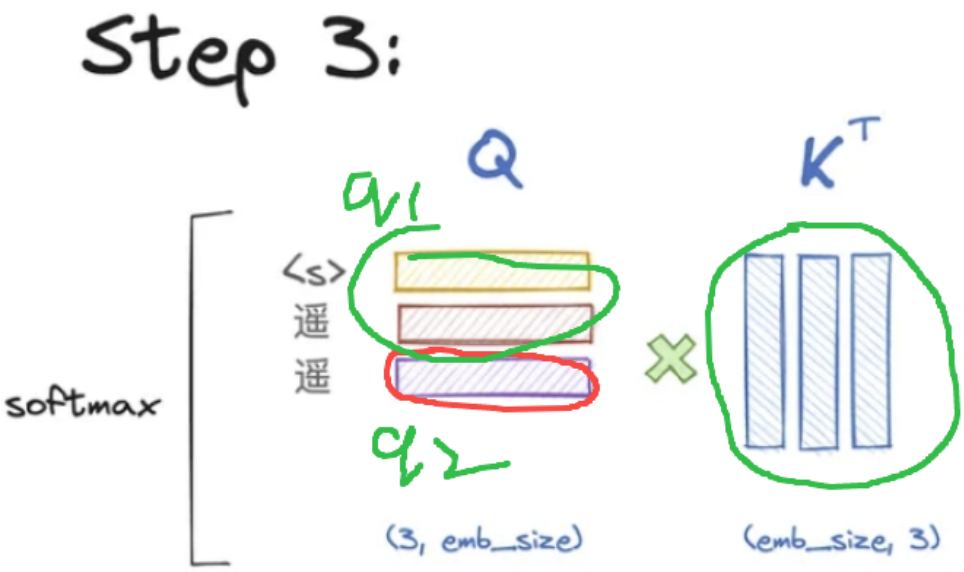
第三步计算同理，如下图所示：**我们处理3个输入Token，维度为3×dim 乘以 dim×dim的wq和wk权重矩阵分别得到Q和K，第三个Token来自于第2步中的预测。**



从图中就可以看到，我们在这3步中积攒了大量的重复计算（第一、第二行），这些信息明明在前两步中已经被计算过了，却还是要在第3步中做重复的计算。

因此，在第k次计算时，输入序列的长度将变为k，维度则为k×dim。这表明，随着计算次数的增加，输入序列的维度会持续增长，从而可能引起重复计算，为了剖析计算过程中的这种冗余，**我们可以将原本k×dim维度的Query矩阵拆分为两部分：**

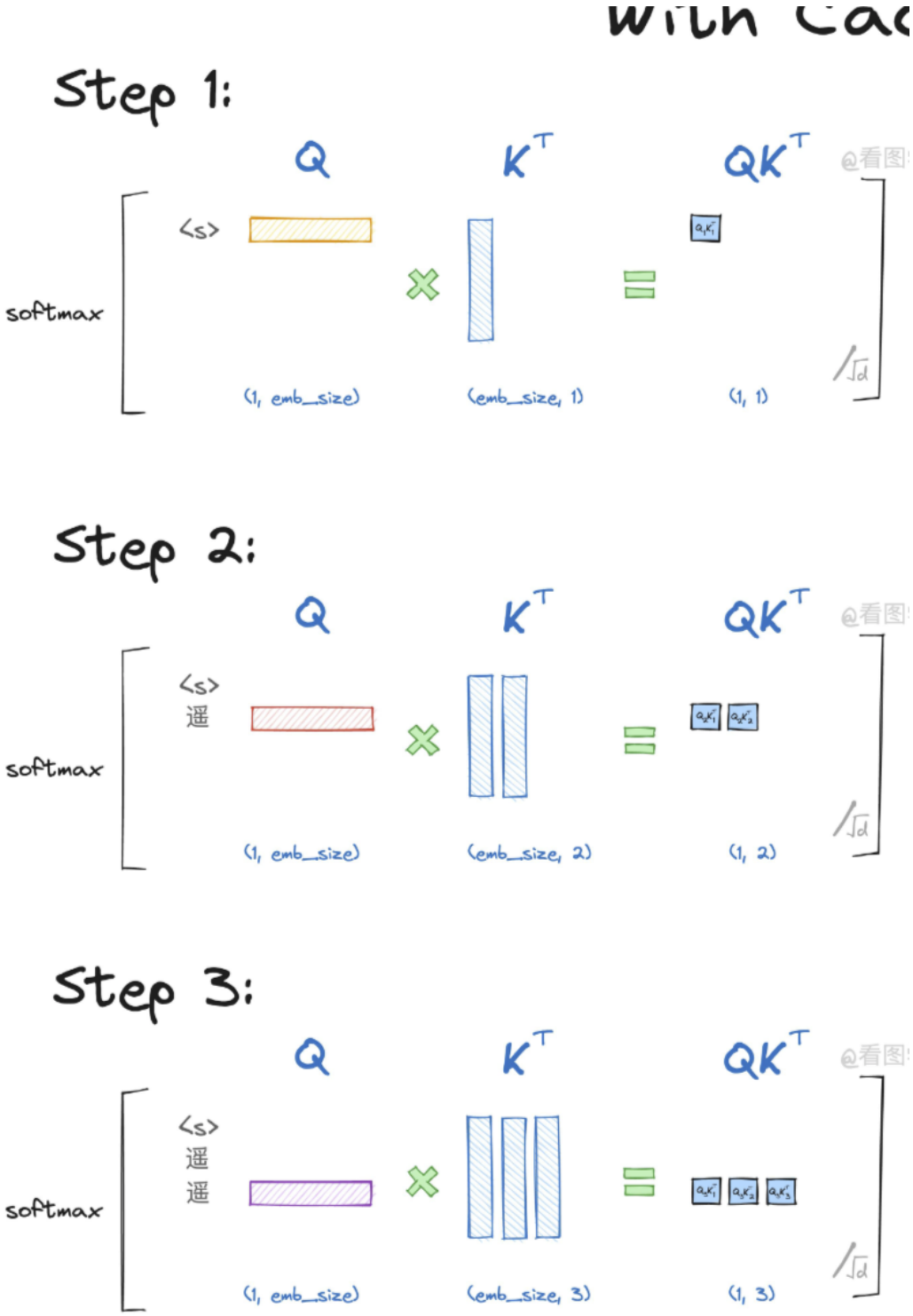
1. 第一部分是包含第0行至第k-1行的Query1矩阵，其维度为(k-1)×dim；
2. **第二部分是仅包含第k行的Query2矩阵，维度为1×dim**。在执行第k次自回归计算时，我们只需要关注Query矩阵的第k行(也就是Query2矩阵)和Key矩阵第0到k列进行矩阵相乘就可以。



简化Attention的计算

在上一段中我们说过了，在执行第k步自回归计算时，我们只需要关注**Query2**矩阵也就是Query矩阵的最后一

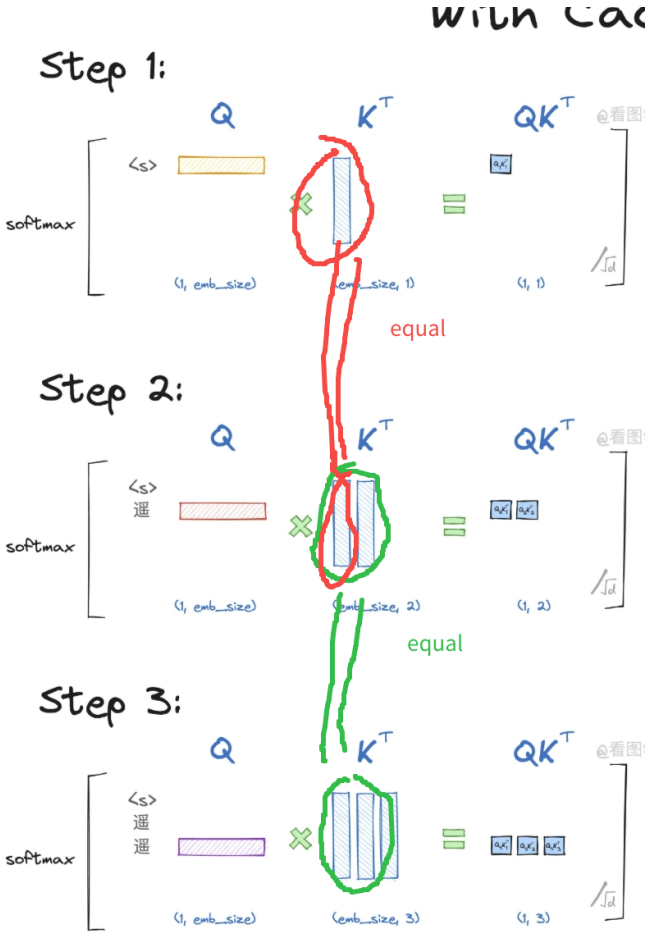
行，和Key矩阵所有列相乘得到结果就可以了。第一步当中，q1是空的，q2是第一行。第二步当中，q2是第2行，q1是第一行。



KV-Cache

K-Cache

KVCache顾名思义就是缓存一部分K矩阵和V矩阵，就像我们上文所说的K**矩阵同样可以分为Key1矩阵和Key2矩阵**，Key1矩阵是在前k-1步计算中得到的，Key2是在当前步骤中得到的。我们看图：



下一步的Key矩阵中的前k-1列和上一步中Key矩阵的前k'列是相同的，**例如在第3步中Key矩阵的第1和第2两列和第2步中的Key矩阵中是相同的。为什么会相同呢，在第2步中Key矩阵是这么得到的：**

[input token 1, input token 2] × Wk(dim×dim的权重矩阵) = K矩阵

而在第3步中则有：

[input token 1, input token 2, **input token 3 当前步骤的新词**] × Wk = K矩阵

因此第3步中得到的Key矩阵的前两列和第2步的K矩阵是相同的，既然是相同的，我们为什么不在计算Key矩阵的时候保存前k-1列，在第k步的时候只计算第k列（当前的input token×Wk矩阵）随后再拼接成一个完整的Key矩阵。这种记录Key矩阵前k-1列的方法就是所谓的KV-Cache。

如何改良

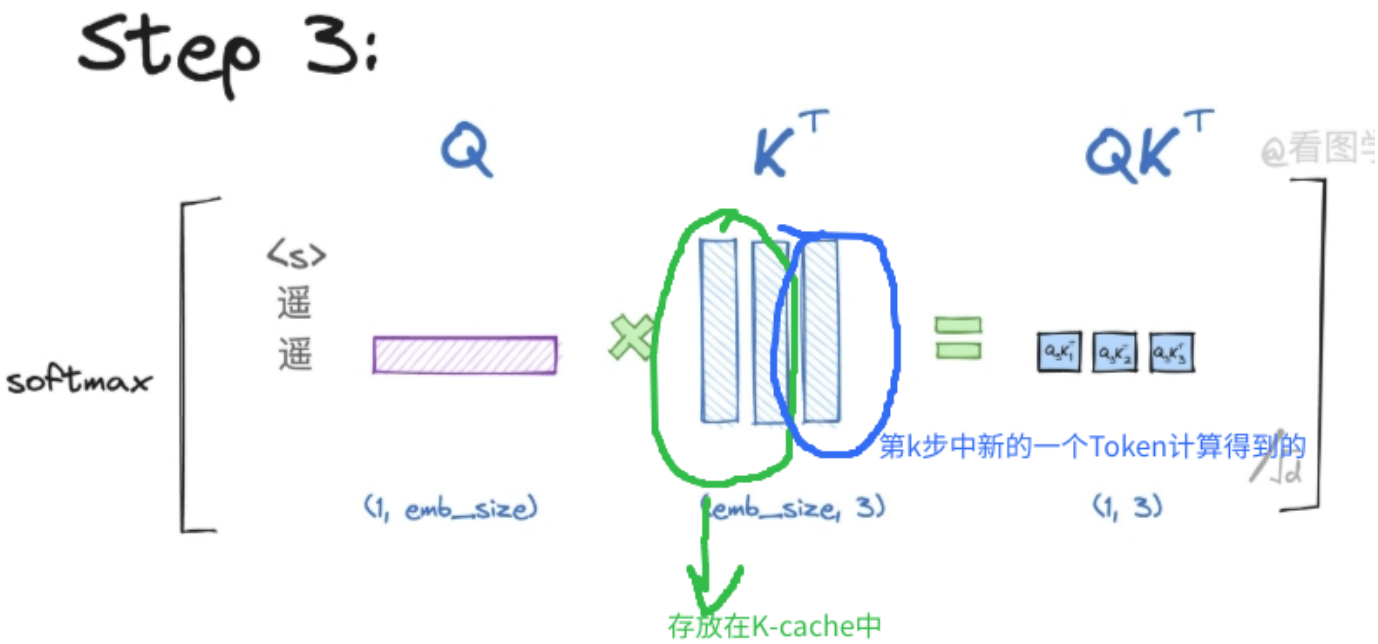
原来的步骤--[input token 1, input token 2, **input token 3 当前步骤的新词**] × Wk = K矩阵

改良后

K1列，K2列缓存下来

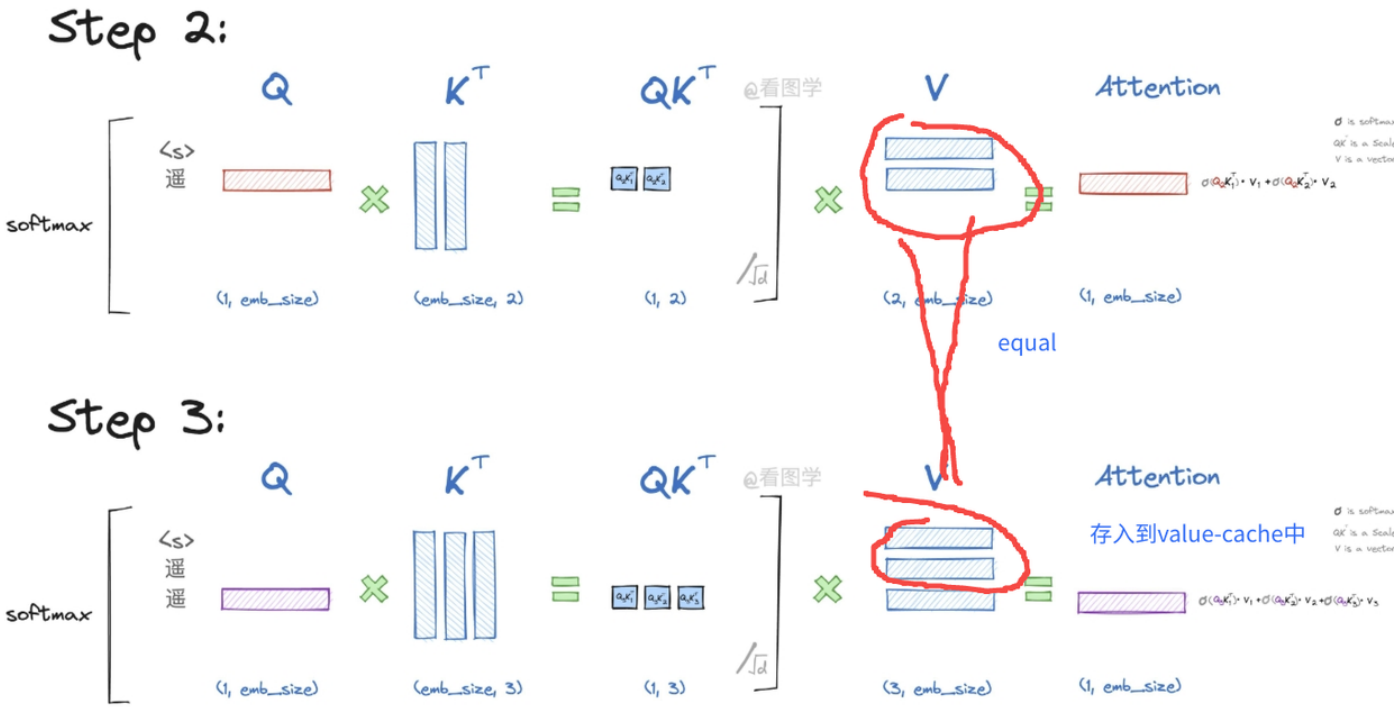
现有的步骤**input token 3 当前步骤的新词 × Wk 得到K3 列**

在计算的时候需要完整的K矩阵拼接起来，



V-Cache

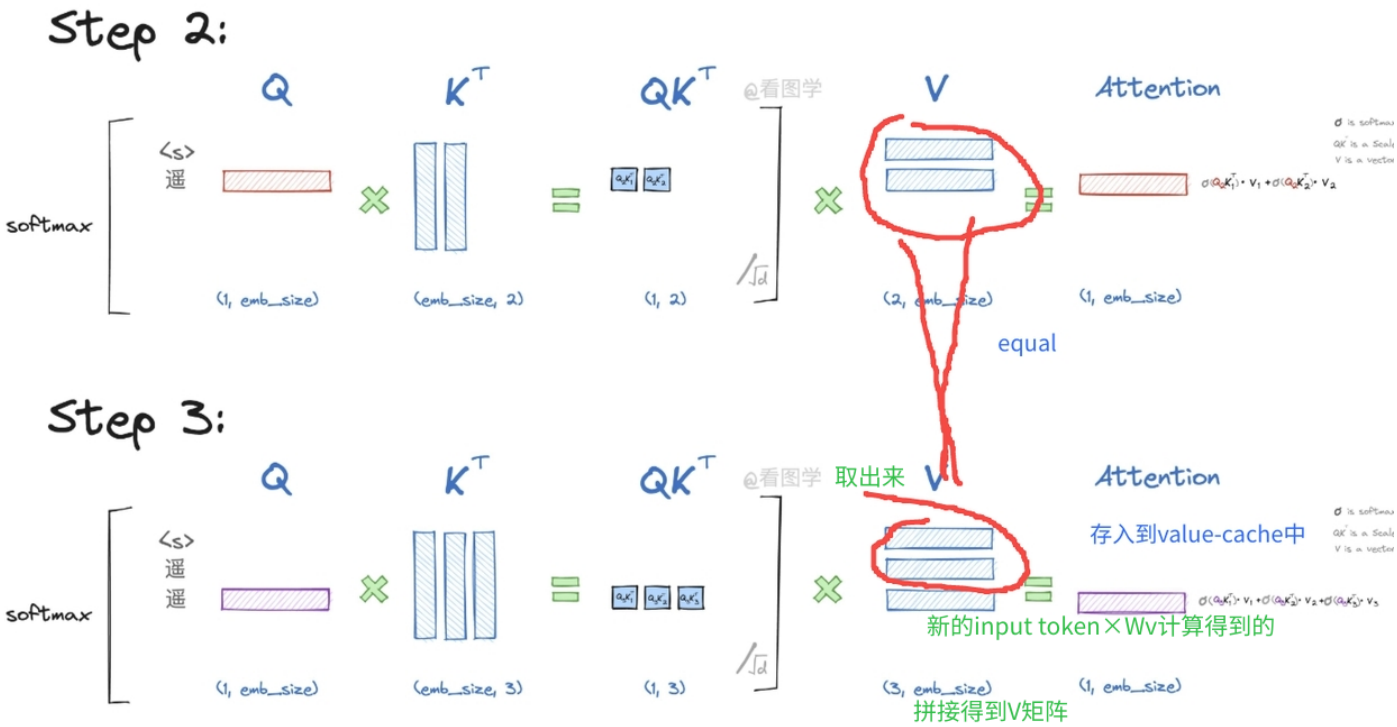
对于V矩阵同样的，V矩阵也是由输入的多个token和Wv权重矩阵相乘得到的，在第k步和第k-1步得到的矩阵Value1和Value2有一部分是重叠的：



所以我们需要将前一步计算出来的Value1矩阵放在一块Value-Cache区域中，**等到当前步的时候我们只需要将当前的输入token和Wv矩阵进行相乘得到Value2矩阵，再将它们拼接起来就可以得到完整的Value矩阵并开始注意力的计算。**

原来的步骤：[input token 1, input token 2, **input token 3 当前步骤的新词**] × Wv = V矩阵

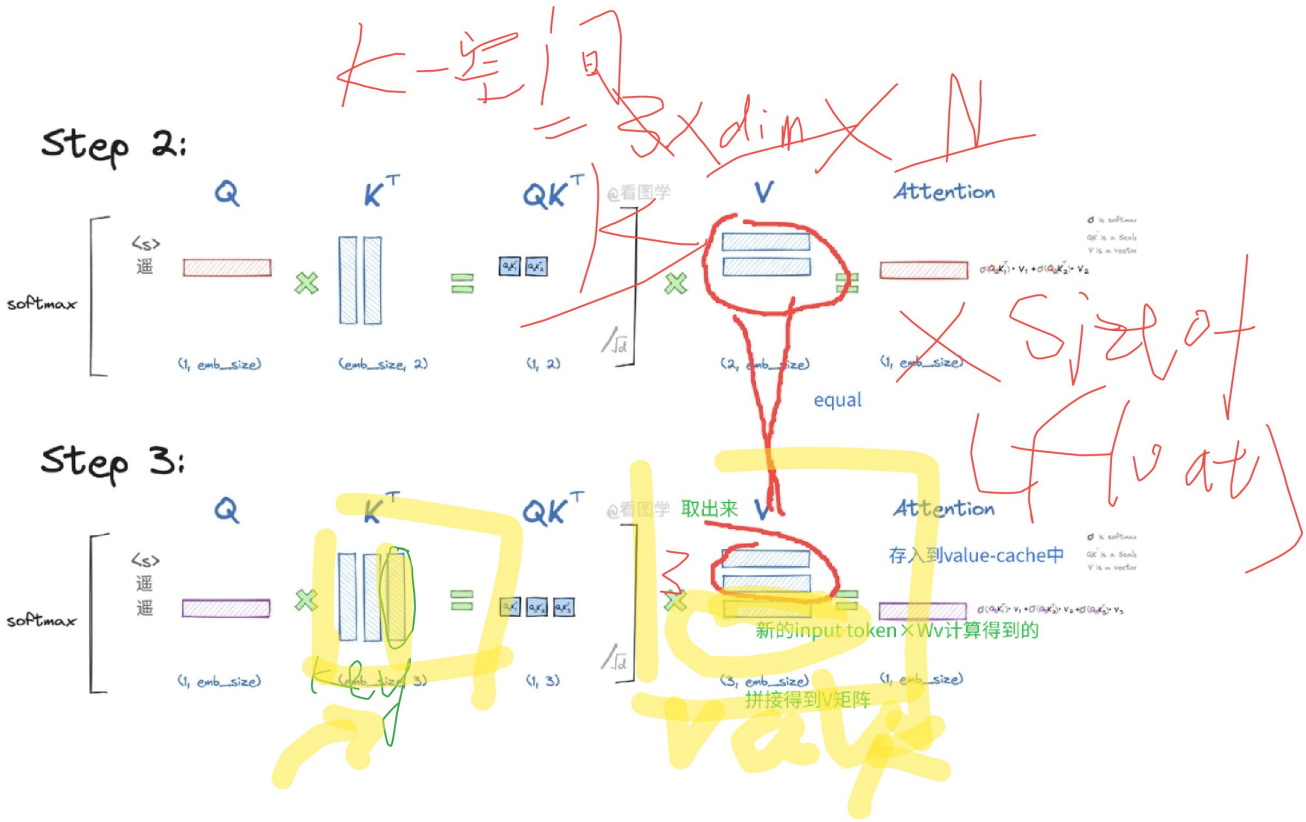
现有的步骤：V1行和V2行都被缓存了，**input token 3 × Wv得到V3行，然后再拼接起来得到V矩阵。**



为KV-Cache申请空间

```
1   // kv cache
2   tensor::Tensor key_cache(base::DataType::kDataTypeFp32, config_->layer_num_,
    config_->seq_len_, config_->kv_dim_, true, alloc);
3
4   tensor::Tensor value_cache(base::DataType::kDataTypeFp32, config_-
    >layer_num_, config_->seq_len_,  config_->kv_dim_, true, alloc);
```

从上文的分析中，我们可以得知，为了给执行了**k**步计算的Transformer块准备KV Cache的空间，我们所需申请的空间大小应为k×dim。如果整个模型包含了**N**个这样的Transformer块，那么总共需要申请的显存空间将是**N×k×dim×sizeof(float)**。



由于在每一步自回归预测中，我们都会增加一个输入单词，因此单词的总数最多不会超过最大序列长度（max_seq_len）。因此，所需申请的显存空间大小可以表示为max_seq_len×N×dim×sizeof(float)，所以我们在**LLama2Model::init_mem()** 方法中申请了这块空间用于后续使用。

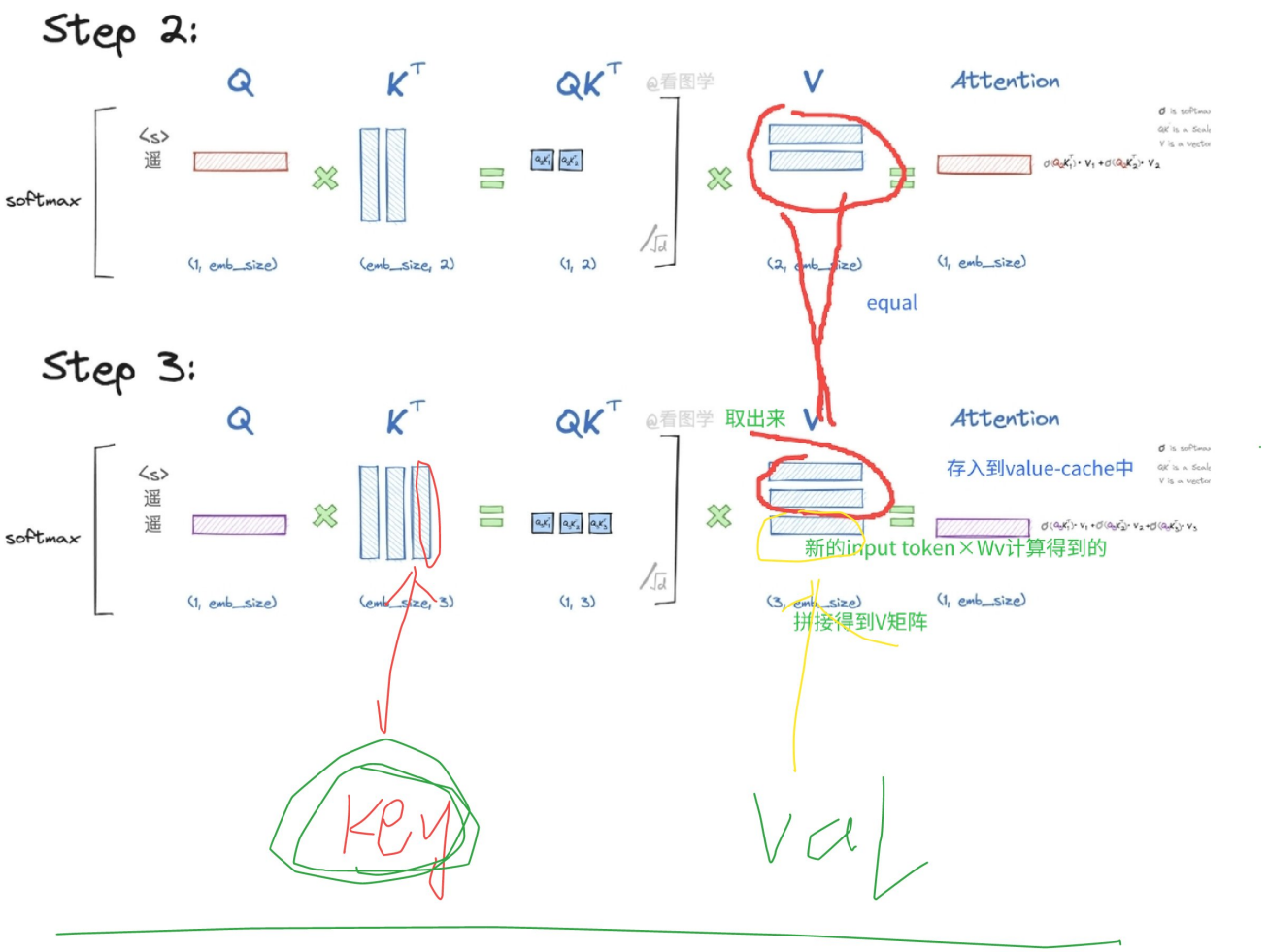
对KV-Cache空间的拆分

```
std::pair<tensor::Tensor, tensor::Tensor>
    LLama2Model::slice_kv_cache(int32_t layer_idx,int32_t token_pos) const {
    // (N,max_selen,dim)
    // 索引到第几个transformer块,layer_idx个transformer块，第token_pos个位置的
    int32_t layer_offset = layer_idx * config_->seq_len_ * config_->kv_dim_;
    int32_t cache_offset = layer_offset + token_pos * config_->kv_dim_;
    简单来说，对于第layer_idx个transformer块和第token pos步，对应的存放位置为cache_offet
    它的索引就是(layer_index, token pos, :) kv_cache[layer_index,token_pos,:]


    // 把原指针做一个封装
    float* key_cache_ptr =
        const_cast<float*>(get_buffer(ModelBufferType::kKeyCache).ptr<float>(cache_offset));
    float* val_cache_ptr =
```

```
15          const_cast<float*>(get_buffer(ModelBufferType::kValueCache).ptr<float>
    (cache_offset));
16
17      auto key_cache = std::make_shared<base::Buffer>(config_->kv_dim_ *
    sizeof(float), nullptr,key_cache_ptr, true);
18      auto val_cache = std::make_shared<base::Buffer>(config_->kv_dim_ *
    sizeof(float), nullptr, val_cache_ptr, true);
19
20
21
22      key_cache->set_device_type(device_type_);
23      val_cache->set_device_type(device_type_);
24      tensor::Tensor key(base::DataType::kDataTypeFp32, config_->kv_dim_);
25      tensor::Tensor val(base::DataType::kDataTypeFp32, config_->kv_dim_);
26      key.assign(key_cache);
27      val.assign(val_cache);
28      return {key, val}; // 表示的是
29  }
```

在上一节中，我们已经了解到KV-Cache的总空间大小为max_seqlen×layer_num×dim。因此，在第token_pos步时，我们可以通过索引位置(token_pos, layer_idx)来获取第layer_idx层的KV-Cache，具体代码见第3至第4行。接下来，我们将这个索引位置对应的区域封装为两个Tensor，并将这两个Tensor指定特定类型后返回，以便用于存储缓存数据。


Step 2:
Q
K^T
QK^T
@看图学
V
Attention
softmax
<s>
遥
(1, emb_size)
(emb_size, 2)
(1, 2)
(2, emb_size)
(1, emb_size)
equal
Step 3:
Q
K^T
QK^T
@看图学 取出来
V
Attention
存入到value-cache中
softmax
<s>
遥
遥
(1, emb_size)
(emb_size, 3)
(1, 3)
(3, emb_size)
(1, emb_size)
新的input token×Wv计算得到的
拼接得到V矩阵
key
val