Transformers KV 缓存详解-加速LLM推理


lucas大叔
NLPer，LLM应用探索者、实践者

Transformer架构可以说是现代深度学习领域最具影响力的创新之一。它是在 2017 年著名论文《Attention Is All You Need✦》中提出的，现已成为大多数语言相关建模的首选方法，包括所有大语言模型（LLM），如 GPT 系列，以及许多计算机视觉任务。

随着这些模型的复杂性和规模的增长，优化其推理速度的需求也随之增长，尤其是在用户希望立即得到回复的聊天应用中。键值 (KV) 缓存是实现这一目的的一种巧妙技巧，让我们来看看它是如何工作的以及何时使用。

Transformer架构概述

在深入研究 KV 缓存之前，我们需要先了解transformer中使用的注意力机制。要想了解 KV 缓存✦是如何优化transformer推理的，就必须了解注意力机制的工作原理。

我们将重点讨论用于生成文本的自回归模型✦。这些所谓的decoder模型包括 GPT 系列、Gemini、Claude 或 GitHub Copilot。它们的训练任务很简单：预测序列中的下一个token。在推理过程中，提供给模型一些文本，其任务就是预测这些文本应该如何继续。

从高层次的角度来看，大多数transformer都由几个基本构件组成：

- 用于将输入文本分割成word或sub-word等子部分的**tokenizer**。
- 将生成的tokens（及其在文本中的相对位置）转换为向量的**嵌入层**。
- 几个基本的**神经网络层**，包括dropout、层归一化✦和常规前馈线性层。

上述清单中缺少的最后一个组成部分是稍微复杂一些的自注意力(self-attention)模块。

自注意力模块可以说是transformer架构中唯一高级的逻辑模块。它是每个transformer的基石，使transformer在生成输出时能够关注输入序列的不同部分。正是这种机制使transformer能够有效地建模远距离依赖关系。

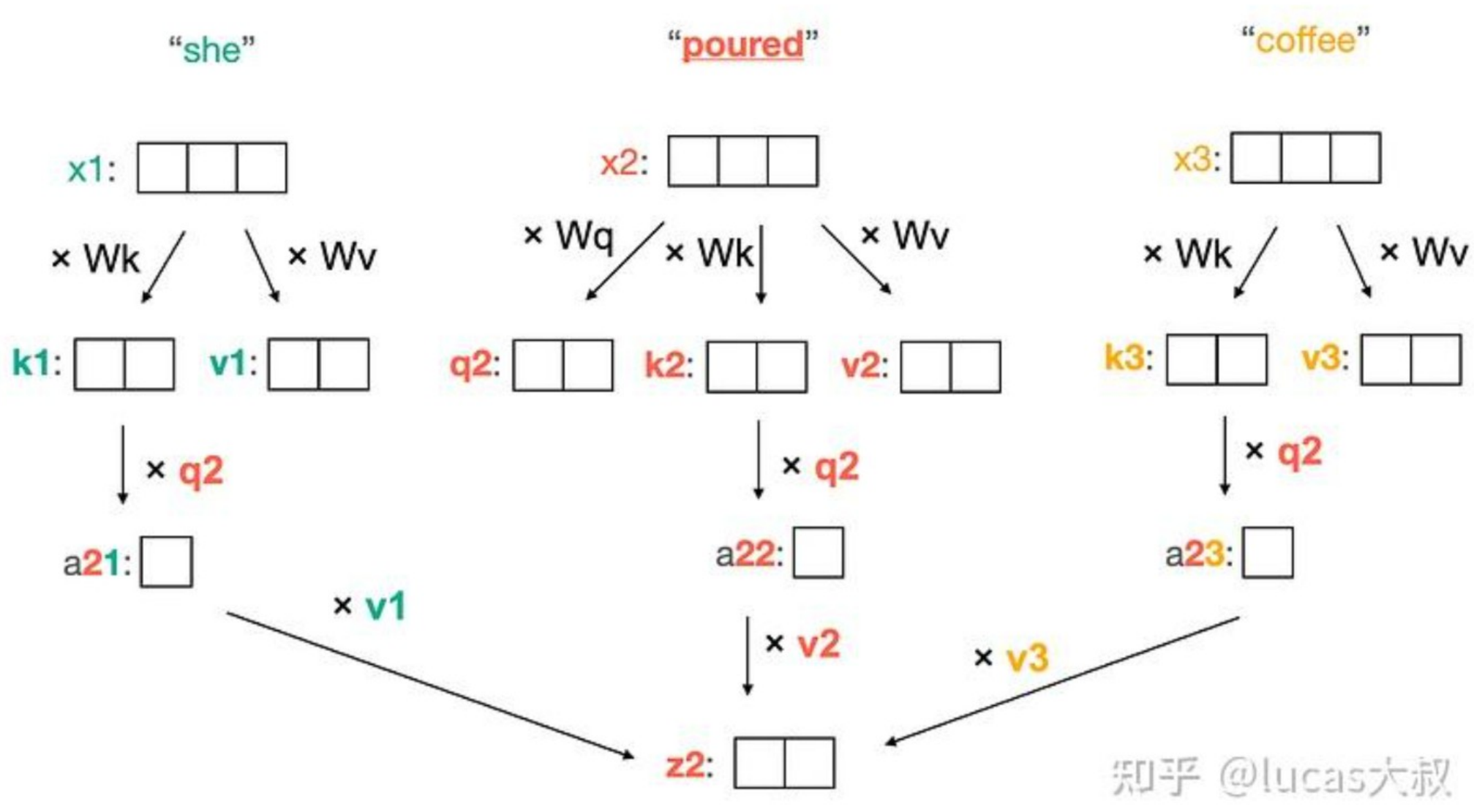
让我们来详细了解一下自注意力模块✦。

基本自注意力模块

自注意力是一种机制，它允许模型在生成下一个token时 "关注 "输入序列的特定部分。例如，在生成 "She poured the coffee into the cup"这个句子时，模型可能会更多地关注 "poured "和 "coffee "这两个词，以预测 "into "为下一个词，因为这两个词提供了下一个词可能出现的上下文（与 "she "和 "the "相对而言）。

从数学角度讲，自注意力的目标是将每个输入（嵌入token）转换为所谓的上下文向量，该向量综合了给定文本中所有输入的信息。请看文本 "She poured coffee"。注意力将计算三个上下文向量，每个输入token（假设token为单词）对应一个。

为了计算上下文向量，自注意力计算了三种中间向量：查询(Q)、键(K)和值(V)。下图分步展示了如何计算第二个单词 "poured "的上下文向量：



我们将三个标记化的输入分别记为 x1、x2 和 x3。图中把它们描绘成三维向量，但实际上，它们会有成百上千个维度。

第一步，自注意力将每个输入分别乘以两个权重矩阵 Wk 和 Wv。现在正在计算上下文向量的输入（在我们的例子中为 x2）会额外乘以第三个权重矩阵 Wq。所有三个 W 矩阵都是常见的神经网络权重，在学习过程中随机初始化和优化。这一步的输出是每个输入的键（k）和值（v）向量，以及正在处理的输入的附加查询（q）向量。

第二步，每个输入的key向量乘以正在处理的输入的查询向量（此处为q2）。然后对输出进行归一化处理（图中未显示），得出注意力权重。在我们的例子中，a21 是输入 "She"和 "poured "之间的注意力权重。

最后，每个注意力权重乘以相应的value向量。在我们的例子中，上下文向量 z2 与输入 x2 "poured"相对应。上下文向量是自注意力模块的输出。

如果你觉得读代码比读图表更容易，那么不妨看看 Sebastian Raschka 对基本自注意力模块的实现。该代码是他的著作《Build A Large Language Model (From Scratch)》的一部分：

```
import torch

class SelfAttention_v2(torch.nn.Module):

    def __init__(self, d_in, d_out, qkv_bias=False):
        super().__init__()
        self.W_query = torch.nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key   = torch.nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = torch.nn.Linear(d_in, d_out, bias=qkv_bias)

    def forward(self, x):
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)

        attn_scores = queries @ keys.T
        attn_weights = torch.softmax(
          attn_scores / keys.shape[-1]**0.5, dim=-1
        )

        context_vec = attn_weights @ values
        return context_vec
```

Sebastian的代码对矩阵进行如下操作：forward() 方法中的 x 对应于图中的 x1、x2 和 x3 向量，它们堆叠在一起成为一个有三行的矩阵。这样，只需将 x 与 W_key 相乘，就能得到 keys，一个由三行组成的矩阵（在我们的例子中为 k1、k2 和 k3）。

从对自注意力的简要解释中，我们可以得到一个重要启示，那就是在每次向前传递时，先用查询乘以键，然后再乘以值。请在继续阅读时牢记这一点。

高级自注意力模块

上述自注意力变体是其最简单的普通形式。当今最大的 LLM 通常使用略有改动的变体，这些变体通常在三个方面与我们的基本形式有所不同：

- 注意力是因果关系。
- 注意力权重使用了dropout功能。
- 采用多头注意力。

因果注意力是指模型在预测下一个token时，只考虑序列中的前一个token，而不能 "瞻前顾后 "地预测未来的词。回到我们的例子 "She poured coffee."，当模型得到单词 "She"并试图预测下一个单词（"poured"是正确的）时，它不应该计算或获取 "coffee"和其他单词之间的注意力权重，因为 "coffee"这个单词还没有出现在文本中。因果注意力通常是通过将注意权重矩阵的 "前瞻 "部分屏蔽为零来实现的。

其次，为了减少训练过程中的过拟合，通常会对注意力权重进行dropout。这意味着在每次前向传递时，其中一些权重会被随机设置为零。

最后，基本注意力可称为单头注意力，即只有一组 Wk、Wq 和 Wv 矩阵。提高模型容量的一个简单方法是改用多头注意力✦。这可以归结为拥有多组 W 矩阵，从而拥有多个查询、键和值矩阵，以及每个输入的多个上下文向量。

此外，一些transformer还对注意力模块进行了额外的修改，以提高速度或准确性。其中比较流行的三种是：

- 分组查询注意力(Grouped-query attention)：不单独查看每个输入token，而是对token进行分组，使模型能够同时关注相关的词组，从而加快处理速度。Llama 3、Mixtral 和 Gemini 都采用了这种方法。
- 分页注意力(Paged attention)：注意力被分解成 "pages"或token块，因此模型一次只处理一个page，这样在处理很长的序列时速度会更快。
- 滑窗注意力(Sliding-window attention)：模型只关注每个token周围固定 "窗口 "内的邻居token，因此它只关注局部上下文，而无需查看整个序列。

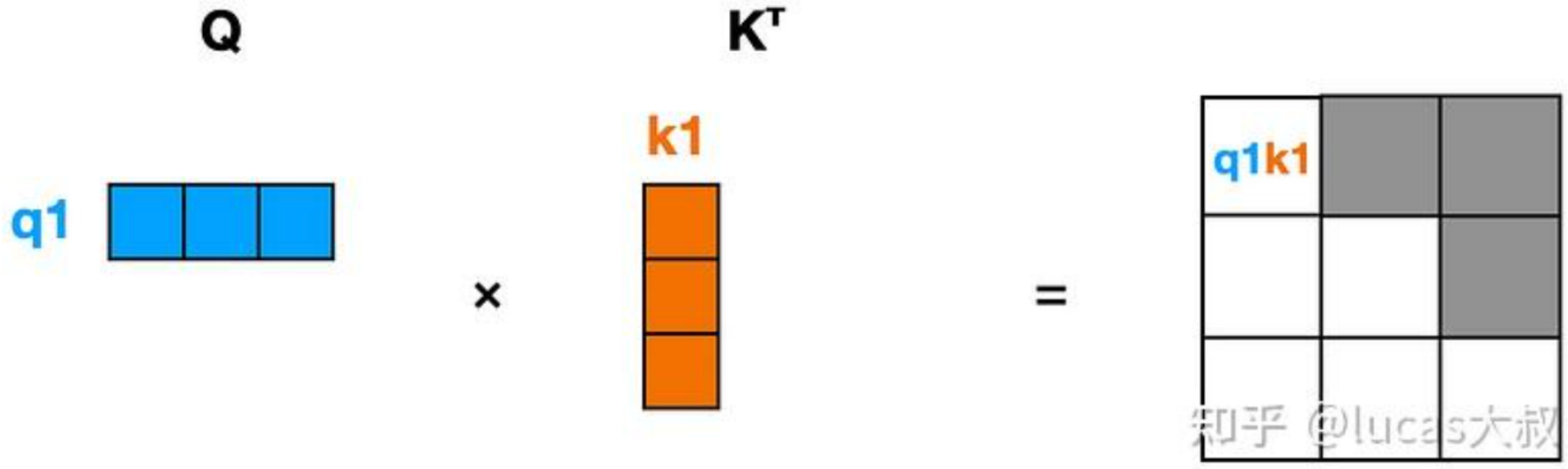
所有这些实现自注意力的高级方法都无法改变其基本前提和所依赖的基本机制：我们总是需要用查询乘以键，然后再乘以值。事实证明，在推理时，这些乘法的效率非常低。让我们来看看为什么会这样。

什么是键值缓存

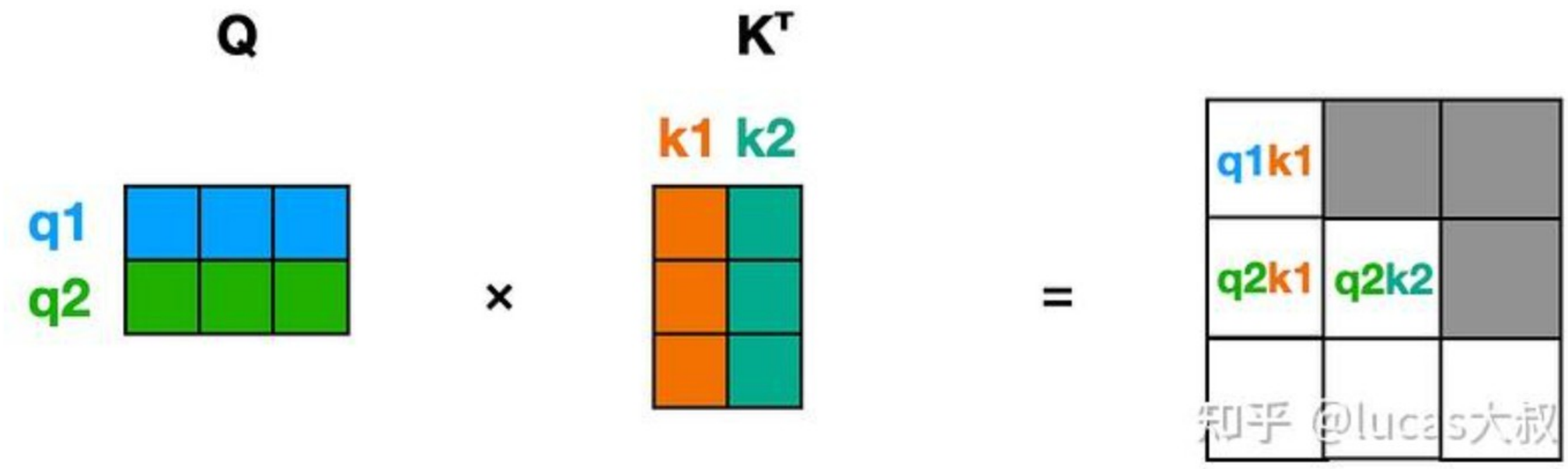
在推理过程中，transformer每次生成一个token。当我们通过 "She"来提示模型开始生成时，它就会生成一个词，比如 "poured"（为了避免干扰，我们仍然假设一个token就是一个词）。然后，把 "She poured"传递给模型，它就会生成 "coffee"。接下来，我们传递 "She poured coffee"，并从模型得到序列结束token，这表明它认为生成已经完成。

这意味着我们已经运行了三次前向传递，每次都是将查询乘以键值来获得注意力分数（后面的乘以值也是如此）。

在第一次前向传递中，只有一个输入token（"She"），因此只有一个键向量和一个查询向量。我们将它们相乘，得出 q1k1 注意力得分。



接下来，我们将 "She poured"传递给模型。现在它看到了两个输入token，因此注意力模块内部的计算过程如下：

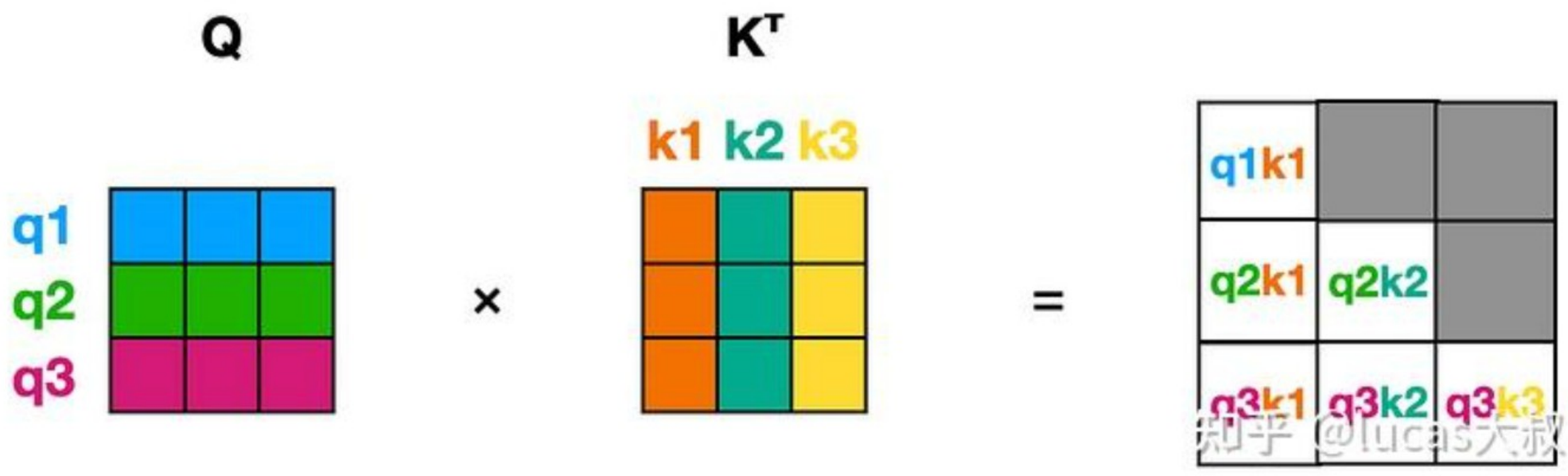


我们做了乘法来计算三个项，但 q1k1 的计算是不必要的--我们之前已经计算过了！这个 q1k1 元素与上一次前向传递中的元素相同，因为

- q1 的计算方法是输入（"She"）的嵌入乘以 Wq 矩阵
- k1 的计算方法是输入（"She"）的嵌入乘以 Wk 矩阵
- 在推理时，嵌入和权重矩阵都是恒定的

请注意注意力分数矩阵中的灰色条目：这些条目被屏蔽为零，以实现因果注意力。例如，右上角 q1k3 的位置没有显示给模型，因为我们在生成第二个单词时并不知道第三个单词（和 k3）。

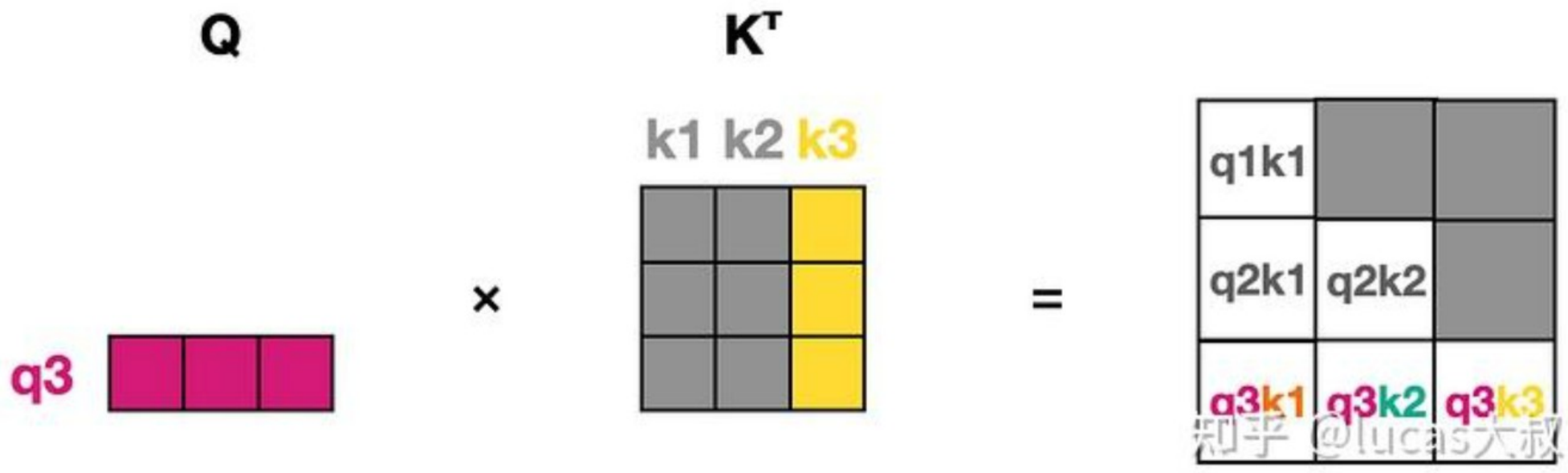
最后，下面是第三次前向传递中的查询×键的计算说明。



我们需要计算六个值，其中一半我们已经知道，不需要重新计算！

你可能已经对键值缓存有了一些直觉。在推理过程中，当计算键（K）和值（V）矩阵时，我们会将其元素存储到缓存中。缓存是一个辅助存储器，可以进行高速检索。当后续token生成时，只计算新token的键和值。

例如，在缓存的情况下，第三次前向传递是这样的：



在处理第三个token时，我们不需要重新计算前一个token的注意力分数，可以从缓存中检索前两个token的键和值，从而节省计算时间。

评估键值缓存的影响

键值缓存可能会对推理时间产生重大影响。这种影响的大小取决于模型架构。可缓存计算越多，推理时间缩短的可能性就越大。

让我们使用 EleutherAI 提供的 GPT-Neo-1.3B 模型来分析 K-V 缓存对生成时间的影响。

首先，定义一个计时器上下文管理器来计算生成时间：

```
import time


class Timer:
    def __enter__(self):
        self._start = time.time()
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self._end = time.time()
        self.duration = self._end - self._start

    def get_duration(self) -> float:
        return self.duration
```

接下来，从Hugging Face Hub加载模型，设置tokenizer，并定义prompt：

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

model_name = "EleutherAI/gpt-neo-1.3B"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

input_text = "Why is a pour-over the only acceptable way to drink coffee?
```

最后，定义运行模型推理的函数：

```
def generate(use_cache):
    input_ids = tokenizer.encode(
        input_text,
        return_tensors="pt").to(device),
    )
 output_ids = model.generate(
     input_ids,
     max_new_tokens=100,
     use_cache=use_cache,
 )
```

请注意我们传递给 model.generate 的 use_cache 参数：它可以控制是否使用 K-V 缓存。有了这种设置，就可以测量有 K-V 缓存和无 K-V 缓存时的平均生成时间：

```
for use_cache in (False, True):
    gen_times = []
    for _ in range(10):
      with Timer() as t:
        generate(use_cache=use_cache)
      gen_times += [t.duration]
    print(
        f"Average inference time with use_cache={use_cache}: ",
        f"{np.round(np.mean(gen_times), 2)} seconds",
    )
```

在 Python 3.10.12 上使用 torch==2.5.1+cu121 和 transformers==4.46.2 执行了这段代码，并获得了以下输出：

```
Average inference time with use_cache=False: 9.28 seconds
Average inference time with use_cache=True: 3.19 seconds
```

正如你所看到的，在这种情况下，缓存的速度几乎提高了三倍。

挑战与权衡

通常情况下，天下没有免费的午餐。我们刚刚看到的生成速度提升只能以增加内存使用量为代价，
这就需要在生产系统中进行周到的管理。

时延与内存的权衡

在缓存中存储数据会占用内存空间。内存资源有限的系统可能难以容纳这些额外的内存开销，从而可能导致内存不足错误。当需要处理较长的输入时，情况尤其如此，因为缓存所需的内存与输入长度呈线性增长。

另一个需要注意的问题是，缓存消耗的额外内存不能用于存储数据批次。因此，可能需要减少批次大小，使其保持在内存限制范围内，从而降低系统的吞吐量。

如果缓存消耗的内存成为一个问题，我们可以用额外的内存来换取部分模型精度。具体来说，我们可以截断序列、修剪注意力头或量化模型：

- 序列截断指的是限制最大输入序列长度，从而以丢失长期上下文为代价限制缓存大小。在与长上下文相关的任务中，模型的准确性可能会受到影响。
- 减少层数或注意力头数量，从而减少模型大小和缓存内存需求，是另一种回收内存的策略。不过，降低模型复杂度可能会影响其准确性。
- 最后是量化，即使用低精度数据类型（如 float16 而不是 float32）进行缓存，以减少内存使用量。然而，这同样会影响模型精度。

总之，K-V 缓存提供的更快延时是以增加内存使用为代价的。如果内存充足，这就不是问题。但是，如果内存成为瓶颈，我们可以通过各种方式简化模型来收回内存，从而从时延-内存权衡转到时延-精度权衡。

生产系统中的 K-V 缓存管理

在拥有众多用户的大规模生产系统中，需要对 K-V 缓存进行适当管理，以确保一致、可靠的响应时间，同时防止内存消耗过多。其中最关键的两个方面是缓存失效（何时清除缓存）和缓存重用（如何多次使用同一缓存）。

缓存失效

三种最流行的缓存失效策略是基于会话的清除(session-based clearing)、存活时间失效(time-to-live invalidation)和基于上下文相关性的方法(contextual)。让我们按这个顺序来探讨它们。

最基本的缓存失效策略是基于会话的清除。只需在用户会话或与模型的对话结束时清除缓存。这种简单的策略非常适合会话时间短且相互独立的应用。

在客户支持聊天机器人应用中，每个用户会话通常代表一次单独对话，用户在对话中就特定问题寻求帮助。在这种情况下，不太可能再次需要缓存中的内容。一旦用户结束聊天或会话因未活跃而超时，清除 K-V 缓存是个不错的选择，这样可以为应用腾出内存来处理新用户。

不过，在单个会话较长的情况下，有比基于会话的清除更好的解决方案。在存活时间（TTL）失效中，缓存内容会在一段时间后自动清除。当缓存数据的相关性随着时间的推移可预见地减弱时，这种策略是一个不错的选择。

假设有一个提供实时更新的新闻聚合应用。缓存的键和值可能只与热点新闻相关。实施 TTL 策略，使缓存条目在例如一天后过期，这样就能确保快速生成有关最新动态的类似查询响应，而旧新闻不会占满内存。

最后，三种流行的缓存失效策略中最复杂的是基于上下文相关性的策略。在这种情况下，一旦缓存内容与当前上下文或用户交互无关，我们就会立即将其清除。当应用在同一会话中处理不同的任务或主题，而之前的上下文对新的上下文没有价值时，这种策略就非常理想。

试想一下作为集成开发环境插件工作的编码助手。当用户正在处理一组特定文件时，缓存应该保留。但是，一旦他们切换到不同的代码库✦，之前的键和值就会变得无关紧要，因此可以删除它们以释放内存。不过，基于上下文相关性的方法在实施上可能具有挑战性，因为它们需要精确定位发生上下文切换✦的事件或时间点。

缓存重用

缓存管理的另一个重要方面是其重复使用。在某些情况下，一次生成的缓存可以再次使用，以加快生成速度，并避免在不同用户的缓存实例中多次存储相同的数据，从而节省内存。

缓存重用机会通常出现在共享上下文和/或需要热启动时。

在多个请求共享一个共同上下文的情况下，可以重复使用该共享部分的缓存。在电商平台中，某些产品可能有标准描述或规格，经常被多个客户询问。其中可能包括产品详细信息（"55 英寸 4K 超高清智能 LED 电视机"）、保修信息（"附带 2 年制造商保修服务，包括零件和人工。"）或客户说明（"为获得最佳效果，请使用兼容的壁挂支架安装电视机，此支架单独出售。"）。通过缓存这些共享产品描述的键值对，客户支持聊天机器人可以更快地生成对常见问题的回复。

同样，我们也可以预先计算和缓存常用提示或查询的初始 K-V 对。假设声控虚拟助手应用，用户经常以 "今天天气如何？"或 "设置一个 10 分钟的提醒 "这样的短语开始交互。通过预先计算和缓存这些常用查询的键值对，助理可以更快地做出响应。

总结

键-值（K-V）缓存是transformer模型中的一种技术。在这种技术中，先前步骤的键和值矩阵被存储起来，并在生成后续token时重复使用。它可以减少冗余计算，加快推理时间。这种加速是以增加内存消耗为代价的。当内存成为瓶颈时，我们可以通过简化模型来回收部分内存，从而牺牲模型的准确性。在大规模生产系统中实施 K-V 缓存需要谨慎的缓存管理，包括选择缓存失效策略和探索缓存重用的机会。