Transformer 之多头注意力

鱼先生
公众号【Afunby的 AI Lab】

已关注

50 人赞同了该文章

写在前边：

学习 Transformer 的过程中，找到了Ketan Doshi 博客中关于Transformer系统的介绍文章，感觉非常棒，于是进行了翻译。**该系列一共有4篇文章。本篇文章为该系列的第3篇。**

原文链接在文末。翻译主要采用 “DeepL+人工”的方式进行，并加入了一些自己的理解。

第一篇：Transformer 之整体介绍

第二篇：Transformer 之逐层介绍

第四篇：Transformer 之注意力计算原理

Transformers Explained Visually (Part 3): Multi-head Attention, deep dive

---

一、注意力超参数（Attention Hyperparameters）

Transformer 的数据维度由三个超参数决定。

1. Embedding Size：嵌入大小。嵌入向量的大小（例子中使用的嵌入向量大小为6）。这个维度在整个 Transformer 模型中都是向前传递的，因此有时也被称为 " model size - 模型大小 "等其他名称。
2. Query Size (与 Key 和 Value size 相等)：查询大小（等于键和值大小）。分别用来产生Query、Key 和 Value矩阵的三个线性层的权重大小（例子中使用的查询大小为3）
3. number of Attention heads：注意力头个数。例子中使用两个注意力头。
4. batch_size：批量大小。

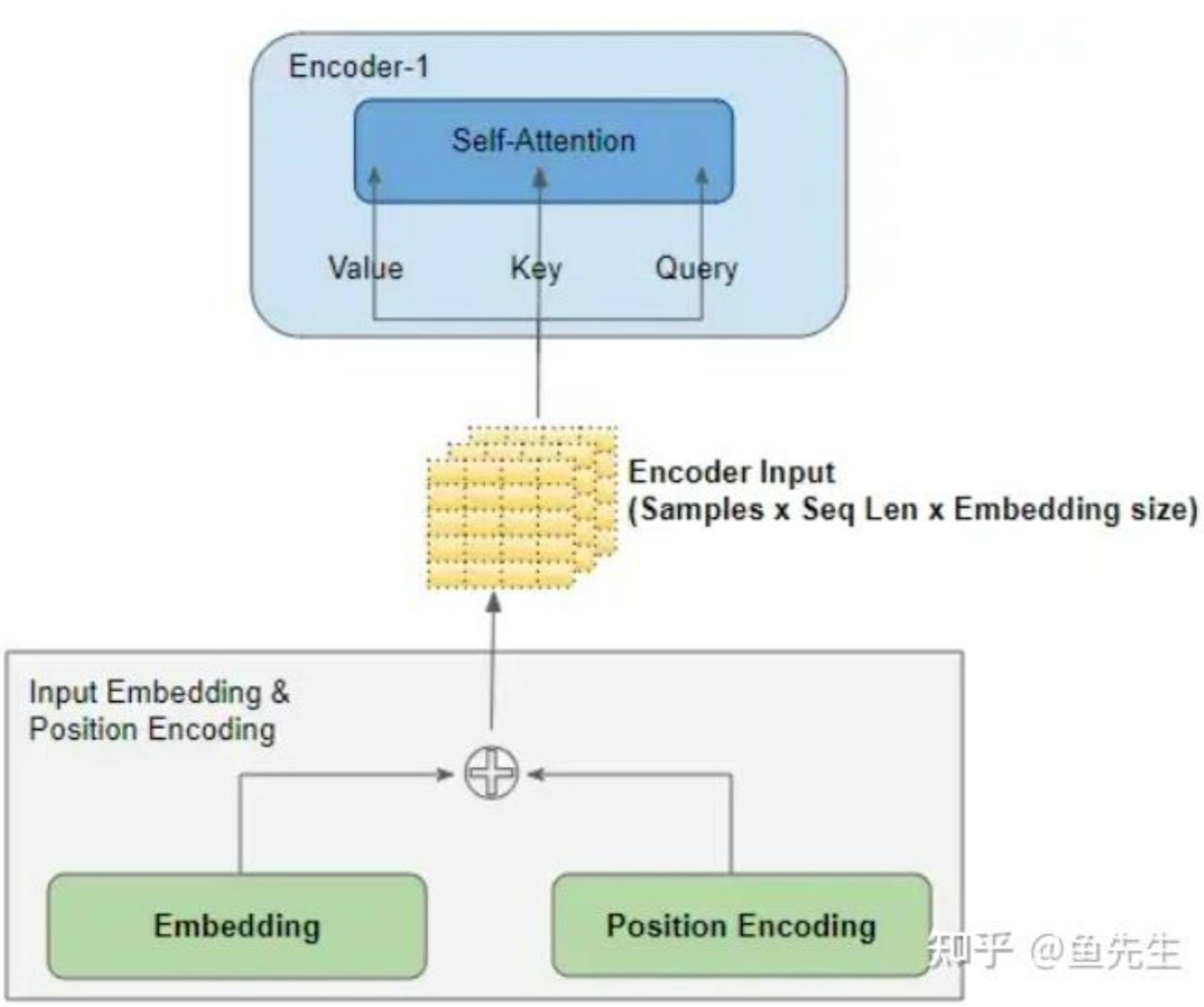
二、数据维度的变化

1. Input Layer

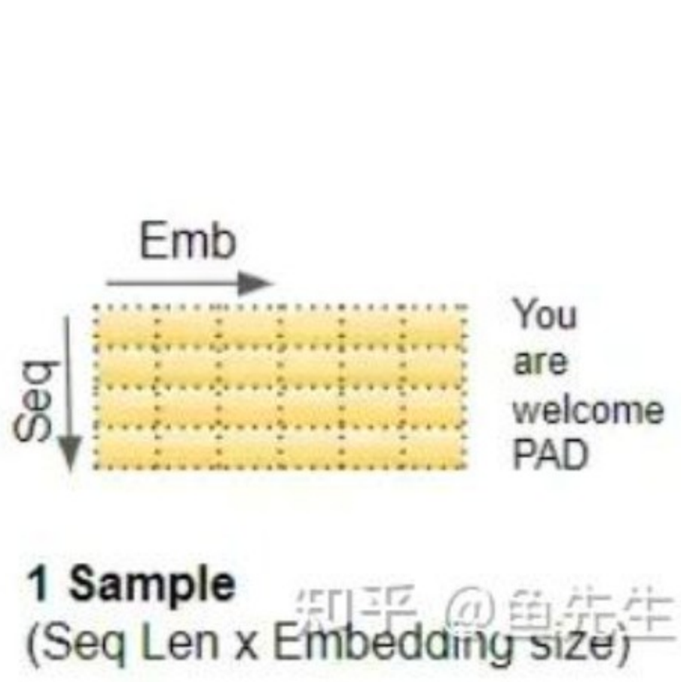
经过词嵌入和位置编码后，进入编码器之前，输入的数据维度为：

（batch_size, seq_length, embedding_size)

之后数据进入编码器堆栈中的第一个 Encoder，与 Query、Key、Value 矩阵相乘。

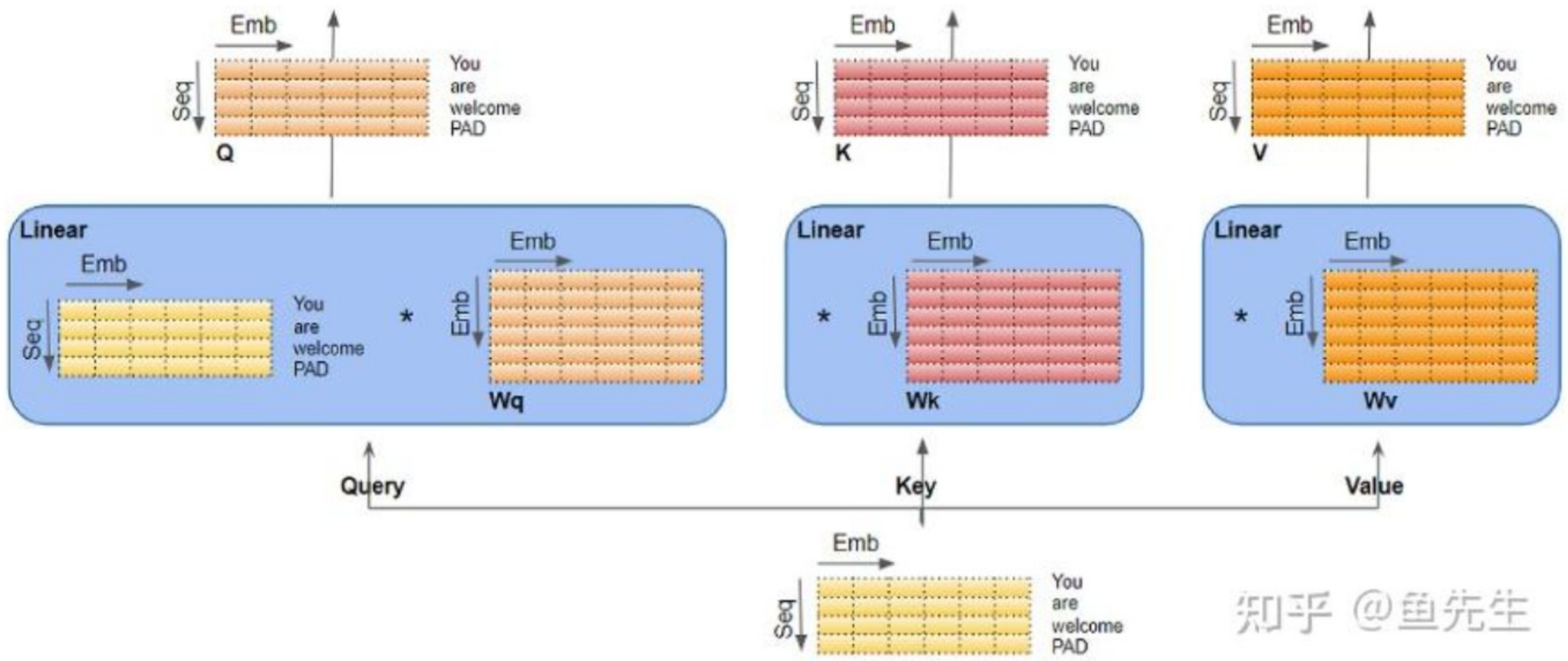


为方便理解，以下的图示与介绍中将去掉 batch_size 维度，聚焦于剩下的维度：



2. Linear Layers for Query, Key, and Value

Query, Key, Value 实际上是三个独立的线性层。每个线性层都有自己独立的权重。输入数据与三个线性层分别相乘，产生 Q、K、V。



3. 通过注意力头切分数据

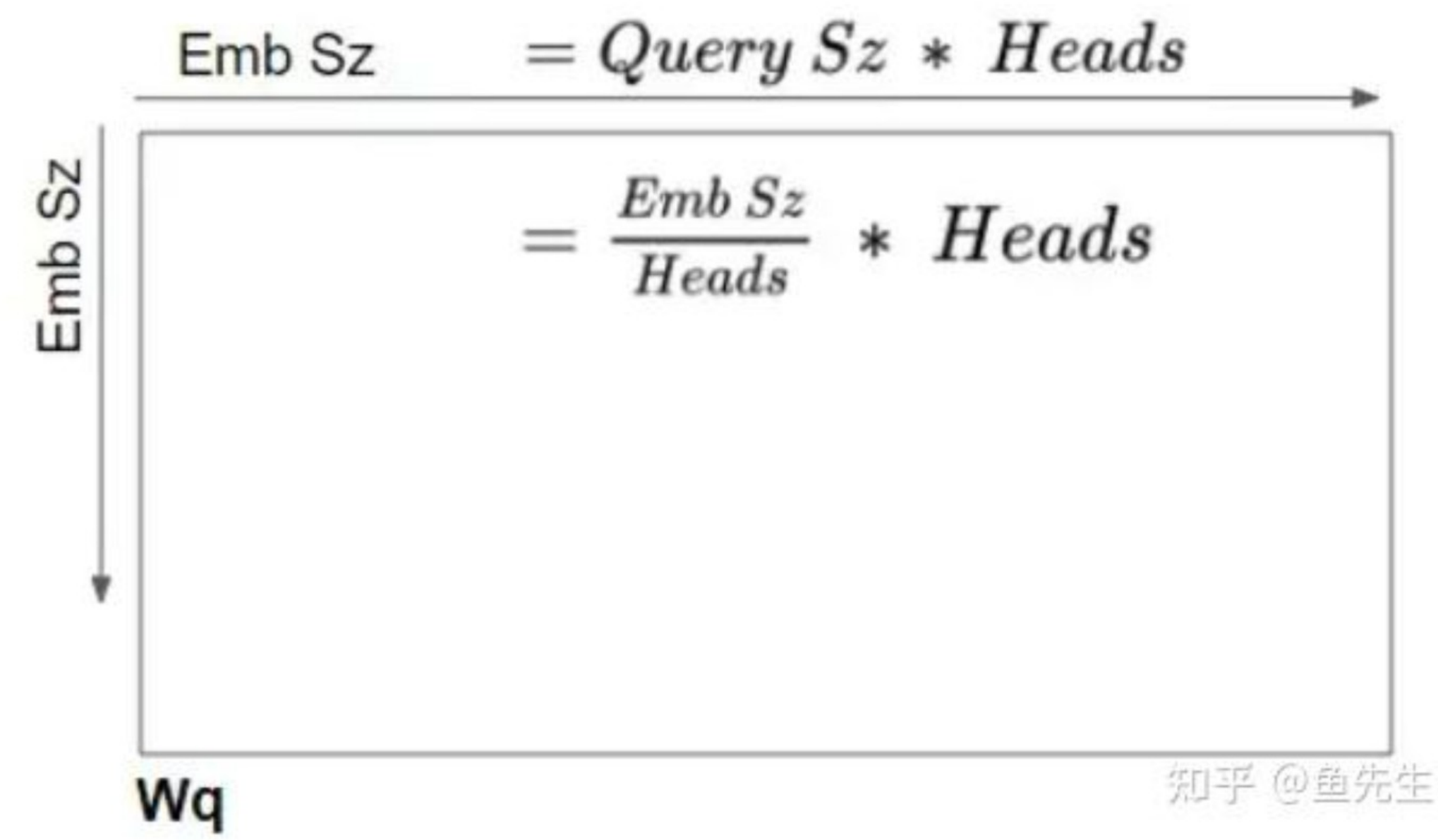
现在，数据被分割到多个注意头中，以便每个注意头能够独立地处理它。

需要注意的是，“切分”只是逻辑上的切分。对于参数矩阵 Query, Key, Value 而言，并没有物理切分成对应于每个注意力头的独立矩阵，仅逻辑上每个注意力头对应于 Query, Key, Value 的独立一部分。同样，各注意力头没有单独的线性层，而是所有的注意力头共用线性层，只是不同的注意力头在独属于其的逻辑部分上进行操作。

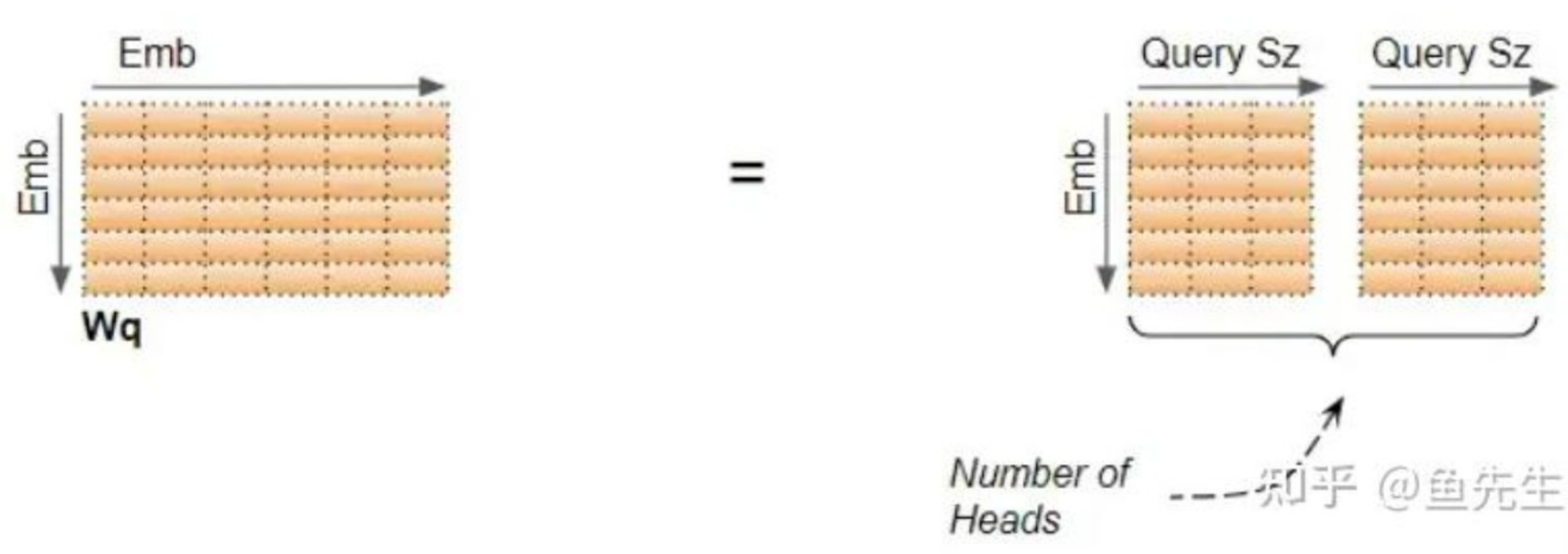
1. 线性层权重矩阵的切分

这种逻辑分割，是通过将输入数据以及线性层权重，均匀划分到各注意头中来完成的。我们可以通过选择下面的 Query Size大小来实现：

$$Query\ Size = Embedding\ Size / Number\ of\ heads$$



在我们的例子中，这就是为什么 Query Size=6/2=3。尽管层权重（和输入数据）均为单一矩阵，我们可以认为它是**“将每个头的独立层权重 ‘堆叠’在一起"成为一个矩阵。**




```
self.W_Q = nn.Linear(embedding_size, Query_size * n_heads, bias=False)

self.W_K = nn.Linear(embedding_size, Query_size * n_heads, bias=False)

self.W_V = nn.Linear(embedding_size, Value_size * n_heads, bias=False)
```

基于此，所有 Heads 的计算可通过对一个的矩阵操作来实现，而不需要 N 个单独操作。这使得计算更加有效，同时保持模型的简单：所需线性层更少，同时获得了多头注意力的效果。

回顾前一小节的内容，input 的维度是：

（batch_size, seq_length, embedding_size)

线性层的维度是：

（batch_size, embedding_size, Query_size * n_heads）

embedding_size = Query_size * n_heads，实际上线性层的维度并未进行变化变化。

得到的 Q、K 和 V 矩阵形状是：

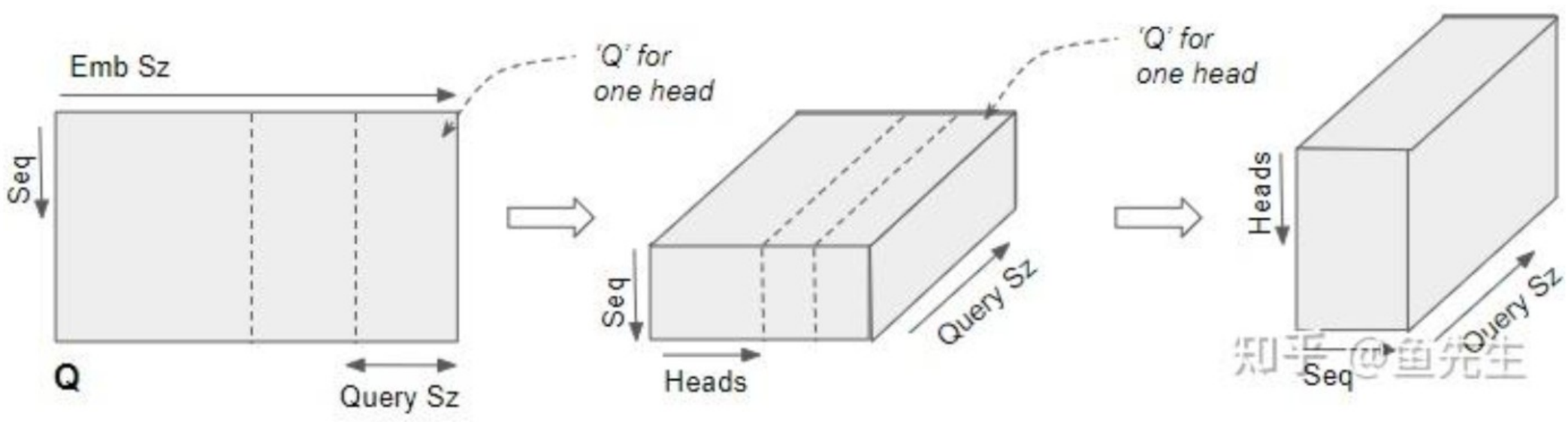
（batch_size, seq_length, Query_size * n_heads）

2. 改变 Q、K 和 V 矩阵形状

经由线性层输出的 Q、K 和 V 矩阵要经过 Reshape 操作，以实现一个明显的Head 维度。现在每个 "切片 "对应于每个头的一个矩阵。是得到 Q、K、V之后，在用 Q、K 计算 attention score之前才划分的多头。

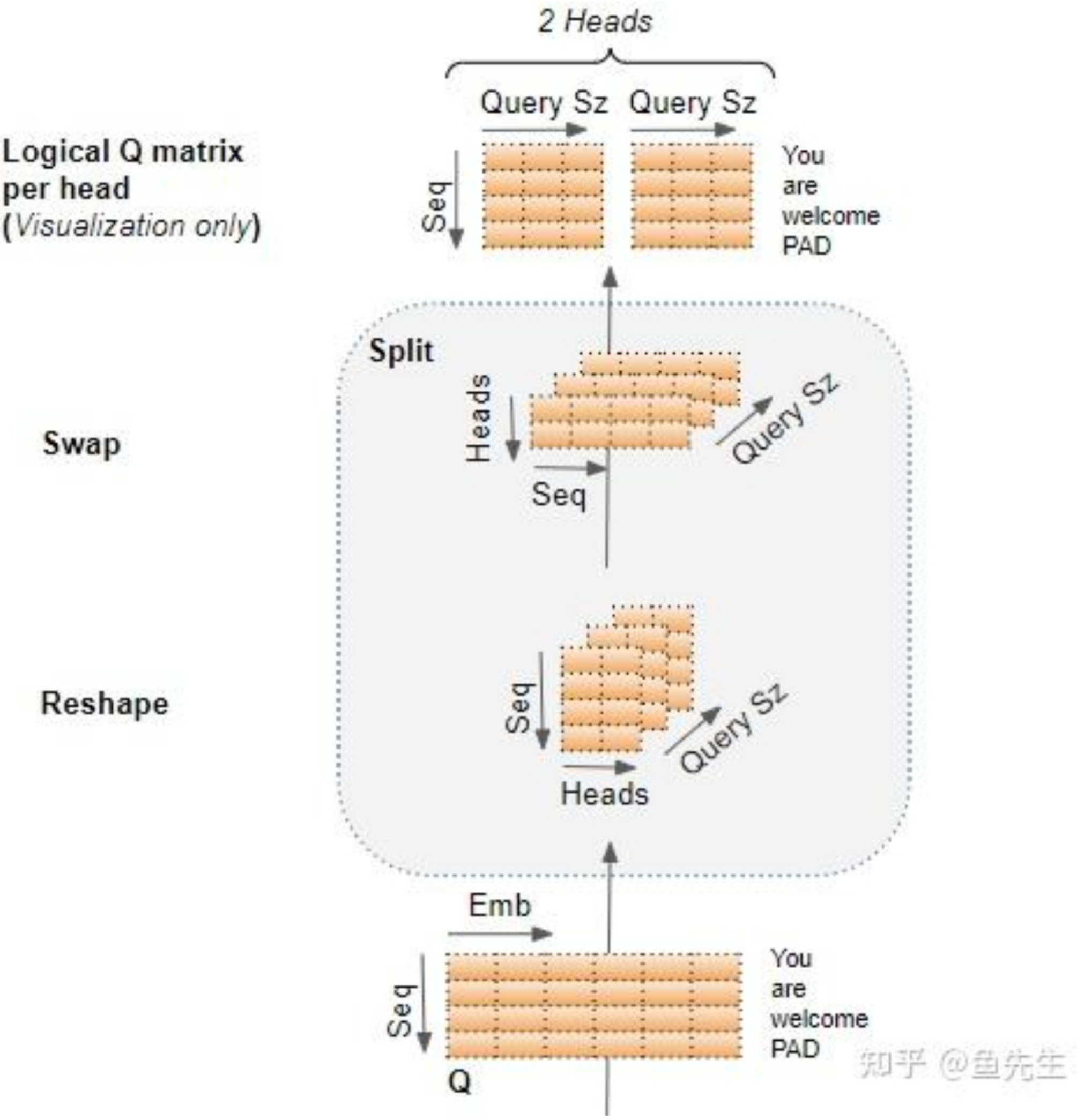
通过交换 n_heads 和 seq_length 这两个维度改变 Q、K 和 V 矩阵的形状。图示中虽然未表达出 Batch 维度，但对应于每一个注意力头的 'Q' 的维度是:

（batch_size, n_heads, seq_length, Query size）



在上图中，我们可以看到从线性层出来后，分割 Q 矩阵的完整过程。

最后一个阶段只是为了形象化--实际上 Q 矩阵仍然是一个单一矩阵，但可以把它看作是每个注意力头的逻辑上独立的 Q 矩阵。

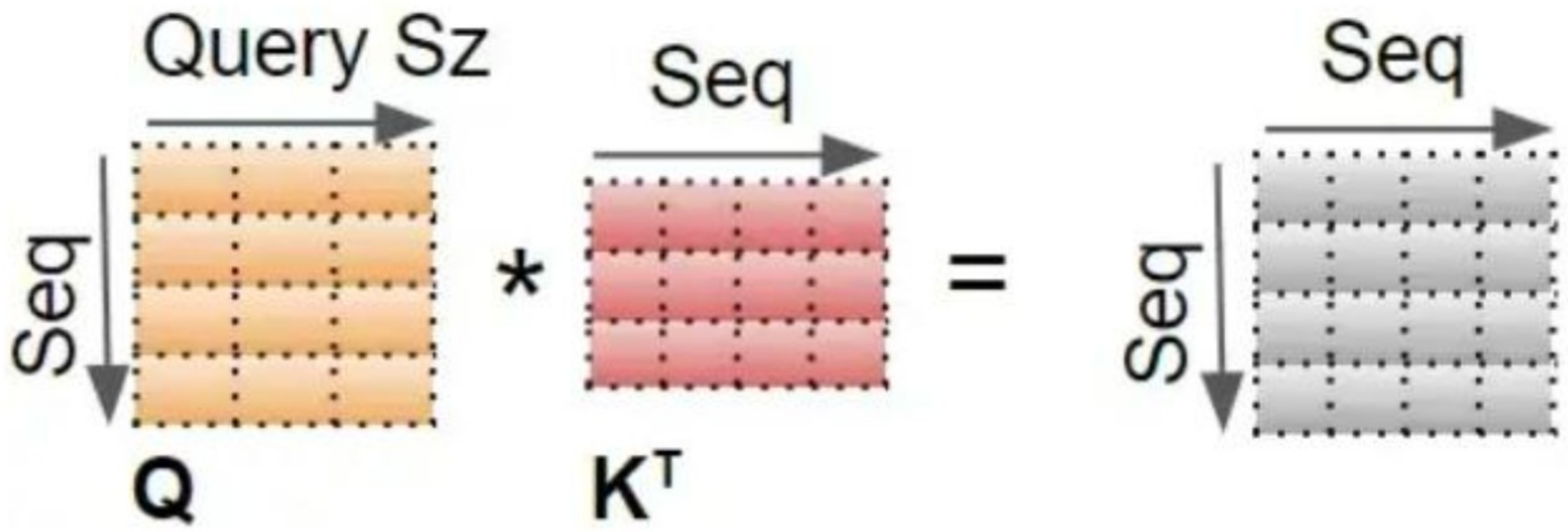


三、为每个头计算注意力分数（Compute the Attention Score for each head）

现在有分属各头的 Q、K、V 三个矩阵，每个头的 Q、K 用来计算注意力分数。

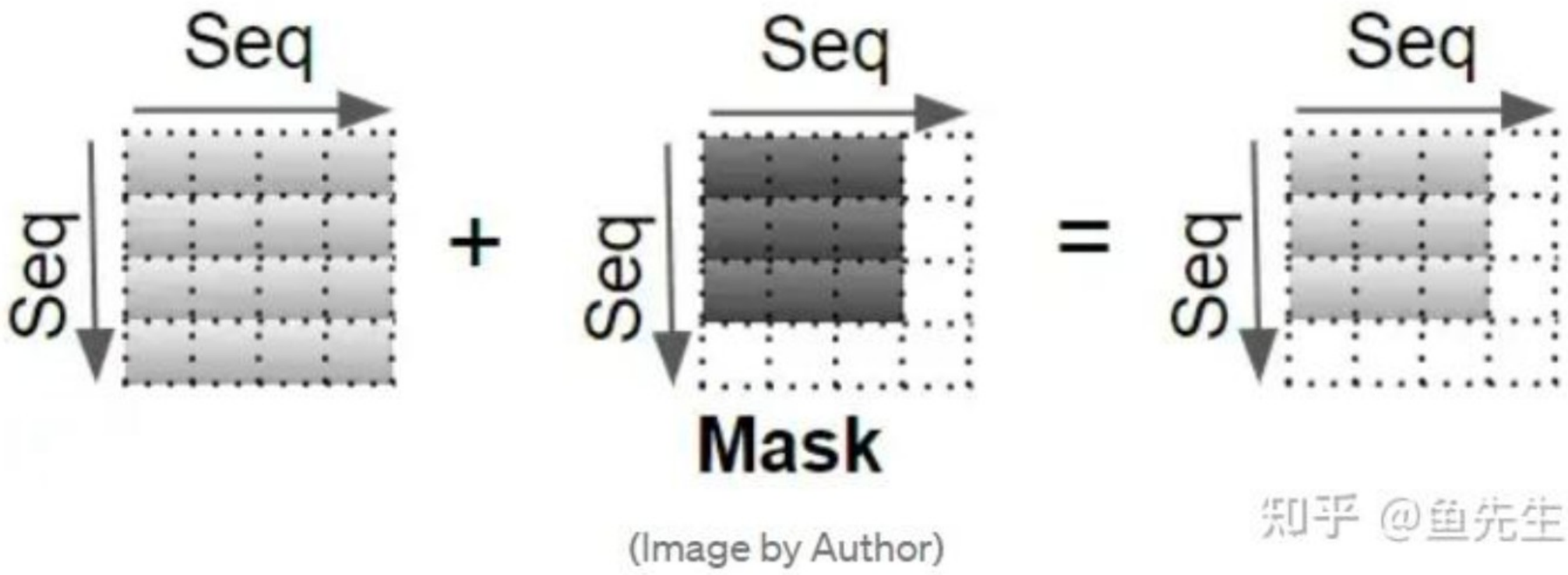
为方便理解，将只展示单头计算。使用最后两个维度（seq_length, Query size），跳过前两个维度（batch_size, n_heads）。从本质上讲，我们可以想象，计算对于每个头和每个批次中的每个样本都是 "循环"进行的。（虽然实际上它们是作为一个单一矩阵操作进行的，而非作为一个循环）。

1. 第一步是在 Q 和 K 的转置矩阵做一个矩阵乘法。



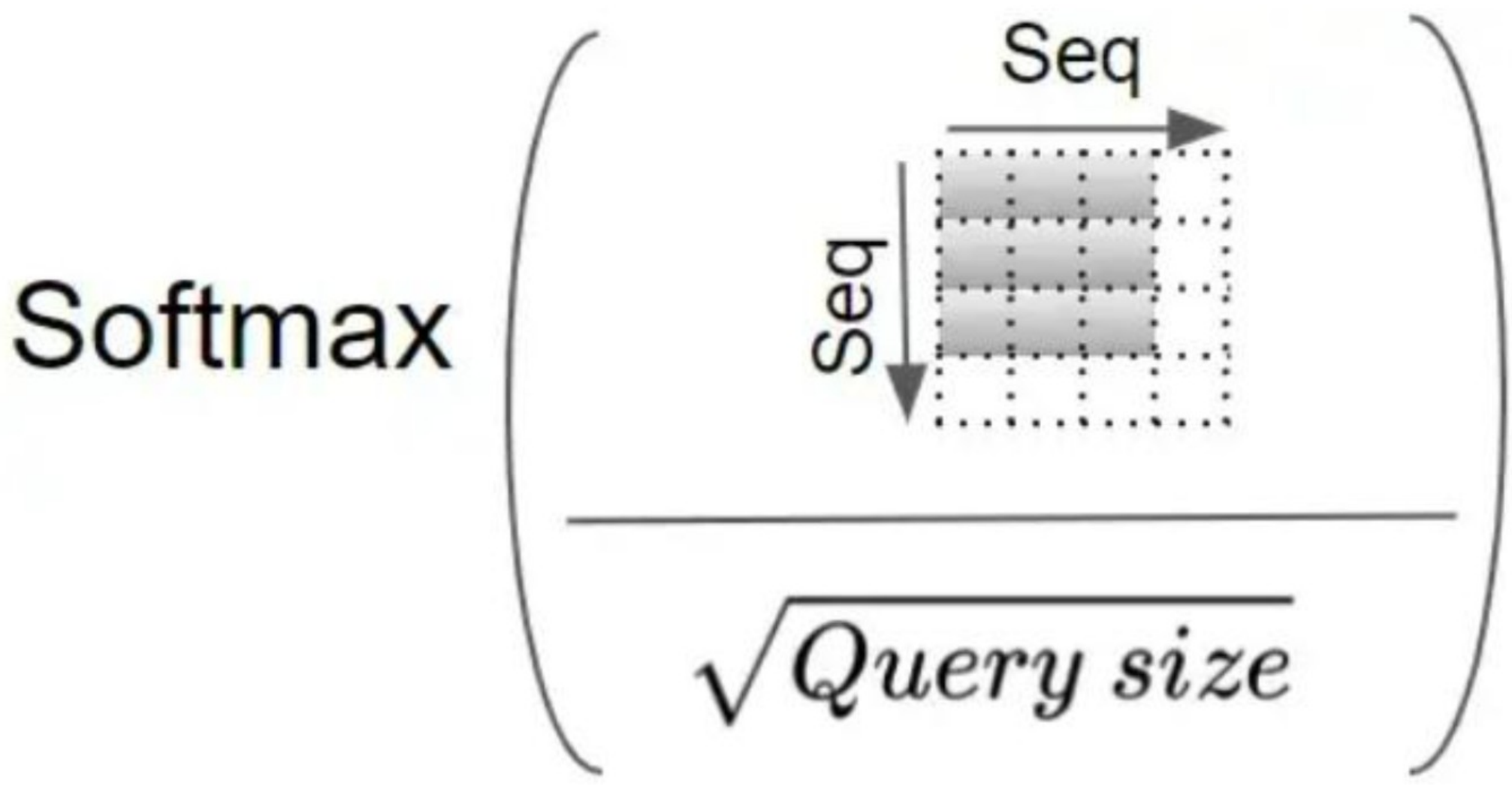
(Image by Author)

1. 将屏蔽值被添加到结果中。在 Encoder Self-attention 中，掩码用于掩盖填充值，这样它们就不会参与到注意分数中。



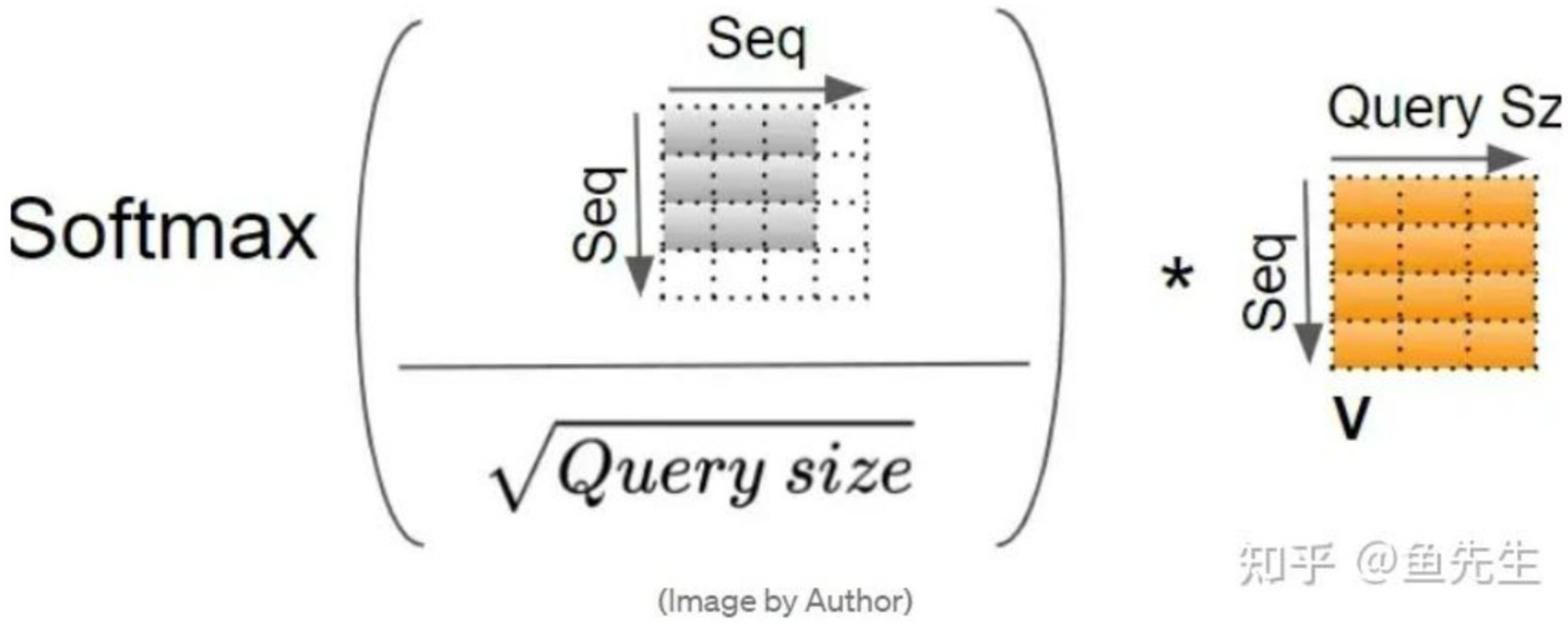
(Image by Author)

1. 上一步的结果通过除以 Query size 的平方根进行缩放，然后对其应用Softmax。



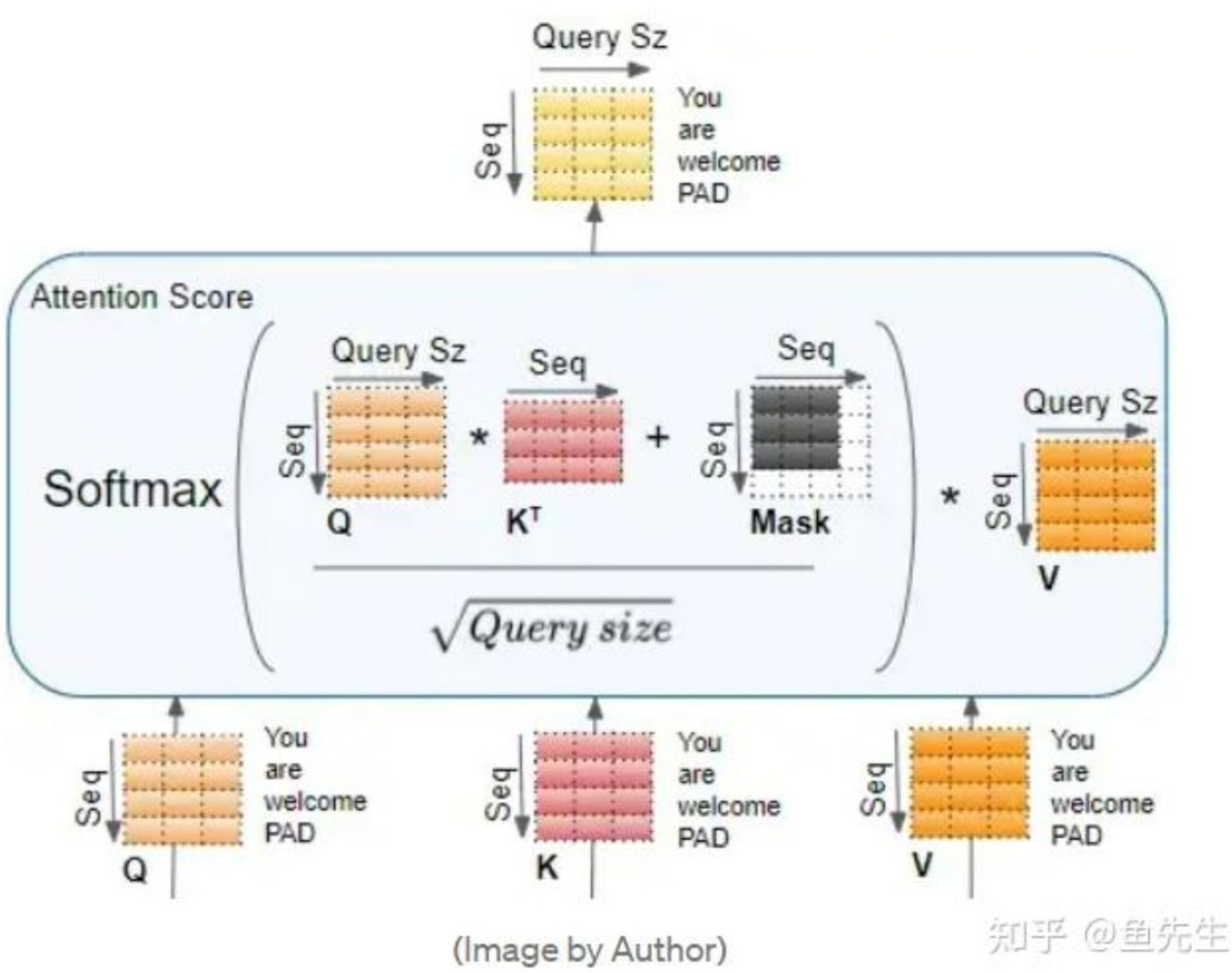
(Image by Author)

1. 在 Softmax 的输出和 V 矩阵之间进行另一个矩阵乘法。



(Image by Author)

在 Encoder Self-attention 中，一个注意力头的完整注意力计算如下图所示：



(Image by Author)

而每个注意力头的输出形状为：

（batch_size，n_heads，seq_length，Query size）

注意：

实际上此处最后一个维度的大小为 value_size，只是在 Transformer 中的 value_size=key_size=query_size

四、融合每个头的注意力分数（Merge each Head's Attention Scores together）

我们现在对每个头都有单独的注意力分数，需要将其合并为一个分数。这个合并操作本质上是与分割操作相反，通过重塑结果矩阵以消除 **n_heads** 维度来完成的。其步骤如下：

1. 交换头部和序列维度来重塑注意力分数矩阵。换句话说，矩阵的形状从

（batch_size，n_heads，seq_length，Query_size）

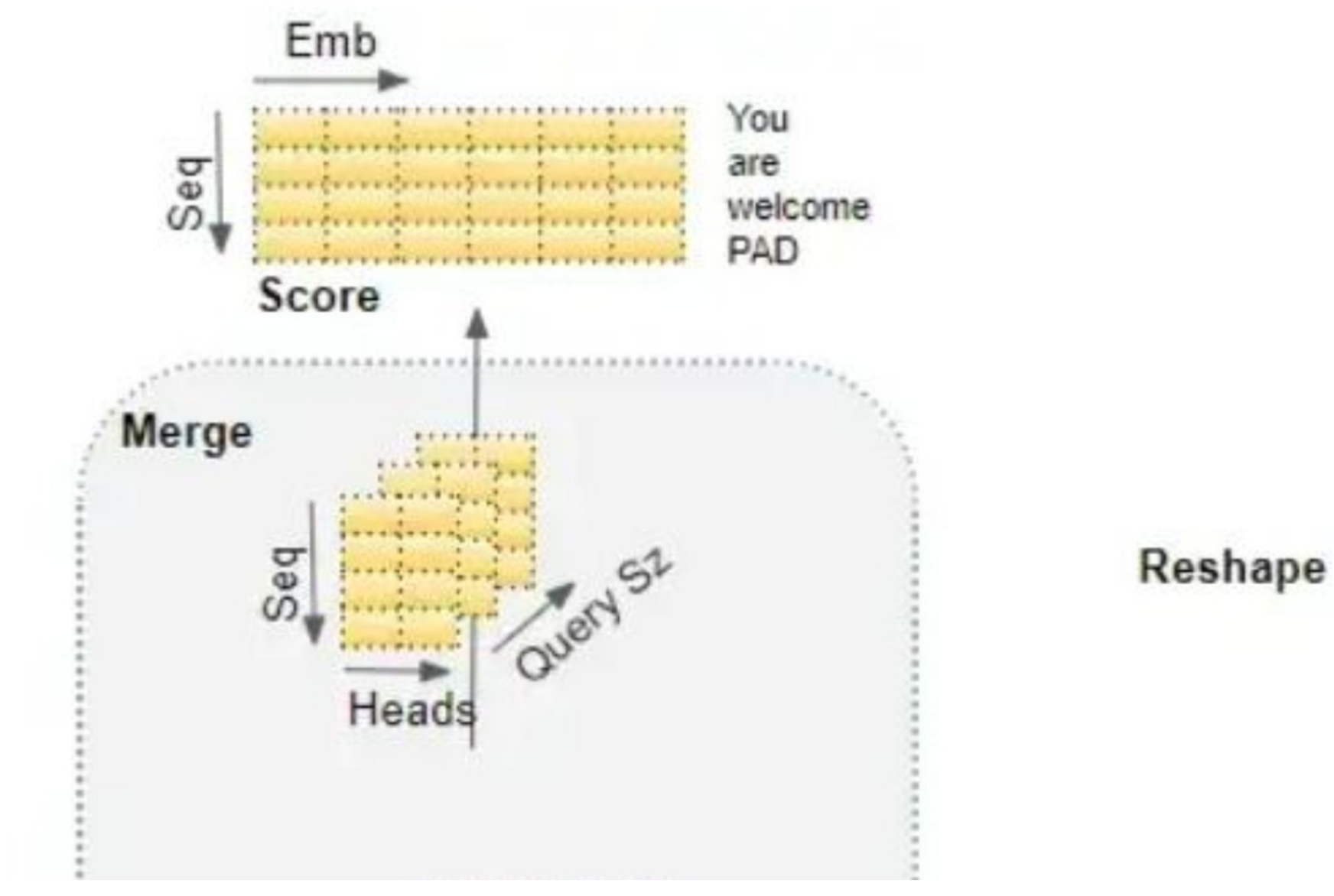
变成：

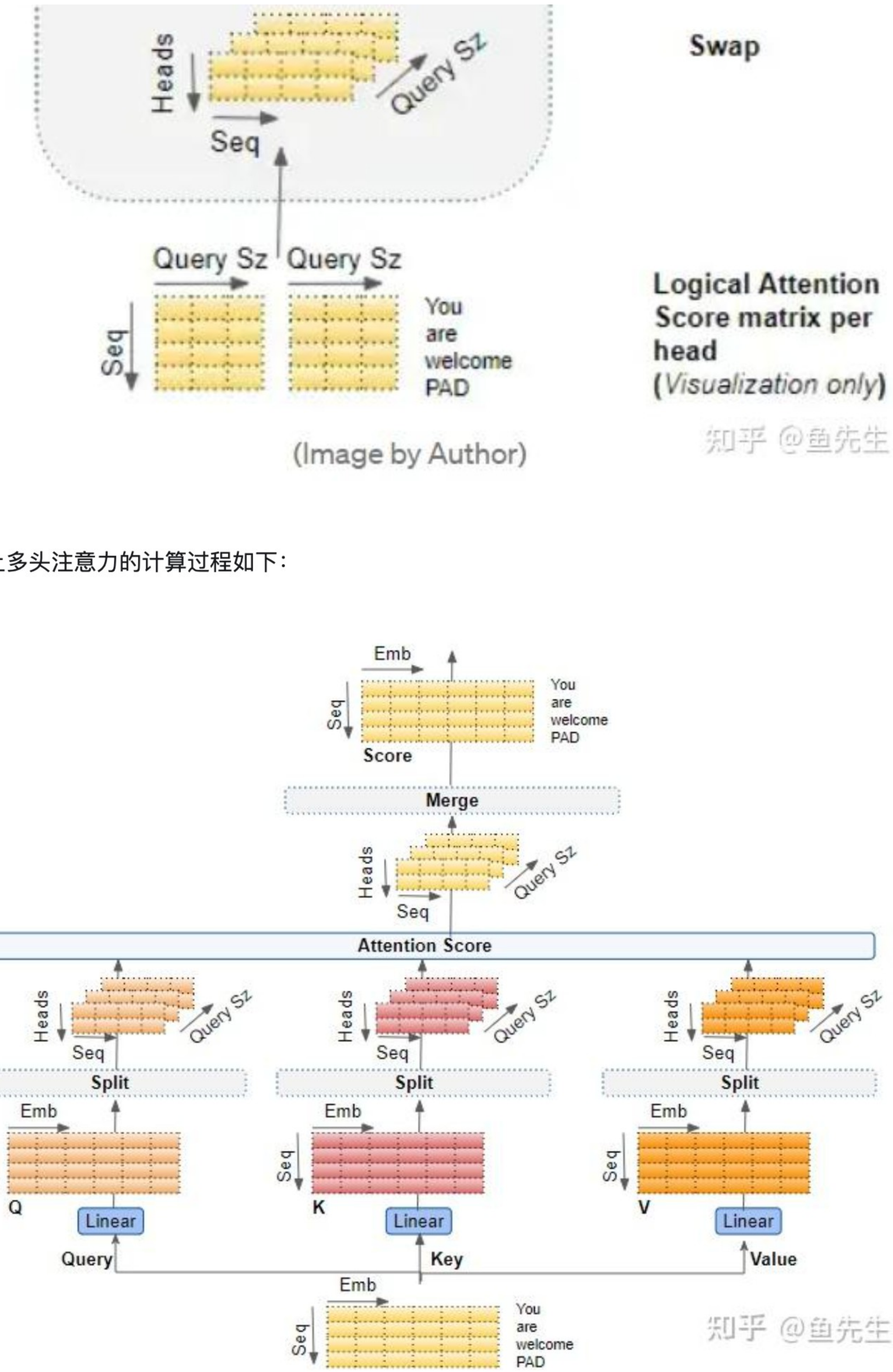
（batch_size，seq_length，n_heads，Query_size）

1. 通过重塑 **(Batch, Sequence，Head * Query size)**折叠头部维度。这就有效地将每个头的注意得分向量连接成一个合并的注意得分。

由于 embedding_size=n_heads * query_size ，合并后的分数是（**batch_size，seq_length，embedding_size**）。在下面的图片中，我们可以看到分数矩阵的完整合并过程:

(从代码的角度看，这一步是通过点乘一个参数矩阵的线性变换得到的)

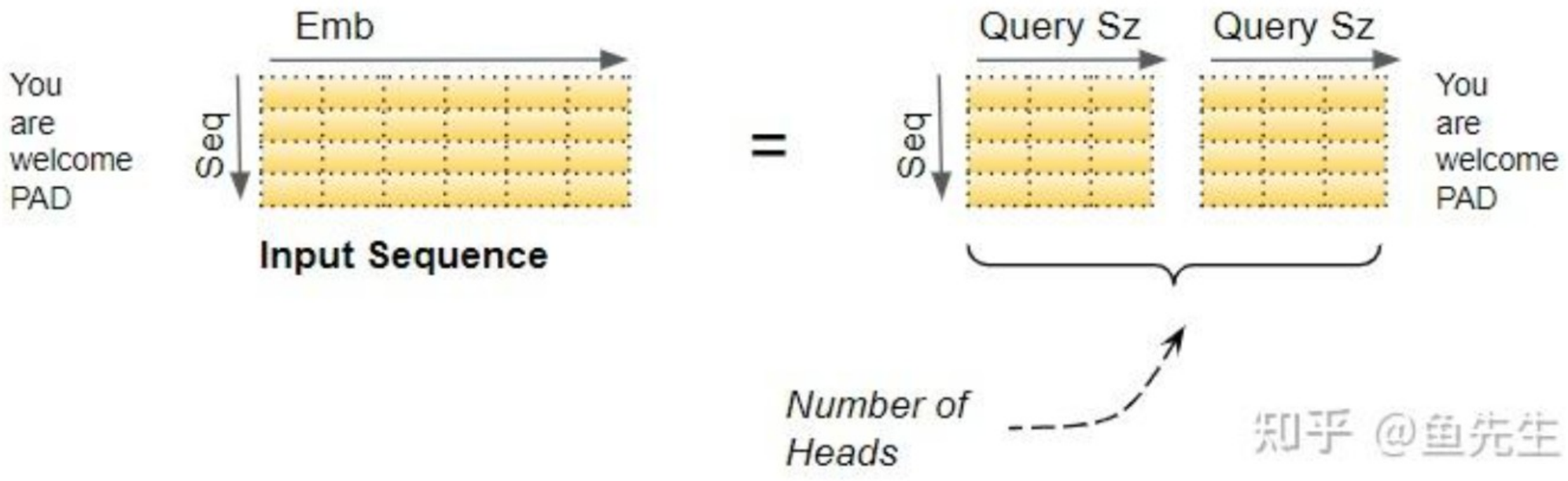




整体上多头注意力的计算过程如下：

五、多头分割可以捕捉到更丰富的表示（Multi-head split captures richer interpretations)

一个嵌入向量捕捉了一个词的含义。在 Multi-head Attention 的机制下，正如我们所看到的，输入（和目标）序列的**嵌入向量在逻辑上被分割到多个头。这意味着，嵌入向量的不同部分可以表征每个词在不同方面的含义，而每个词不同的含义与序列中的其他词有关。这使得 Transformer 能够捕捉到对序列更丰富的表示。**

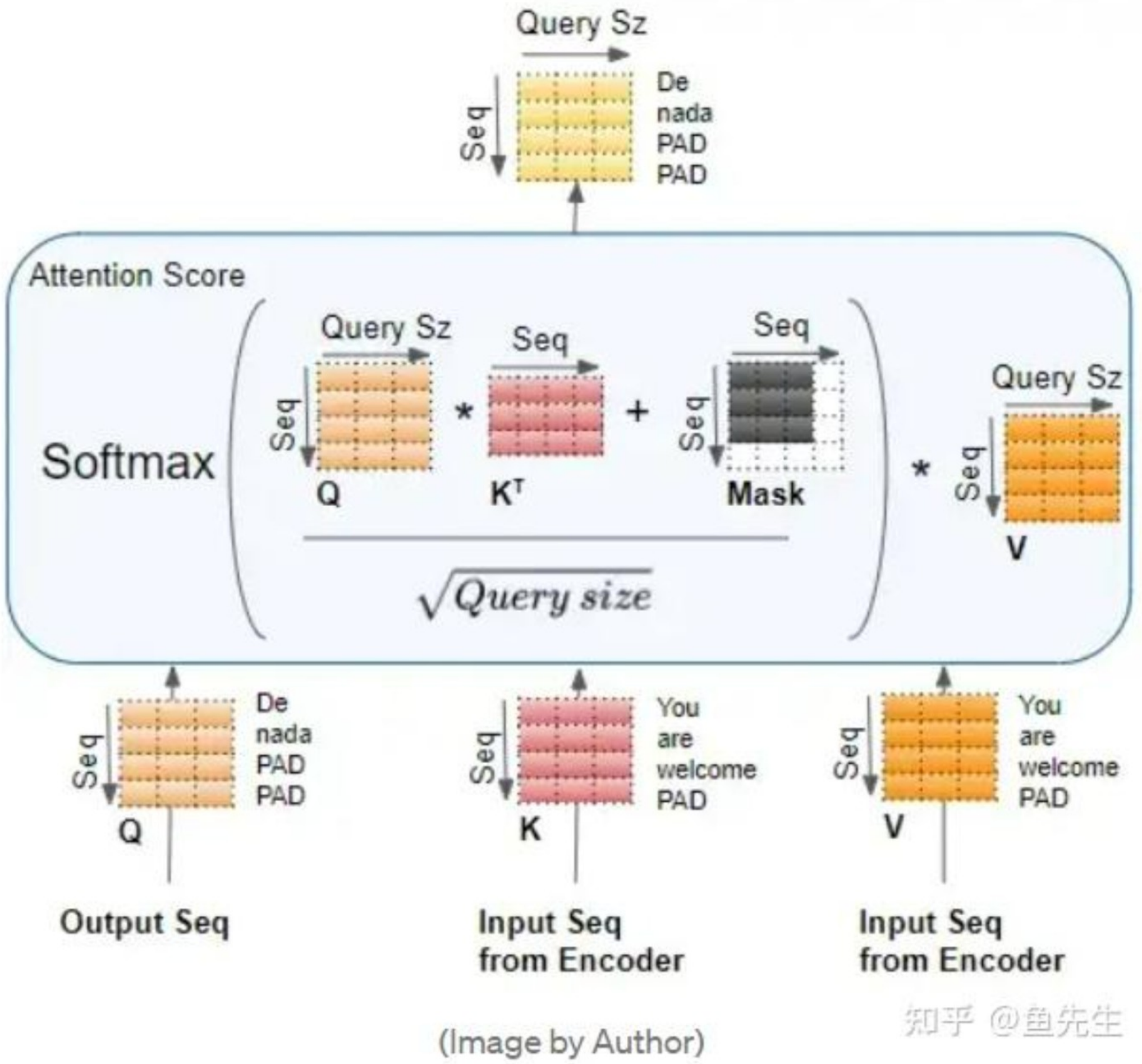


例如，嵌入向量的某一部分可以捕获一个名词的词性，而另一个部分可以捕获一个名词的单复数。这在翻译中很重要，许多语言中，动词的使用与这些因素有关。

以上虽然不是一个现实的例子，但它可能有助于建立直觉。

六、解码器中的“编码器-解码器注意力”与“掩码操作”(Decoder Encoder-Decoder Attention and Masking)

“编码器-解码器注意力”从两个来源获得输入。因此，与计算输入中每个词与其他词之间相互作用的 Encoder Self-Attention 不同；也与计算每个目标词与其他目标词之间相互作用的 Decoder-Self-Attention 不同，“编码器-解码器注意力”计算的是每个 input word 与每个 target word之间的相互作用。



(Image by Author)

因此，所产生的注意分数中的每一个单元都对应于一个 **Q**（ie. target sequence word）与所有其他 **K**（ie. input sequence）词和所有 **V**（ie. input sequence）词之间的相互作用。

七、Conclusion

希望这能让你对Transformer中的Attention模块的作用有一个很好的认识。当与我们在第二篇文章中提到的整个Transformer的端到端流程放在一起时，我们现在已经涵盖了整个Transformer架构的详细操作。

我们现在明白了Transformer到底是做什么的。但是我们还没有完全回答为什么 Transformer 的 Attention 要进行这样的计算。为什么它要使用查询、键和值的概念，为什么它要执行我们刚才看到的矩阵乘法?

我们有一个模糊的直观想法，即它 "抓住了每个词与其他词之间的关系"，但这到底是什么意思?这到底是如何让转化器的注意力有能力理解序列中每个词的细微差别的?

这是一个有趣的问题，也是本系列的最后一篇文章的主题。一旦我们了解了这一点，我们就会真正理解Transformer架构的优雅之处。