

Transformer

鲁老师 2023年5月4日 大约 19 分钟 深度学习 Transformer 注意力机制

Transformer[1]论文提出了一种自注意力机制（Self-Attention），Self-Attention的最核心的公式为：

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

这个公式中的 Q 、 K 和 V 分别代表Query、Key和Value，单看这个公式，其实并不能很好地理解Attention到底在做什么，本文从Transformer所使用的Self-Attention，介绍Attention背后的原理。

Self-Attention：从向量点乘说起

我们先从 $\text{Softmax}(\mathbf{X}\mathbf{X}^\top)\mathbf{X}$ 这样一个公式开始。

首先需要复习一下向量点乘（Dot Product）的概念。对于两个行向量 \mathbf{x} 和 \mathbf{y} ：

$$\begin{aligned}\mathbf{x} &= [x_0, x_1, \dots, x_n] \\ \mathbf{y} &= [y_0, y_1, \dots, y_n] \\ \mathbf{x} \cdot \mathbf{y} &= x_0y_0 + x_1y_1 + \dots + x_ny_n\end{aligned}$$

i 注

本文公式中变量加粗表示该变量为向量或矩阵

向量点乘的几何意义是：向量 \mathbf{x} 在向量 \mathbf{y} 方向上的投影再与向量 \mathbf{y} 的乘积，能够反应两个向量的相似度。向量点乘结果大，两个向量越相似。

一个矩阵 \mathbf{X} 由 n 行向量组成。比如，我们可以将某一行向量 \mathbf{x}_i 理解成一个词的词向量，共有 n 个行向量组成 $n \times n$ 的方形矩阵：

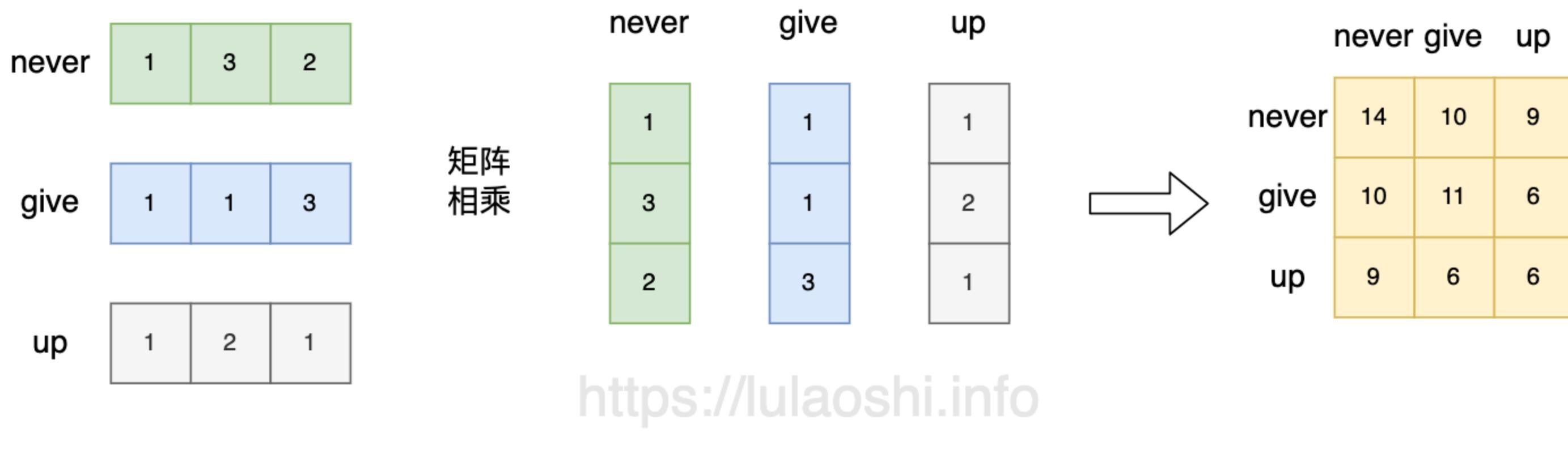
$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_n \end{bmatrix} \quad \mathbf{X}^\top = \begin{bmatrix} \mathbf{x}_0^\top & \mathbf{x}_1^\top & \cdots & \mathbf{x}_n^\top \end{bmatrix}$$

矩阵 \mathbf{X} 与矩阵的转置 \mathbf{X}^\top 相乘， \mathbf{X} 中的每一行与 \mathbf{X}^\top 的每一列相乘得到目标矩阵的一个元素， $\mathbf{X}\mathbf{X}^\top$ 可表示为：

$$\mathbf{X}\mathbf{X}^\top = \begin{bmatrix} \mathbf{x}_0 \cdot \mathbf{x}_0 & \mathbf{x}_0 \cdot \mathbf{x}_1 & \cdots & \mathbf{x}_0 \cdot \mathbf{x}_n \\ \mathbf{x}_1 \cdot \mathbf{x}_0 & \mathbf{x}_1 \cdot \mathbf{x}_1 & \cdots & \mathbf{x}_1 \cdot \mathbf{x}_n \\ \vdots & & & \\ \mathbf{x}_n \cdot \mathbf{x}_0 & \mathbf{x}_n \cdot \mathbf{x}_1 & \cdots & \mathbf{x}_n \cdot \mathbf{x}_n \end{bmatrix}$$

以 \mathbf{XX}^\top 中的第一行第一列元素为例，其实是向量 \mathbf{x}_0 与 \mathbf{x}_0 自身做点乘，其实就是 \mathbf{x}_0 自身与自身的相似度，那第一行第二列元素就是 \mathbf{x}_0 与 \mathbf{x}_1 之间的相似度。

下面以词向量矩阵为例，这个矩阵中，每行为一个词的词向量。矩阵与自身的转置相乘，生成了目标矩阵，目标矩阵其实就是一个词的词向量与各个词的词向量的相似度。



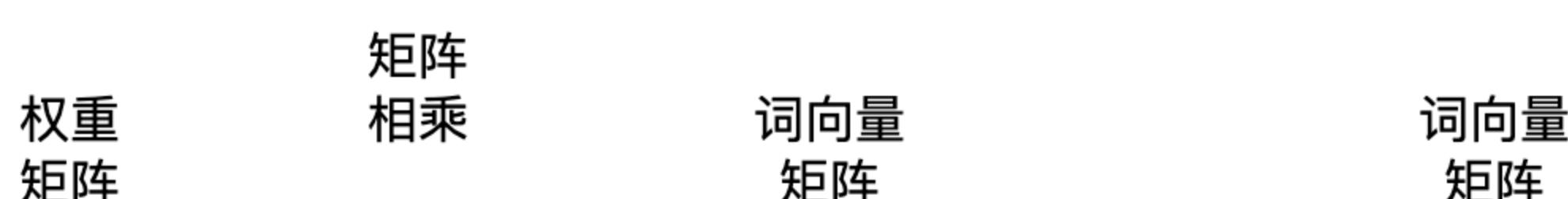
词向量矩阵相乘

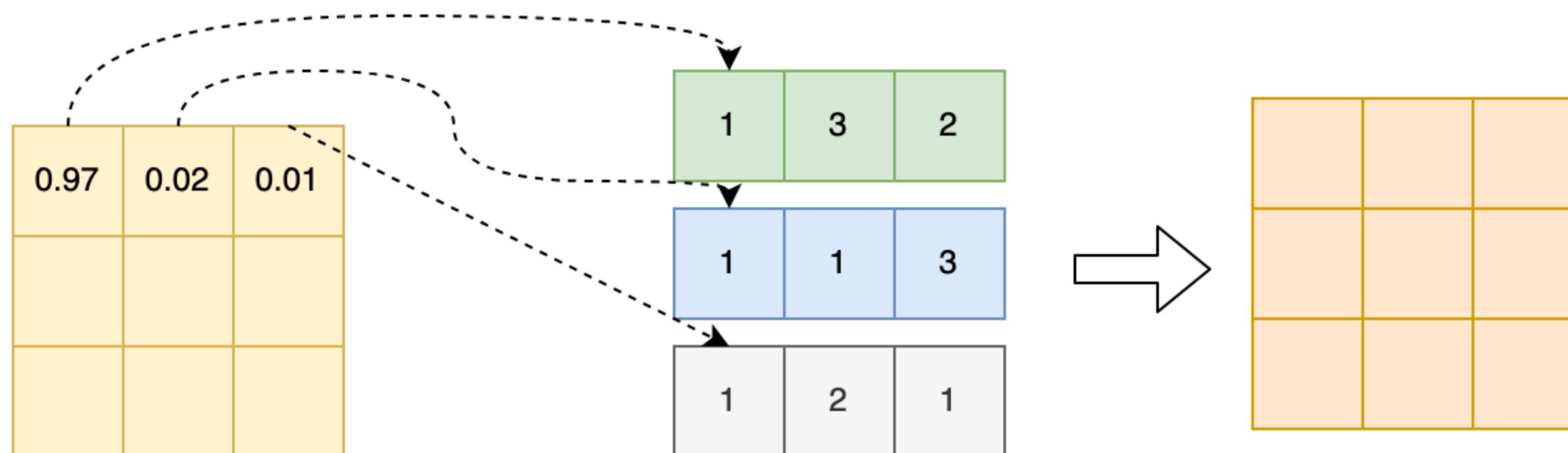
如果再加上Softmax呢？我们进行下面的计算： $\text{Softmax}(\mathbf{XX}^\top)$ 。Softmax的作用是对向量做归一化，那么就是对相似度的归一化，得到了一个归一化之后的权重矩阵，矩阵中，某个值的权重越大，表示相似度越高。



相似度矩阵的归一化

在这个基础上，再进一步： $\text{Softmax}(\mathbf{XX}^\top)\mathbf{X}$ ，将得到的归一化的权重矩阵与词向量矩阵相乘。权重矩阵中某一行分别与词向量的一列相乘，词向量矩阵的一列其实代表着不同词的某一维度。经过这样一个矩阵相乘，相当于一个加权求和的过程，得到结果词向量是经过加权求和之后的新表示，而权重矩阵是经过相似度和归一化计算得到的。





<https://lulaoshi.info>

通过与权重矩阵相乘，完成加权求和过程

上述过程用 PyTorch 实现：

```

1 import torch
2 import torch.nn as nn
3
4 x = torch.tensor([[1, 3, 2], [1, 1, 3], [1, 2, 1]], dtype=torch.float64)
5
6 attention_scores = torch.matmul(x, x.transpose(-1, -2))
7 attention_scores = nn.functional.softmax(attention_scores, dim=-1)
8
9 print(attention_scores)

```

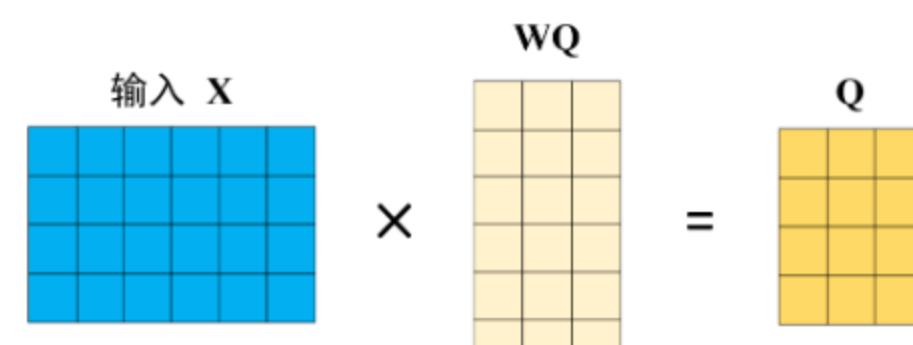
Self-Attention中的Q、K、V

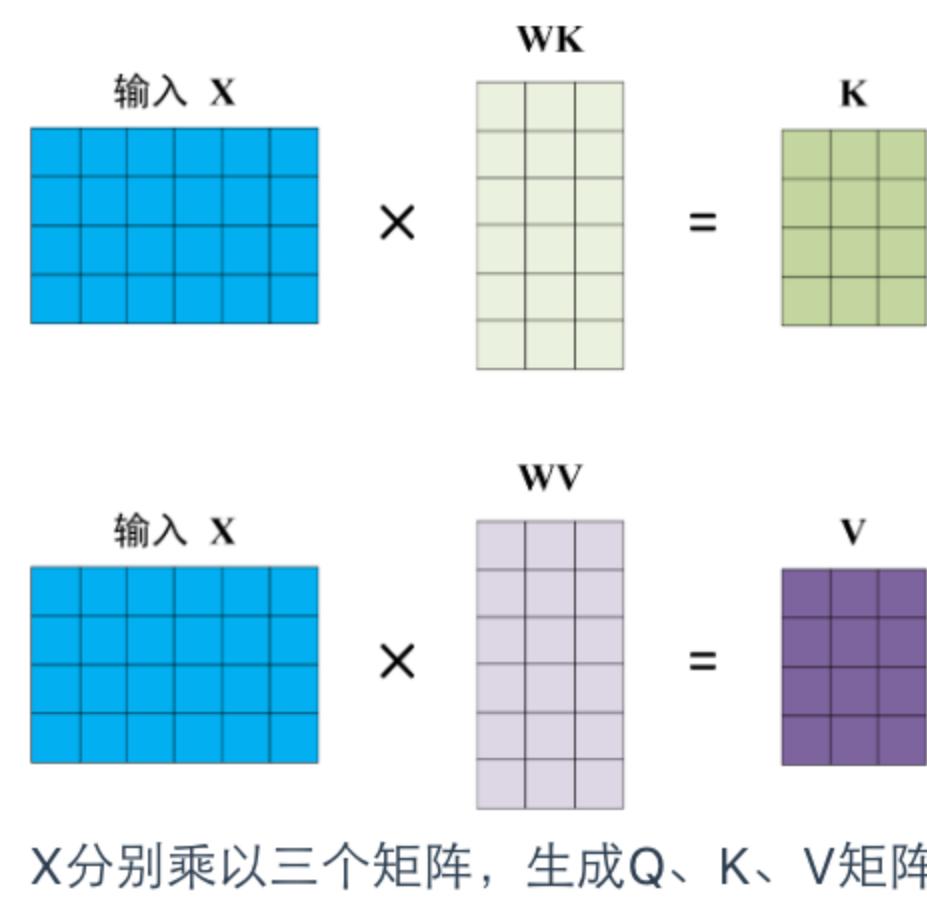
注意力Attention机制的最核心的公式为： $\text{Softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$ ，与我们刚才分析的 $\text{Softmax}(\mathbf{X}\mathbf{X}^\top)\mathbf{X}$ 有几分相似。Transformer论文中将这个Attention公式描述为：Scaled Dot-Product Attention。其中， Q 为Query、 K 为Key、 V 为Value。 Q 、 K 、 V 是从哪儿来的呢？在Transformer的Encoder中所使用的 Q 、 K 、 V 其实都是从同样的输入矩阵 X 线性变换而来的。

我们可以简单理解成：

$$\begin{aligned} Q &= XW^Q \\ K &= XW^K \\ V &= XW^V \end{aligned}$$

用图片演示为：

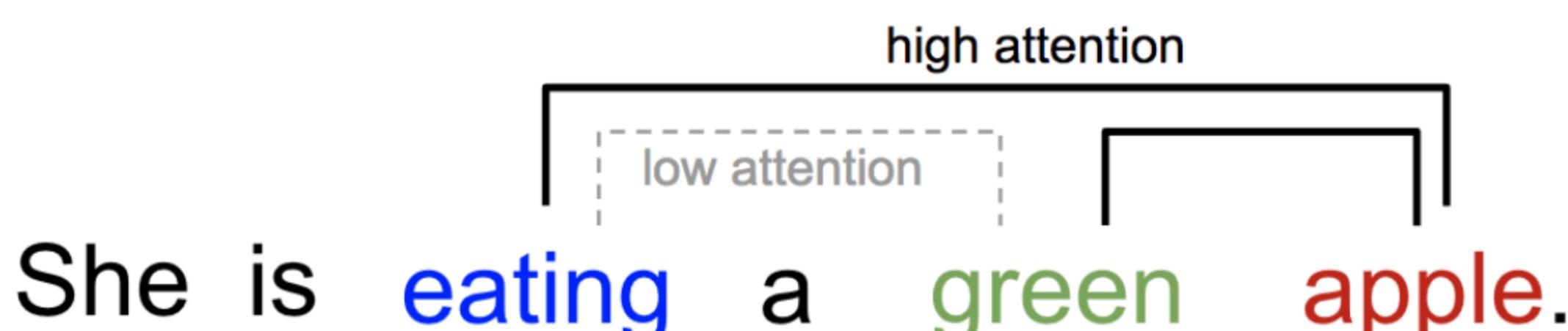




X分别乘以三个矩阵，生成Q、K、V矩阵

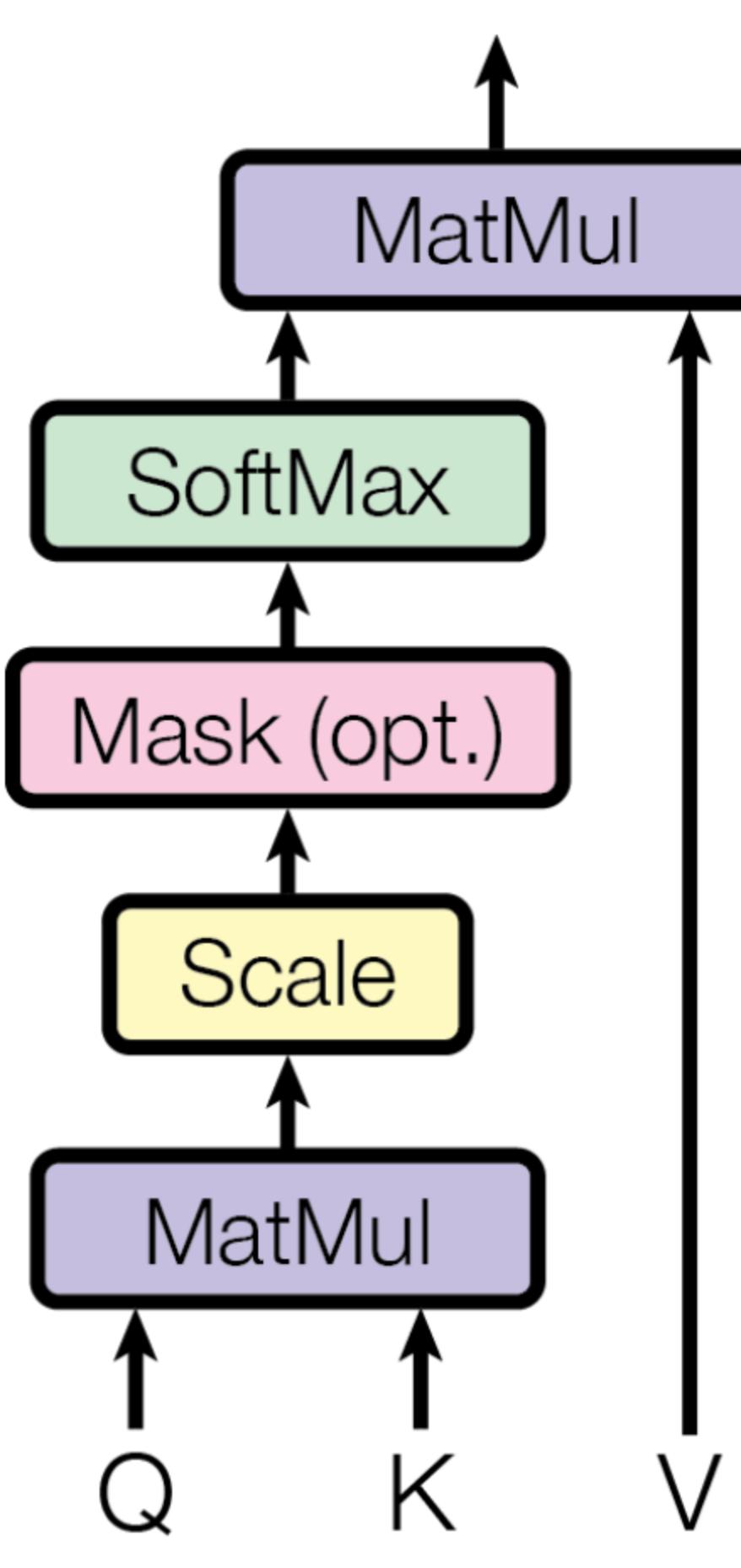
其中， W^Q ， W^K 和 W^V 是三个可训练的参数矩阵。输入矩阵 X 分别与 W^Q ， W^K 和 W^V 相乘，生成 Q 、 K 和 V ，相当于经历了一次线性变换。Attention不直接使用 X ，而是使用经过矩阵乘法生成的这三个矩阵，因为使用三个可训练的参数矩阵，可增强模型的拟合能力。

Self-Attention可以解释一个句子内不同词的相互关系。比如下面的句子中，“eating”后面会有一种食物，“eating”与“apple”的Attention Score更高，而“green”只是修饰食物“apple”的修饰词，和“eating”关系不大。



句子内不同词之间的Attention示意图

下面这张图是Transformer论文中描述Attention计算过程的示意图。

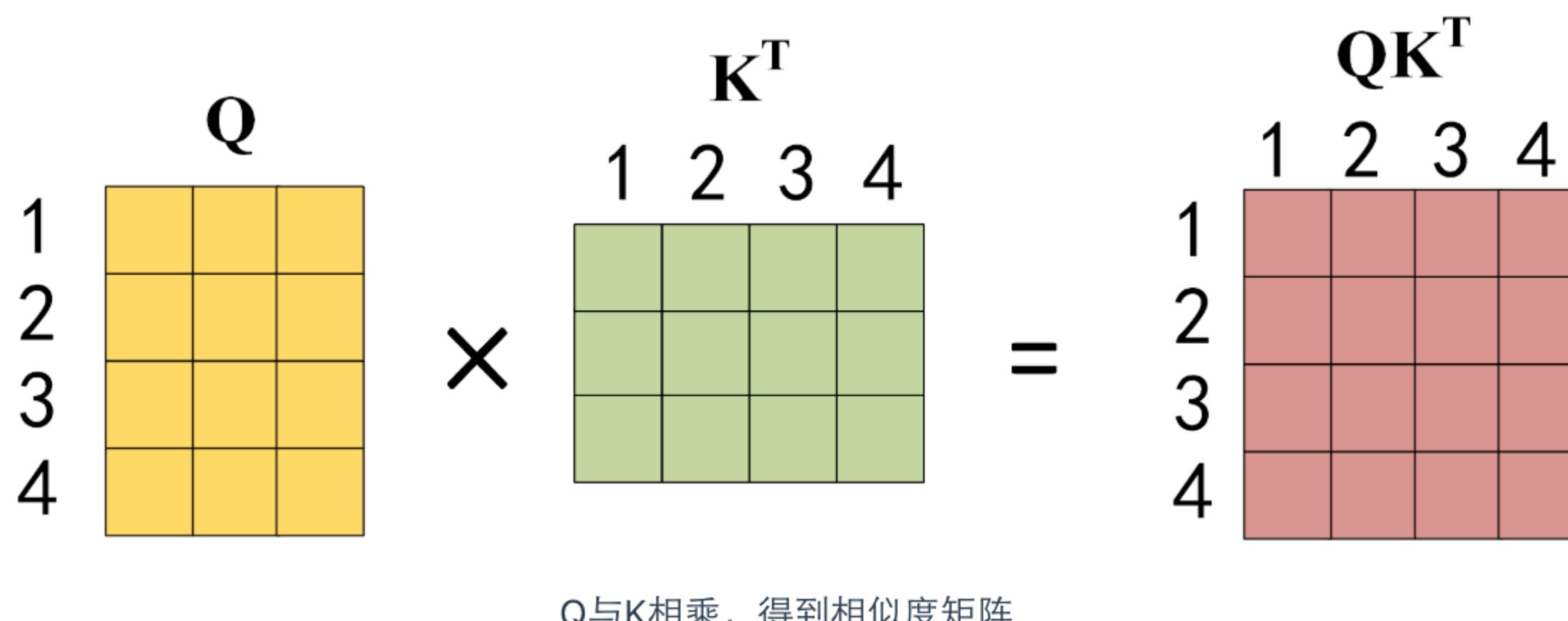


在这张图中， Q 与 K^T 经过MatMul，生成了相似度矩阵。对相似度矩阵每个元素除以 $\sqrt{d_k}$ ， d_k 为 K 的维度大小，得到了Attention Score。这个除法被称为Scale。当 d_k 很大时， QK^T 的乘法结果方差变大，进行Scale可以使方差变小，训练时梯度更新更稳定。

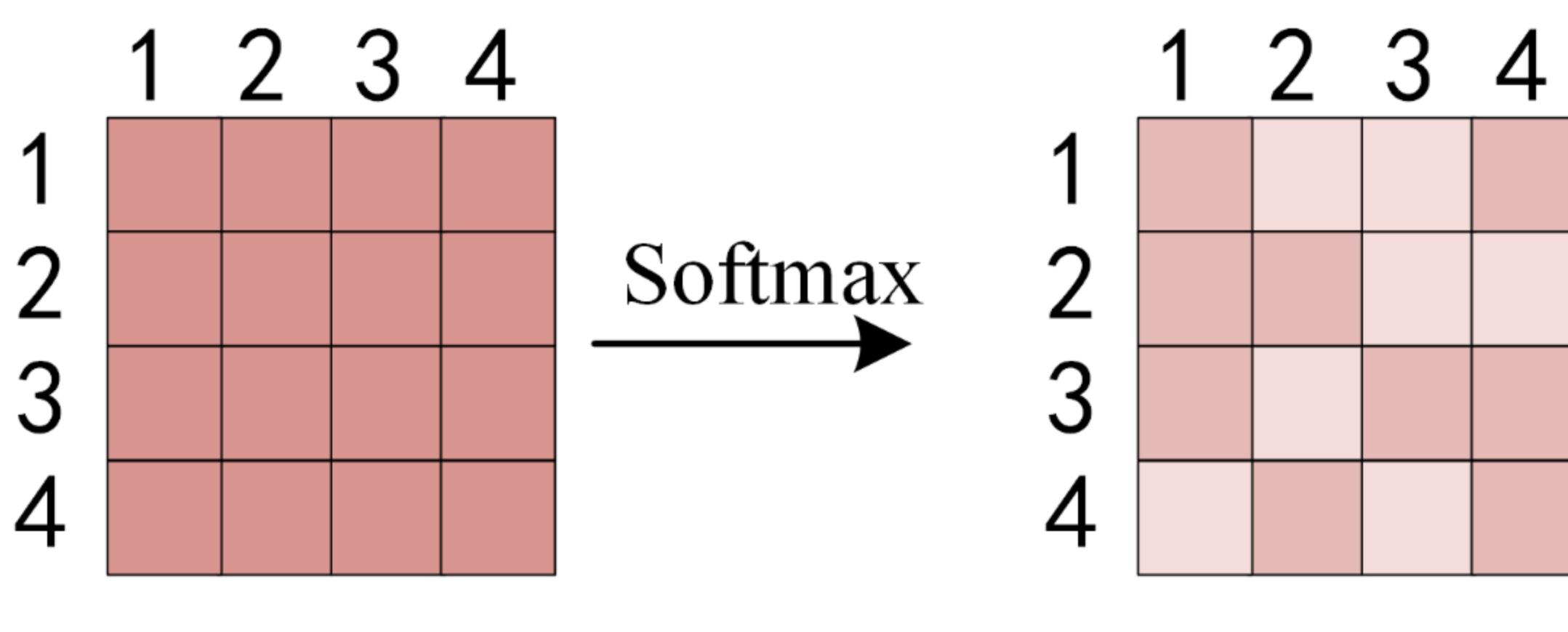
Self-Attention的计算过程如下：

第一步： X 与 W 进行矩阵乘法，生成 Q 、 K 和 V 。

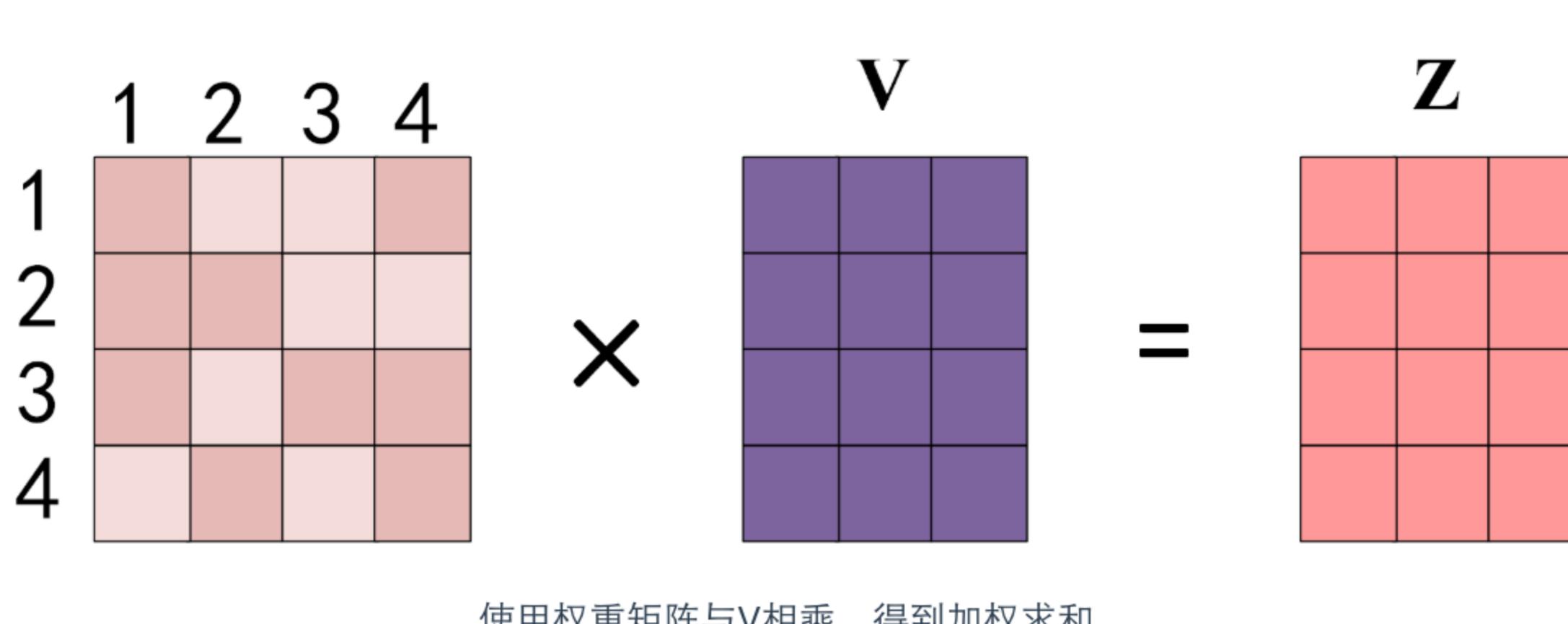
第二步：进行 QK^T 计算，得到相似度，如下图所示。



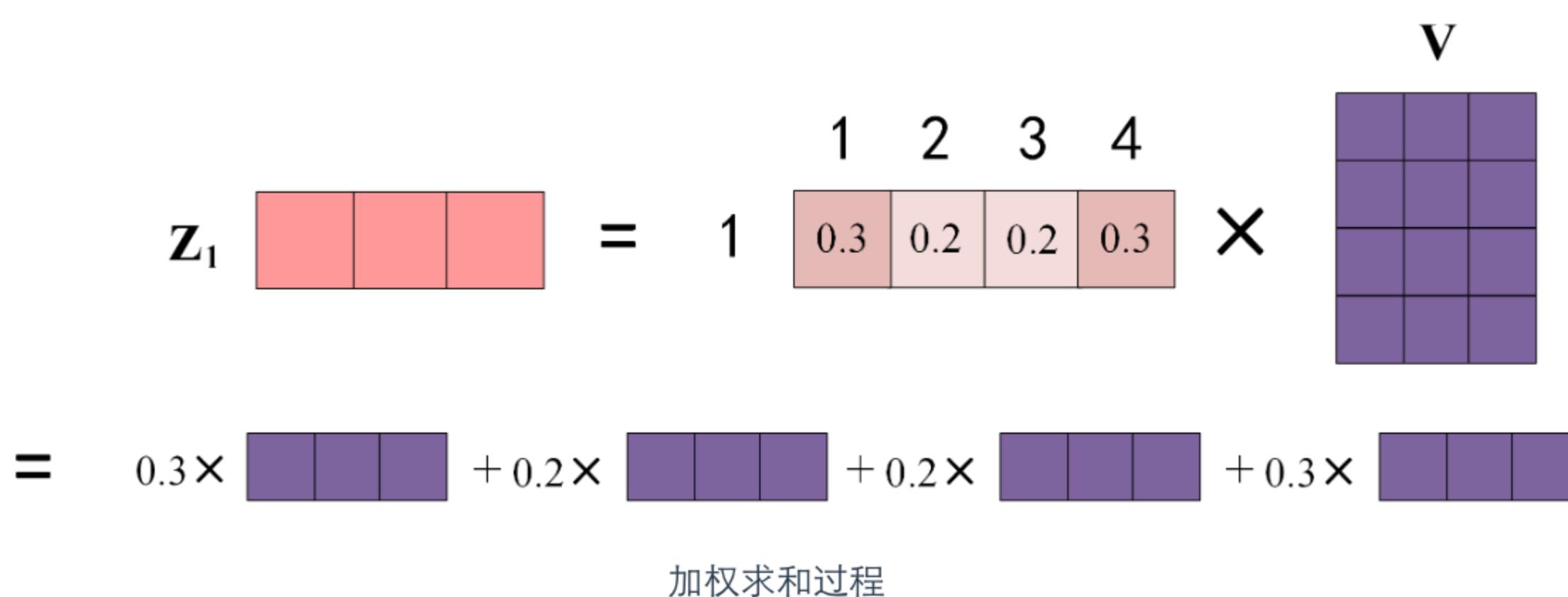
第三步：将刚得到的相似度除以 $\sqrt{d_k}$ ，再进行Softmax。经过Softmax的归一化后，每个值是一个大于0小于1的权重系数（Attention Score），且每行总和为0，这是一个权重矩阵。



第四步：使用刚得到的权重矩阵，与 V 相乘，计算加权求和。



上图中权重矩阵的第1行表示单词1与其他所有单词的权重系数，最终单词1的输出 Z_1 等于所有单词 i 的值 V_i 根据权重系数加权求和，如下图所示：



:::caution

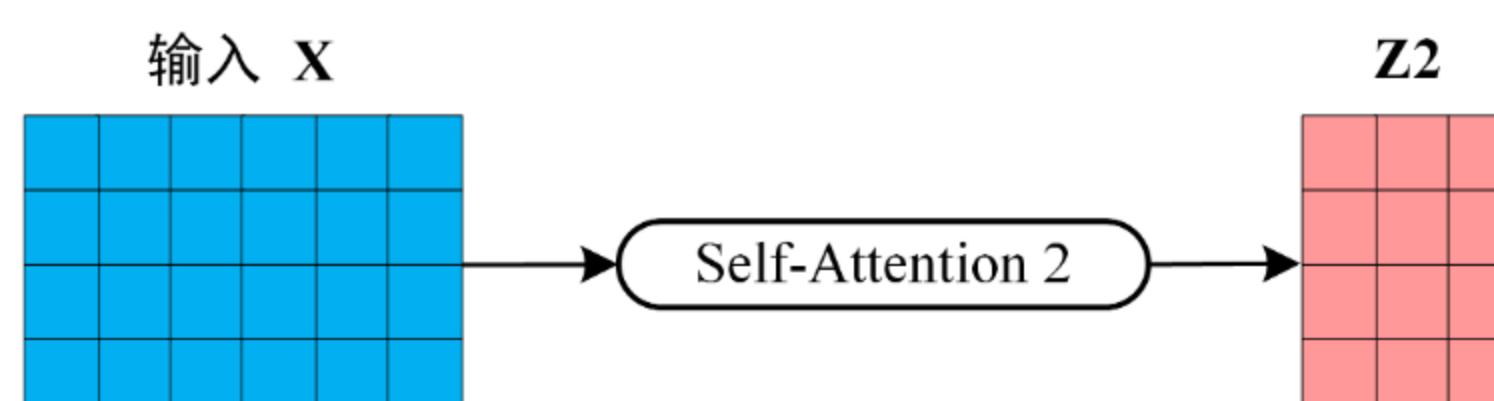
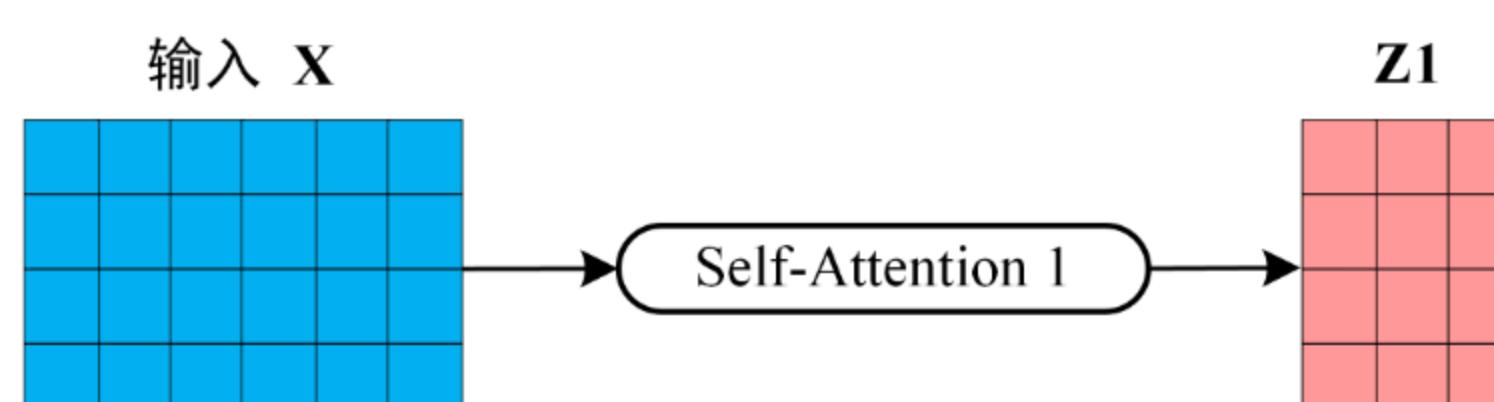
注意，Transformer Encoder中的Attention使用的是同样的输入矩阵 X ，而Decoder的Attention与之有些区别。

:::

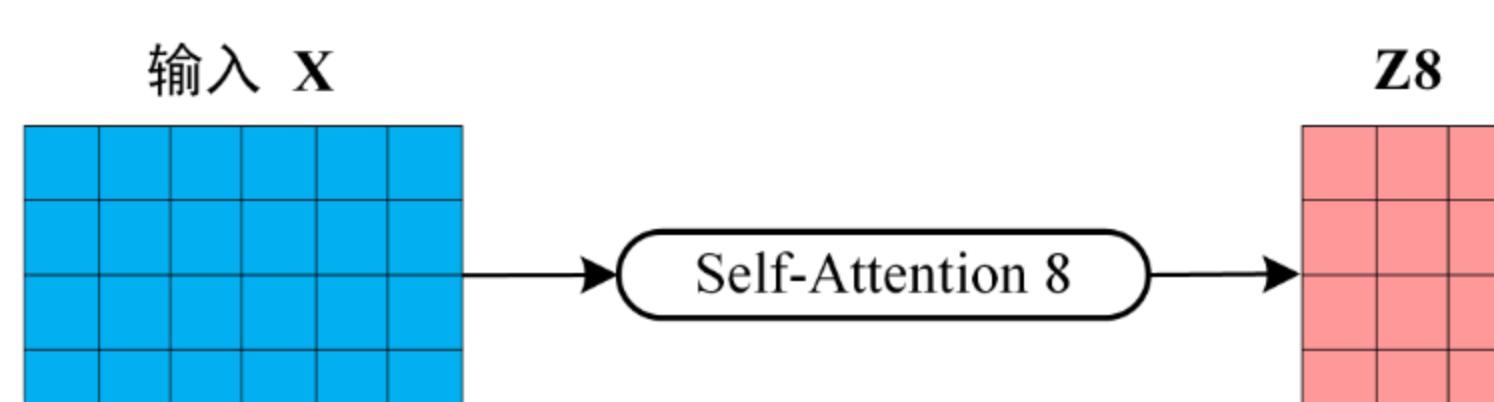
Multi-Head Attention

为了增强拟合性能，Transformer 对 Attention 继续扩展，提出了多头注意力（Multi-Head Attention）。刚才我们已经理解了， Q 、 K 、 V 是输入 X 与 W^Q 、 W^K 和 W^V 分别相乘得到的， W^Q 、 W^K 和 W^V 是可训练的参数矩阵。现在，对于同样的输入 X ，我们定义多组不同的 W^Q 、 W^K 、 W^V ，比如 W_0^Q 、 W_0^K 、 W_0^V ， W_1^Q 、 W_1^K 和 W_1^V ，每组分别计算生成不同的 Q 、 K 、 V ，最后学习到不同的参数。

比如我们定义8组参数，同样的输入 X ，将得到8个不同的输出 Z_0 到 Z_7 。



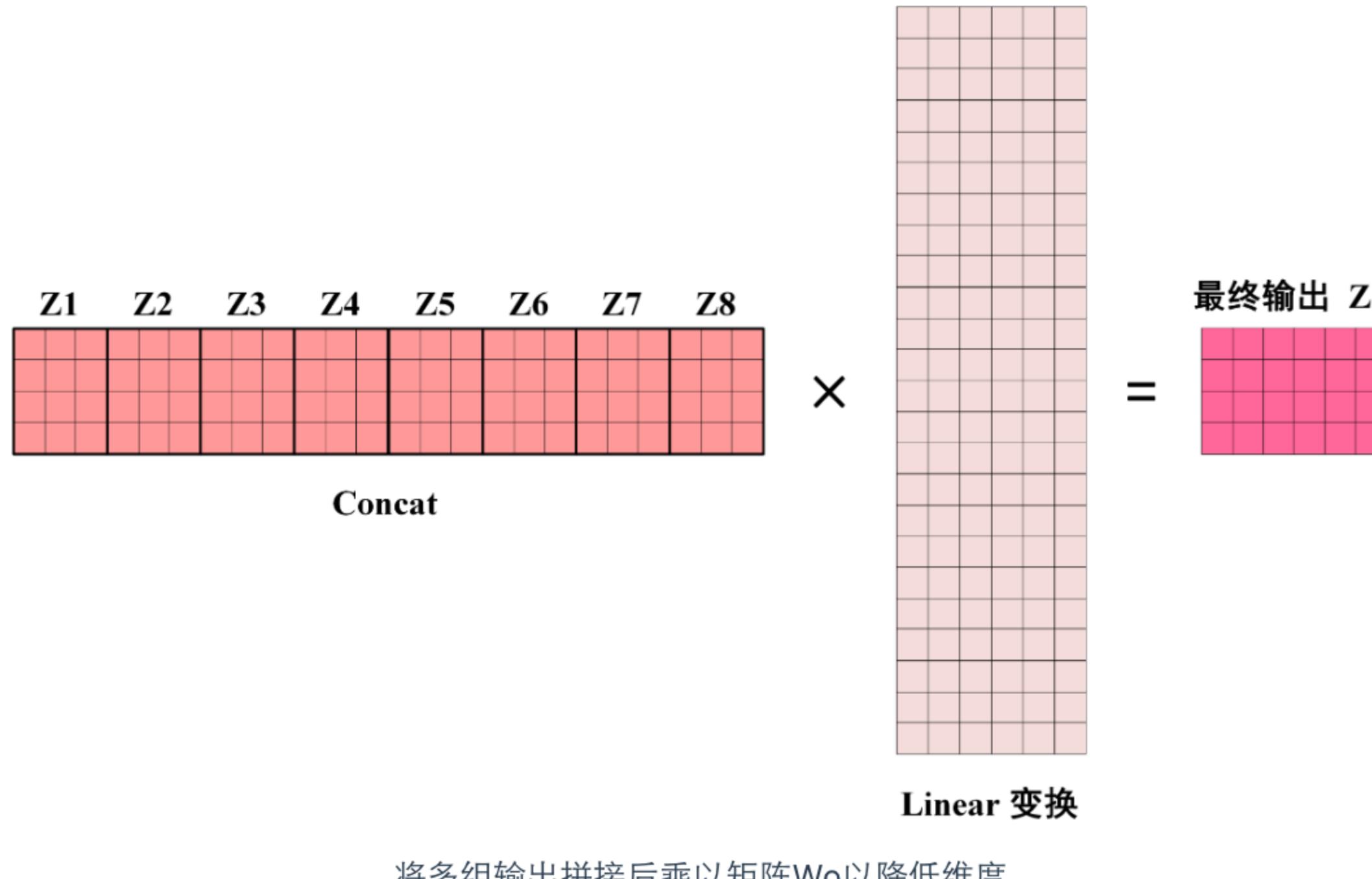
• • • •



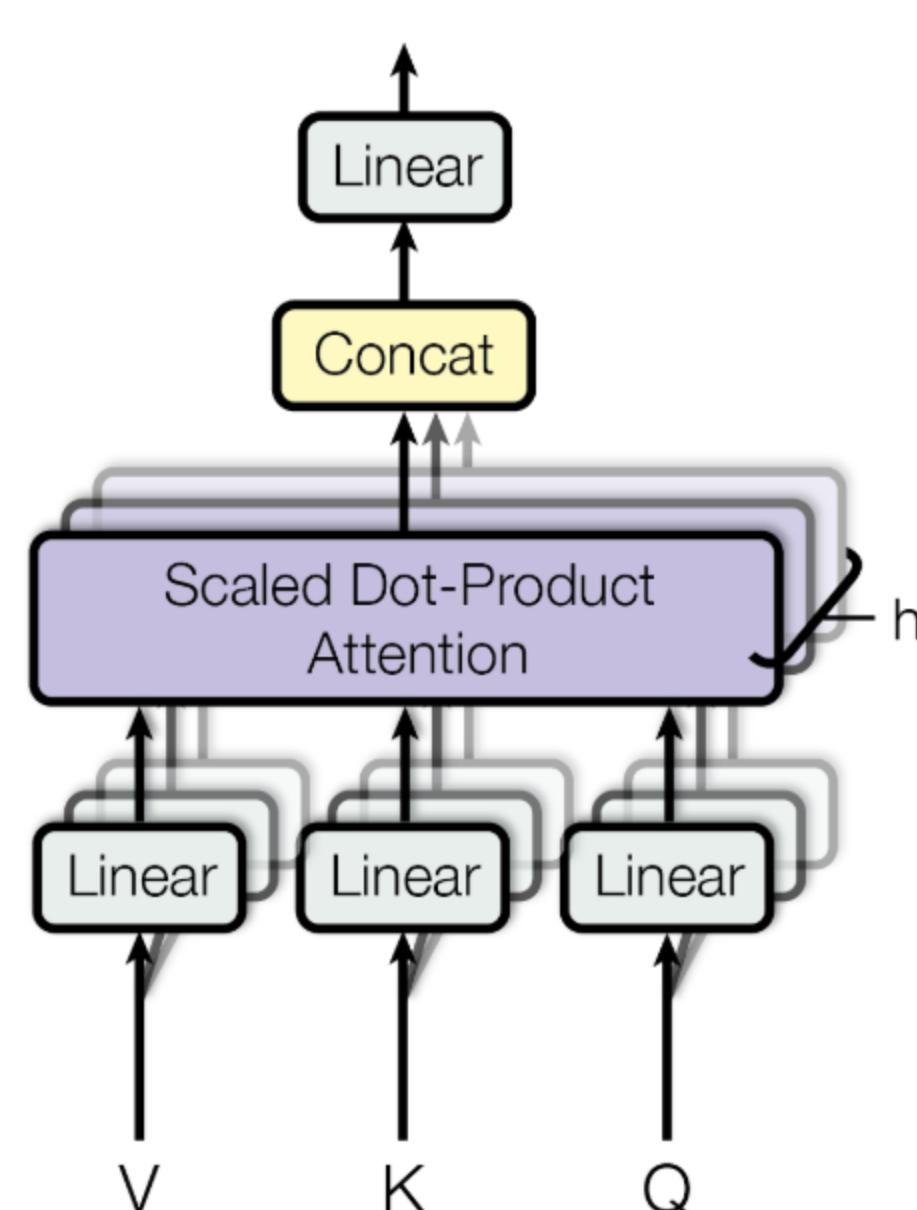
同一个输入 X ，使用多个Self-Attention，即使用多个不同的 W ，生成多组 Q 、 K 、 V

比如我们定义8组参数，同样的输入 X ，将得到8个不同的输出 Z_0 到 Z_7 。

在输出到下一层前，我们需要将8个输出拼接到一起（Concat），乘以矩阵 W^O ，进行一次线性变换，将维度降低回我们想要的维度。



再去观察Transformer论文中给出的多头注意力图示，似乎更容易理解了：



Transformer论文给出的多头注意力图示

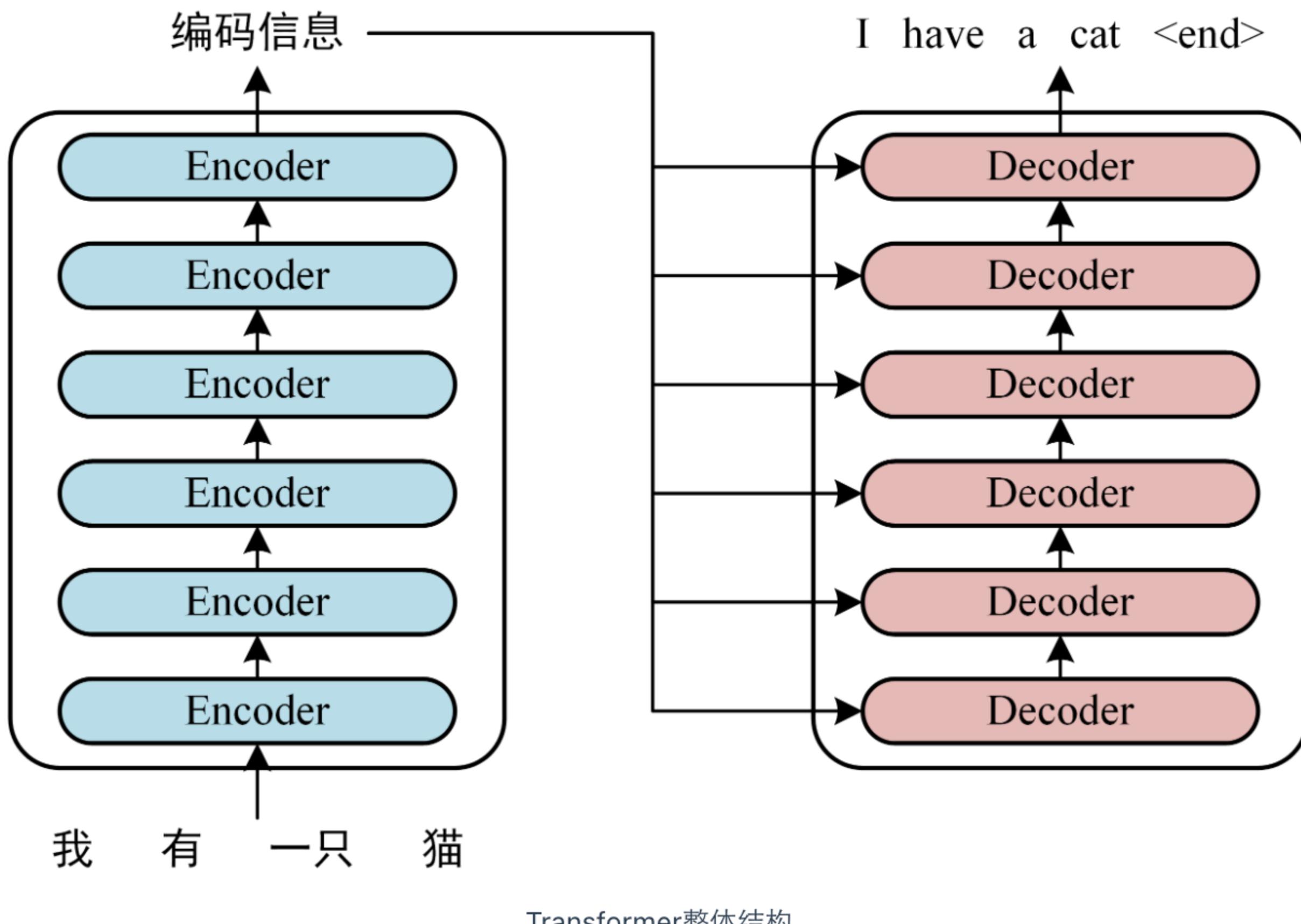
Transformer模型结构

刚才我们对Self-Attention以及由Self-Attention构造的Multi-Head Attention进行了了解析，下面我们从模型角度来分析一下Transformer模型。

Transformer主要面向机器翻译任务，训练数据是两种语言组成的句子对。Transformer模型使用的是WMT翻译数据集，[我的这篇文章](#)对数据预处理流程进行了解析。下面以一个中英翻译为例，解释一下Transformer模型的结构。例子中，源语言为中文“我有一只猫”，要翻译的目标语言为英文“I have a

cat"。

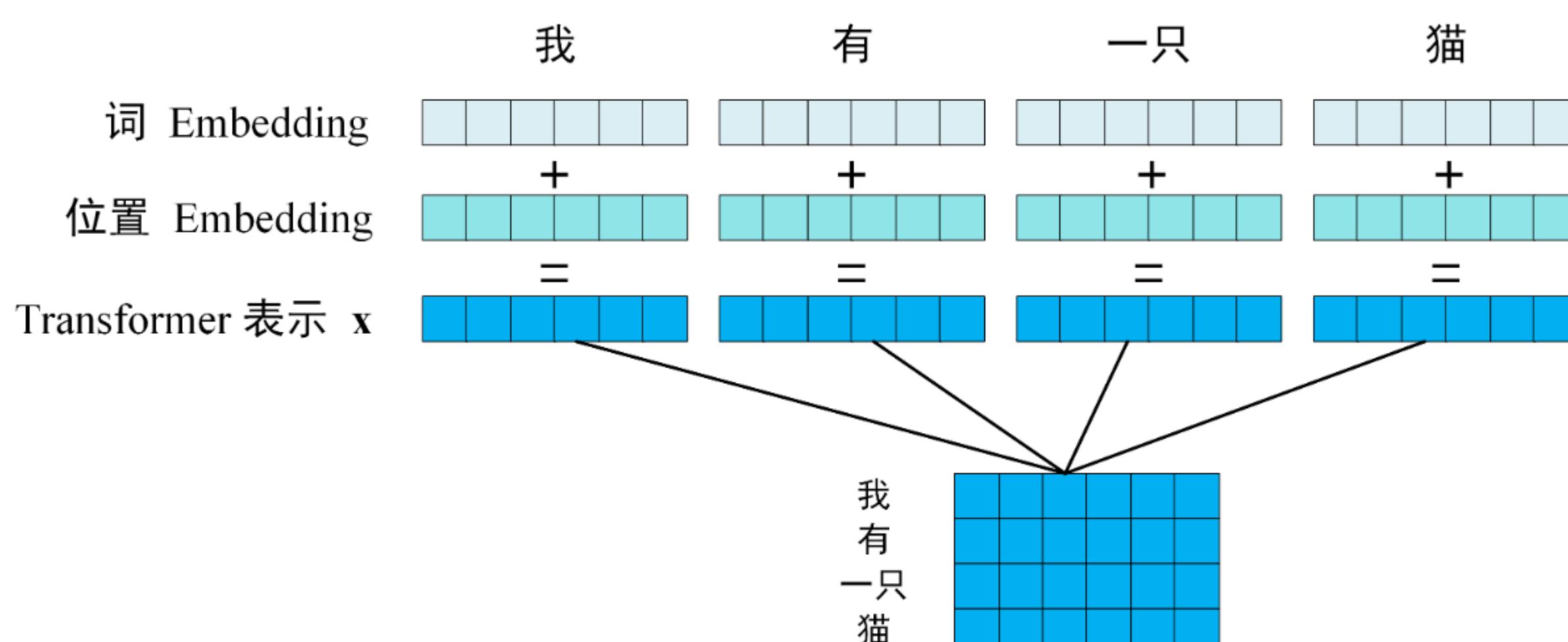
Transformer模型基于Encoder-Decoder架构。一般地，在Encoder-Decoder中，Encoder部分将一部分信息抽取出来，生成中间编码信息，发送到Decoder中。下图中，左侧为Transformer的Encoder部分，右侧为Transformer的Decoder部分，Encoder和Decoder分别有 $N = 6$ 个Block。



Transformer整体结构

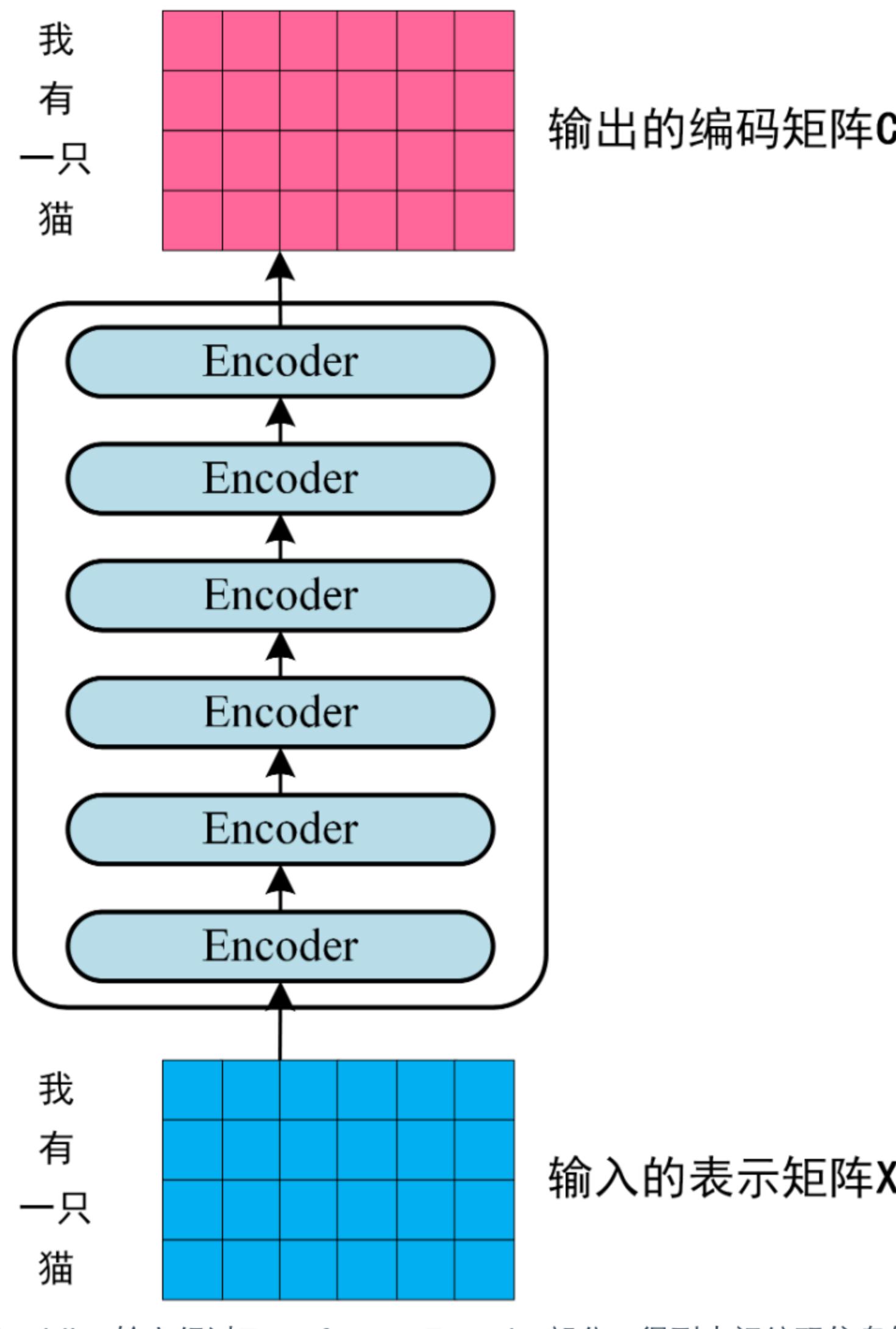
对于输入，Transformer的工作流程有主要三步：

第一步： 获取输入句子的每一个单词的表示向量 X ， X 由单词的词向量 Embedding 和与单词位置有关的 Positional Embedding 相加得到。



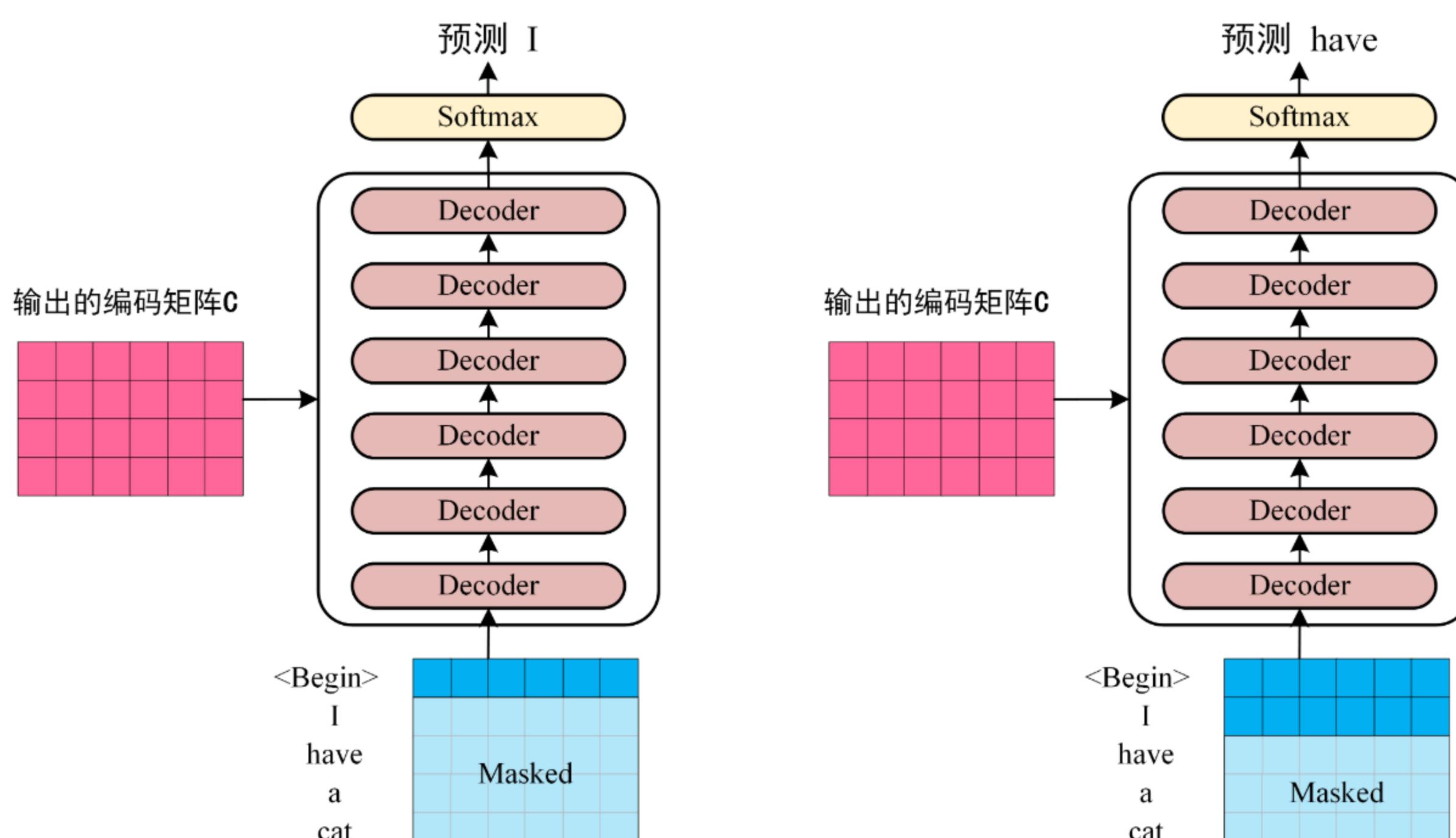
Transformer从输入到进入Encoder之前先经过两层Embedding转换，得到一个中间表示X

第二步： 将得到的单词表示矩阵 (如上图所示，每一行是一个单词的向量表示 \mathbf{X}_i) 传入Encoder部分 (共6个Encoder Block) 中，经过 6 个 Encoder Block 后可以得到句子所有单词的中间编码信息矩阵 C ，如下图。Encoder部分输入为 \mathbf{X} ， \mathbf{X} 维度为： $(len \times d)$ ， len 是句子中单词个数， d 是表示词向量的维度 (Transformer-base: $d = 512$ ，Transformer-big: $d = 1024$)。每一个 Encoder Block 输出的矩阵维度与输入完全一致。



Embedding输入经过Transformer Encoder部分，得到中间编码信息矩阵C

第三步： 将Encoder输出的编码信息矩阵 C 传递到Decoder部分，Decoder依次会根据已经翻译过的单词（第 0 个单词到第 $i-1$ 个单词）翻译下一个单词 i 。最下方第一个Decoder Block的输入为目标语言经过两层Embedding之后的词矩阵表示 X 。下图左侧为单词“I”的Decoder翻译过程，右侧为单词“have”的Decoder的翻译过程。在Decoder过程中，翻译到单词 i 的时候需要通过**Mask**操作遮盖住 i 之后的单词。



Transformer Decoder翻译过程，左侧为单词“I”的Decoder翻译过程，右侧为单词“have”的Decoder的翻译过程

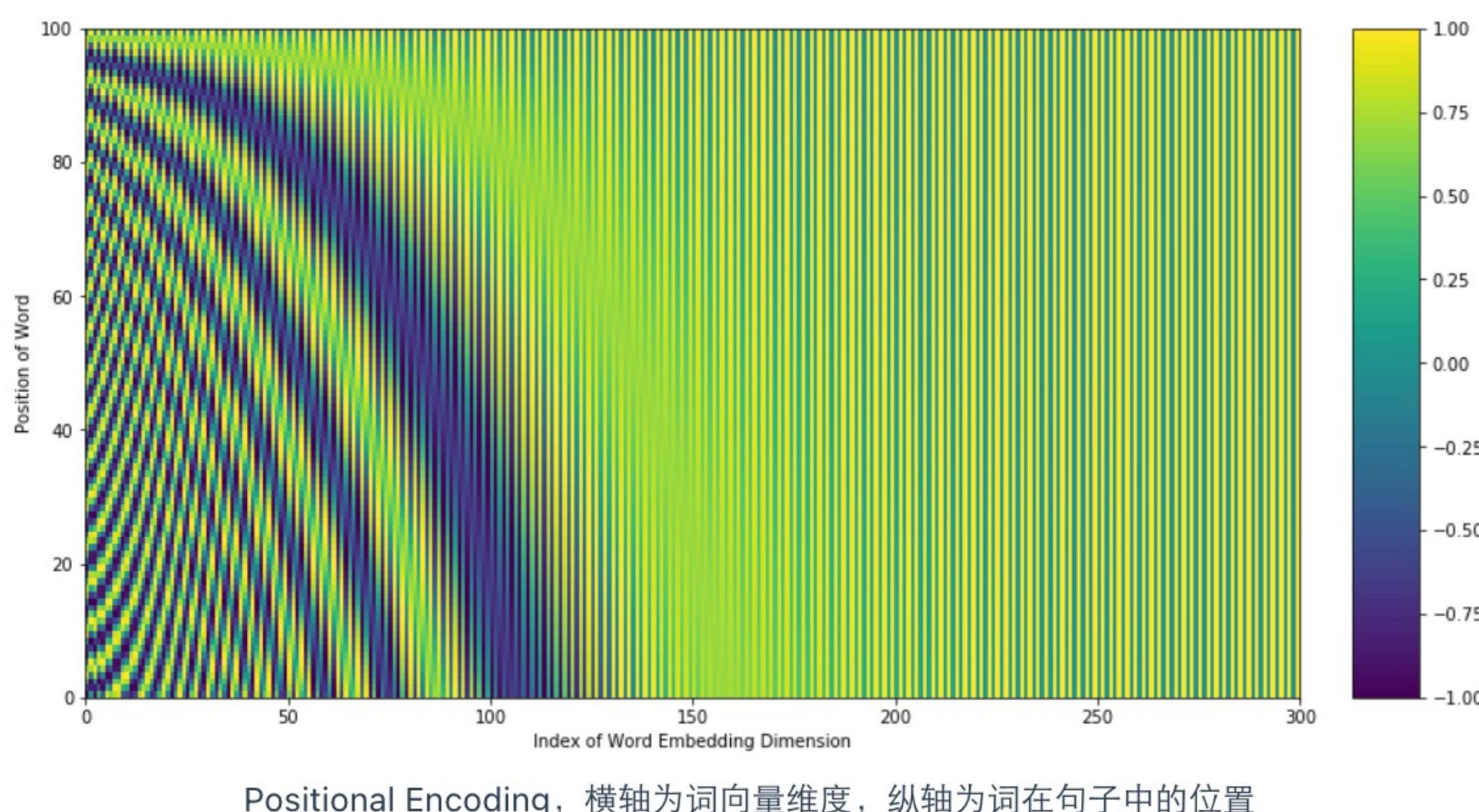
上图中，Decoder部分根据目标语言的词矩阵表示 X ，以及Encoder部分传过来的中间编码矩阵 C ，依次预测下一个词。图左侧根据句子开始符"<Begin>"，预测第一个单词 "I"; 右侧根据开始符"<Begin>" 和单词 "I"，预测单词 "have"，以此类推。

Positional Encoding

Transformer模型的输入为一系列词，词需要转化为词向量。一般的语言模型都需要使用Embedding层，用以将词转化为词向量。在此基础上，Transformer增加了位置编码（Positional Encoding）。**Transformer没有采用RNN的结构，不能利用单词的顺序信息，但顺序信息对于NLP任务来说非常重要。**为解决无位置信息的问题，Transformer使用位置编码来增加位置信息。

$$\begin{aligned} PE_{(pos,2i)} &= \sin(pos/10000^{2i/d}) \\ PE_{(pos,2i+1)} &= \cos(pos/10000^{2i/d}) \end{aligned}$$

其中， pos 表示单词在句子中的位置， d 表示词向量的维度， $2i$ 表示偶数维度， $2i + 1$ 表示奇数维度\$ (2i \leq d, 2i+1 \leq d)\$。



PE公式生成的是 $[-1, 1]$ 区间内的实数。词向量维度为 d ， d 是模型设计时就确定的。给定一个词，其在句子中的位置为 pos ，就是上图中的某一行。Positional Encoding就是从上图中找到一行，将这行加到词向量上。

使用这种公式计算 PE 有以下的好处：可以让模型容易地计算出相对位置，对于固定长度的间距 k ， $PE(pos + k)$ 可以用 $PE(pos)$ 计算得到。因为：

$$\begin{aligned} \sin(A + B) &= \sin A \cos B + \cos A \sin B \\ \cos(A + B) &= \cos A \cos B - \sin A \sin B \end{aligned}$$

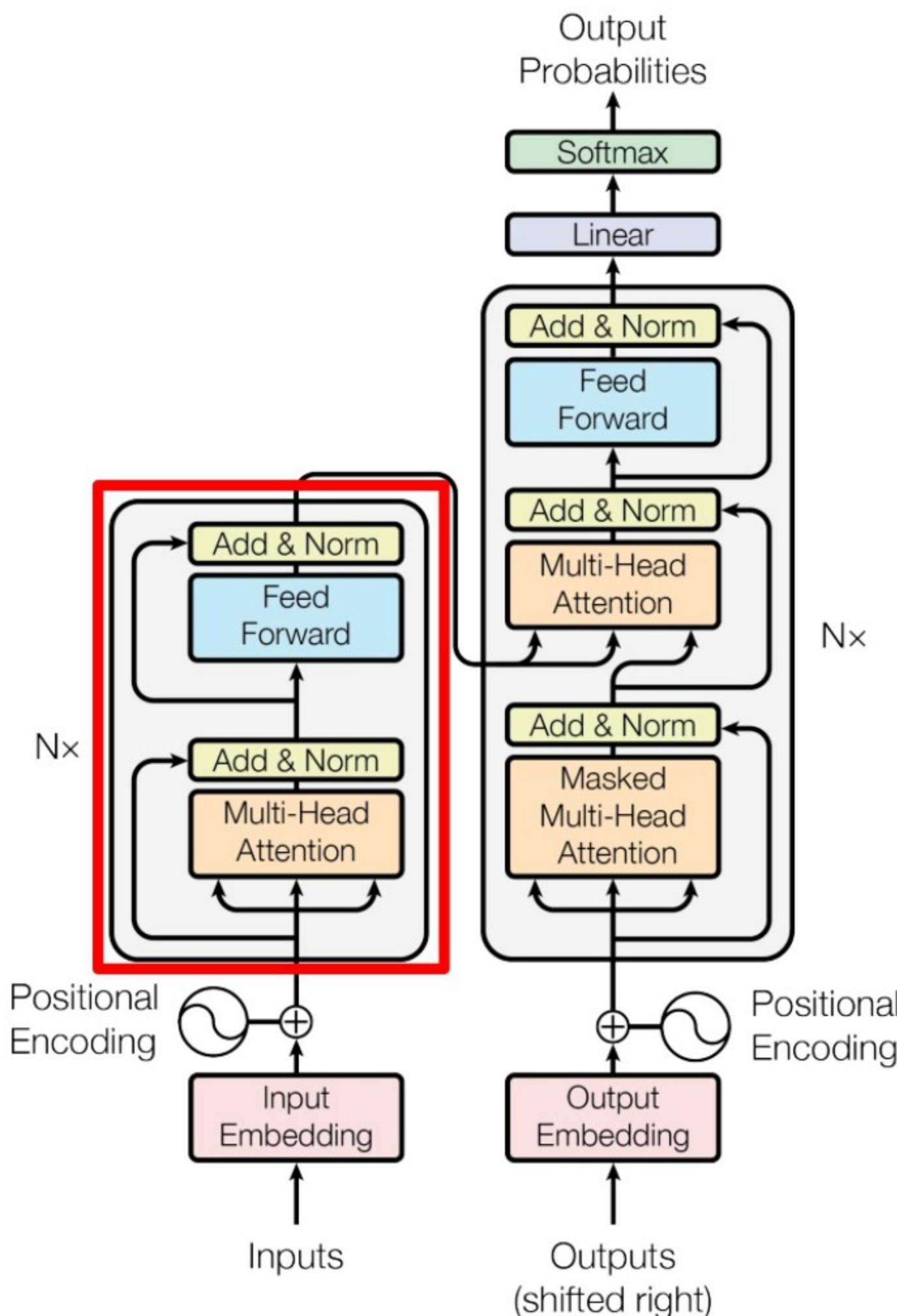
将词向量Embedding和PE相加，就可以得到词的表示向量 \mathbf{X} ， \mathbf{X} 就是 Transformer 的输入。

Encoder Block

下图红色方框中是 Transformer 的 Encoder Block，可以看到是由 Multi-Head Attention, Add & Norm, Feed Forward, Add & Norm 组成的。整个Encoder由 $N = 6$ 个同样的Encoder Block堆叠而成。

所有Encoder Block的输入输出维度相同： $(len \times d)$ ， len 是句子中单词个数， d 是表示词向量的维度 (Transformer-base: $d = 512$, Transformer-big: $d = 1024$)。第一个Encoder Block的输入是经过两层Embedding得到的，后续几个Encoder Block的输入是前一个Encoder Block的输出。

前文已经介绍了Multi-Head Attention的计算过程，现在了解一下 Add & Norm 和 Feed Forward 部分。



Encoder Block由Multi-Head Attention、Add & Norm和Feed Forward组成

Add & Norm

Add & Norm 层由 Add 和 Norm 两部分组成。Add 类似ResNet提出的残差连接，以解决深层网络训练不稳定的问题。Norm为归一化层，即Layer Normalization，通常用于 RNN 结构。

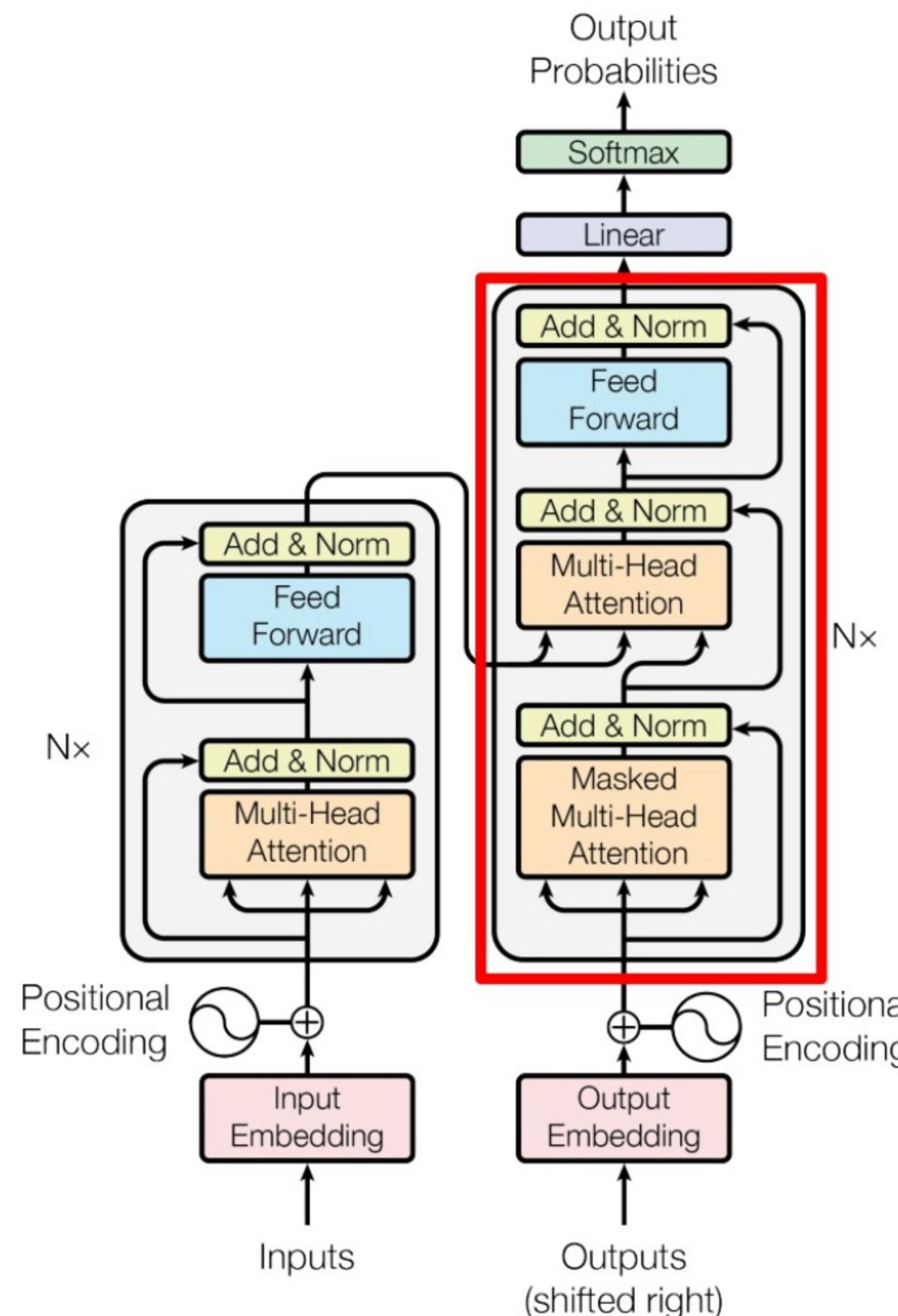
Feed Forward

Feed Forward 层比较简单，由两个全连接层构成，第一层的激活函数为 ReLu，第二层不使用激活函数，对应的公式如下。

$$(\max(0, \mathbf{X}\mathbf{W}_1 + \mathbf{b}_1))\mathbf{W}_2 + \mathbf{b}_2$$

对于输入 \mathbf{X} , Feed Forward 最终得到的输出矩阵的维度与输入 \mathbf{X} 一致。

Decoder Block



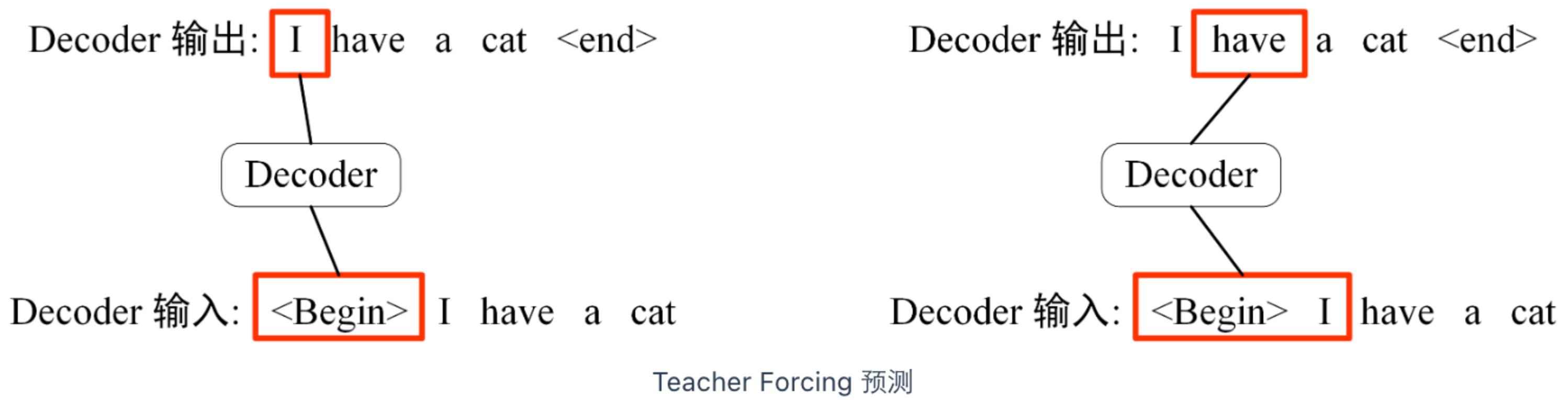
Decoder Block 也由 Multi-Head Attention、Add & Norm 和 Feed Forward 组成，但 Decoder Block 中的 Multi-Head Attention 与 Encoder Block 不同

上图红色方框部分为 Transformer 的 Decoder Block 结构，与 Encoder Block 相似，但是存在一些区别：

- 每个 Decoder Block 包含两个 Multi-Head Attention 层。
- 第一个 Multi-Head Attention 层采用了 Masked 操作。
- 第二个 Multi-Head Attention 层的 \mathbf{K} 和 \mathbf{V} 矩阵使用 Encoder 的编码信息矩阵 \mathbf{C} ，而 \mathbf{Q} 使用上一个 Decoder Block 的输出。

第一个 Multi-Head Attention

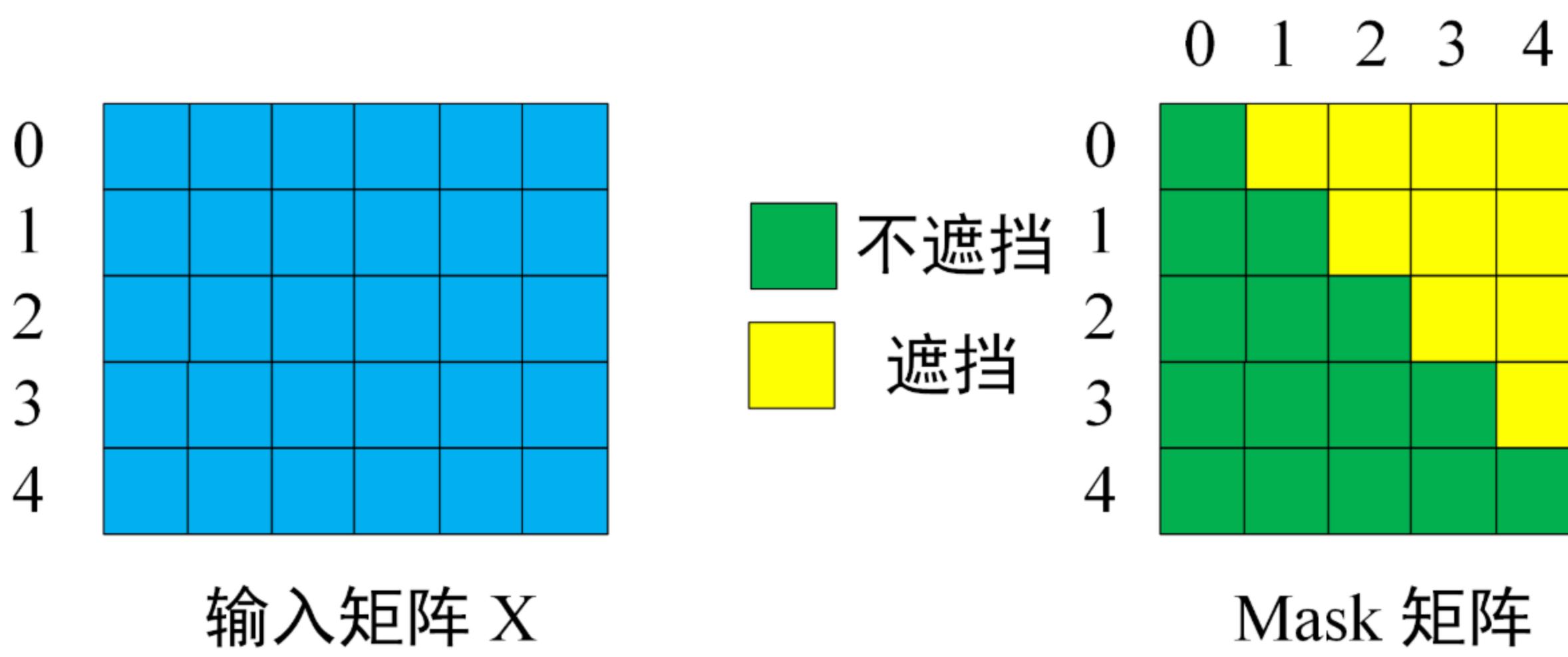
Decoder Block 的第一个 Multi-Head Attention 采用了 Mask 操作，因为在翻译的过程中是顺序翻译的，即翻译完第 i 个单词，才可以翻译第 $i + 1$ 个单词。通过 Mask 操作可以防止第 i 个单词知道 $i + 1$ 个单词之后的信息。下面以 "我有一只猫" 翻译成 "I have a cat" 为例，了解一下 Mask 操作。



这里使用了类似 Teacher Forcing 的概念。在 Decoder 的时候，是需要根据之前的翻译，求解当前最有可能的翻译，如下图所示。首先根据输入 "<Begin>" 预测出第一个单词为 "I"，然后根据输入 "<Begin> I" 预测下一个单词 "have"。

Decoder 可以在训练的过程中使用 Teacher Forcing 并且并行化训练，即将正确的单词序列 (<Begin> I have a cat) 和对应输出 (I have a cat <end>) 传递到 Decoder。那么在预测第 i 个输出时，就要将第 $i + 1$ 之后的单词掩盖住，注意，Mask 操作是在 Self-Attention 的 Softmax 之前使用的。下面用 0 1 2 3 4 分别表示 "<Begin> I have a cat <end>"。

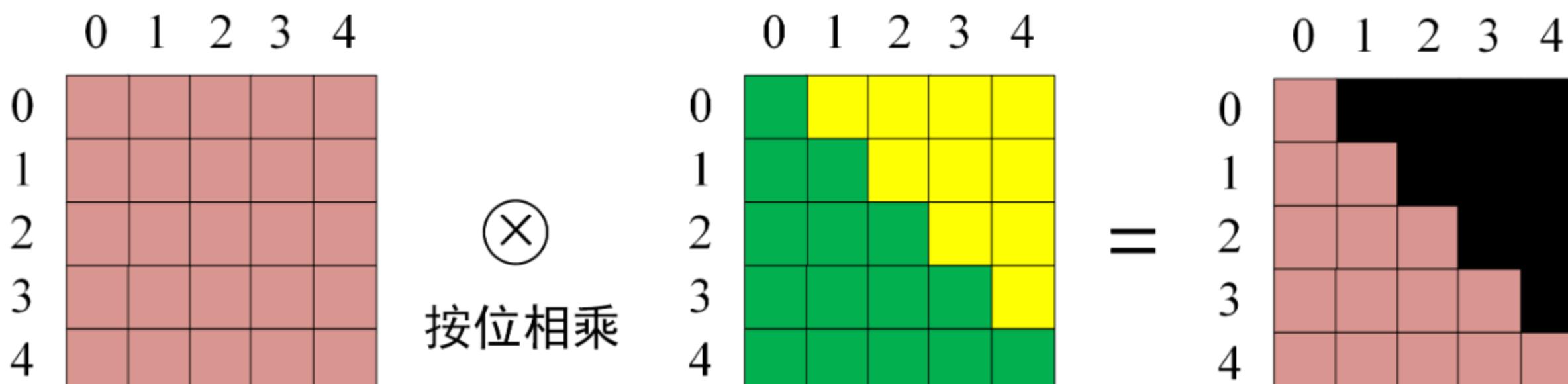
第一步：根据 Decoder 的输入矩阵生成 Mask 矩阵，输入矩阵包含 "<Begin> I have a cat" (0, 1, 2, 3, 4) 共5个单词的表示向量，Mask矩阵是一个 5×5 的上三角矩阵。Mask目的是翻译单词 0 时只能使用单词 0 的信息，而翻译单词 1 可以使用单词 0, 1 的信息，即只能使用之前的信息。



Mask矩阵

第二步：接下来的操作和之前的 Self-Attention 一样，通过输入矩阵 X 计算得到 Q, K, V 矩阵。然后计算 Q 和 K^\top 的乘积 QK^\top 。

第三步：在得到 QK^\top 之后需要进行 Softmax，计算 Attention Score，我们在 Softmax 之前需要使用 Mask 矩阵遮挡住每一个单词之后的信息，遮挡操作如下：



\mathbf{QK}^T

Mask 矩阵

Mask \mathbf{QK}^T

Mask矩阵工作原理：在Self-Attention中Softmax之前使用，使得后续词不参与Attention Score的计算

在 $\mathbf{Masked QK}^T$ 上进行 Softmax，每一行的和都为 1。但是单词 0 在单词 1, 2, 3, 4 上的 Attention Score 都为 0。那么下一步 $\mathbf{Masked QK}^T$ 与 V 相乘，得到的输出也是经过Mask之后的。

第二个Multi-Head Attention

Decoder Block 第二个 Multi-Head Attention 主要的区别在于 Attention 的 K, V 矩阵不是来自上一个 Decoder Block 的输出计算的，而是来自Encoder的编码信息矩阵 C 。

对于第二个 Multi-Head Attention，根据 Encoder 的输出 C 计算得到 K, V ，根据上一个 Decoder Block 的输出 Z 得到 Q 。

Attention在Transformer中的应用

文章一开始解释了Self-Attention和Multi-Head Attention。通过对Transformer模型的深入解读，可以看到，模型一共使用了三种Multi-Head Attention：

1. Encoder Block中使用的Attention。第一个Encoder Block的Query、Key和Value来自训练数据经过两层Embedding转化，之后的Encoder Block的Query、Key和Value来自上一个Encoder Block的输出。
2. Decoder Block中的第一个Attention。与Encoder Block中的Attention类似，只不过增加了Mask，在预测第 i 个输出时，要将第 $i + 1$ 之后的单词掩盖住。第一个Decoder Block的Query、Key和Value来自训练数据经过两层Embedding转化，之后的Decoder Block的Query、Key和Value来自上一个Decoder Block的输出。
3. Decoder Block中的第二个Attention。这是一个 Encoder-Decoder Attention，它建立起了 Encoder 和 Decoder 之间的联系，Query来自第2种 Decoder Attention的输出，Key和Value 来自 Encoder 的输出。

预测输出词概率

最后一个Decoder Block输出得到的编码矩阵，是一个浮点数组成的矩阵，如何用这些浮点矩阵预测所要翻译的词？经过 $N = 6$ 层的Decoder Block堆叠后，最后的 Linear 和 Softmax 层预测下一个翻译单词的概率。

Linear 层将 Decoder Block 输出的编码矩阵，转化成一个跟词表大小一样的 logit 矩阵。词表通常很大，比如 WMT 翻译任务中，英德词表有三万多个 subword。

最后的损失函数 Loss Function 可以是交叉熵，或者 KL 散度。

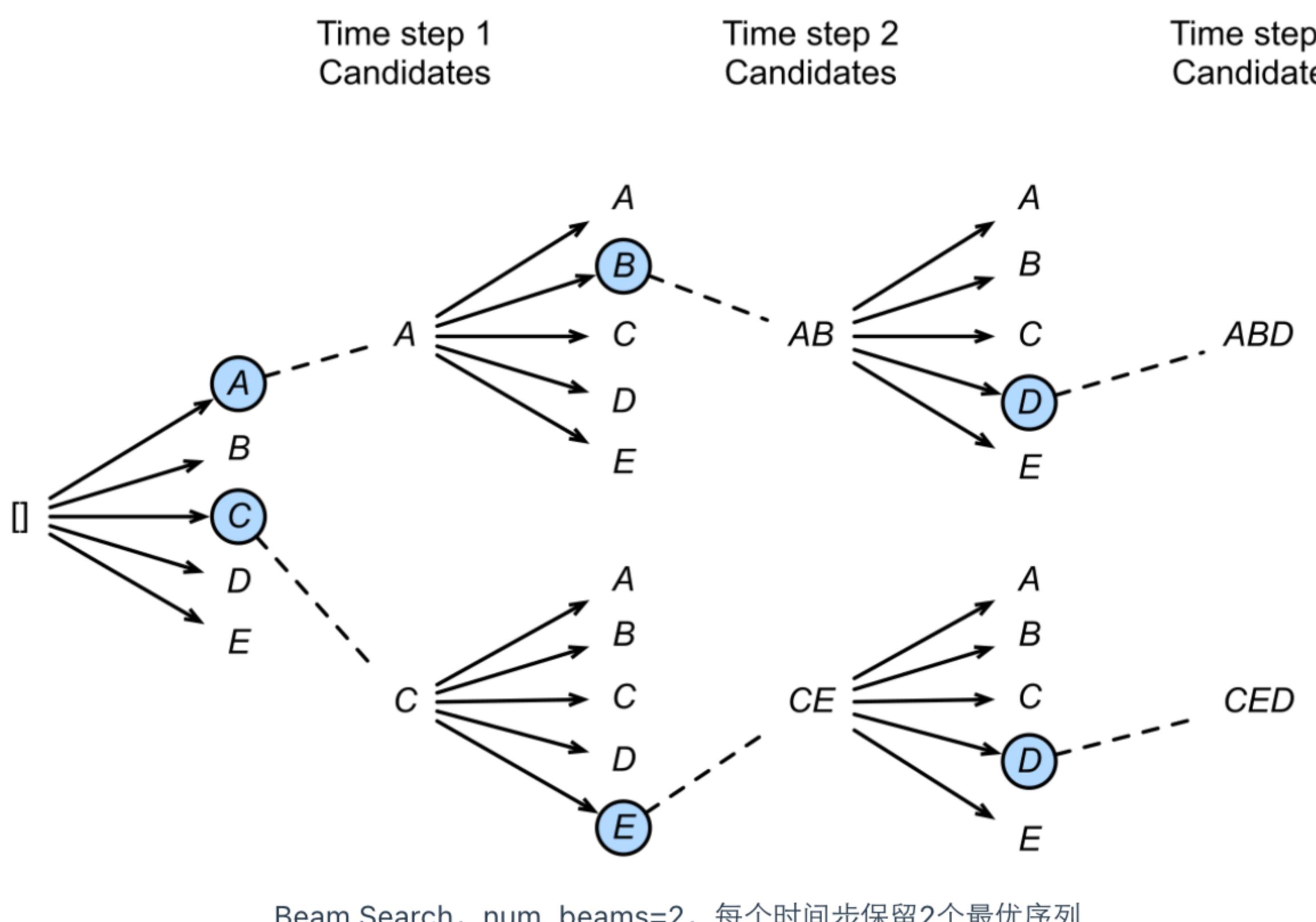
由于Decoder使用了Mask，在预测时也需要注意，我们只关注当前要预测的词的 logit，使用这个 logit 预测输出词。

Beam Search

前文讨论的主要训练过程，在推理时，Decoder侧最后的 Softmax 将 logit 转化为概率，选择概率最大的词作为预测词。Encoder侧输入是源语言句子。第一个时间步，Decoder侧首先以开始符 "`<Begin>`" 作为输入，预测下一个概率最大的词；第二个时间步，Decoder侧以开始符和第一个预测词作为输入，来预测下一个概率最大词。每次基于上一次预测结果，选择最优的解，这是一种贪心搜索算法。贪心搜索从局部来说可能是最优的，但是从全局角度并不一定是最优的。

Beam Search对贪心算法进行了改进。在每一个时间步预测时，不再只选择最优解，而是保留 `num_beams` 个结果。当 `num_beams=1` 时 Beam Search 就退化成了贪心搜索。

下图中，每个时间步有ABCDE共5种可能的输出，设置 `num_beams=2`，也就是说每个时间步都会保留到当前步为止条件概率最优的2个序列。



Beam Search, `num_beams=2`, 每个时间步保留2个最优序列

在第一个时间步，A和C是最优的两个，因此得到了两个结果[A],[C]，其他三个就被抛弃了；第二步会基于这两个结果继续进行生成，在A这个分支可以得到5个候选人，[AA],[AB],[AC],[AD],[AE]，C也同理得到5个，此时会对这10个进行统一排序，再保留最优的两个，即图中的[AB]和[CE]；第三步同理，也会从新的10个候选人里再保留最好的两个，最后得到了[ABD],[CED]两个结果。可以发现，Beam Search在每一步需要考察的候选人数量是贪心搜索的 `num_beams` 倍，因此是一种牺牲时间换取准确率的方法。

参考资料

- Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need. 31st Conference on Neural Information Processing Systems 2017(NIPS 2017). Long Beach, CA, USA: 2017: 5998–6008.

2. [The Illustrated Transformer](#)

3. [The Annotated Transformer](#)

4. [Transformer 模型详解](#)

5. [Beam Search](#)