Transformer 模型详解

2017 年，Google 在论文 Attention is All you need 中提出了 Transformer 模型，其使用 Self-Attention 结构取代了在 NLP 任务中常用的 RNN 网络结构。相比 RNN 网络结构，其最大的优点是可以并行计算。Transformer 的整体模型架构如图 1.1 所示：

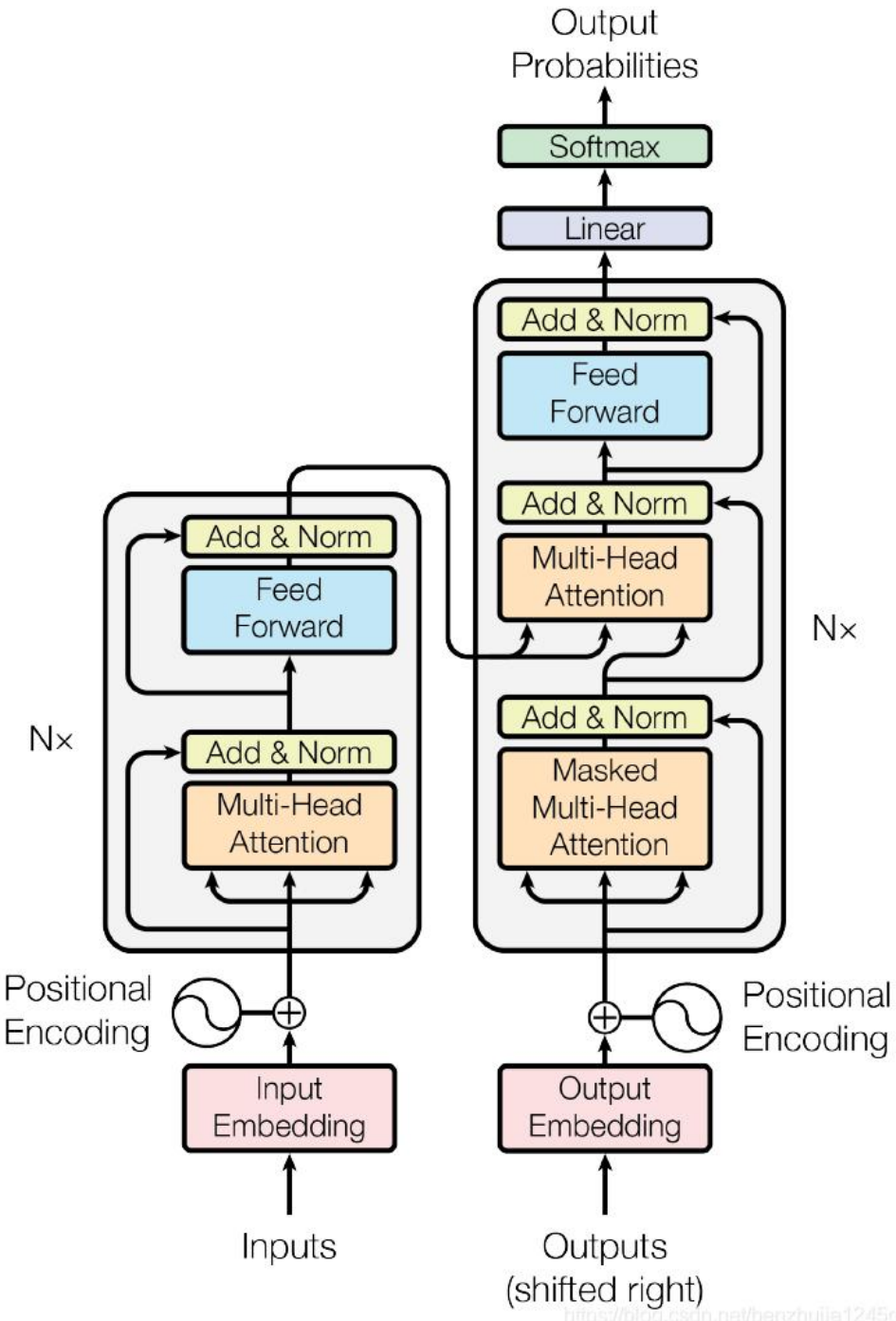

图 1.1 Transformer 模型架构

1.1 Transformer 概览

首先，让我们先将 Transformer 模型视为一个黑盒，如图 1.2 所示。在机器翻译任务中，将一种语言的一个句子作为输入，然后将其翻译成另一种语言的一个句子作为输出。

图 1.2 Transformer 模型（黑盒模式）

Transformer 本质上是一个 Encoder-Decoder 架构。因此中间部分的 Transformer 可以分为两个部分：编码组件和解码组件。如图 1.3 所示：

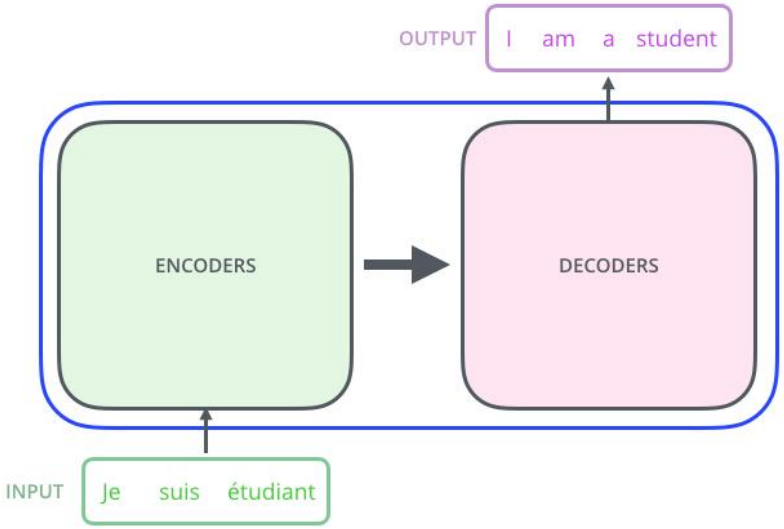

图 1.3 Transformer 模型（Encoder-Decoder 架构模式）

其中，编码组件由多层编码器（Encoder）组成（在论文中作者使用了 6 层

编码器，在实际使用过程中你可以尝试其他层数）。解码组件也是由相同层数的解码器（Decoder）组成（在论文也使用了 6 层）。如图 1.4 所示：

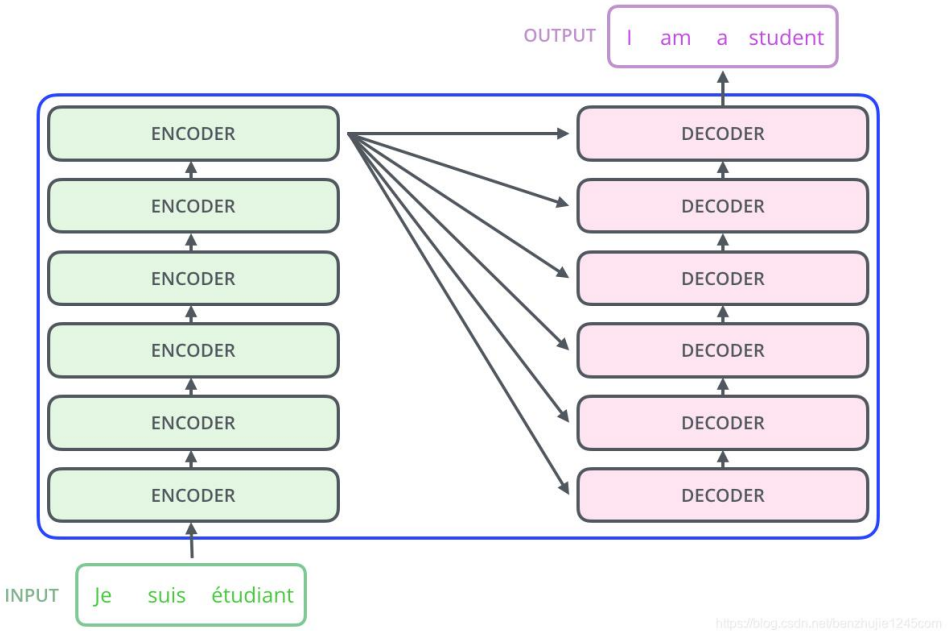

图 1.4

每个编码器由两个子层组成：Self-Attention 层（自注意力层）和 Position-wise Feed Forward Network（前馈网络，缩写为 FFN）如图 1.5 所示。每个编码器的结构都是相同的，但是它们使用不同的权重参数。

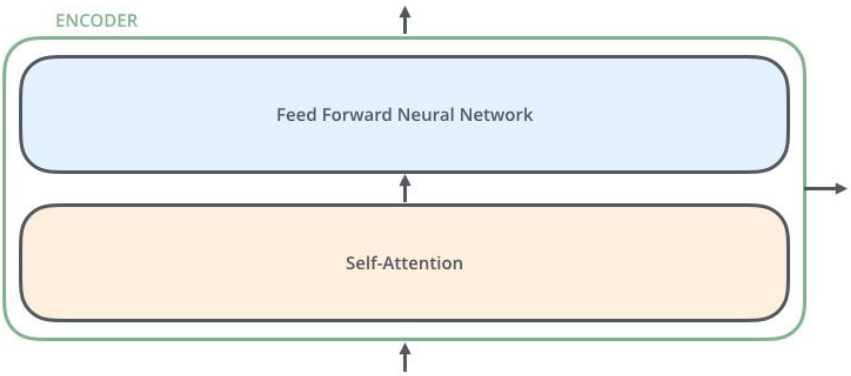

图 1.5

编码器的输入会先流入 Self-Attention 层。它可以让编码器在对特定词进行编码时使用输入句子中的其他词的信息（可以理解为：当我们翻译一个词时，不仅只关注当前的词，而且还会关注其他词的信息）。后面我们将会详细介绍 Self-Attention 的内部结构。然后，Self-Attention 层的输出会流入前馈网络。

解码器也有编码器中这两层，但是它们之间还有一个注意力层（即 Encoder-Decoder Attention），其用来帮忙解码器关注输入句子的相关部分（类似于 seq2seq 模型中的注意力）。

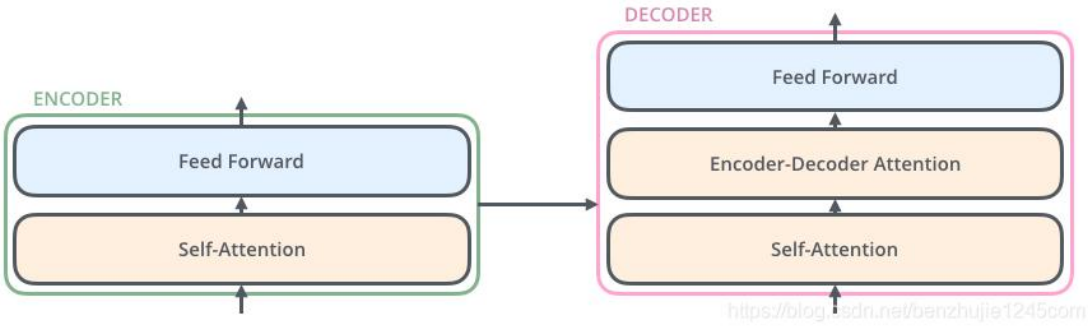

图 1.6

1.2 引入张量

现在我们已经了解了模型的主要组成部分，让我们开始研究各种向量/张量，以及他们在这些组成部分之间是如何流动的，从而将输入经过已训练的模型转换为输出。

和通常的 NLP 任务一样，首先，我们使用词嵌入算法（Embedding）将每个词转换为一个词向量。在 Transformer 论文中，词嵌入向量的维度是 512。

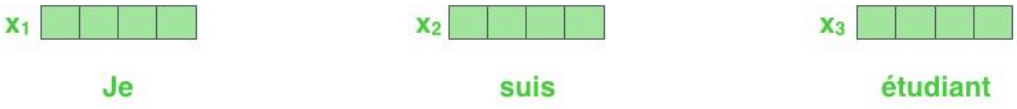

图 1.7 每个词被嵌入到大小为 512 的向量中。我们将用这些简单的框代表这些向量。

嵌入仅发生在最底层的编码器中。所有编码器都会接收到一个大小为 512 的向量列表——底部编码器接收的是词嵌入向量，其他编码器接收的是上一个编码器的输出。这个列表大小是我们可以设置的超参数——基本上这个参数就是训练数据集中最长句子的长度。

对输入序列完成嵌入操作后，每个词都会流经编码器的两层。

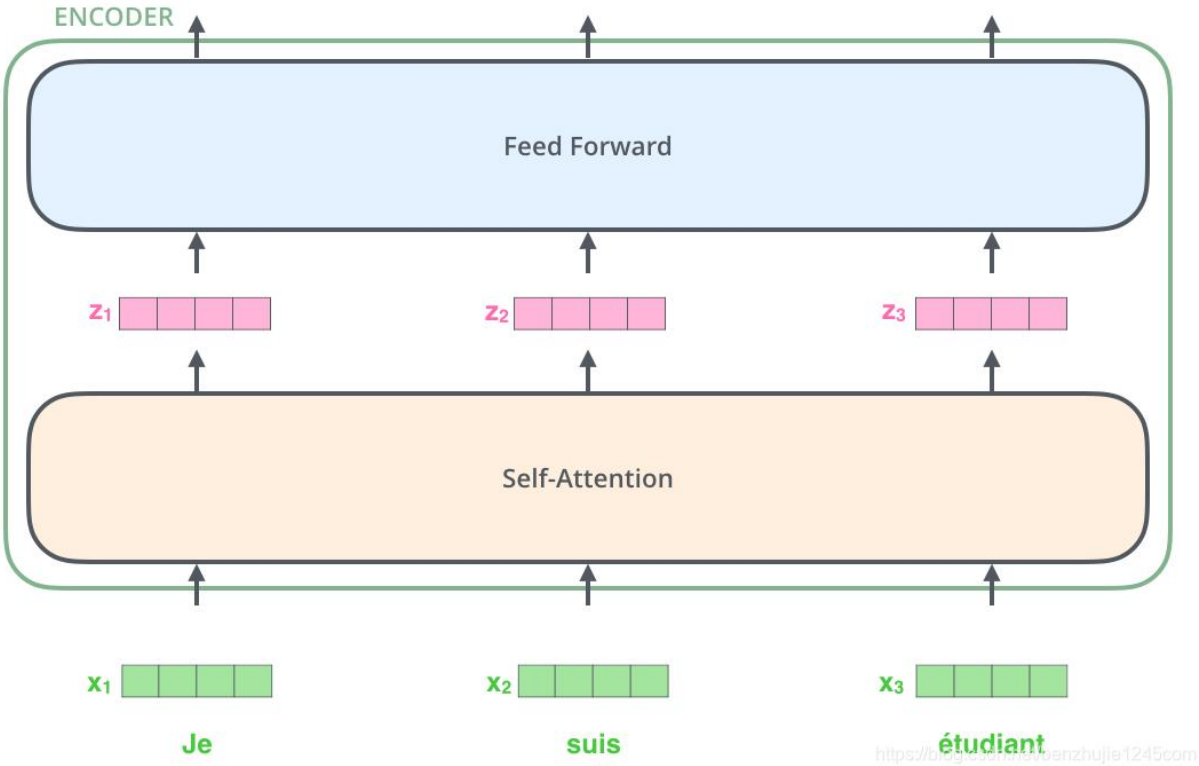

图 1.8

接下来，我们将换一个更短的句子作为示例，来说明在编码器的每个子层中发生了什么。

上面我们提到，编码器会接收一个向量作为输入。编码器首先将这些向量传递到 Self-Attention 层，然后传递到前馈网络，最后将输出传递到下一个编码

器。

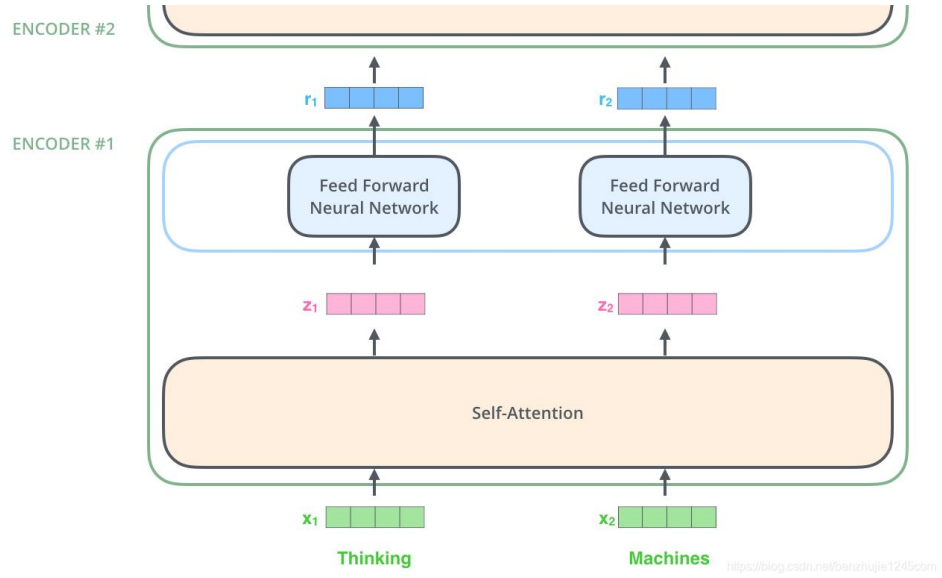

图 1.9

1.3 Self-Attention（自注意力）

1.3.1 Self-Attention 概览

首先我们通过一个例子，来对 Self-Attention 有一个直观的认识。假如，我们要翻译下面这个句子：

The animal didn’ t cross the street because it was too tired

这个句子中的 it 指的是什么？是指 animal 还是 street ？对人来说，这是一个简单的问题，但是算法来说却不那么简单。

当模型在处理 it 时，Self-Attention 机制使其能够将 it 和 animal 关联起来。

当模型处理每个词（输入序列中的每个位置）时，Self-Attention 机制使得模型不仅能够关注当前位置的词，而且能够关注句子中其他位置的词，从而可以更好地编码这个词。

如果你熟悉 RNN，想想如何维护隐状态，使 RNN 将已处理的先前词/向量的表示与当前正在处理的词/向量进行合并。Transformer 使用 Self-Attention 机制将其他词的理解融入到当前词中。

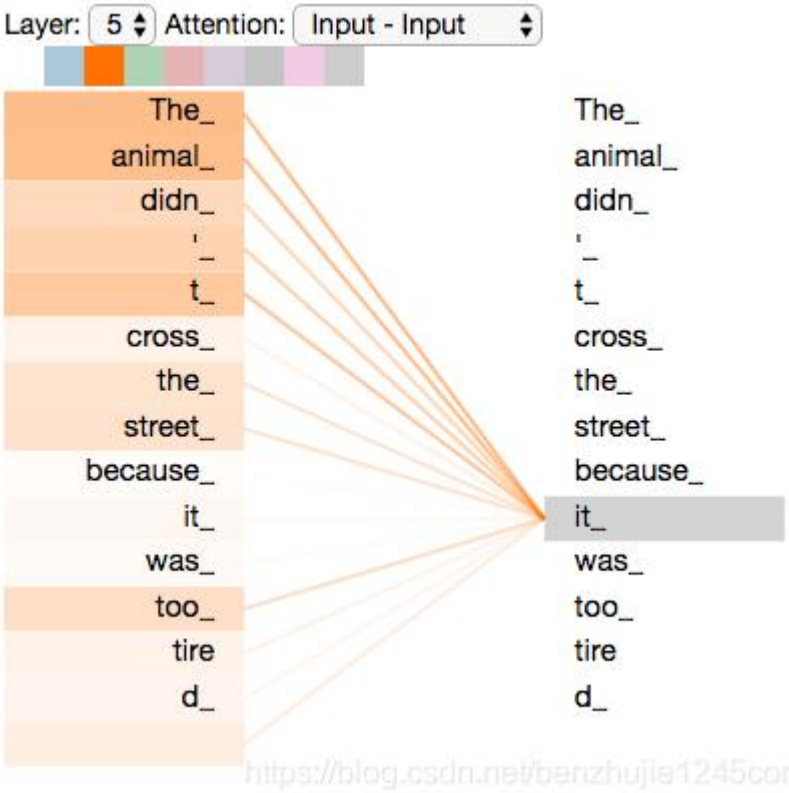

图 1.10 当我们在编码器 #5（堆栈中的顶部编码器）中对单词”it“进行编码时，有一部分注意力集中在”The animal“上，并将它们的部分信息融入到”it“的编码中。

1.3.2 Self-Attention 机制

下面我们来看一下 Self-Attention 的具体机制。其基本结构如图 1.11 所

示:

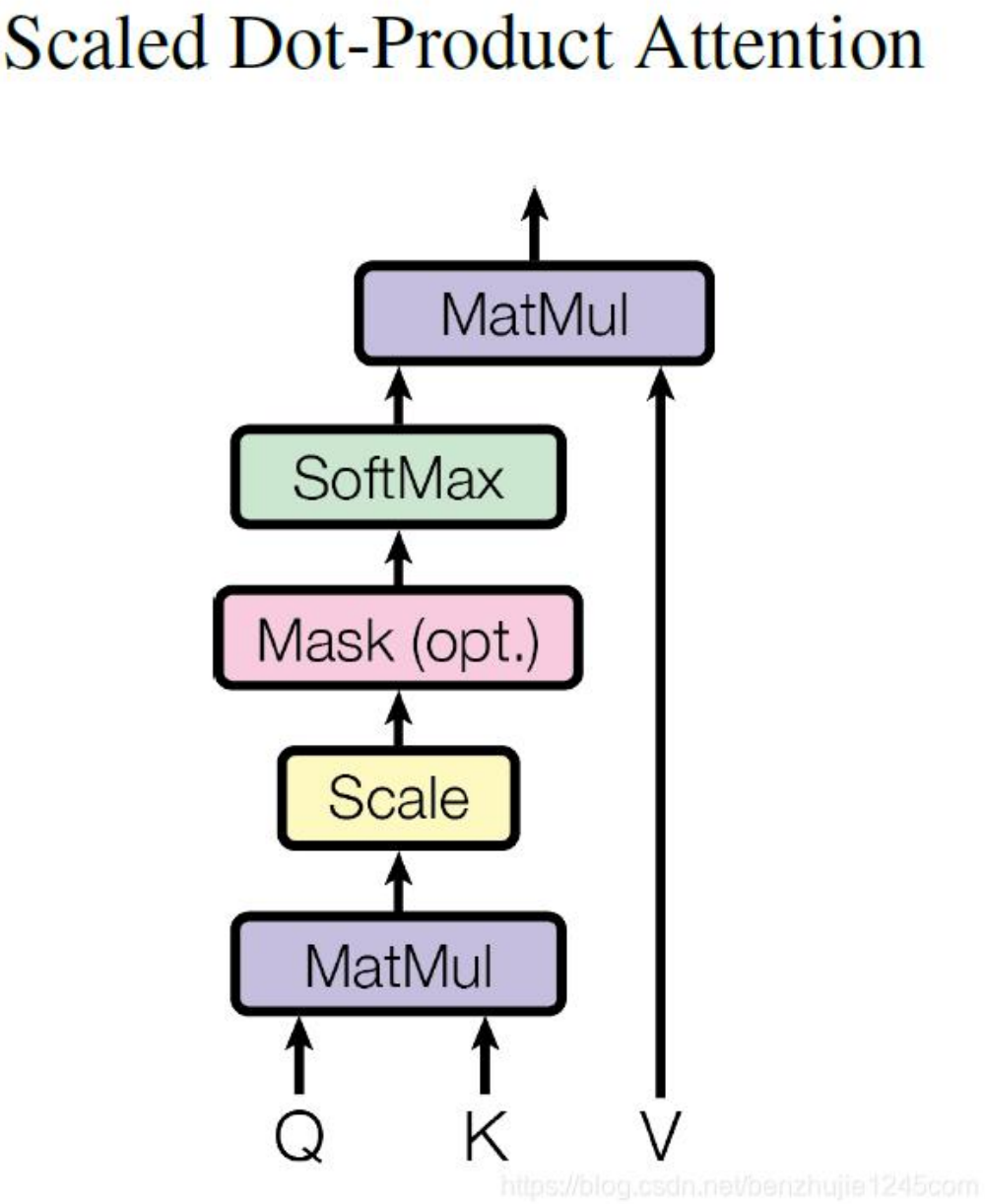

图 1.11 Scaled Dot-Product Attention（缩放点积注意力）

对于 Self Attention 来讲，Q（Query），K（Key）和 V（Value）三个矩阵均来自同一输入，并按照以下步骤计算：

首先计算 Q 和 K 之间的点积，为了防止其结果过大，

会除以 $\sqrt{d_k}$，其中 d_k 为 Key 向量的维度。

然后利用 Softmax 操作将其结果归一化为概率分布，再乘以矩阵 V 就得到权重求和的表示。

整个计算过程可以表示为：

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

为了更好的理解 Self-Attention，下面我们通过具体的例子进行详细说明。

1.3.3 Self-Attention 详解

下面通过一个例子，让我们看一下如何使用向量计算 Self-Attention。计算 Self-Attention 的步骤如下：

第 1 步：对编码器的每个输入向量（在本例中，即每个词的词向量）创建三个向量：Query 向量、Key 向量和 Value 向量。它们是通过词向量分别和 3 个矩阵相乘得到的，这 3 个矩阵通过训练获得。

请注意，这些向量的维数小于词向量的维数。新向量的维数为 64，而 embedding 和编码器输入/输出向量的维数为 512。新向量不一定非要更小，这是为了使多头注意力计算保持一致的结构性选择。

AI 大模型入门路线，PDF+课件资料包已全部备好，需要的扫码添加，我会发给你的~

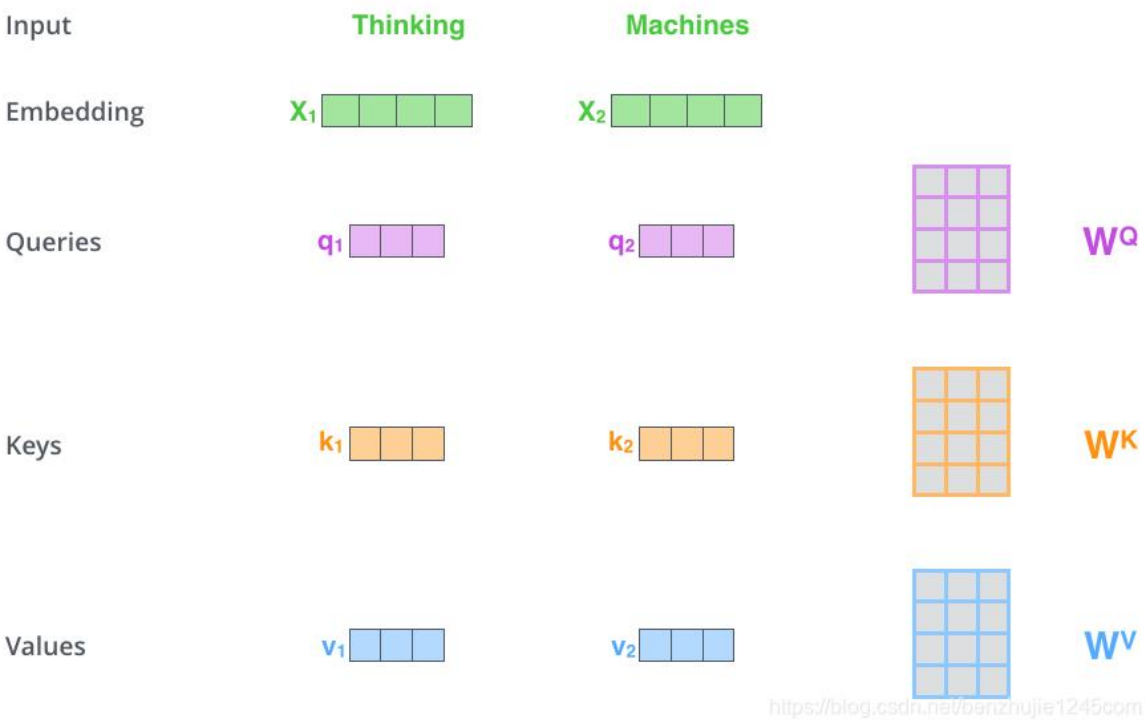

图 1.12

图 1.12 中，x_1 乘以权重矩阵 W^Q 得到 q_1，即与该单词关联的 Query 向量。最终会为输入句子中的每个词创建一个 Query，一个 Key 和一个 Value 向量。

什么是 Query，Key 和 Value 向量?

它们是一种抽象，对于注意力的计算和思考非常有用。继续阅读下面的注意力计算过程，你将了解这些向量所扮演的角色。

第 2 步：计算注意力分数。假设我们正在计算这个例子中第一个词 “Thinking” 的自注意力。我们需要根据 “Thinking” 这个词，对句子中的每个词都计算一个分数。这些分数决定了我们在编码 “Thinking” 这个词时，需要对句子中其他位置的每个词放置多少的注意力。

这些分数，是通过计算 “Thinking” 的 Query 向量和需要评分的词的 Key 向量的点积得到的。如果我们计算句子中第一个位置词的注意力分数，则第一个分数是 q_1 和 k_1 的点积，第二个分数是 q_1 和 k_2 的点积。

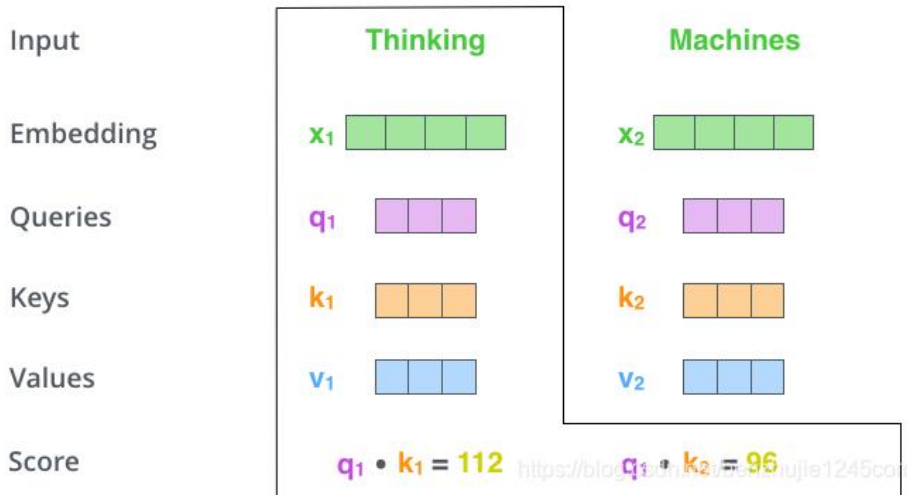

图 1.13

第 3 步：将每个分数除以 $\sqrt{d_k}$（d_k 是 Key 向量的维度）。目的是在反向传播时，求梯度更加稳定。实际上，你也可以除以其他数。

第 4 步：将这些分数进行 Softmax 操作。Softmax 将分数进行归一化处理，使得它们都为正数并且和为 1。

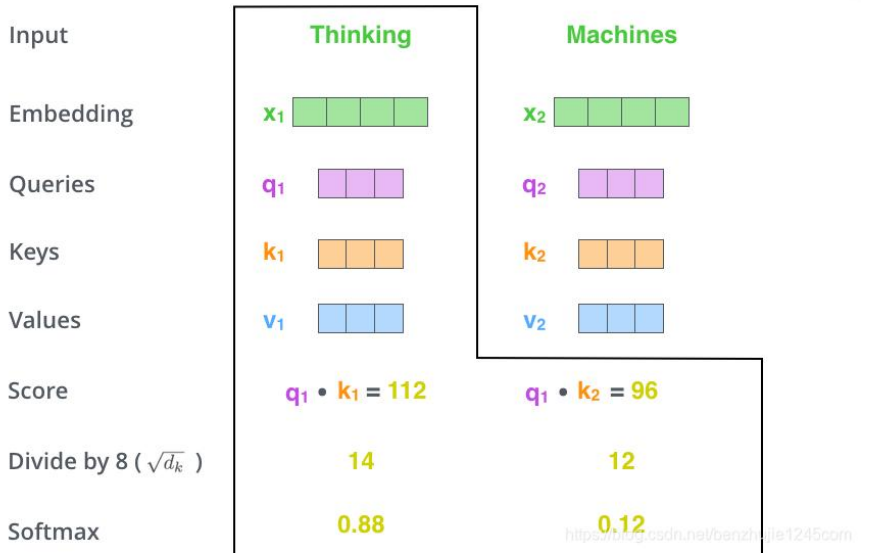

图 1.14

这些 Softmax 分数决定了在编码当前位置的词时，对所有位置的词分别有多少的注意力。很明显，当前位置的词汇有最高的分数，但有时注意一下与当前位置的词相关的词是很有用的。

第 5 步：将每个 Softmax 分数分别与每个 Value 向量相乘。这种做法背后的直觉理解是：对于分数高的位置，相乘后的值就越大，我们把更多的注意力放在它们身上；对于分数低的位置，相乘后的值就越小，这些位置的词可能是相关性不大，我们就可以忽略这些位置的词。

第 6 步：将加权 Value 向量（即上一步求得的向量）求和。这样就得到了自注意力层在这个位置的输出。

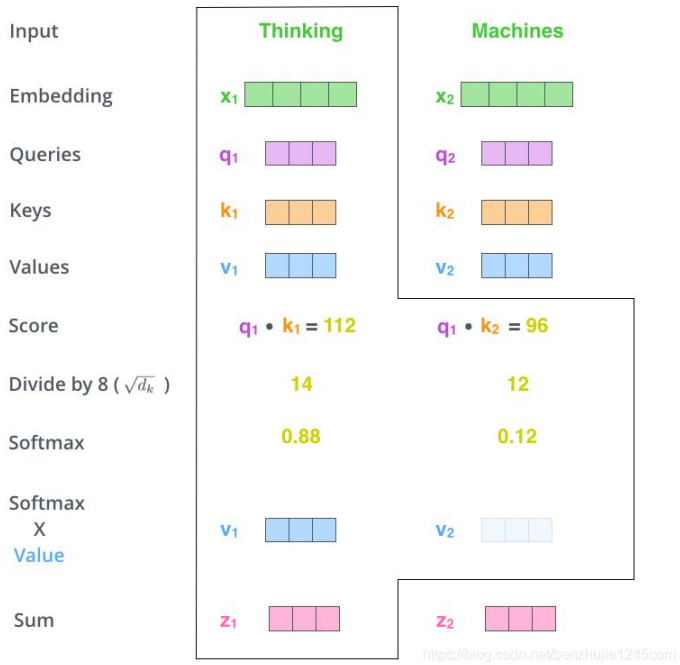

图 1.15

这样就完成了自注意力的计算。生成的向量会输入到前馈网络中。但是在实际实现中，此计算是以矩阵形式进行，以便实现更快的处理速度。下面我们来看看如何使用矩阵计算。

1.3.4 使用矩阵计算 Self-Attention

第 1 步：计算 Query，Key 和 Value 矩阵。首先，将所有词向量放到一个矩阵 X 中，然后分别和 3 个我们训练过的权重矩阵（W^Q，W^k 和 W^V）相乘，得到 $\mathbf{Q}$，K 和 V 矩阵。

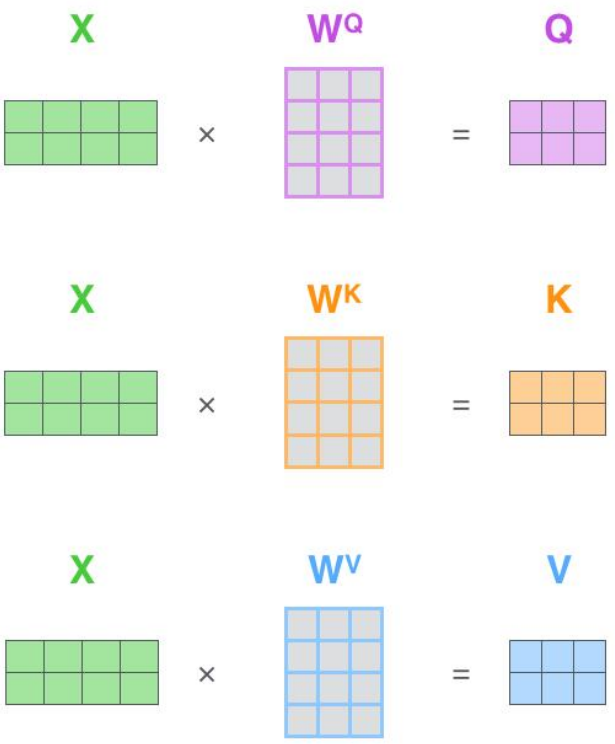

图 1.16 矩阵 X 中的每一行，表示输入句子中的每一个词的词向量（长度为 512，在图中为 4 个方框）。矩阵 Q，K 和 V 中的每一行，分别表示 Query 向量，Key 向量和 Value 向量（它们的长度都为 64，在图中为 3 个方框）。

第 2 步：计算自注意力。由于这里使用了矩阵进行计算，可以将前面的第 2 步到第 6 步压缩为一步。

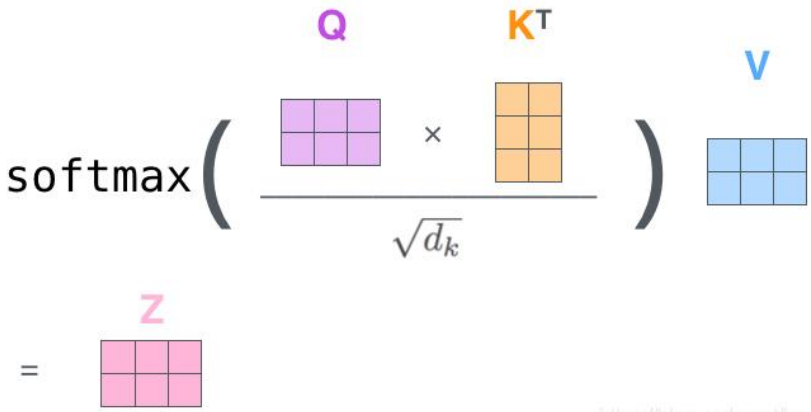

图 1.17 矩阵形式的自注意力计算

1.4 多头注意力机制（Multi-head Attention）

在 Transformer 论文中，通过添加一种多头注意力机制，进一步完善了自注意力层。具体做法：首先，通过 h hh 个不同的线性变换对 Query、Key 和 Value 进行映射；然后，将不同的 Attention 拼接起来；最后，再进行一次线性变换。基本结构如图 1.18 所示：

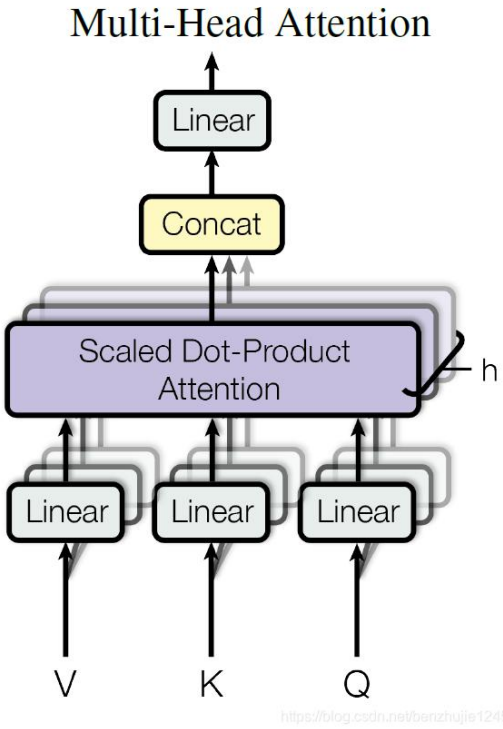

图 1.18

每一组注意力用于将输入映射到不同的子表示空间，这使得模型可以在不同子表示空间中关注不同的位置。整个计算过程可表示为：

每一组注意力用于将输入映射到不同的子表示空间，这使得模型可以在不同子表示空间中关注不同的位置。整个计算过程可表示为：

$$MultiHead(Q,K,V) = Concat(head_1,\cdots,head_h)\boldsymbol{W}^O$$
$$head_i = Attention(Q\boldsymbol{W}_i^Q, K\boldsymbol{W}_i^K, V\boldsymbol{W}_i^V)$$

其中，$\boldsymbol{W}_i^Q \in \mathbb{R}^{d_{model}\times d_k}$，$\boldsymbol{W}_i^K \in \mathbb{R}^{d_{model}\times d_k}$，$\boldsymbol{W}_i^V \in \mathbb{R}^{d_{model}\times d_v}$ 和 $\boldsymbol{W}^O \in \mathbb{R}^{hd_v\times d_{model}}$。在论文中，指定 $h=8$（即使用 8 个注意力头）和 $d_k = d_v = d_{model}/h = 64$。

在多头注意力下，我们为每组注意力单独维护不同的 Query、Key 和 Value 权重矩阵，从而得到不同的 Query、Key 和 Value 矩阵。如前所述，我们将 $\boldsymbol{X}$ 乘以 $\boldsymbol{W}^Q$、$\boldsymbol{W}^K$ 和 $\boldsymbol{W}^V$ 矩阵，得到 Query、Key 和 Value 矩阵。

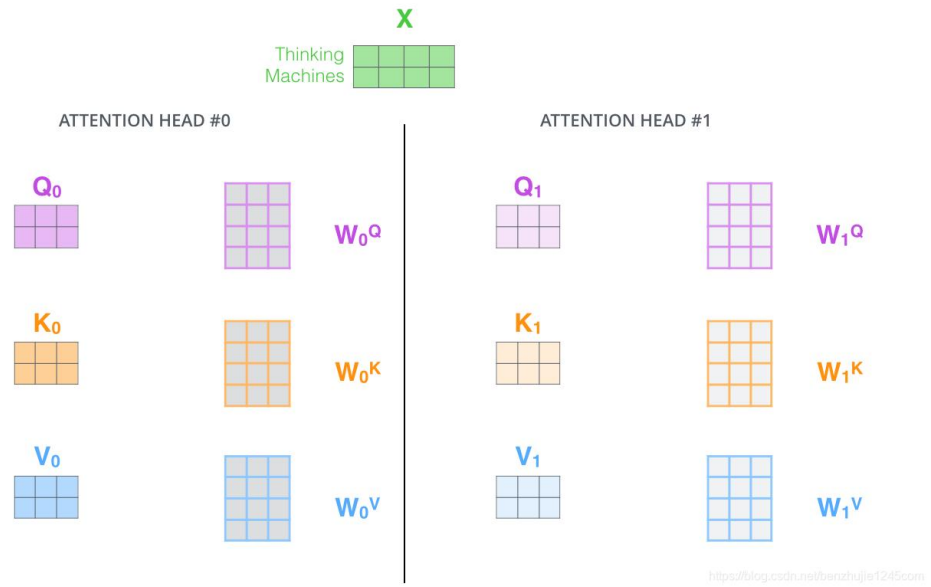

图 1.19

按照上面的方法，使用不同的权重矩阵进行 8 次自注意力计算，就可以得到 8 个不同的 Z \boldsymbol{Z} 矩阵。

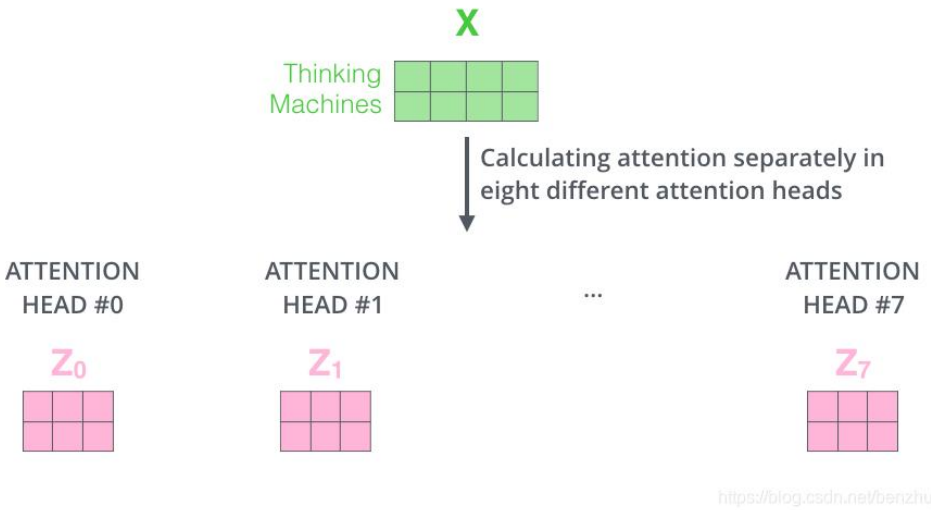

图 1.20

接下来就有点麻烦了。因为前馈神经网络层接收的是 1 个矩阵（每个词的词向量），而不是上面的 8 个矩阵。因此，我们需要一种方法将这 8 个矩阵整合为一个矩阵。具体方法如下：

1. 把 8 个矩阵 $\{Z_0, Z_1, \cdots, Z_7\}$ 拼接起来。
2. 把拼接后的矩阵和一个权重矩阵 W^O 相乘。
3. 得到最终的矩阵 Z，这个矩阵包含了所有注意力头的信息。这个矩阵会输入到 FFN 层。

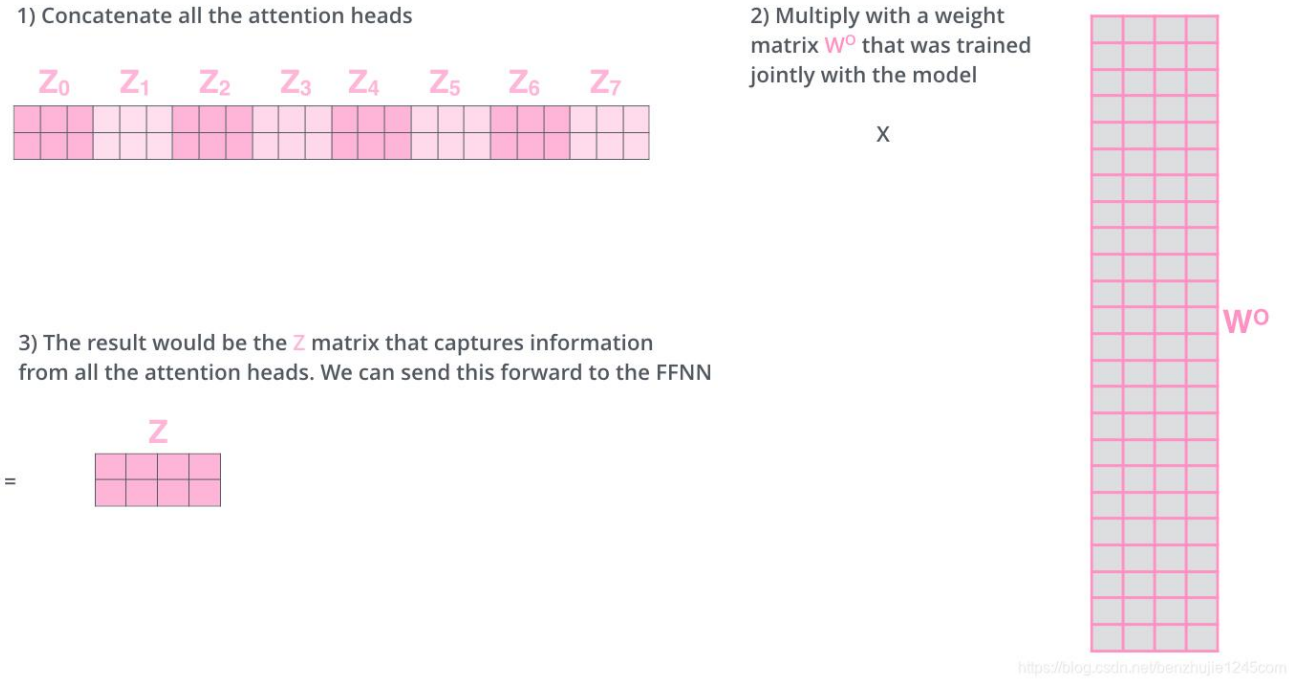

图 1.21

这差不多就是多头注意力的全部内容了。下面将所有内容放到一张图中，以便我们可以统一查看。

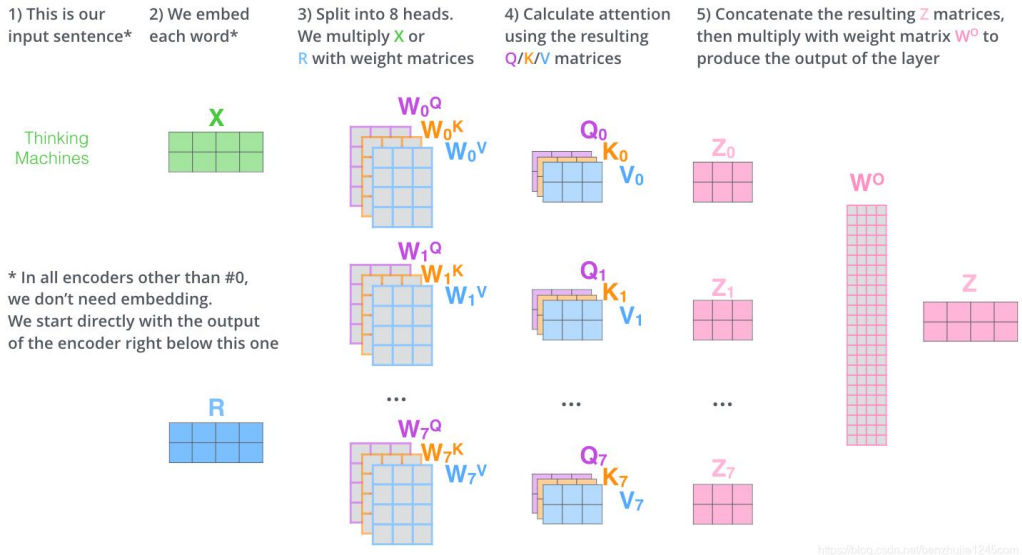

图 1.22

现在让我们重新回顾一下前面的例子，看看在对示例句中的“it”进行编码时，不同的注意力头关注的位置分别在哪：

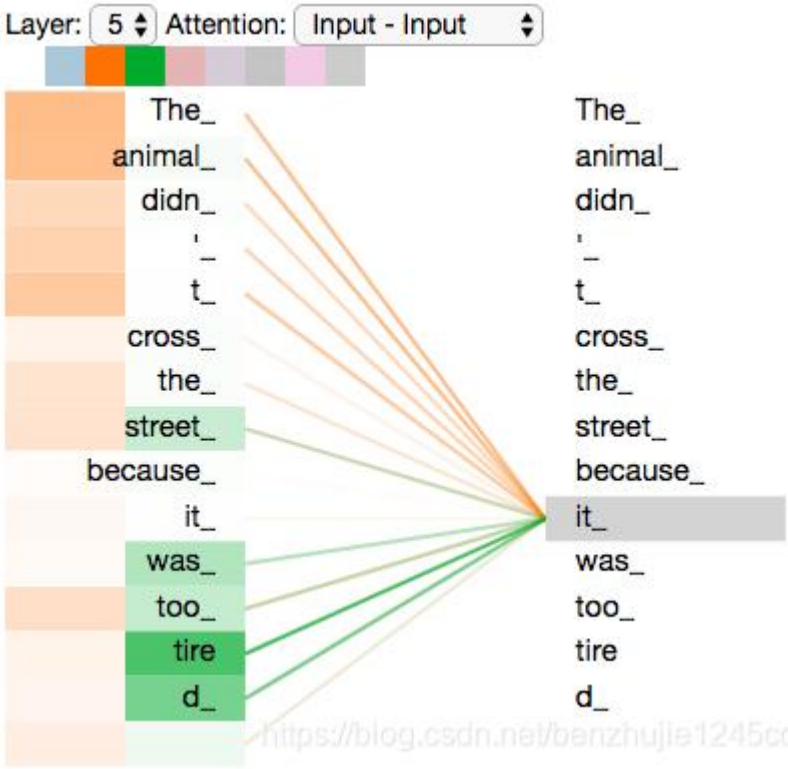

图 1.23

当我们对“it”进行编码时，一个注意力头关注“The animal”，另一个注意力头关注“tired”。从某种意义上来说，模型对“it”的表示，融入了“animal”和“tired”的部分表达。

Multi-head Attention 的本质是，在参数总量保持不变的情况下，将同样的 Query，Key，Value 映射到原来的高维空间的不同子空间中进行 Attention

的计算，在最后一步再合并不同子空间中的 Attention 信息。这样降低了计算每个 head 的 Attention 时每个向量的维度，在某种意义上防止了过拟合；由于 Attention 在不同子空间中有不同的分布，Multi-head Attention 实际上是寻找了序列之间不同角度的关联关系，并在最后拼接这一步骤中，将不同子空间中捕获到的关联关系再综合起来。

1.5 位置前馈网络（Position-wise Feed-Forward Networks）

位置前馈网络就是一个全连接前馈网络，每个位置的词都单独经过这个完全相同的前馈神经网络。其由两个线性变换组成，即两个全连接层组成，第一个全连接层的激活函数为 ReLU 激活函数。可以表示为：

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

在每个编码器和解码器中，虽然这个全连接前馈网络结构相同，但是不共享参数。整个前馈网络的输入和输出维度都是 $d_{model} = 512$，第一个全连接层的输出和第二个全连接层的输入维度为 $d_{ff} = 2048$。

1.6 残差连接和层归一化

编码器结构中有一个需要注意的细节：每个编码器的每个子层（Self-Attention 层和 FFN 层）都有一个残差连接，再执行一个层标准化操作，整个计算过程可以表示为：

$$sub\_layer\_output = LayerNorm(x + SubLayer(x))$$

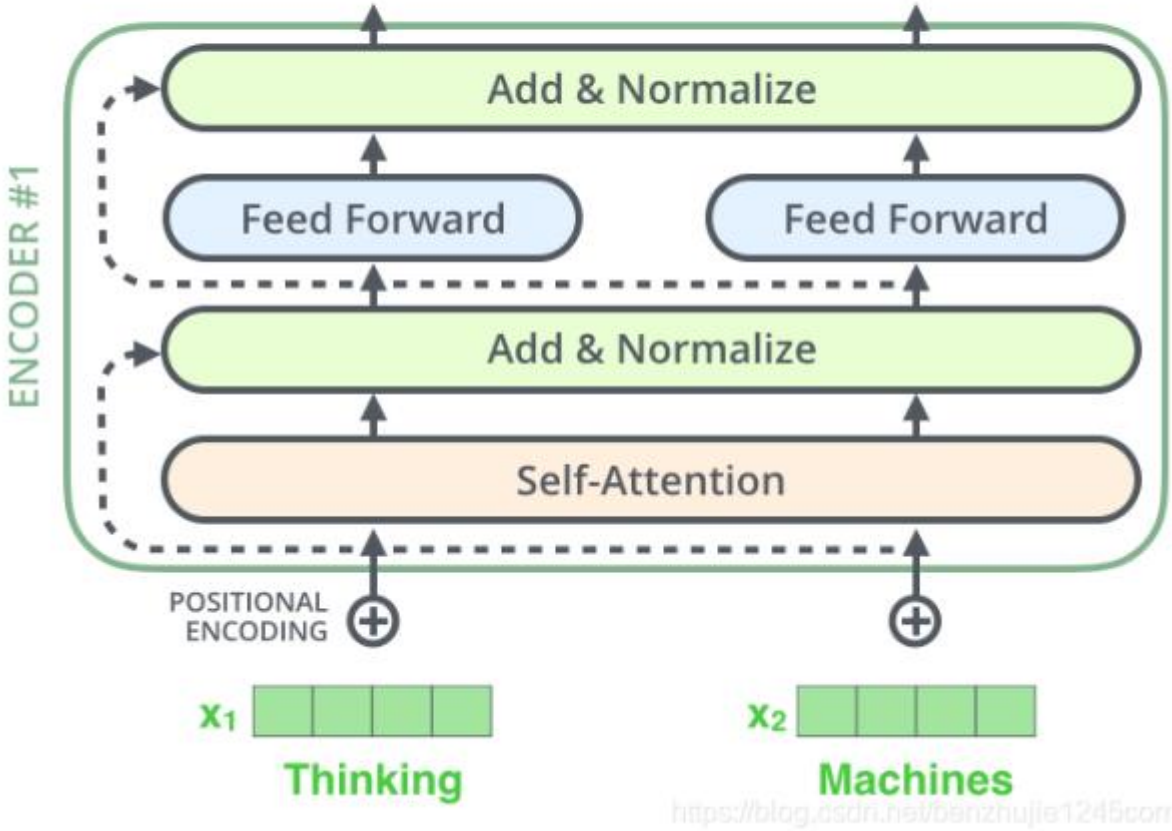

图 1.24

将向量和自注意力层的层标准化操作可视化，如下图所示：

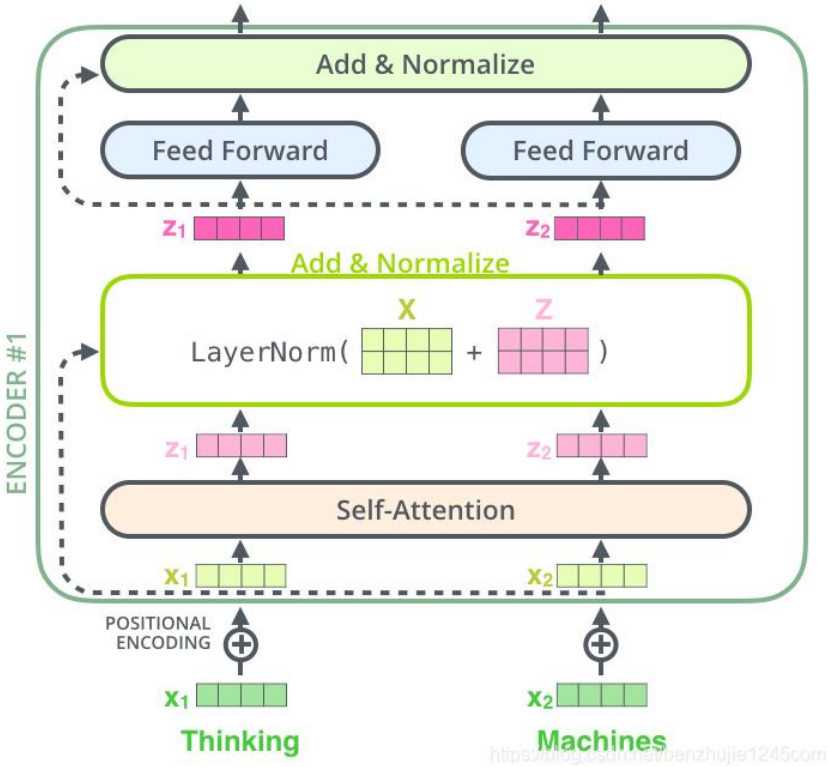

图 1.25

上面的操作也适用于解码器的子层。假设一个 Transformer 是由 2 层编码器和 2 层解码器组成，其如下图所示：

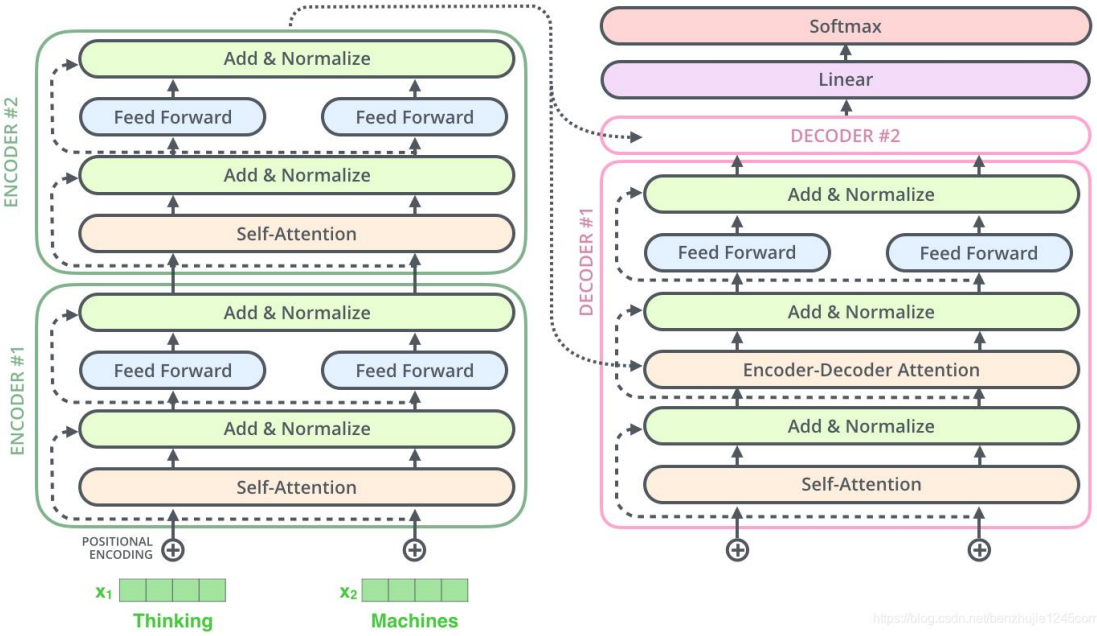

图 1.26

为了方便进行残差连接，编码器和解码器中的所有子层和嵌入层的输出维度需要保持一致，在 Transformer 论文中 $d_{model} = 512$ d_{model}

=512。

1.7 位置编码

到目前为止，我们所描述的模型中缺少一个东西：表示序列中词顺序的方法。为了解决这个问题，Transformer 模型为每个输入的词嵌入向量添加一个向量。这些向量遵循模型学习的特定模式，有助于模型确定每个词的位置，或序列中不同词之间的距离。

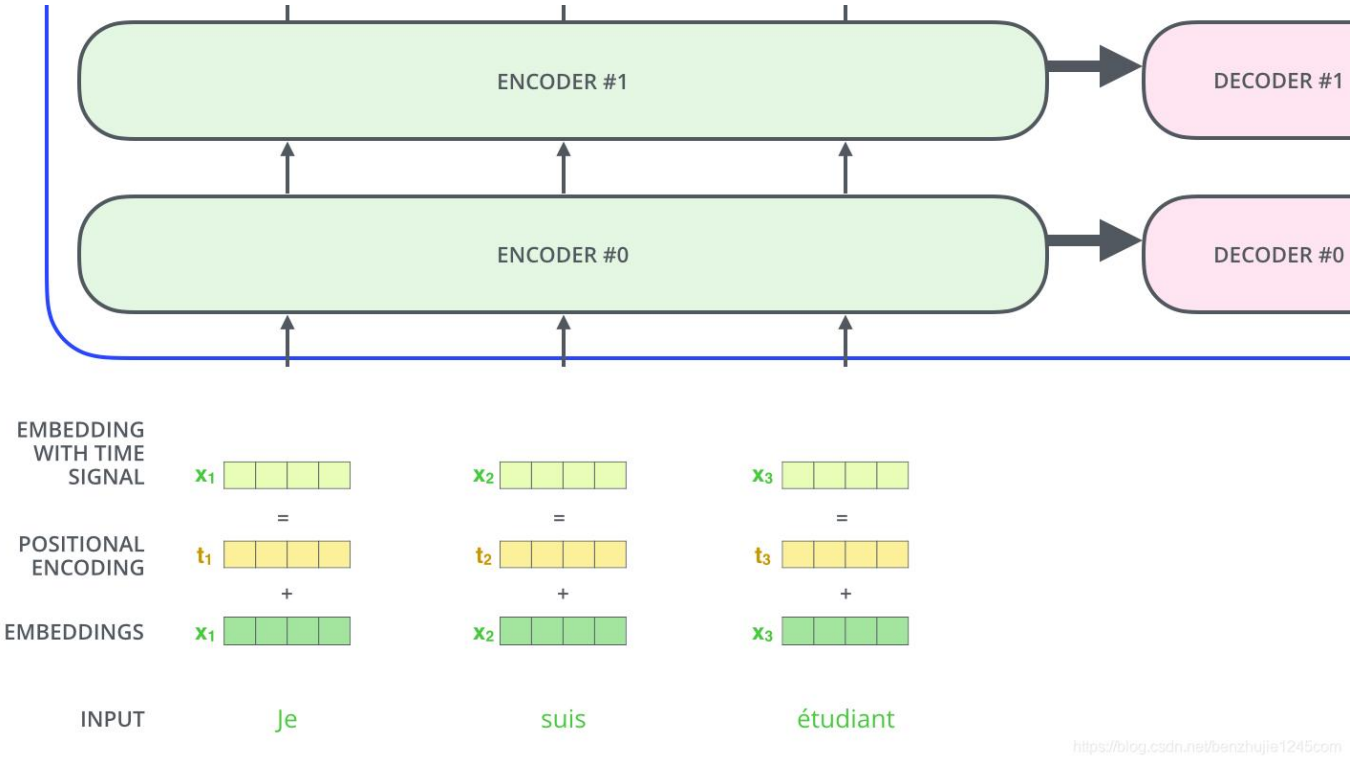

图 1.27

如果我们假设词嵌入向量的维度是 4，那么实际的位置编码如下：

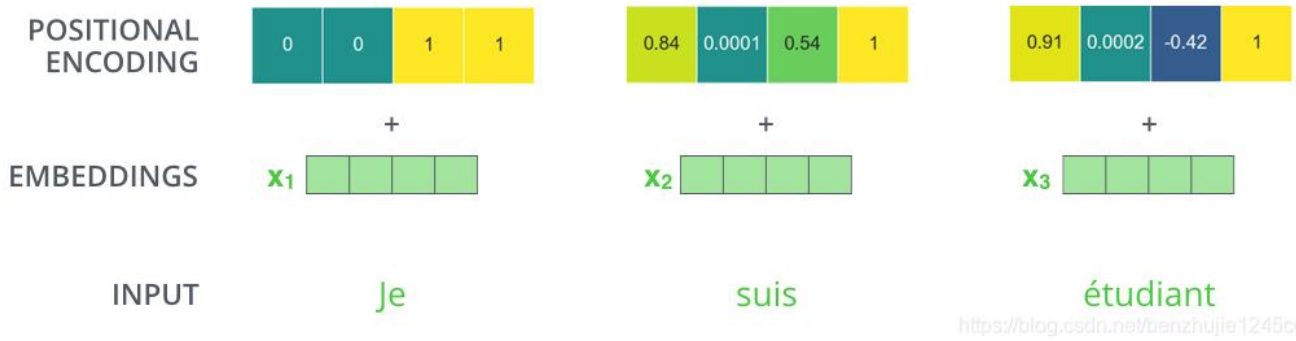

图 1.28

那么位置编码向量到底遵循什么模式？其具体的数学公式如下：

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

其中，pos 表示位置，i 表示维度。上面的函数使得模型可以学习到 token 之间的相对位置关系：任意位置的 $PE_{(pos+k)}$ 都可以被 $PE_{(pos)}$ 的线性函数表示：

$$\cos(\alpha+\beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)$$

$$\sin(\alpha+\beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)$$

在下图中，我们将这些值进行可视化。每一行对应一个向量的位置编码。所以第一行对应于输入序列中第一个词的位置编码。每一行包含 64 个值，每个值的范围在 -1 和 1 之间。

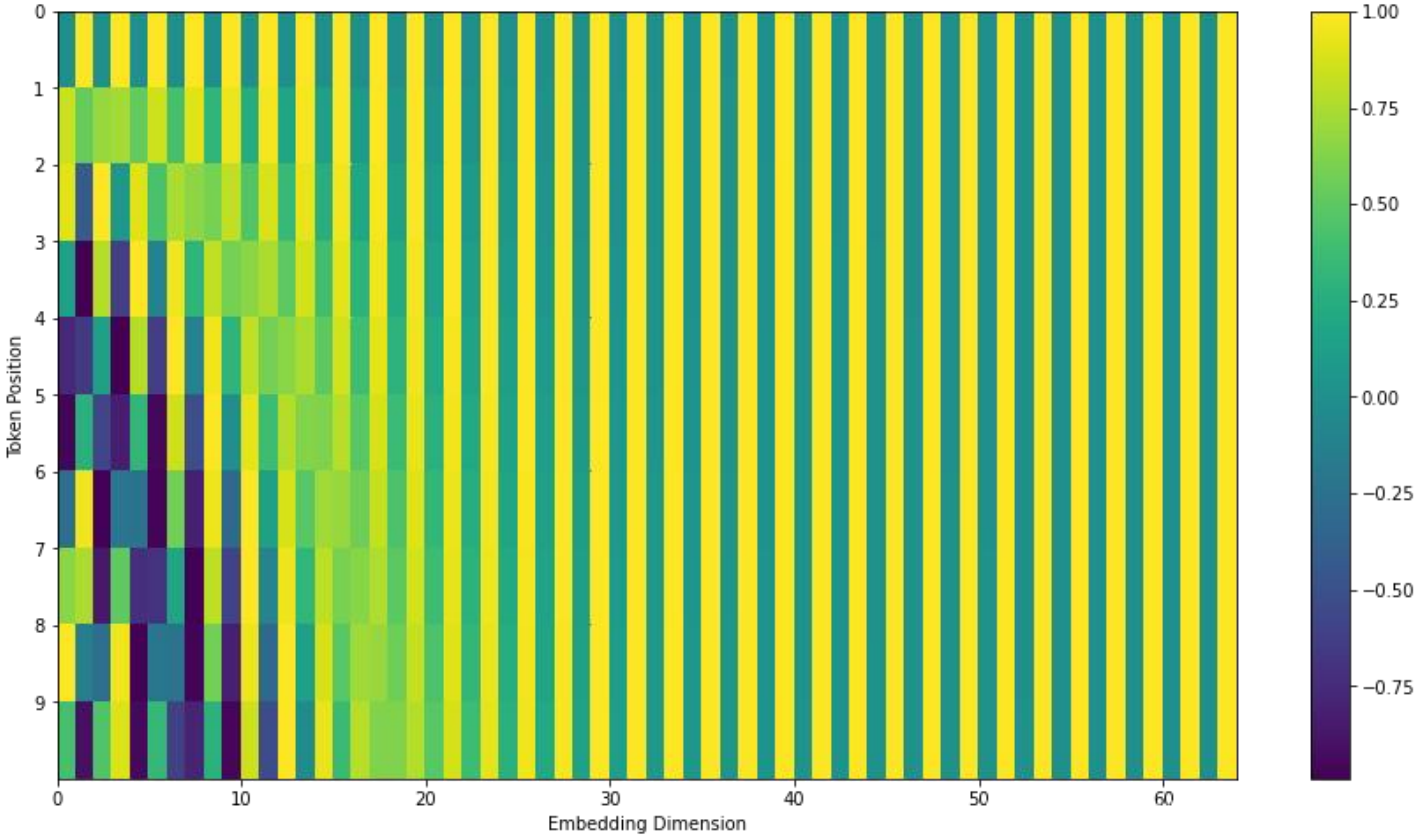

图 1.29

需要注意的是，官方提供的示例代码（TensorFlow 1.x 版本 中的 get_timing_signal_1d() 函数和 TensorFlow 2.x 版本 中的 call() 函数）与 Transformer 论文中的方法稍微存在一定差异：Transformer 论文中，sine 函数和 cosine 函数产生的值交织在一起；而官方提供的代码中，左半部分的值全是由 sine 函数产生的，右半部分的值全是由 cosine 函数产生的，然后将它们拼接起来。官方代码生成的位置编码值的可视化图如下：

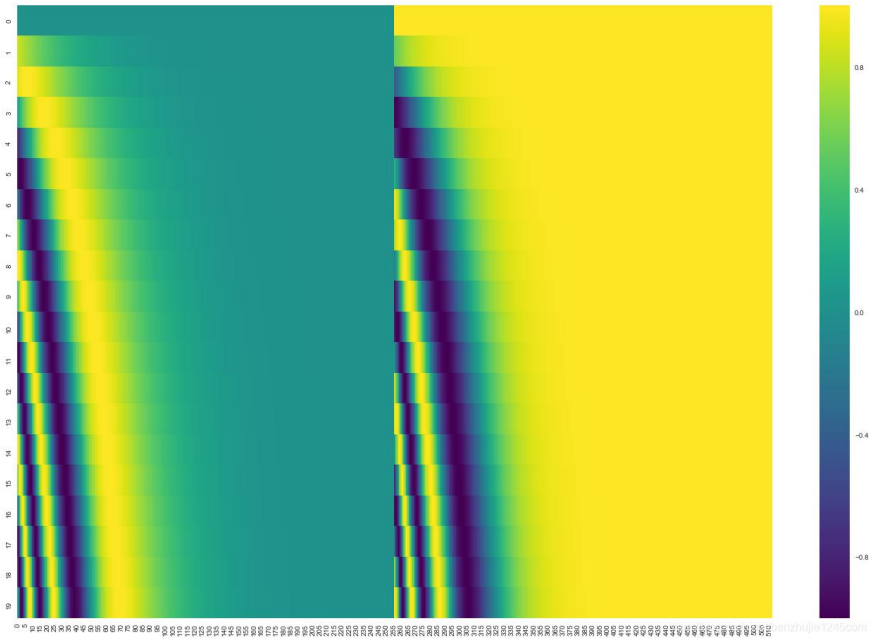

图 1.30

这不是唯一一种生成位置编码的方法。但这种方法的优点是：可以扩展到未知的序列长度。例如，当我们训练后的模型被要求翻译一个句子，而这个句子的长度大于训练集中所有句子的长度。

1.8 解码器

现在我们已经介绍了编码器的大部分概念，我们也了解了解码器的组件的原理。现在让我们看下编码器和解码器是如何协同工作的。

通过上面的介绍，我们已经了解第一个编码器的输入是一个序列，最后一个编码器的输出是一组注意力向量 Key 和 Value。这些向量将在每个解码器的 Encoder-Decoder Attention 层被使用，这有助于解码器把注意力集中在输入序列的合适位置。

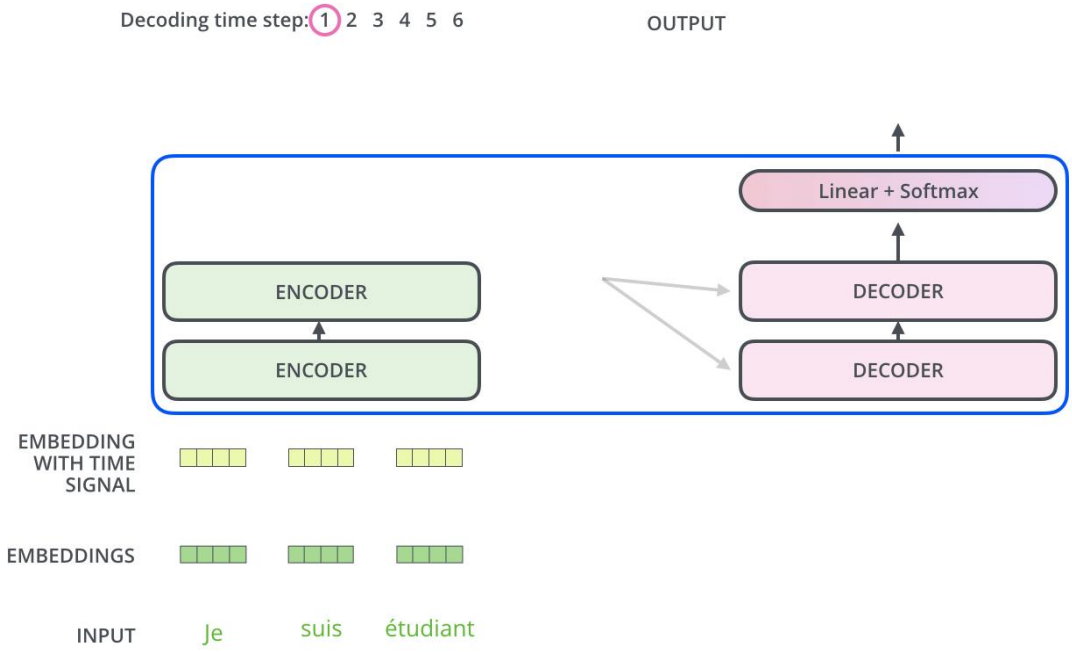

图 1.31

在完成了编码阶段后，我们开始解码阶段。解码阶段的每个时间步都输出一个元素。

接下来会重复这个过程，直到输出一个结束符，表示 Transformer 解码器已完成其输出。每一步的输出都会在下一个时间步输入到下面的第一个解码器，解码器像编码器一样将解码结果显示出来。就像我们处理编码器输入一样，我们也为解码器的输入加上位置编码，来指示每个词的位置。

Encoder-Decoder Attention 层的工作原理和多头自注意力机制类似。不同之处是：Encoder-Decoder Attention 层使用前一层的输出构造 Query 矩阵，而 Key 和 Value 矩阵来自于编码器栈的输出。

1.9 Mask（掩码）

Mask 表示掩码，它对某些值进行掩盖，使其在参数更新时不产生效果。Transformer 模型里面涉及两种 mask，分别是 Padding Mask 和 Sequence Mask。其中，Padding Mask 在所有的 scaled dot-product attention 里面都需要用到，而 Sequence Mask 只有在 Decoder 的 Self-Attention 里面用到。

1.9.1 Padding Mask

什么是 Padding mask 呢？因为每个批次输入序列的长度是不一样的，所以我们要对输入序列进行对齐。具体来说，就是在较短的序列后面填充 0（但是如果输入的序列太长，则是截断，把多余的直接舍弃）。因为这些填充的位置，其实是没有什么意义的，所以我们的 Attention 机制不应该把注意力放在这些位置上，所以我们需要进行一些处理。

具体的做法：把这些位置的值加上一个非常大的负数（负无穷），这样的话，经过 Softmax 后，这些位置的概率就会接近 0。

1.9.2 Sequence Mask

Sequence Mask 是为了使得 Decoder 不能看见未来的信息。也就是对于一个序列，在 t tt 时刻，我们的解码输出应该只能依赖于 t tt 时刻之前的输出，而不能依赖 t tt 之后的输出。因为我们需要想一个办法，把 t tt 之后的信息给隐藏起来。

具体的做法：产生一个上三角矩阵，上三角的值全为 0。把这个矩阵作用在每个序列上，就可以达到我们的目的。

总结：对于 Decoder 的 Self-Attention，里面使用到的 scaled dot-product attention，同时需要 Padding Mask 和 Sequence Mask，具体实现就是两个 Mask 相加。其他情况下，只需要 Padding Mask。

1.10 最后的线性层和 Softmax 层

解码器栈的输出是一个 float 向量。我们怎么把这个向量转换为一个词呢？通过一个线性层再加上一个 Softmax 层实现。

线性层是一个简单的全连接神经网络，其将解码器栈的输出向量映射到一个更长的向量，这个向量被称为 logits 向量。

现在假设我们的模型有 10000 个英文单词（模型的输出词汇表）。因此 logits 向量有 10000 个数字，每个数表示一个单词的分数。

然后，Softmax 层会把这些分数转换为概率（把所有的分数转换为正数，并且加起来等于 1）。最后选择最高概率所对应的单词，作为这个时间步的输出。

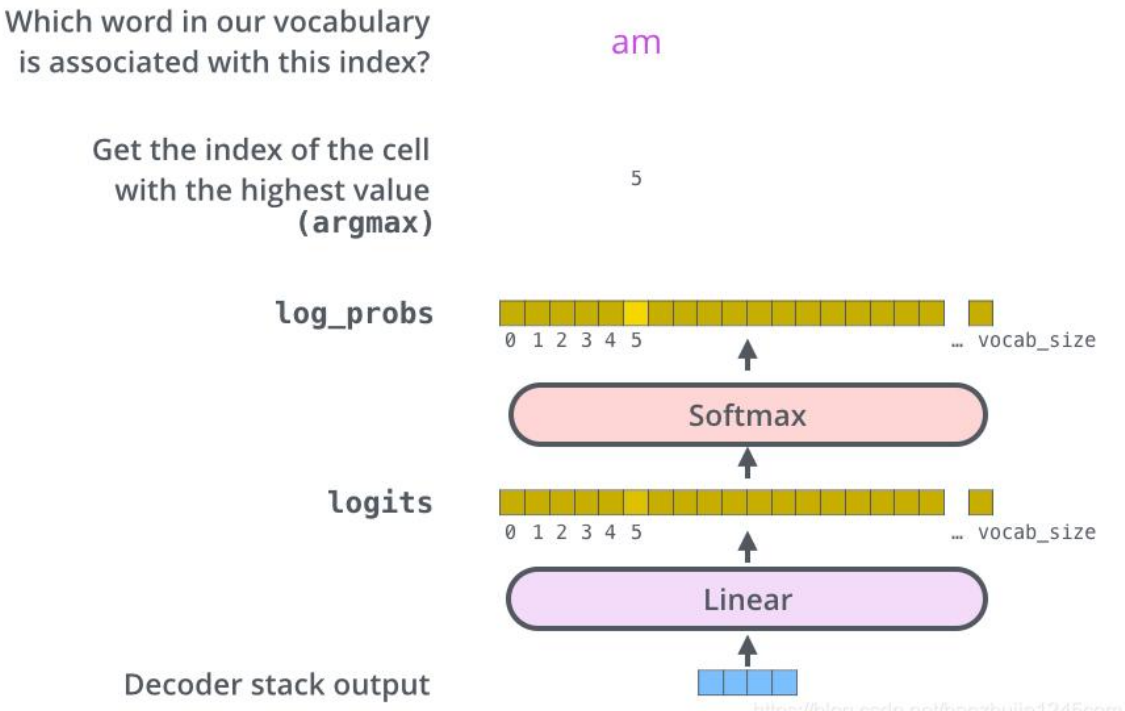

图 1.33

1.11 嵌入层和最后的线性层

在 Transformer 论文，提到一个细节：编码组件和解码组件中的嵌入层，以及最后的线性层共享权重矩阵。不过，在嵌入层中，会将这个共享权重矩阵乘以 $\sqrt{d_{model}}$ 。

1.12 正则化操作

为了提高 Transformer 模型的性能，在训练过程中，使用了以下的正则化操作：

1. Dropout。对编码器和解码器的每个子层的输出使用 Dropout 操作，是在进行残差连接和层归一化之前。词嵌入向量和位置编码向量执行相加操作后，执行 Dropout 操作。Transformer 论文中提供的参数 $P_{drop} = 0.1$。
2. Label Smoothing（标签平滑）。Transformer 论文中提供的参数 $\epsilon_{ls} = 0.1$.