今天阅读的来自谷歌大脑的同学于 2017 年发表的论文《Attention Is All You Need》，目前论文被引次数高达 6100 次。

Attention 机制是 Bengio 等同学在 2014 年提出的，并广泛应用于深度学习各个领域，如计算机视觉CV、自然语言处理NLP 等。其中，Seq2Seq 模型采用了 RNN 和 Attention 的结合成功应用于机器翻译领域，在诸多任务中都有显著的提升。

在这篇文论文中，作者提出了 Transformer 网络架构，其摒弃了传统的 RNN、LSTM 架构，完全基于 Attention 机制，并在机器翻译领域获得明显的质量提升。

1. Introduction

传统的基于 RNN 的 Seq2Seq 模型由于难以处理长序列的的句子，且因顺序性也无法并行处理。

而完全基于 CNN 的 Seq2Seq 模型虽然可以实现并行化，但是非常耗内存。

Self-attention，也称为 intra-attention，是一种将序列的不同位置联系起来并计算序列表示的注意力机制。Self-attention 已成功应用于各个任务中，包括阅读理解、摘要生成、句子表示等任务中。

而本文介绍的 Transformer 是一个完全使用 Self-attention 的模型，即解决了计算量和并行效率的问题，又提高了实验的结果。

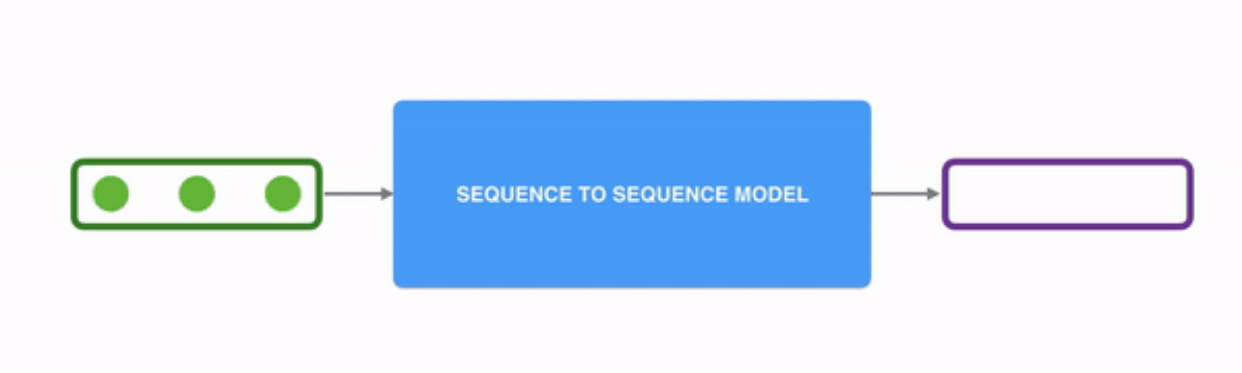
2. Pre-requisites首要事

2.1 Seq2Seq

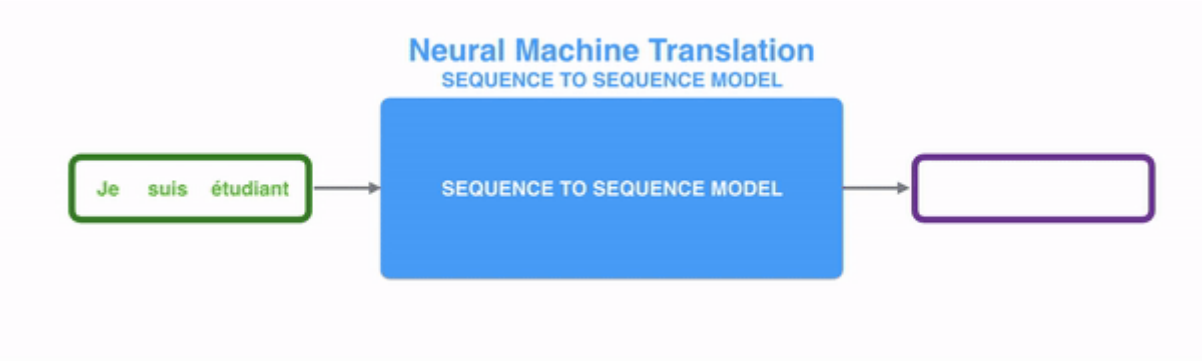
Sequence-to-sequence 模型（以下简称 Seq2Seq）是一种深度学习模型，Seq2Seq 的应用广泛，常应用于机器翻译，语音识别，自动问答等领域。谷歌翻译也在 2016 开始使用这个模型。

接下来介绍的 Seq2Seq 是没加 Attention 的传统 Seq2Seq，而我们现在经常说的 Seq2Seq 是加了 Attention 的模型。

Seq2Seq 可以理解为输入一个序列，然后经过一个黑盒后可以得到另一个序列：

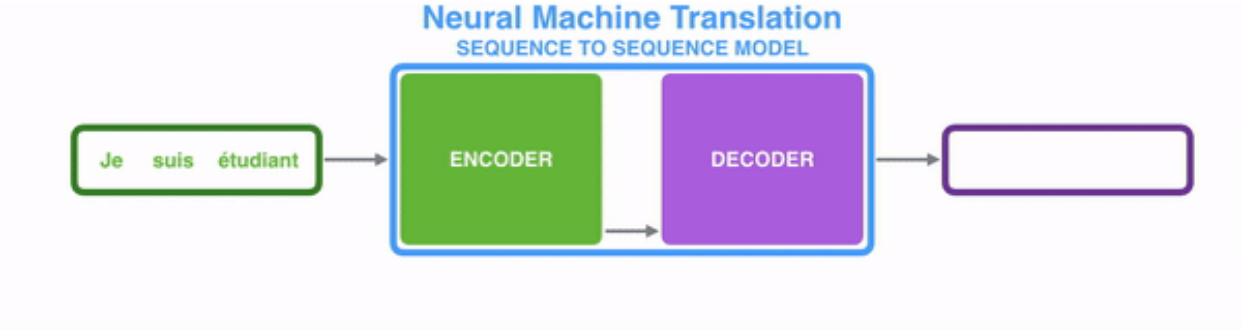


如果将 Seq2Seq 应用于机器翻译领域的话，就是输入一种语言，然后得到另一个语言：



而这个黑盒中，其实就是 Encoder-Decoder 框架。

大概可以可以理解为输入一个**序列**，然后编码器进行编码得到一个**上下文信息 C**，然后通过解码器进行**逐一解码**，最后得到另一个**序列**。



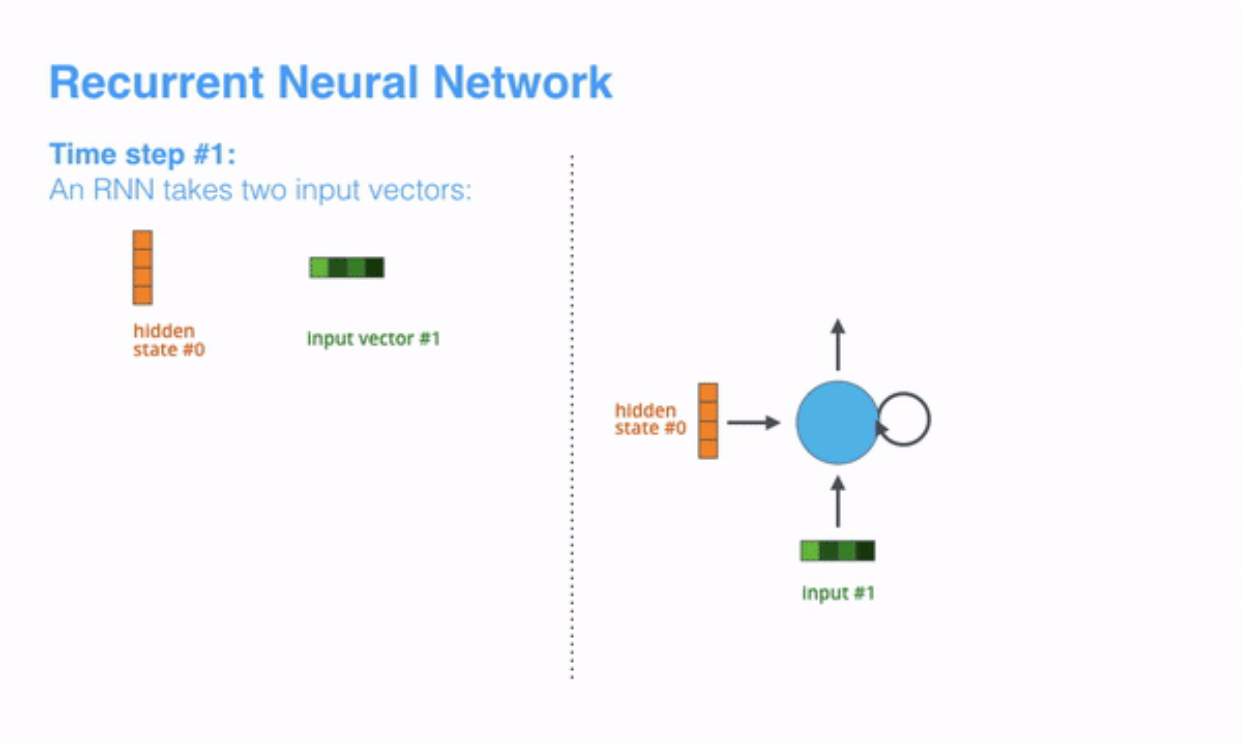
这边我们需要注意几点：

- **输入/输出序列**：输入输出序列都是 Embedding 向量；
- **上下文信息Context**：上下文信息 C 是一个向量，其维度与编码器的数量有关，通常大小为 256、512、1024 等。
- **逐一解码**：解码器需要根据上下文 C 和先前生成了历史信息 来生成此刻的 。

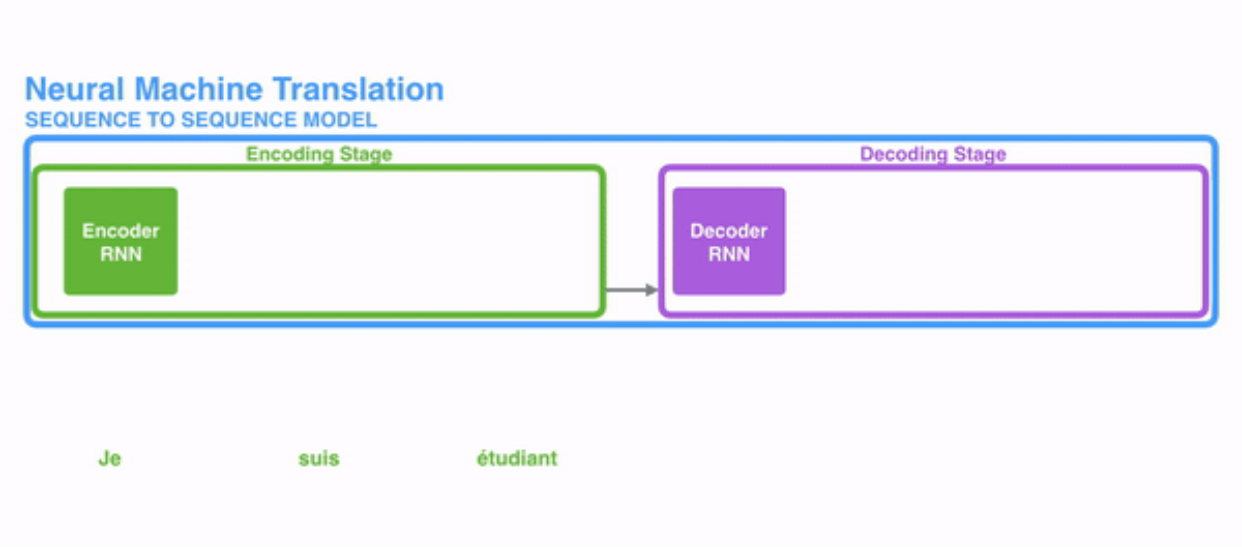
这里的编码器和解码器根据不同的模型和应用都是可以自由变换和组合的，常见的有 CNN/RNN/LSTM/GRU 等。

而 Seq2Seq 使用的是 RNN 模型。

我们知道 RNN 模型需要两个输入，并且有两个输出：



所以在编码器之间进行传递的其实隐藏层的状态。大概的工作过程为：



注意，编码器中最后一个 RNN 的隐藏状态就是要传给解码器的上下文信息 Context。

2.2 Attention

对于使用 Encoder-Decoder 框架的模型来说，上下文信息 C 的质量决定着是模型性能。

而 Seq2Seq 采用的 RNN 有一个致命的缺陷：由于反向传播会随着网络层数的加深而出现梯度爆炸或者梯度消失，所以无法处理长序列

问题。

为了解决这个问题，IUB 大学的同学和斯坦福大学的同学分别于 2014 年和 2015 年提出一种解决方案。这两篇论文分别介绍和完善了一种叫 Attention 的技术，极大地提高了机器翻译系统的质量。

Attention 允许模型根据需要来关注输入序列的某些相关部分。这种机制类似于人类的注意力，在看图片或者阅读时，我们会特别关注某一个焦点。Attention 应用广泛，如 CV 领域的感受野，NLP 领域的关键 Token 定位等等。

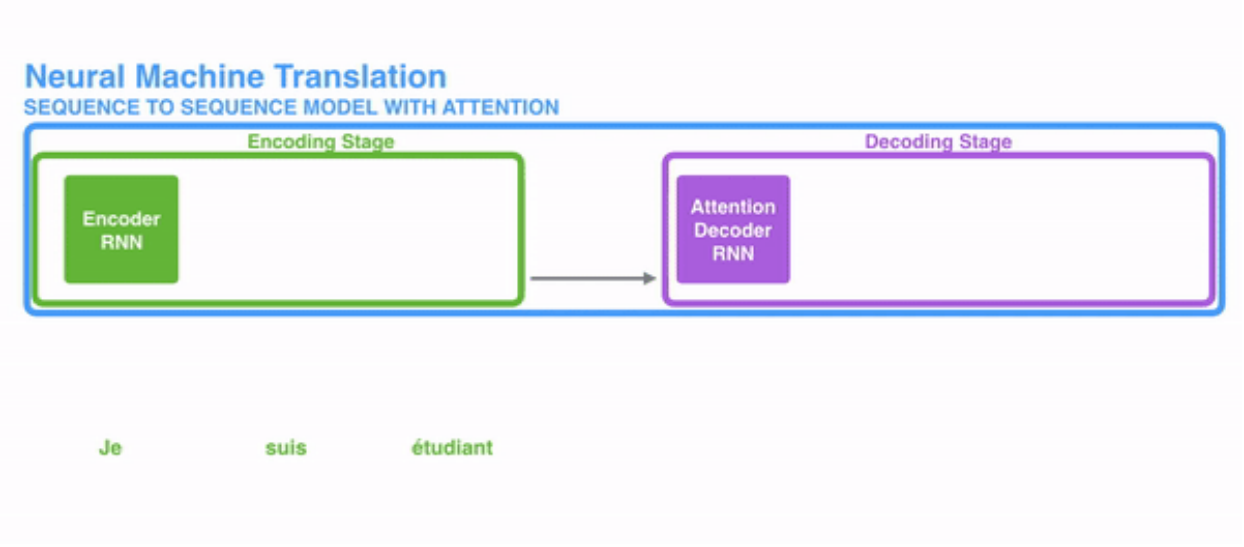
Attention 机制主要应用在 Seq2Seq 的解码器中。

如下图所示，Attention 使解码器能够在生成英语翻译之前将注意力集中在单词 “etudiant” （法语中的“student”）上：

Attention 机制与传统的 Seq2Seq 主要有两个区别：

1. **更多的 Context 信息**：编码器不传递编码阶段的最后一个隐藏状态，而是将所有的隐藏状态传递给解码器；
2. **解码时加入 Attention**：Attention 模型在产生输出之前都会加一个 Attention 操作。

我们看一下加入后的效果：



我们来看一下解码器中加入的 Attention 具体的操作步骤：

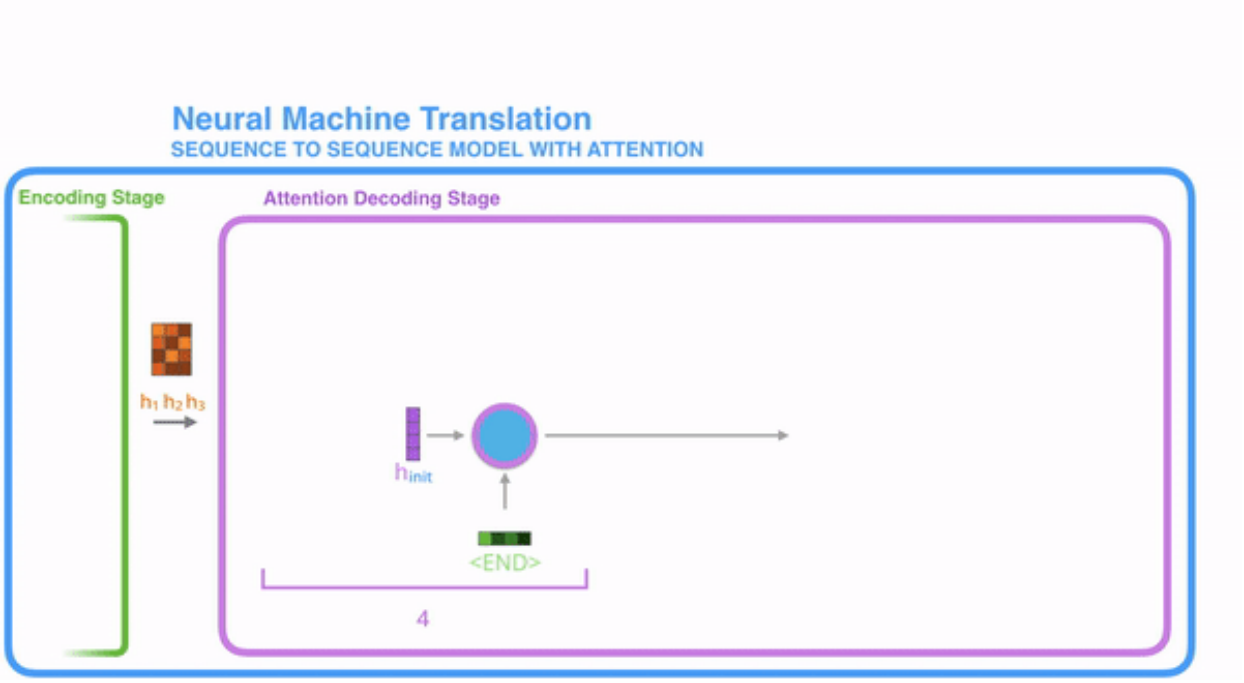
1. 查看编码器隐藏状态的集合（每个编码器隐藏状态都与输入句子的某个单词有很大关联）；
2. 给每个隐藏状态打分（计算编码器的隐藏状态与解码器的隐藏状态的相似度）；
3. 将每个隐藏状态打分通过的 Softmax 函数计算最后的概率；
4. 将第 3 步计算的概率作为各个隐藏状态的权重，并加权求和得到当前 Decoder 所需的 Context 信息。

强调一下：这种 Attention 操作在解码器每次解码的时候都需要进行。

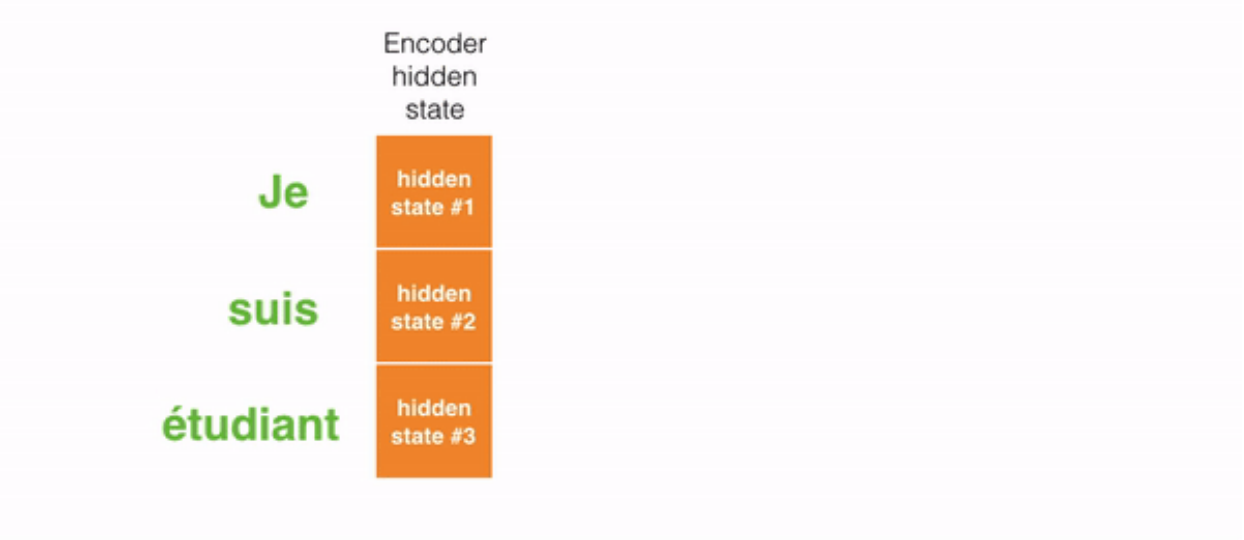
现在我们来汇总一下所有的过程，看一下 Attention 的工作流程：

1. 解码器中的第一个 RNN 有两个输入：一个是表示 <END> 标志的 Embedding 向量，另一个来自解码器的初始隐藏状态；
2. RNN 处理两个输入，并产生一个输出和一个当前 RNN 隐藏层状态向量（h4），输出将直接被舍去；
3. 然后是 **Attention 操作**：利用**编码器传来的隐藏层状态集合**和刚刚得到 RNN 的隐藏层状态向量（h4）去计算当前的上下文向量（C4）；
4. 然后拼接 h4 和 C4，并将拼接后的向量送到前馈神经网络中；
5. 前馈神经网络的到的输出即为当前的输出单词的 Embedding 向量；
6. 将此 RNN 得到的单词向量并和隐藏层状态向量（h4），作为下一个 RNN 的输入，重复计算直到解码完成。

配合动图食用效果更佳：



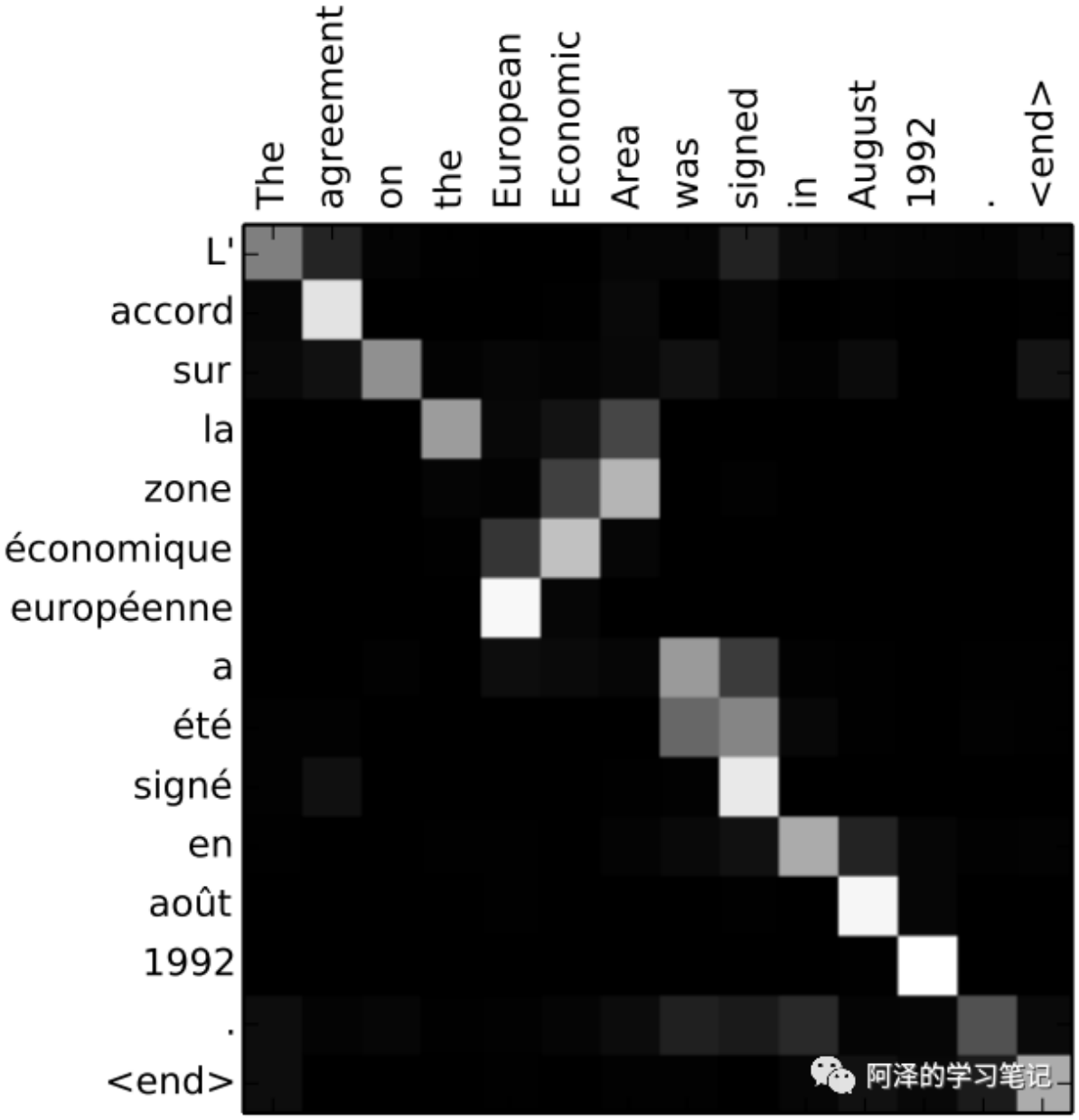
我们也可以换另一种方式来表达我们的解码过程：



左边是隐藏层状态的集合，右边是对隐藏的一个加权结果。

这里要注意，这里的模型并不是盲目地将输出中的第一个单词与输入中的第一个单词对齐，事实上，它从训练的时候就已经学会了如何排列语言对中的单词。

下面再给出 Attention 论文中的一个例子（模型在输出 “European Economic Area” 时，其与法语中的顺序恰好的是相反的）：



3. Transformer

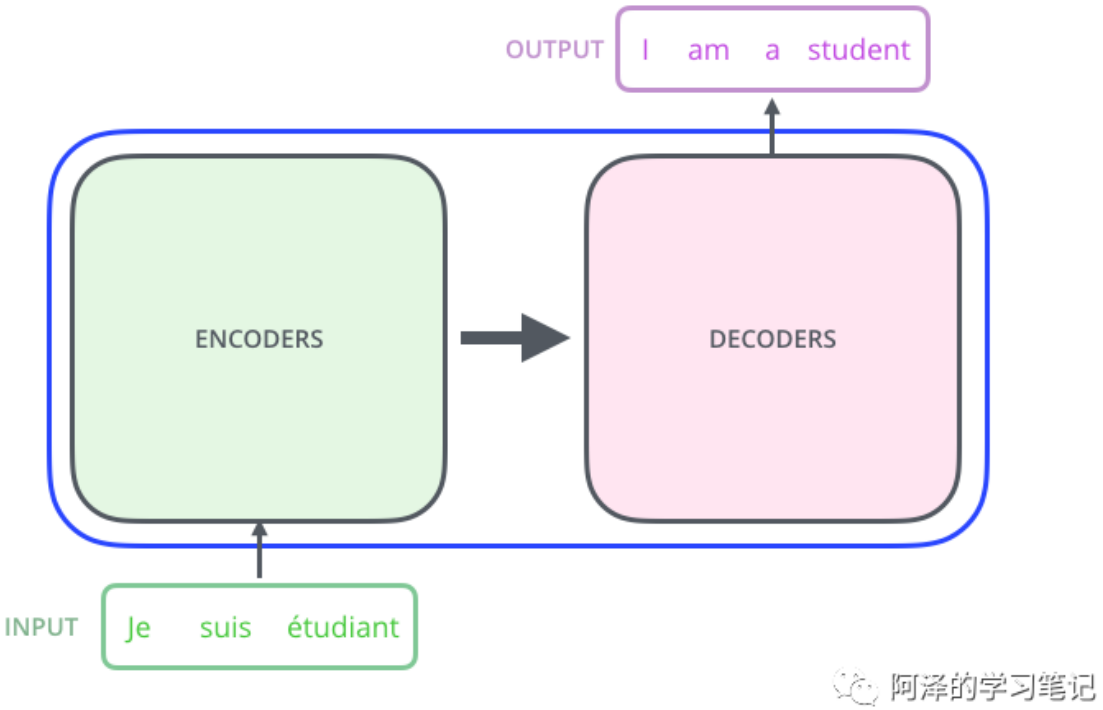
加了 Attention 的 Seq2Seq 模型效果得到了显著提升，并成功应用于诸多领域，但仍然存在很多问题。比如说：

- RNN 在处理长序列问题时容易出现梯度爆炸或者梯度消失的问题（虽然 LSTM 可以一定程度上缓解这个问题）；
- RNN 因其顺序性，无法实现并行化，所以训练速度很慢。

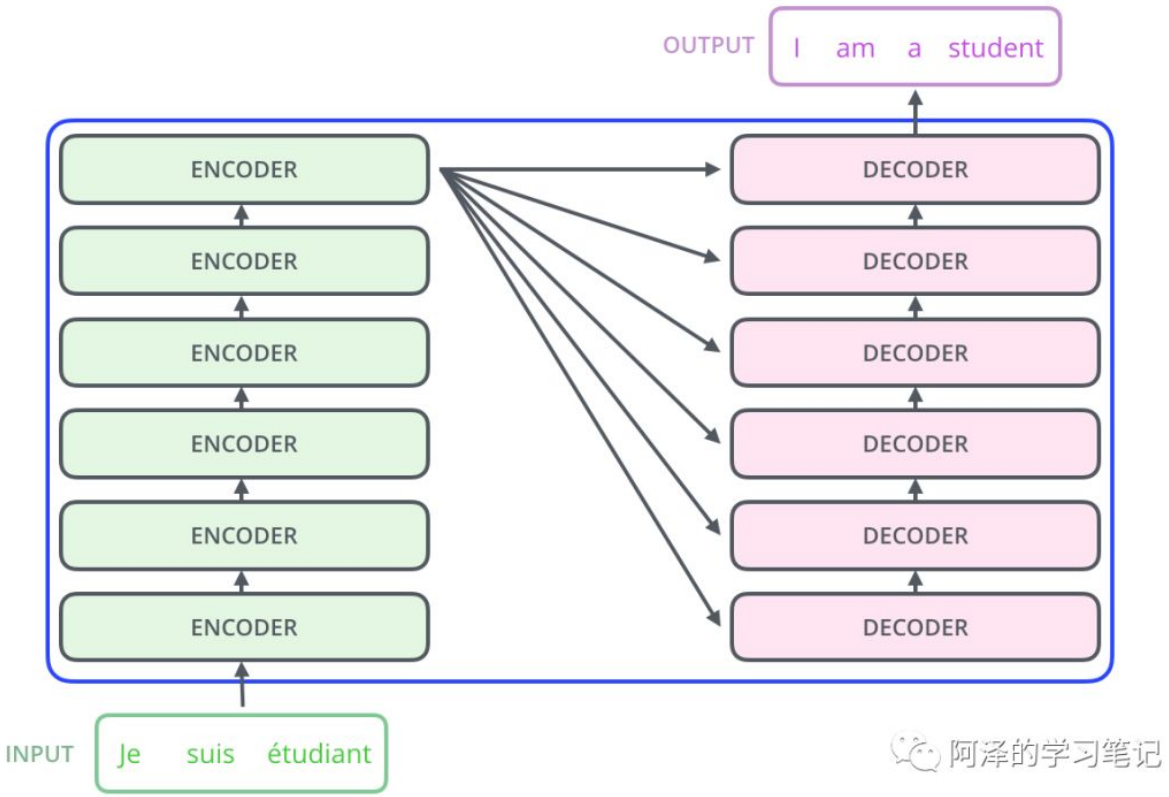
Transformer 在 Seq2Seq 的基础上进行了改进，只使用 Attention 机制，舍去了 CNN/RNN/LSTM 等结构，提高了模型的并行度。

3.1 Architecture

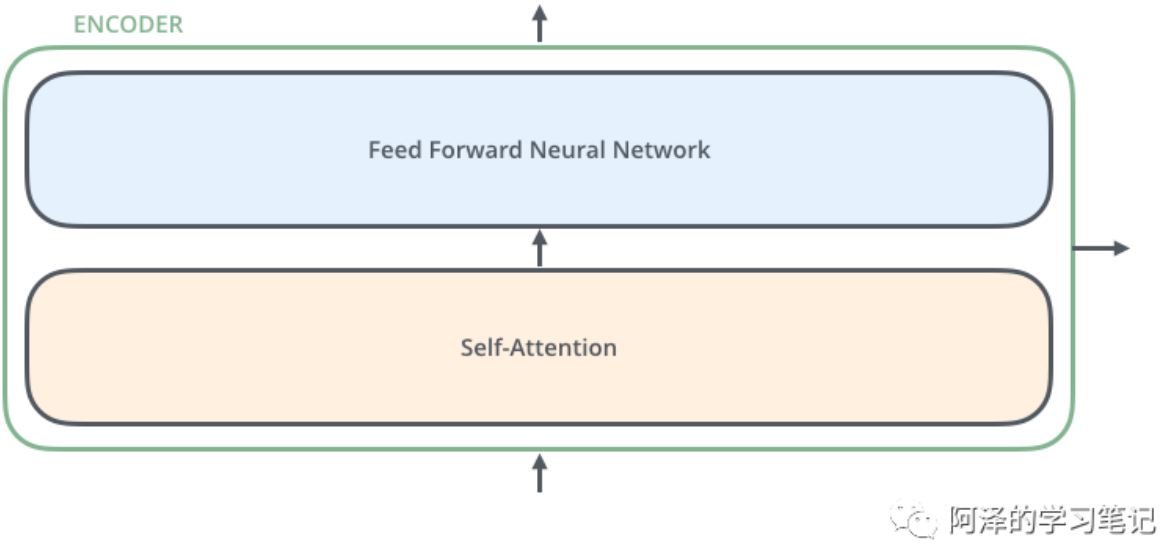
我们先概览一下 Transformer 模型的架构，Transformer 同样基于 Encoder-Decoder 架构：



编码部分是堆了六层编码器，解码部分也堆了六个解码器。



所有的编码器在结构上都是相同的，每一个都被分成两个子层：



编码器的输入

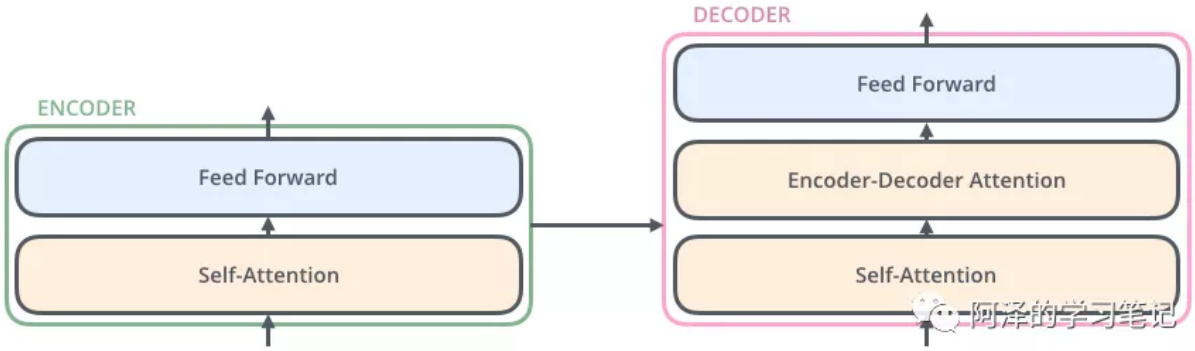
首先经过一层 Self-Attention，这一层主要用来帮助编码器在对特定的单词进行编码时查看输入句子中的其他单词。

Self-Attention 层的输出进入给前馈神经网络（Feed Forward Neural Network，以下简称 Feed Forward），所有前馈神经网络的结构都相同并且相互独立。

解码器拥有三层

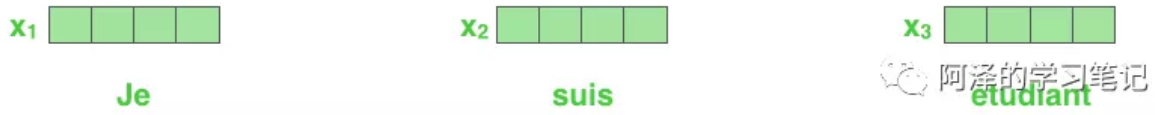
除了刚刚介绍的 Self-Attention 和 Feed Forward 外，还有一层 Encoder-Decoder Attention 层（以下简称 Attention 层，区别于 Self-Attention），

Attention 层位于 Self-Attention 和 Feed Forward 层之间，主要用来帮助解码器将注意力集中在输入语句的相关部分（类似于 Seq2Seq 模型中的 Attention）



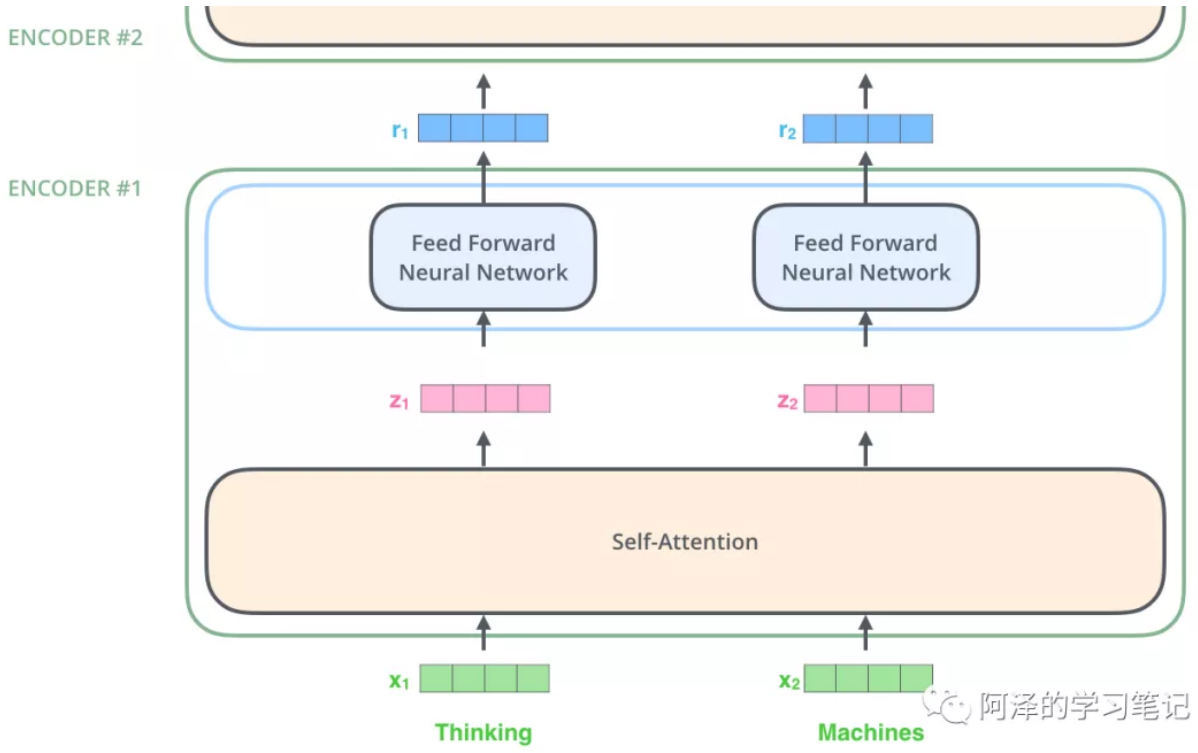
3.2 Encoder Side

与一般 NLP 方法一样，我们需要将每个输入字转换成 Embedding 向量，Transformer 采用 512 维的向量。



Embedding 只发生在最下层的编码器中，其他编码器接收下层的输出作为输入。

序列中的每个 Embedding 向量都需要经过编码器的 Self-Attention 层和 Feed Forward 层：



这里我们需要注意：在 Self-Attention 层中，这些单词之间存在依赖关系；但 Feed Forward 层没有依赖，所以可以在 Feed Forward 层并行化训练。

3.3 Self-Attention

谷歌论文中首次提出 Self-Attention 概念，我们来看一下它是怎么工作的。

假设我们现在要翻译下面的这个这句话：

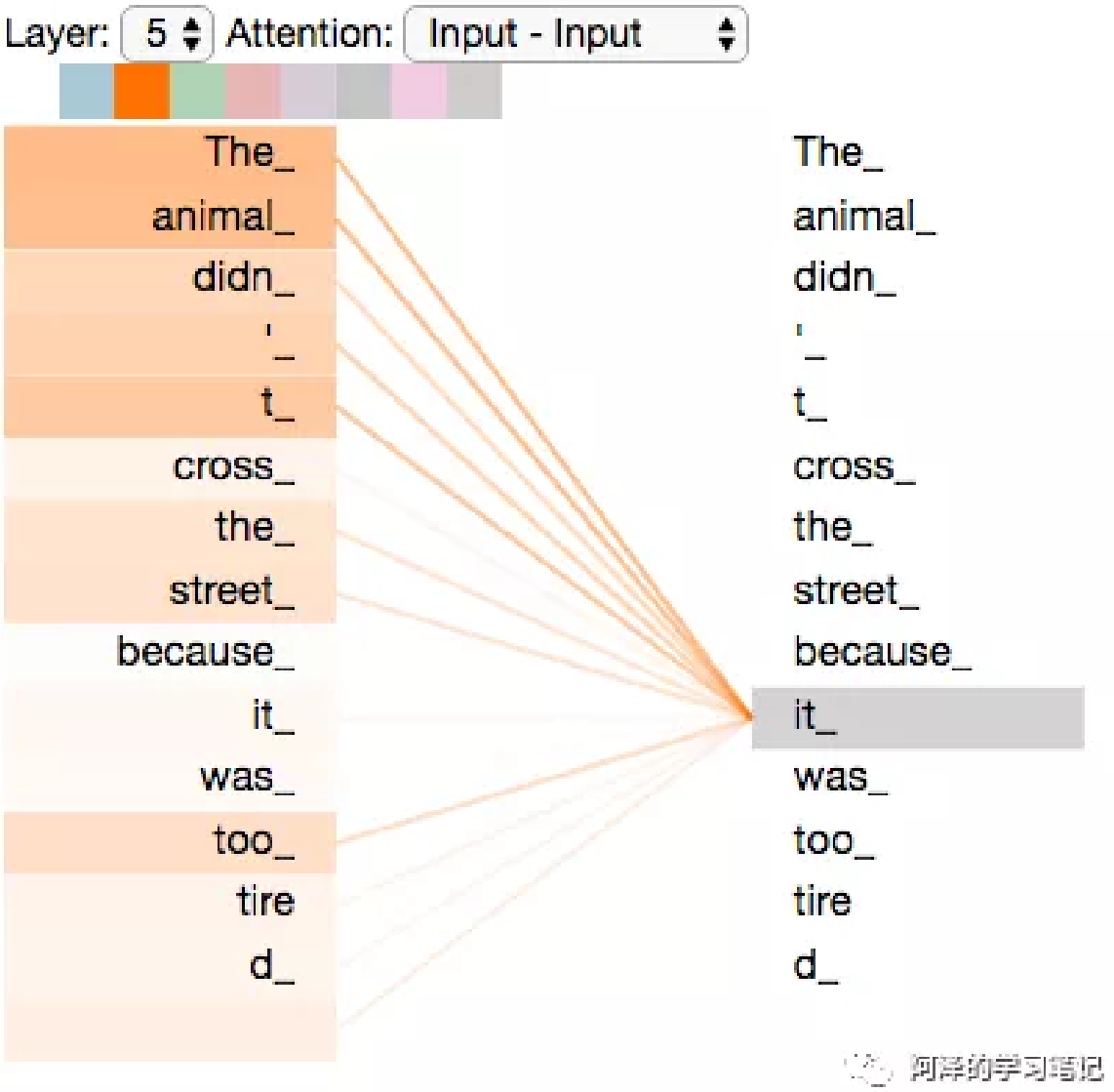
“The animal didn't cross the street because it was too tired”

这个句子中的 it 指的是什么呢？是指 street 还是 animal 呢？这对人来说比较简单，但是对算法来说就没那么简单了。

而 Self-Attention 就可以解决这个问题，在处理 it 时会将它和 animal 联系起来。

Self-Attention 允许当前处理的单词查看输入序列中的其他位置，从而寻找到有助于编码这个单词的线索。

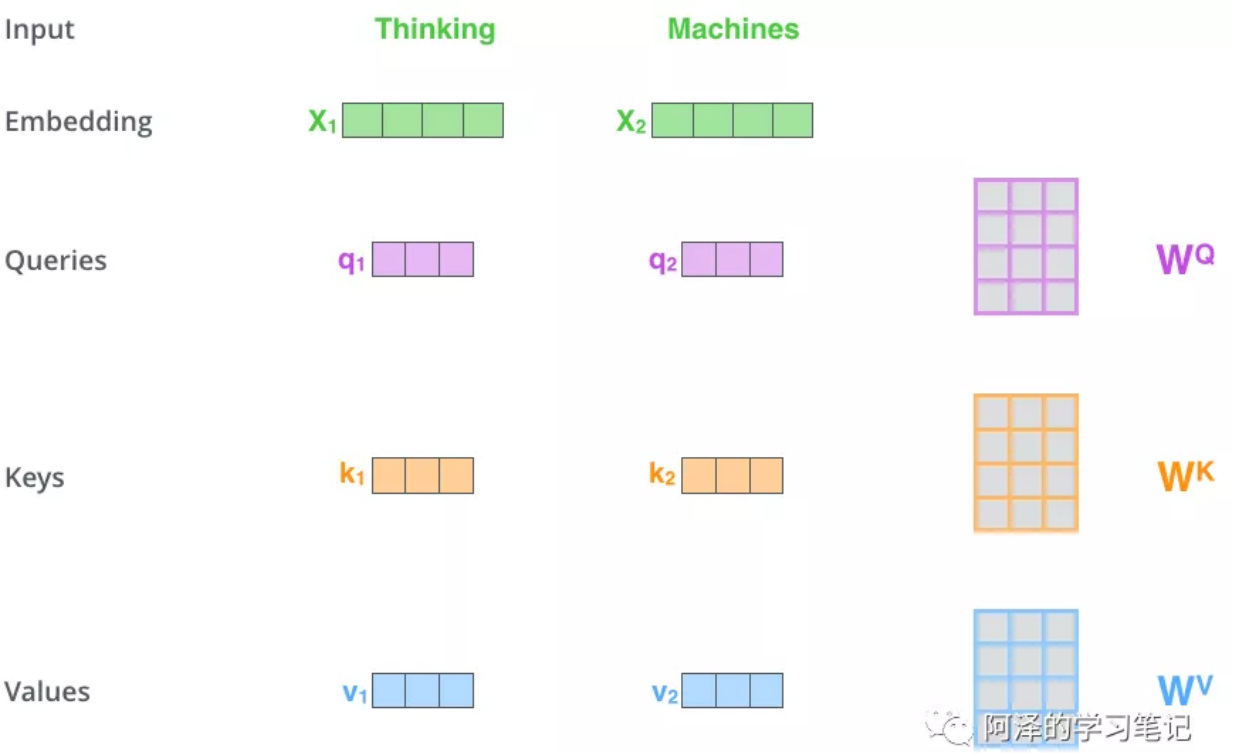
以下图为例，展示了第五层的编码器，当我们在编码 it 时，Attention 机制将 The animal 和自身的 it 编码到新的 it 上。



我们来看下 Self-Attention 是如何聪明的识别出来的。

第一步，我们对于每个单词来说我们都一个 Embedding 向量，下图绿色部分。

此外，对于每个单词我们还会有三个向量分别为：查询向量（Querry）、键向量（Key）和值向量（Value），这些向量是单词的 Embedding 向量分别与对应的查询矩阵 、键矩阵 和值矩阵 相乘得来的。



这边要注意，对于 Embedding 向量来说是 512 维，而对于新向量来说是 64 维。（大小是架构设定的。）

那么 Querry、Key 和 Value 向量到底是什么呢?

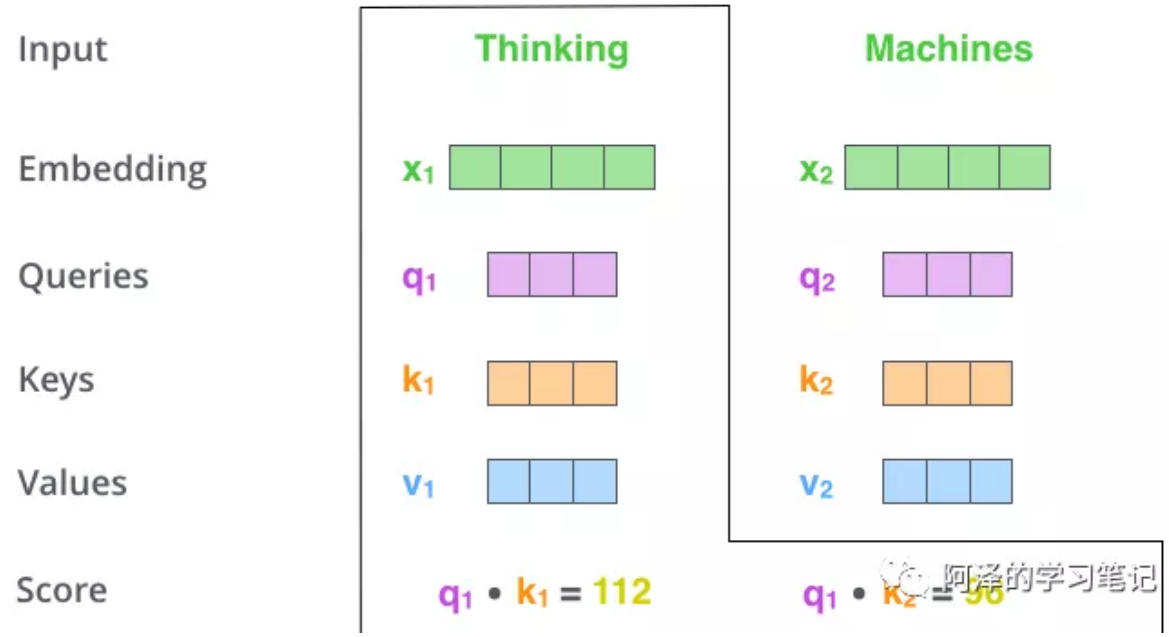
这三个概念是对 Attention 的计算和思考非常有用的三个抽象概念，我们接下来会详细叙述这些向量的作用。

第二步，我们来看下 Self-Attention 是如何计算的。

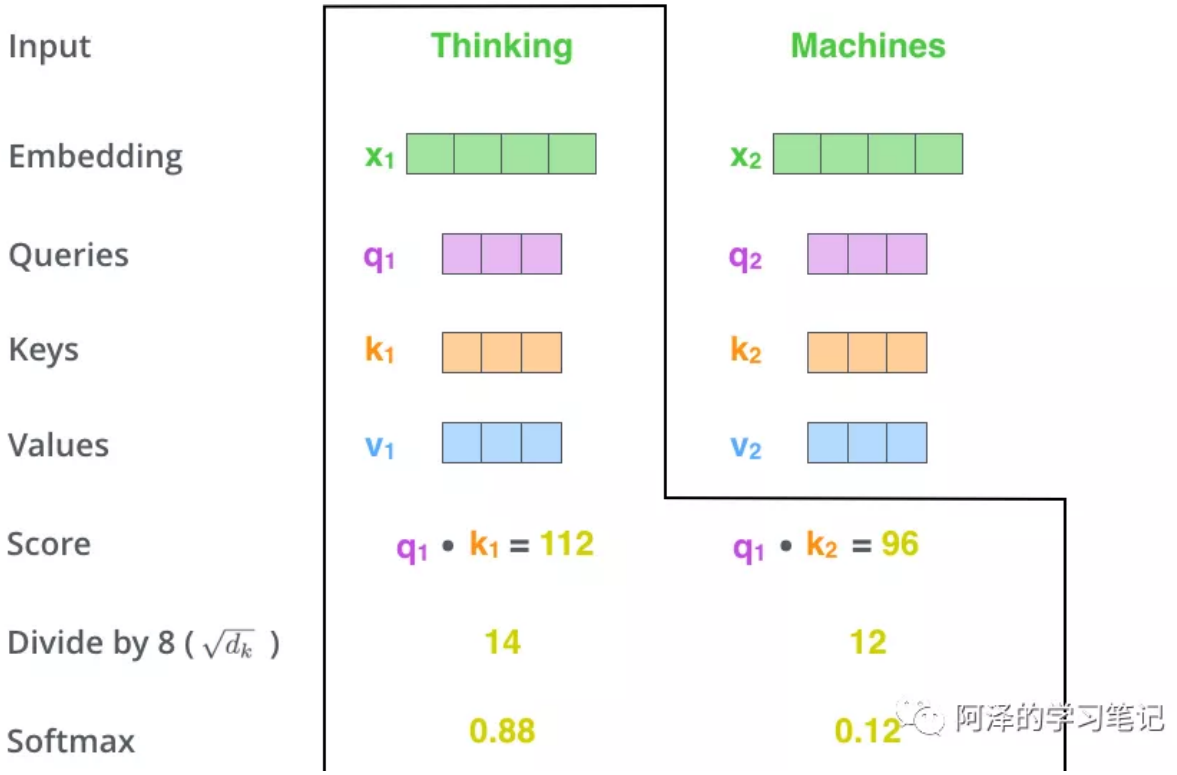
假设我们正在计算 Thinking 的 Self-Attention。我们需要将这个单词和句子中的其他单词得分，这个分数决定了我们将一个单词编码到某个位置时，需要将多少注意力放在句子的其他部分。

这个得分是通过当前单词的 Querry 向量和其他词的 Key 向量做内积得到的。

（也可以理解为当前单词的是由句子的所有单词加权求和得到的，现在计算的是当前单词和其他单词的分数，这个分数将用于后面计算各个单词对当前单词的贡献权重。）

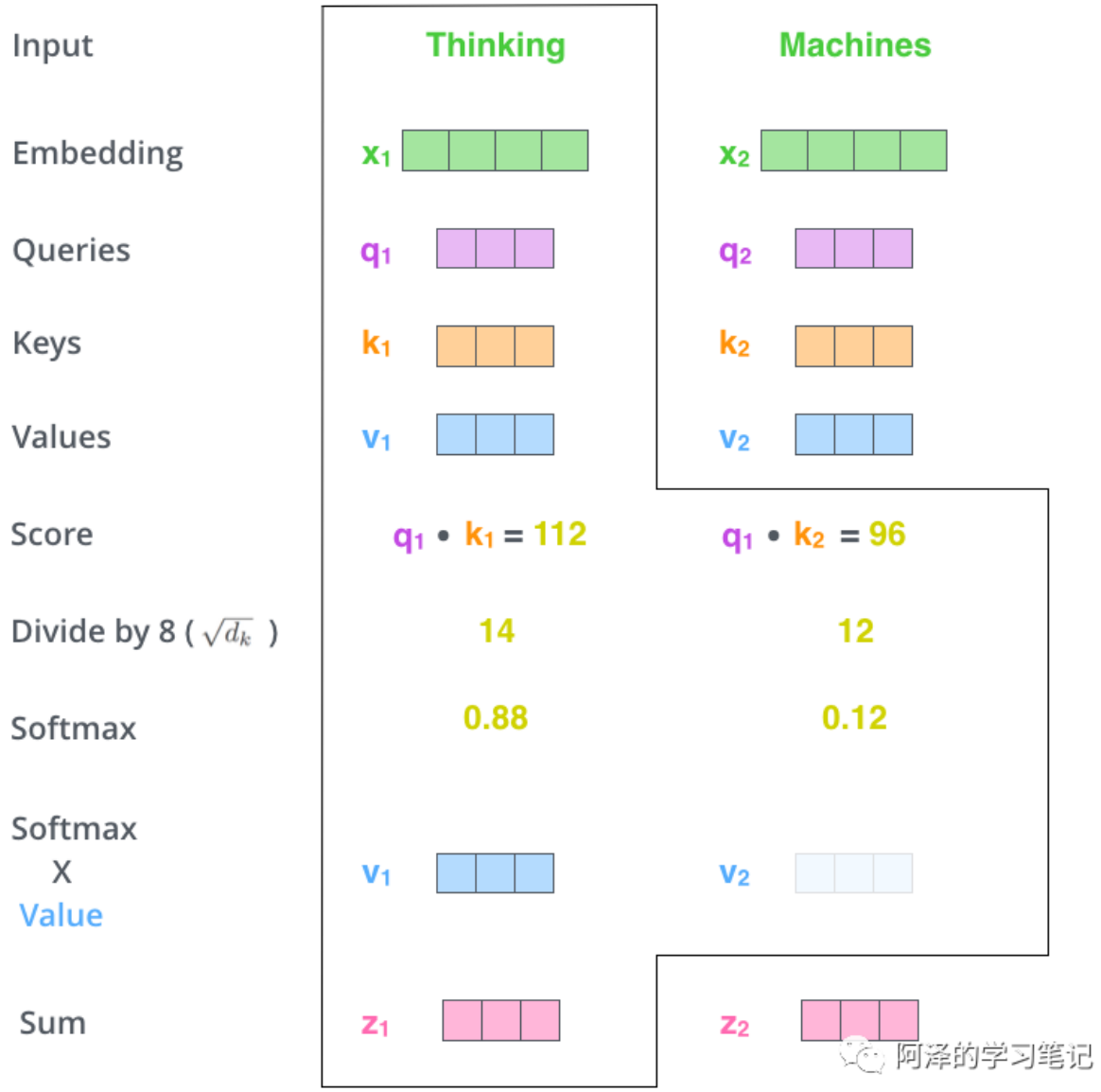


第三步，将这个分数除以 8（Value 向量是 64 维，取平方根，主要是为了稳定梯度）。然后将分数通过 Softmax 标准化，使它们都为正，加起来等于1。



经过 Softmax 后的分数决定了序列中每个单词在当前位置的表达量（如，对着 Thinking 这个位置来说，= 0.88 * Thinking + 0.12 * Machines）。Softmax 分数越高表示与当前单词的相关性更大。

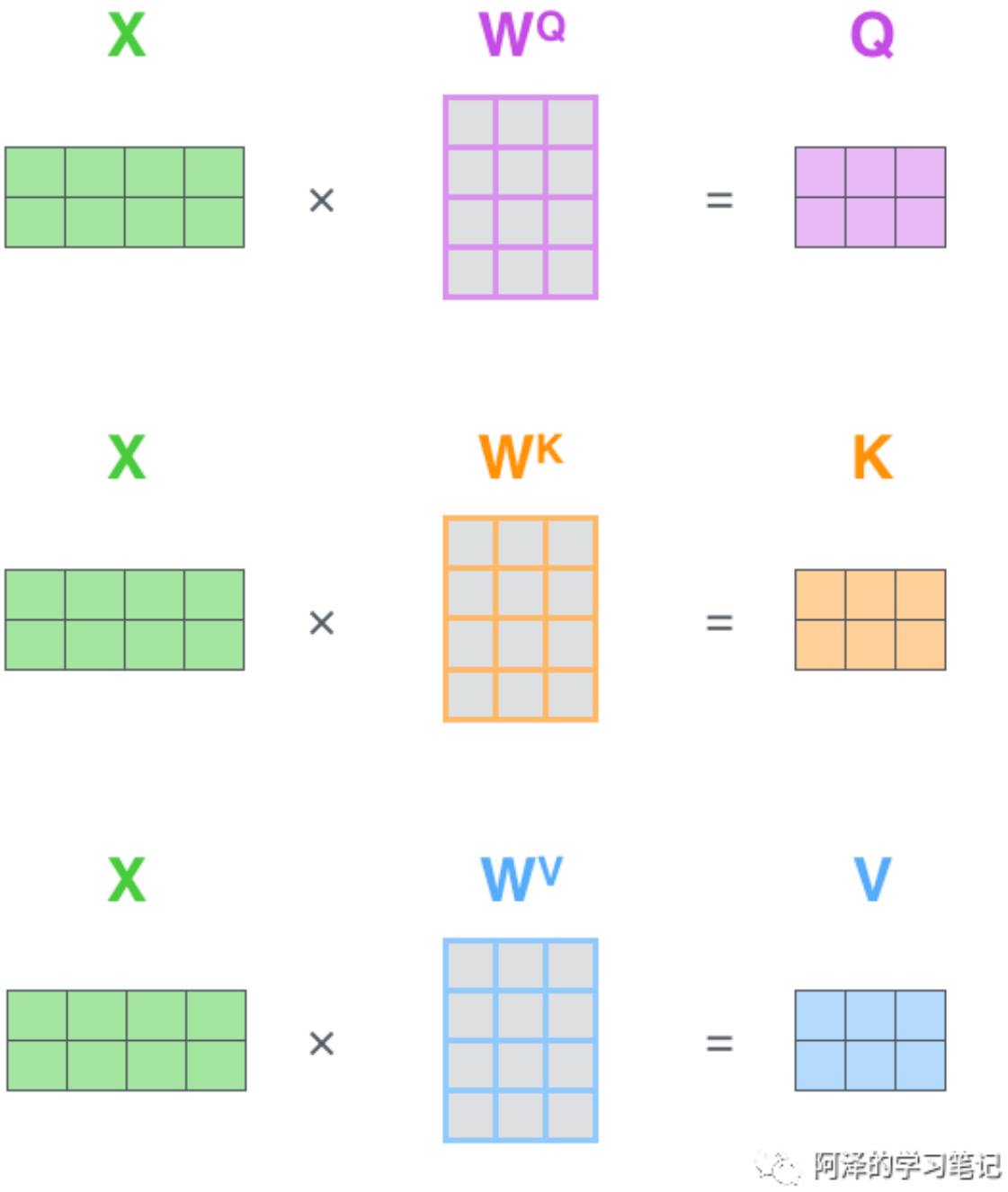
第四步，将每个单词的 Value 向量乘以 Softmax 分数并相加得到一个汇总的向量，这个向量便是 Self-Attention 层的输出向量。



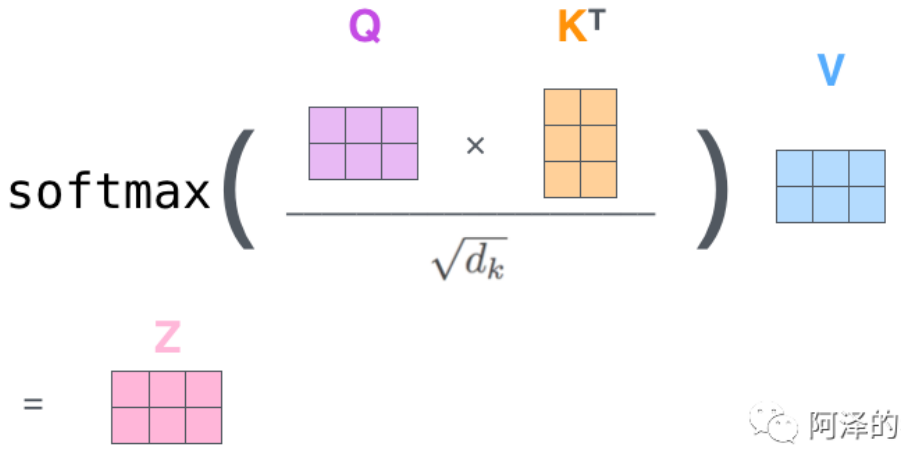
以上便是 Self-Attention 的计算过程，得到的这个向量会输送给下一层的 Feed Forward 网络。

在实际的实现过程中，为了快速计算，我们是通过矩阵运算来完成的。简单的看一下 Self-Attention 的矩阵运算：

首先是输入矩阵与查询矩阵、键矩阵和值矩阵。

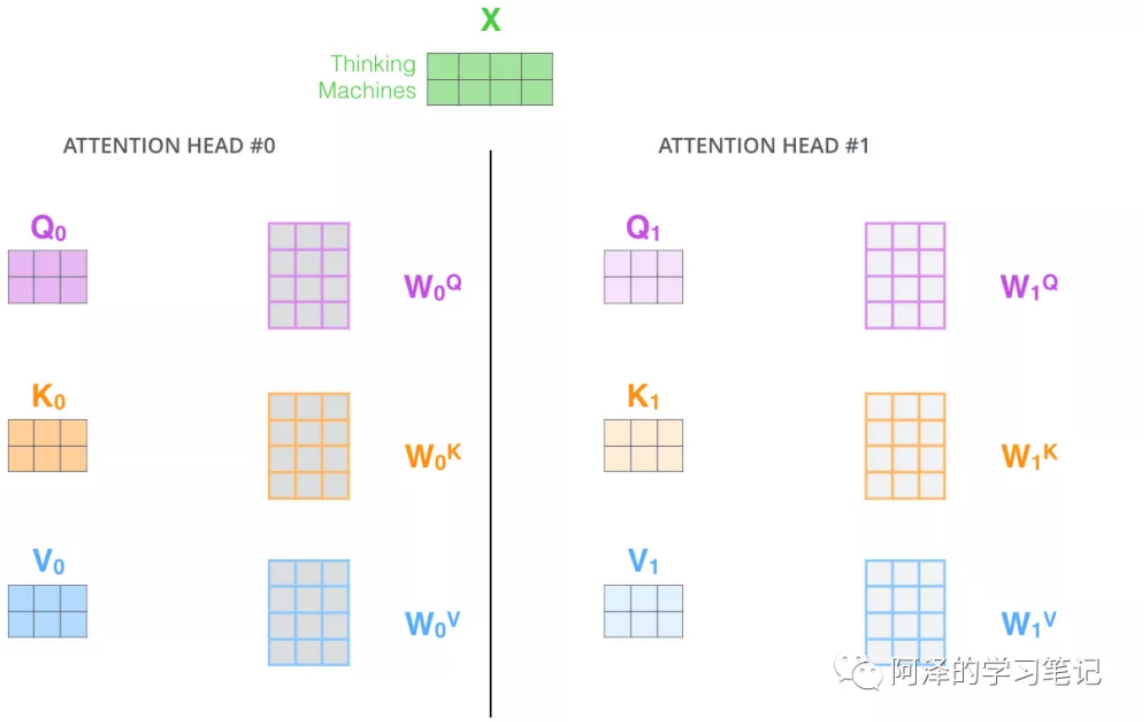


然后用 softmax 计算权重，并加权求和：

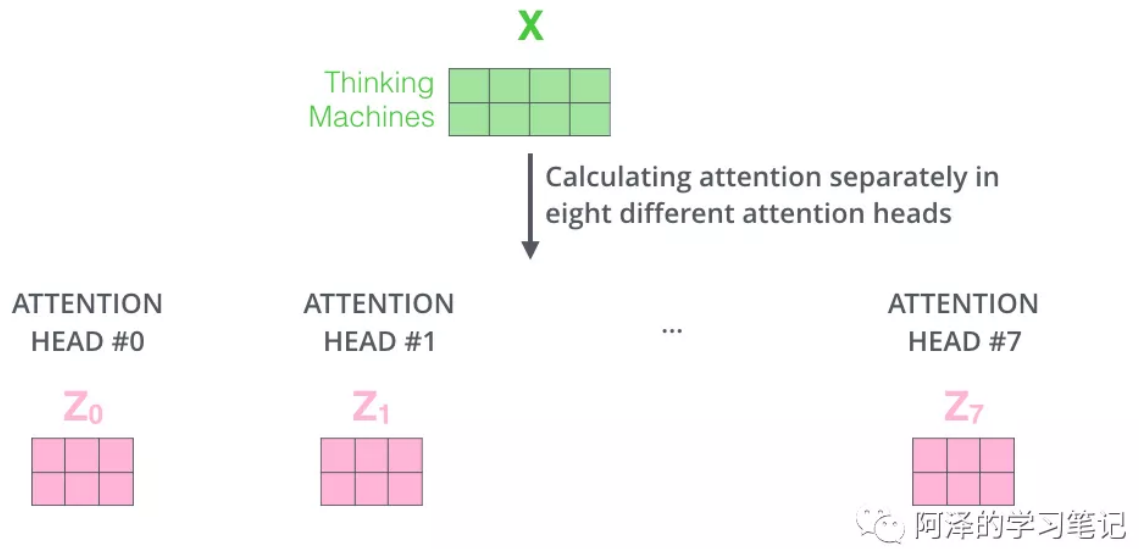


3.4 Multi-Head Attention

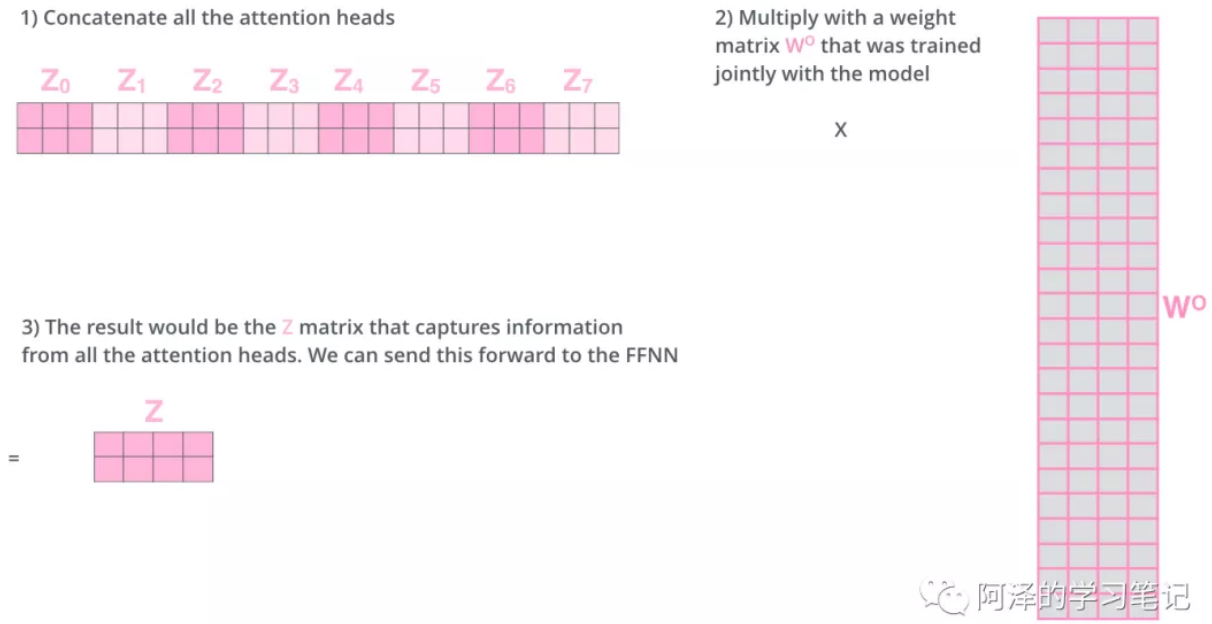
谷歌论文中模型框架中画的 Multi-Head Attention 层，Multi-Head Attention 其实就是包含了多个 Self-Attention 。所以 Multi-Head Attention 有多个不同的查询矩阵、键矩阵和值矩阵，**为 Attention 层提供了多个表示空间**。



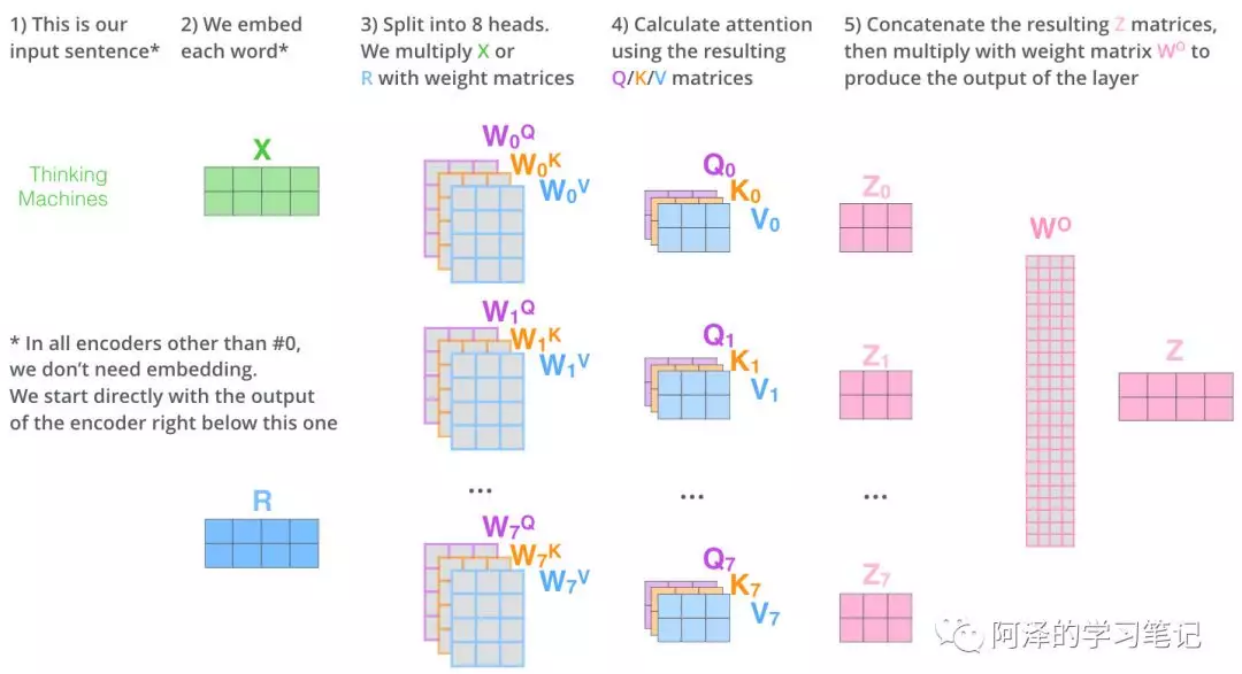
Transformer 中每层 Multi-Head Attention 都会使用八个独立的矩阵，所以也会得到 8 个独立的 Z 向量：



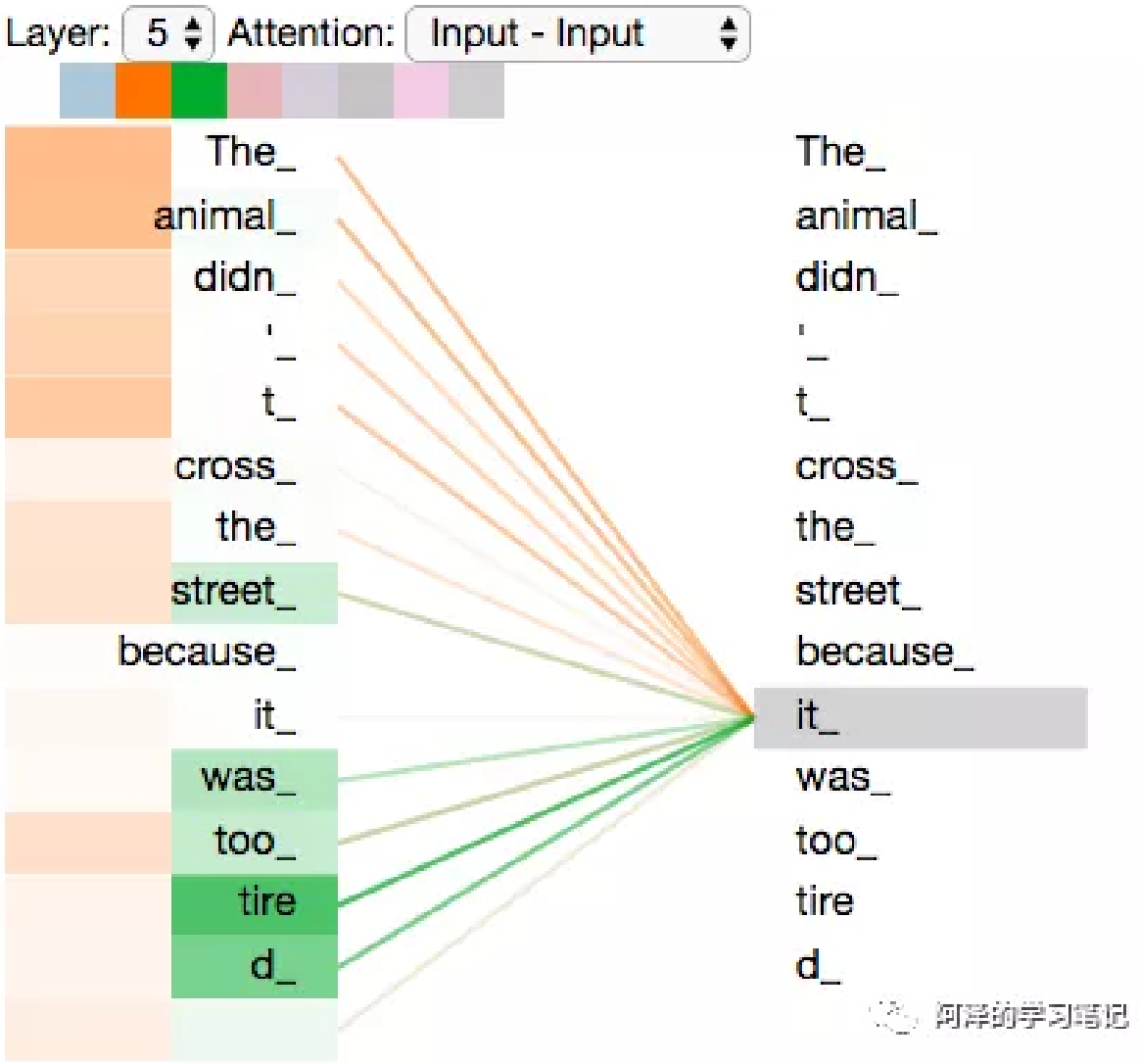
但是 Feed Forward 层并不需要 8 个矩阵，它只需要一个矩阵（每个单词对应一个向量）。所以我们需要一种方法把这8个压缩成一个矩阵。这边还是采用矩阵相乘的方式将 8 个 Z 向量拼接起来，然后乘上另一个权值矩阵 W，得到后的矩阵可以输送给 Feed Forward 层。



以上便是 Multi-Head Attention 的内容，我们把它们都放在一起看一看：

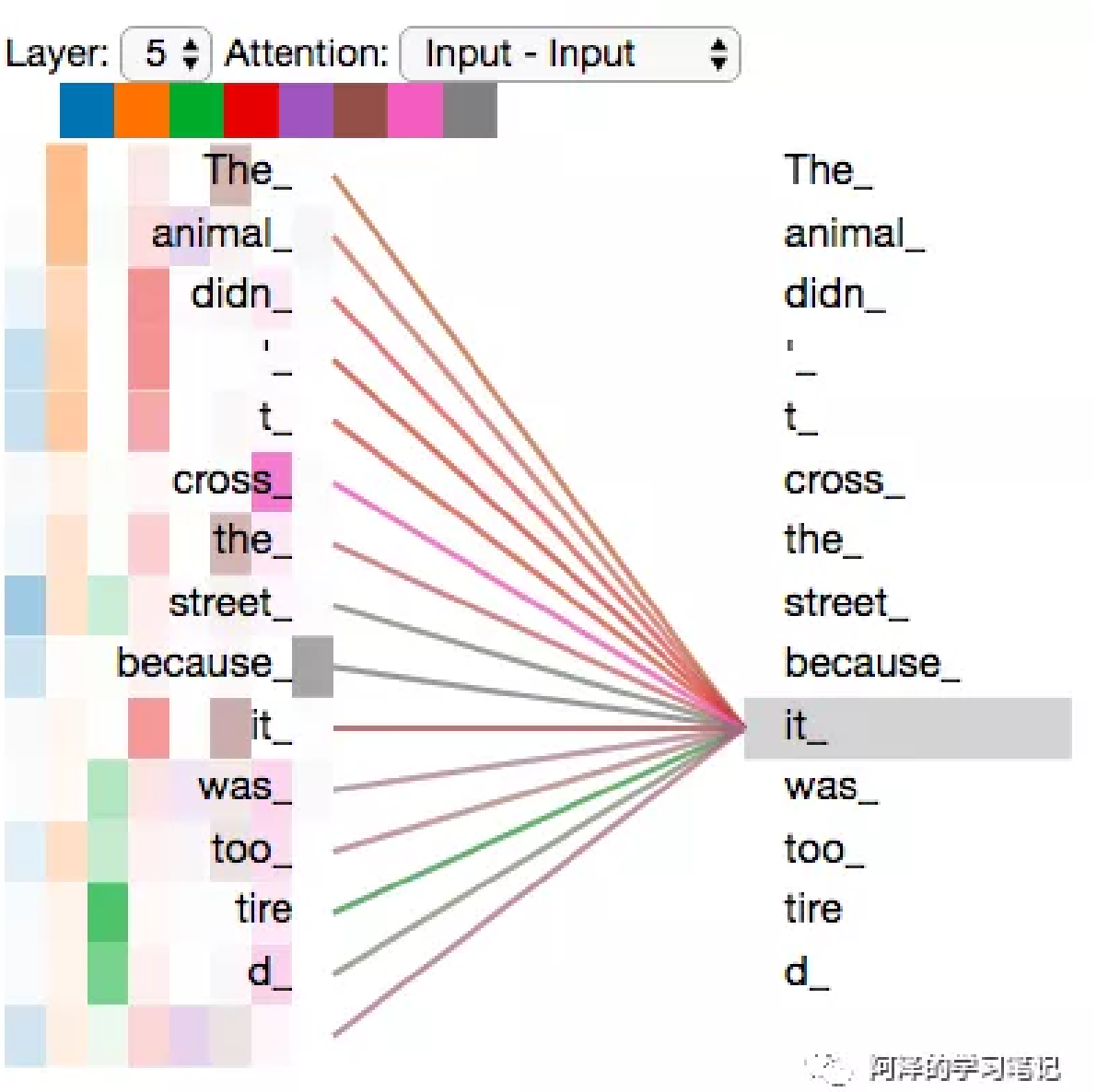


我们来看一下 Multi-Head Attention 的例子，看看不同的 Attention 把 it 编码到哪里去了：



这边先只关注两个 Attention，可以看到一个 Attention 关注了 animal，另一个专注了 tired。

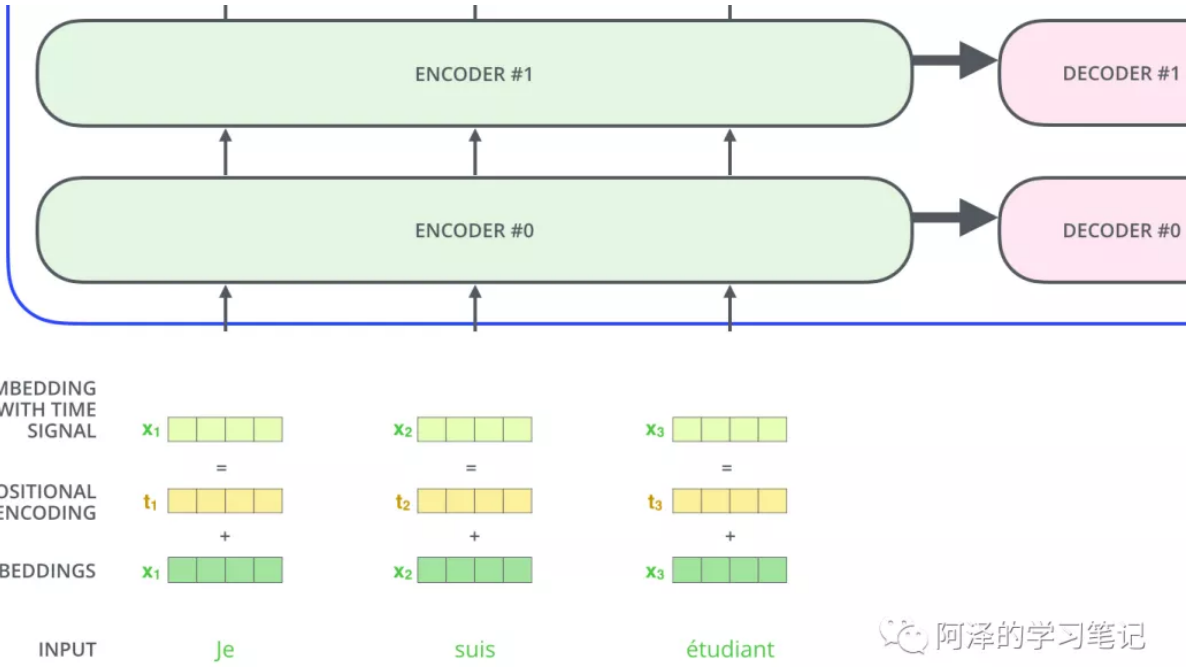
但如果我们把八个 Attention 都打开就很难解释了：



3.5 Positional Encoding

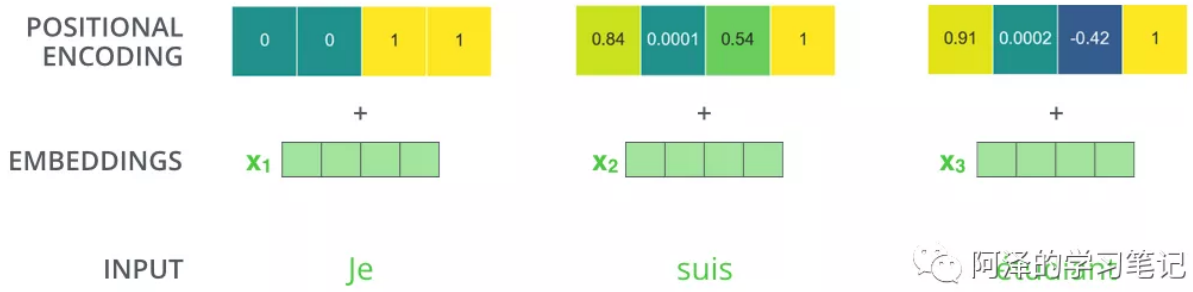
我们的模型可以完成单词的 Attention 编码了，但是目前还只是一个词袋结构，还需要描述一下单词在序列中顺序问题。

为了解决这个问题，Transformer 向每个输入的 Embedding 向量添加一个位置向量，有助于确定每个单词的绝对位置，或与序列中不同单词的相对位置：

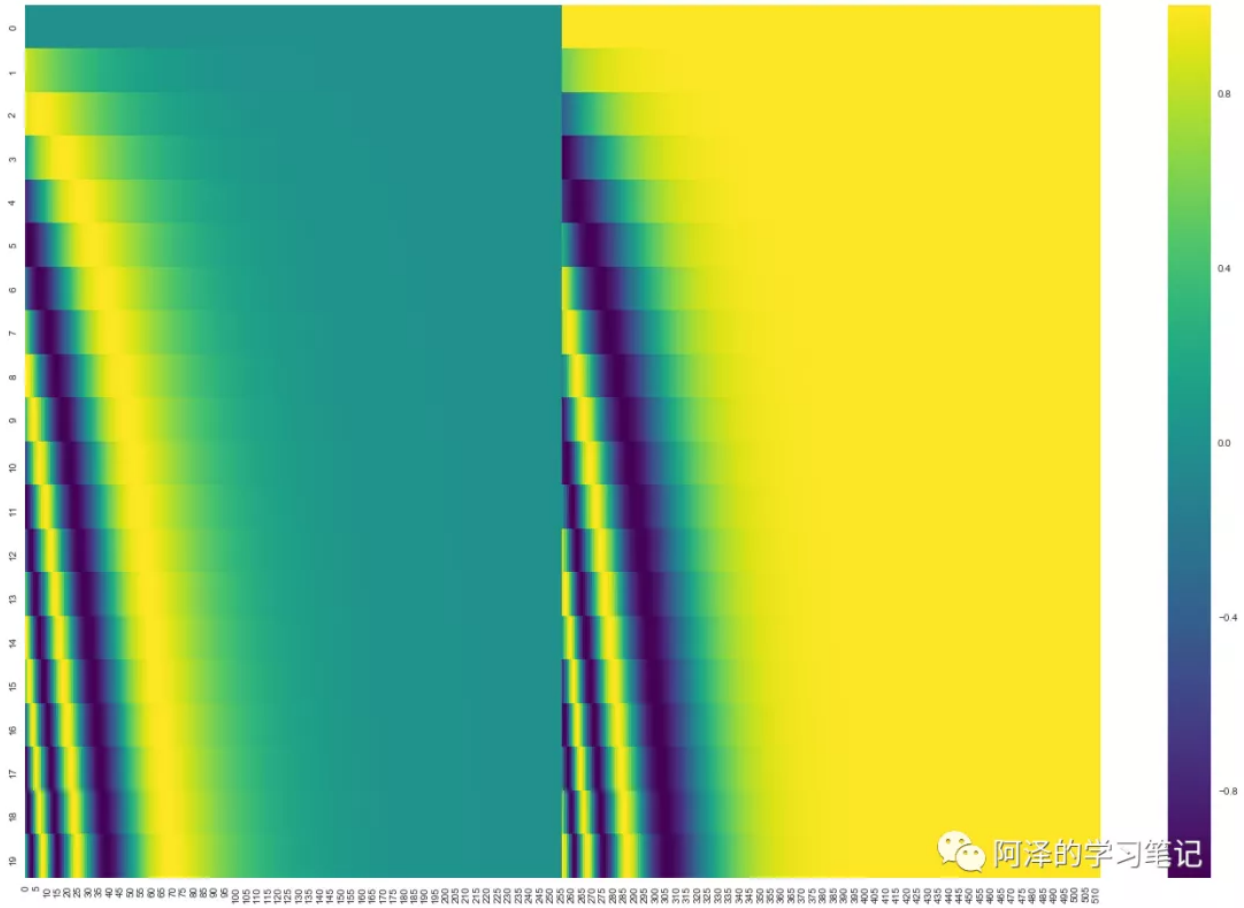


这种方式之所以有用，大概率是因为，将配置信息添加到 Embedding 向量中可以在 Embedding 向量被投影到Q/K/V 向量后，通过 Attention 的点积提供 Embedding 向量之间有效的距离信息。

举个简单的例子，以 4 维为例：



下图显示的是，每一行对应一个位置编码向量。所以第一行就是我们要添加到第一个单词的位置向量。每一行包含512个值——每个值的值在1到-1之间（用不同的颜色标记）。



这张图是一个实际的位置编码的例子，包含 20 个单词和 512 维的 Embedding 向量。

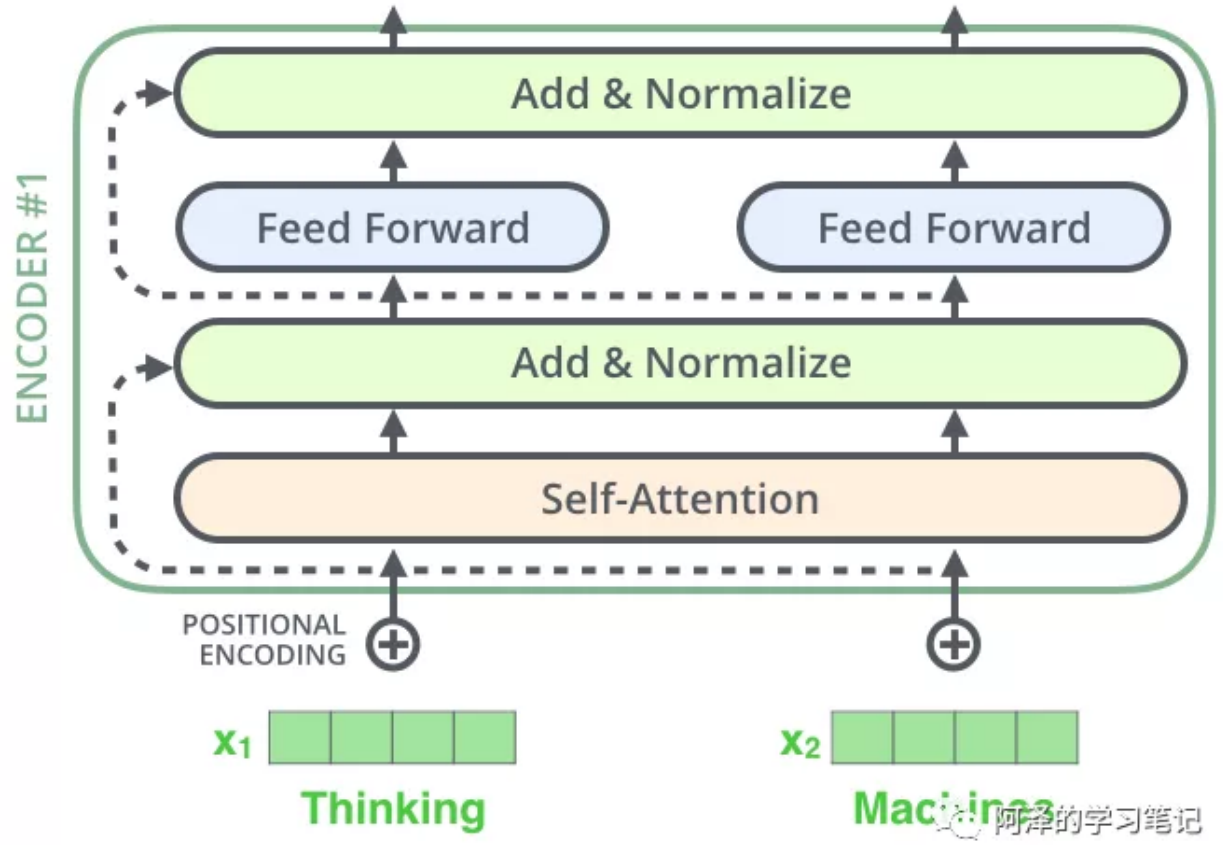
可以看到它从中间一分为二。这是因为左半部分的值是由一个 Sin 函数生成的，而右半部分是由另一个 Cos 函数生成的。

然后将它们连接起来，形成每个位置编码向量。

这样做有一个很大的优势：他可以将序列扩展到一个非常的长度。使得模型可以适配比训练集中出现的句子还要长的句子。

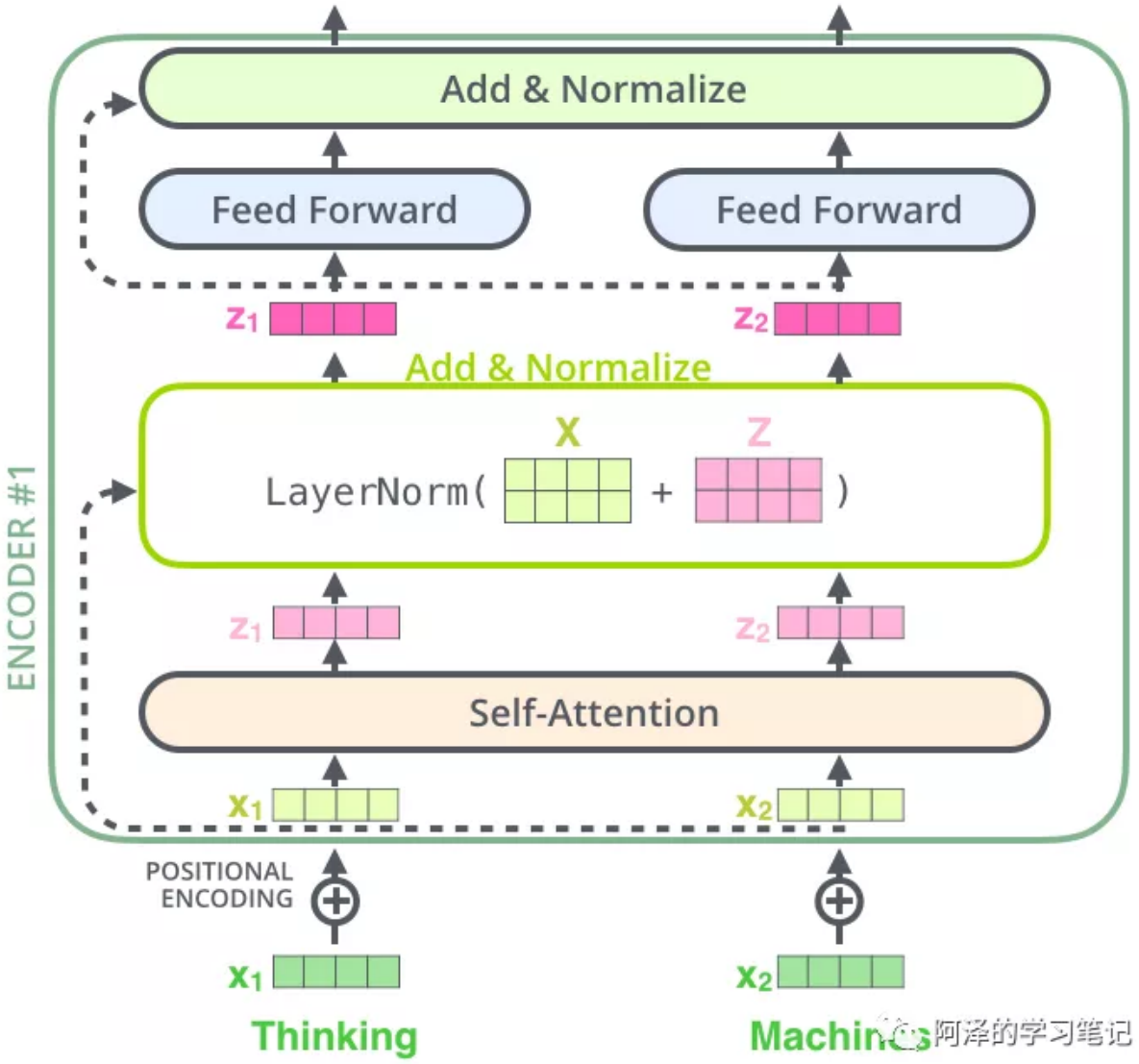
3.6 Residuals残差

这里还要提一个编码器的细节，每个编码器中的每个子层（Self-Attention, Feed Forward）都有一个围绕它的虚线，然后是一层 ADD & Normalize 的操作。

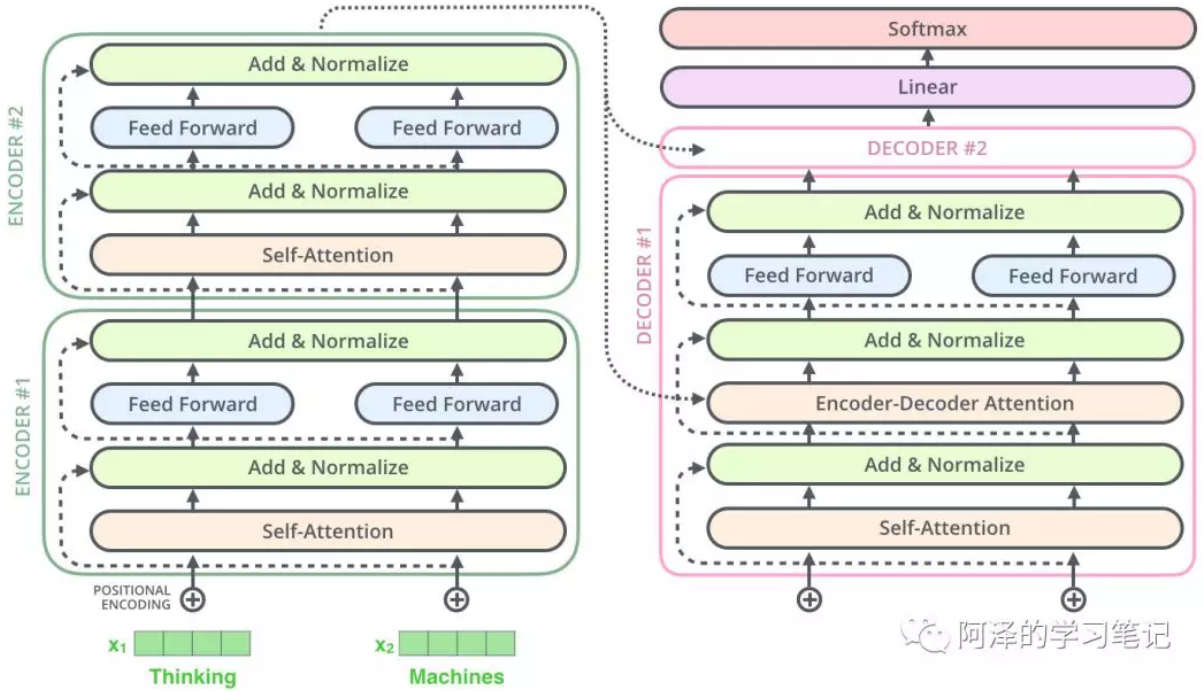


这个虚线其实就是一个残差，为了防止出现梯度消失问题。

而 Add & Normalize 是指将上一层传过来的数据和通过残差结构传过来的数据相加，并进行归一化：



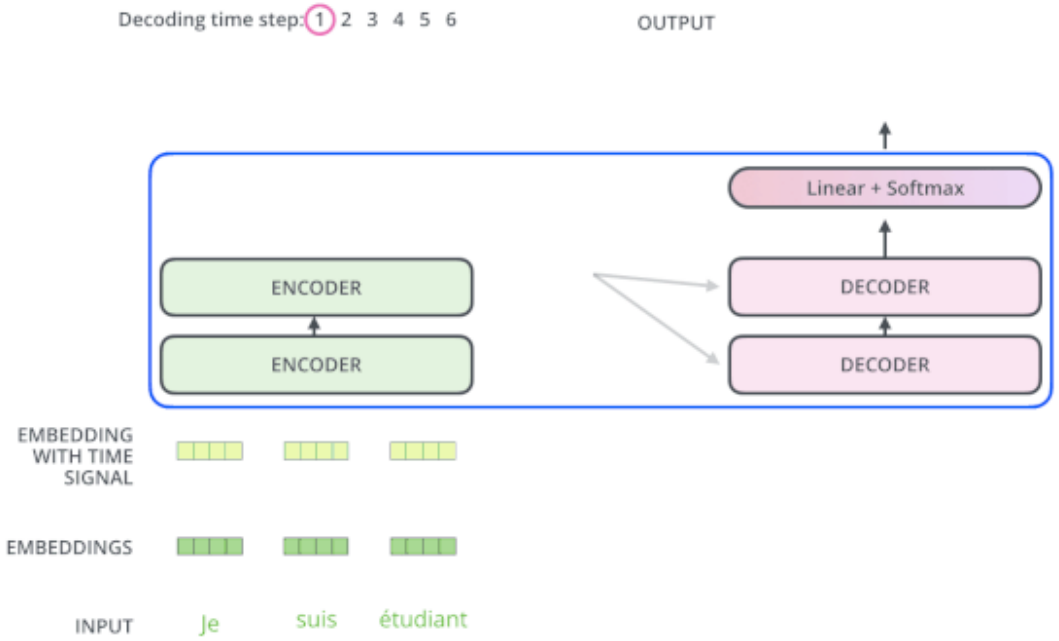
同样适用于解码器的子层:



3.7 Decoder Side

看完编码器后我们再看一下解码器。

编码器首先处理输入序列，然后将顶部编码器的输出转换成一组 Attention 矩阵 K 和 V，这两个矩阵主要是给每个解码器的 ” Encoder-Decoder Attention “ 层使用的，这有助于解码器将注意力集中在输入序列中的适当位置：



下图展示了翻译过程，不断重复此过程直到出现结束符号为止。像我们处理编码器的输入一样，我们将输出单词的 Embedding 向量和位置向量合并，并输入到解码器中，然后通过解码器得到最终的输出结果。

解码器中的 Self-Attention 层操作方式与编码器中的略有不同。

在解码器中，Self-Attention 层只允许注意到输出单词注意它前面的单词信息。在实现过程中通过在将 Self-Attention 层计算的 Softmax 步骤时，屏蔽当前单词的后面的位置来实现的（设置为-inf）。

解码器中的 “Encoder-Decoder Attention” 层的工作原理与 “Multi-Head Attention” 层类似，只是它从其下网络中创建查询矩阵，并从编码器堆栈的输出中获取键和值矩阵（刚刚传过来的 K/V 矩阵）。

3.8 Softmax Layer

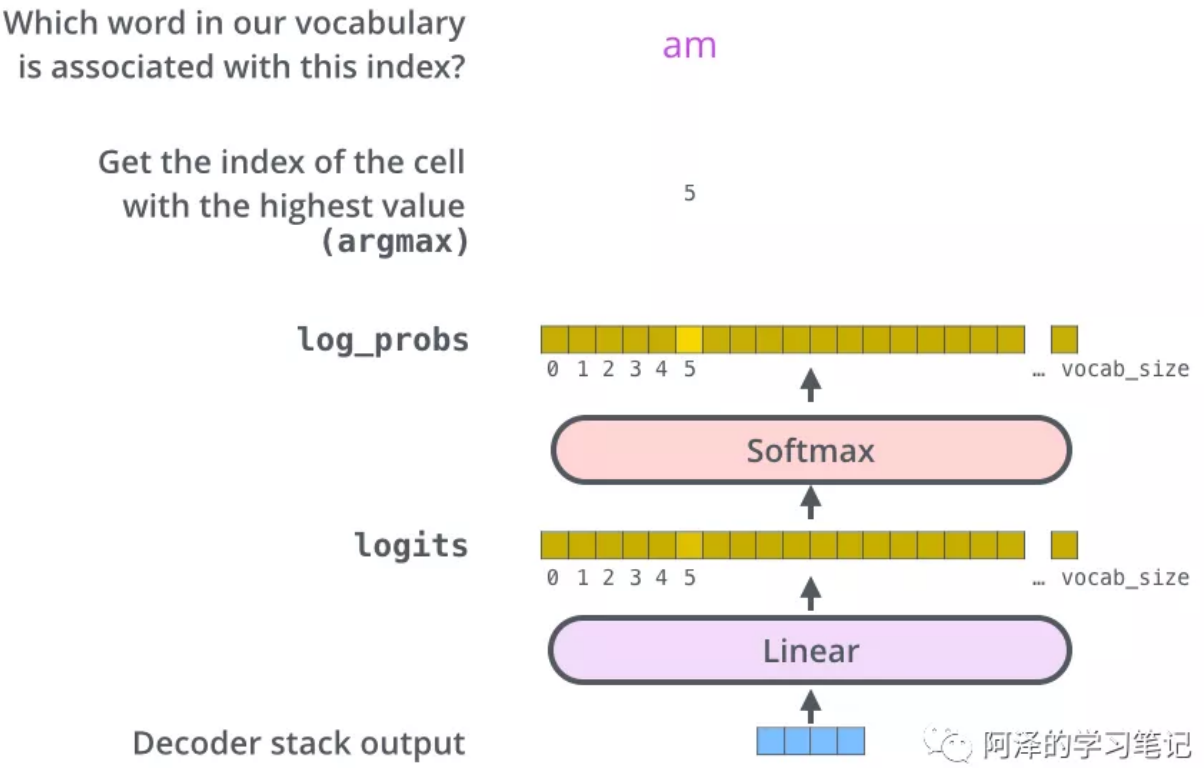
解码器输出浮点数向量，我们怎么把它变成一个单词呢?

这就是最后一层 Linear 和 Softmax 层的工作了。

Linear 层是一个简单的全连接网络，它将解码器产生的向量投影到一个更大的向量上，称为 logits 向量。

假设我们有 10,000 个不同的英语单词，这时 logits 向量的宽度就是 10,000 个单元格，每个单元格对应一个单词的得分。这就解释了模型是怎么输出的了。

然后利用 Softmax 层将这些分数转换为概率。概率最大的单元格对应的单词作为此时的输出。



3.9 Training

到目前为止我们介绍玩了 Transformer 的前向传播过程，我们再来看一下它是如何训练的。

训练时我们需要一个标注好的数据集。

为了形象化，我们假设词汇表只包含 5 个单词和一个结束符号：

Output Vocabulary

WORD	a	am	I	thanks	student	<eos>
INDEX	0	1	2	3	4	5

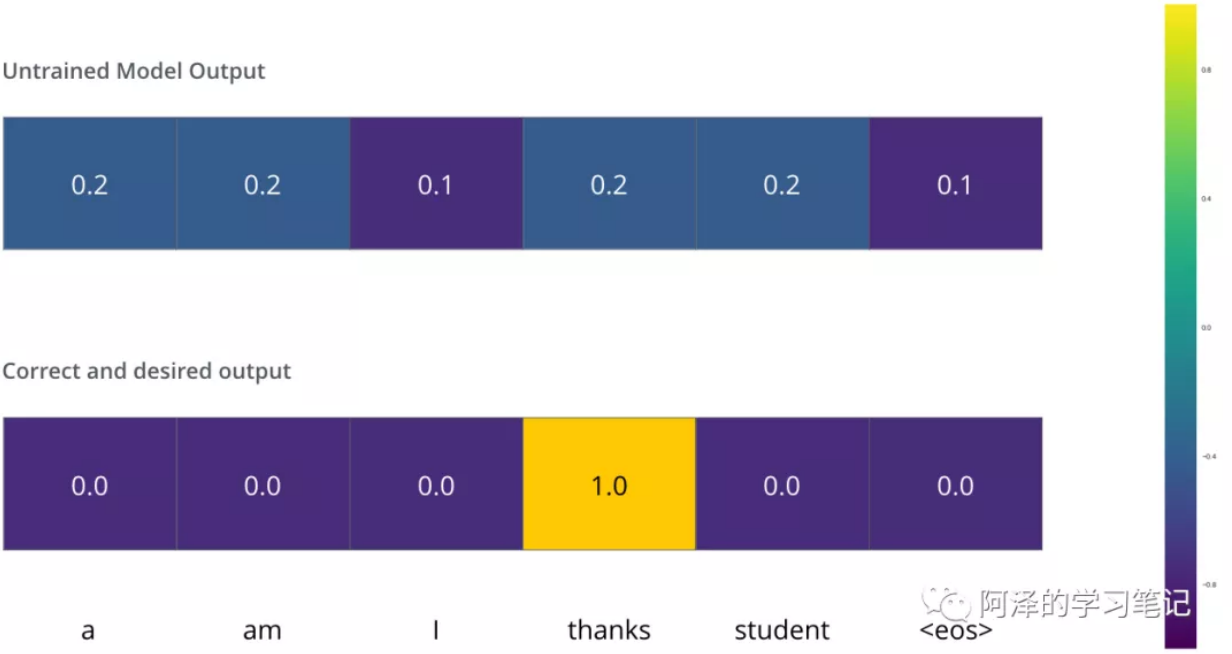
然后我们给每个单词一个 One-Hot 向量：

Output Vocabulary

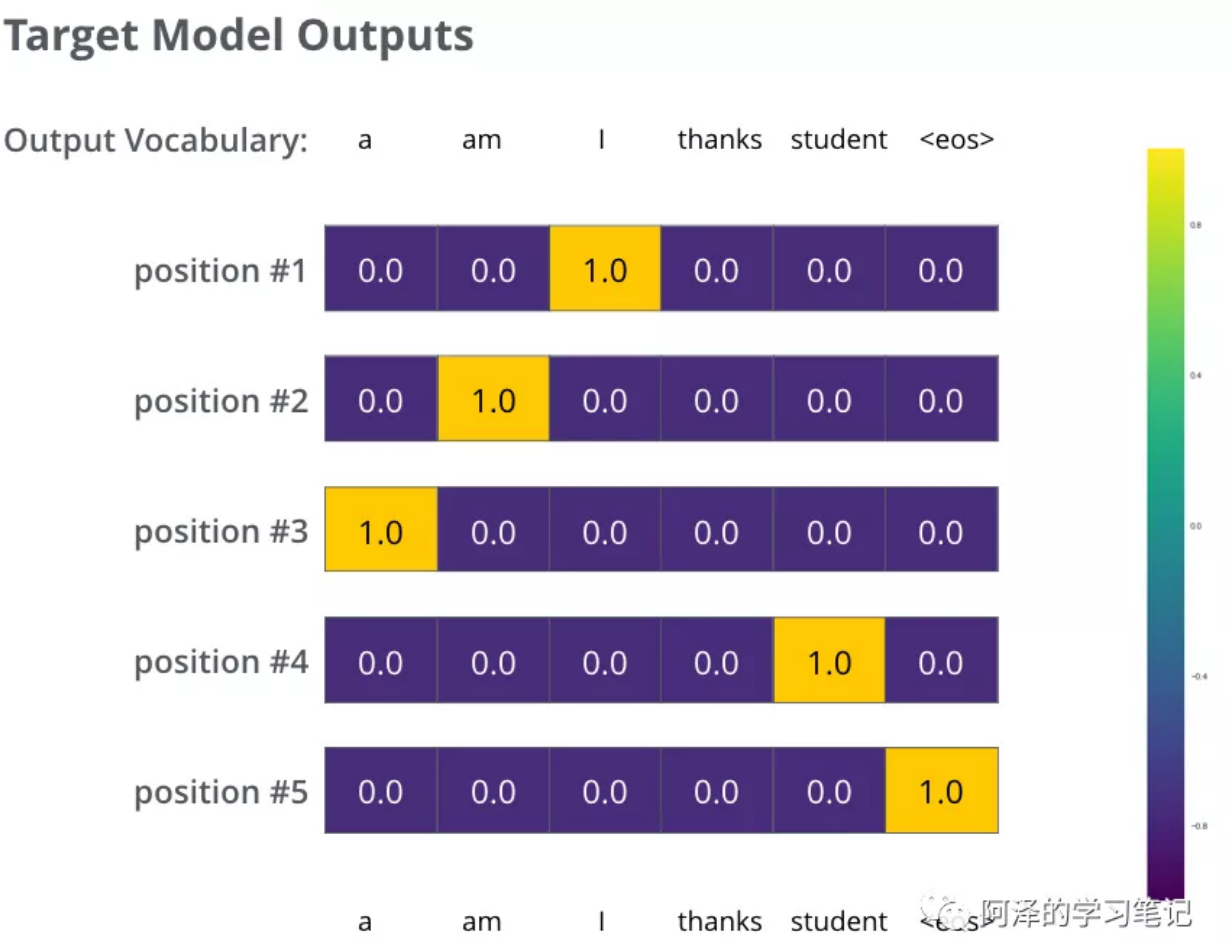
WORD	a	am	I	thanks	student	<eos>
INDEX	0	1	2	3	4	5

One-hot encoding of the word “am”

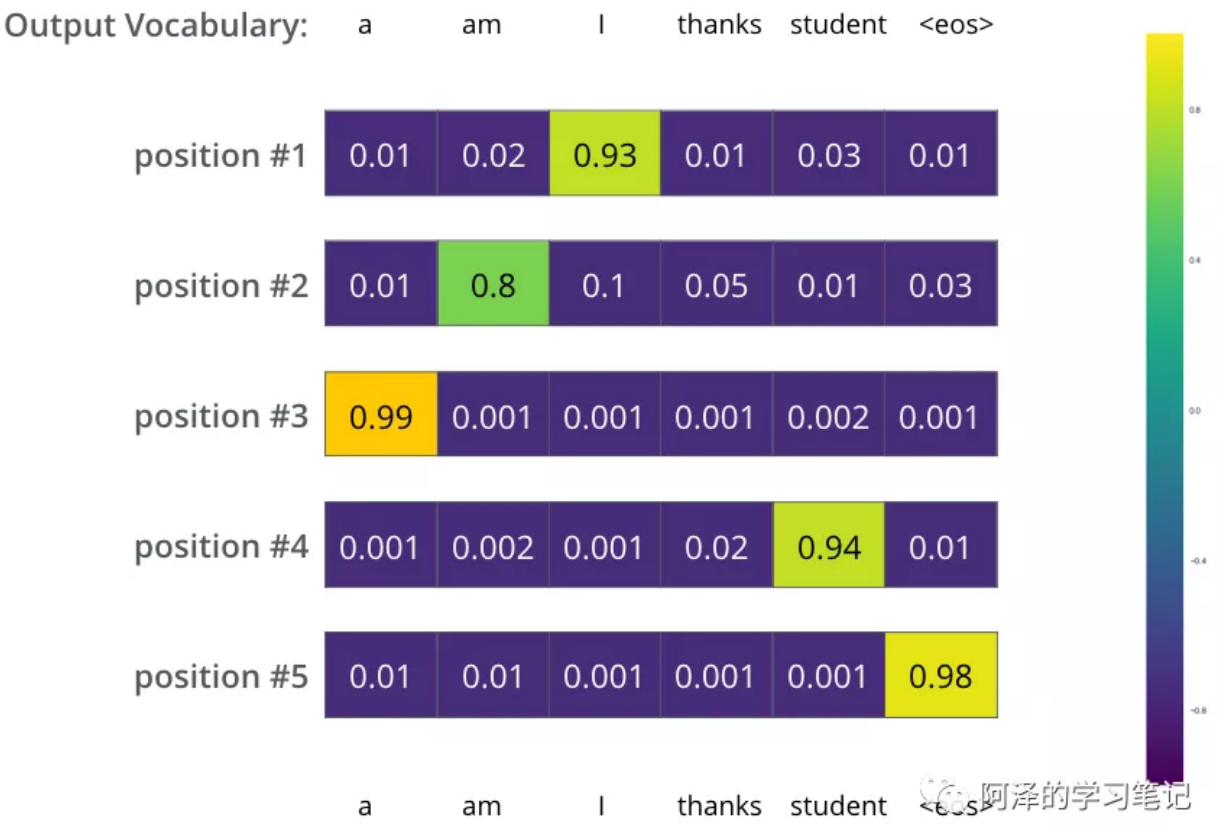
假设我们刚开始进行训练，模型的参数都是随机初始化的，所以模型的输出和期望输出有所偏差。



我们计算两者的损失函数并通过反向传播的方式来更新模型。



在一个足够大的数据集上对模型进行足够长的时间的训练之后，我们希望生成的概率分布是这样的：



当训练好的得到模型后，我们需要为某个句子进行翻译。有两种方式确定输出单词：

- Greedy Decoding：直接取概率最大的那个单词；
- Beam Search：取最大的几个单词作为候选，分别运行一次模型，然后看一下哪组错误更少。这里的超参成为 Beam Size。

以上介绍完了 Transformer。

4. Conclusion

总结：Transformer 提出了 Self-Attention 方式来代替 RNN 从而防止出现梯度消失和无法并行化的问题，并通过 Multi-Head Attention 机制集成了 Attention 丰富了特征的表达，最终在精度和速度上较 Seq2Seq 模型都有了很大的提升。

5. Reference

1. 《Attention is all you need》

2. 《The Illustrated Transformer》
3. 《Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)》