

Transformer

参考：

- [Attention Is All You Need](#)
- [图解Transformer](#)
- [李宏毅ppt](#)

前言：前一段时间谷歌推出的BERT模型在11项NLP任务中夺得SOTA结果，引爆了整个NLP界。而BERT取得成功的一个关键因素是Transformer的强大作用。谷歌的Transformer模型最早是用于机器翻译任务，当时达到了SOTA效果。Transformer改进了RNN最被人诟病的训练慢的缺点，利用self-attention机制实现快速并行。并且Transformer可以增加到非常深的深度，充分发掘DNN模型的特性，提升模型准确率。在本文中，我们将研究Transformer模型，把它掰开揉碎，理解它的工作原理。

Transformer由论文《Attention is All You Need》提出，现在是谷歌云TPU推荐的参考模型。论文相关的Tensorflow的代码可以从GitHub获取，其作为Tensor2Tensor包的一部分。哈佛的NLP团队也实现了一个基于PyTorch的版本，并注释该论文。

在本文中，我们将试图把模型简化一点，并逐一介绍里面的核心概念，希望让普通读者也能轻易理解。

Attention is All You Need: <https://arxiv.org/abs/1706.03762>

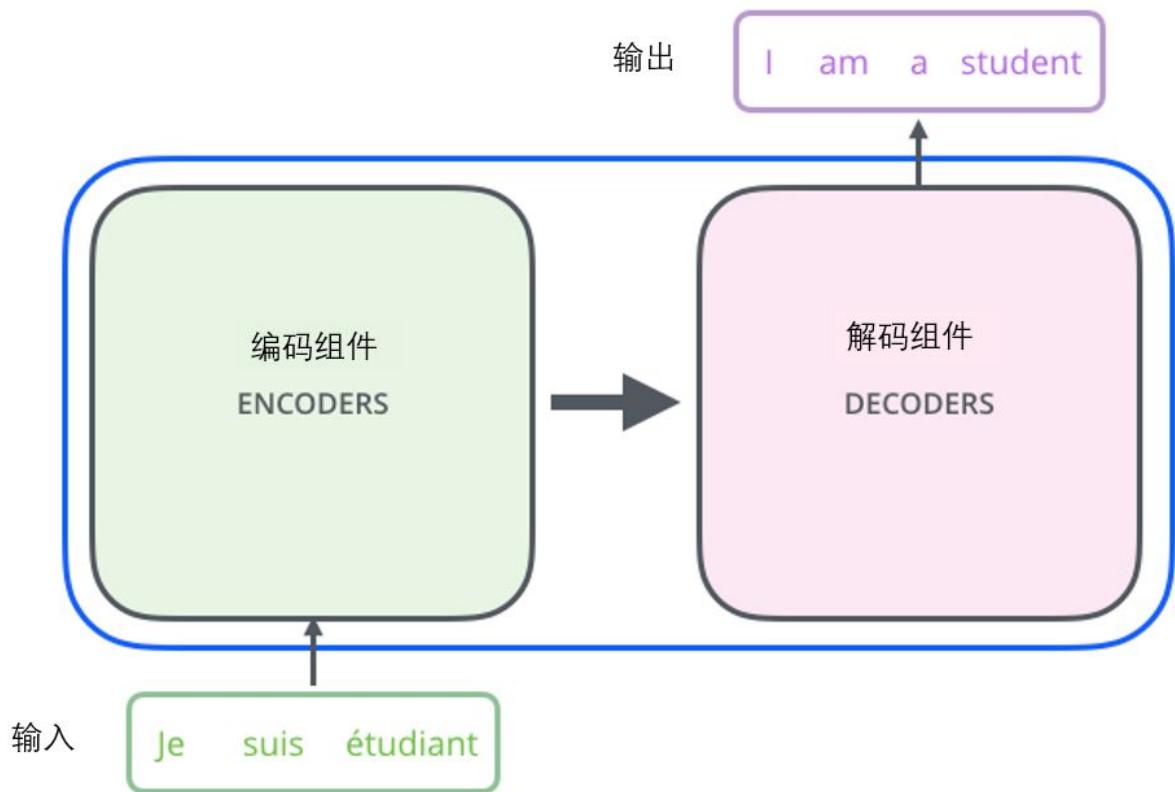
铺垫准备工作

宏观：Encode Decode模型

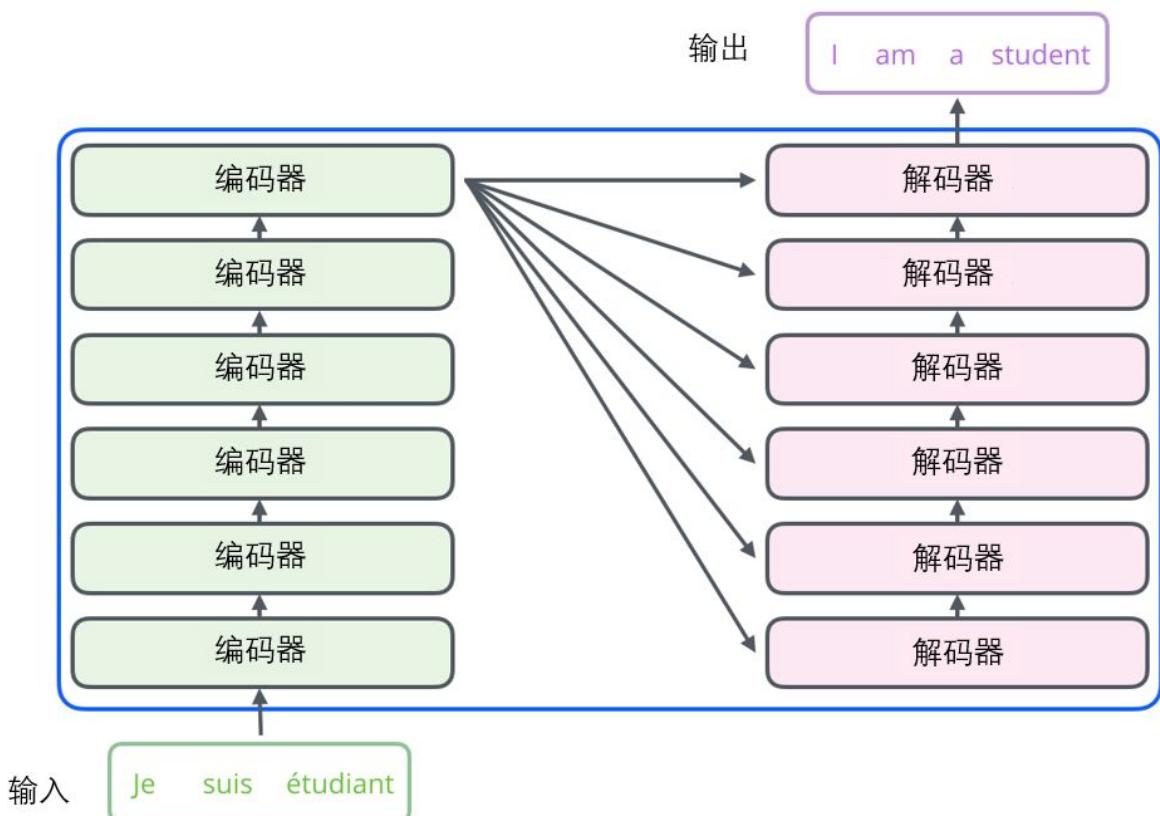
首先将这个模型看成是一个黑箱操作。在机器翻译中，就是输入一种语言，输出另一种语言。



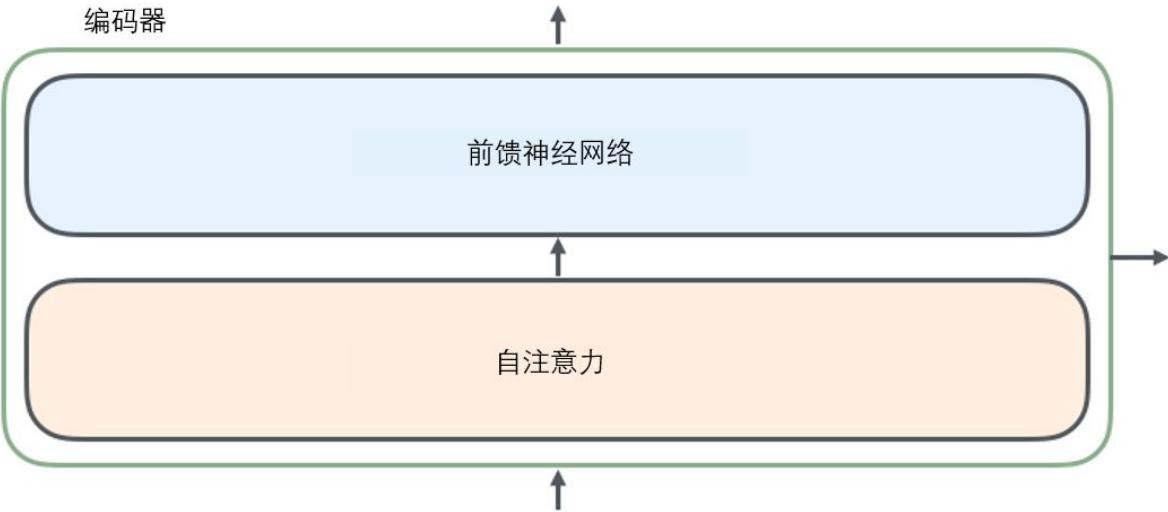
那么拆开这个黑箱，我们可以看到它是由编码组件、解码组件和它们之间的连接组成。



编码组件部分由一堆编码器（encoder）构成（论文中是将6个编码器叠在一起——数字6没有什么神奇之处，你也可以尝试其他数字）。解码组件部分也是由相同数量（与编码器对应）的解码器（decoder）组成的。



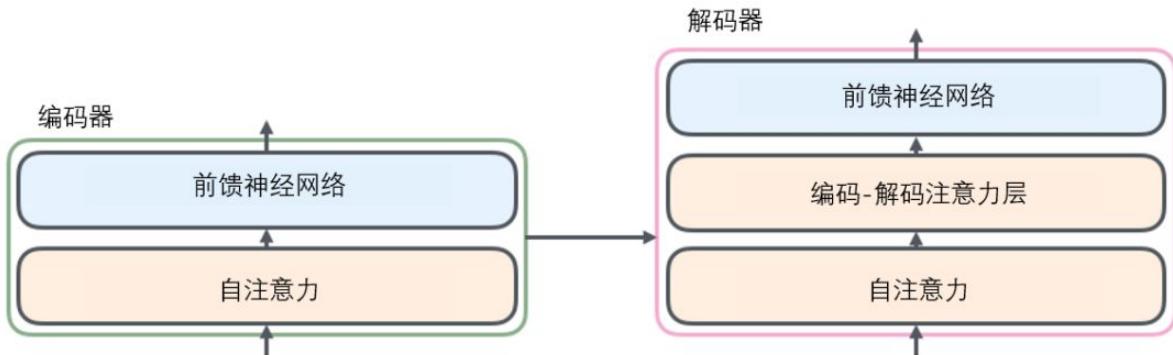
所有的编码器在结构上都是相同的，但它们没有共享参数。每个解码器都可以分解成两个子层。



从编码器输入的句子首先会经过一个自注意力 (self-attention) 层，这层帮助编码器在对每个单词编码时关注输入句子的其他单词。我们将在稍后的文章中更深入地研究自注意力。

自注意力层的输出会传递到前馈 (feed-forward) 神经网络中。每个位置的单词对应的前馈神经网络都完全一样（译注：另一种解读就是一层窗口为一个单词的一维卷积神经网络）。

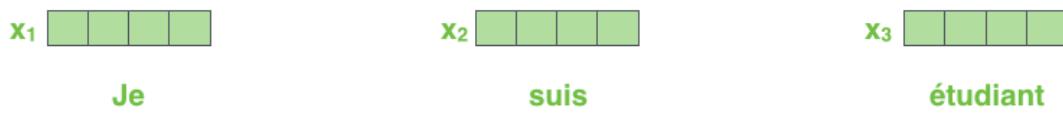
解码器中也有编码器的自注意力 (self-attention) 层和前馈 (feed-forward) 层。除此之外，这两个层之间还有一个注意力层，用来关注输入句子的相关部分（和seq2seq模型的注意力作用相似）。



将张量引入图景

我们已经了解了模型的主要部分，接下来我们看一下各种向量或张量（译注：张量概念是矢量概念的推广，可以简单理解矢量是一阶张量、矩阵是二阶张量。）是怎样在模型的不同部分中，将输入转化为输出的。

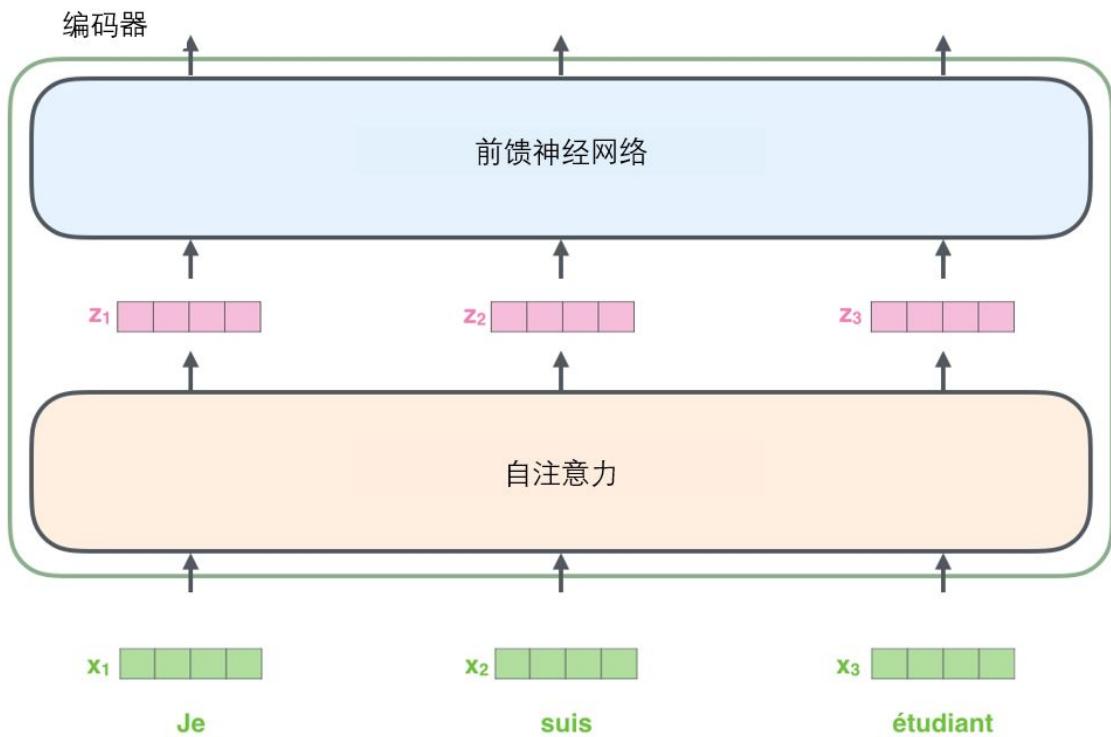
像大部分NLP应用一样，我们首先将每个输入单词通过词嵌入算法转换为词向量。



每个单词都被嵌入为512维的向量，我们用这些简单的方框来表示这些向量。

词嵌入((Word Embedding))过程发生在最底层的编码器之前。所有的编码器都有一个相同的特点，即它们接收一个向量列表，列表中的每个向量大小为512维。在底层（最开始）编码器中它就是词向量，但是在其他编码器中，它就是下一层编码器的输出（也是一个向量列表）。向量列表(Embedding Size)大小是我们可以设置的超参数——般是我们训练集中最长句子的长度。

将输入序列进行词嵌入之后，每个单词都会流经编码器中的两个子层。



接下来我们看看Transformer的一个核心特性，在这里输入序列中每个位置的单词都有自己独特的路径流入编码器。在自注意力层中，这些路径之间存在依赖关系。而前馈（feed-forward）层没有这些依赖关系。因此在前馈（feed-forward）层时可以并行执行各种路径。

一个编码器接收向量列表作为输入，接着将向量传递到自注意力层进行处理，然后传递到前馈神经网络层中，将输出结果传递到下一个编码器中。

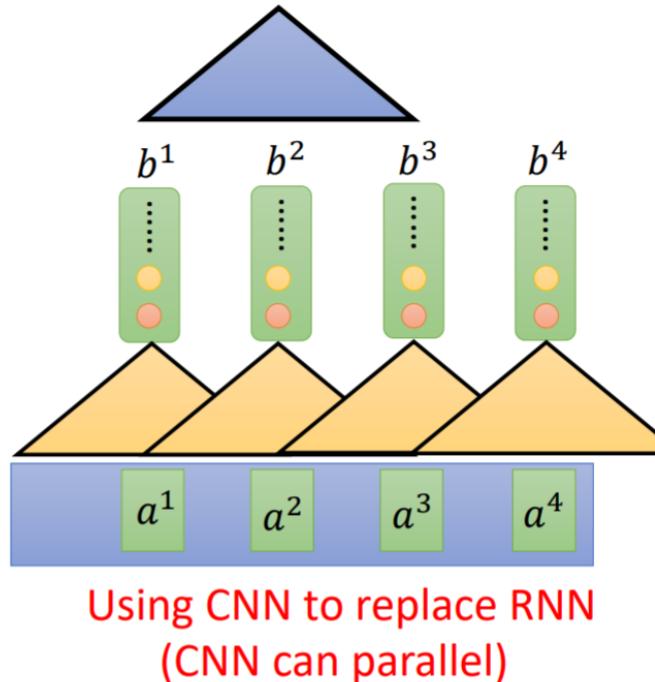
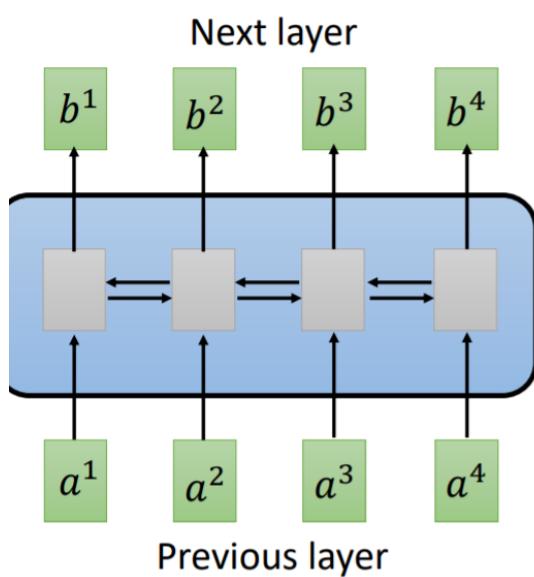
输入序列的每个单词都经过自编码过程。然后，他们各自通过前向传播神经网络——完全相同的网络，而每个向量都分别通过它。

RNN及其局限

上小节提到了编码器和解码器都用到了self-attention。提到及其翻译，最长想到的架构就是RNN。

Sequence

Filters in higher layer can consider longer sequence



RNN在输出的时候，需要看到整个输入序列。弊端：不容易平行化。比如要计算 b_4 ,需要看到 a_1-a_4 .

也有人提出用CNN替换，CNN可以平行化。缺点只能看到很小的部分，解决方法，利用多层CNN，也可以使模型看到很远。

所以有人提出用self-attention替换RNN，任何可以用RNN做到的事情，都可以用self-attention做到。

自注意力机制

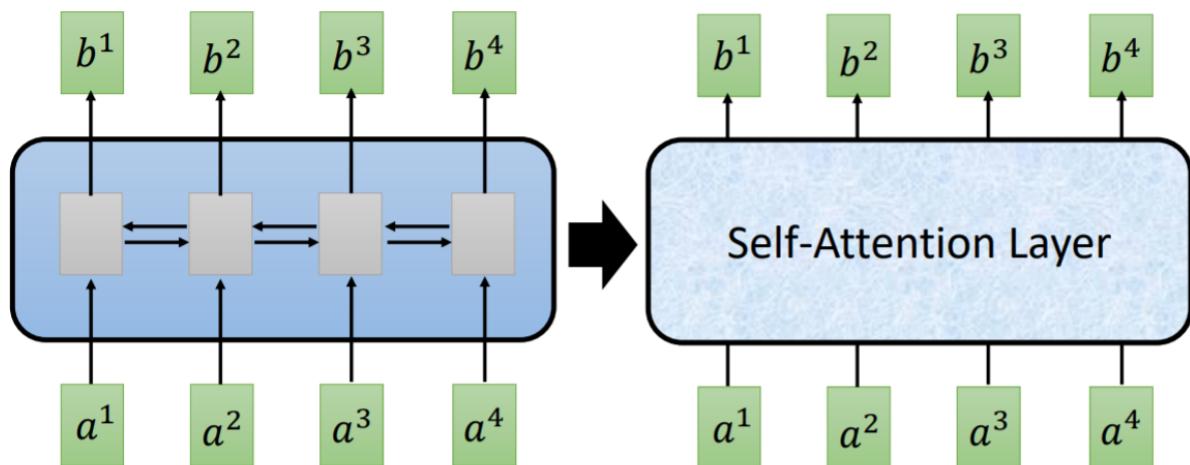
模型结构。RNN在计算 b_4 的时候，需要计算出 a_3 的值以此来推，它弥补了RNN的不足， b_1-b_4 都是可以同时平行计算出来的，可同时看到了 a_1-a_4 。

事实上上边说的是编码器部分，编码器self-attention是可以平行计算的。对于解码器，是像RNN一样逐个计算出的，因为当前输出需要用到之前的输出。

Self-Attention

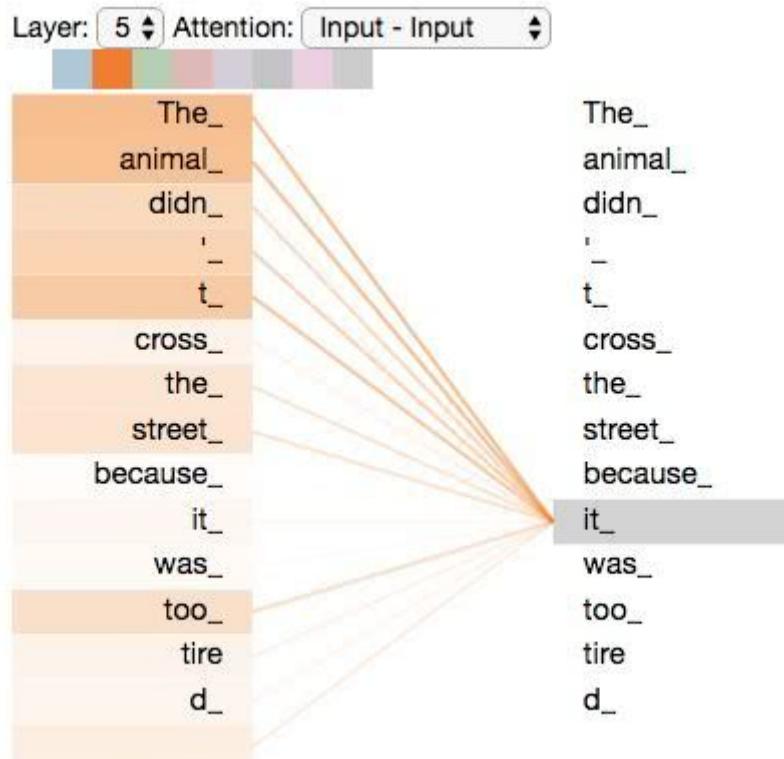
b^i is obtained based on the whole input sequence.

b^1, b^2, b^3, b^4 can be parallelly computed.



You can try to replace any thing that has been done by RNN with self-attention.

self -attention 可以看attention下的文章。



随着模型处理输入序列的每个单词，自注意力会关注整个输入序列的所有单词，帮助模型对本单词更好地进行编码。

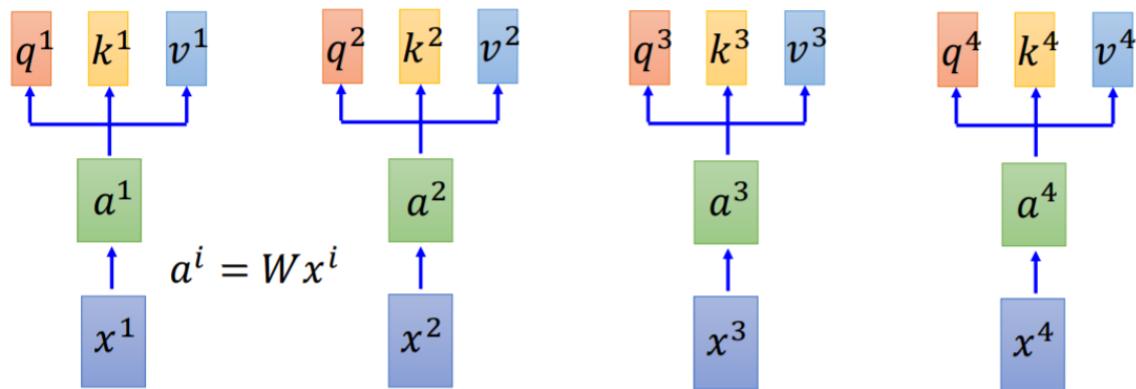
self-attention的微观实现

首先我们了解一下如何使用向量来计算自注意力，然后来看它实怎样用矩阵来实现。

首先介绍李宏毅课程的理解：

Self-attention

<https://arxiv.org/abs/1706.03762>



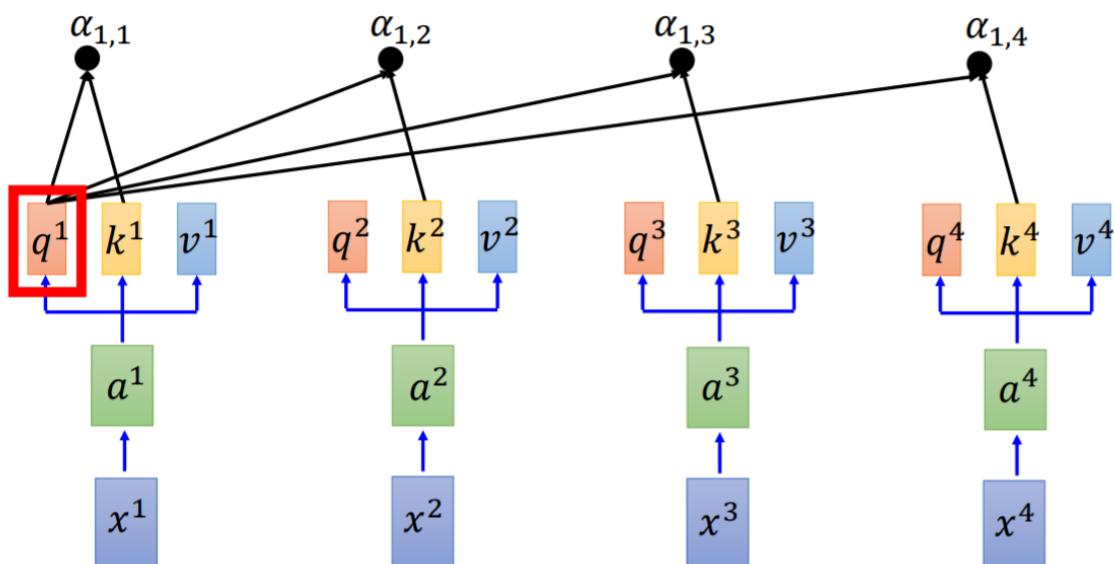
首先输入序列 x^1-x^4 ，通过乘一个 W 矩阵，这个过程就是embedding，也可以理解通过一个全连接层。得到的 a^1-a^4 就是embedding后的词向量。分别于 Wq , Wk , Wv 乘得到 q, k, v

Self-attention

拿每個 query q 去對每個 key k 做 attention

d is the dim of q and k

$$\text{Scaled Dot-Product Attention: } \alpha_{1,i} = \underbrace{q^1 \cdot k^i}_{\text{dot product}} / \sqrt{d}$$



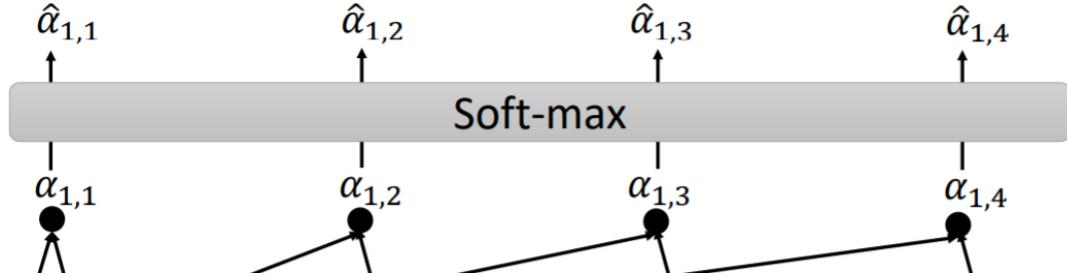
然后拿每个q去对每个k做attention。

在 [Attention Is All You Need](#) 这篇论文中，有提到两种较为常见的注意力机制：additive attention 和 dot-product attention。并讨论到，当 query 和 key 向量维度 dk 较小时，这两种注意力机制效果相当，但当 dk 较大时，additive attention 要优于 dot-product attention. 但是 dot-product attention 在计算方面更具有优势。为了利用 dot-product attention 的优势且消除当 dk 较大时 dot-product attention 的不足，原文采用 scaled dot-product attention。

scaled的意思是加了缩放因子，就是除了根号d。原因是因为点积得到的维度都很大，使得结果处于 softmax函数梯度很小的区域，需要除一个来平衡。

Self-attention

$$\hat{\alpha}_{1,i} = \exp(\alpha_{1,i}) / \sum_j \exp(\alpha_{1,j})$$

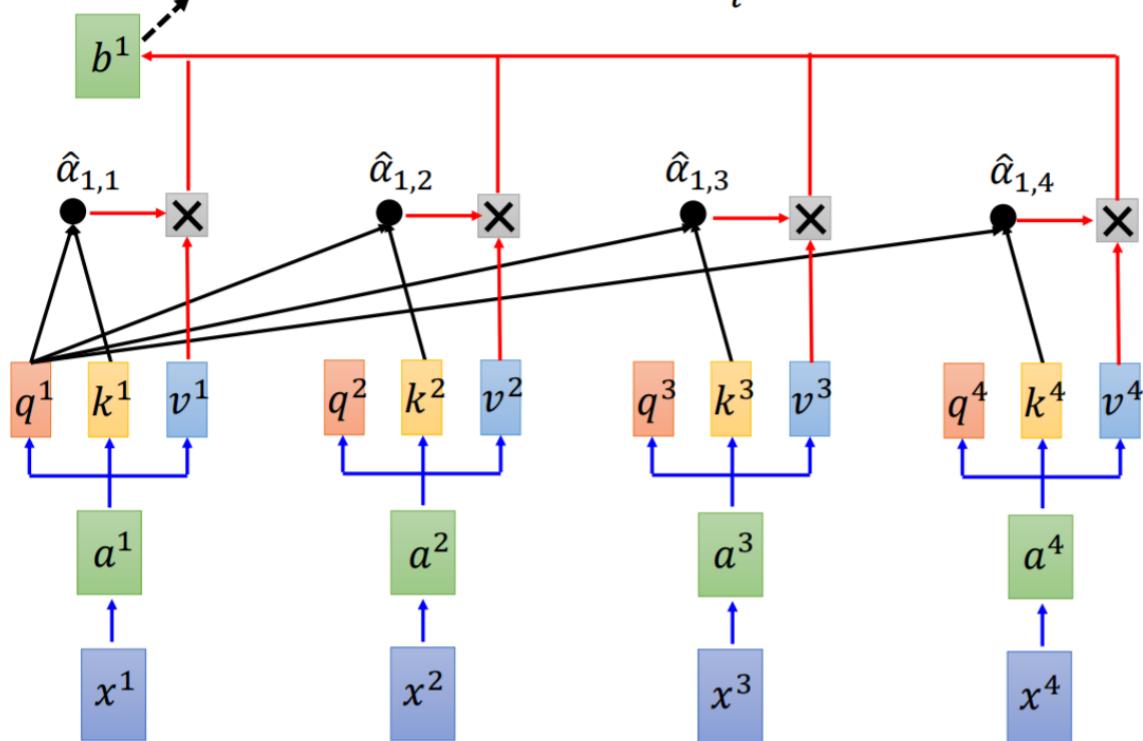


将每个 α 通过soft-max。这个softmax分数决定了每个单词对编码当下位置的贡献。显然，已经在这个位置上的单词将获得最高的softmax分数，但有时关注另一个与当前单词相关的单词也会有帮助。

Self-attention

Considering the whole sequence

$$b^1 = \sum_i \hat{\alpha}_{1,i} v^i$$

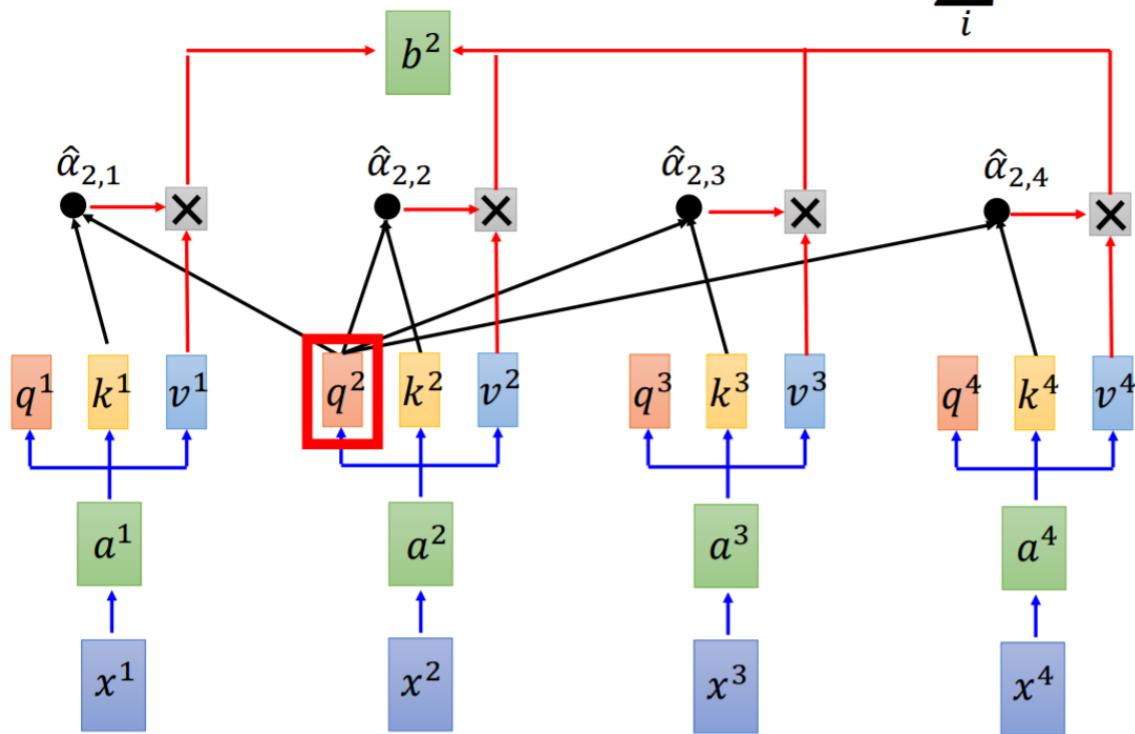


b^1 是用到了整个序列。

Self-attention

拿每個 query q 去對每個 key k 做 attention

$$b^2 = \sum_i \hat{a}_{2,i} v^i$$



同理，在同一时间可以计算 b_2, \dots 结果都是并行计算出来的。下面通过微观矩阵进行表示，更能说明为什么可以平行化。

Self-attention

$$\begin{matrix} q^1 & q^2 & q^3 & q^4 \end{matrix} = \begin{matrix} W^q \\ Q \end{matrix} \quad \begin{matrix} a^1 & a^2 & a^3 & a^4 \end{matrix} = \begin{matrix} I \\ I \end{matrix}$$

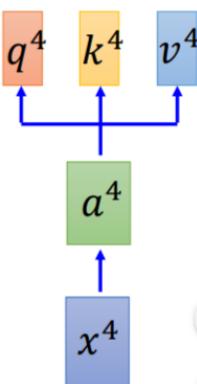
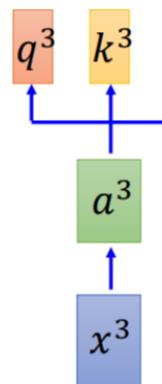
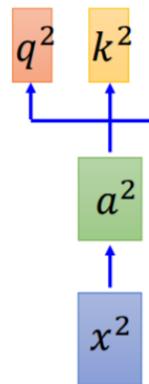
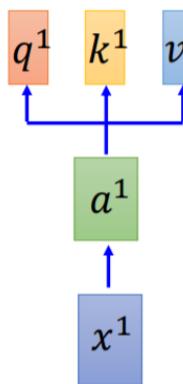
$$q^i = W^q a^i$$

$$\begin{matrix} k^1 & k^2 & k^3 & k^4 \end{matrix} = \begin{matrix} W^k \\ K \end{matrix} \quad \begin{matrix} a^1 & a^2 & a^3 & a^4 \end{matrix} = \begin{matrix} I \\ I \end{matrix}$$

$$k^i = W^k a^i$$

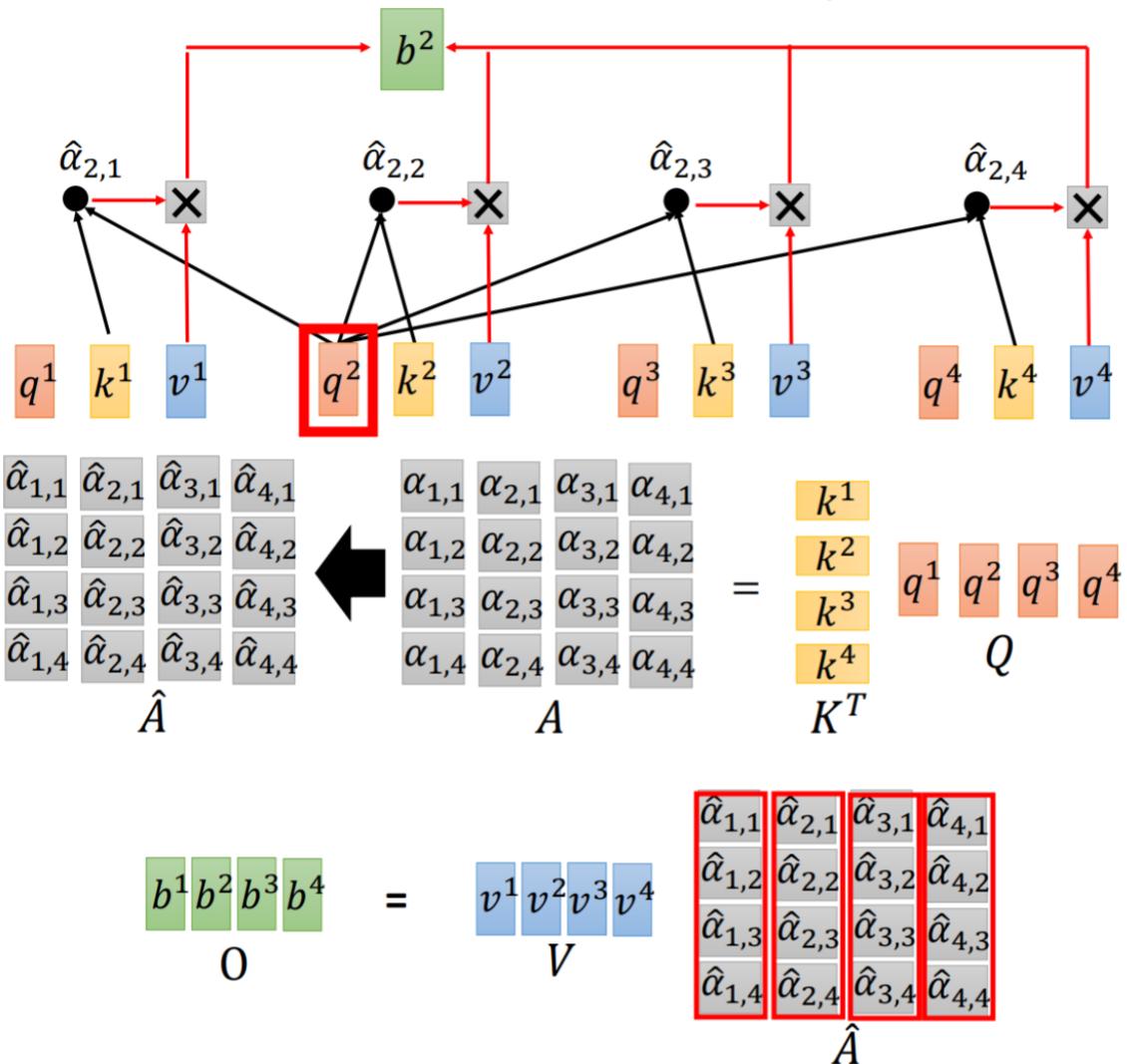
$$v^i = W^v a^i$$

$$\begin{matrix} v^1 & v^2 & v^3 & v^4 \end{matrix} = \begin{matrix} W^v \\ V \end{matrix} \quad \begin{matrix} a^1 & a^2 & a^3 & a^4 \end{matrix} = \begin{matrix} I \\ I \end{matrix}$$



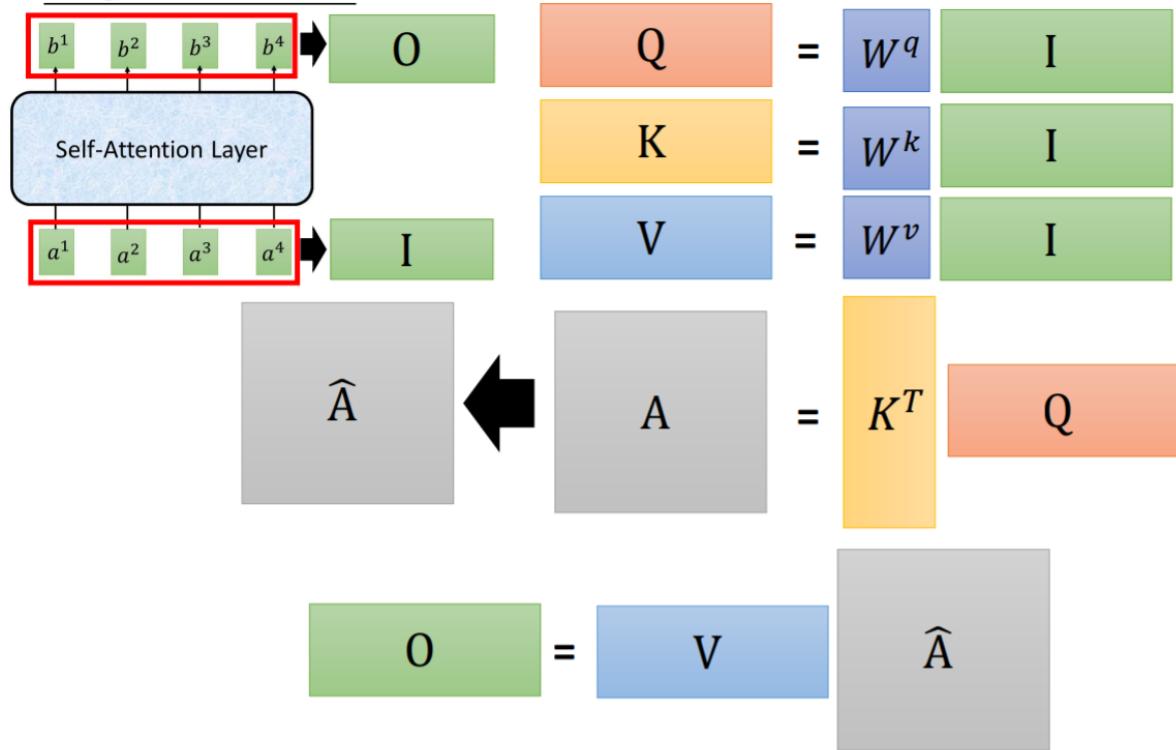
Self-attention

$$b^2 = \sum_i \hat{\alpha}_{2,i} v^i$$



得到 \hat{A} (经过 soft-max) 之后，最后输出结果是经过加权求和。所以 $output = V \cdot \hat{A}$

Self-attention



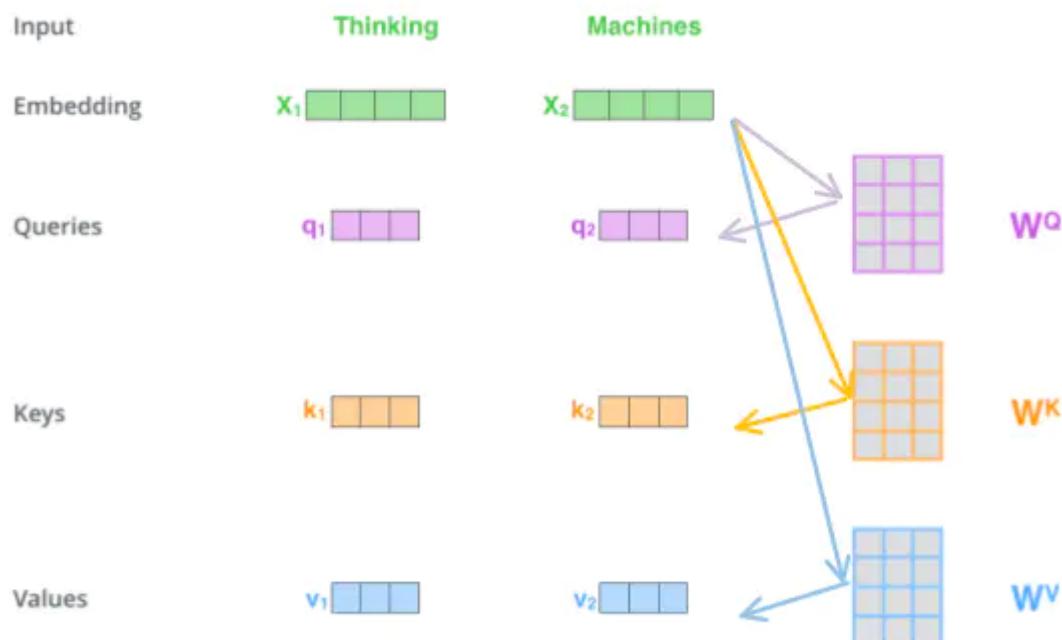
反正就是一堆矩阵乘法，用 GPU 可以加速

下边是另一篇文章的解释

意思大体相同

计算自注意力的第一步就是从每个编码器的输入向量（每个单词的词向量）中生成三个向量。也就是说对于每个单词，我们创造一个查询向量、一个键向量和一个值向量。这三个向量是通过词嵌入与三个权重矩阵后相乘创建的。

可以发现这些新向量在维度上比词嵌入向量更低。他们的维度是64，而词嵌入和编码器的输入/输出向量的维度是512. 但实际上不强求维度更小，这只不过是一种基于架构上的选择，它可以使多头注意力（multiheaded attention）的大部分计算保持不变。



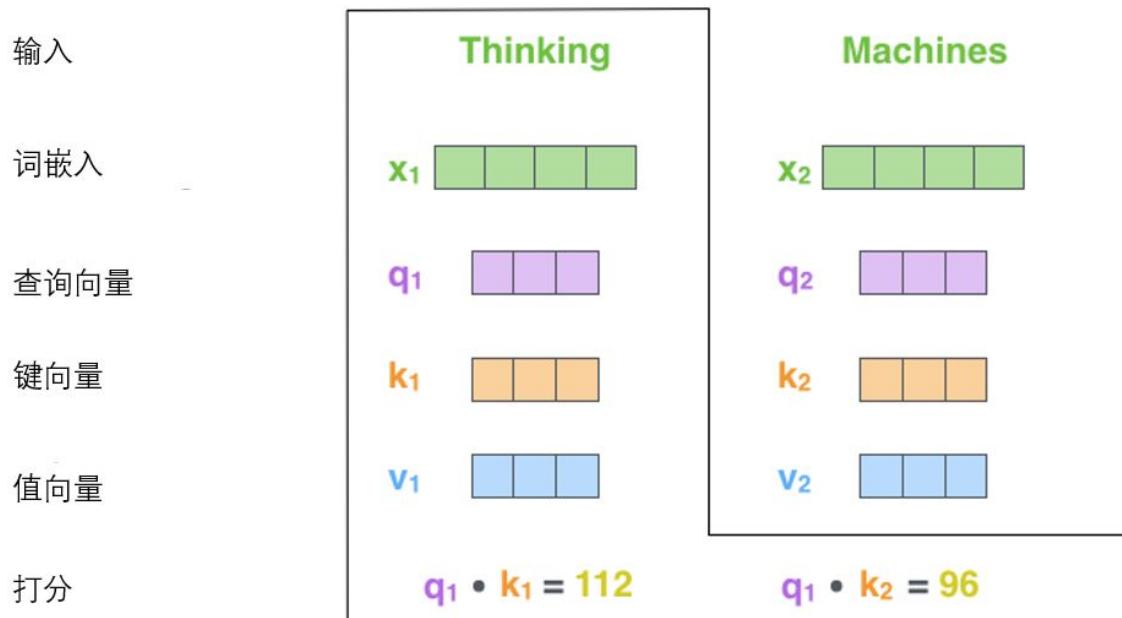
X_1 与 WQ 权重矩阵相乘得到 q_1 , 就是与这个单词相关的查询向量。最终使得输入序列的每个单词的创建一个查询向量、一个键向量和一个值向量。

什么是查询向量、键向量和值向量向量？

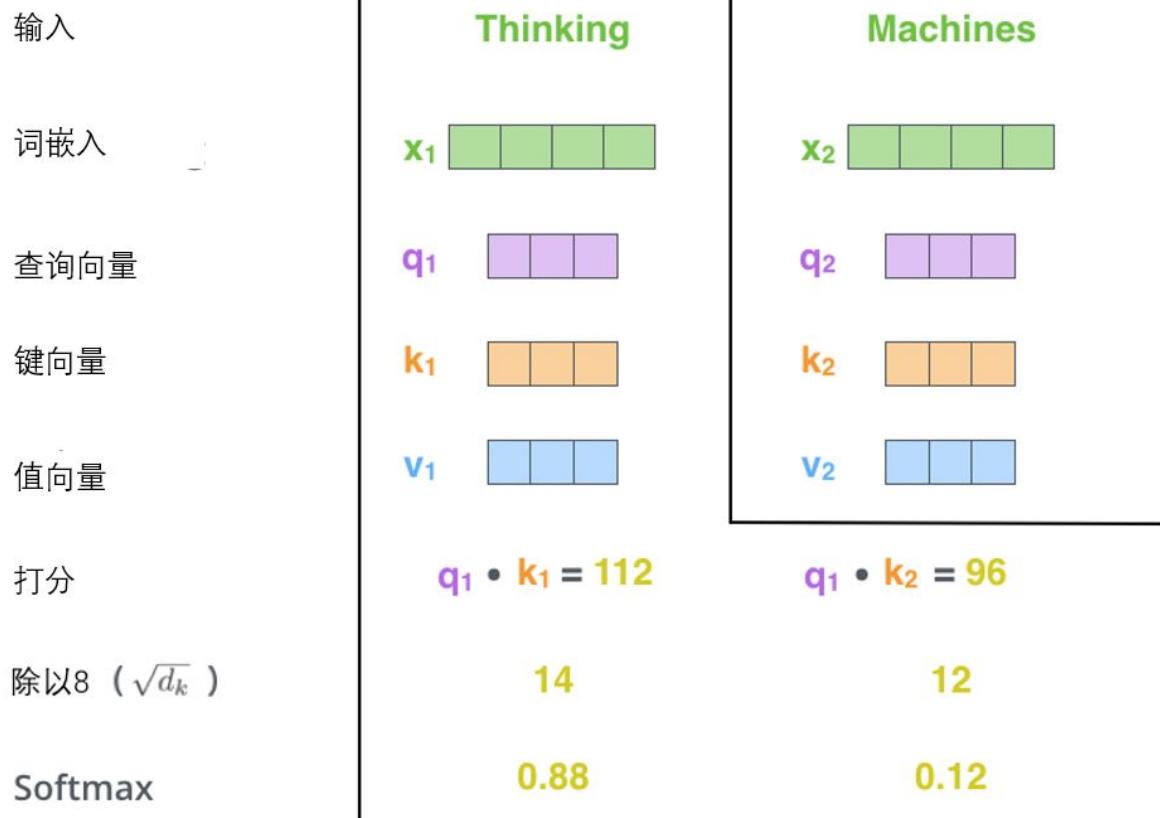
它们都是有助于计算和理解注意力机制的抽象概念。请继续阅读下文的内容，你就会知道每个向量在计算注意力机制中到底扮演什么样的角色。

计算自注意力的第二步是计算得分。假设我们在为这个例子中的第一个词“Thinking”计算自注意力向量，我们需要拿输入句子中的每个单词对“Thinking”打分。这些分数决定了在编码单词“Thinking”的过程中有多重视句子的其它部分。

这些分数是通过打分单词（所有输入句子的单词）的键向量与“Thinking”的查询向量相点积来计算的。所以如果我们是处理位置最靠前的词的自注意力的话，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。



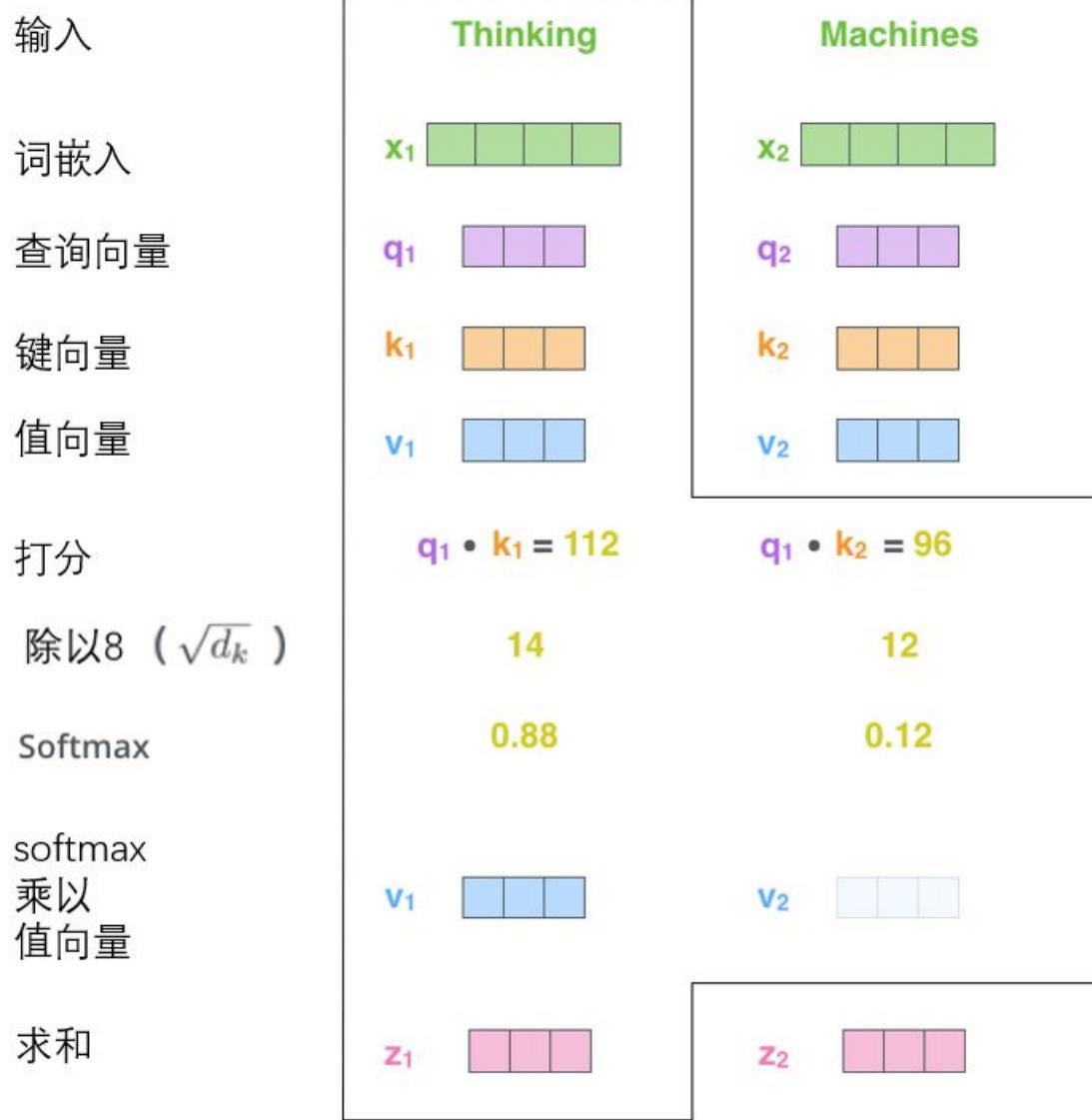
第三步和第四步是将分数除以8(8是论文中使用的键向量的维数64的平方根，这会让梯度更稳定。这里也可以使用其它值，8只是默认值)，然后通过softmax传递结果。softmax的作用是使所有单词的分数归一化，得到的分数都是正值且和为1。



这个softmax分数决定了每个单词对编码当下位置 (“Thinking”) 的贡献。显然，已经在这个位置上的单词将获得最高的softmax分数，但有时关注另一个与当前单词相关的单词也会有帮助。

第五步是将每个值向量乘以softmax分数(这是为了准备之后将它们求和)。这里的直觉是希望关注语义上相关的单词，并弱化不相关的单词(例如，让它们乘以0.001这样的小数)。

第六步是对加权值向量求和 (译注：自注意力的另一种解释就是在编码某个单词时，就是将所有单词的表示 (值向量) 进行加权求和，而权重是通过该词的表示 (键向量) 与被编码词表示 (查询向量) 的点积并通过softmax得到。)，然后即得到自注意力层在该位置的输出(在我们的例子中是对于第一个单词)。



这样自注意力的计算就完成了。得到的向量就可以传给前馈神经网络。然而实际上，这些计算是以矩阵形式完成的，以便算得更快。那我们接下来就看看如何用矩阵实现的。

通过矩阵运算实现自注意力机制

第一步是计算查询矩阵、键矩阵和值矩阵。为此，我们将将输入句子的词嵌入装进矩阵 X 中，将其乘以我们训练的权重矩阵(WQ , WK , WV)。

$$X \times W^Q = Q$$

$$X \times W^K = K$$

$$X \times W^V = V$$

X 矩阵中的每一行对应于输入句子中的一个单词。我们再次看到词嵌入向量(512，或图中的4个格子)和q/k/v向量(64，或图中的3个格子)的大小差异。

最后，由于我们处理的是矩阵，我们可以将步骤2到步骤6合并为一个公式来计算自注意力层的输出。

$$\text{softmax} \left(\frac{Q \times K^T}{\sqrt{d_k}} \right) V = Z$$

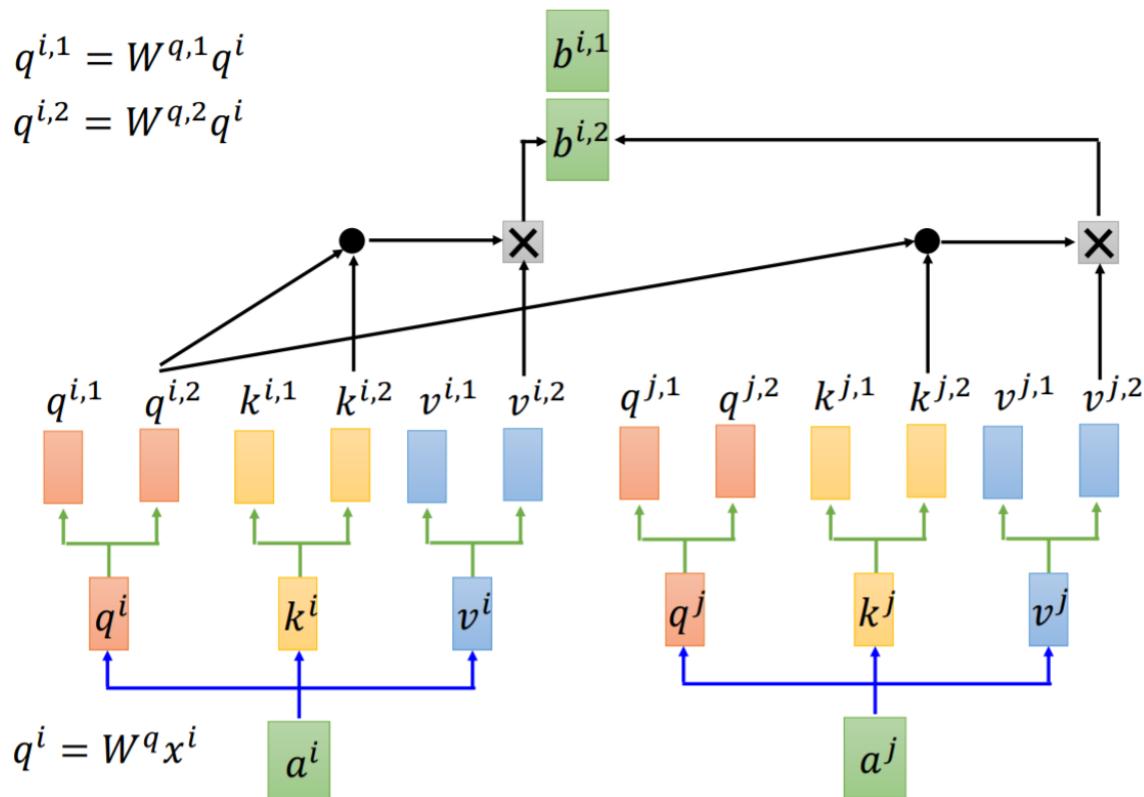
“多头”注意力

通过增加一种叫做“多头”注意力 (“multi-headed” attention) 的机制，论文进一步完善了自注意力层，并在两方面提高了注意力层的性能：

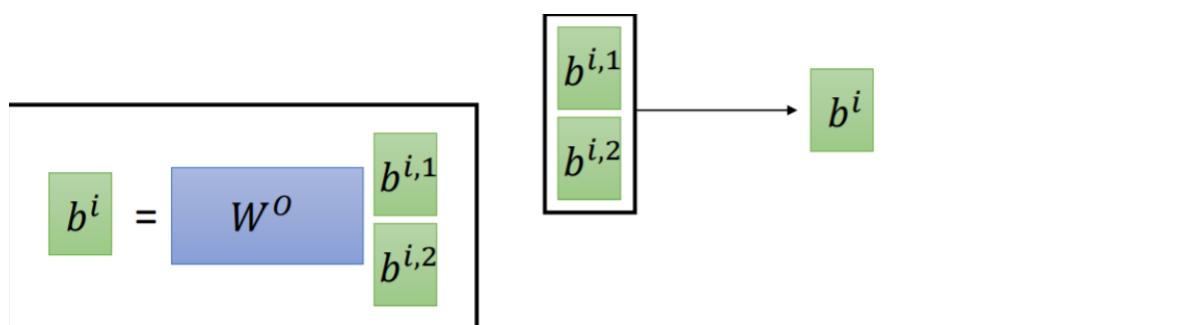
1. 它扩展了模型专注于不同位置的能力。在上面的例子中，虽然每个编码都在 z_1 中有或多或少的体现，但是它可能被实际的单词本身所支配。如果我们翻译一个句子，比如“The animal didn't cross the street because it was too tired”，我们会想知道“it”指的是哪个词，这时模型的“多头”注意机制会起到作用。

2. 它给出了注意力层的多个“表示子空间” (representation subspaces)。接下来我们将看到，对于“多头”注意机制，我们有多个查询/键/值权重矩阵集(Transformer使用八个注意力头，因此我们对于每个编码器/解码器有八个矩阵集合)。这些集合中的每一个都是随机初始化的，在训练之后，每个集合都被用来自将输入词嵌入(或来自较低编码器/解码器的向量)投影到不同的表示子空间中。

Multi-head Self-attention (2 heads as example)

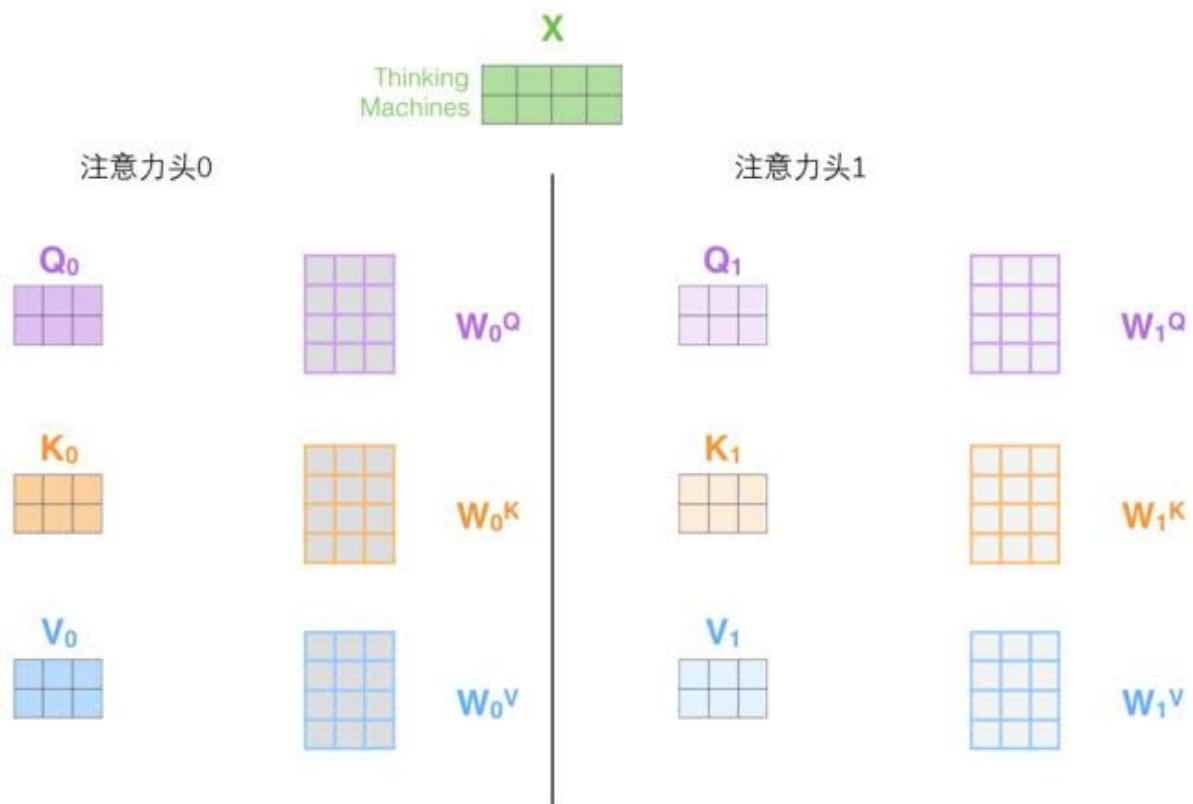


举例两个头来说，把 q_i 乘不同矩阵得到两个不同 q 。同理 k, v



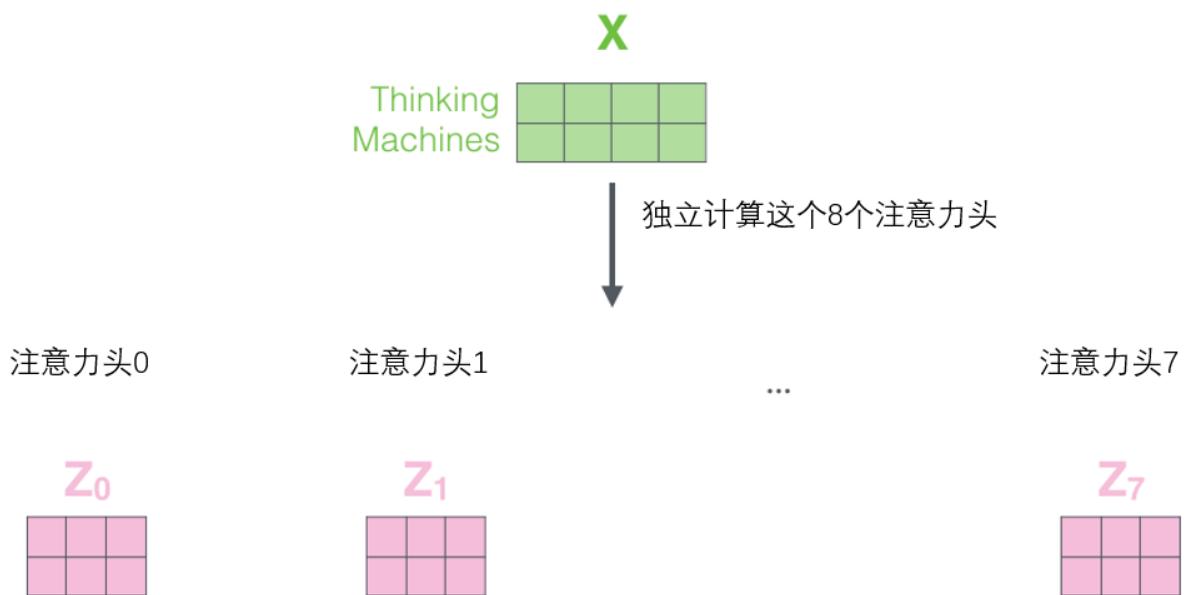
得到不同 b 的维度之后降维，每个head可以看到不同的位置。

另一个解释：



在“多头”注意力机制下，我们为每个头保持独立的查询/键/值权重矩阵，从而产生不同的查询/键/值矩阵。和之前一样，我们拿 X 乘以 $WQ/WK/WV$ 矩阵来产生查询/键/值矩阵。

如果我们做与上述相同的自注意力计算，只需八次不同的权重矩阵运算，我们就会得到八个不同的 Z 矩阵。



这给我们带来了一点挑战。前馈层不需要8个矩阵，它只需要一个矩阵(由每一个单词的表示向量组成)。所以我们需要一种方法把这八个矩阵压缩成一个矩阵。那该怎么做？其实可以直接把这些矩阵拼接在一起，然后用一个附加的权重矩阵 W_O 与它们相乘。

1) 将所有注意力头拼接起来



2) 乘以矩阵 W^0 , 它在模型中是联合训练的

x



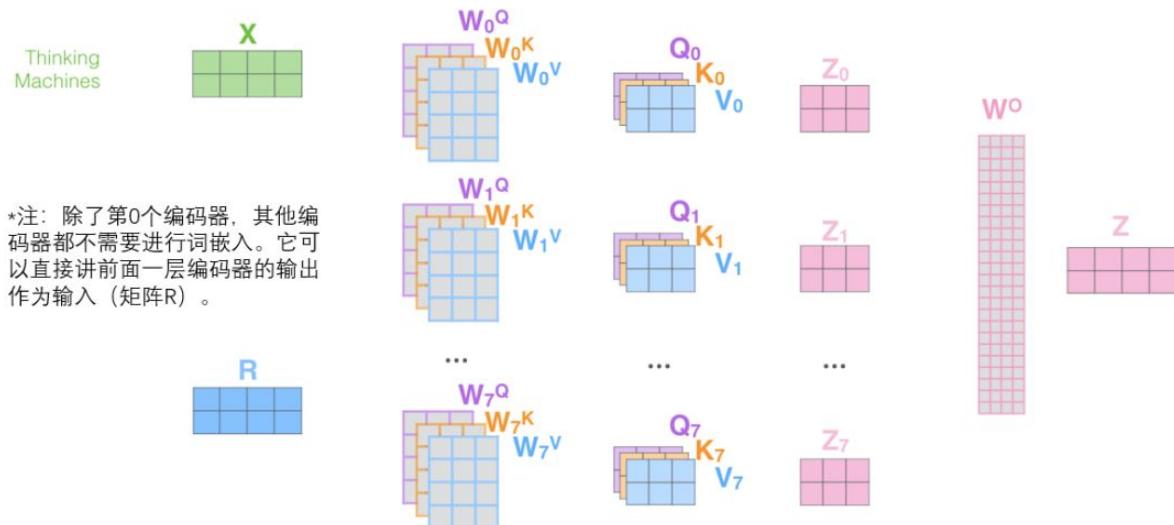
3) 结果是一个融合所有注意力头信息的矩阵Z, 我们可以将其送到前馈神经网络

$$= \begin{matrix} Z \\ \vdots \\ Z \end{matrix}$$

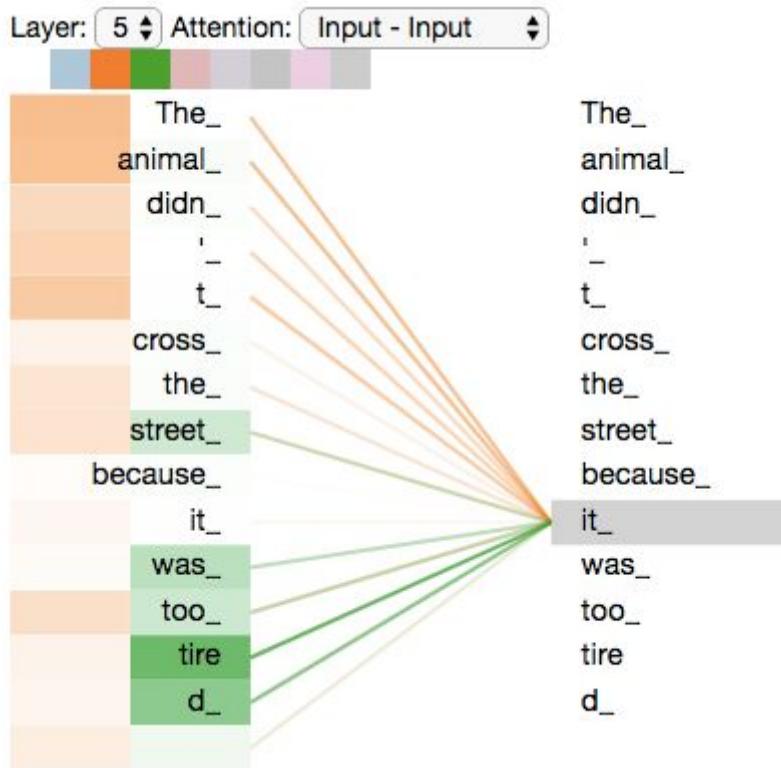
W^0

这几乎就是多头自注意力的全部。这确实有好多矩阵，我们试着把它们集中在一个图片中，这样可以一眼看清。

- 1) 这是我们输入句子*
- 2) 编码每一个单词
- 3) 将其分为8个头，将矩阵X或R乘以各个权重矩阵
- 4) 通过输出的查询/键/值 (Q/K/V) 矩阵计算注意力
- 5) 将所有注意力头拼接起来，乘以权重矩阵 W^0

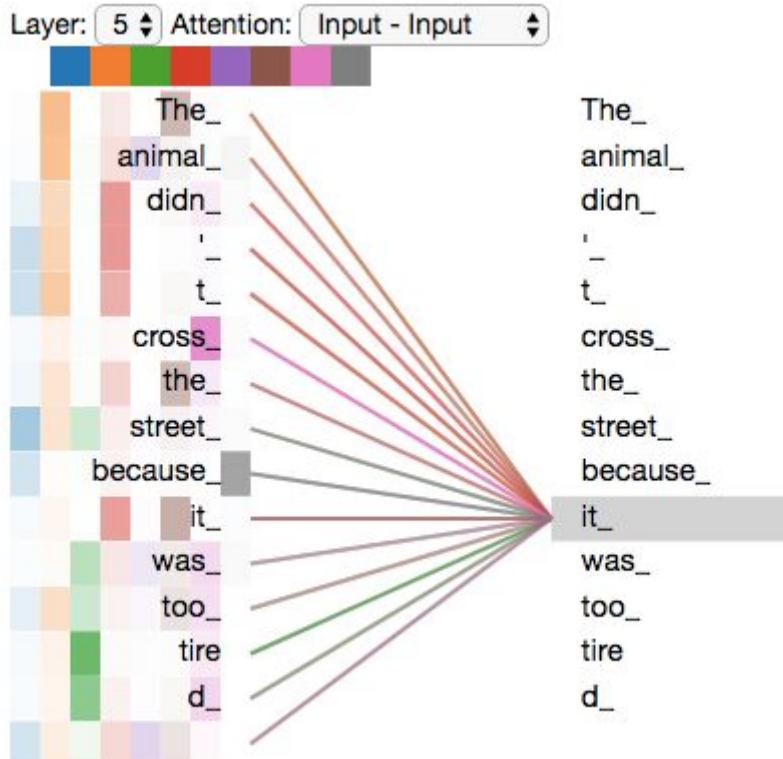


既然我们已经摸到了注意力机制的这么多“头”，那么让我们重温之前的例子，看看我们在例句中编码“it”一词时，不同的注意力“头”集中在哪里：



当我们编码“it”一词时，一个注意力头集中在“animal”上，而另一个则集中在“tired”上，从某种意义上说，模型对“it”一词的表达在某种程度上是“animal”和“tired”的代表。

然而，如果我们把所有的attention都加到图示里，事情就更难解释了：



使用位置编码表示序列的顺序

到目前为止，我们对模型的描述缺少了一种理解输入单词顺序的方法。

为了解决这个问题，Transformer为每个输入的词嵌入添加了一个向量。这些向量遵循模型学习到的特定模式，这有助于确定每个单词的位置，或序列中不同单词之间的距离。这里的直觉是，将位置向量添加到词嵌入中使得它们在接下来的运算中，能够更好地表达的词与词之间的距离。

为了让模型理解单词的顺序，我们添加了位置编码向量，这些向量的值遵循特定的模式。

如果我们假设词嵌入的维数为4，则实际的位置编码如下：

| | | | | | | | | | | | | | | | |
|------|---|-------|-------|---|---|--|------|--------|------|---|---|------|--------|-------|---|
| 位置编码 | <table border="1"><tr><td>0</td><td>0</td><td>1</td><td>1</td></tr></table> | 0 | 0 | 1 | 1 | <table border="1"><tr><td>0.84</td><td>0.0001</td><td>0.54</td><td>1</td></tr></table> | 0.84 | 0.0001 | 0.54 | 1 | <table border="1"><tr><td>0.91</td><td>0.0002</td><td>-0.42</td><td>1</td></tr></table> | 0.91 | 0.0002 | -0.42 | 1 |
| 0 | 0 | 1 | 1 | | | | | | | | | | | | |
| 0.84 | 0.0001 | 0.54 | 1 | | | | | | | | | | | | |
| 0.91 | 0.0002 | -0.42 | 1 | | | | | | | | | | | | |
| 词嵌入 | x_1 | x_2 | x_3 | | | | | | | | | | | | |

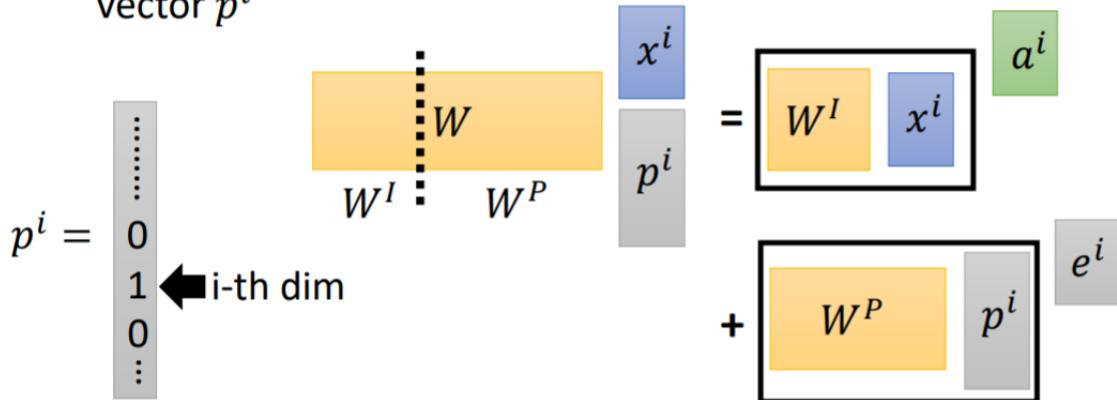
输入 je $suis$ étudiant

尺寸为4的迷你词嵌入位置编码实例

至于为什么是直接把位置向量相加。

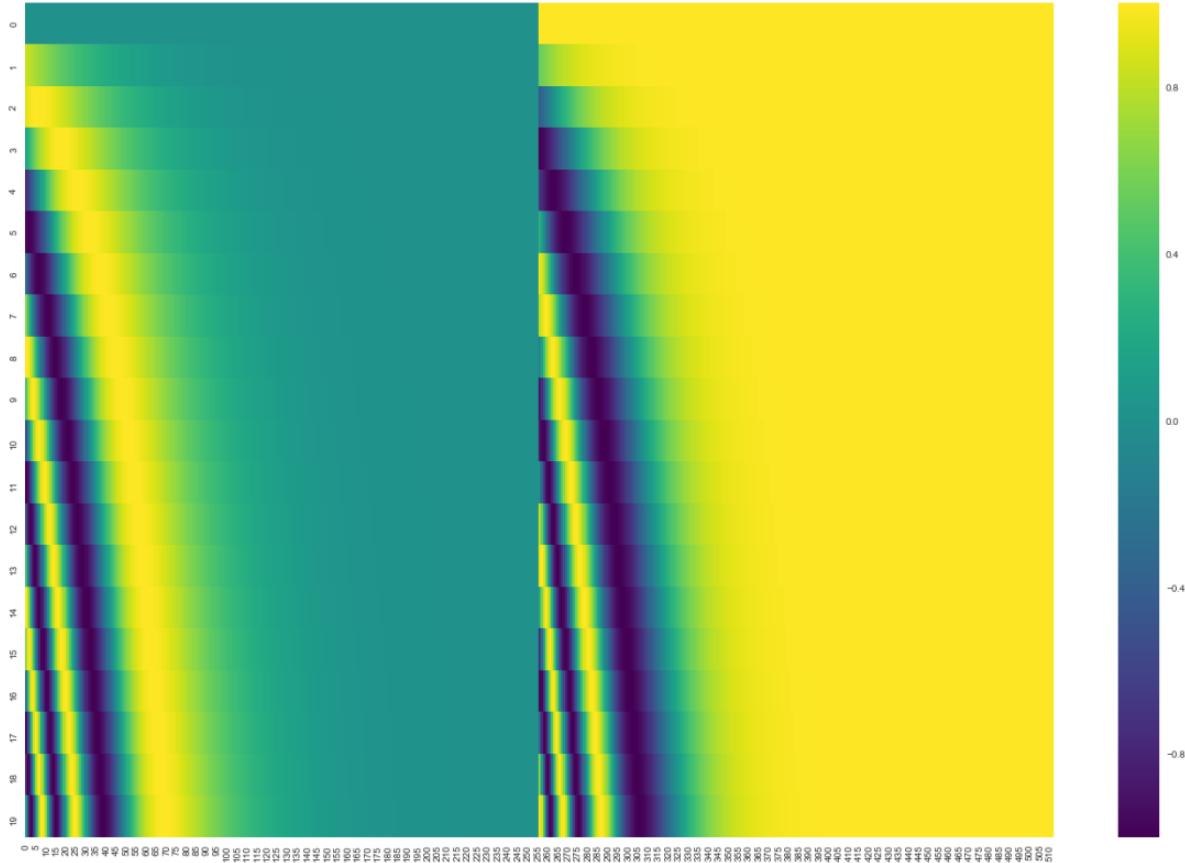
Positional Encoding

- No position information in self-attention.
- Original paper: each position has a unique positional vector e^i (not learned from data)
- In other words: each x^i appends a one-hot vector p^i



假设不是相加，而是通过外接一个one-hot向量 p_i 。也就是输入变成了 x_i+p_i ，原来的w矩阵也需要加长一个wp。计算的结果是和直接相加得到的结果相同的。wp的生成原论文是通过一个神奇算式计算出来的。

在下图中，每一行对应一个词向量的位置编码，所以第一行对应着输入序列的第一个词。每行包含512个值，每个值介于1和-1之间。我们已经对它们进行了颜色编码，所以图案是可见的。



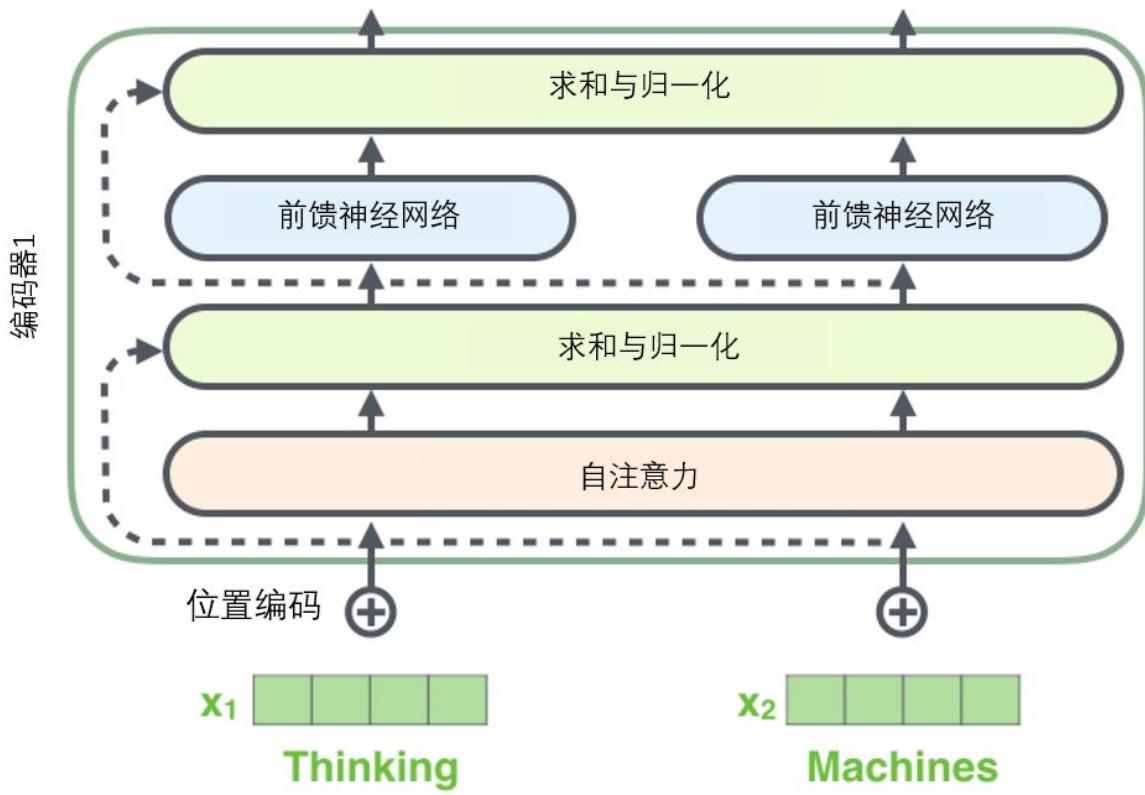
20字(行)的位置编码实例，词嵌入大小为512(列)。你可以看到它从中间分裂成两半。这是因为左半部分的值由一个函数(使用正弦)生成，而右半部分由另一个函数(使用余弦)生成。然后将它们拼在一起而得到每一个位置编码向量。

原始论文里描述了位置编码的公式(第3.5节)。你可以在 `get_timing_signal_1d()` 中看到生成位置编码的代码。这不是唯一可能的位置编码方法。然而，它的优点是能够扩展到未知的序列长度(例如，当我们训练出的模型需要翻译远比训练集里的句子更长的句子时)。

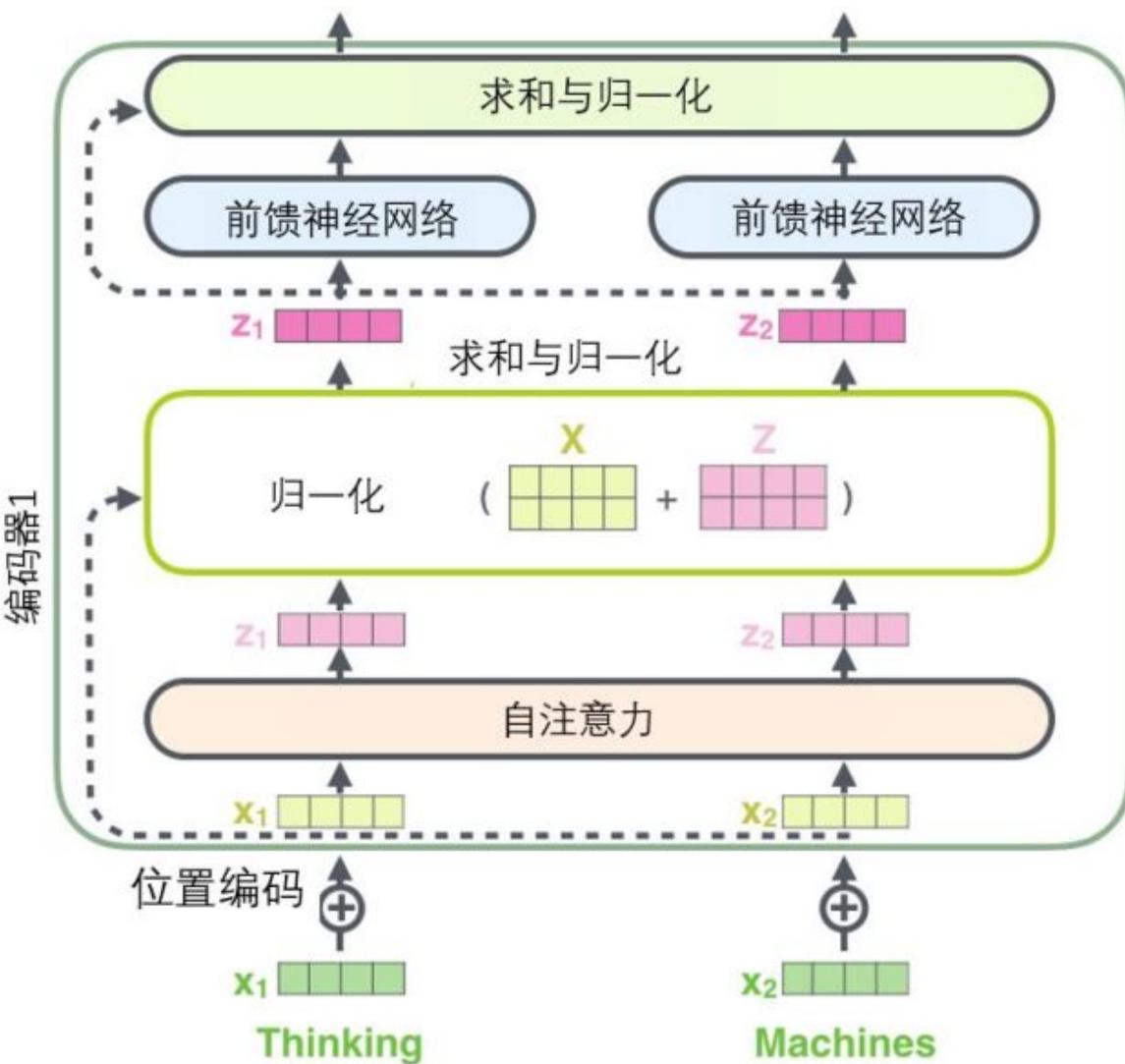
残差模块

在继续进行下去之前，我们需要提到一个编码器架构中的细节：在每个编码器中的每个子层（自注意力、前馈网络）的周围都有一个残差连接，并且都跟随着一个“层-归一化”步骤。

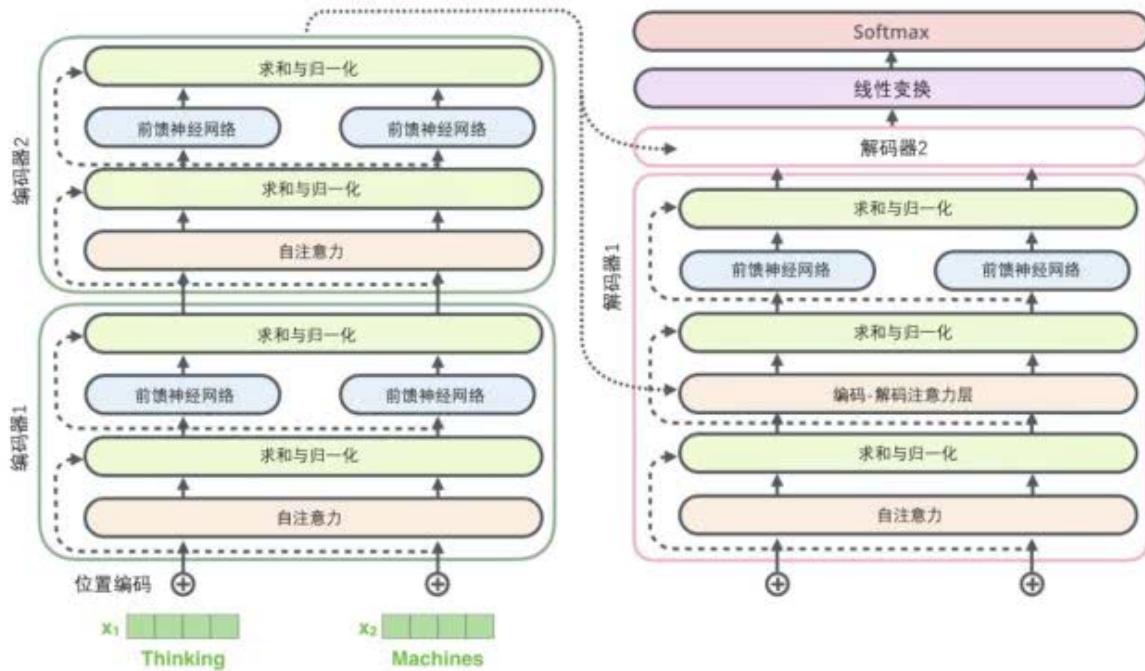
层-归一化步骤：<https://arxiv.org/abs/1607.06450>



如果我们去可视化这些向量以及这个和自注意力相关联的层-归一化操作，那么看起来就像下面这张图描述一样：



解码器的子层也是这样样的。如果我们想象一个2层编码-解码结构的transformer，它看起来会像下面这张图一样：

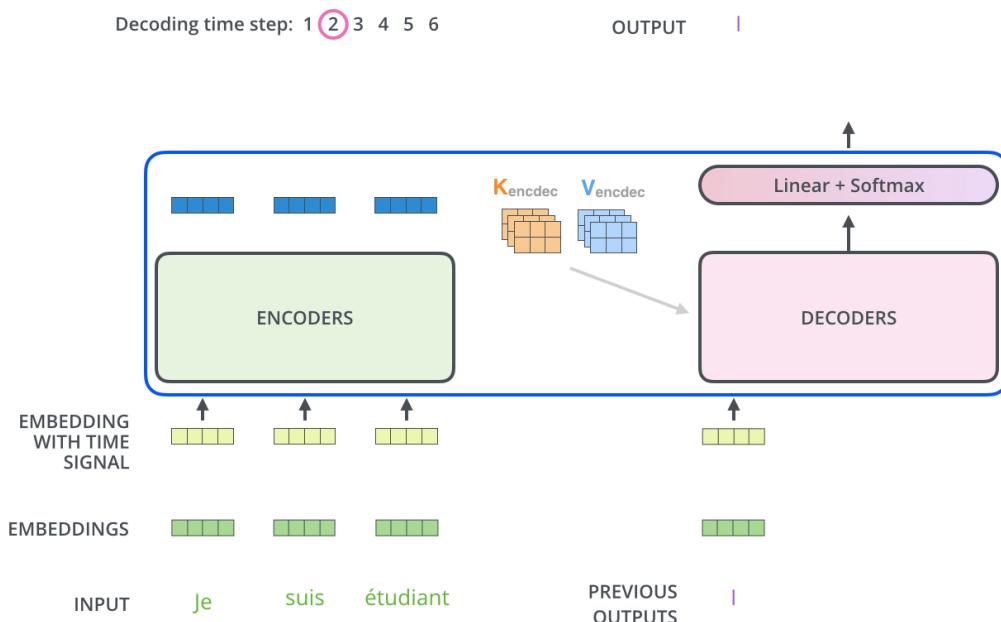


解码组件

编码器通过处理输入序列开启工作。顶端编码器的输出之后会变转化为一个包含向量K（键向量）和V（值向量）的注意力向量集。这些向量将被每个解码器用于自身的“编码-解码注意力层”，而这些层可以帮助解码器关注输入序列哪些位置合适。

在完成编码阶段后，则开始解码阶段。解码阶段的每个步骤都会输出一个输出序列（在这个例子里，是英语翻译的句子）的元素

接下来的步骤重复了这个过程，直到到达一个特殊的终止符号，它表示transformer的解码器已经完成了它的输出。每个步骤的输出在下一个时间步被提供给底端解码器，并且就像编码器之前做的那样，这些解码器会输出它们的解码结果。另外，就像我们对编码器的输入所做的那样，我们会嵌入并添加位置编码给那些解码器，来表示每个单词的位置。



而那些解码器中的自注意力层表现的模式与编码器不同：**在解码器中，自注意力层只被允许处理输出序列中更靠前的那些位置**。在softmax步骤前，它会把后面的位置给隐去（把它们设为-inf）。

这个“编码-解码注意力层”工作方式基本就像多头自注意力层一样，只不过它是通过在它下面的层来创造查询矩阵，并且从编码器的输出中取得键/值矩阵。

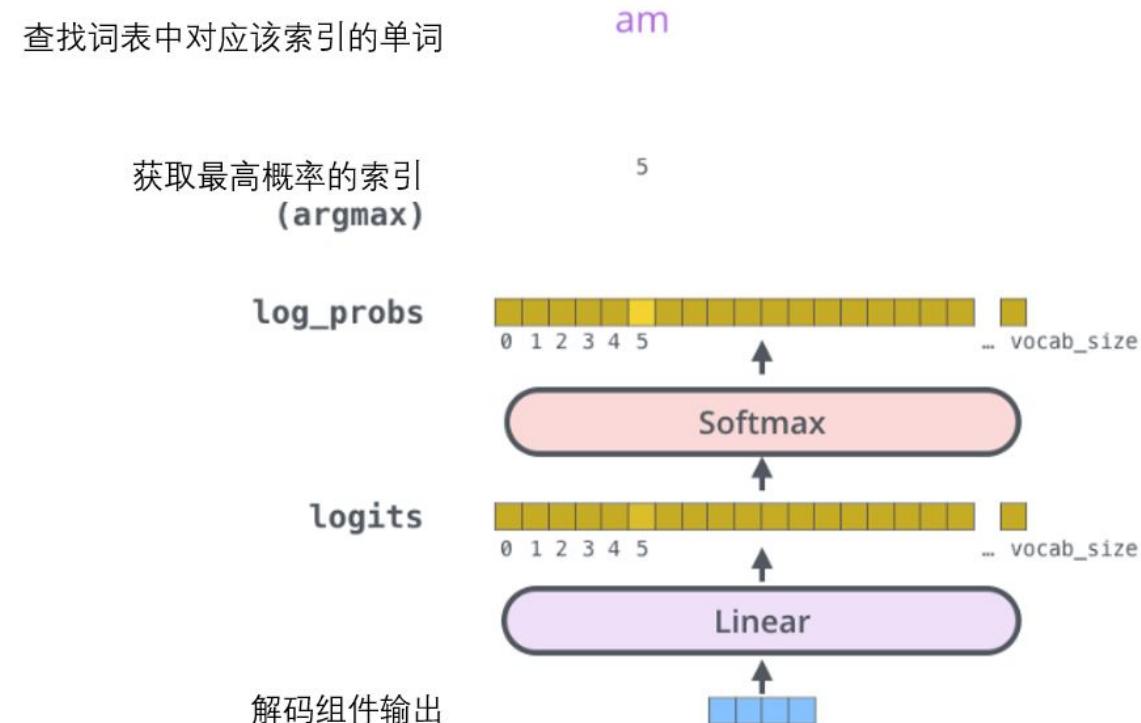
最终的线性变换和Softmax层

解码组件最后会输出一个实数向量。我们如何把浮点数变成一个单词？这便是线性变换层要做的工作，它之后就是Softmax层。

线性变换层是一个简单的全连接神经网络，它可以把解码组件产生的向量投射到一个比它大得多的、被称作对数几率 (logits) 的向量里。

不妨假设我们的模型从训练集中学习一万个不同的英语单词（我们模型的“输出词表”）。因此对数几率向量为一万个单元格长度的向量——每个单元格对应某一个单词的分数。

接下来的Softmax 层便会把那些分数变成概率（都为正数、上限1.0）。概率最高的单元格被选中，并且它对应的单词被作为这个时间步的输出。

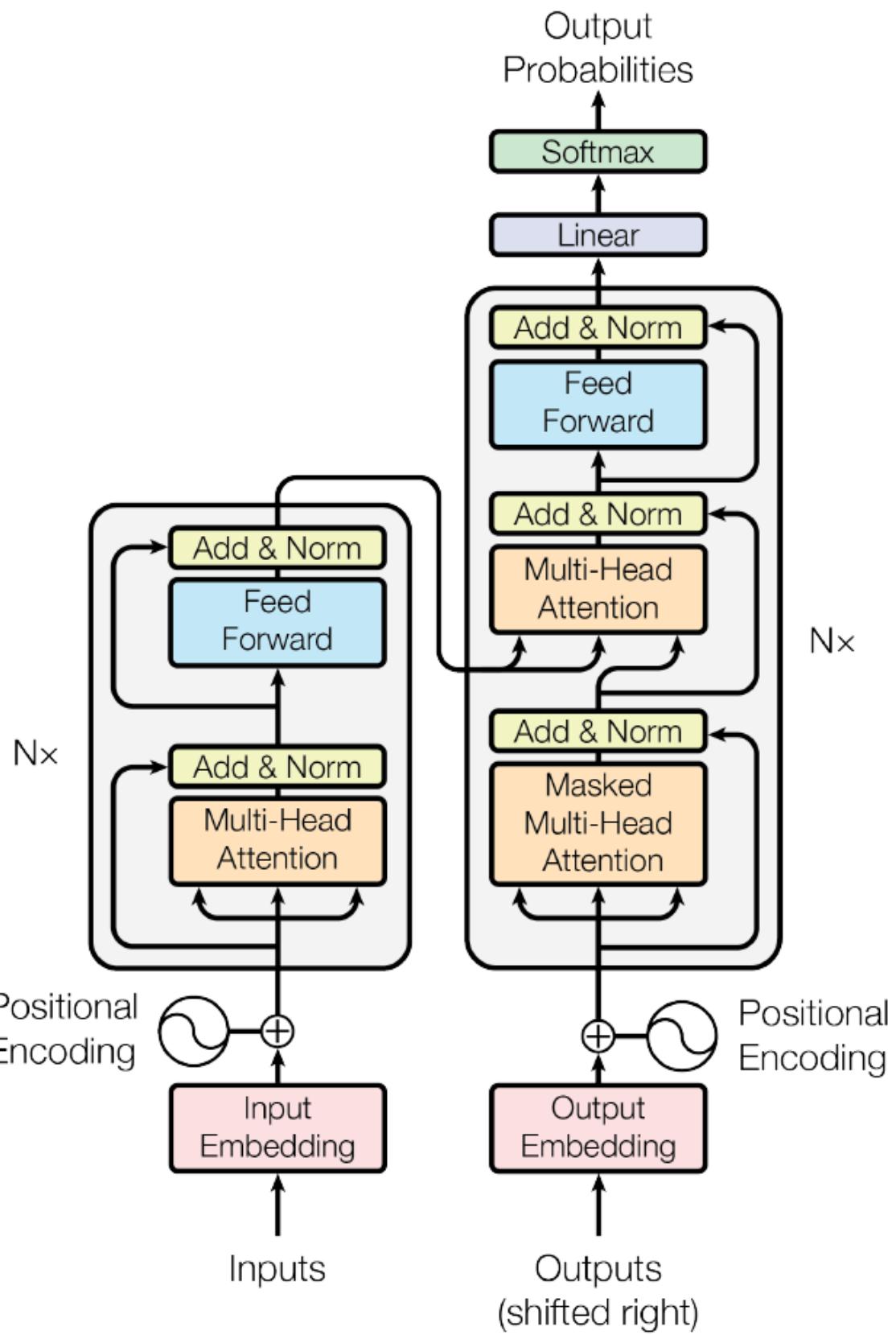


这张图片从底部以解码器组件产生的输出向量开始。之后它会转化出一个输出单词。

论文正文

整体架构

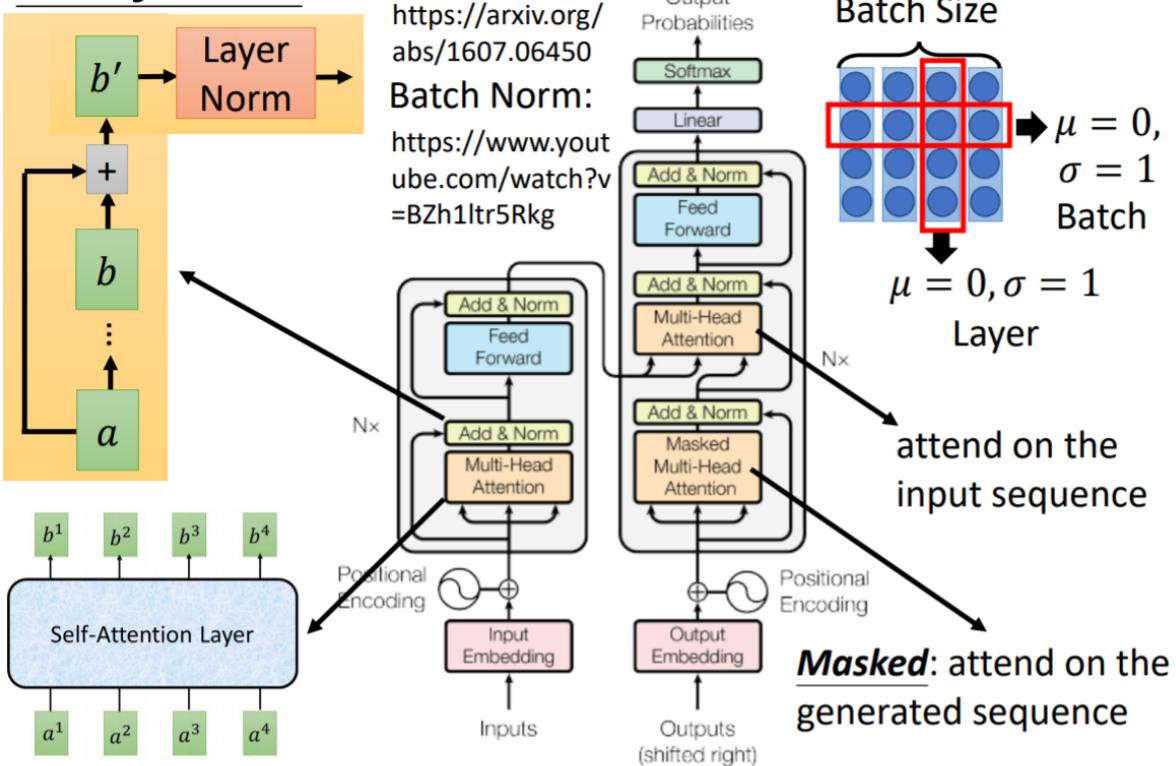
下图是论文的详细结构模型



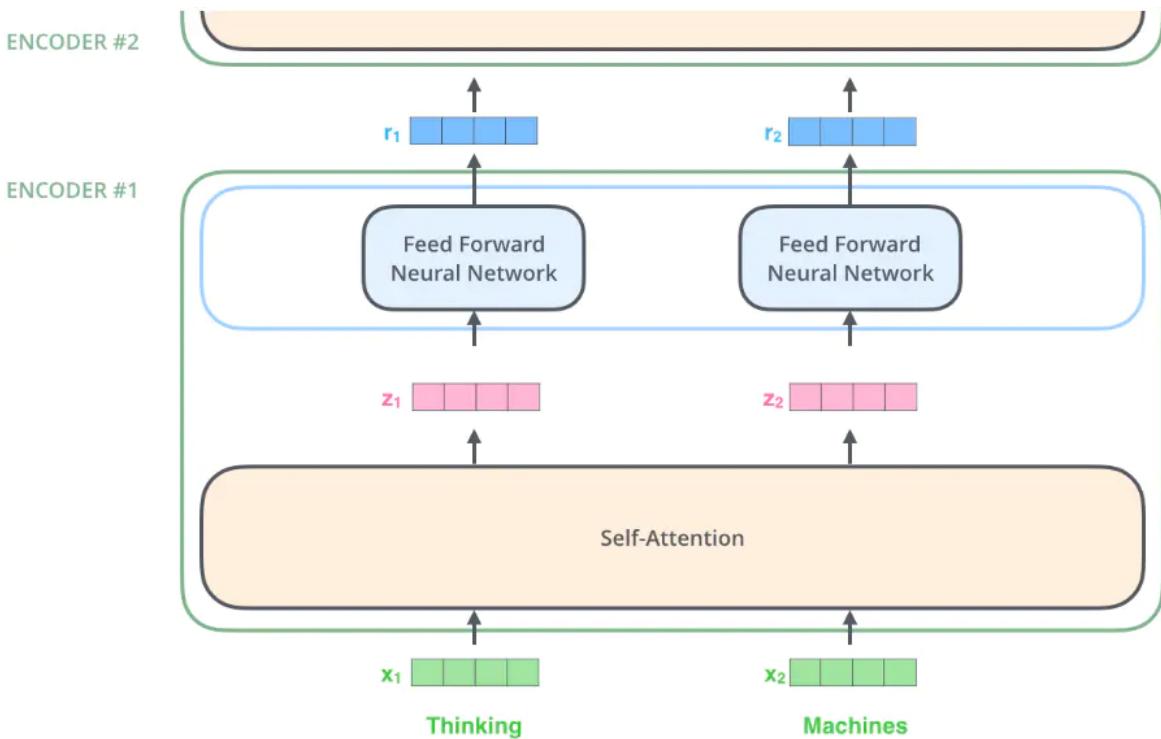
左边是编码器，右边是解码器。

Encoder

Transformer



编码器的结构如上图。



1. Input 经过 embedding 后，要做 positional encodings
2. 然后是 Multi-head attention
3. 再经过 position-wise Feed Forward
4. 每个子层之间有残差连接

Encoder由 $N=6$ 个相同的layer组成，layer指的就是上图左侧的单元，最左边有个“ N_x ”，这里是 $x6$ 个。每个Layer由两个sub-layer组成，分别是多头注意力 (multi-head self-attention mechanism) 和fully connected feed-forward network。其中每个sub-layer都加入了residual connection和normalisation，因此可以将sub-layer的输出表示为：

$$\text{sub_layer_output} = \text{LayerNorm}(x + (\text{SubLayer}(x)))$$

Decoder

Decoder的输入是前一个输入的输出。masked是做attention的时候只计算已经产生过的东西。

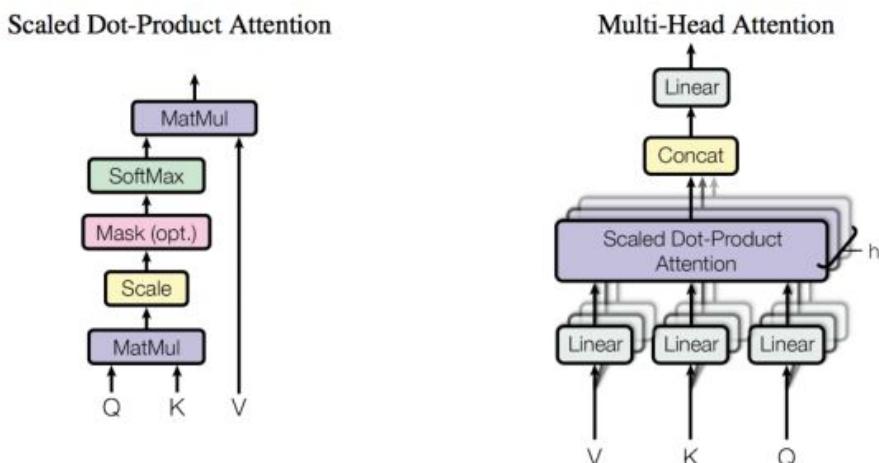
1. 如上图所示，也有 positional encodings, Multi-head attention 和 FFN，子层之间也要做残差连接，
2. 但比 encoder 多了一个 Masked Multi-head attention，
3. 最后要经过 Linear 和 softmax 输出概率。

Decoder和Encoder的结构差不多，但是多了一个attention的sub-layer，这里先明确一下decoder的输入输出和解码过程：

- 输出：对应i位置的输出词的概率分布
- 输入：encoder的输出 & 对应i-1位置decoder的输出。所以中间的attention不是self-attention，它的K, V来自encoder, Q来自上一位置decoder的输出
- 解码：这里要特别注意一下，编码可以并行计算，一次性全部encoding出来，但解码不是一次把所有序列解出来的，而是像rnn一样一个一个解出来的，因为要用上一个位置的输入当作attention的query

新加的attention多加了一个mask，因为训练时的output都是ground truth，这样可以确保预测第i个位置时不会接触到未来的信息。

加了mask的attention原理如图（另附multi-head attention）：



Positional Encoding

原来的paper的位置编码是人设的。

除了主要的Encoder和Decoder，还有数据预处理的部分。Transformer抛弃了RNN，而RNN最大的优点就是在时间序列上对数据的抽象，所以文章中作者提出两种Positional Encoding的方法，将encoding后的数据与embedding数据求和，加入了相对位置信息。

这里作者提到了两种方法：

1. 用不同频率的sine和cosine函数直接计算
2. 学习出一份positional embedding ([参考文献](#))

经过实验发现两者的结果一样，所以最后选择了第一种方法，公式如下：

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

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

作者提到，方法1的好处有两点：

1. 任意位置的 PE_{pos+k} 都可以被 PE_{pos} 的线性函数表示，三角函数特性复习下：

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

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

2. 如果是学习到的positional embedding，（个人认为，没看论文）会像词向量一样受限于词典大小。也就是只能学习到“位置2对应的向量是(1,1,1,2)”这样的表示。所以用三角公式明显不受序列长度的限制，也就是可以对比所遇到序列的更长的序列进行表示。

优点

作者主要讲了以下三点：

| Layer Type | Complexity per Layer | Sequential Operations | Maximum Path Length |
|-----------------------------|--------------------------|-----------------------|---------------------|
| Self-Attention | $O(n^2 \cdot d)$ | $O(1)$ | $O(1)$ |
| Recurrent | $O(n \cdot d^2)$ | $O(n)$ | $O(n)$ |
| Convolutional | $O(k \cdot n \cdot d^2)$ | $O(1)$ | $O(\log_k(n))$ |
| Self-Attention (restricted) | $O(r \cdot n \cdot d)$ | $O(1)$ | $O(n/r)$ |

1. Total computational complexity per layer (每层计算复杂度)
2. Amount of computation that can be parallelized, as measured by the minimum number of sequential operations required

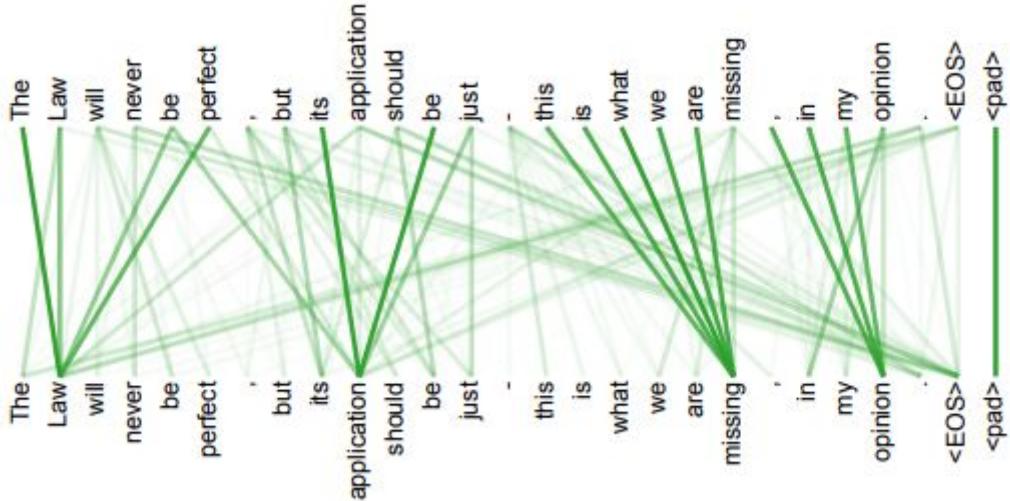
作者用最小的序列化运算来测量可以被并行化的计算。也就是说对于某个序列

x_1, x_2, \dots, x_n , self-attention可以直接计算 x_i, x_j 的点乘结果，而rnn就必须按照顺序从 x_1 计算到 x_n

3. Path length between long-range dependencies in the network

这里Path length指的是要计算一个序列长度为n的信息要经过的路径长度。cnn需要增加卷积层数来扩大视野，rnn需要从1到n逐个进行计算，而self-attention只需要一步矩阵计算就可以。所以也可以看出，self-attention可以比rnn更好地解决长时依赖问题。当然如果计算量太大，比如序列长度n>序列维度d这种情况，也可以用窗口限制self-attention的计算数量

4. 另外，从作者在附录中给出的例子可以看出，self-attention模型更可解释，attention结果的分布表明了该模型学习到了一些语法和语义信息



每两个词都会进行attention，线条越粗表示指向的概率越大。位置不一样的词，指向的位置不一样。

缺点

缺点在原文中没有提到，是后来在Universal Transformers中指出的，在这里加一下吧，主要是两点：

1. 实践上：有些rnn轻易可以解决的问题transformer没做到，比如复制string，或者推理时碰到的sequence长度比训练时更长（因为碰到了没见过的position embedding）
2. 理论上：transformers非computationally universal ([图灵完备](#))，（我认为）因为无法实现“while”循环

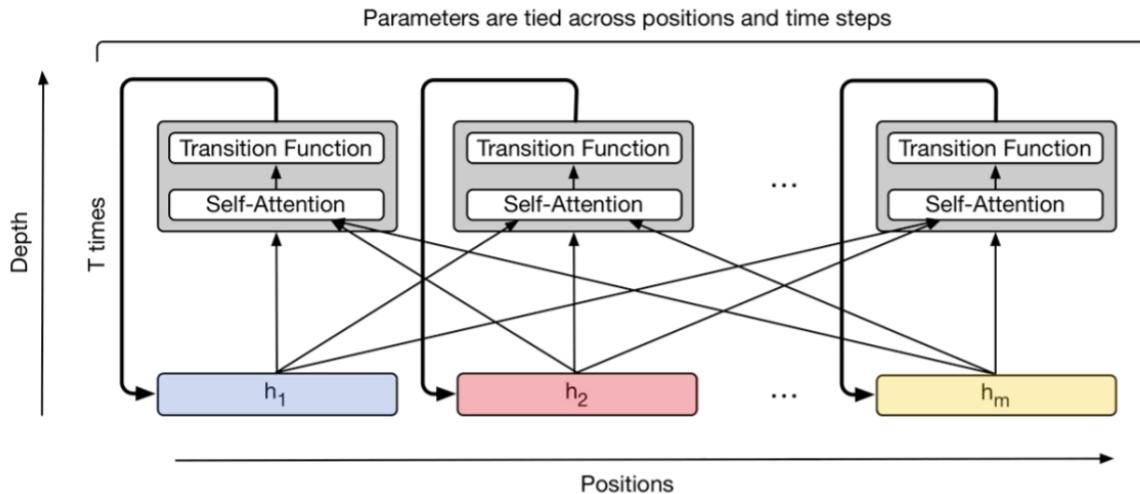
Transformer是第一个用纯attention搭建的模型，不仅计算速度更快，在翻译任务上也获得了更好的结果。Google现在的翻译应该是在此基础上做的，但是请教了一两个朋友，得到的答案是主要看数据量，数据量大可能用transformer好一些，小的话还是继续用rnn-based model

应用：

seq2seq

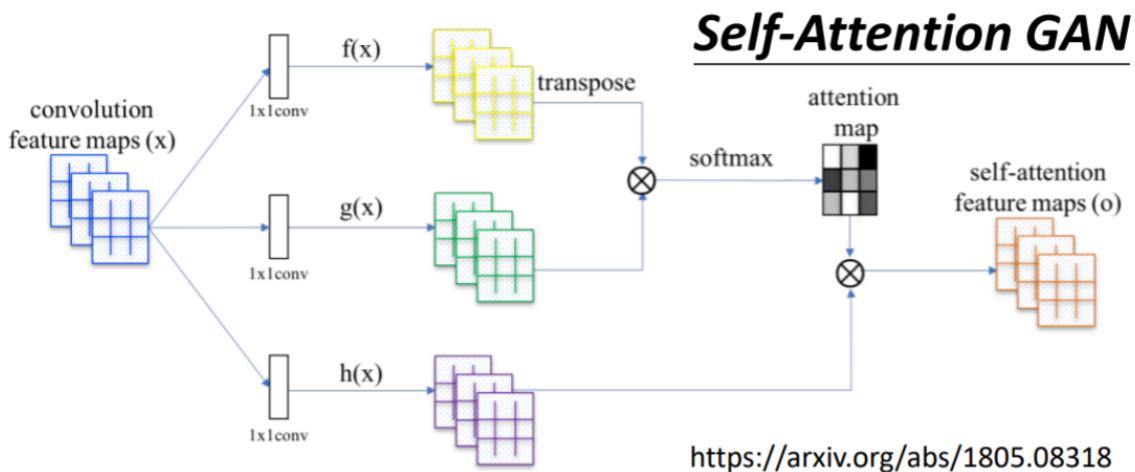
summarizer

Universal Transformer



<https://ai.googleblog.com/2018/08/moving-beyond-translation-with.html>

每一层都是transformer，在深度上都是rnn



也可以应用在图像上，每一个像素都去匹配其他像素

