这么多年，终于有人讲清楚 Transformer 了!

CSDN 2020-10-19

作者 | Jay Alammar

译者 | 香槟超新星，责编 | 夕颜

出品 | CSDN（ID:CSDNnews）

以下为译文：

注意力机制是一种在现代深度学习模型中无处不在的方法，它有助于提高神经机器翻译应用程序性能的概念。在本文中，我们将介绍Transformer这种模型，它可以通过注意力机制来提高

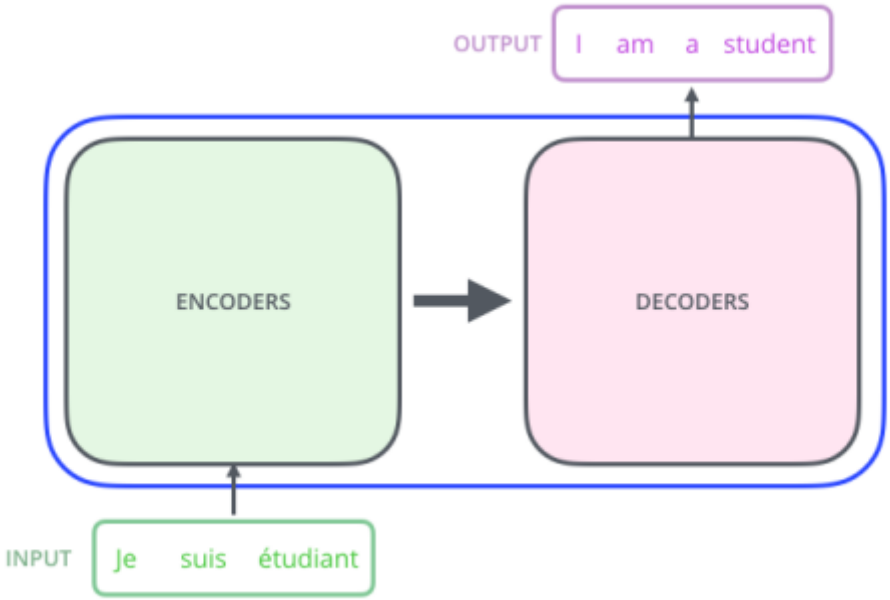
训练模型的速度。在特定任务中，Transformer的表现优于Google神经机器翻译模型。但是，最大的好处来自于Transformer如何适用于并行化。实际上，Google Cloud建议使用Transformer作为参考模型来使用其Cloud TPU产品。因此，我们试试将模型分解开吧，看看它是如何工作的。

Attention is All You Need一文中提出了Transformer。它的TensorFlow实现是Tensor2Tensor包的一部分。哈佛大学的NLP团队创建了一份指南，用PyTorch实现对这篇文章进行注释。在本文中，我们将试着尽可能地简化讲解，并逐一介绍概念，希望能让那些对这方面没有深入知识的人们更容易理解Transformer。

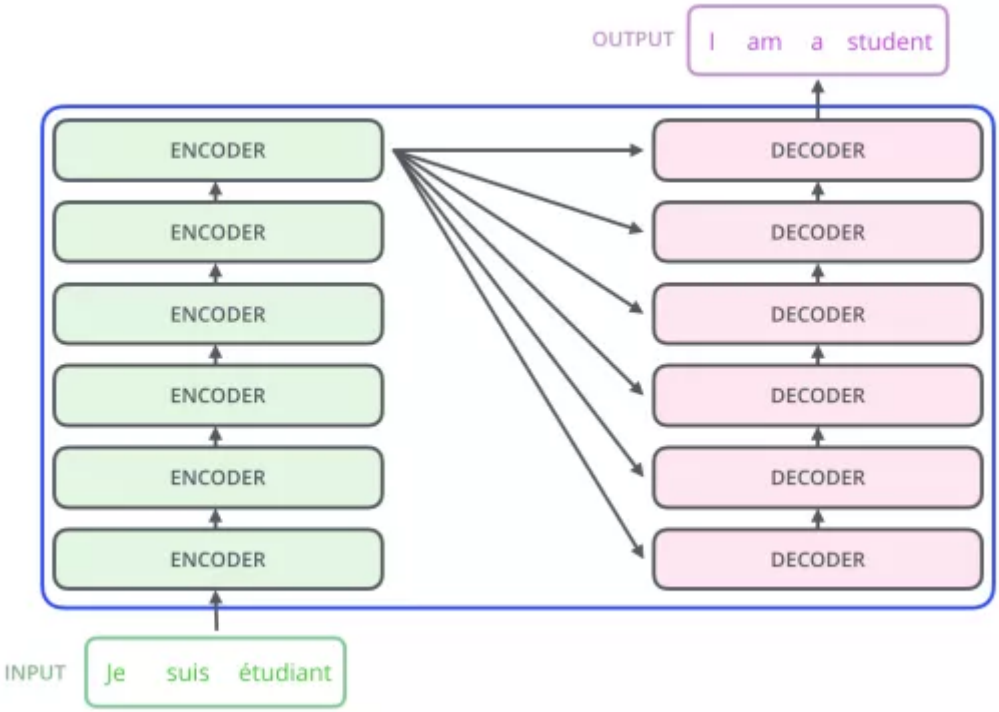
Transformer概览

首先，让我们先将模型视为一个黑盒。在机器翻译应用程序中，这个模型将拿一种语言中的一个句子，然后以另一种语言输出其翻译。

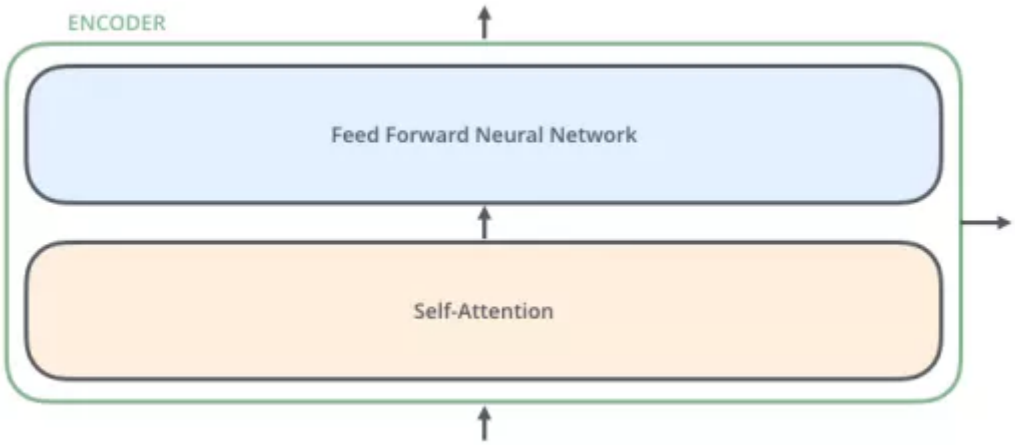
打开擎天柱的引擎盖（Optimus Prime，Transformer与变形金刚是同一个词，故而产生这个梗），我们能看到编码组件，解码组件，以及它们之间的连接。



编码组件是一个编码器组成的堆栈（论文上一个摞一个地堆叠了六个编码器——六这个数字本身没有什么神奇之处，人们肯定可以尝试其他个数）。解码组件是一个由相同数量的解码器组成的堆栈。



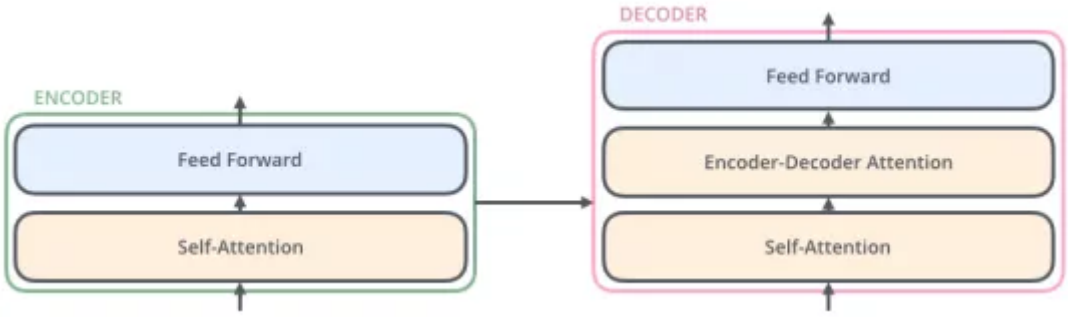
编码器的结构均相同（但它们权重不同）。每一层都可以被分为两个子层：



编码器的输入首先流经自注意力层，这一层可以帮助编码器在对特定单词进行编码时查看输入句子中的其他单词。稍后我们将会进一步关注自注意力层。

自注意层的输出被送到前馈神经网络（feed-forward neural network）。每个位置都独立应用了完全相同的前馈网络。

解码器也具有这两层，但是在它们之间还有一个注意力层，可以帮助解码器专注于输入语句的相关部分上（类似于seq2seq模型中的注意力机制）。



引入张量（Tensor）

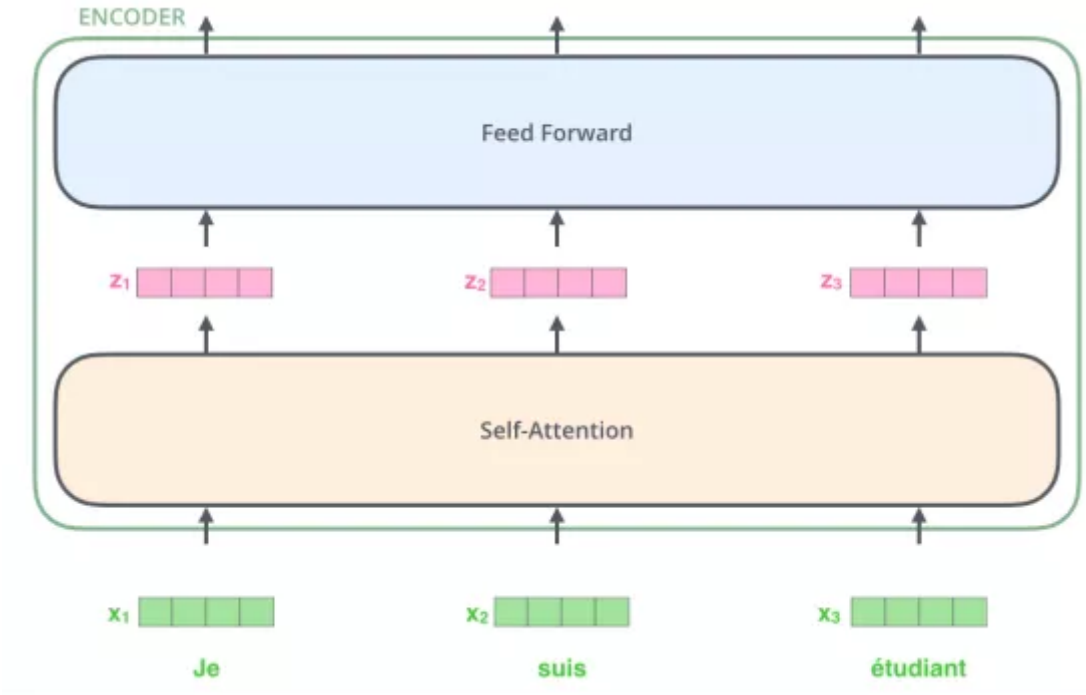
现在，既然我们已经了解了模型的主要组成部分，那就开始研究一下各种向量/张量，以及它们在这些组成部分之间是如何流动的，才能使经过训练的模型把输入转化为输出。

通常，在NLP应用程序中，我们首先使用embedding算法将输入的每个字变成向量。

每个单词都被嵌入到大小为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。对于所有编码器都适用的抽象概念是，它们都会收到一系列向量，每个向量的大小均为512——在最底层的编码器中是单词的嵌入，但在其他编码器中将是直接在下面的编码器的输出。向量列表的大小是一个我们可以设置的超参数，基本上来说，这个参数就是训练数据集中最长句子的长度。

在将我们输入序列中的单词嵌入以后，每个单词都分别流经编码器的两层。

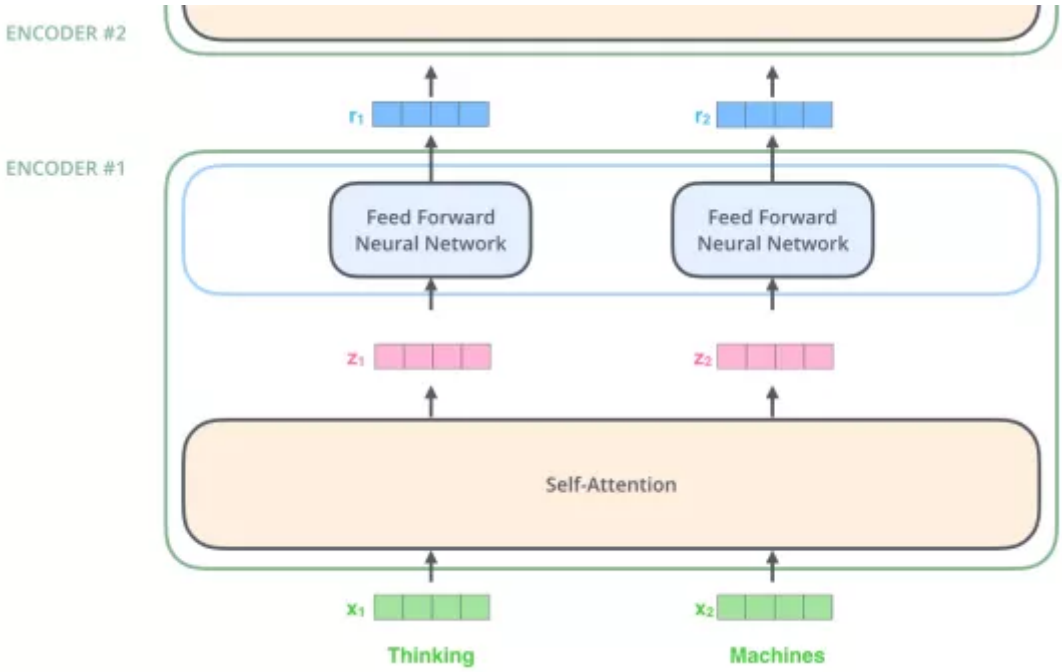


从这里开始，我们就可以看到Transformer的一个关键属性了，那就是每个位置的单词都沿着自己的路径流经编码器。自注意力层中的这些路径之间存在依赖性。但是，前馈层不具有这些依赖性，因此各种路径可以在流过前馈层的同时被并行执行。

接下来，我们将换一个较短的句子作为示例，然后看一下在编码器的每个子层中都发生了些什么。

现在，我们才是在编码！

正如我们已经提到的，编码器接收一个向量列表作为输入。它首先将这些向量传递到自注意力层，然后传递到前馈神经网络，然后将输出向上发送到下一个编码器，以这样的一个流程来处理向量列表。



每个位置的单词都会经过一个自注意力流程。然后，它们中的每个会都通过前馈神经网络——完全相同的网络，每个向量分别独立流过。

自注意力机制概览

不要因为我一直在讲“自注意力”（self-attention）这个词，就误认为这是每个人都应该熟悉的概念。在阅读Attention is All You Need论文之前，我自己从未碰到过这个概念。让我们来提炼总结一下它的工作原理。

比方说，下面的句子是我们要翻译的输入：

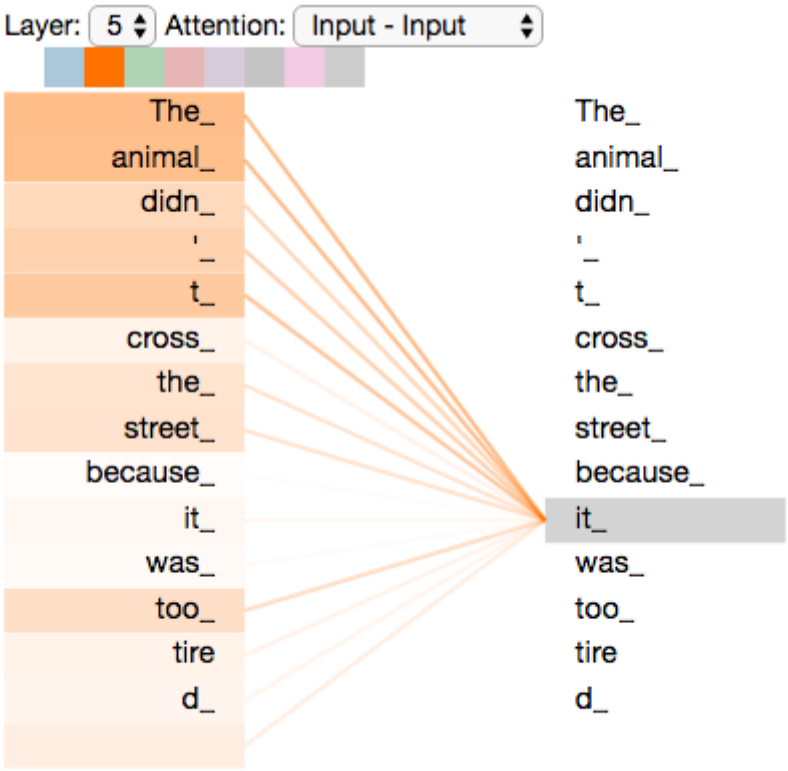
“The animal didn't cross the street because it was too tired.”

这句话中的“it”指的是什么？是指街道还是动物？对人类来说，这是一个简单的问题，但对算法而言却不那么简单。

当模型处理“ it”一词时，自注意力机制使其能够将“it”与“animal”相关联。

在模型处理每个单词（输入序列中的每个位置）时，自注意力使其能够查看输入序列中的其他位置，以寻找思路来更好地对该单词进行编码。

如果你熟悉RNN，请想一下如何通过保持隐状态来使RNN将其已处理的先前单词/向量的表示与当前正在处理的单词/向量进行合并。Transformer使用自注意力机制来将相关词的理解编码到当前词中。



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，注意力机制的一部分集中在“The Animal”上，并将其表示的一部分合并到“it”的编码中。

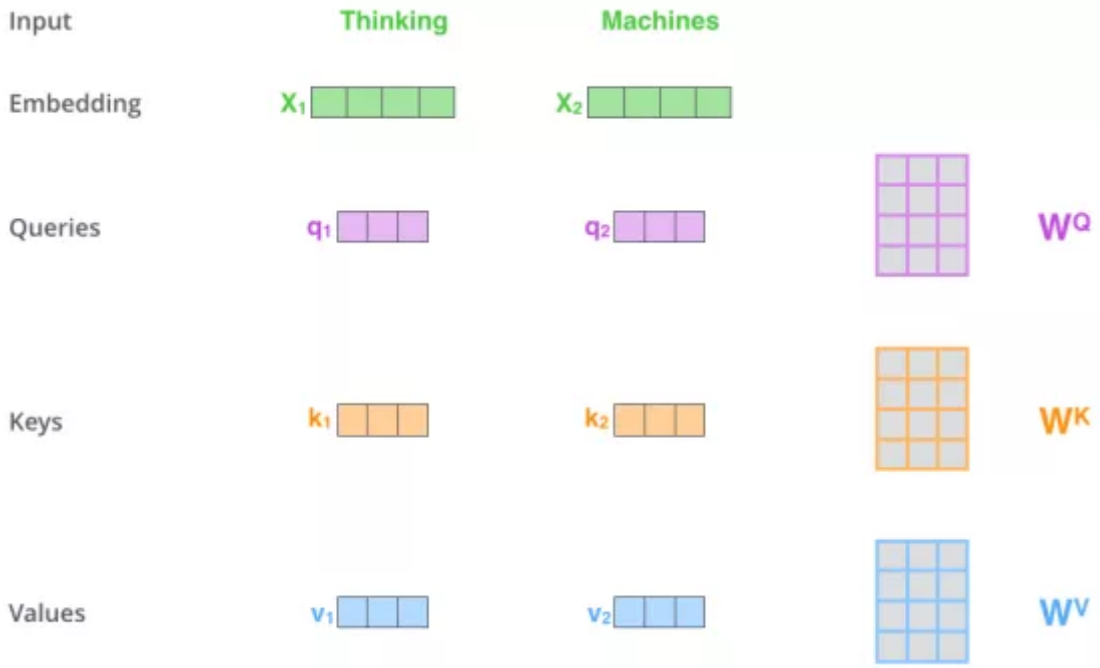
一定要去看一下Tensor2Tensor notebook，你可以在在里面加载Transformer模型，并使用交互式可视化检查一下。

自注意力详解

首先，让我们看一下如何使用向量来计算自注意力，然后着眼于如何使用矩阵来实现。

计算自注意力的第一步是依据每个编码器的输入向量（在这种情况下，是每个单词的embedding）创建三个向量。因此，对于每个单词，我们创建一个Query向量，一个Key向量和一个Value向量。通过将embedding乘以我们在训练过程中训练的三个矩阵来创建这些向量。

请注意，这些新向量的维数小于embedding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（大部分）计算保持一致的结构性选择。



x1乘以WQ权重矩阵可得出q1，即与该单词关联的“Query”向量。我们最终为输入句子中的每个单词创建一个“Query”，一个“Key”和一个“Value”投射。

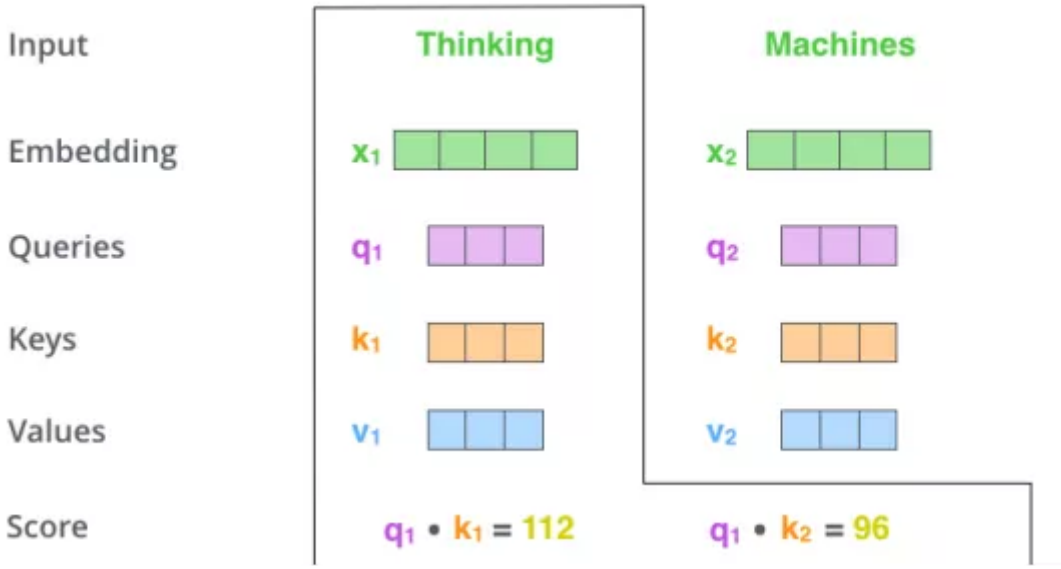
什么是“Query”，“Key”和“Value”向量?

它们是一种抽象，对于注意力的计算和思考方面非常有用。继续阅读下面的注意力计算方式，你几乎就能了解所有这些媒介所起的作用了。

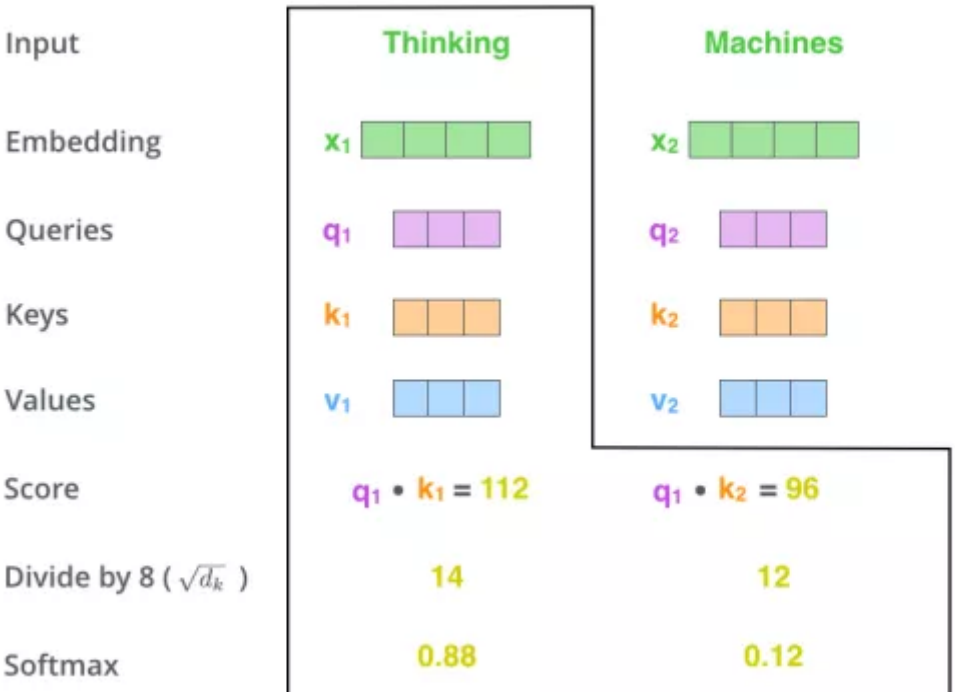
计算自注意力的第二步是计算一个分数（score）。假设我们正在计算这个例子中第一个单词“Thinking”的自注意力。我们需要根据该单词对输入句子中的每个单词打分。这个分数决定了当我们为某个位置的单词编码时，在输入句子的其他部分上的重视程度。

分数是通过将Query向量的点积与我们要评分的各个单词的Key向量相乘得出的。因此，如果我们正在处理位置＃1上的单词的自注意，则第一个分数将是q1和k1的点积。第二个得分将是

q1和k2的点积。



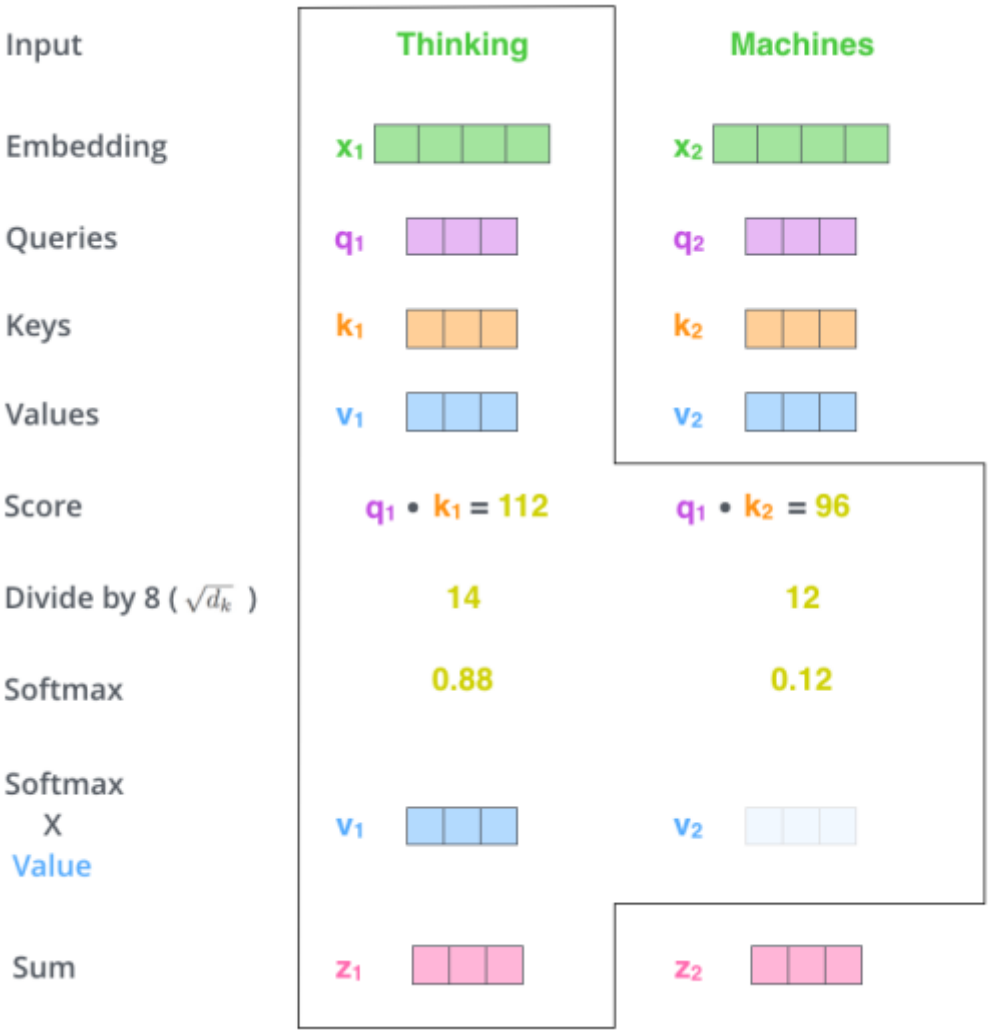
第三和第四步是将分数除以8（论文中使用的Key向量维数的平方根，即64。这将引入更稳定的渐变。此处也许会存在其他可能的值，但这是默认值），然后将结果通过一个softmax操作传递。Softmax对分数进行归一化，使它们均为正数，并且和为一。



这个softmax分数将会决定在这个位置上的单词会在多大程度上被表达。显然，当前位置单词的softmax得分最高，但有时候，注意一下与当前单词相关的另一个单词也会很有用。

第五步是将每个Value向量乘以softmax分数（对后续求和的准备工作）。这里直觉的反应是保持我们要关注的单词的value完整，并压过那些无关的单词（例如，通过把它们乘以0.001这样的很小的数）。

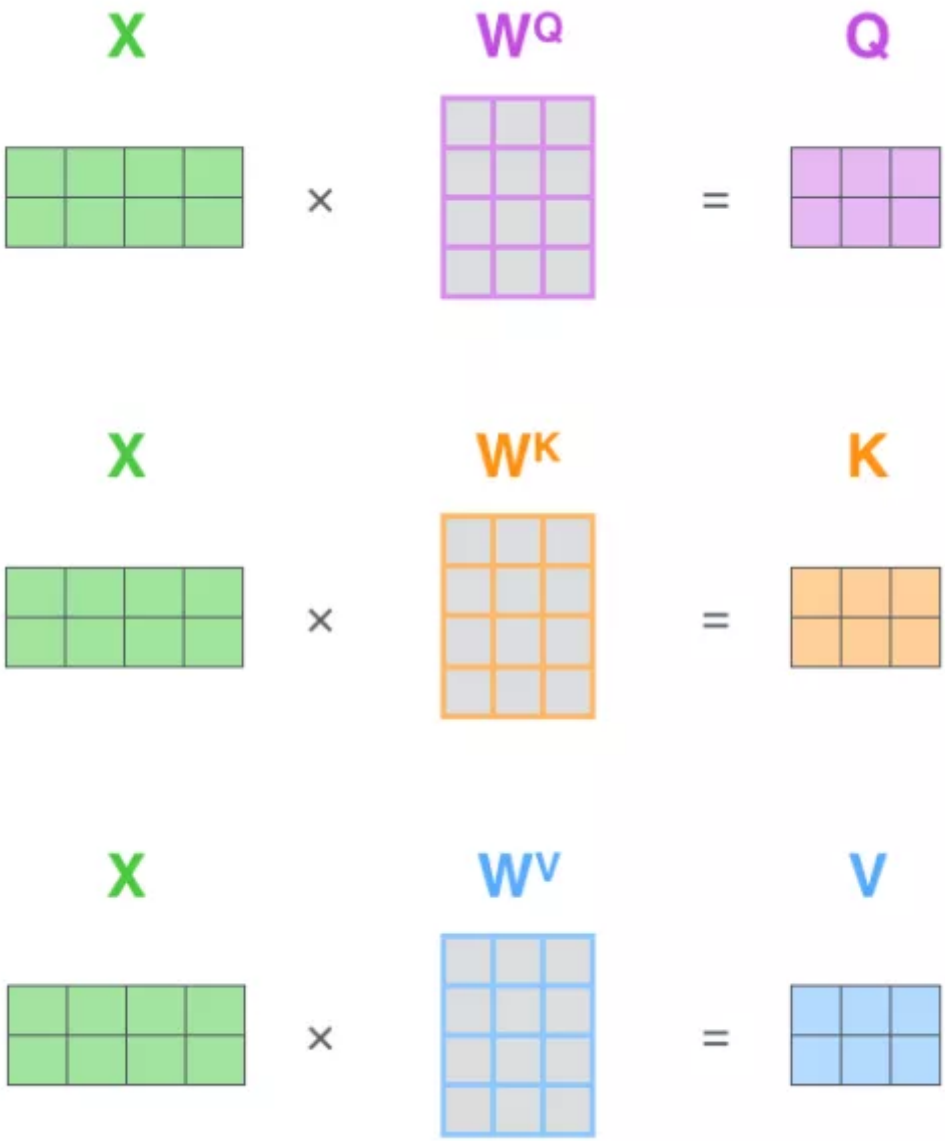
第六步是对加权向量进行求和。这将在此位置（对于第一个单词）产生自注意层的输出。



这样就完成了自注意力的计算。生成的向量是可以被发送到前馈神经网络的。但是，在实际的实现过程中，此计算以矩阵形式进行，以实现更快的处理速度。现在，看完了单词级计算，让我们接着看矩阵计算吧。

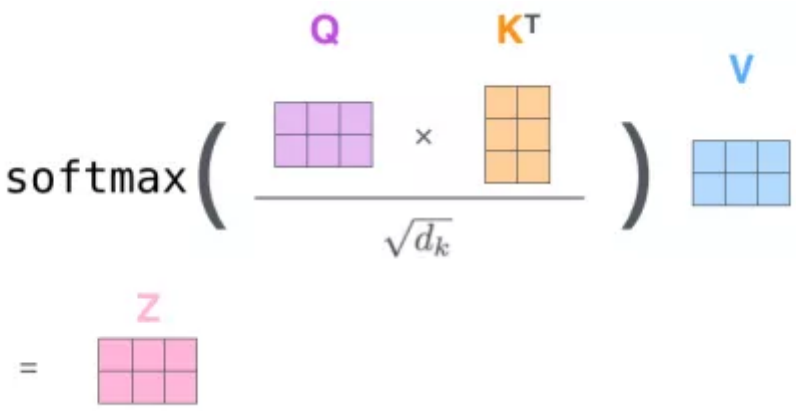
自注意力的矩阵计算

第一步是计算Query，Key和Value矩阵。我们将嵌入内容打包到矩阵X中，然后将其乘以我们训练过的权重矩阵（WQ，WK，WV）。



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

最后，由于我们要处理的是矩阵，因此我们可以通过一个公式将步骤2到6压缩来计算自注意力的输出。



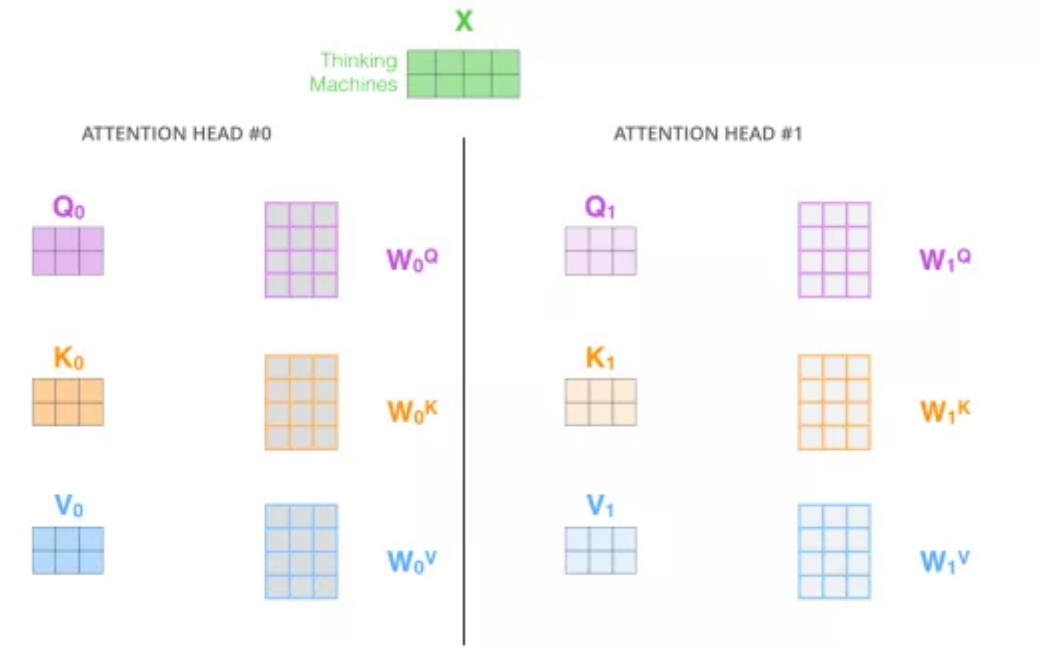
矩阵形式的自注意力计算

长着很多头的野兽

论文通过添加一种名为“多头”注意力的机制，进一步完善了自注意力层。这样可以通过两种方式提高注意力层的性能：

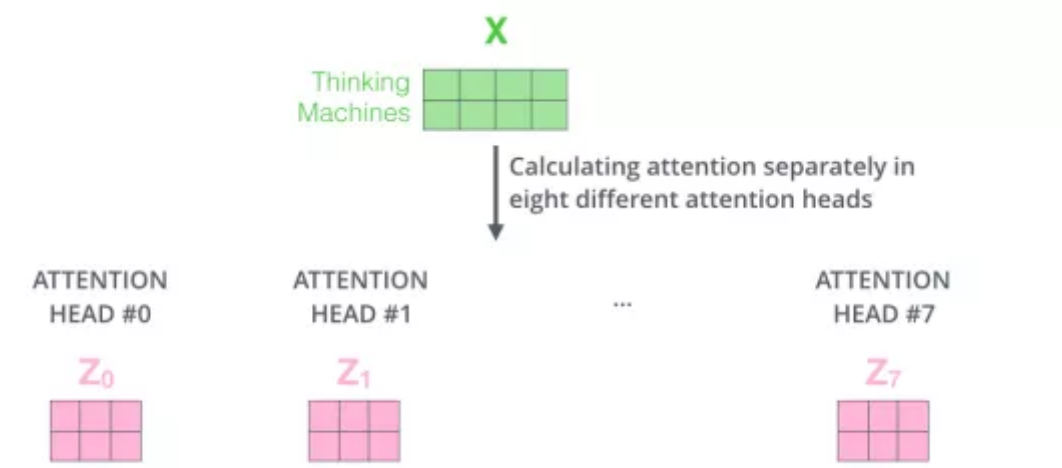
1、它扩展了模型专注于不同位置的能力。是的，在上面的例子中，z1包含所有其他编码的一小部分，但是它可能由实际单词本身主导。如果我们要翻译这样的句子，例如“The animal didn’t cross the street because it was too tired”，那么我们会想知道这里面的“it”指的是什么。

2、它为注意力层提供了多个“表示子空间”（representation subspaces）。正如我们接下来将要看到的，在多头注意力机制下，我们拥有多组Query/Key/Value权重矩阵（Transformer使用八个注意力头，因此每个编码器/解码器最终都能得到八组）。这些集合中的每一个都是随机初始化的。然后，在训练之后，将每个集合用于将输入的embedding（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。



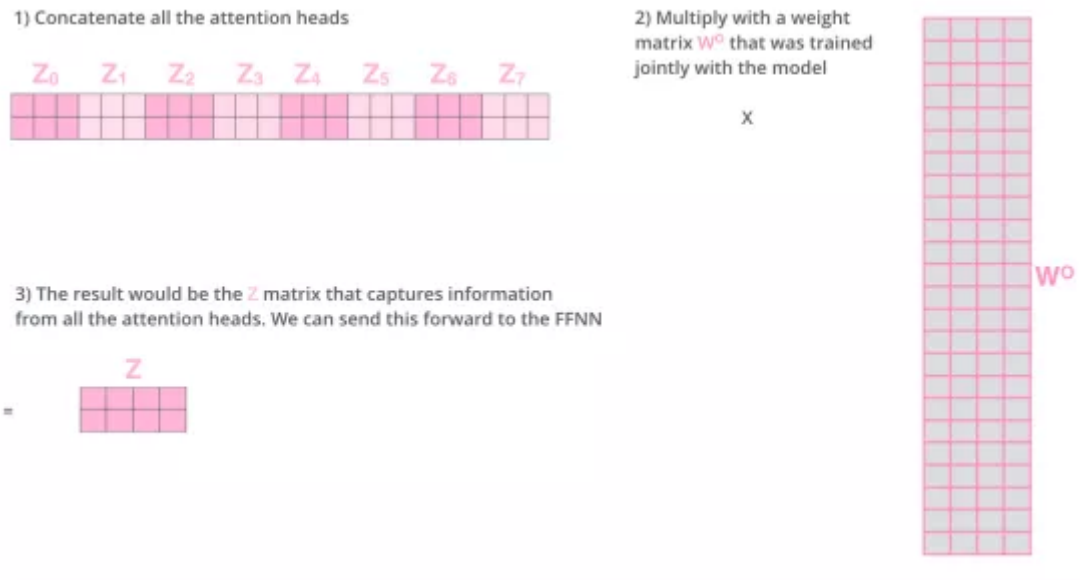
在多头注意力下，我们单独为每个头维护不同的Q / K / V权重矩阵，从而就会得到不同的Q / K / V矩阵。就像之前那样，我们将X乘以WQ / WK / WV矩阵以生成Q / K / V矩阵。

如果我们执行上面概述的自注意力计算，每次使用不同的权重矩阵，计算八次，我们最终将得到八个不同的Z矩阵。

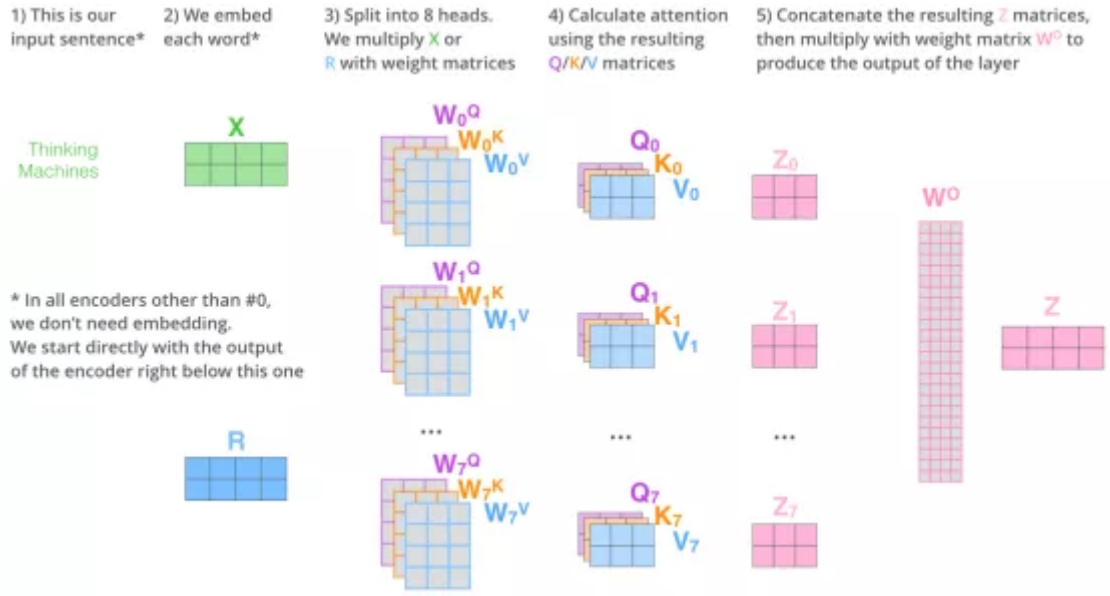


这给我们带来了一些挑战。前馈层所预期的并不是8个矩阵，而是一个单一的矩阵（每个单词一个向量）。因此，我们需要一种方法来将这八个矩阵压缩为单个矩阵。

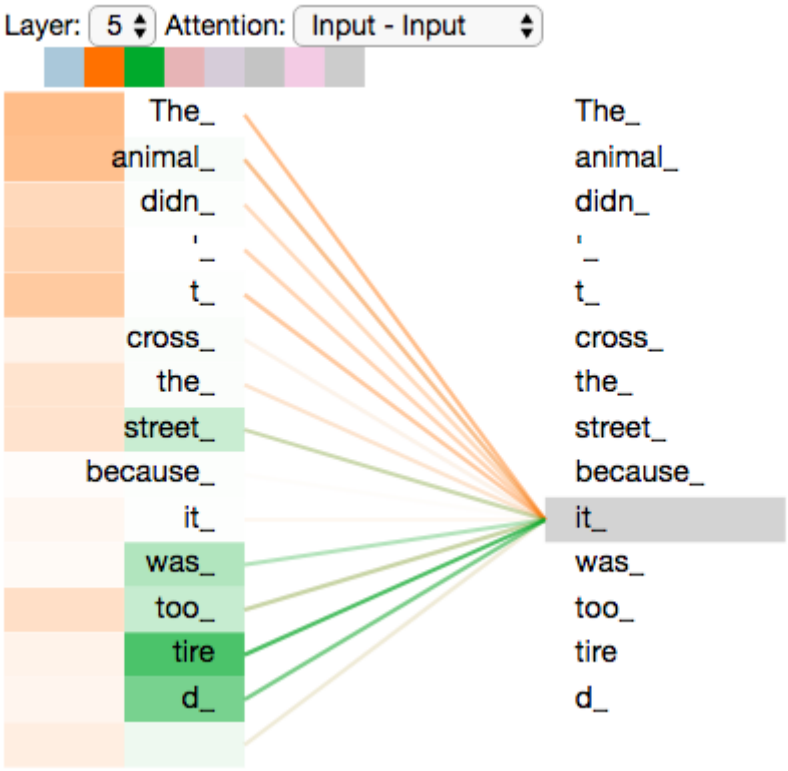
我们该怎么做？我们把这些矩阵合并，然后将它们乘以一个另外的权重矩阵WO。



这差不多就是多头注意力的全部内容。我发现其中的矩阵还是很多的。下面我试试将它们全部放在一个视图中，以便我们可以统一查看。

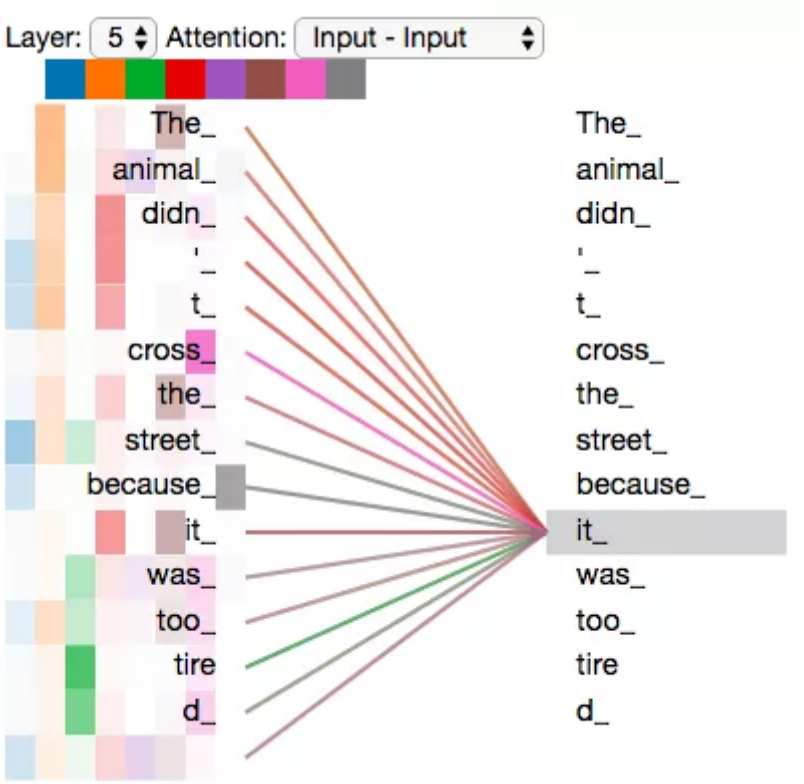


既然我们已经涉及到注意力头的内容，那么让我们重新回顾一下前面的例子，看看在示例句中对“ it”一词进行编码时，不同的注意力头关注的位置分别在哪：



当我们对“it”一词进行编码时，一个注意力头专注于“the animal”一词，而另一个则专注于“tired”一词——从某种意义上来说，模型对单词“it”的表示既依赖于对“animal”的表示又依赖于对“tired”的表示。

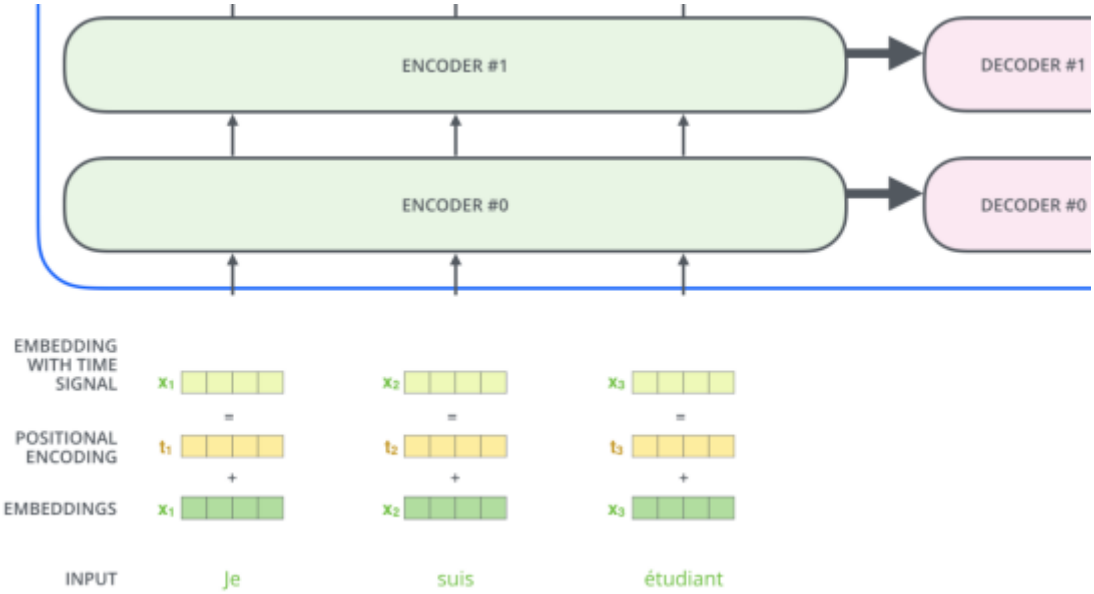
但是，如果我们将所有的注意力头都加到图片中，则可能会比较难以直观解释：



使用位置编码表示序列的顺序

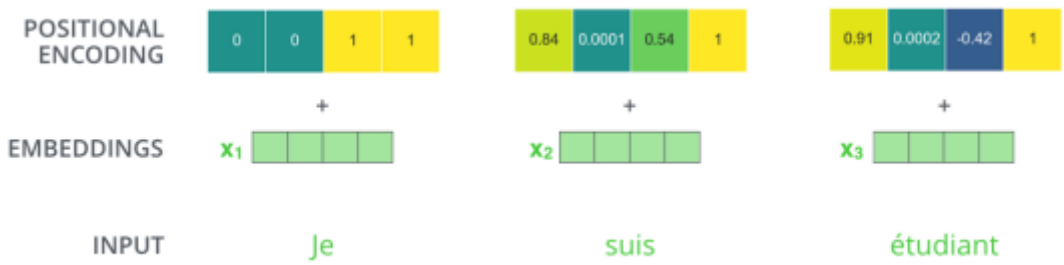
到目前为止，我们对这个模型的描述中尚且缺少一种表示输入序列中单词顺序的方法。

为了解决这个问题，Transformer为每个输入的embedding添加一个向量。这些向量遵循模型学习的特定模式，能够帮助我们确定每个单词的位置，或序列中不同单词之间的距离。在这个地方我们的直觉会是，将这些值添加到embedding中后，一旦将它们投影到Q / K / V向量中，以及对注意力点积，就可以在embedding向量之间提供有意义的距离。



为了使模型感知到单词的顺序，我们添加了位置编码向量，它的值遵循特定的规律。

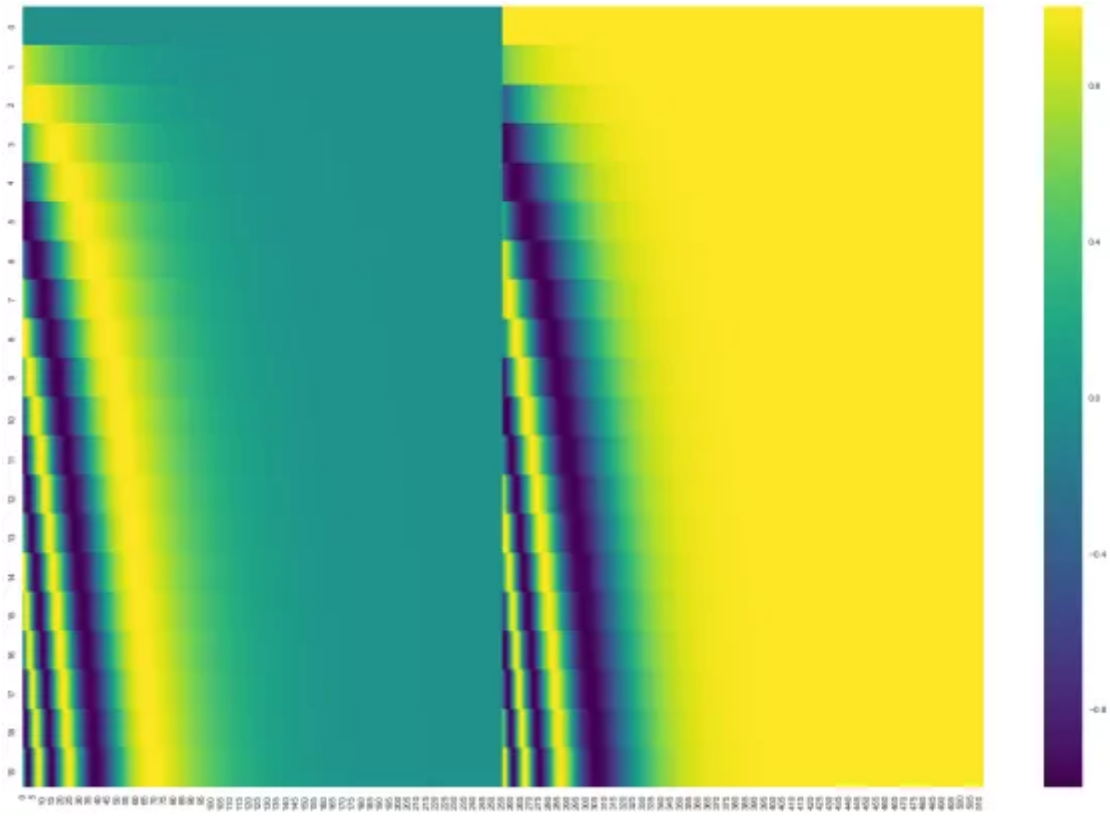
如果我们假设embedding的维数为4，则实际的位置编码则应如下图所示：



一个真实示例，其embedding大小为4的位置编码

这种规律看起来会是什么样的?

在下图中，每行对应一个向量的位置编码。因此，我们要把第一行添加到输入序列中第一个单词的embedding向量。每行包含512个值，每个值都在1到-1之间。我们对它们进行了颜色编码，从而使变化规律更加明显。



一个真实例子的位置编码，embedding大小为512（列），20个单词（行）。你会发现，它看起来像是从中心位置向下分开的。这是因为左半部分的值是由一个函数（使用正弦函数）生成的，而右半部分的值是由另一个函数（使用余弦函数）生成的。然后它们被合并起来形成每个位置的编码向量。

论文中描述了位置编码用到的公式（第3.5节）。你可以在get_timing_signal_1d（）中查看用于生成位置编码的代码。这不是唯一的位置编码方法。但是，它的优势在于能够放大到看不见的序列长度（例如，我们训练后的模型被要求翻译一个句子，而这个句子比我们训练集中的任何句子都长）。

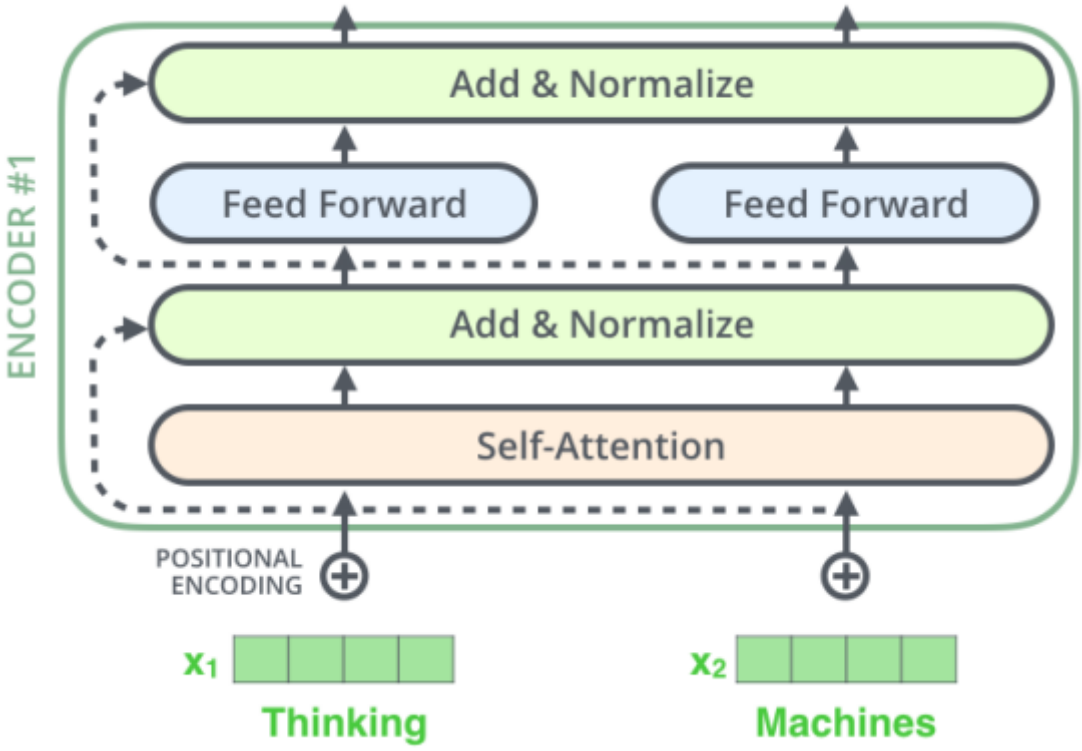
（代码地址：
https://github.com/tensorflow/tensor2tensor/blob/23bd23b9830059fbc349381b70d9429b5c40a139/tensor2tensor/layers/common_attention.py）

2020年7月更新：上面显示的位置编码来自Transformer的Tranformer2Transformer实现。论文中用的方法略有不同，论文中没有直接链接，而是将两个信号交织。下面的图显示了这种方式的样子。这是用来生成它的代码：
https://github.com/jalammar/jalammar.github.io/blob/master/notebookes/transformer/transformer_positional_encoding_graph.ipynb

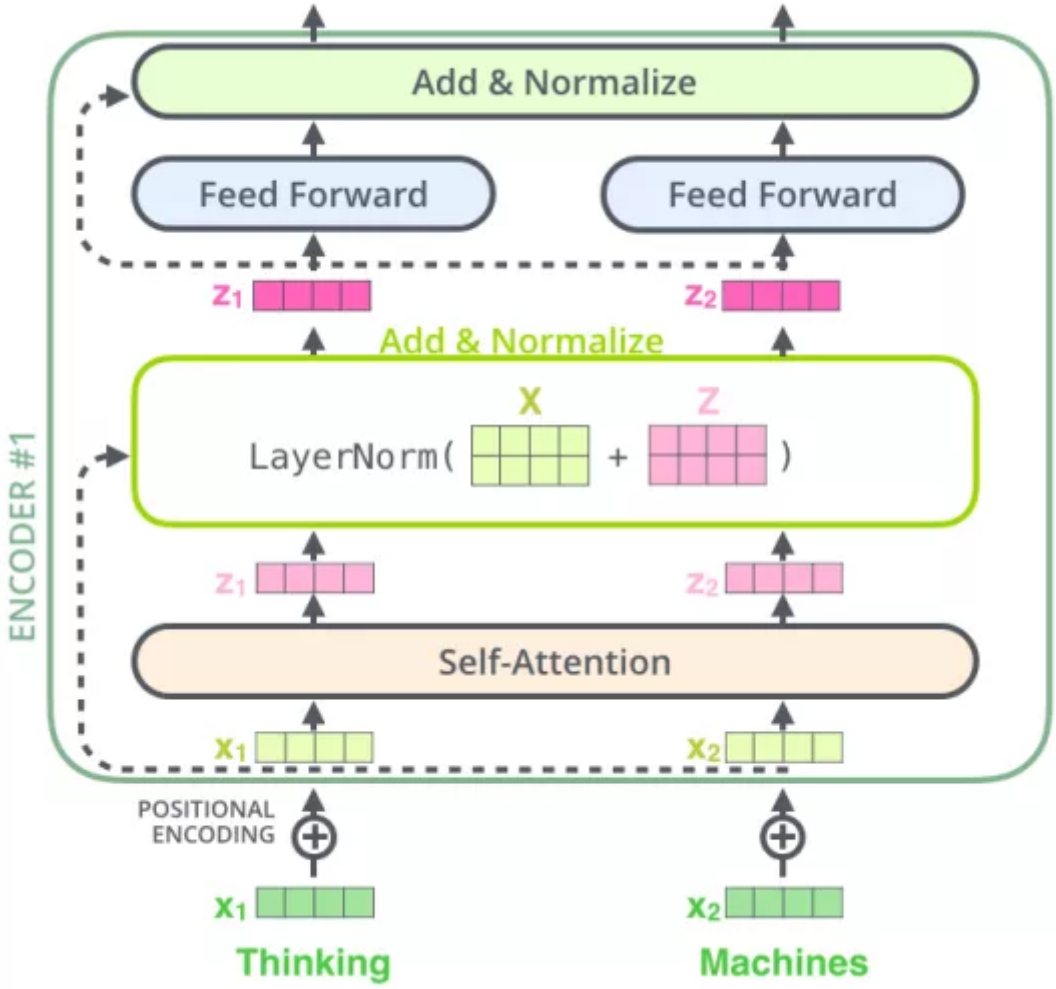
残差

在继续进行讲解之前，我们需要提一下编码器结构中的一个细节，那就是每个编码器中的每个子层（自注意力，ffnn）在其周围都有残差连接，后续再进行层归一化（layer-normalization）步骤。

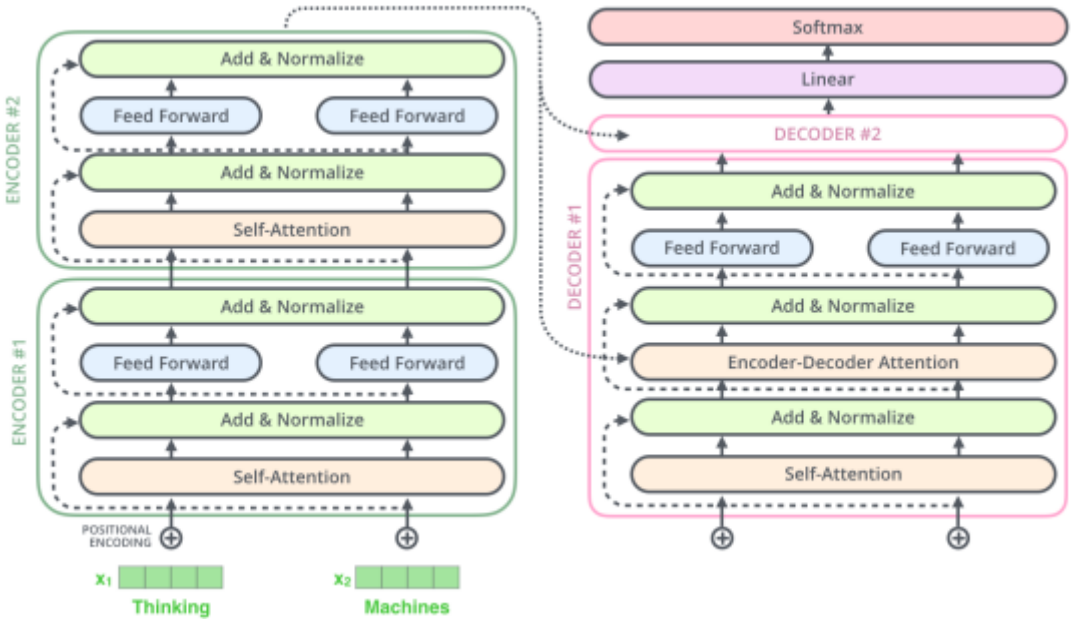
（layer-normalization：https://arxiv.org/abs/1607.06450）



如果我们要对向量和与自注意力相关的层规范操作进行可视化，则看起来应该像这样：



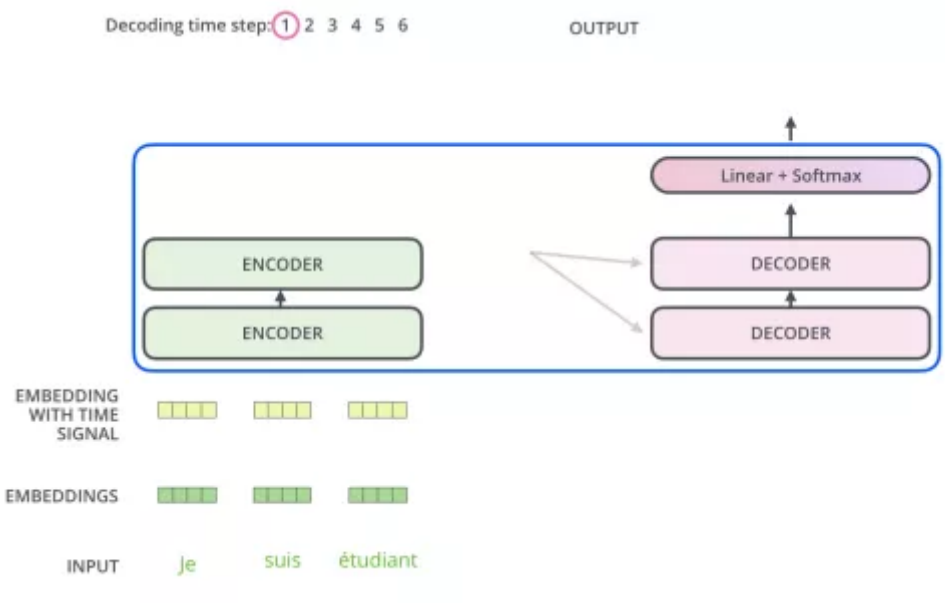
这也适用于解码器的子层。如果我们设想由2个编码器解码器堆栈组成的Transformer，它看起来像这样：



解码器端

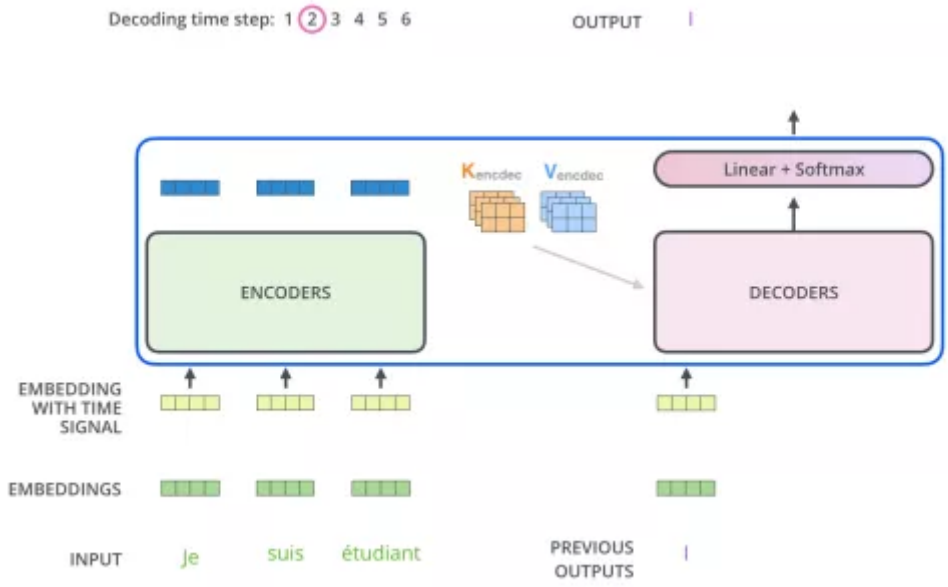
现在，我们已经讲解了编码器方面的大多数概念，同时也基本了解了解码器各组件是如何工作的。然而，接下来让我们看一下它们如何协同。

编码器首先处理输入序列。然后，顶部编码器的输出转换为注意力向量K和V的集合。每个解码器将在其“编码器-解码器注意力”层中使用它们，这有助于解码器将重心放在输入序列中合适的位置：



在完成编码阶段之后，我们开始解码阶段。解码阶段的每个步骤都从输出序列中输出一个元素（在这个例子下，为语句的英文翻译）。

后续步骤一直重复该过程，直到得到一个特殊符号，标志着Transformer解码器已完成其输出。每个步骤的输出都被馈送到下一个步骤的底部解码器，并且解码器会像编码器一样，将其解码结果冒泡。就像我们对编码器输入所做的操作一样，我们给这些解码器输入做嵌入并添加位置编码来表示每个单词的位置。



解码器中的自注意力层与编码器中的略有不同：

在解码器中，自注意力层仅被允许参与到输出序列中的较早位置。这是通过在自注意力计算中的softmax步骤之前屏蔽将来的位置（将它们设置为-inf）来完成的。

“编码器-解码器注意力”层的工作方式与多头自注意力类似，不同之处在于它从下一层创建其Queries矩阵，并从编码器堆栈的输出中获取Keys和Values矩阵。

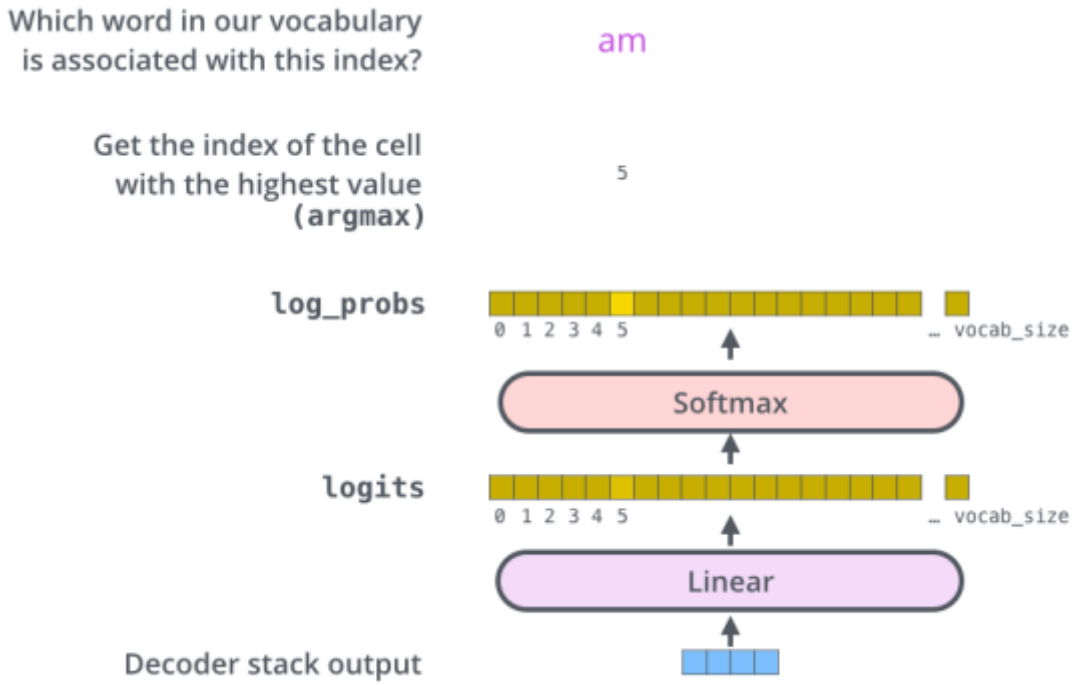
最终的线性层和Softmax层

解码器堆栈输出一组浮点数组成的向量。我们如何把它变成一个词？最后的线性层，以及它之后Softmax层做的就是这项工作。

线性层（he Linear layer）是一个简单的完全连接的神经网络，将解码器堆栈产生的向量投射到一个大得多的对数向量中。

我们假设自己的模型从训练数据集中共学会了10,000个不同的英语单词（我们模型的“输出词汇表”）。这将使对数向量的宽度变为10,000个单元，每个单元对应各个单词的得分。我们将会通过这样的方式来解释模型的输出。

然后，softmax层将会把这些分数转换为概率（全部为正数，各项相加和为1.0）。概率最高的单元被选中，且与该单元相关联的单词将成为该步的输出。



该图从底部开始，生成的向量作为解码器堆栈的输出，后续会被转换为文字输出。

训练过程回顾

现在，我们已经讲解了一个训练完毕的Transformer的前向过程，那么再看一下模型的训练过程也是很有用的。

在训练过程中，未经训练的模型将历经完全相同的前向过程。但是，由于我们正在用已标记的训练数据集对其进行训练，因此我们可以将其输出与正确的输出进行比较。

为了直观地视觉化讲解这一点，我们假设输出词汇表仅包含六个单词（“a”，“am”，“i”，“thanks”，“student”和“<eos>”（“end of sentence”的缩写））。

Output Vocabulary

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

我们模型的输出词汇表是在预处理阶段创建的，那时候还没有开始训练。

一旦定义好了输出词汇表，我们就可以使用一个相同宽度的向量来表示词汇表中的每个单词了。这也被称为one-hot encoding。因此，例如，我们可以使用下面这个向量来表示单词“am”：

Output Vocabulary

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

One-hot encoding of the word “am”

示例：我们输出词汇表的one-hot encoding

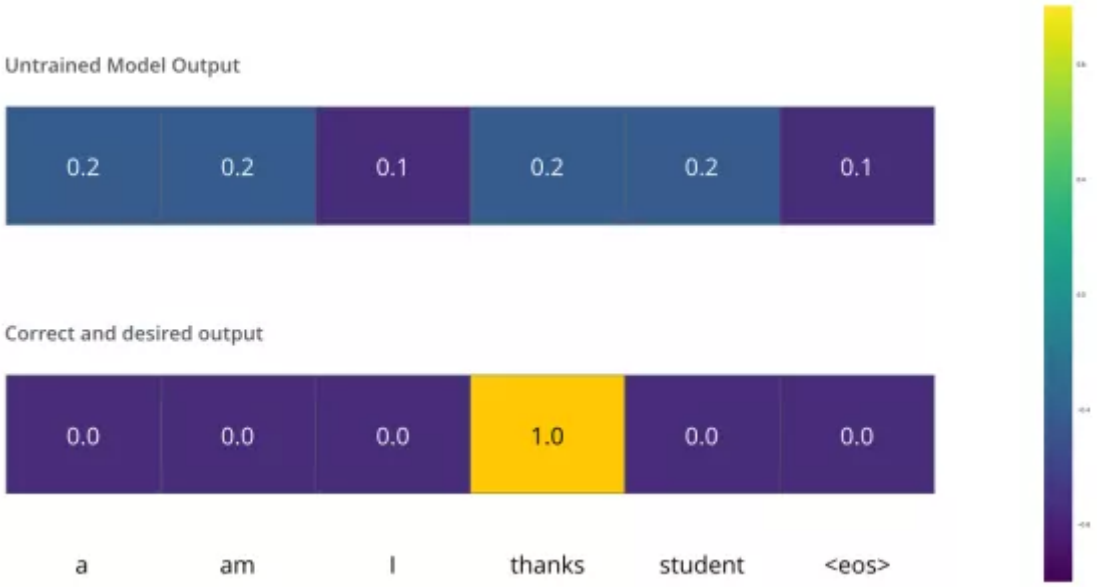
回顾完了之后，接下来让我们讨论一下模型的损失函数（loss function）——我们在训练阶段想要优化的指标，以期最终可以得到一个非常准确模型。

13 CSDN

损失函数

假设我们正在训练我们的模型。假设这是我们训练阶段的第一步，我们用一个简单的例子训练它，使其将“merci”转换为“thanks”。

这意味着，我们希望输出的是一个能表示单词“thanks”的概率分布。但是，由于该模型尚未经过训练，因此目前这还不太可能发生。

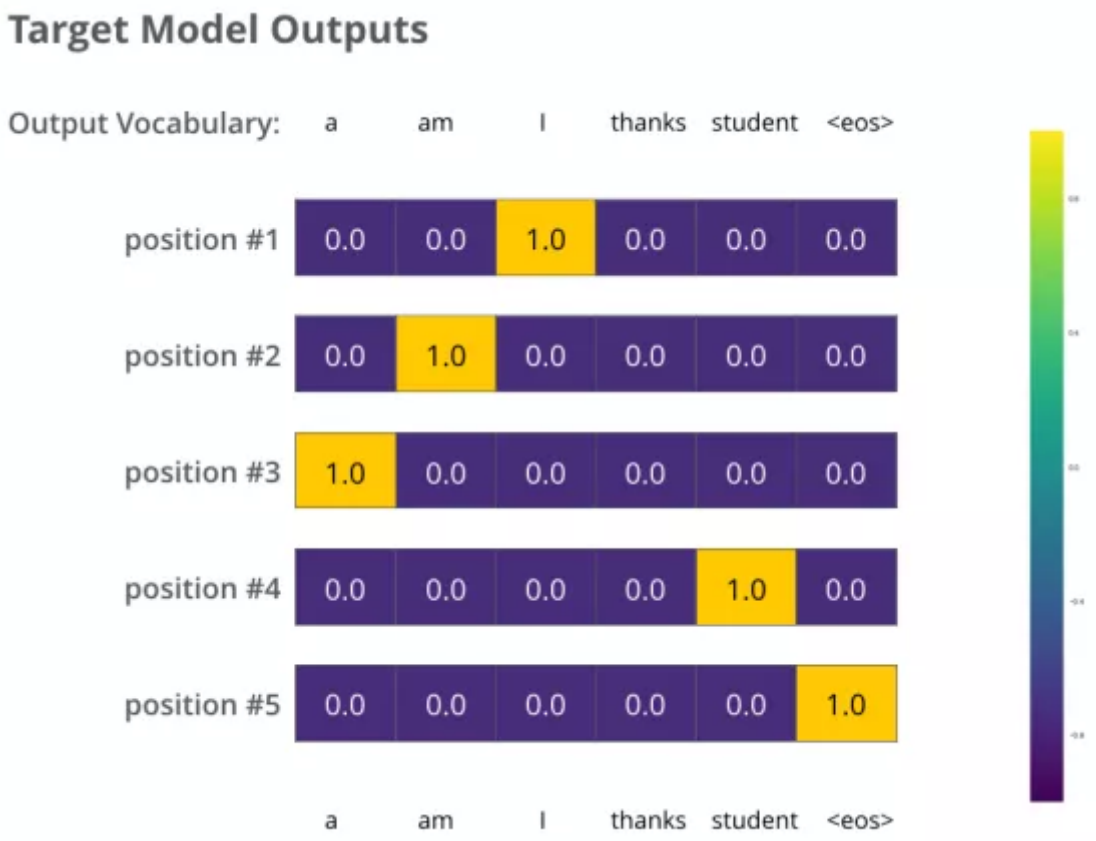


由于模型的参数（权重）在初始化的的时候都是随机分配的，因此（未经训练的）模型为每个单元格/单词生成的概率分布值都是随机的。我们可以将其与实际输出进行比较，然后使用反向传播来调整所有模型的权重，让输出结果更接近我们想要的输出。

如何比较两个概率分布？我们只需用一个减去另一个就可以。欲知更多详细信息，请查阅交叉熵（cross-entropy）和Kullback-Leibler散度（Kullback–Leibler divergence）相关内容。
（https://colah.github.io/posts/2015-09-Visual-Information/
https://www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained）

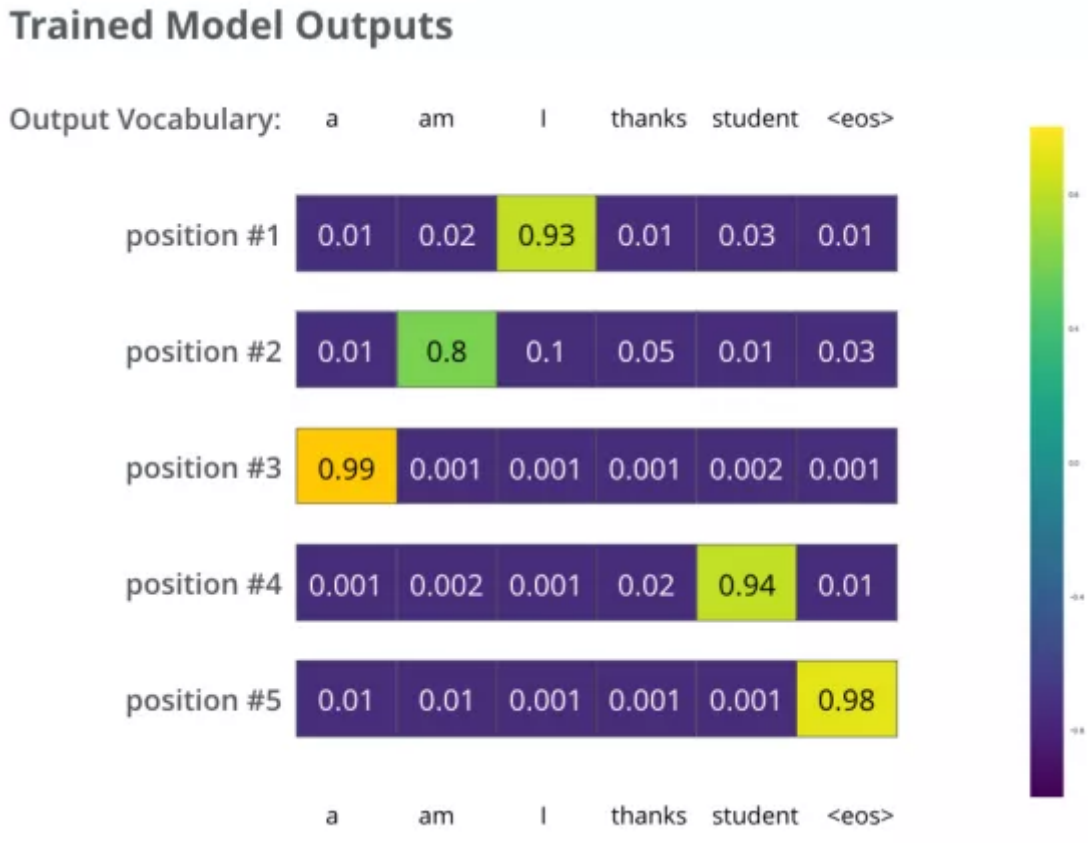
但是请注意，这个例子过于简单了。更贴近实际一点，我们将使用由不止一个单词组成的句子。例如，输入：“je suis étudiant”，预期输出：“I am a student”。这实际上意味着，我们希望自己的模型连续输出一些概率分布，其中：

- 每个概率分布都由一个宽度为vocab_size的向量表示（在我们的简单示例中vocab_size为6，但更贴近实际情况的数量往往为30,000或50,000）
- 第一个概率分布在单词“i”的相关单元中具有最高概率
- 第二个概率分布在单词“am”的相关单元中具有最高概率
- 依此类推，直到第五个输出分布标志着“ <end of sentense>”符号，该符号也具有自己的单元格，也处在10,000个元素词汇表中。



在训练示例中针对一个样本句子，我们将会参照这个目标概率分布训练我们的模型。

将模型在一个在足够大的数据集上训练足够长的时间之后，我们希望产生的概率分布能像下面这样：



理想情况下，经过训练，该模型将输出我们所期待的正确译文。当然，这并不能表明该短语是否属于训练数据集（请

参阅：交叉验证https://www.youtube.com/watch?v=TIgfjmp-4BA）。请注意，即使不可能成为该步的输出，每个位置也会获得一点概率——这是softmax的一个非常有用的特性，可以帮助训练过程。

由于该模型每次生成一个输出，因此我们可以假定模型会从概率分布中选择具有最高概率的一个单词，然后丢弃其余的。这是其中的一种方法（称为贪婪解码，greedy decoding）。还有另一种方法是，比如先确定前两个单词（例如，“ I”和“ a”），然后下一步，运行模型两次：第一次假设第一个输出位置为单词“I”，第二次假设第一个输出位置是单词“a”，并且最终采用在位置＃1和＃2误差更小的版本。我们在#2和#3等位置重复此操作。此方法称为“beam search”，在我们举的例子中，beam_size为2（这意味着在内存中始终都保留有两个部分假设（未完成的翻译）），top_beams也为2（意味着我们将返回两份译文）。对于这些超参数你都可以自己进行试验。

阅读更多

希望本文能对你有用，让你能开始逐渐理解Transformer的主要概念。如果你想更进一步，建议按照以下步骤逐步学习：

- 阅读Attention is All You Need论文，Transformer博客文章（Transformer：A Novel Neural Network Architecture for Language Understanding）以及Tensor2Tensor公告。

博客地址：https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

- 观看ŁukaszKaiser详细讲解模型的演讲（https://www.youtube.com/watch?v=rBCqOTEfxvg）
- 探索Tensor2Tensor repo中提供的Jupyter Notebook
- 探索Tensor2Tensor repo

进一步学习：

- Depthwise Separable Convolutions for Neural Machine Translation（https://arxiv.org/abs/1706.03059）
- One Model To Learn Them All（https://arxiv.org/abs/1706.05137）