The Illustrated Transformer

图解变压器

Discussions: Hacker News (65 points, 4 comments) (https://news.ycombinator.com/item?id=18351674), Reddit r/MachineLearning (29 points, 3 comments) (https://www.reddit.com/r/MachineLearning/comments/8uh2yz/p_the_illustrated_transformer_a_visual_look_at/)

讨论： Hacker News （65分， 4 评论）， Reddit r/MachineLearning （29 分， 3 评论）

Translations: Arabic (https://www.mundhor.site/post/post14), Chinese (Simplified) 1 (https://blog.csdn.net/yujianmin1990/article/details/85221271), Chinese (Simplified) 2 (https://blog.csdn.net/qq_36667170/article/details/124359818), French 1 (https://a-coles.github.io/2020/11/15/transformer-illustre.html), French 2 (https://lbourdois.github.io/blog/nlp/Transformer/), Italian (https://medium.com/@val.mannucci/il-transformer-illustrato-it-37a78e3e2348), Japanese (https://tips-memo.com/translation-jayalmmar-transformer), Korean (https://nlpinkorean.github.io/illustrated-transformer/), Persian (http://dml.qom.ac.ir/2022/05/17/illustrated-transformer/), Russian (https://habr.com/ru/post/486358/), Spanish 1 (https://www.ibidemgroup.com/edu/transformer-ilustrado-jay-alammar/), Spanish 2 (https://hackernoon.com/el-transformador-ilustrado-una-traduccion-al-espanol-0y73wwp), Vietnamese (https://trituenhantao.io/tin-tuc/minh-hoa-transformer/)

翻译： 阿拉伯语， 中文（简体） 1， 中文（简体） 2， 法语 1， 法语 2， 意大利语， 日语， 韩语， 波斯语， 俄语， 西班牙语 1， 西班牙语 2， 越南语

Watch: MIT's Deep Learning State of the Art (https://youtu.be/53YvP6gdD7U?t=432) lecture referencing this post

观看：麻省理工学院的深度学习艺术讲座引用这篇文章

In the previous post, we looked at Attention (https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/) – a ubiquitous method in modern deep learning models. Attention is a concept that helped improve the performance of neural machine translation applications. In this post, we will look at **The Transformer** – a model that uses attention to boost the speed with which these models can be trained. The Transformer outperforms the Google Neural Machine Translation model in specific tasks. The biggest benefit, however, comes from how The Transformer lends itself to parallelization. It is in fact Google Cloud's recommendation to use The Transformer as a reference model to use their Cloud TPU (https://cloud.google.com/tpu/) offering. So let's try to break the model apart and look at how it functions.

在上一篇文章中，我们研究了注意力——现代深度学习模型中一种无处不在的方法。注意力是一个有助于提高神经机器翻译应用程序性能的概念。在这篇文章中，我们将看看 The Transformer——一个利用注意力来提高这些模型训练速度的模型。转换器在特定任务中优于谷歌神经机器翻译模型。然而，最大的好处来自变形金刚如何适应并行化。事实上，谷歌云建议使用The Transformer作为参考模型来使用他们的Cloud TPU产品。因此，让我们尝试将模型分解并查看其功能。

The Transformer was proposed in the paper Attention is All You Need (https://arxiv.org/abs/1706.03762). A TensorFlow implementation of it is available as a part of the Tensor2Tensor (https://github.com/tensorflow/tensor2tensor) package. Harvard's NLP group created a guide annotating the paper with PyTorch implementation (http://nlp.seas.harvard.edu/2018/04/03/attention.html). In this post, we will attempt to oversimplify things a bit and introduce the concepts one by one to hopefully make it easier to understand to people without in-depth knowledge of the subject matter.

变压器是在论文《注意力是你所需要的一切》中提出的。它的TensorFlow实现可以作为Tensor2Tensor包的一部分。哈佛大学的NLP小组创建了一个指南，用PyTorch实现来注释论文。在这篇文章中，我们将尝试过度简化事情，并逐一介绍概念，希望使没有深入了解该主题的人更容易理解。

2020 Update: I've created a "Narrated Transformer" video which is a gentler approach to the topic:

2020 年更新：我创建了一个"旁白变形金刚"视频，这是对该主题的更温和的方法：

A High-Level Look 高级外观

Let's begin by looking at the model as a single black box. In a machine translation application, it would take a sentence in one language, and output its translation in another.

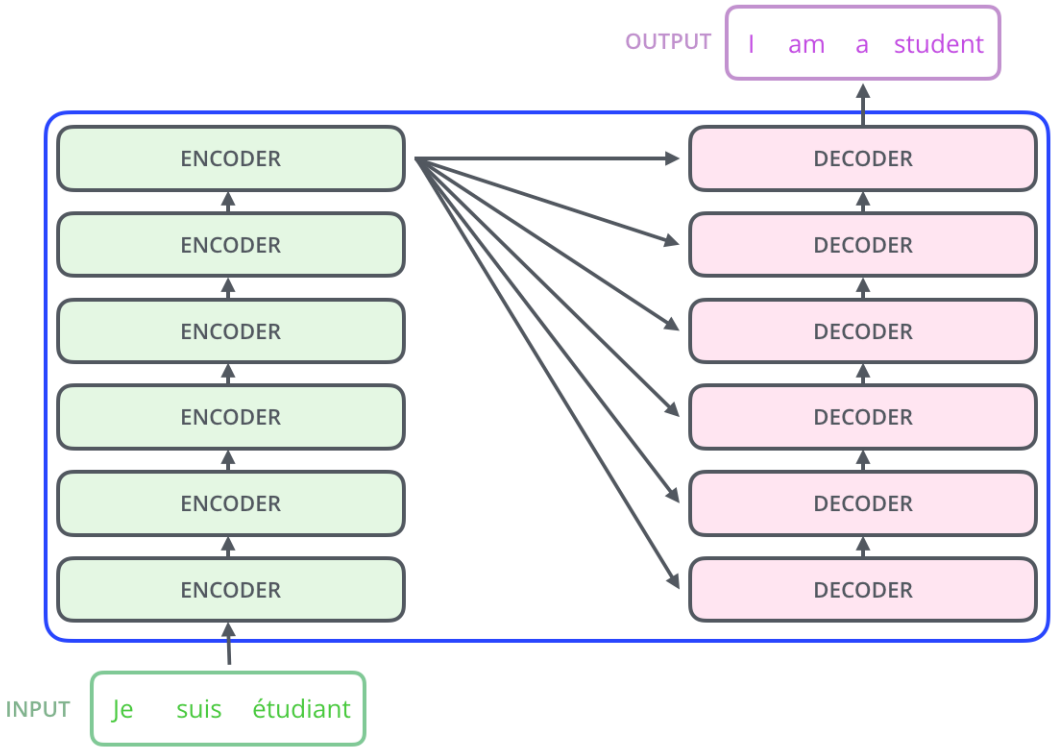
让我们首先将模型视为单个黑盒。在机器翻译应用程序中，它将采用一种语言的句子，然后输出另一种语言的翻译。

Popping open that Optimus Prime goodness, we see an encoding component, a decoding component, and connections between them.

打开擎天柱的优点，我们看到一个编码组件，一个解码组件，以及它们之间的联系。

The encoding component is a stack of encoders (the paper stacks six of them on top of each other – there's nothing magical about the number six, one can definitely experiment with other arrangements). The decoding component is a stack of decoders of the same number.

编码组件是一堆编码器（纸张将其中六个堆叠在一起 - 数字六没有什么神奇之处，人们绝对可以尝试其他安排）。解码组件是一堆相同编号的解码器。



The encoders are all identical in structure (yet they do not share weights). Each one is broken down into two sub-layers:

编码器在结构上都是相同的（但它们不共享权重）。每个子层都分为两个子层：

The encoder's inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word. We'll look closer at self-attention later in the post.

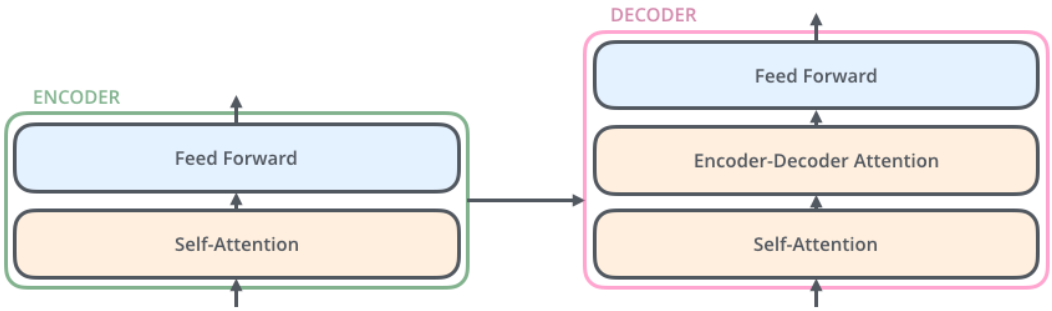
编码器的输入首先流经自我注意层 - 该层可帮助编码器在编码特定单词时查看输入句子中的其他单词。我们将在帖子后面仔细研究自我关注。

The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.

自我注意层的输出被馈送到前馈神经网络。完全相同的前馈网络独立应用于每个位置。

The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence (similar what attention does in seq2seq models (https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/)).

解码器具有这两个层，但在它们之间是一个注意力层，可帮助解码器专注于输入句子的相关部分（类似于 seq2seq 模型中的注意力）。



Bringing The Tensors Into The Picture

将张量带入图片

Now that we've seen the major components of the model, let's start to look at the various vectors/tensors and how they flow between these components to turn the input of a trained model into an output.

现在我们已经了解了模型的主要组件，让我们开始看看各种向量/张量以及它们如何在这些组件之间流动，以将训练模型的输入转换为输出。

As is the case in NLP applications in general, we begin by turning each input word into a vector using an embedding algorithm (https://medium.com/deeper-learning/glossary-of-deep-learning-word-embedding-f90c3cec34ca).

与一般的NLP应用程序一样，我们首先使用嵌入算法将每个输入词转换为向量。

Each word is embedded into a vector of size 512. We'll represent those vectors with these simple boxes.

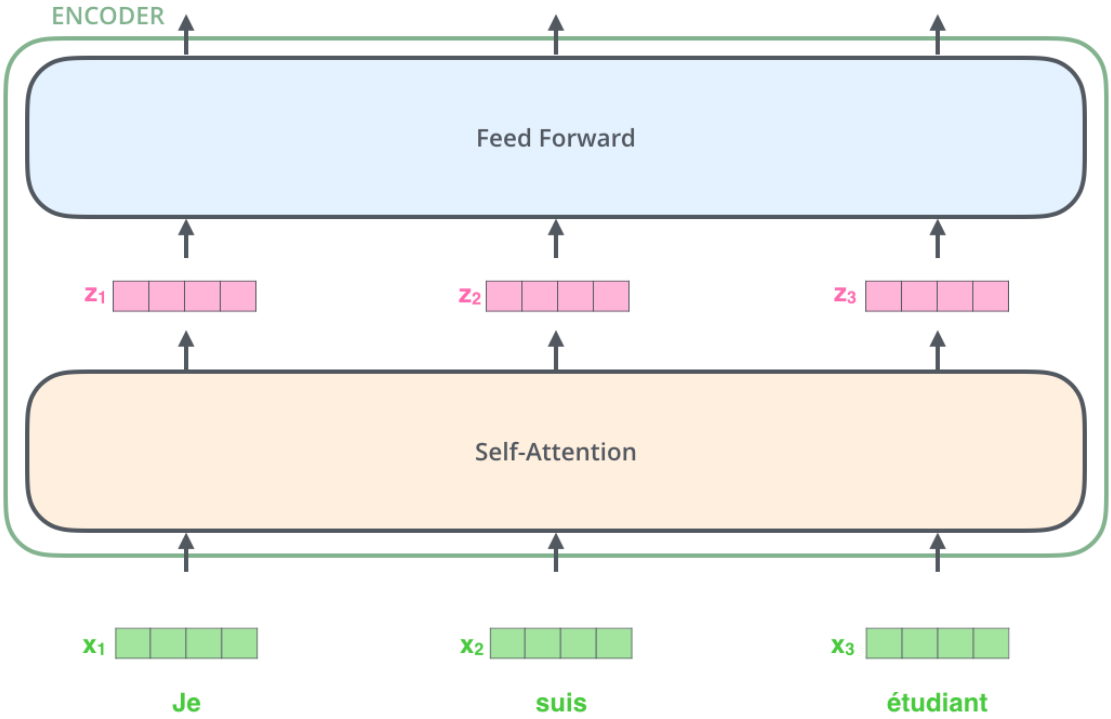
每个单词都嵌入到大小为 512 的向量中。我们将用这些简单的框来表示这些向量。

The embedding only happens in the bottom-most encoder. The abstraction that is common to all the encoders is that they receive a list of vectors each of the size 512 – In the bottom encoder that would be the word embeddings, but in other encoders, it would be the output of the encoder that's directly below. The size of this list is hyperparameter we can set – basically it would be the length of the longest sentence in our training dataset.

嵌入仅发生在最底部的编码器中。所有编码器的共同抽象是它们接收一个大小为 512 的向量列表 – 在底部编码器中，这将是单词嵌入，但在其他编码器中，它将是编码器的输出正下方。这个列表的大小是我们可以设置的超参数——基本上它是我们训练数据集中最长句子的长度。

After embedding the words in our input sequence, each of them flows through each of the two layers of the encoder.

将单词嵌入到我们的输入序列中后，每个单词都流经编码器的两层中的每一层。



Here we begin to see one key property of the Transformer, which is that the word in each position flows through its own path in the encoder. There are dependencies between these paths in the self-attention layer. The feed-forward layer does not have those dependencies, however, and thus the various paths can be executed in parallel while flowing through the feed-forward layer.

在这里，我们开始看到变压器的一个关键属性，即每个位置的单词在编码器中流经其自己的路径。在自我注意层中，这些路径之间存在依赖关系。但是，前馈层没有这些依赖关系，因此各种路径可以在流经前馈层时并行执行。

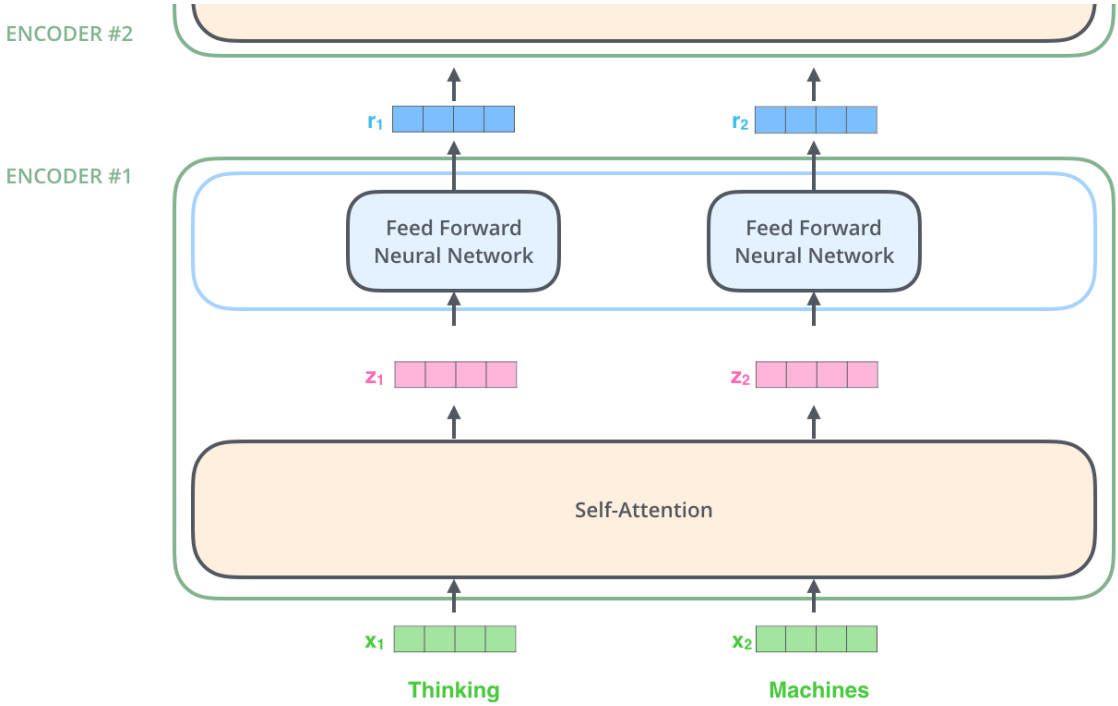
Next, we'll switch up the example to a shorter sentence and we'll look at what happens in each sub-layer of the encoder.

接下来，我们将示例切换为较短的句子，并查看编码器的每个子层中发生的情况。

Now We're Encoding! 现在我们正在编码！

As we've mentioned already, an encoder receives a list of vectors as input. It processes this list by passing these vectors into a 'self-attention' layer, then into a feed-forward neural network, then sends out the output upwards to the next encoder.

正如我们已经提到的，编码器接收一个向量列表作为输入。它通过将这些向量传递到"自我注意"层来处理此列表，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。



The word at each position passes through a self-attention process. Then, they each pass through a feed-forward neural network -- the exact same network with each vector flowing through it separately.

每个位置的单词都经过一个自我注意的过程。然后，它们各自通过一个前馈神经网络——完全相同的网络，每个向量分别流经它。

Self-Attention at a High Level

高水平的自我关注

Don't be fooled by me throwing around the word "self-attention" like it's a concept everyone should be familiar with. I had personally never came across the concept until reading the Attention is All You Need paper. Let us distill how it works.

不要被我抛出"自我关注"这个词所迷惑，就像这是一个每个人都应该熟悉的概念一样。我个人从未遇到过这个概念，直到阅读了"注意力就是你需要的一切"论文。让我们提炼一下它是如何工作的。

Say the following sentence is an input sentence we want to translate:

假设以下句子是我们想要翻译的输入句子：

```
"The animal didn't cross the street because it was too tired" " The animal didn't cross the street because it was too tired "
```

What does "it" in this sentence refer to? Is it referring to the street or to the animal? It's a simple question to a human, but not as simple to an algorithm.

这句话中的"它"指的是什么？它指的是街道还是动物？这对人类来说是一个简单的问题，但对算法来说却没有那么简单。

When the model is processing the word "it", self-attention allows it to associate "it" with "animal".

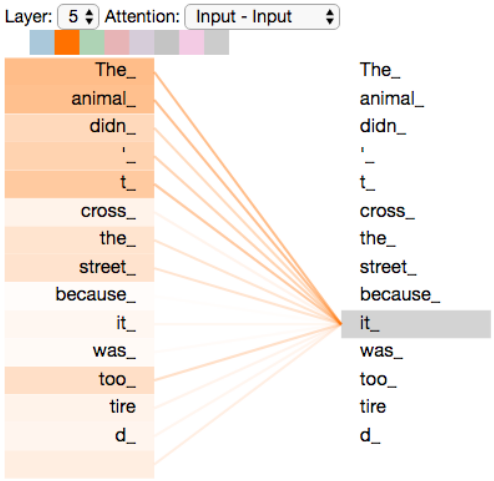
当模型处理单词"it"时，自我注意允许它将"it"与"animal"相关联。

As the model processes each word (each position in the input sequence), self attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word.

当模型处理每个单词（输入序列中的每个位置）时，自我注意允许它查看输入序列中的其他位置，以寻找有助于更好地编码该单词的线索。

If you're familiar with RNNs, think of how maintaining a hidden state allows an RNN to incorporate its representation of previous words/vectors it has processed with the current one it's processing. Self-attention is the method the Transformer uses to bake the "understanding" of other relevant words into the one we're currently processing.

如果您熟悉 RNN，请考虑保持隐藏状态如何允许 RNN 将其先前处理的单词/向量的表示与当前正在处理的单词/向量合并。自我注意是变形金刚用来将对其他相关单词的"理解"烘焙到我们目前正在处理的单词中的方法。



As we are encoding the word "it" in encoder #5 (the top encoder in the stack), part of the attention mechanism was focusing on "The Animal", and baked a part of its representation into the encoding of "it".

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专注于“动物”，并将其表示的一部分烘焙到“it”的编码中。

Be sure to check out the Tensor2Tensor notebook (https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb) where you can load a Transformer model, and examine it using this interactive visualization.

请务必查看 Tensor2Tensor 笔记本，您可以在其中加载转换器模型，并使用此交互式可视化对其进行检查。

Self-Attention in Detail

细节上的自我关注

Let’s first look at how to calculate self-attention using vectors, then proceed to look at how it’s actually implemented – using matrices.

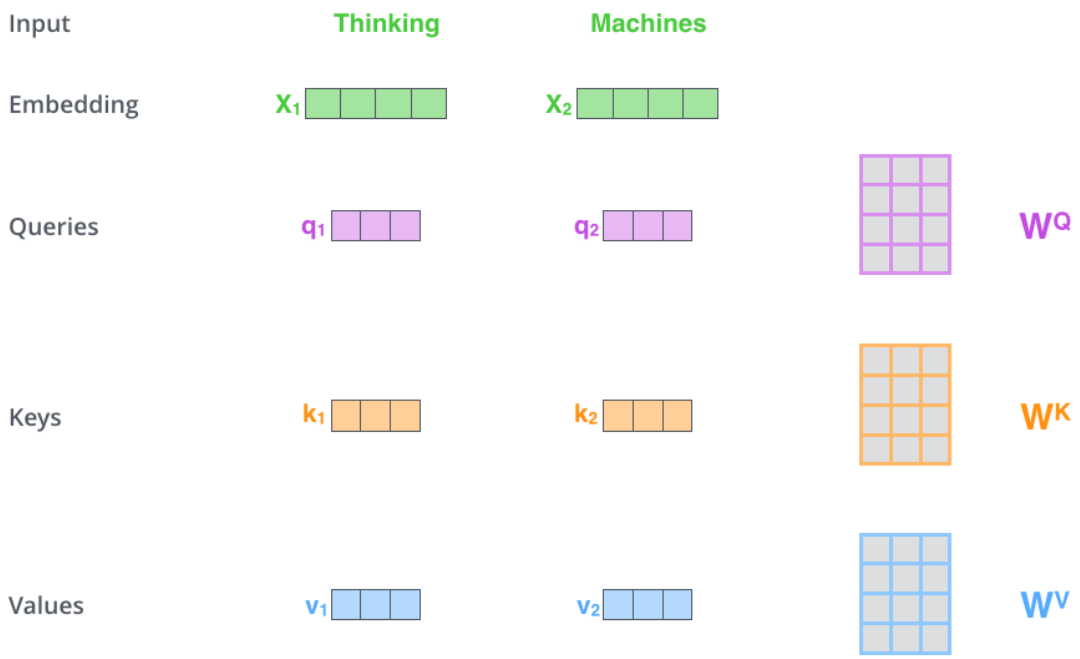
让我们首先看看如何使用向量计算自我注意，然后继续看看它实际上是如何使用矩阵实现的。

The **first step** in calculating self-attention is to create three vectors from each of the encoder’s input vectors (in this case, the embedding of each word). So for each word, we create a Query vector, a Key vector, and a Value vector. These vectors are created by multiplying the embedding by three matrices that we trained during the training process.

计算自我注意的第一步是从编码器的每个输入向量创建三个向量（在本例中为每个单词的嵌入）。因此，对于每个单词，我们创建一个查询向量、一个键向量和一个值向量。这些向量是通过将嵌入乘以我们在训练过程中训练的三个矩阵来创建的。

Notice that these new vectors are smaller in dimension than the embedding vector. Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512. They don’t HAVE to be smaller, this is an architecture choice to make the computation of multiheaded attention (mostly) constant.

请注意，这些新向量的维度小于嵌入向量。它们的维数为 64，而嵌入和编码器输入/输出向量的维数为 512。它们不必更小，这是一种架构选择，可以使多头注意力的计算（大部分）恒定。



Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

将 x1 乘以 WQ 权重矩阵得到 q1，即与该单词关联的“查询”向量。我们最终为输入句子中的每个单词创建一个“查询”、“键”和“值”投影。

What are the “query”, “key”, and “value” vectors?

什么是“查询”、“键”和“值”向量?

They’re abstractions that are useful for calculating and thinking about attention. Once you proceed with reading how attention is calculated below, you’ll know pretty much all you need to know about the role each of these vectors plays.

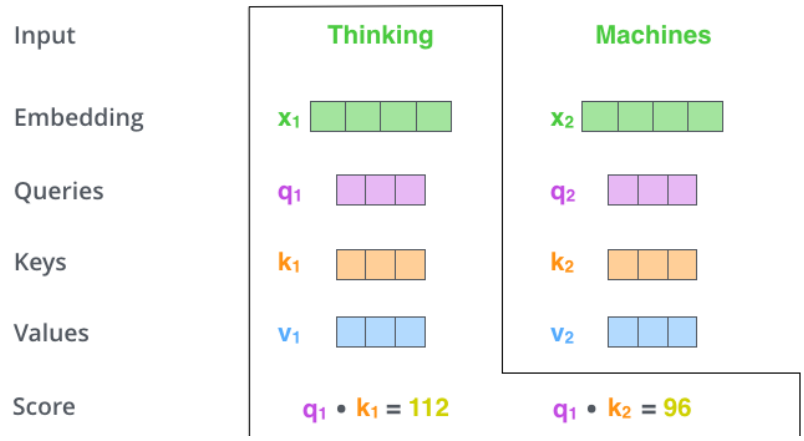
它们是对计算和思考注意力有用的抽象概念。一旦你继续阅读下面的注意力是如何计算的，你就会知道几乎所有你需要知道的关于这些向量所扮演的角色。

The **second step** in calculating self-attention is to calculate a score. Say we’re calculating the self-attention for the first word in this example, “Thinking”. We need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

计算自我注意力的第二步是计算分数。假设我们正在计算此示例中第一个单词“思考”的自我注意。我们需要根据该单词对输入句子的每个单词进行评分。分数决定了当我们在某个位置对单词进行编码时，对输入句子的其他部分的关注程度。

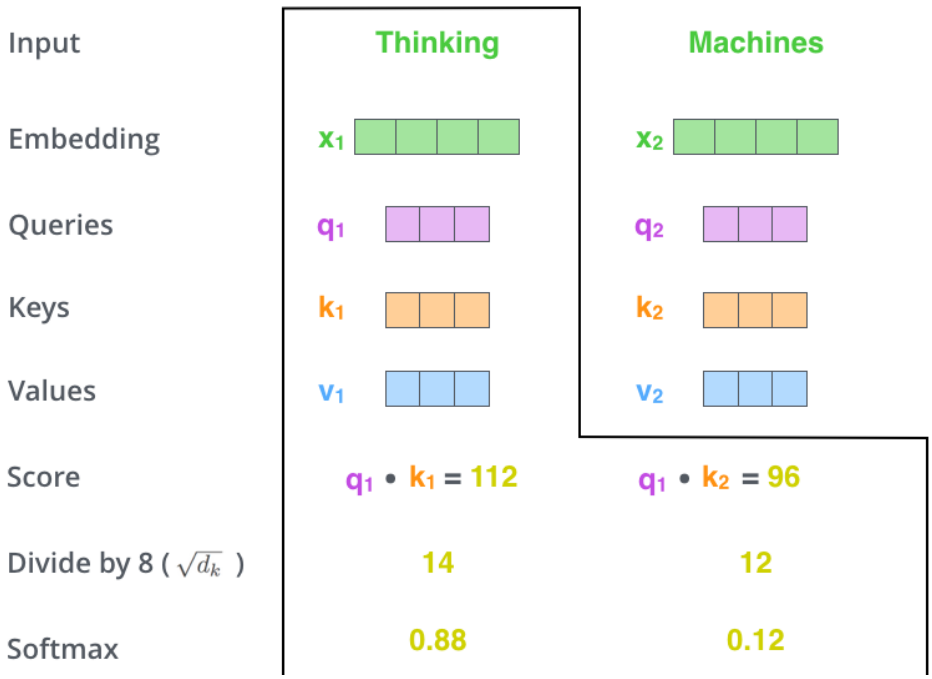
The score is calculated by taking the dot product of the query vector with the key vector of the respective word we’re scoring. So if we’re processing the self-attention for the word in position #1, the first score would be the dot product of q1 and k1. The second score would be the dot product of q1 and k2.

分数的计算方法是将查询向量的点积与我们评分的相应单词的关键向量相提并论。因此，如果我们处理位置 #1 中单词的自我注意，第一个分数将是 q1 和 k1 的点积。第二个分数是 q1 和 k2 的点积。



The **third and fourth steps** are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper – 64. This leads to having more stable gradients. There could be other possible values here, but this is the default), then pass the result through a softmax operation. Softmax normalizes the scores so they're all positive and add up to 1.

第三步和第四步是将分数除以 8（论文中使用的关键向量维度的平方根 – 64。这导致具有更稳定的梯度。这里可能还有其他可能的值，但这是默认值），然后通过 softmax 操作传递结果。Softmax 对分数进行归一化，因此它们都是正数，加起来为 1。



This softmax score determines how much each word will be expressed at this position. Clearly the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word.

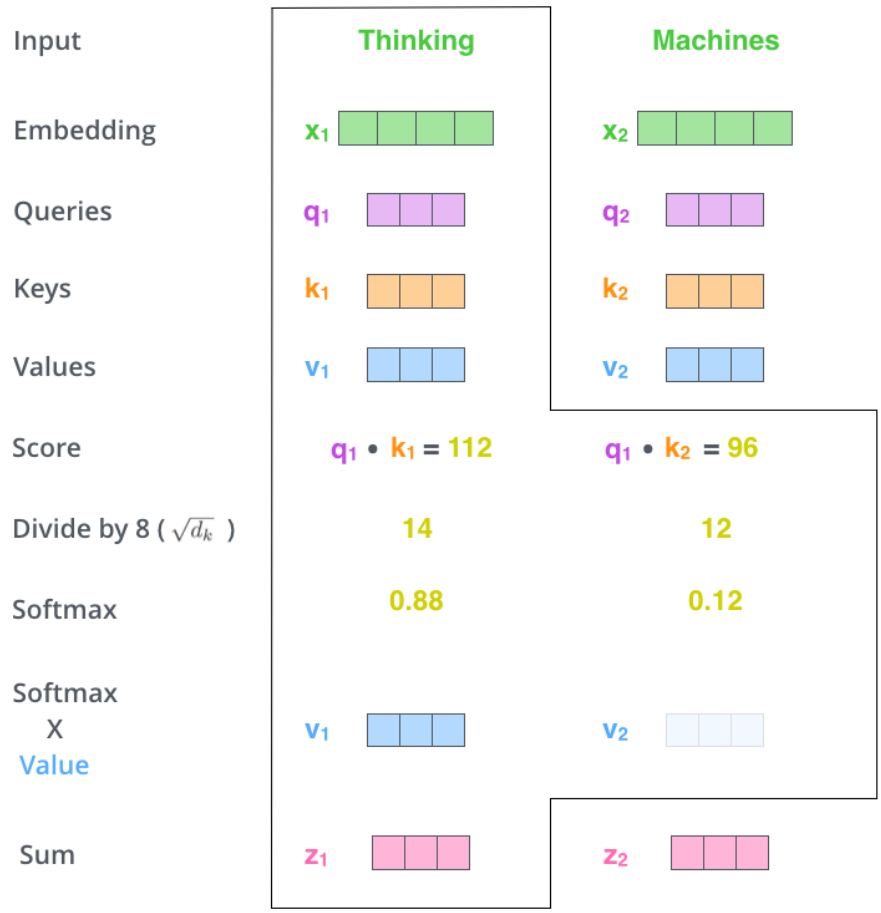
这个softmax分数决定了每个单词在这个位置的表达量。显然，这个位置的单词将具有最高的softmax分数，但有时关注与当前单词相关的另一个单词很有用。

The **fifth step** is to multiply each value vector by the softmax score (in preparation to sum them up). The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).

第五步是将每个值向量乘以softmax分数（准备将它们相加）。这里的直觉是保持我们想要关注的单词的值不变，并淹没不相关的单词（例如，将它们乘以像 0.001 这样的小数字）。

The **sixth step** is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).

第六步是总结加权值向量。这会在此位置（对于第一个单词）产生自我注意层的输出。



That concludes the self-attention calculation. The resulting vector is one we can send along to the feed-forward neural network. In the actual implementation, however, this calculation is done in matrix form for faster processing. So let's look at that now that we've seen the intuition of the calculation on the word level.

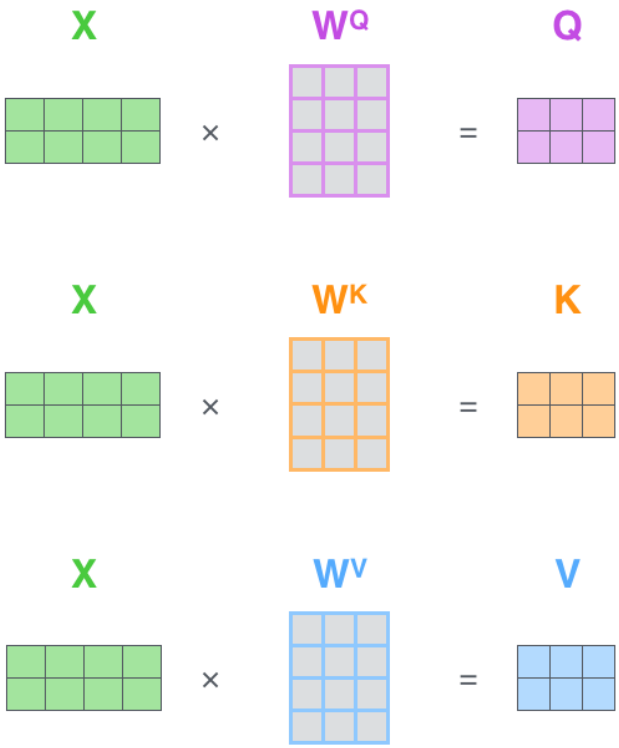
自我注意力计算到此结束。生成的向量是我们可以发送到前馈神经网络的向量。然而，在实际实现中，这种计算是以矩阵形式完成的，以便更快地处理。因此，现在让我们看看，我们已经看到了单词级别的计算直觉。

Matrix Calculation of Self-Attention

自我注意的矩阵计算

The first step is to calculate the Query, Key, and Value matrices. We do that by packing our embeddings into a matrix X, and multiplying it by the weight matrices we've trained (WQ, WK, WV).

第一步是计算查询、键和值矩阵。为此，我们将嵌入打包到矩阵X中，并将其乘以我们训练的权重矩阵（WQ，WK，WV）。

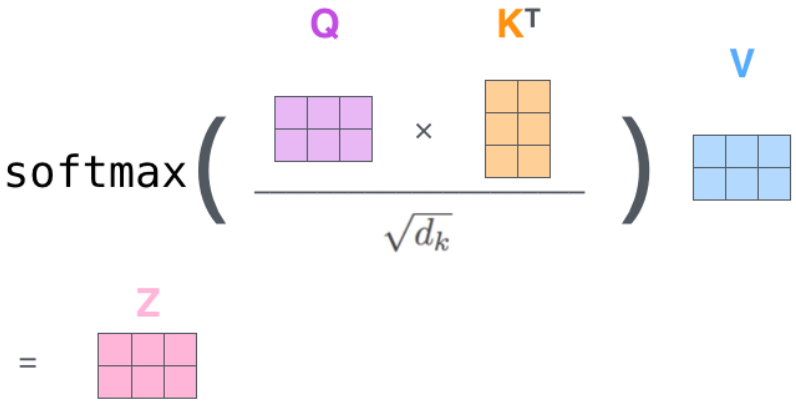


Every row in the X matrix corresponds to a word in the input sentence. We again see the difference in size of the embedding vector (512, or 4 boxes in the figure), and the q/k/v vectors (64, or 3 boxes in the figure)

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 512 或 4 个框）和 q/k/v 向量（图中 64 个或 3 个框）的大小差异

Finally, since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.

最后，由于我们正在处理矩阵，我们可以将步骤 2 到 6 压缩在一个公式中，以计算自我注意层的输出。



The self-attention calculation in matrix form

矩阵形式的自我注意计算

The Beast With Many Heads

多头兽

The paper further refined the self-attention layer by adding a mechanism called "multi-headed" attention. This improves the performance of the attention layer in two ways:

该论文通过添加一种称为"多头"注意的机制进一步完善了自我注意层。这通过两种方式提高了注意力层的性能：

1. It expands the model's ability to focus on different positions. Yes, in the example above, z1 contains a little bit of every other encoding, but it could be dominated by the actual word itself. If we're translating a sentence like "The animal didn't cross the street because it was too tired", it would be useful to know which word "it" refers to.

   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 包含一点其他编码，但它可能由实际单词本身主导。如果我们翻译一个句子，比如"动物没有过马路，因为它太累了"，知道"它"指的是哪个词会很有用。

2. It gives the attention layer multiple "representation subspaces". As we'll see next, with multi-headed attention we have not only one, but multiple sets of Query/Key/Value weight matrices (the Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder). Each of these sets is randomly initialized. Then, after training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace.

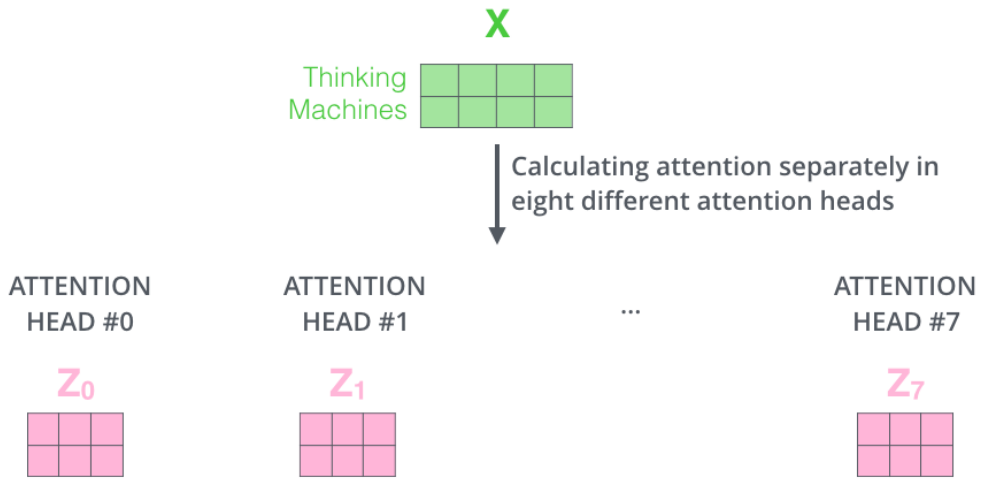
   它为注意力层提供了多个"表示子空间"。正如我们接下来将看到的，对于多头注意力，我们不仅有一组，而是多组查询/键/值权重矩阵（转换器使用八个注意力头，所以我们最终为每个编码器/解码器有八组）。这些集合中的每一个都是随机初始化的。然后，在训练后，每个集合用于将输入嵌入（或来自较低编码器/解码器的向量）投影到不同的表示子空间中。

With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices. As we did before, we multiply X by the WQ/WK/WV matrices to produce Q/K/V matrices.

通过多头注意，我们为每个头部维护单独的 Q/K/V 权重矩阵，从而产生不同的 Q/K/V 矩阵。和以前一样，我们将 X 乘以 WQ/WK/WV 矩阵以产生 Q/K/V 矩阵。

If we do the same self-attention calculation we outlined above, just eight different times with different weight matrices, we end up with eight different Z matrices

如果我们做上面概述的相同的自我注意计算，只有八次不同的时间使用不同的权重矩阵，我们最终得到八个不同的 Z 矩阵

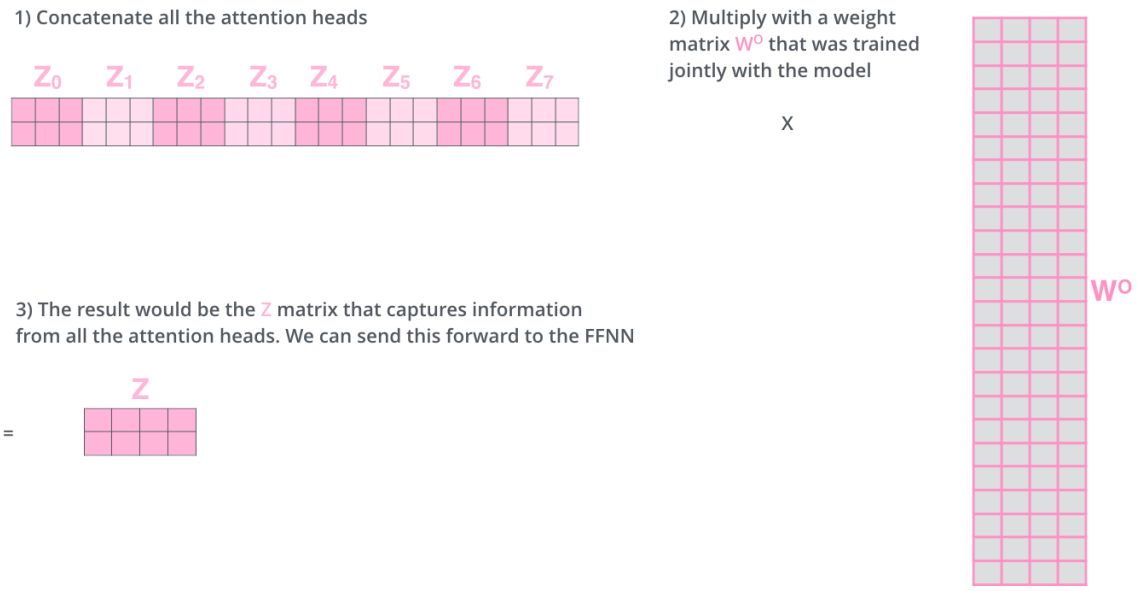


This leaves us with a bit of a challenge. The feed-forward layer is not expecting eight matrices – it's expecting a single matrix (a vector for each word). So we need a way to condense these eight down into a single matrix.

这给我们带来了一些挑战。前馈层不需要八个矩阵 - 它期望单个矩阵（每个单词的向量）。因此，我们需要一种方法将这八个压缩成一个矩阵。

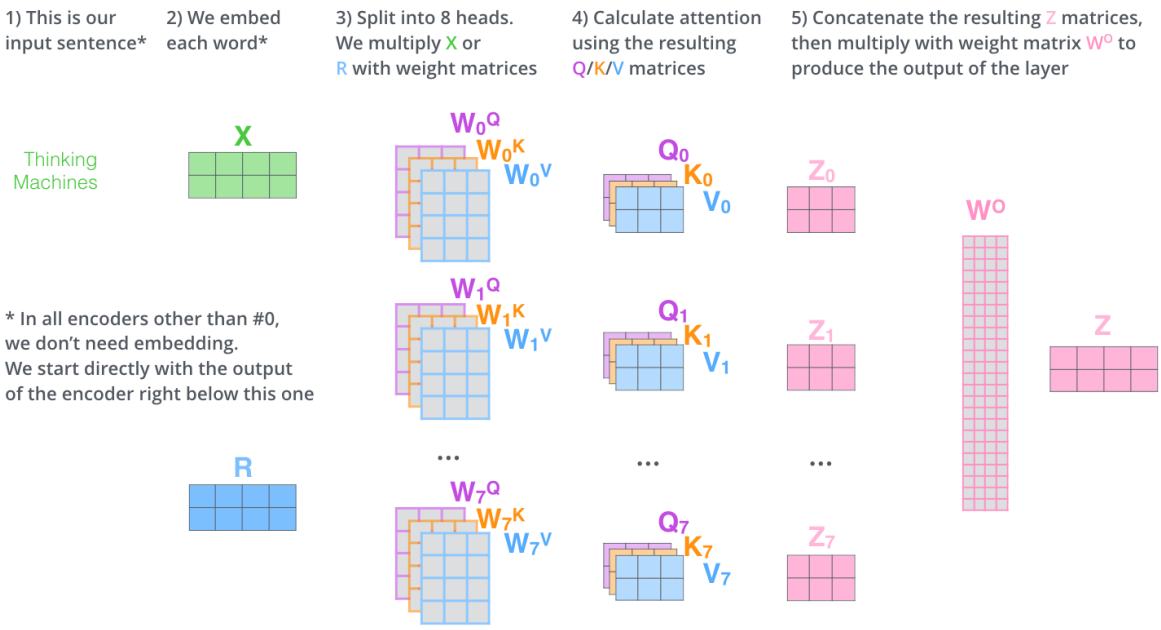
How do we do that? We concat the matrices then multiply them by an additional weights matrix WO.

我们如何做到这一点？我们将矩阵连接起来，然后将它们乘以一个额外的权重矩阵 WO。



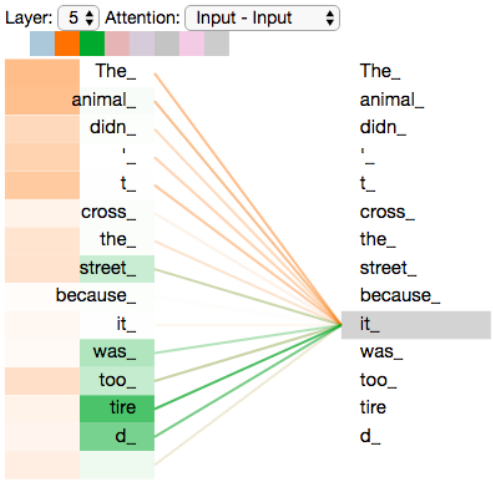
That's pretty much all there is to multi-headed self-attention. It's quite a handful of matrices, I realize. Let me try to put them all in one visual so we can look at them in one place

这几乎就是多头自我关注的全部内容。我意识到这是相当多的矩阵。让我尝试将它们全部放在一个视觉对象中，以便我们可以在一个地方查看它们



Now that we have touched upon attention heads, let's revisit our example from before to see where the different attention heads are focusing as we encode the word "it" in our example sentence:

现在我们已经触及了注意力头，让我们重新审视一下之前的例子，看看当我们在例句中编码单词"it"时，不同的注意力头集中在哪里：

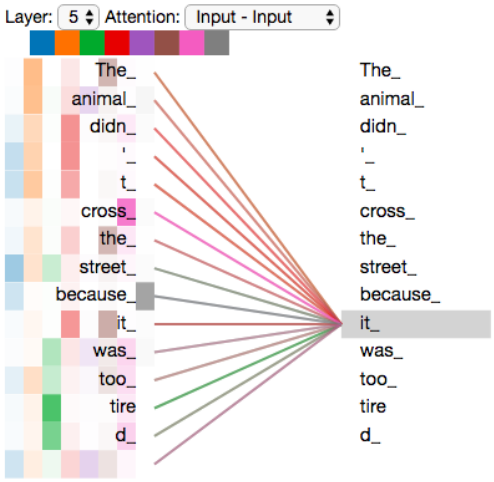


As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".

当我们对“它”这个词进行编码时，一个注意力头最关注“动物”，而另一个注意力集中在“疲惫”上——从某种意义上说，模型对“它”这个词的表示融入了“动物”和“疲惫”的一些表示。

If we add all the attention heads to the picture, however, things can be harder to interpret:

但是，如果我们将所有注意力都添加到图片中，事情可能更难解释：



Representing The Order of The Sequence Using Positional Encoding

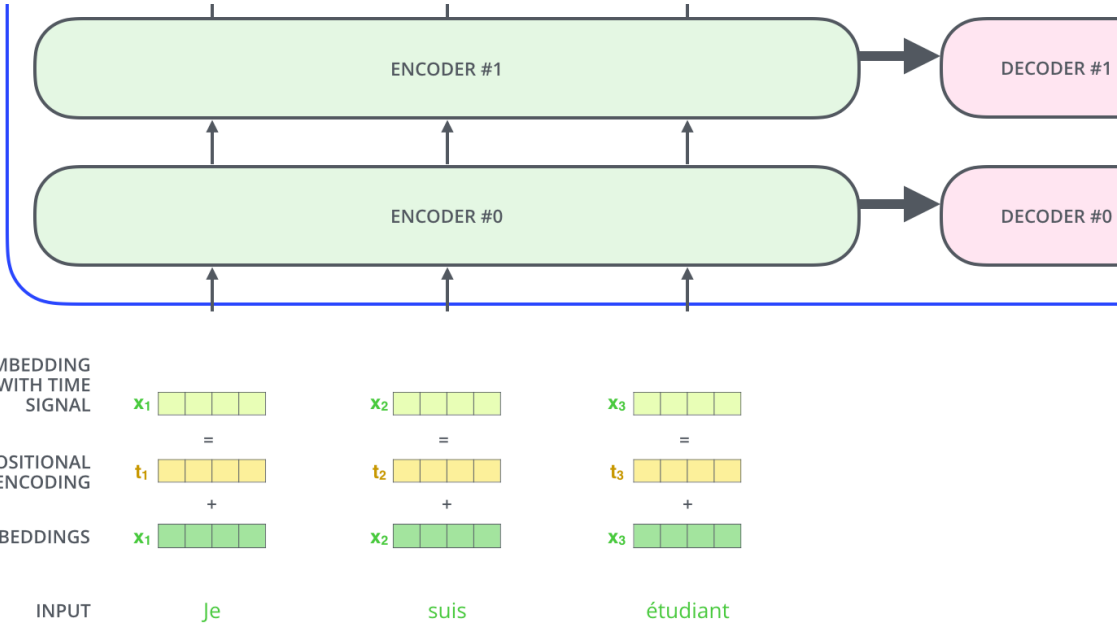
使用位置编码表示序列的顺序

One thing that's missing from the model as we have described it so far is a way to account for the order of the words in the input sequence.

到目前为止，模型中缺少的一件事是解释输入序列中单词顺序的方法。

To address this, the transformer adds a vector to each input embedding. These vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or the distance between different words in the sequence. The intuition here is that adding these values to the embeddings provides meaningful distances between the embedding vectors once they're projected into Q/K/V vectors and during dot-product attention.

为了解决这个问题，转换器向每个输入嵌入添加一个向量。这些向量遵循模型学习的特定模式，这有助于它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，将这些值添加到嵌入中，一旦嵌入向量投影到 Q/K/V 向量中，并且在点积注意期间，嵌入向量之间就会提供有意义的距离。

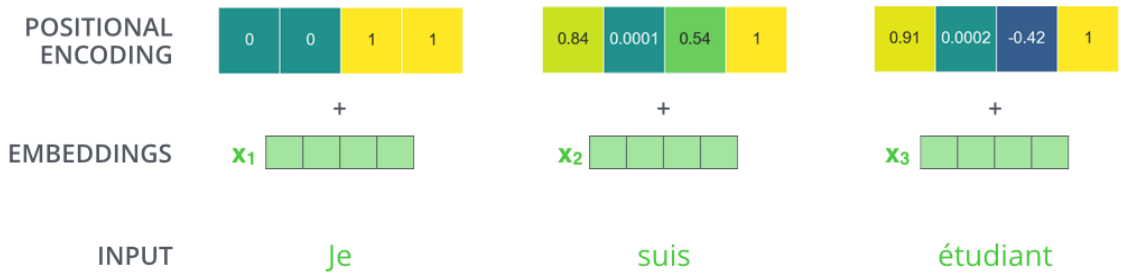


To give the model a sense of the order of the words, we add positional encoding vectors -- the values of which follow a specific pattern.

为了让模型了解单词的顺序，我们添加了位置编码向量 - 其值遵循特定的模式。

If we assumed the embedding has a dimensionality of 4, the actual positional encodings would look like this:

如果我们假设嵌入的维数为 4，则实际的位置编码如下所示：



A real example of positional encoding with a toy embedding size of 4

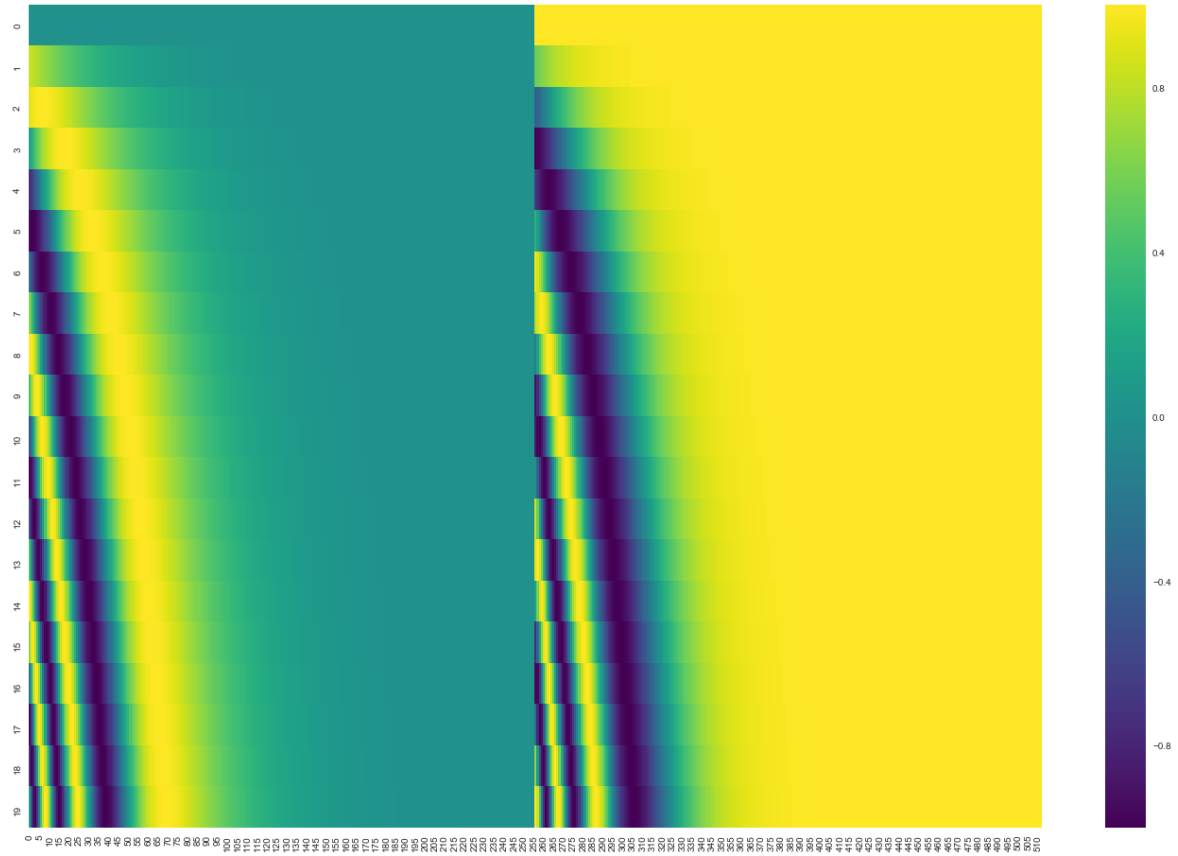
玩具嵌入大小为 4 的位置编码的真实示例

What might this pattern look like?

这种模式可能是什么样子的?

In the following figure, each row corresponds to a positional encoding of a vector. So the first row would be the vector we'd add to the embedding of the first word in an input sequence. Each row contains 512 values – each with a value between 1 and -1. We've color-coded them so the pattern is visible.

在下图中，每一行对应于一个向量的位置编码。因此，第一行是我们添加到输入序列中第一个单词嵌入的向量。每行包含 512 个值 - 每个值的值介于 1 和 -1 之间。我们对它们进行了颜色编码，因此图案可见。



A real example of positional encoding for 20 words (rows) with an embedding size of 512 (columns). You can see that it appears split in half down the center. That's because the values of the left half are generated by one function (which uses sine), and the right half is generated by another function (which uses cosine). They're then concatenated to form each of the positional encoding vectors.

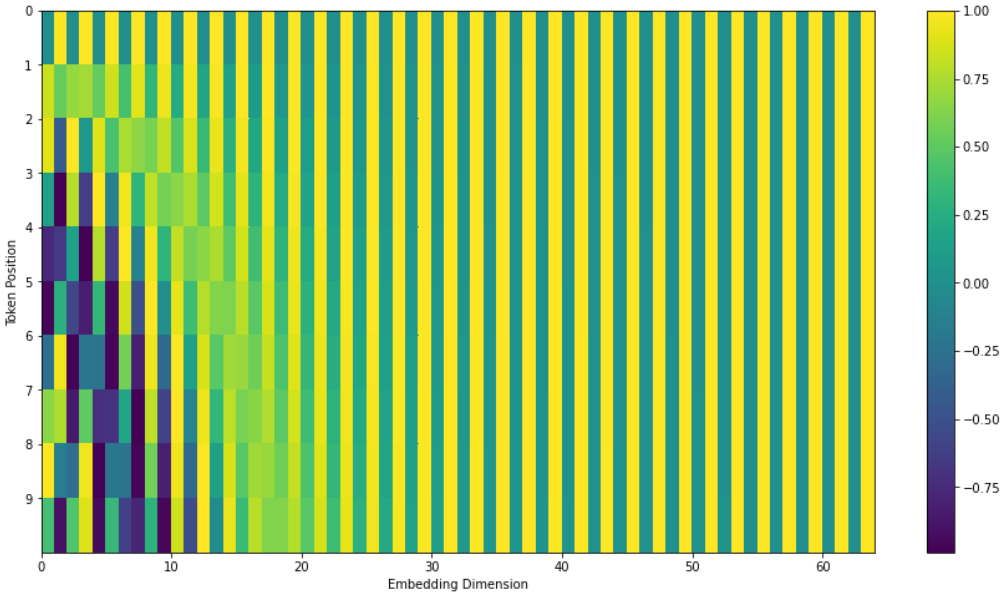
嵌入大小为 512（列）的 20 个单词（行）的位置编码的真实示例。您可以看到它似乎在中心向下分成两半。这是因为左半部分的值是由一个函数（使用正弦）生成的，而右半部分的值是由另一个函数（使用余弦）生成的。然后将它们连接起来形成每个位置编码向量。

The formula for positional encoding is described in the paper (section 3.5). You can see the code for generating positional encodings in `get_timing_signal_1d()` (https://github.com/tensorflow/tensor2tensor/blob/23bd23b9830059fbc349381b70d9429b5c40a139/tensor2tensor/layers/common_a This is not the only possible method for positional encoding. It, however, gives the advantage of being able to scale to unseen lengths of sequences (e.g. if our trained model is asked to translate a sentence longer than any of those in our training set).

论文中描述了位置编码的公式（第3.5节）。您可以在 中看到 `get_timing_signal_1d()` 用于生成位置编码的代码。这不是位置编码的唯一可能方法。然而，它的优势在于能够扩展到看不见的序列长度（例如，如果我们训练的模型被要求翻译一个句子比我们训练集中的任何句子都长）。

July 2020 Update: The positional encoding shown above is from the Tensor2Tensor implementation of the Transformer. The method shown in the paper is slightly different in that it doesn't directly concatenate, but interweaves the two signals. The following figure shows what that looks like. Here's the code to generate it (https://github.com/jalammar/jalammar.github.io/blob/master/notebookes/transformer/transformer_positional_encoding_graph.ipynb)

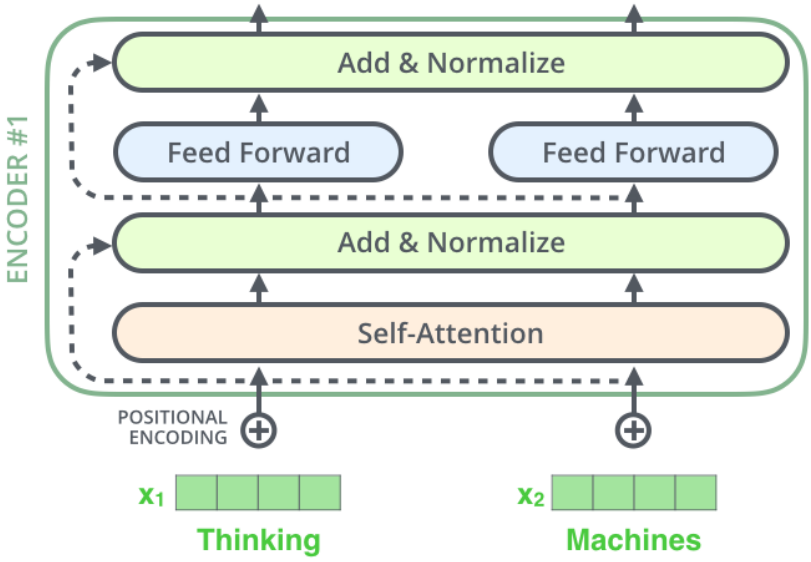
2020 年 7 月更新：上面显示的位置编码来自转换器的 Tensor2Tensor 实现。论文中显示的方法略有不同，因为它不直接连接，而是交织两个信号。下图显示了它的外观。下面是生成它的代码：



The Residuals 残余物

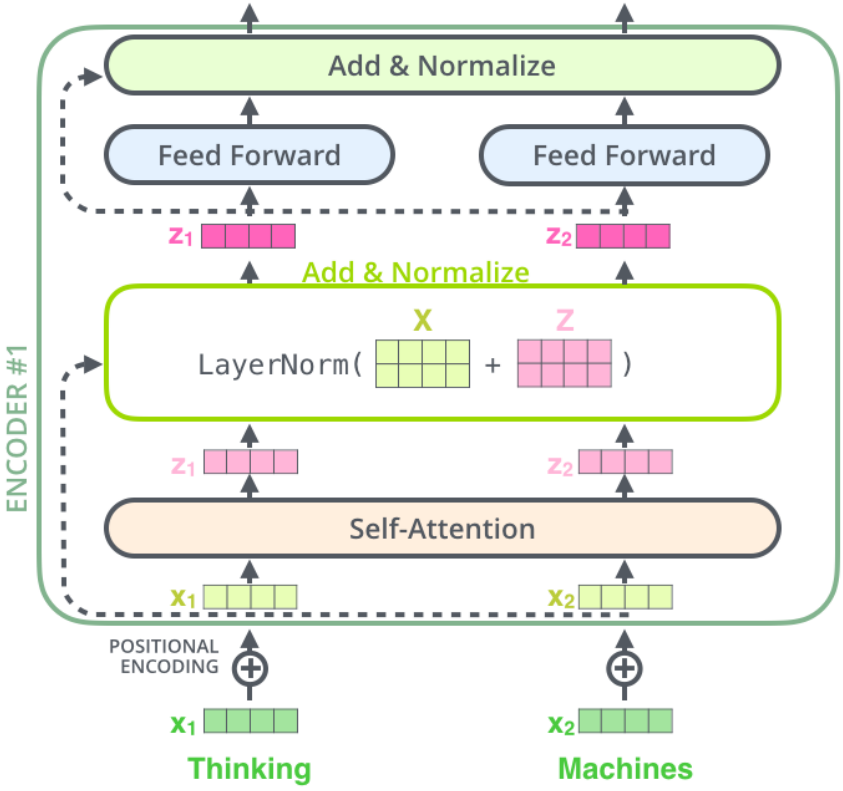
One detail in the architecture of the encoder that we need to mention before moving on, is that each sub-layer (self-attention, ffnn) in each encoder has a residual connection around it, and is followed by a layer-normalization (https://arxiv.org/abs/1607.06450) step.

在继续之前，我们需要提及编码器架构中的一个细节是，每个编码器中的每个子层（自我注意，ffnn）周围都有一个残差连接，然后是层规范化步骤。



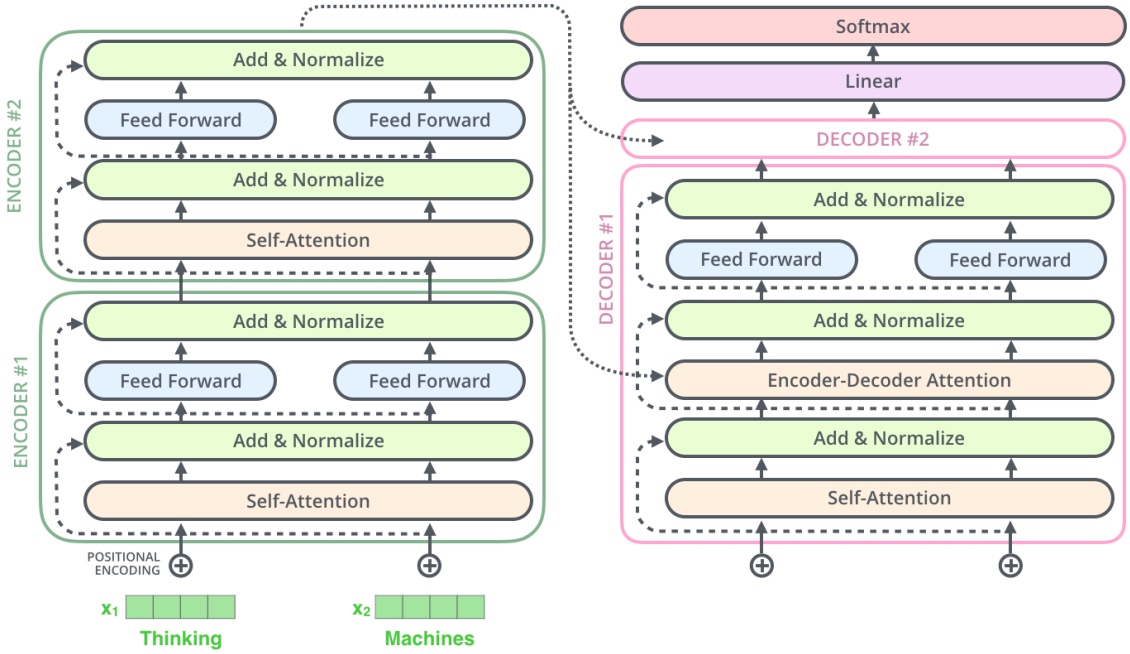
If we're to visualize the vectors and the layer-norm operation associated with self attention, it would look like this:

如果我们要可视化与自我注意相关的向量和层范数运算，它看起来像这样：



This goes for the sub-layers of the decoder as well. If we're to think of a Transformer of 2 stacked encoders and decoders, it would look something like this:

这也适用于解码器的子层。如果我们要考虑一个由 2 个堆叠编码器和解码器的 Transformer 组成，它看起来像这样：



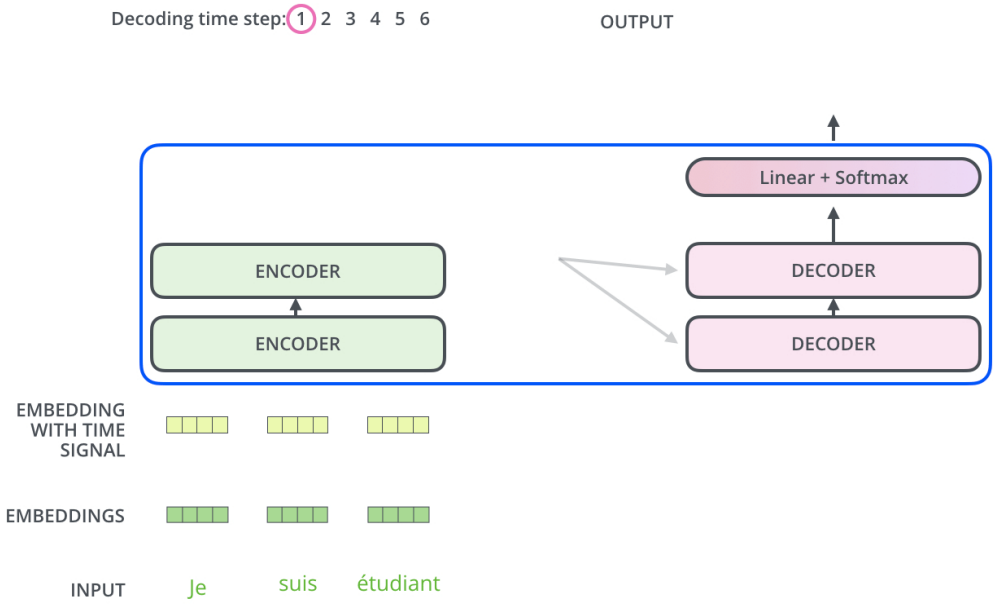
The Decoder Side 解码器端

Now that we've covered most of the concepts on the encoder side, we basically know how the components of decoders work as well. But let's take a look at how they work together.

现在我们已经介绍了编码器端的大多数概念，我们基本上也知道解码器的组件是如何工作的。但是，让我们来看看它们是如何协同工作的。

The encoder start by processing the input sequence. The output of the top encoder is then transformed into a set of attention vectors K and V. These are to be used by each decoder in its "encoder-decoder attention" layer which helps the decoder focus on appropriate places in the input sequence:

编码器首先处理输入序列。然后将顶部编码器的输出转换为一组注意力向量 K 和 V。每个解码器在其“编码器-解码器注意”层中使用这些，这有助于解码器专注于输入序列中的适当位置：

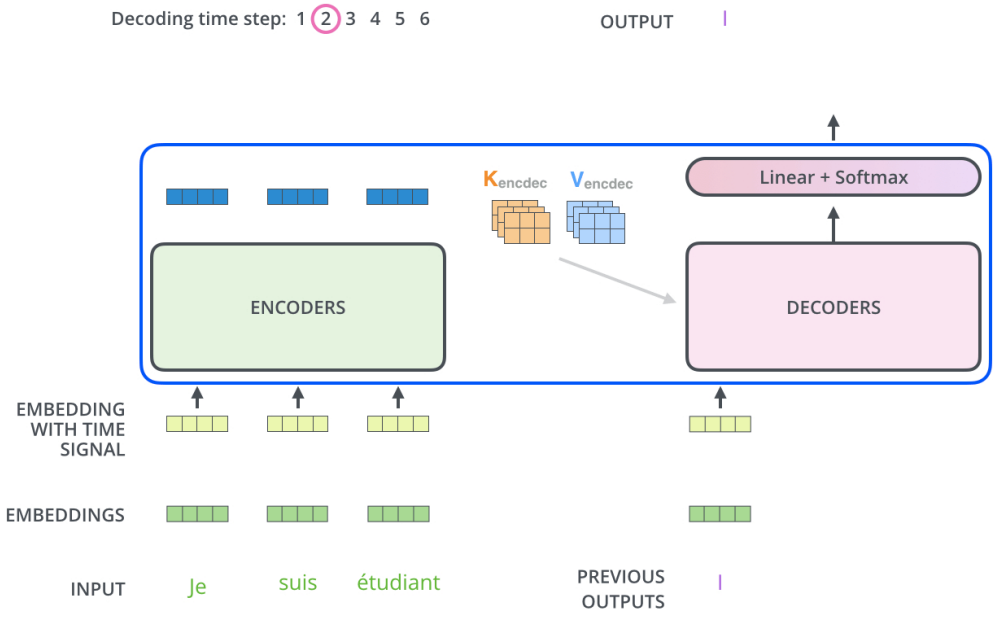


After finishing the encoding phase, we begin the decoding phase. Each step in the decoding phase outputs an element from the output sequence (the English translation sentence in this case).

完成编码阶段后，我们开始解码阶段。解码阶段的每个步骤都会从输出序列中输出一个元素（在本例中为英语翻译句子）。

The following steps repeat the process until a special symbol is reached indicating the transformer decoder has completed its output. The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results just like the encoders did. And just like we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to indicate the position of each word.

以下步骤重复该过程，直到到达指示转换器解码器已完成其输出的特殊符号。每个步骤的输出在下一个时间步中馈送到底部解码器，解码器就像编码器一样冒出解码结果。就像我们对编码器输入所做的那样，我们在这些解码器输入中嵌入并添加位置编码，以指示每个单词的位置。



The self attention layers in the decoder operate in a slightly different way than the one in the encoder:

解码器中的自注意层的工作方式与编码器中的自注意层略有不同：

In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to `-inf`) before the softmax step in the self-attention calculation.

在解码器中，自我注意层只允许关注输出序列中的较早位置。这是通过在自我注意计算中的softmax步骤之前屏蔽未来位置（将它们设置为 `-inf` ）来完成的。

The “Encoder-Decoder Attention” layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack.

“编码器-解码器注意力”层的工作方式与多头自我注意类似，只是它从其下方的层创建其查询矩阵，并从编码器堆栈的输出中获取键和值矩阵。

The Final Linear and Softmax Layer

最终的线性层和软最大值层

The decoder stack outputs a vector of floats. How do we turn that into a word? That’s the job of the final Linear layer which is followed by a Softmax Layer.

解码器堆栈输出浮点数向量。我们如何把它变成一个词？这是最终线性层的工作，然后是Softmax层。

The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.

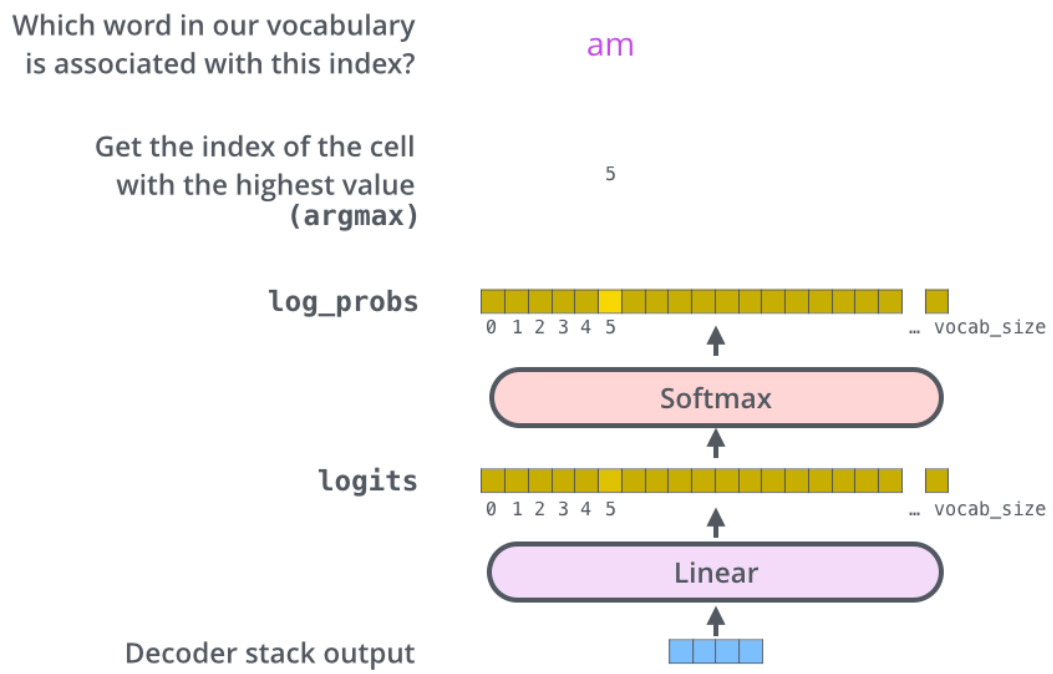
线性层是一个简单的完全连接的神经网络，它将解码器堆栈产生的向量投射到一个更大的向量中，称为logits向量。

Let’s assume that our model knows 10,000 unique English words (our model’s “output vocabulary”) that it’s learned from its training dataset. This would make the logits vector 10,000 cells wide – each cell corresponding to the score of a unique word. That is how we interpret the output of the model followed by the Linear layer.

假设我们的模型知道 10，000 个独特的英语单词（我们模型的“输出词汇表”），这些单词是从其训练数据集中学习的。这将使对数向量宽 10，000 个单元格 - 每个单元格对应于一个唯一单词的分数。这就是我们如何解释模型的输出，然后是线性层。

The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.

然后，softmax层将这些分数转换为概率（全部为正，加起来为1.0）。选择概率最高的像元，并生成与之关联的单词作为此时间步长的输出。



This figure starts from the bottom with the vector produced as the output of the decoder stack. It is then turned into an output word.

该图从底部开始，作为解码器堆栈输出生成的矢量。然后将其转换为输出字。

Recap Of Training 培训回顾

Now that we’ve covered the entire forward-pass process through a trained Transformer, it would be useful to glance at the intuition of training the model.

现在我们已经通过经过训练的转换器涵盖了整个正向传递过程，那么浏览一下训练模型的直觉会很有用。

During training, an untrained model would go through the exact same forward pass. But since we are training it on a labeled training dataset, we can compare its output with the actual correct output.

在训练期间，未经训练的模型将经历完全相同的正向传递。但是由于我们在标记的训练数据集上训练它，我们可以将其输出与实际正确的输出进行比较。

To visualize this, let's assume our output vocabulary only contains six words("a", "am", "i", "thanks", "student", and "<eos>" (short for 'end of sentence')).

为了形象化这一点，让我们假设我们的输出词汇表只包含六个单词（"a"、"am"、"i"、"谢谢"、"学生"和""（"句尾"的缩写））。

Output Vocabulary

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

The output vocabulary of our model is created in the preprocessing phase before we even begin training.

我们模型的输出词汇表是在我们开始训练之前的预处理阶段创建的。

Once we define our output vocabulary, we can use a vector of the same width to indicate each word in our vocabulary. This also known as one-hot encoding. So for example, we can indicate the word "am" using the following vector:

一旦我们定义了输出词汇表，我们就可以使用相同宽度的向量来指示词汇表中的每个单词。这也称为独热编码。例如，我们可以使用以下向量来表示单词"am"：

Output Vocabulary

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

One-hot encoding of the word "am"

0.0	1.0	0.0	0.0	0.0	0.0

Example: one-hot encoding of our output vocabulary

示例：输出词汇表的独热编码

Following this recap, let's discuss the model's loss function – the metric we are optimizing during the training phase to lead up to a trained and hopefully amazingly accurate model.

在此回顾之后，让我们讨论模型的损失函数 - 我们在训练阶段优化的指标，以导致一个经过训练且有望令人惊讶的准确模型。

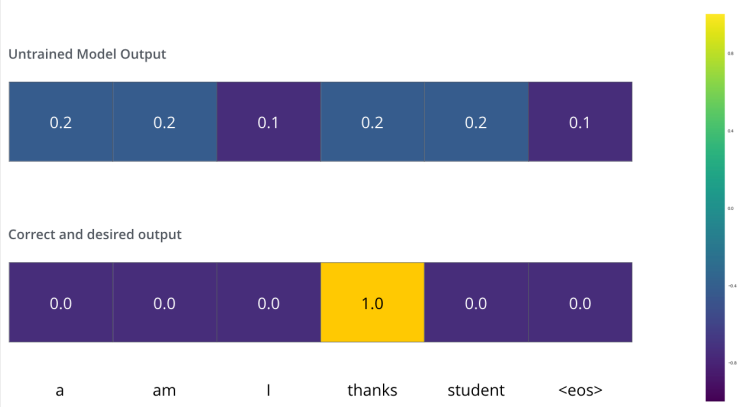
The Loss Function 损失函数

Say we are training our model. Say it's our first step in the training phase, and we're training it on a simple example – translating "merci" into "thanks".

假设我们正在训练我们的模型。假设这是我们在培训阶段的第一步，我们正在通过一个简单的例子来训练它 - 将"怜悯"翻译成"谢谢"。

What this means, is that we want the output to be a probability distribution indicating the word "thanks". But since this model is not yet trained, that's unlikely to happen just yet.

这意味着，我们希望输出是一个概率分布，指示单词"谢谢"。但由于这个模型还没有经过训练，所以现在不太可能发生。



Since the model's parameters (weights) are all initialized randomly, the (untrained) model produces a probability distribution with arbitrary values for each cell/word. We can compare it with the actual output, then tweak all the model's weights using backpropagation to make the output closer to the desired output.

由于模型的参数（权重）都是随机初始化的，因此（未经训练的）模型为每个单元格/单词生成具有任意值的概率分布。我们可以将其与实际输出进行比较，然后使用反向传播调整模型的所有权重，以使输出更接近所需的输出。

How do you compare two probability distributions? We simply subtract one from the other. For more details, look at cross-entropy (https://colah.github.io/posts/2015-09-Visual-Information/) and Kullback–Leibler divergence (https://www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained).

您如何比较两个概率分布？我们只是从另一个中减去一个。有关更多详细信息，请查看交叉熵和Kullback-Leibler散度。

But note that this is an oversimplified example. More realistically, we'll use a sentence longer than one word. For example – input: "je suis étudiant" and expected output: "i am a student". What this really means, is that we want our model to successively output probability distributions where:

但请注意，这是一个过于简化的示例。更现实地说，我们将使用一个比一个单词更长的句子。例如 – 输入："je suis étudiant"和预期输出："我是学生"。这真正意味着，我们希望我们的模型连续输出概率分布，其中：

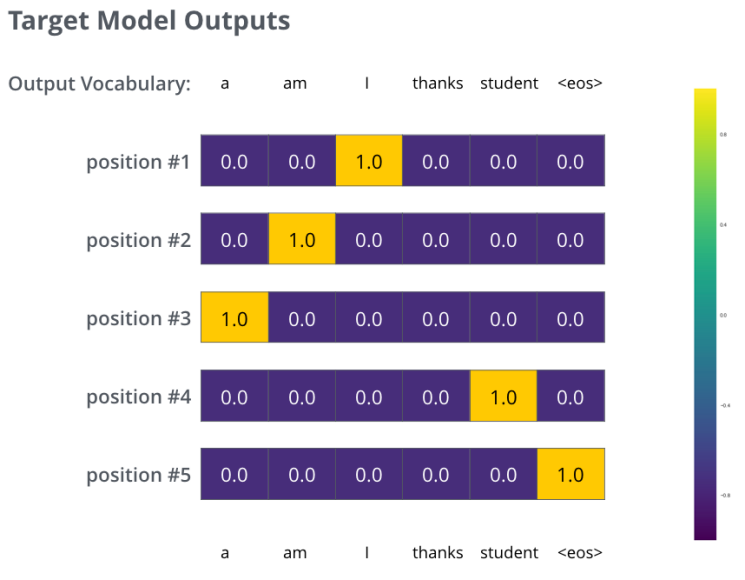
- Each probability distribution is represented by a vector of width vocab_size (6 in our toy example, but more realistically a number like 30,000 or 50,000)

  每个概率分布都由宽度vocab_size向量表示（在我们的玩具示例中为 6，但更现实的数字为 30，000 或 50，000）
- The first probability distribution has the highest probability at the cell associated with the word "i"

  第一个概率分布在与单词"i"关联的单元格处具有最高的概率
- The second probability distribution has the highest probability at the cell associated with the word "am"

  第二个概率分布在与单词"am"相关的单元格处具有最高的概率
- And so on, until the fifth output distribution indicates '`<end of sentence>`' symbol, which also has a cell associated with it from the 10,000 element vocabulary.

  依此类推，直到第五个输出分布指示 '' `<end of sentence>` 符号，该符号还有一个来自 10，000 个元素词汇表的单元格与之关联。



The targeted probability distributions we'll train our model against in the training example for one sample sentence.

我们将在一个样本句子的训练示例中训练模型的目标概率分布。

After training the model for enough time on a large enough dataset, we would hope the produced probability distributions would look like this:

在足够大的数据集上训练模型足够长的时间后，我们希望生成的概率分布如下所示：

Hopefully upon training, the model would output the right translation we expect. Of course it's no real indication if this phrase was part of the training dataset (see: cross validation (https://www.youtube.com/watch?v=TIgfjmp-4BA)). Notice that every position gets a little bit of probability even if it's unlikely to be the output of that time step -- that's a very useful property of softmax which helps the training process.

希望经过训练，模型将输出我们期望的正确翻译。当然，这并不能真正表明这个短语是否是训练数据集的一部分（参见：交叉验证）。请注意，每个位置都有一点概率，即使它不太可能是该时间步的输出 - 这是softmax的一个非常有用的属性，有助于训练过程。

Now, because the model produces the outputs one at a time, we can assume that the model is selecting the word with the highest probability from that probability distribution and throwing away the rest. That's one way to do it (called greedy decoding). Another way to do it would be to hold on to, say, the top two words (say, 'I' and 'a' for example), then in the next step, run the model twice: once assuming the first output position was the word 'I', and another time assuming the first output position was the word 'a', and whichever version produced less error considering both positions #1 and #2 is kept. We repeat this for positions #2 and #3…etc. This method is called "beam search", where in our example, beam_size was two (meaning that at all times, two partial hypotheses (unfinished translations) are kept in memory), and top_beams is also two (meaning we'll return two translations). These are both hyperparameters that you can experiment with.

现在，由于模型一次生成一个输出，因此我们可以假设模型正在从该概率分布中选择概率最高的单词，并丢弃其余的单词。这是一种方法（称为贪婪解码）。另一种方法是保留，比如说，前两个单词（例如，"I"和"a"），然后在下一步中，运行模型两次：一次假设第一个输出位置是单词"I"，另一次假设第一个输出位置是单词"a"，考虑到位置 #1 和 #2，哪个版本产生的错误较小，则保留。我们对位置 #2 和 #3 重复此操作...等。这种方法称为"波束搜索"，在我们的示例中，beam_size是两个（意味着始终将两个部分假设（未完成的翻译）保存在内存中），top_beams也是两个（意味着我们将返回两个翻译）。这些都是您可以试验的超参数。

Go Forth And Transform

勇往直前，转型

I hope you've found this a useful place to start to break the ice with the major concepts of the Transformer. If you want to go deeper, I'd suggest these next steps:

我希望你已经发现这是一个有用的地方，开始打破变形金刚的主要概念。如果你想更深入，我建议这些后续步骤：

- Read the Attention Is All You Need (https://arxiv.org/abs/1706.03762) paper, the Transformer blog post (Transformer: A Novel Neural Network Architecture for Language Understanding (https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html)), and the Tensor2Tensor

announcement (https://ai.googleblog.com/2017/06/accelerating-deep-learning-research.html).

阅读《关注是你需要的一切》论文、Transformer 博客文章（《Transformer： A Novel Neural Network Architecture for Language Understanding》）和 Tensor2Tensor 公告。

- Watch Łukasz Kaiser's talk (https://www.youtube.com/watch?v=rBCqOTEfxvg) walking through the model and its details

  观看Łukasz Kaiser的演讲，了解模型及其细节

- Play with the Jupyter Notebook provided as part of the Tensor2Tensor repo (https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb)

  使用 Jupyter Notebook 作为 Tensor2Tensor 存储库的一部分提供

- Explore the Tensor2Tensor repo (https://github.com/tensorflow/tensor2tensor).

  探索 Tensor2Tensor 存储库。

Follow-up works: 后续工作：

- Depthwise Separable Convolutions for Neural Machine Translation

  用于神经机器翻译的深度可分离卷积 (https://arxiv.org/abs/1706.03059)

- One Model To Learn Them All

  一个模型来学习它们 (https://arxiv.org/abs/1706.05137)

- Discrete Autoencoders for Sequence Models

  用于序列模型的离散自动编码器 (https://arxiv.org/abs/1801.09797)

- Generating Wikipedia by Summarizing Long Sequences

  通过总结长序列生成维基百科 (https://arxiv.org/abs/1801.10198)

- Image Transformer 图像转换器 (https://arxiv.org/abs/1802.05751)
- Training Tips for the Transformer Model

  变压器模型的训练技巧 (https://arxiv.org/abs/1804.00247)

- Self-Attention with Relative Position Representations

  具有相对位置表示的自我注意 (https://arxiv.org/abs/1803.02155)

- Fast Decoding in Sequence Models using Discrete Latent Variables

  使用离散潜在变量在序列模型中快速解码 (https://arxiv.org/abs/1803.03382)

- Adafactor: Adaptive Learning Rates with Sublinear Memory Cost

  Adafactor：具有次线性内存成本的自适应学习率 (https://arxiv.org/abs/1804.04235)

Acknowledgements 确认

Thanks to Illia Polosukhin (https://twitter.com/ilblackdragon), Jakob Uszkoreit (http://jakob.uszkoreit.net/), Llion Jones (https://www.linkedin.com/in/llion-jones-9ab3064b), Lukasz Kaiser (https://ai.google/research/people/LukaszKaiser), Niki Parmar (https://twitter.com/nikiparmar09), and Noam Shazeer (https://dblp.org/pers/hd/s/Shazeer:Noam) for providing feedback on earlier versions of this post.

感谢Illia Polosukhin，Jakob Uszkoreit，Llion Jones，Lukasz Kaiser，Niki Parmar和Noam Shazeer对本文早期版本的反馈。

Please hit me up on Twitter (https://twitter.com/JayAlammar) for any corrections or feedback.

请在推特上与我联系以获取任何更正或反馈。

Written on June 27, 2018

写于2018年6月27日