

# Transformer

## 参考文献:

1. Attention Is All You Need: <https://arxiv.org/abs/1706.03762>
2. Attention? Attention!: <https://lilianweng.github.io/posts/2018-06-24-attention/>

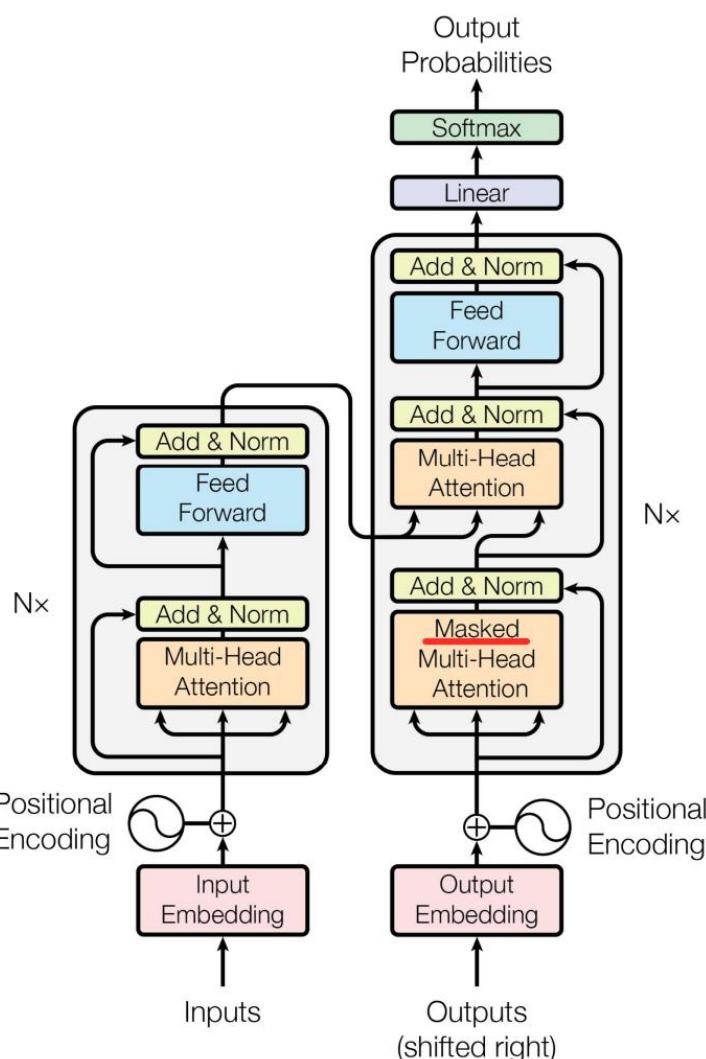
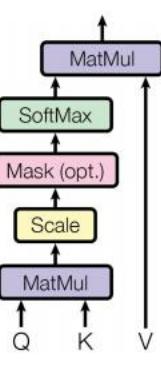


Figure 1: The Transformer - model architecture.

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

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O \\ \text{where } \text{head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned}$$

Scaled Dot-Product Attention



Multi-Head Attention

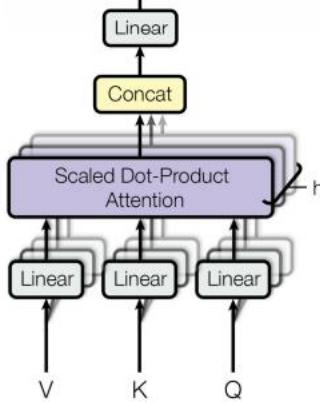


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

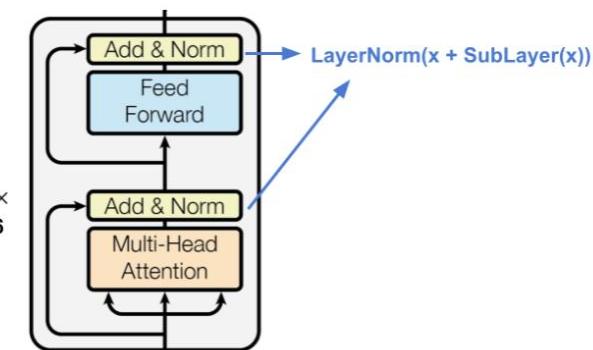


Fig. 15. The transformer's encoder. (Image source: Vaswani, et al., 2017)

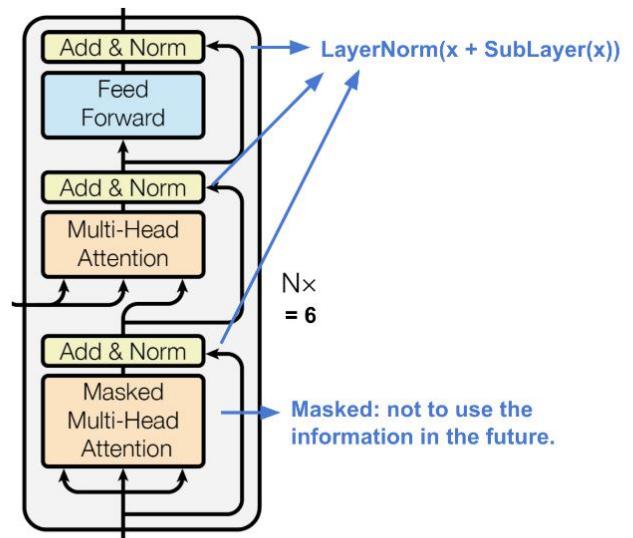


Fig. 16. The transformer's decoder. (Image source: Vaswani, et al., 2017)

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

where  $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

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

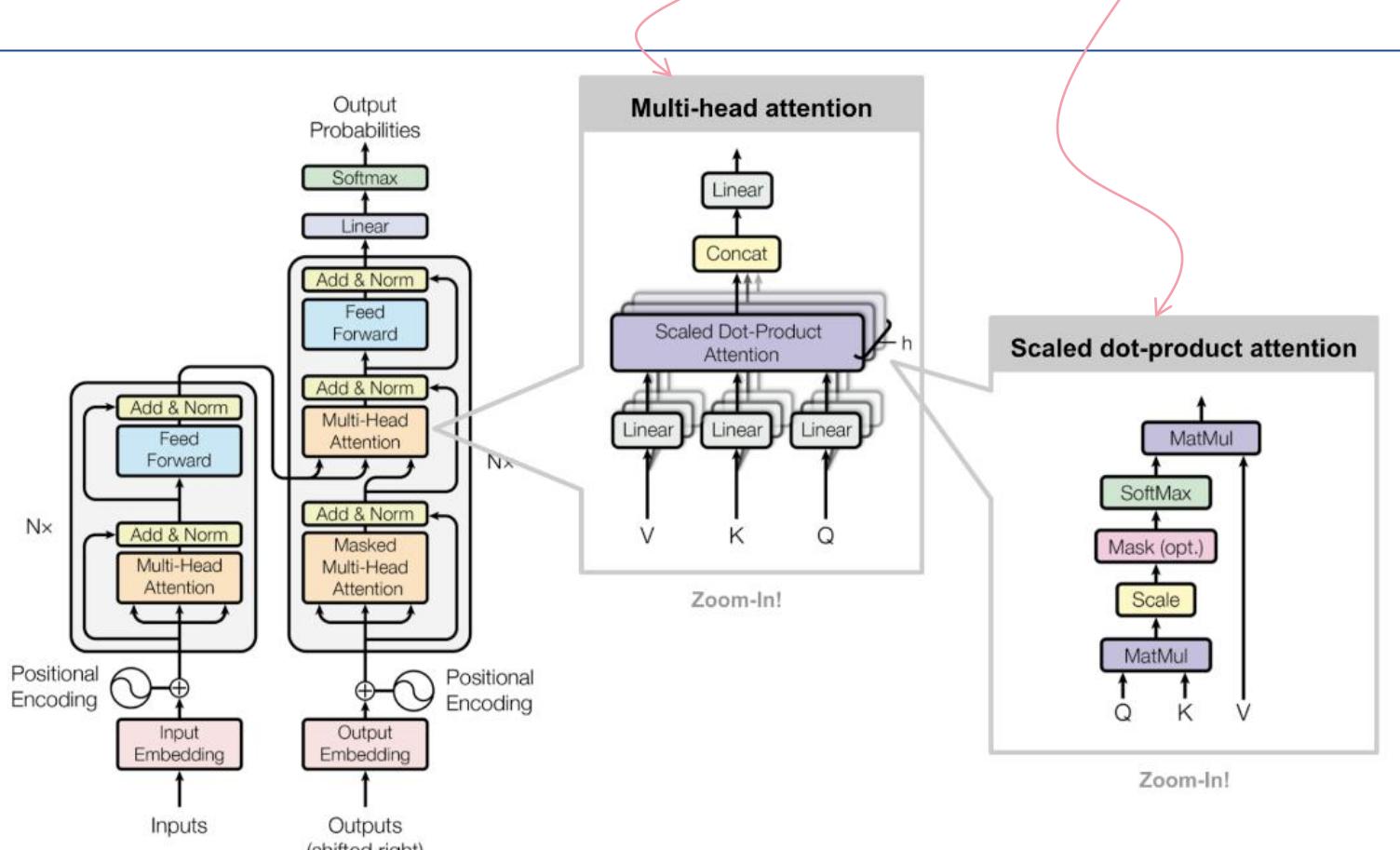


Fig. 17. The full model architecture of the transformer. (Image source: Fig 1 & 2 in Vaswani, et al., 2017.)

## 参考文献:

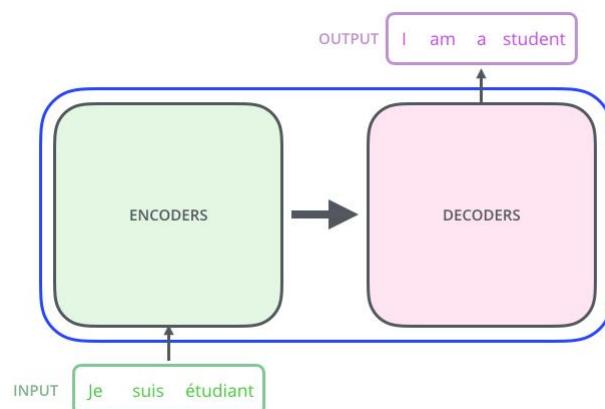
1. Attention Is All You Need: <https://arxiv.org/abs/1706.03762>
2. The Illustrated Transformer: <https://jalammar.github.io/illustrated-transformer/>

## 1 A High-Level Look

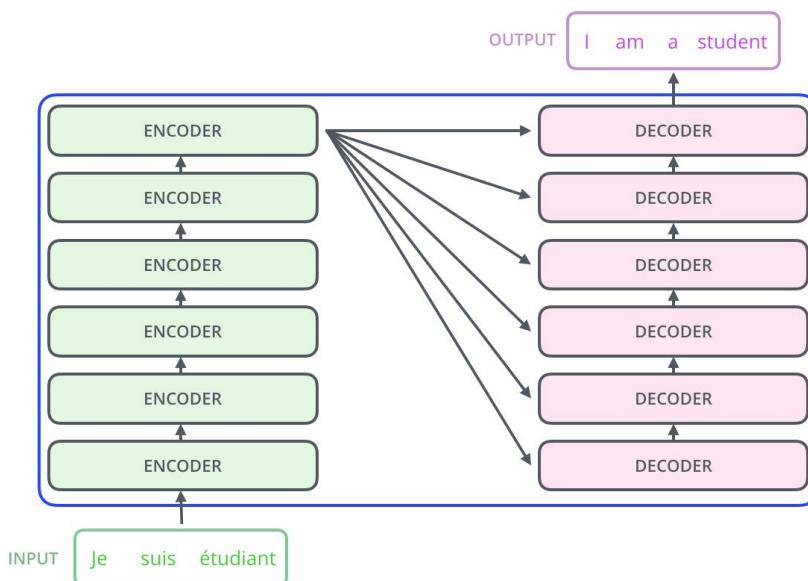
Let's begin by looking at the model as a single black box. In a machine translation application, it would take a sentence in one language, and output its translation in another.



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



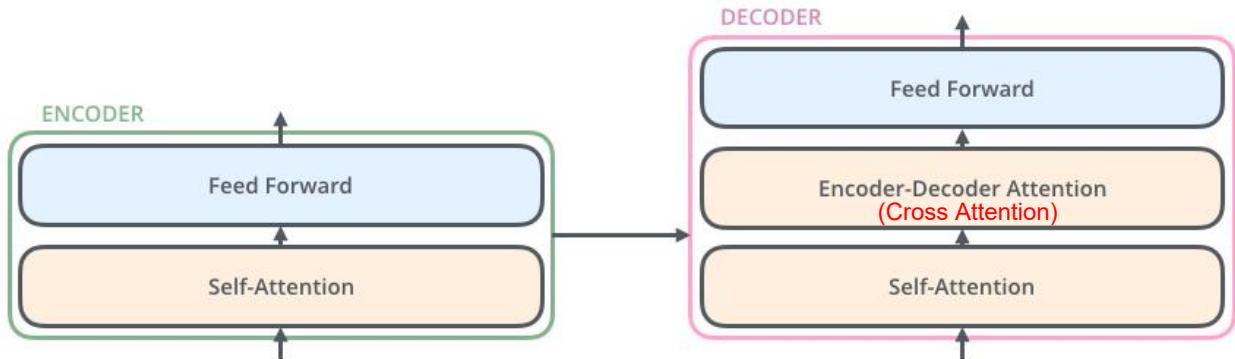
The encoding component is a stack of encoders (the paper **stacks six** of them on top of each other – there's nothing magical about the number six, one can definitely experiment with other arrangements). The decoding component is a stack of decoders of the same number.



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

- 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.
- 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.

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](#)).

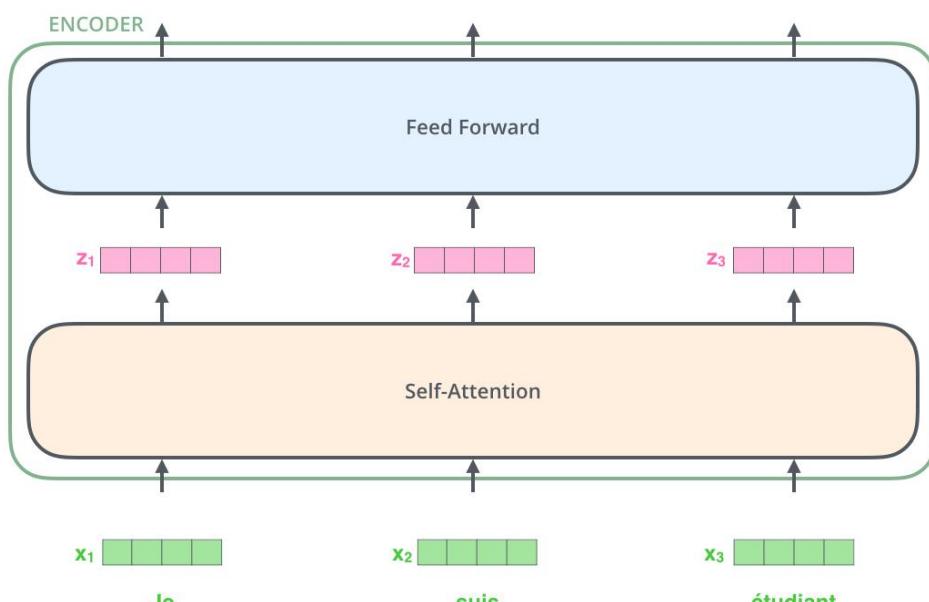


## 2 Bringing The Tensors Into The Picture

As is the case in NLP applications in general, we begin by turning each input word into a vector using an [embedding algorithm](#).

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.

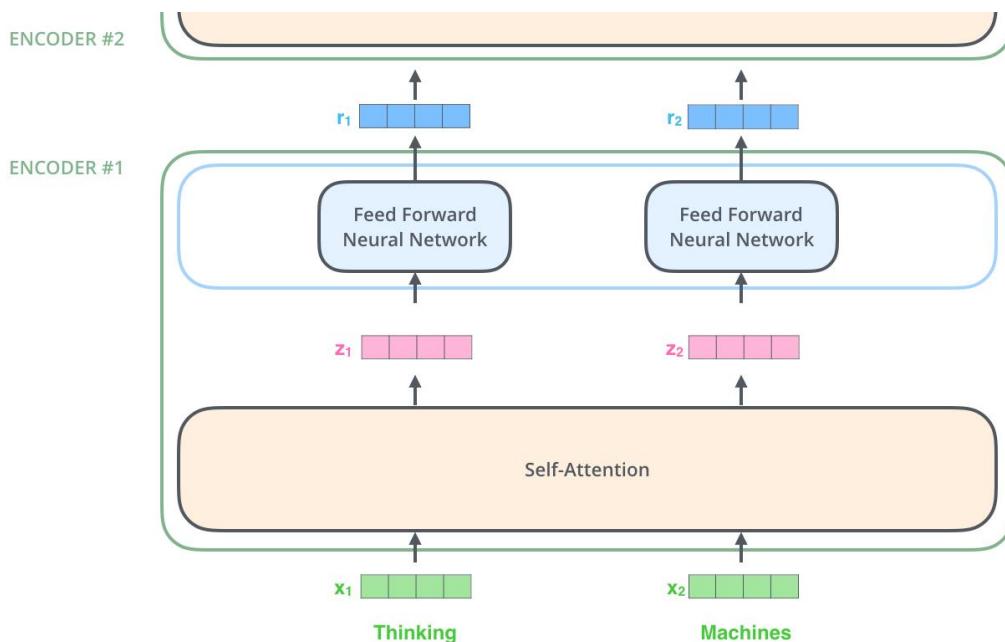
After embedding the words in our input sequence, each of them flows through each of the two layers of the encoder.



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.

## 3 Now We're Encoding!

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.



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.

## 4 Self-Attention at a High Level

(略)

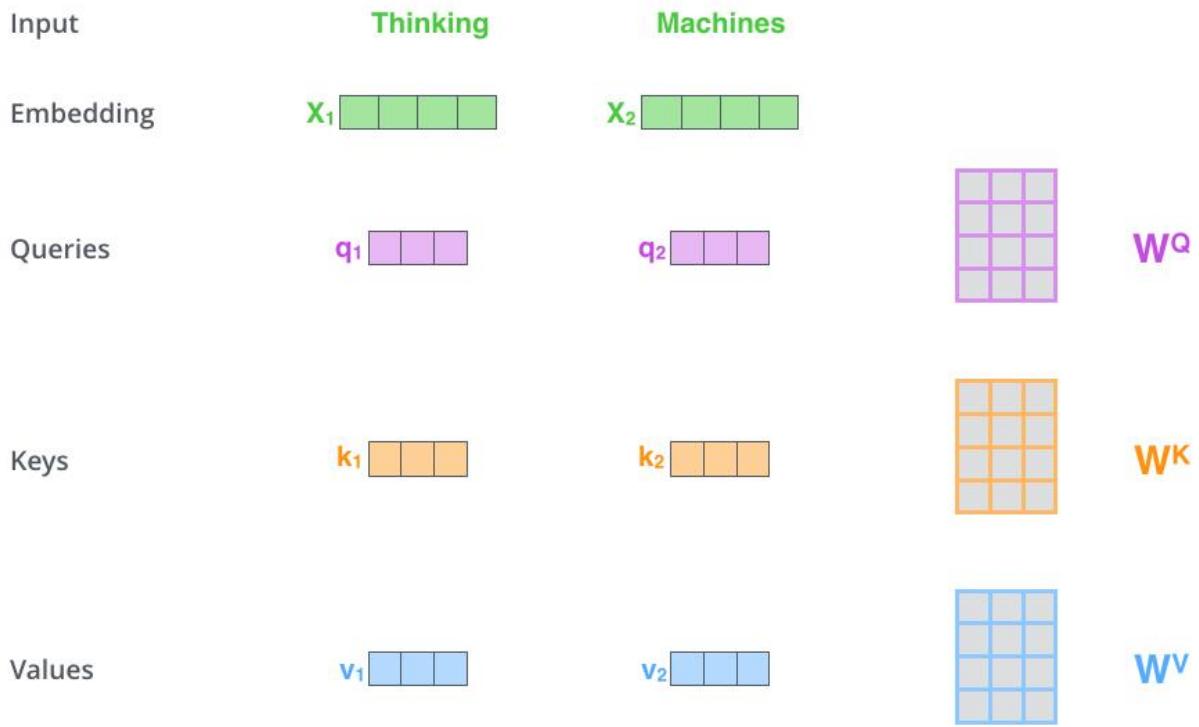
## 5 Self-Attention in Detail

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

### ① The first step

- 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.

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.



Multiplying  $x_1$  by the  $WQ$  weight matrix produces  $q_1$ , 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.

## ② 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.
- 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  $q_1$  and  $k_1$ . The second score would be the dot product of  $q_1$  and  $k_2$ .

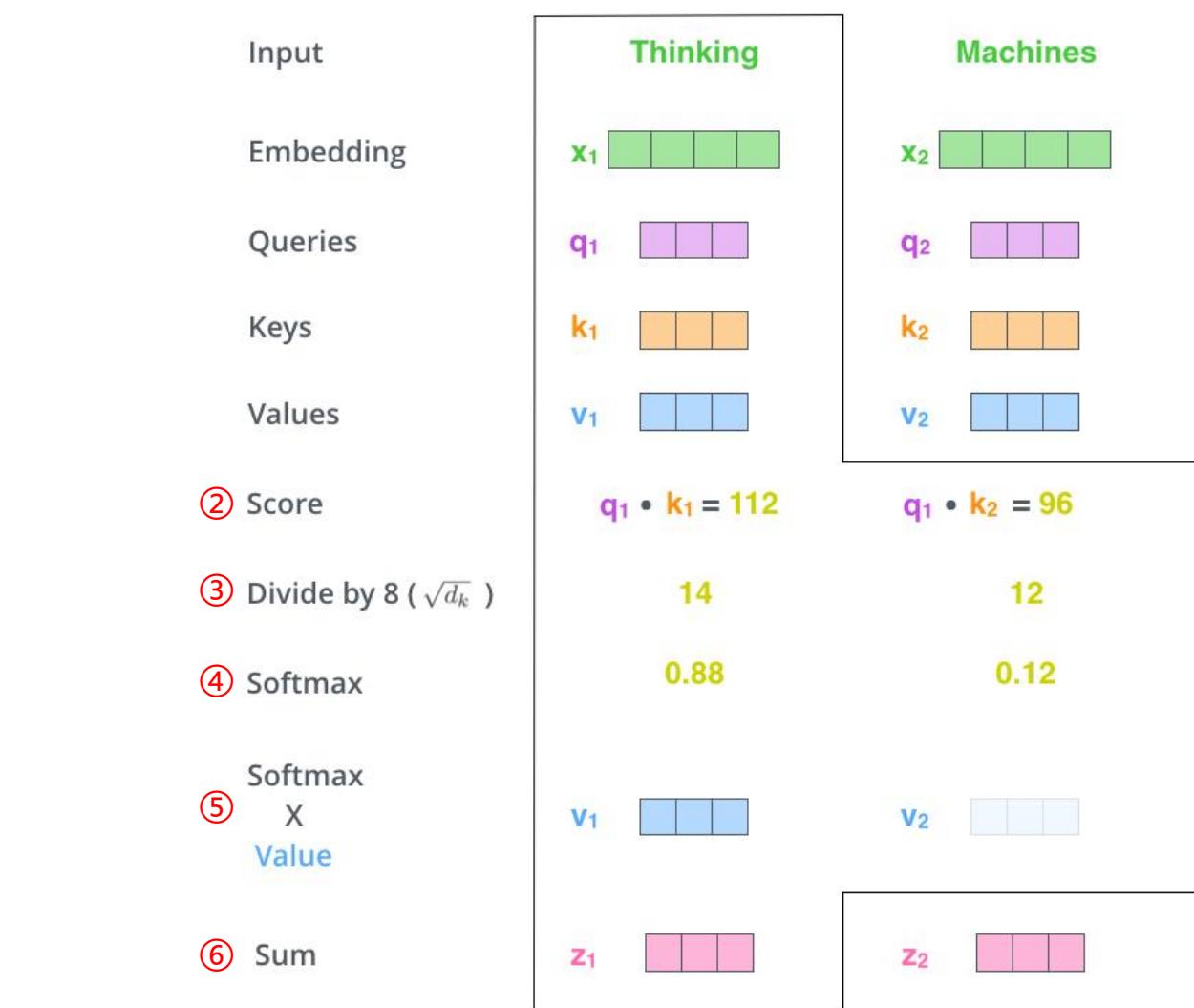
Input	<b>Thinking</b>	<b>Machines</b>	
Embedding	$x_1$	$x_2$	
Queries	$q_1$	$q_2$	
Keys	$k_1$	$k_2$	
Values	$v_1$	$v_2$	
Score	$q_1 \cdot k_1 = 112$		$q_1 \cdot k_2 = 96$

③④ 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.

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.

⑤ 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).

⑥ 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).



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.

## 6 Matrix Calculation of Self-Attention

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**).

$$\begin{array}{ccc} \text{X} & \quad \quad \quad \text{W}^Q & \quad \quad \quad \text{Q} \\ \begin{matrix} \boxed{\text{green}} & \boxed{\text{green}} & \boxed{\text{green}} \\ \boxed{\text{green}} & \boxed{\text{green}} & \boxed{\text{green}} \end{matrix} & \times & \begin{matrix} \boxed{\text{purple}} & \boxed{\text{purple}} & \dots \\ \boxed{\text{purple}} & \boxed{\text{purple}} & \dots \end{matrix} & = & \begin{matrix} \boxed{\text{purple}} & \boxed{\text{purple}} & \dots \\ \boxed{\text{purple}} & \boxed{\text{purple}} & \dots \end{matrix} \end{array}$$

$$\begin{array}{ccc} \text{X} & \quad \quad \quad \text{W}^K & \quad \quad \quad \text{K} \\ \begin{matrix} \boxed{\text{green}} & \boxed{\text{green}} & \boxed{\text{green}} \\ \boxed{\text{green}} & \boxed{\text{green}} & \boxed{\text{green}} \end{matrix} & \times & \begin{matrix} \boxed{\text{orange}} & \boxed{\text{orange}} & \dots \\ \boxed{\text{orange}} & \boxed{\text{orange}} & \dots \end{matrix} & = & \begin{matrix} \boxed{\text{orange}} & \boxed{\text{orange}} & \dots \\ \boxed{\text{orange}} & \boxed{\text{orange}} & \dots \end{matrix} \end{array}$$

$$\begin{array}{ccc} \text{X} & \quad \quad \quad \text{W}^V & \quad \quad \quad \text{V} \\ \begin{matrix} \boxed{\text{green}} & \boxed{\text{green}} & \boxed{\text{green}} \\ \boxed{\text{green}} & \boxed{\text{green}} & \boxed{\text{green}} \end{matrix} & \times & \begin{matrix} \boxed{\text{blue}} & \boxed{\text{blue}} & \dots \\ \boxed{\text{blue}} & \boxed{\text{blue}} & \dots \end{matrix} & = & \begin{matrix} \boxed{\text{blue}} & \boxed{\text{blue}} & \dots \\ \boxed{\text{blue}} & \boxed{\text{blue}} & \dots \end{matrix} \end{array}$$

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)

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.

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

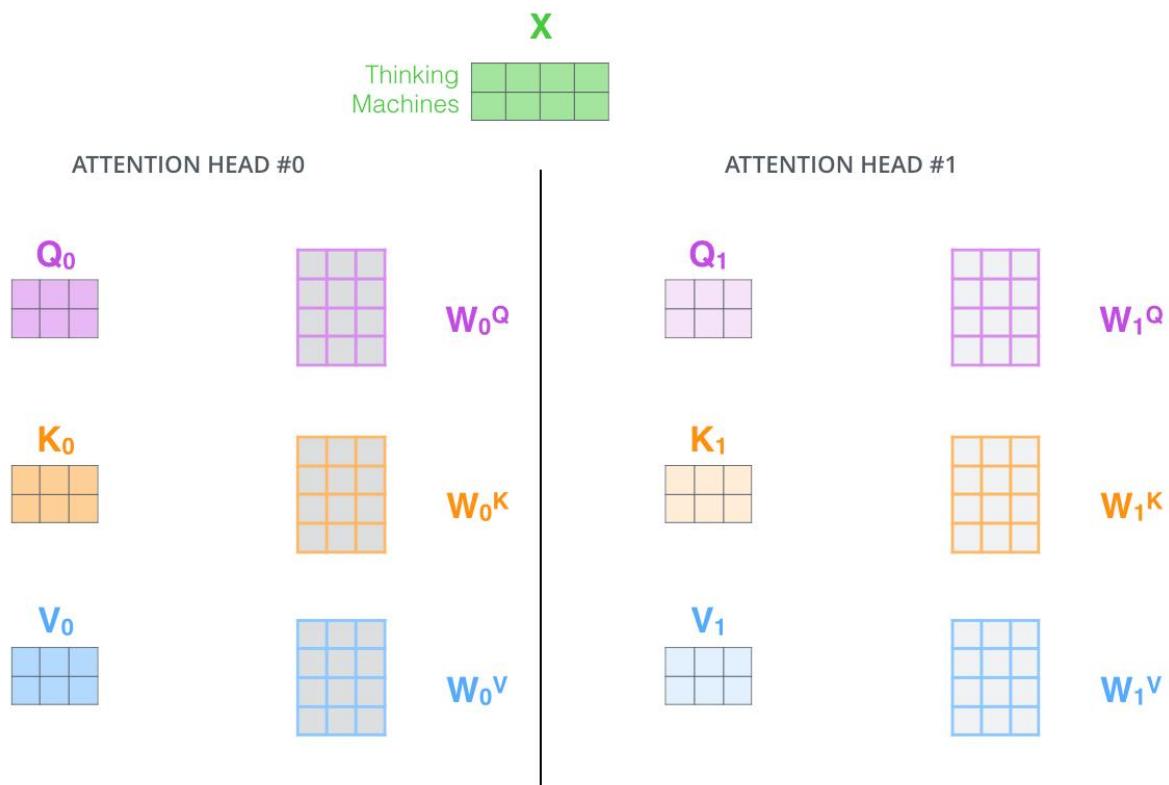
Diagram illustrating the final calculation:

$$\begin{array}{c} \text{Q} \quad \quad \quad \text{K}^T \\ \begin{matrix} \boxed{\text{purple}} & \boxed{\text{purple}} & \dots \\ \boxed{\text{purple}} & \boxed{\text{purple}} & \dots \end{matrix} \quad \quad \quad \begin{matrix} \boxed{\text{orange}} & \boxed{\text{orange}} & \dots \\ \boxed{\text{orange}} & \boxed{\text{orange}} & \dots \end{matrix} \\ \times \\ \sqrt{d_k} \\ \hline \text{V} \\ \begin{matrix} \boxed{\text{blue}} & \boxed{\text{blue}} & \dots \\ \boxed{\text{blue}} & \boxed{\text{blue}} & \dots \end{matrix} \\ = \\ \text{Z} \\ \begin{matrix} \boxed{\text{pink}} & \boxed{\text{pink}} & \dots \\ \boxed{\text{pink}} & \boxed{\text{pink}} & \dots \end{matrix} \end{array}$$

## 7 The Beast With Many Heads

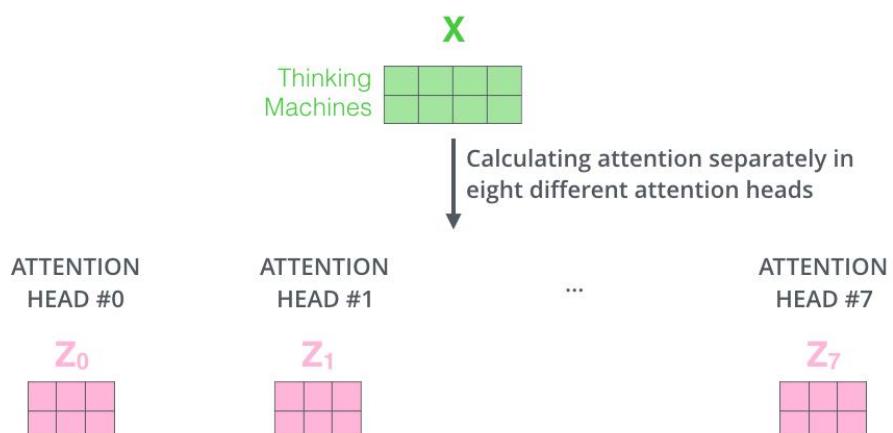
The paper further refined the self-attention layer by adding a mechanism called “**multi-headed**” attention. This improves the performance of the attention layer in two ways:

- It expands the model’s ability to focus on different positions. Yes, in the example above,  $z_1$  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.
- 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.



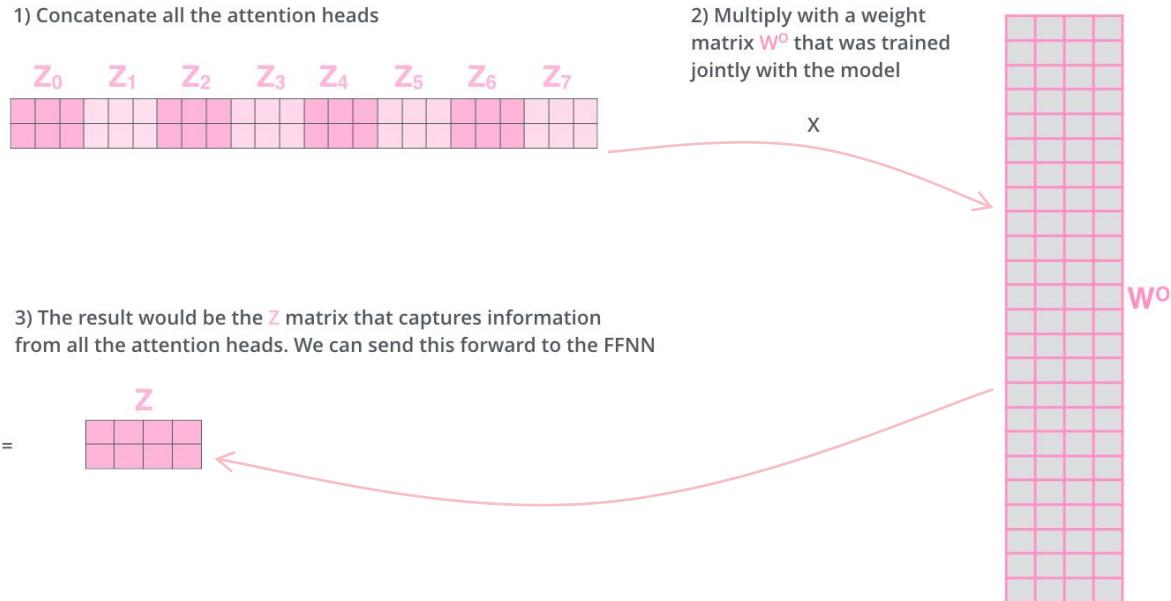
With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices. As we did before, we multiply  $X$  by the  $WQ/WK/WV$  matrices to produce Q/K/V matrices.

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.



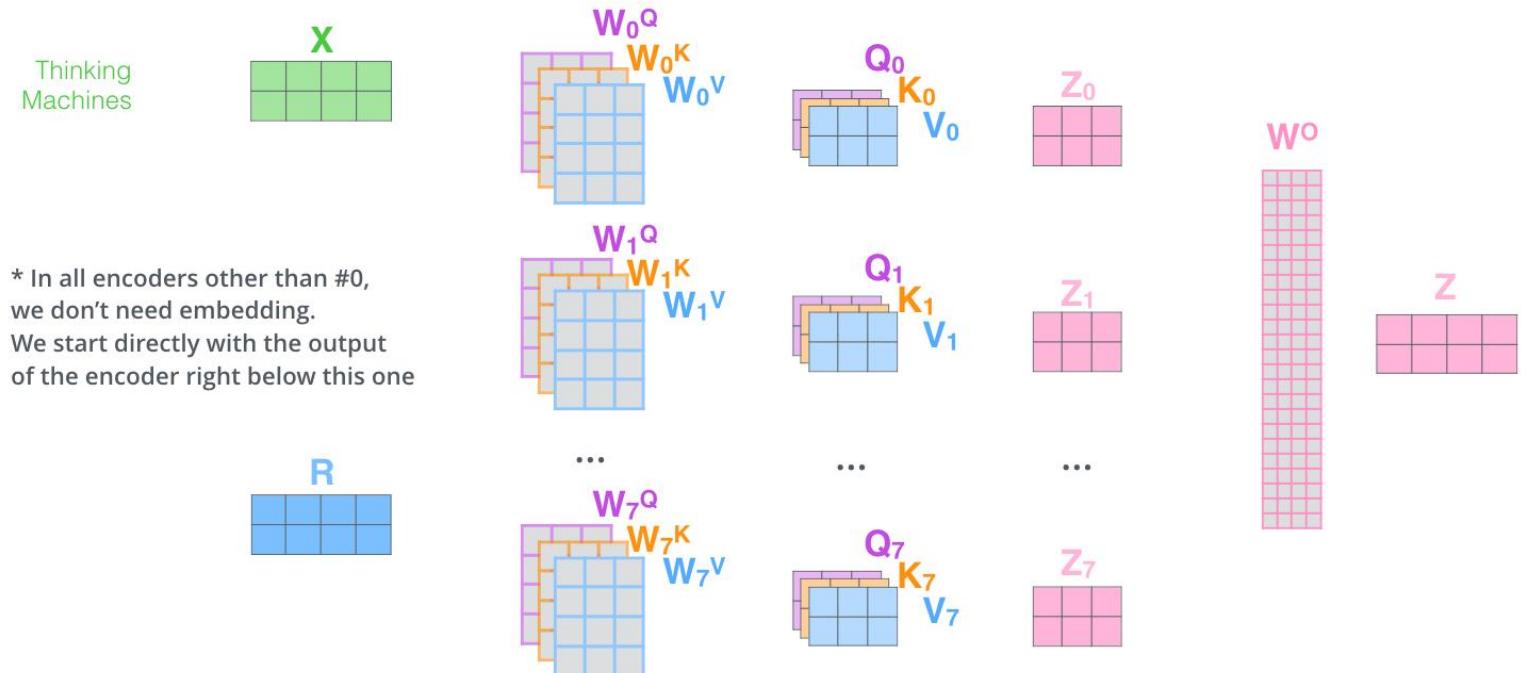
This leaves us with a bit of a challenge. The feed-forward layer is not expecting eight matrices – it's expecting a single matrix (a vector for each word). So we need a way to **condense these eight down into a single matrix**.

How do we do that? We **concat the matrices then multiply them by an additional weights matrix  $W^O$** .

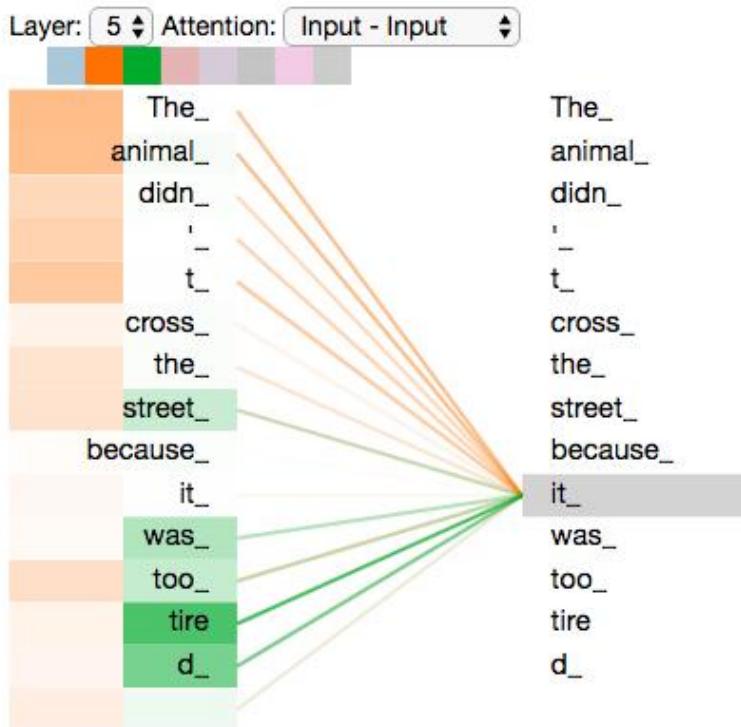


That's pretty much **all** there is to multi-headed self-attention. It's quite a handful of matrices, I realize. Let me try to put them **all** in one visual so we can look at them in one place.

- 1) This is our input sentence\*
- 2) We embed each word\*
- 3) Split into 8 heads. We multiply  $X$  or  $R$  with weight matrices
- 4) Calculate attention using the resulting  $Q/K/V$  matrices
- 5) Concatenate the resulting  $Z$  matrices, then multiply with weight matrix  $W^O$  to produce the output of the layer

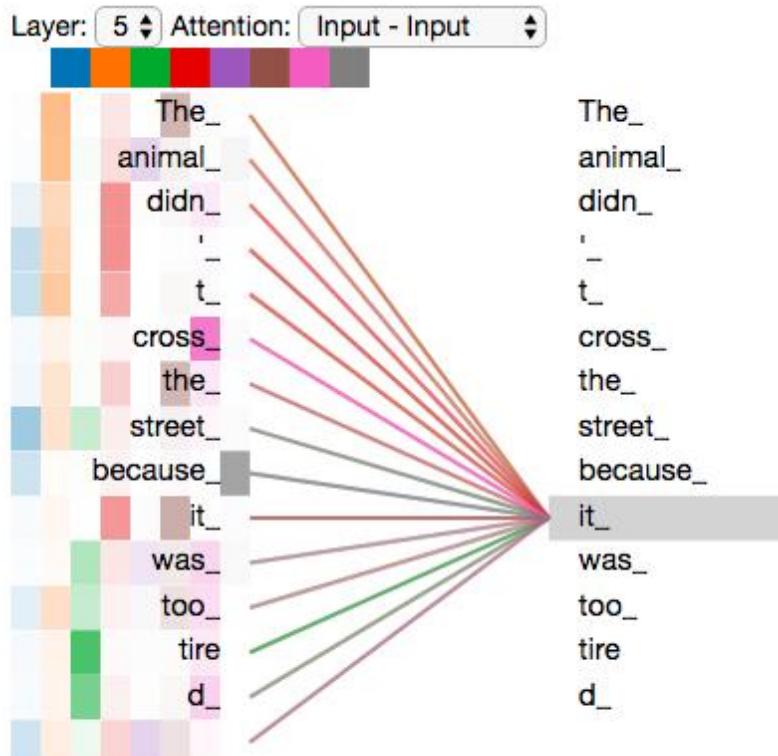


Now that we have touched upon attention heads, let's revisit our example from before to see where the different attention heads are focusing as we encode the word "it" in our example sentence:



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".

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



## 8 Representing The Order of The Sequence Using Positional Encoding

In this work, we use sine and cosine functions of different frequencies:

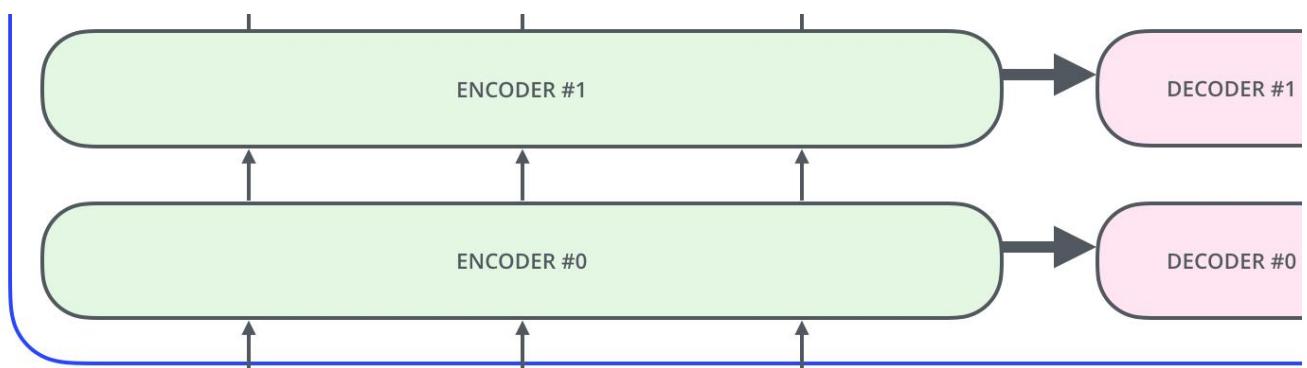
$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{\text{model}}})$$

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

where pos is the position and i is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from  $2\pi$  to  $10000 \cdot 2\pi$ . We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset  $k$ ,  $PE_{pos+k}$  can be represented as a linear function of  $PE_{pos}$ .

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.

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.



INPUT	je	suis	étudiant
EMBEDDING WITH TIME SIGNAL	$x_1$ [green green green]	$x_2$ [green green green]	$x_3$ [green green green]
=	=	=	
POSITIONAL ENCODING	$t_1$ [yellow yellow yellow]	$t_2$ [yellow yellow yellow]	$t_3$ [yellow yellow yellow]
+	+	+	
EMBEDDINGS	$x_1$ [green green green]	$x_2$ [green green green]	$x_3$ [green green green]

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.

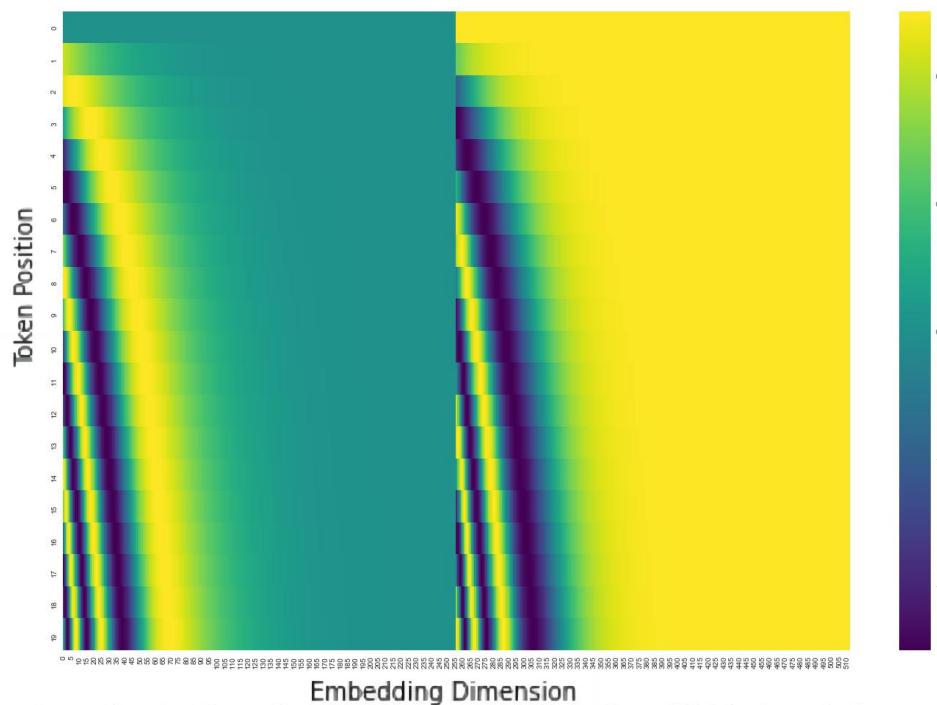
If we assumed the embedding has a dimensionality of 4, the actual positional encodings would look like this:



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

What might this pattern look like?

- 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.



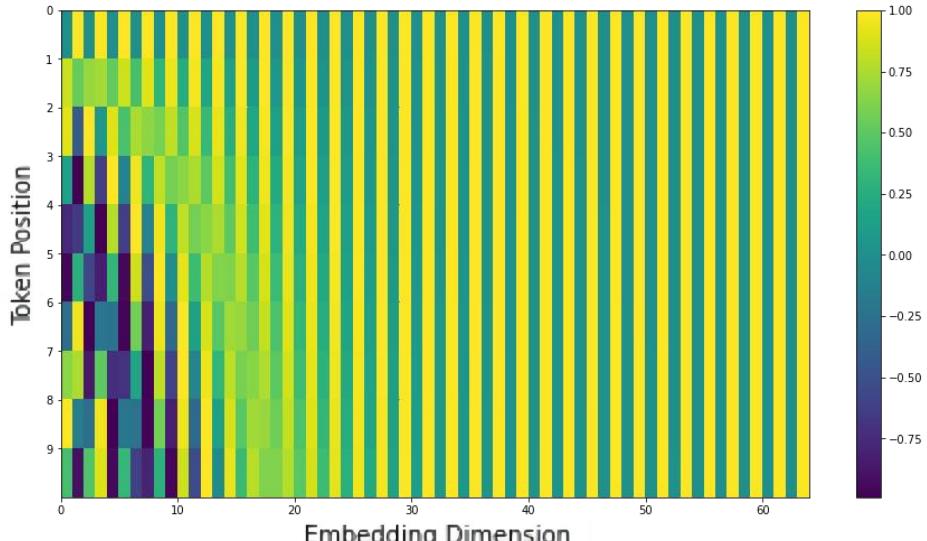
The formula for positional encoding is described in the paper (section 3.5). You can see the code for generating positional encodings in `get_timing_signal_1d()`. This is not the only possible method for positional encoding. It, however, gives the advantage of being able to scale to unseen lengths of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

需要注意的是，官方提供的示例代码（TensorFlow 1.x 版本 中的 `get_timing_signal_1d()` 函数和 TensorFlow 2.x 版本 中的 `call()` 函数）与 Transformer 论文中的方法稍微存在一定差异：Transformer 论文中，`sine` 函数和 `cosine` 函数产生的值交织在一起；而官方提供的代码中，左半部分的值全是由 `sine` 函数产生的，右半部分的值全是由 `cosine` 函数产生的，然后将它们拼接起来。官方代码生成的位置编码值的可视化图如下：  
上  
(引用自：<https://devpress.csdn.net/hefei/63a5663eb878a54545946354.html>)

#### July 2020 Update:

The positional encoding shown above is from the Tranformer2Transformer implementation of the Transformer.

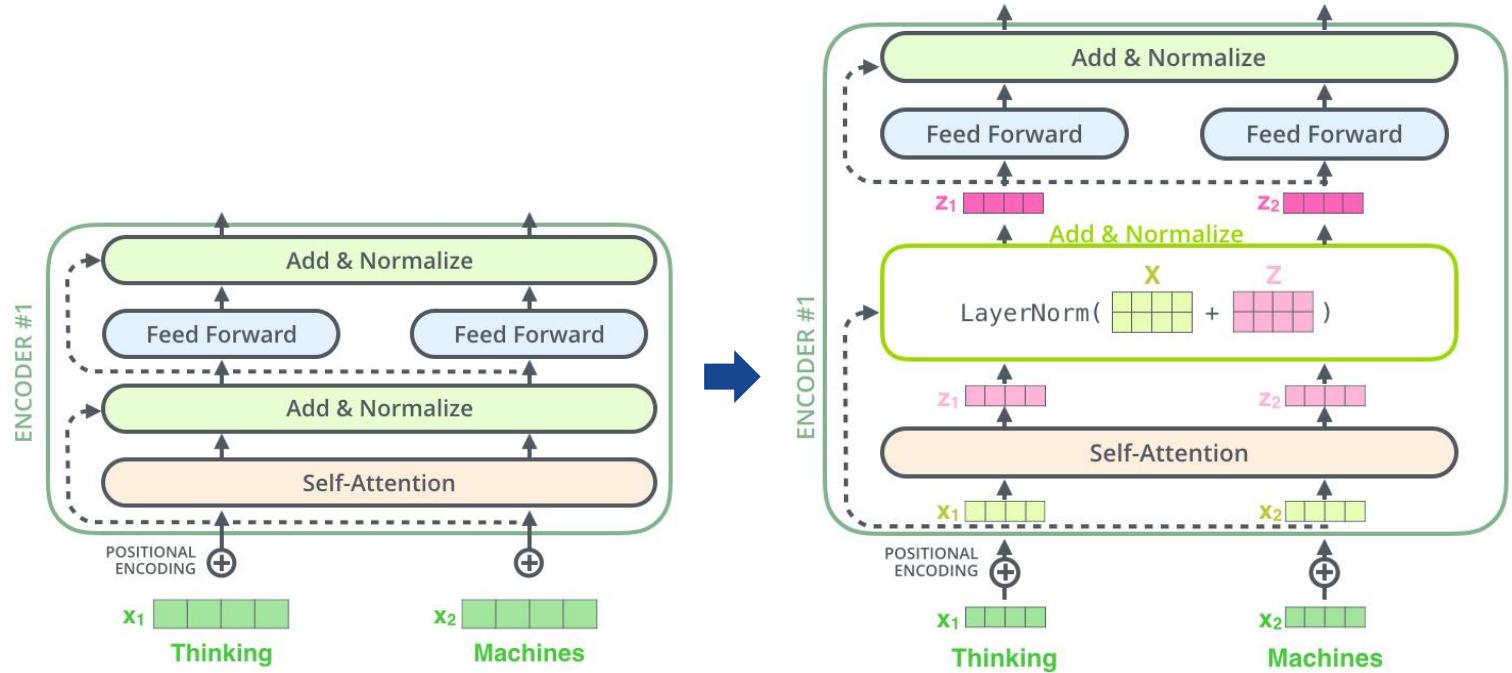
The method shown in the paper is slightly different in that it doesn't directly concatenate, but interweaves the two signals. The following figure shows what that looks like. [Here's the code to generate it](#):



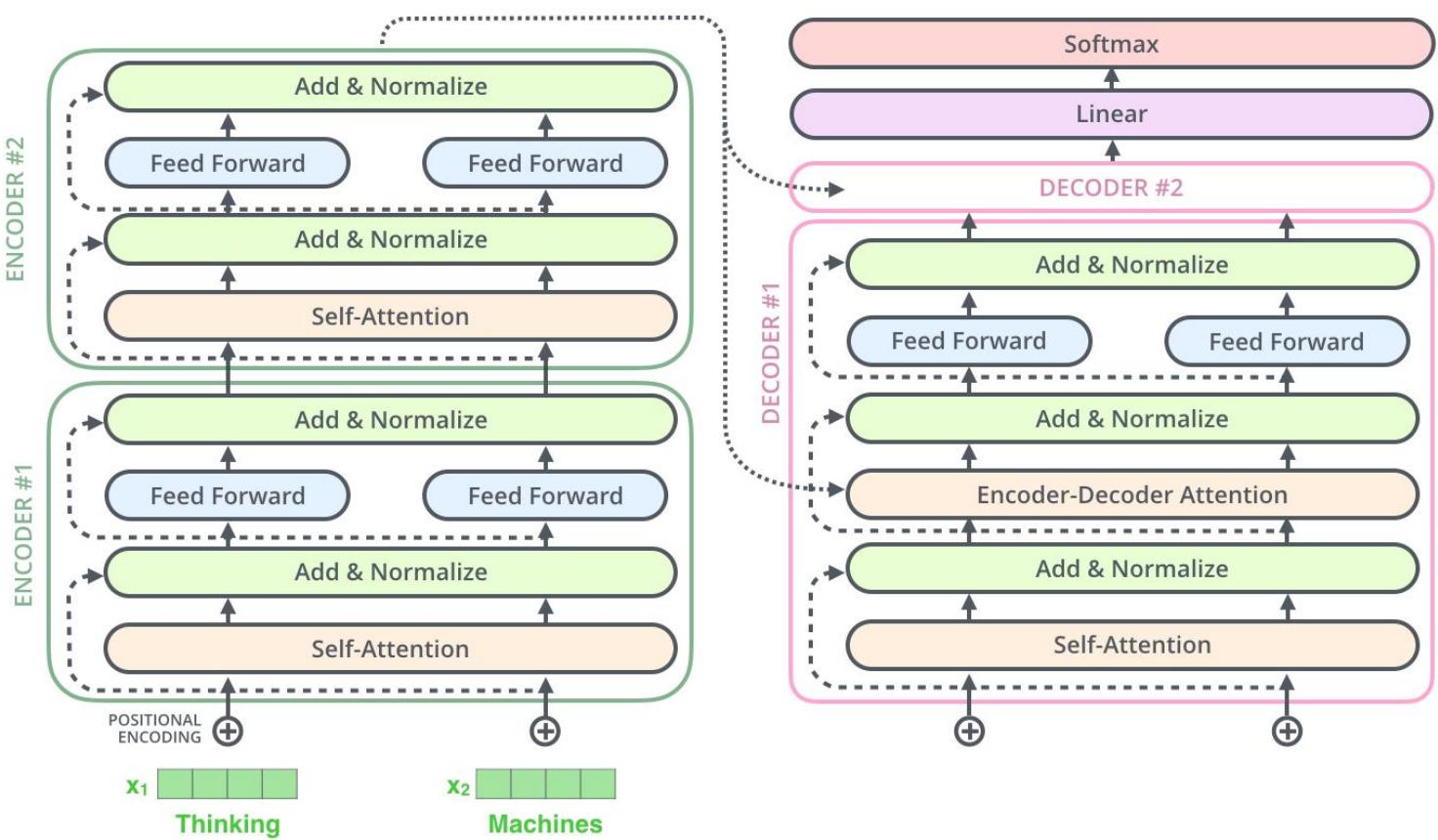
## 9 The Residuals

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](#) step.

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



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:



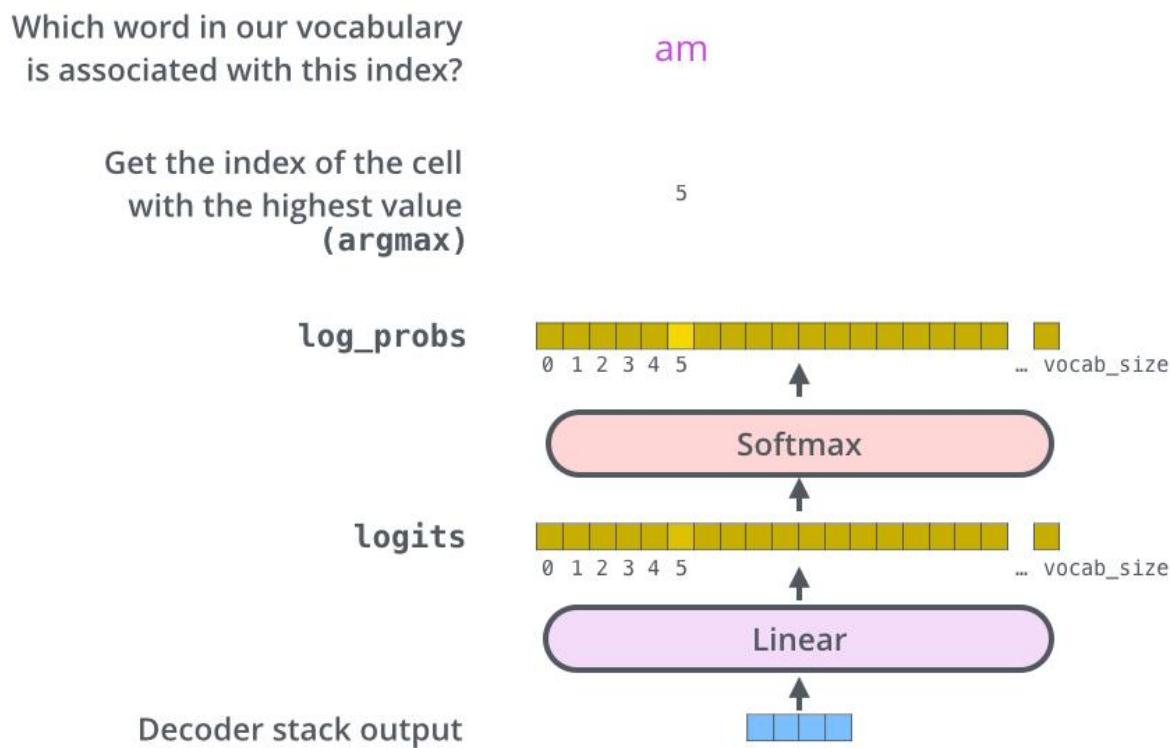
## 10 The Decoder Side

(略, 详见原文)

## 11 The Final Linear and Softmax Layer

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**.

- 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.
- 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.
- 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.



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.

## 12 Recap Of Training

(略, 详见原文)

## 13 The Loss Function

(略, 详见原文)

## 14 Go Forth And Transform

(略, 详见原文)

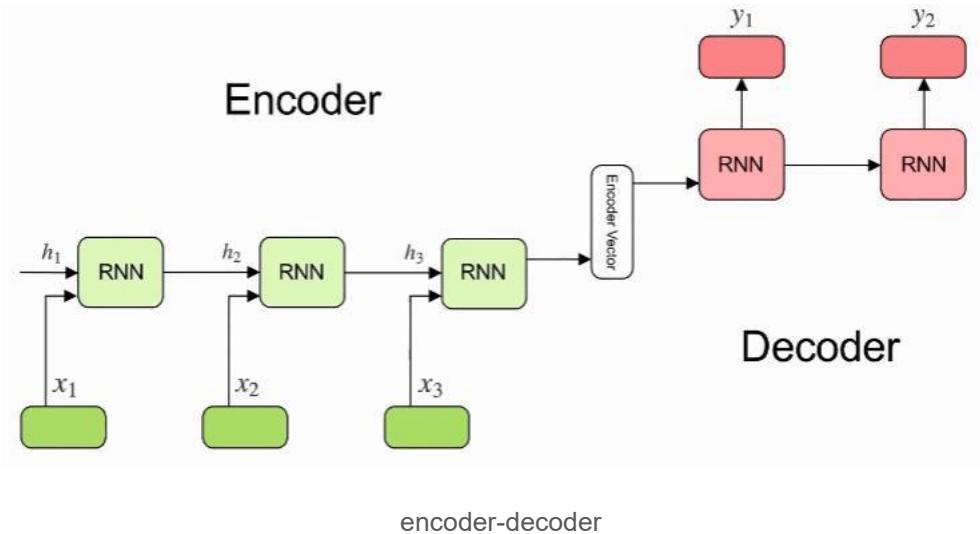
## 参考文献:

1. Attention Is All You Need: <https://arxiv.org/abs/1706.03762>
2. Self-Attention和Transformer: <https://blog.csdn.net/yeen123/article/details/125104680>

# 1 模型的思想

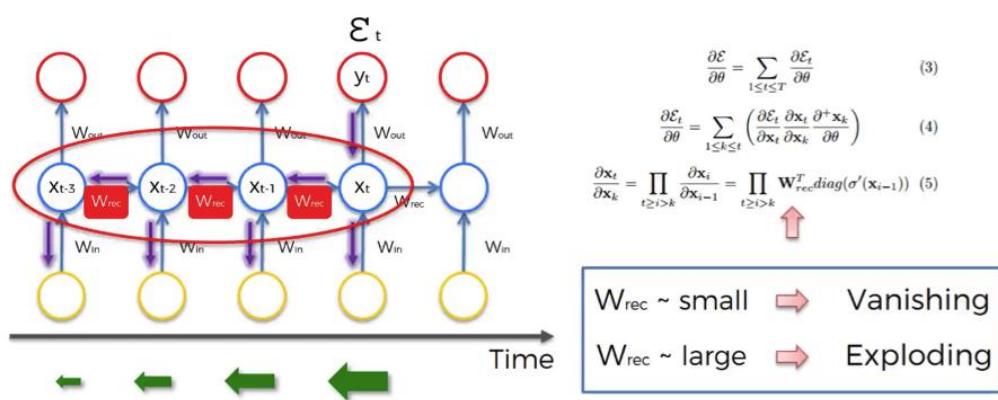
## 1.1 RNN的缺陷

在没有 Transformer 以前，大家做神经机器翻译用的最多的是基于 RNN 的 Encoder-Decoder 模型：



Encoder-Decoder 模型当然很成功，在2018年以前用它是用的很多的。而且也有很强的能力。但是 RNN 天生有缺陷，只要是RNN，**就会有梯度消失问题，核心原因是递归的方式，作用在同一个权值矩阵上，使得如果这个矩阵满足条件的话，其最大的特征值要是小于1的话，就一定会出现梯度消失问题。**后来的 LSTM 和 GRU 也仅仅能缓解这个问题。

## The Vanishing Gradient Problem



Formula Source: Razvan Pascanu et al. (2013)

## 1.2 Transformer 为何优于 RNN

Transformer 抛弃了传统的 CNN 和 RNN，整个网络结构完全是由 Attention机制 组成。作者采用 Attention机制的原因是考虑到 RNN（或者 LSTM，GRU等）的计算限制为是顺序的，也就是说 RNN 相关算法只能从左向右依次计算或者从右向左依次计算，这种机制带来了两个问题：

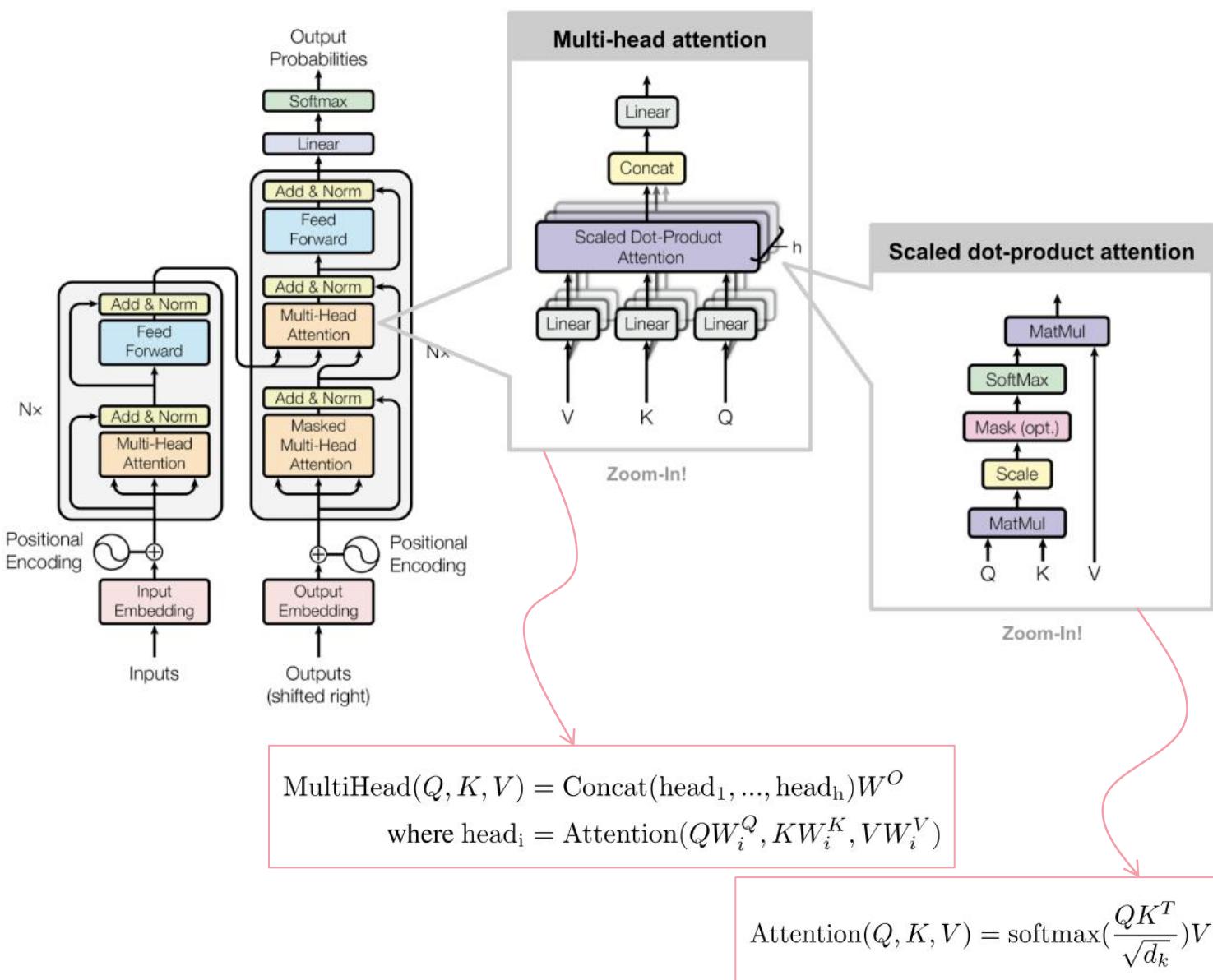
- 时间片 t 的计算依赖 t-1 时刻的计算结果，这样限制了模型的并行能力
- 顺序计算的过程中信息会丢失，尽管LSTM等门机制的结构一定程度上缓解了长期依赖的问题，但是对于特别长期的依赖现象，LSTM依旧无能为力。

Transformer 的提出解决了上面两个问题：

- 首先它使用了Attention机制，将序列中的任意两个位置之间的距离缩小为一个常量；
- 其次它不是类似RNN的顺序结构，因此具有更好的并行性，符合现有的GPU框架。

## 2 Transformer模型架构

Transformer 模型总体的样子如下图所示：总体来说，左边是Encoder部分，右边是Decoder部分。



# 3 Encoder模块

## 3.1 Self-Attention机制

Encoder 模块想要做的事情就是把  $x_1$  转换为另外一个向量  $r_1$ , 这两个向量的维度是一样的。然后就一层层往上传。

转化的过程分成几个步骤:

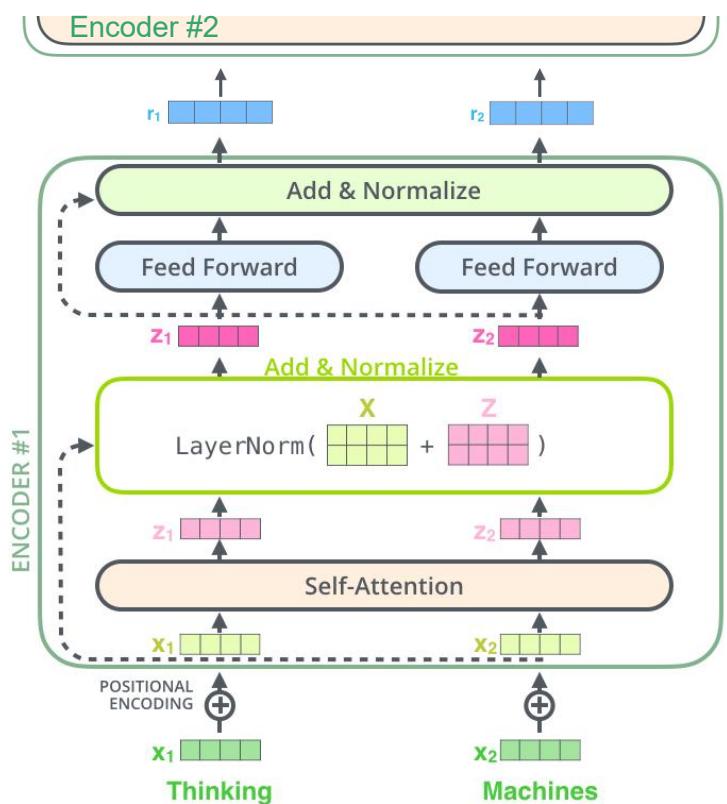
- 第一个步骤就是Self-Attention;
- 第二个步骤就是普通的全连接神经网络。

但是注意, Self-Attention 框里是所有的输入向量共同参与了这个过程, 也就是说,  $x_1$  和  $x_2$  通过某种信息交换和杂糅, 得到了中间变量  $z_1$  和  $z_2$ 。而全连接神经网络是割裂开的,  $z_1$  和  $z_2$  各自独立通过全连接神经网络, 得到了  $r_1$  和  $r_2$ 。

$x_1$  和  $x_2$  互相不知道对方的信息, 但因为在第一个步骤 Self-Attention 中发生了信息交换, 所以  $r_1$  和  $r_2$  各自都有从  $x_1$  和  $x_2$  得来的信息了。

### 为什么叫Self-Attention呢?

就是一个句子内的单词, 互相看其他单词对自己的影响力有多大。所以 Self-Attention 就是说, 句子内各单词的注意力, 应该关注在该句子内其他单词中的哪些单词上。



首先说下Attention和Self-Attention的区别

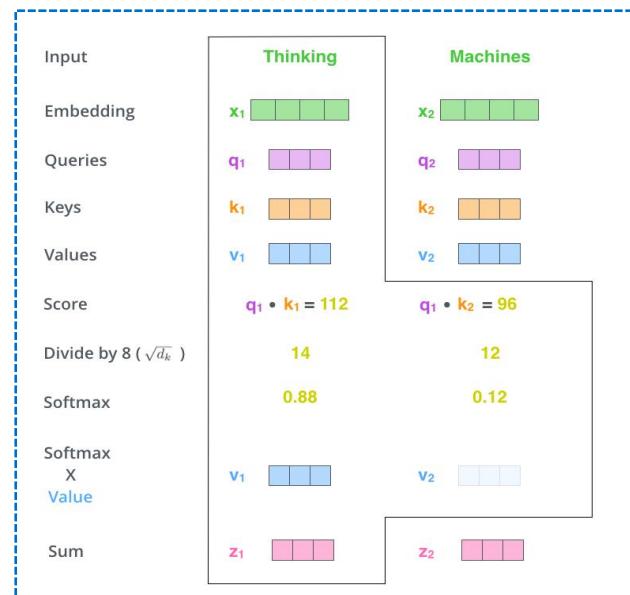
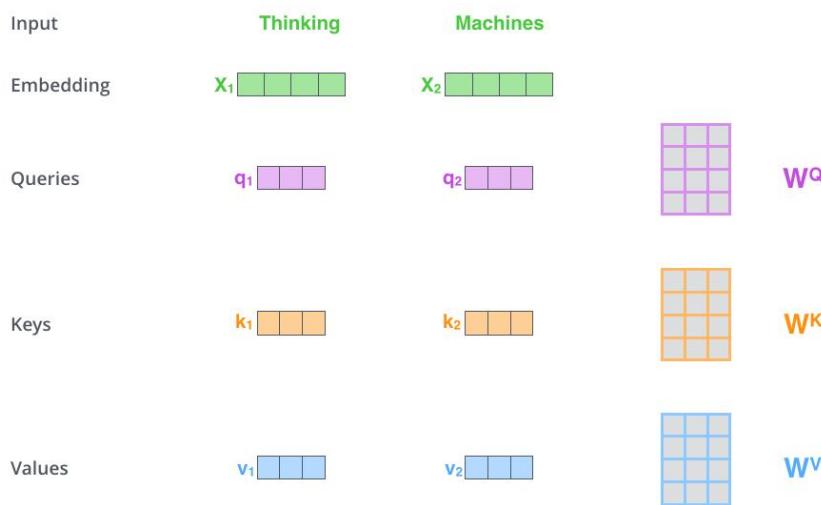
#### Attention和self-attention的区别

以Encoder-Decoder框架为例, 输入Source和输出Target内容是不一样的, 比如对于英-中机器翻译来说, Source是英文句子, Target是对应的翻译出的中文句子, Attention发生在Target的元素Query和Source中的所有元素之间。

Self Attention, 指的不是Target和Source之间的Attention机制, 而是Source内部元素之间或者Target内部元素之间发生的Attention机制, 也可以理解为Target=Source这种特殊情况下的Attention。

两者具体计算过程是一样的, 只是计算对象发生了变化而已。

下图就是 Self-Attention 的计算机制。已知输入的单词 embedding, 即  $x_1$  和  $x_2$ , 想转换成  $z_1$  和  $z_2$ 。



转换方式如下：

先把  $x_1$  转换成三个不一样的向量，分别叫做  $q_1$ 、 $k_1$ 、 $v_1$ ，然后把  $x_2$  转换成三个不一样的向量，分别叫做  $q_2$ 、 $k_2$ 、 $v_2$ 。那把一个向量变成另一个向量的最简单的方式是什么？就是乘以矩阵进行变换。所以，需要三个不同的矩阵  $W^Q$ 、 $W^K$ 、 $W^V$ ，即

$$\begin{aligned} q_1 &= x_1 W^Q & q_2 &= x_2 W^Q \\ k_1 &= x_1 W^K & k_2 &= x_2 W^K \\ v_1 &= x_1 W^V & v_2 &= x_2 W^V \end{aligned}$$

可以注意到，上述过程中，不同的  $x_i$  分享了同一个  $W^Q$ 、 $W^K$ 、 $W^V$ ，通过这个操作， $x_1$  和  $x_2$  已经发生了某种程度上的信息交换。也就是说，单词和单词之间，通过共享权值，已经相互发生了信息的交换。

然后，有了  $q_1$ 、 $k_1$ 、 $v_1$  和  $q_2$ 、 $k_2$ 、 $v_2$ ，怎么才能得到  $z_1$  和  $z_2$  呢？计算过程是这样子的：我们用  $v_1$  和  $v_2$  两个向量的线性组合，来得到  $z_1$  和  $z_2$ ，即

$$\begin{aligned} z_1 &= \theta_{11}v_1 + \theta_{12}v_2 \\ z_2 &= \theta_{21}v_1 + \theta_{22}v_2 \end{aligned}$$

那怎么才能得到组合的权重  $\theta$  呢？有

$$\begin{aligned} [\theta_{11}, \theta_{12}] &= \text{softmax} \left( \frac{q_1 k_1^T}{\sqrt{d_k}}, \frac{q_1 k_2^T}{\sqrt{d_k}} \right) \\ [\theta_{21}, \theta_{22}] &= \text{softmax} \left( \frac{q_2 k_1^T}{\sqrt{d_k}}, \frac{q_2 k_2^T}{\sqrt{d_k}} \right) \end{aligned}$$

通过上述的整个流程，就可以把输入的  $x_1$  和  $x_2$  转换成了  $z_1$  和  $z_2$ 。这就是Self-Attention机制。有了  $z_1$  和  $z_2$ ，再通过全连接层，就能输出该Encoder层的输出  $r_1$  和  $r_2$ 。

讲到这里，你肯定很困惑为什么要有  $q$ 、 $k$ 、 $v$  向量，因为这个思路来自于比较早的信息检索领域， $q$  就是query， $k$  就是key， $v$  就是值，( $k, v$ )就是键值对、也就是用query关键词去找到最相关的检索结果。

用公式表示即为

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

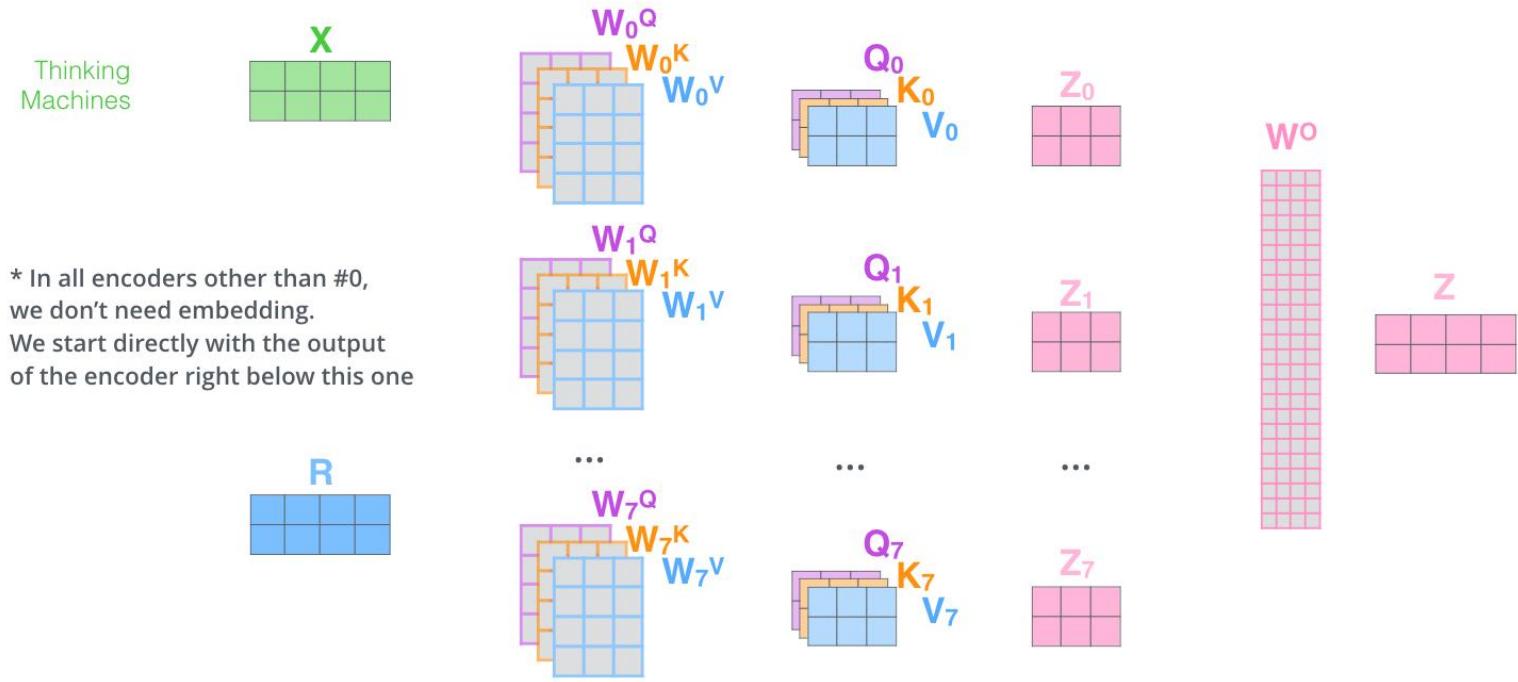
为什么这里要用矩阵而不是神经网络呢？因为矩阵运算能用GPU加速，会更快，同时参数量更少，更节省空间。

注意，上式中的  $d_k$  是向量  $q$  或  $k$  的维度，这两个向量的维度一定是一样的，因为要做点积。但是  $v$  的维度和向量  $q$  或  $k$  的维度不一定相同。上式为什么要除以  $\sqrt{d_k}$  呢？因为为了防止维数过高时  $QK^T$  的值过大导致softmax函数反向传播时发生梯度消失。那为什么是  $\sqrt{d_k}$  而不是  $d_k$  呢？这就是个经验值，从理论上来说，就是还需要让  $QK^T$  的值适度增加，但不能过度增加，如果是  $d_k$  的话，可能就不增加了。

### 3.2 Multi-headed Attention

如果用不同的  $W^Q$ 、 $W^K$ 、 $W^V$ ，就能得到不同的 Q、K、V。Multi-headed Attention就是用了多个不同的  $W^Q$ 、 $W^K$ 、 $W^V$ 。

- 1) This is our input sentence\*  $X$
- 2) We embed each word\*
- 3) Split into 8 heads. We multiply  $X$  or  $R$  with weight matrices  $W_0^Q$ ,  $W_0^K$ ,  $W_0^V$
- 4) Calculate attention using the resulting  $Q/K/V$  matrices
- 5) Concatenate the resulting  $Z$  matrices, then multiply with weight matrix  $W^O$  to produce the output of the layer



### 3.3 词向量 Embedding 输入

Encoder输入的是单词的embedding，通常有两种选择：

- 使用 Pre-trained 的 embeddings 并固化，这种情况下实际就是一个 Lookup Table。
- 对其进行随机初始化（当然也可以选择 Pre-trained 的结果），但设为 Trainable。这样在 training 过程中不断地对 embeddings 进行改进。即 End2End 训练方式。

Transformer 选择后者。

### 3.4 位置编码 Positional Encoding

输入的时候，不仅有单词的向量，还要加上Positional Encoding，即输入模型的整个 Embedding 是 Word Embedding 与 Positional Embedding 直接相加之后的结果。这是想让网络知道这个单词所在句子中的位置是什么，是想让网络做自注意力的时候，不但要知道注意力要聚焦在哪个单词上面，还想要知道单词之间的互相距离有多远。

为什么要知道单词之间的相对位置呢？

因为 Transformer 模型没有用 RNN 也没有卷积，所以为了让模型能利用序列的顺序，必须输入序列中词的位置。所以我们在 Encoder 模块和 Decoder 模块的底部添加了位置编码，这些位置编码和输入的向量的维度相同，所以可以直接相加，从而将位置信息注入。

想要知道单词之间的距离，就得知道单词的坐标。有很多不同衡量距离的方式，这里使用不同频率的 **sin** 和 **cos** 函数：

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

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

- pos 表示 token position, i 表示 embedding dimension。

下面举例说明该公式的用法。

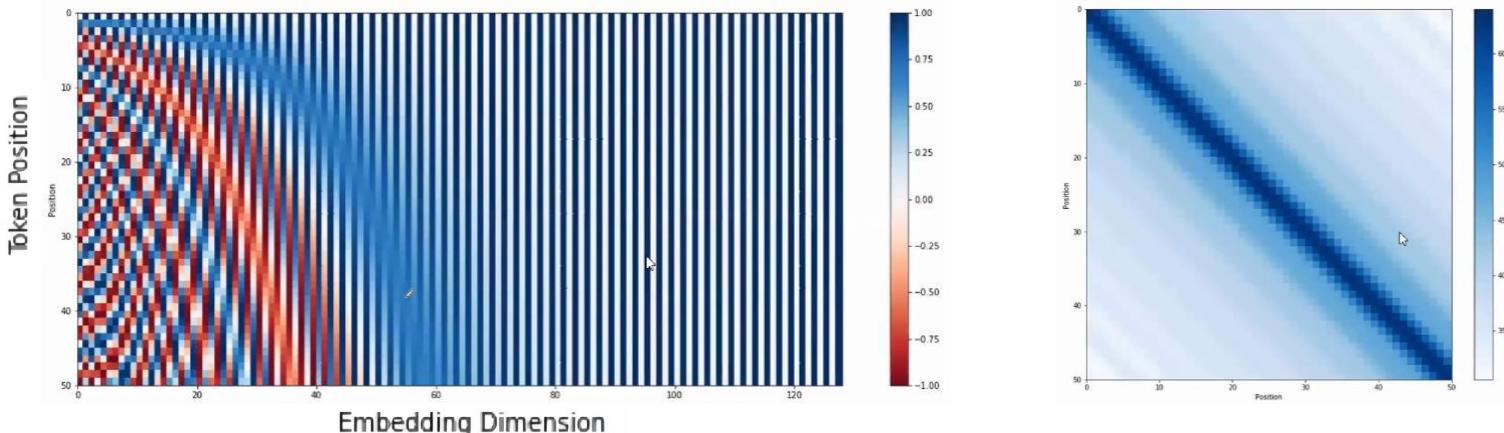
对于一个单词  $w$  位于  $pos \in [0, L - 1]$ , 假设使用 encoding 的维度为 4, 那么这个单词的 encoding 为：

$$e_w = \left[ \sin\left(\frac{\text{pos}}{10000^0}\right), \cos\left(\frac{\text{pos}}{10000^0}\right), \sin\left(\frac{\text{pos}}{10000^{2/4}}\right), \cos\left(\frac{\text{pos}}{10000^{2/4}}\right) \right]$$

$$= \left[ \sin(\text{pos}), \cos(\text{pos}), \sin\left(\frac{\text{pos}}{100}\right), \cos\left(\frac{\text{pos}}{100}\right) \right]$$

				embedding
0	1	2	3	
$2i$	$2i+1$	$2i$	$2i+1$	
$i=0$	$i=0$	$i=1$	$i=1$	

为什么这样做呢？用图形的方式可以直觉上理解。下图为一个长度为50，维度是128的句子的 Positional Encoding（每一行是一个Encoding向量）。下面左图中一行就是一个单词的 Positional Encoding。由该图可以看出，不同位置的 Positional Encoding 是独特的。但是计算 Positional Encoding 的方式不是唯一的，甚至 Positional Encoding 也可以是 train 出来的，并不是必须用作者说的 sin 和 cos。只要能相互计算距离就可以。但是训练出来的不鲁棒，选择正弦曲线版本是因为它可以使模型外推到比训练过程中遇到的序列更长的序列长度。



Positional Encoding 的物理意义是：把50个Positional Encoding两两互相做点积，看相关性。其特点是 Encoding 向量的点积值对称，随着距离增大而减小。（见上面右图）

作者为啥要设计如此复杂的编码规则？原因是 sin 和 cos 的如下特性：

引用自：[https://blog.csdn.net/weixin\\_44294645/article/details/129968986](https://blog.csdn.net/weixin_44294645/article/details/129968986)

$$\begin{cases} \sin(\alpha + \beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta \\ \cos(\alpha + \beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta \end{cases}$$

CSDN @三毛学海

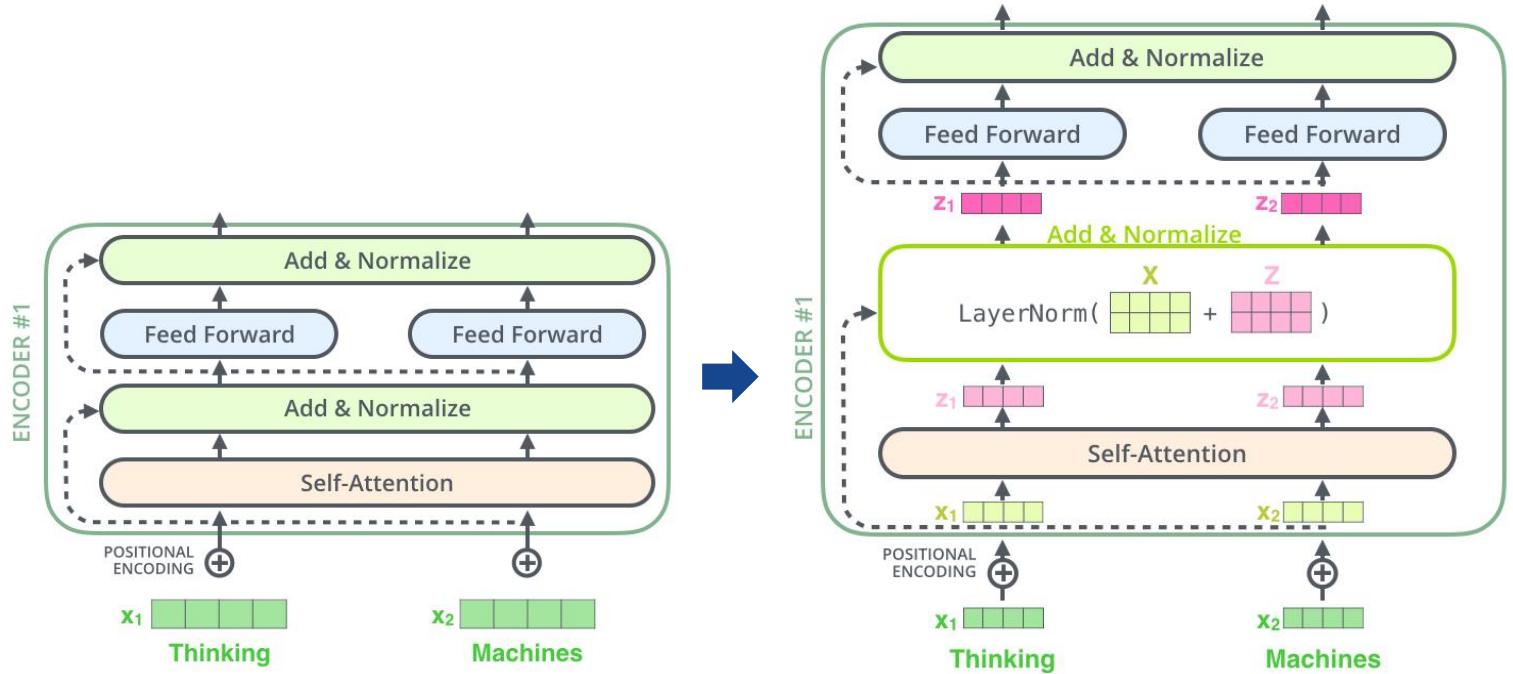
可以将  $PE(pos+k)$  用  $PE(pos)$  进行线性表出：

$$\begin{cases} PE(pos+k, 2i) = PE(pos, 2i) \times PE(k, 2i+1) + PE(pos, 2i+1) \times PE(k, 2i) \\ PE(pos+k, 2i+1) = PE(pos, 2i+1) \times PE(k, 2i+1) - PE(pos, 2i) \times PE(k, 2i) \end{cases}$$

CSDN @三毛学海

假设  $k=1$ , 那么下一个位置的编码向量可以由前面的编码向量线性表示，等价于以一种非常容易学会的方式告诉了网络单词之间的绝对位置，让模型能够轻松学习到相对位置信息。注意编码方式不是唯一的，将单词嵌入向量和位置编码向量相加就可以得到编码器的真正输入了，其输出 shape 是  $(b, N, 512)$ 。

### 3.5 skip connection 和 Layer Normalization



Add & Norm模块接在Encoder端和Decoder端每个子模块的后面，其中Add表示残差连接，Norm表示LayerNorm，残差连接来源于论文[Deep Residual Learning for Image Recognition](#)，LayerNorm来源于论文[Layer Normalization](#)，因此Encoder端和Decoder端每个子模块实际的输出为：

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

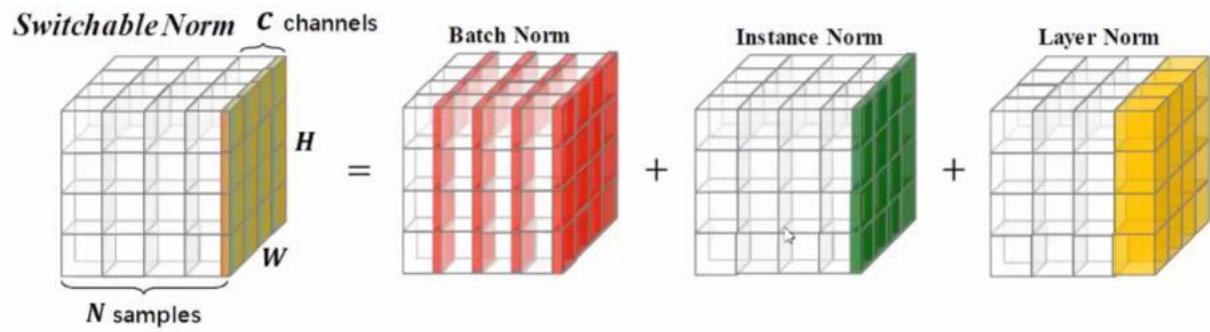
其中 Sublayer( $x$ ) 为子模块的输出。

**skip connection** 最早是在计算机视觉的 ResNet 里面提到的，是微软亚洲研究院的何凯明做的，主要是想解决当网络很深时，误差向后传递会越来越弱，训练就很困难，那如果产生跳跃连接，如果有误差，可以从不同路径传到早期的网络层，这样的话误差就会比较明确地传回来。这就是跳跃层的来历。

跳跃层不是必须的，但在Transformer中，作者建议这样做，在 Self-Attention 的前后和每一个 Feed Forward 前后都用了跳跃层。同时，还用了Normalize，用的是一种新的 **Layer Normalize**，不是常用的Batch Normalize。**是一种正则化的策略，避免网络过拟合。**

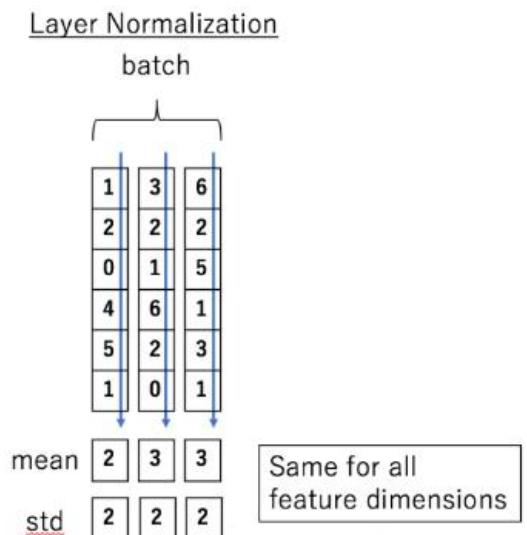
Layer Normalize就是对每一层 $t$ 的所有向量进行求均值 $u^l$ 和方差 $\sigma^l$ ，然后归一化到正态分布后，再学习到合适的均值 $b$ 和方差 $g$ 进行再次缩放，即

$$\begin{aligned}\mu^t &= \frac{1}{H} \sum_{i=1}^H a_i^t, & \sigma^t &= \sqrt{\frac{1}{H} \sum_{i=1}^H (a_i^t - \mu^t)^2} \\ h^t &= f \left[ \frac{g}{\sigma^t} \odot (a^t - \mu^t) + b \right]\end{aligned}$$



Layer Normalize 和 Batch Normalize 唯一的区别就是不考虑其他数据，只考虑自己，这样就避免了不同 batch size 的影响。下图给出一个对不同样本做 Layer Normalization 的实例。

Layer Normalization 的方法可以和 Batch Normalization 对比着进行理解，因为 Batch Normalization 不是 Transformer 中的结构，这里不做详解，详细清楚的解释请看[这里](#)：[NLP中batch normalization与layer normalization](#)。



## 4 Decoder模块

### 4.1 Decoder 的 Mask-Multi-Head-Attention 输入端

模型训练阶段：

- Decoder的初始输入：训练集的标签，并且需要整体右移（Shifted Right）一位
- Shifted Right的原因：T-1时刻需要预测T时刻的输出，所以Decoder的输入需要整体后移一位

举例说明：我爱中国 → I Love China

位置关系：

0-“I”  
1-“Love”  
2-“China”

操作：整体右移一位（Shifted Right）

0-«/s»【起始符】目的是为了预测下一个Token  
1-“I”  
2-“Love”  
3-“China”

- Time Step 1

- 初始输入：起始符 «/s» + Positional Encoding
- 中间输入：（我爱中国）Encoder Embedding
- Decoder：产生预测 I

- Time Step 2

- 初始输入：起始符 «/s» + I + Positional Encoding
- 中间输入：（我爱中国）Encoder Embedding
- Decoder：产生预测 Love

- Time Step 3

- 初始输入：起始符 «/s» + I + Love + Positional Encoding
- 中间输入：（我爱中国）Encoder Embedding
- Decoder：产生预测 China

Transformer Decoder的输入：

- 初始输入：前一时刻 Decoder 输入 + 前一时刻 Decoder 的预测结果 + Positional Encoding
- 中间输入：Encoder Embedding

Attention 的预测流程和普通的 Encoder-Decoder 的模式是一样的，只是用 Self-Attention 替换了 RNN。

在做预测时，步骤如下：

1. 给 Decoder 输入 Encoder 对整个句子 embedding 的结果和一个特殊的开始符号 «/s»。Decoder 将产生预测，在我们的例子中应该是 "I"。
2. 给 Decoder 输入 Encoder 的 embedding 结果和 «/s» I，在这一步 Decoder 应该产生预测 am。
3. 给 Decoder 输入 Encoder 的 embedding 结果和 «/s» I am，在这一步 Decoder 应该产生预测 a。
4. 给 Decoder 输入 Encoder 的 embedding 结果和 «/s» I am a，在这一步 Decoder 应该产生预测 student。
5. 给 Decoder 输入 Encoder 的 embedding 结果和 «/s» I am a student，Decoder 应该生成句子结尾的标记，Decoder 应该输出 «/eos»。
6. 然后 Decoder 生成了 «/eos»，翻译完成。

# 5 Mask (掩码)

mask 表示掩码，它对某些值进行掩盖，使其在参数更新时不产生效果。Transformer 模型里面涉及两种 mask，分别是 padding mask 和 sequence mask。

- padding mask 在所有的 scaled dot-product attention 里面都需要用到；
- sequence mask 只有在 Decoder 的 Self-Attention 里面用到。

## 5.1 Padding Mask

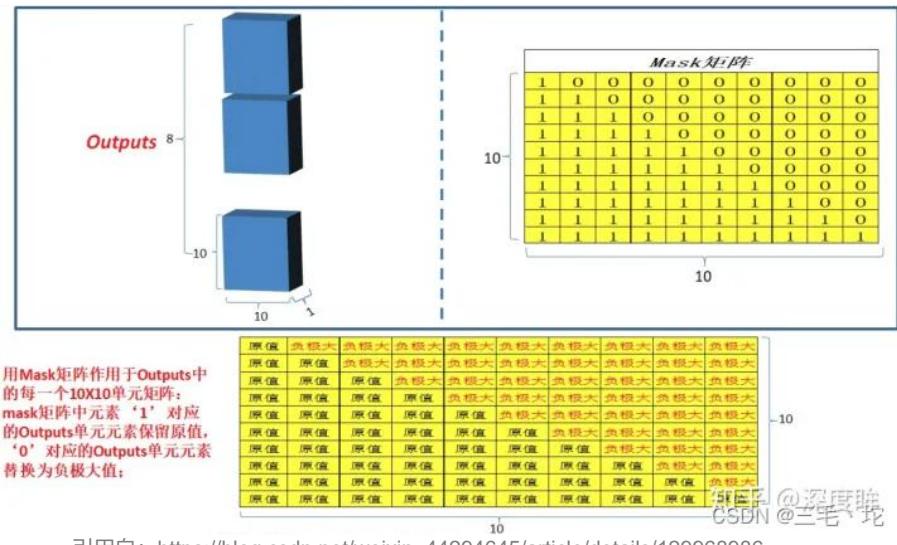
什么是 padding mask 呢？因为每个批次输入序列长度是不一样的也就是说，我们要对输入序列进行对齐。具体来说，就是给在较短的序列后面填充0。但是如果输入的序列太长，则是截取左边的内容，把多余的直接舍弃。因为这些填充的位置，其实是没什么意义的，所以我们的 Attention 机制不应该把注意力放在这些位置上，所以我们需要进行一些处理。

具体是把这些位置的值加上一个非常大的负数(负无穷)，这样的话，经过softmax，这些位置的概率就会接近0！而我们的 padding mask 实际上是一个张量，每个值都是一个 Boolean，值为 false 的地方就是我们要进行处理的地方。

## 5.2 Sequence mask

sequence mask 是为了使得 Decoder 不能看见未来的信息。也就是对于一个序列，在 time\_step 为 t 的时刻，我们的解码输出应该只能依赖于 t 时刻之前的输出，而不能依赖 t 之后的输出。因此我们需要想一个办法，把 t 之后的信息给隐藏起来。sequence mask 的目的是防止 Decoder “seeing the future”，就像防止考生偷看考试答案一样。这里 mask 是一个下三角矩阵，对角线以及对角线左下都是1，其余都是0。下面是个10维度的下三角矩阵：

```
[[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],  
 [1, 1, 0, 0, 0, 0, 0, 0, 0, 0],  
 [1, 1, 1, 0, 0, 0, 0, 0, 0, 0],  
 [1, 1, 1, 1, 0, 0, 0, 0, 0, 0],  
 [1, 1, 1, 1, 1, 0, 0, 0, 0, 0],  
 [1, 1, 1, 1, 1, 1, 0, 0, 0, 0],  
 [1, 1, 1, 1, 1, 1, 1, 0, 0, 0],  
 [1, 1, 1, 1, 1, 1, 1, 1, 0, 0],  
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 0],  
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
```



引用自：[https://blog.csdn.net/weixin\\_44294645/article/details/129968986](https://blog.csdn.net/weixin_44294645/article/details/129968986)

**总结：**对于 Decoder 的 Self-Attention，里面使用到的 scaled dot-product attention，同时需要 padding mask 和 sequence mask 作为 attn\_mask，具体实现就是两个 mask 相加作为 attn\_mask。  
其他情况，attn\_mask 一律等于 padding mask。

举个例子：

假设最大允许的序列长度为10，先令 padding mask 为

```
[0 0 0 0 0 0 0 0 0 0]
```

然后假设当前句子一共有5个单词（加一个起始标识），在输入第三个单词的时候，前面有一个开始标识和两个单词，则此刻的 sequence mask 为

```
[1 1 1 0 0 0]
```

然后 padding mask 和 sequence mask 相加，得

```
[1 1 1 0 0 0 0 0 0 0]
```

Attention 的预测流程和普通的 Encoder-Decoder 的模式是一样的，只是用 Self-Attention 替换了 RNN。

在做预测时，步骤如下：

1. 给 Decoder 输入 Encoder 对整个句子 embedding 的结果和一个特殊的开始符号 `</s>`。Decoder 将产生预测，在我们的例子中应该是 "I"。
2. 给 Decoder 输入 Encoder 的 embedding 结果和 `</s> I`，在这一步 Decoder 应该产生预测 `am`。
3. 给 Decoder 输入 Encoder 的 embedding 结果和 `</s> I am`，在这一步 Decoder 应该产生预测 `a`。
4. 给 Decoder 输入 Encoder 的 embedding 结果和 `</s> I am a`，在这一步 Decoder 应该产生预测 `student`。
5. 给 Decoder 输入 Encoder 的 embedding 结果和 `</s> I am a student`，Decoder 应该生成句子结尾的标记，Decoder 应该输出 `</eos>`。
6. 然后 Decoder 生成了 `</eos>`，翻译完成。

这里有两个训练小技巧，第一个是 label 平滑，第二个就是学习率要有个 worm up 过程，然后再下降。

## 1、Label Smoothing (regularization)

由传统的

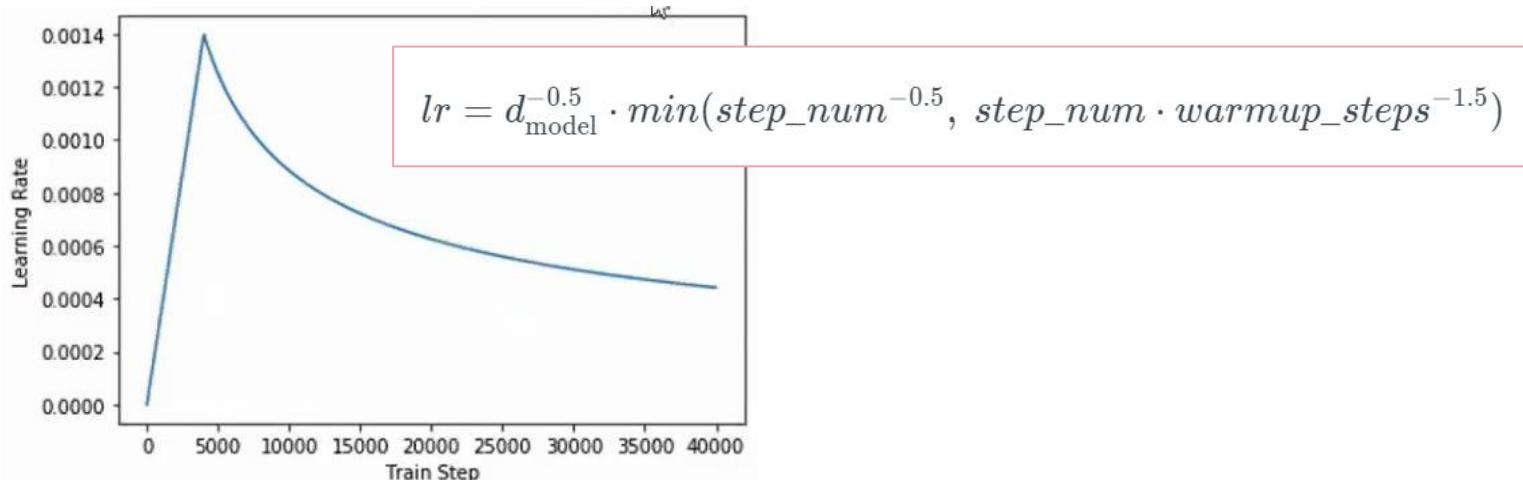
$$P_i = \begin{cases} 1, & \text{if}(i = y) \\ 0, & \text{if}(i \neq y) \end{cases}$$

变为

$$P_i = \begin{cases} (1 - \epsilon), & \text{if}(i = y) \\ \frac{\epsilon}{K-1}, & \text{if}(i \neq y) \end{cases}$$

注： $K$  表示多分类的类别总数， $\epsilon$  是一个较小的超参数。

## 2、Noam Learning Rate Schedule



# 6 Transformer 特点

## 6.1 优点

- 每层计算复杂度比RNN要低。
- 可以进行并行计算。
- 从计算一个序列长度为n的信息要经过的路径长度来看, 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。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大于序列维度D这种情况, 也可以用窗口限制Self-Attention的计算数量。
- 从作者在附录中给出的栗子可以看出, 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。

## 6.2 缺点

在原文中没有提到缺点, 是后来在 Universal Transformers 中指出的, 主要是两点:

- 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)
- 理论上: transformers 不是 computationally universal(图灵完备), 而 RNN 图灵完备。这种非RNN式的模型是非图灵完备的, 无法单独完成NLP中推理、决策等计算问题(包括使用 transformer 的 bert 模型等等)。

## 代码实现

- 哈佛大学自言语言处理组的 notebook, 很详细文字和代码描述, 用 pytorch 实现  
<https://nlp.seas.harvard.edu/2018/04/03/attention.html>
- Google 的 TensorFlow 官方的, 用 tensorflow 实现  
<https://www.tensorflow.org/tutorials/text/transformer>

## 参考文献:

1. Attention Is All You Need: <https://arxiv.org/abs/1706.03762>
2. Transformer入门 (一) —— 结构: <https://blog.csdn.net/yeen123/article/details/125104680>

## Transformer 的 Motivation

Transformer是由谷歌于2017年提出。最初是用在NLP领域，在此之前NLP方向的SOTA模型都是以循环神经网络为基础（RNN, LSTM等）。

- RNN是以串行的方式处理数据，对应到NLP任务上，即按句中词语的先后顺序，每一个时间步长处理一个词语。
- Transformer的巨大创新便在于它并行化的处理：文本中的所有词语都可以在同一时间进行分析，而不是按照序列先后顺序。

为了支持这种并行化的处理方式，Transformer依赖于**注意力机制**。注意力机制可以让模型考虑任意两个词语之间的相互关系，且不受它们在文本序列中位置的影响。通过分析词语之间的两两相互关系，来决定应该对哪些词或短语赋予更多的注意力。

## Transformer的原始框架

Transformer采用Encoder-Decoder架构，下图就是Transformer的结构。其中左半部分是encoder，右半部分是decoder。

PS: Encoder-Decoder 架构不是具体的模型，而是泛指一类结构框架，不同的任务可以用不同的编码器和解码器 (RNN, CNN, LSTM)。  
编码就是将输入Seq转化成固定长度向量，解码就是把之前生成的固定向量再转化为Seq。

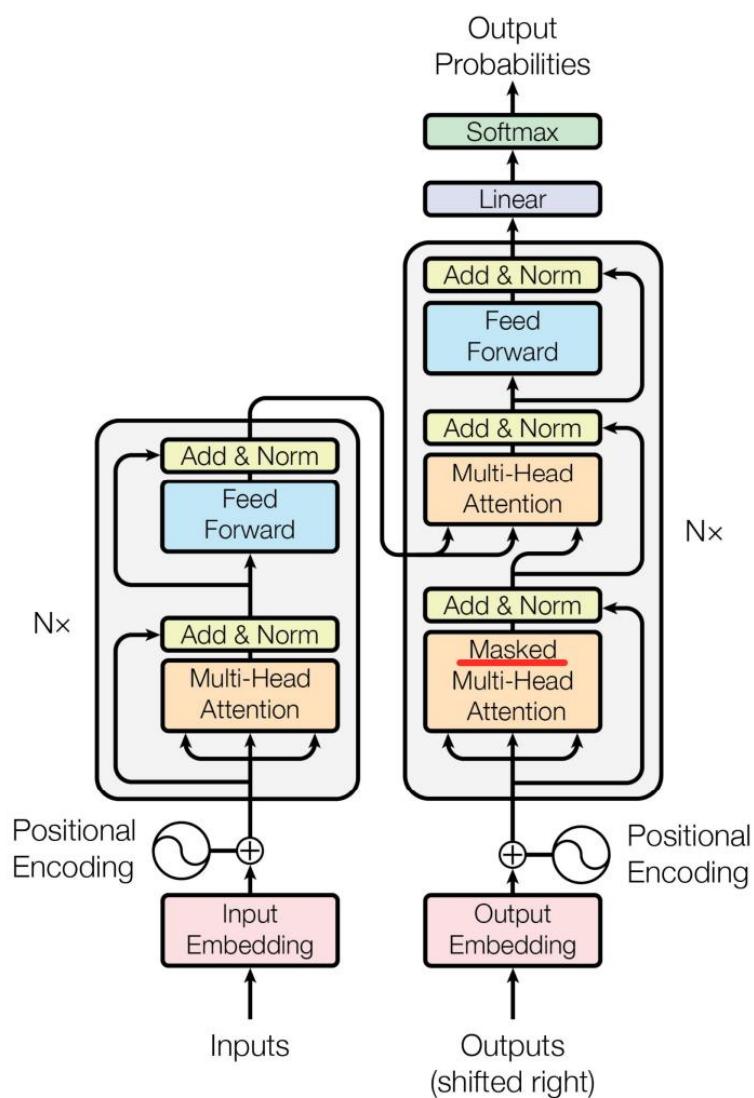
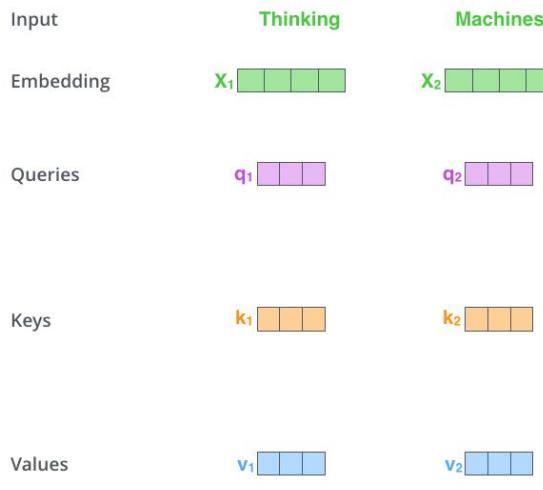


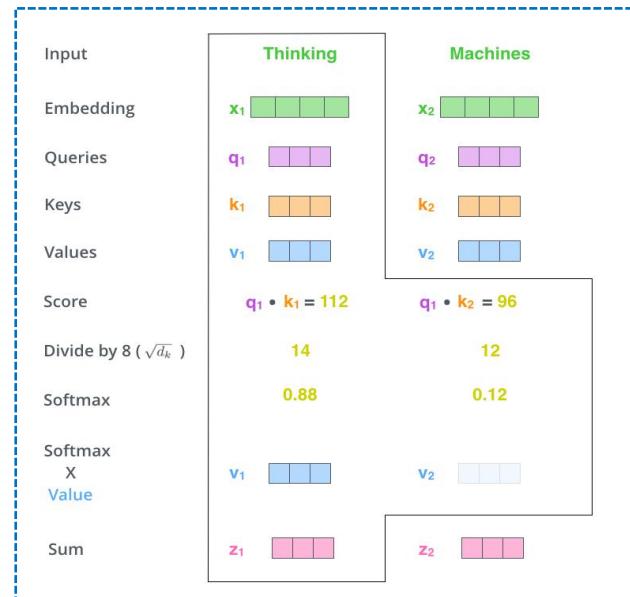
Figure 1: The Transformer - model architecture.

## Encoder 和 Decoder的一些说明

- **Encoder** 的输入包含两个，是一个序列的 token embedding + positional embedding，用正余弦函数对序列中的位置进行计算（偶数位置用正弦，奇数位置用余弦）。
- **Decoder** 单层中的重点是第二个子层，即多头注意力层，它的输入包括两个部分，第一个部分是第一个子层的输出，第二个部分是Encoder层的输出（这是与encoder层的区别之一），这样则将encoder层和decoder层串联起来，以进行词与词之间的信息交换（通过共享权重 $W^Q$ ,  $W^V$ ,  $W^K$ 得到）。
- decoder 层中间的多头自注意力机制的输入是两个参数：encoder 层的输出和 decoder 层中第一层 masked 多头自注意力机制的输出，作用在本层时是： $q=encoder$ 的输出， $k=v=decoder$ 的输出。
- Decoder 单层中的第一个子层中mask，它的作用是防止训练的时候使用未来的输出单词，保证预测位置 $i$ 的信息只能基于比 $i$ 小的输出。比如训练时，第一个单词是不能参考第二个单词的生成结果的，此时就会将第二个单词及其之后的单词都mask掉。因此，encoder 层可以并行计算，一次全部 encoding 出来，但是 decoder 层却一定要像 RNN 一样一个一个解出来，因为要用上一个位置的输入当做 attention 的 query.
- **残差结构**是为了解决梯度消失问题，可以增加模型的复杂性。
- **LayerNorm** 层是为了对attention层的输出进行分布归一化，转换成均值为0方差为1的正态分布。cv中经常会用的是batchNorm，是对一个batchsize中的样本进行一次归一化，而layernorm则是对一层进行一次归一化，二者的作用是一样的，只是针对的维度不同，一般来说输入维度是 (batch\_size, seq\_len, embedding)，batchnorm针对的是 batch\_size 维度进行处理，而 layernorm 则是对 seq\_len 维度进行处理（即batchnorm是对一批样本中进行归一化，而layernorm是对每一个样本进行一次归一化）。
- 使用  $ln$  而不是  $bn$  的原因是因为输入序列的长度问题，每一个序列的长度不同，虽然会经过padding处理，但是 padding的0值其实是无用信息，实际上有用的信息还是序列信息，而不同序列的长度不同，所以这里不能使用  $bn$  一概而论。
- **FFN** 是两层全连接： $w * [delta(w * x + b)] + b$ ，其中的delta是relu激活函数。这里使用FFN层的原因是：为了使用非线性函数来拟合数据。如果说只是为了非线性拟合的话，其实只用到第一层就可以了，但是这里为什么要用两层全连接？是因为第一层的全连接层计算后，其维度是(batch\_size, seq\_len, dff)（其中 dff 是超参数的一种，设置为 2048），而使用第二层全连接层是为了进行维度变换，将 dff 转换为初始的  $d_{model}$  (512)维。



Multiplying  $x_1$  by the  $W_Q$  weight matrix produces  $q_1$ , 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.



图中要用到的参数  $W_Q, W_V, W_K$  是三个随机初始化的矩阵，每个特征词的向量计算公式如下所示：

特征词的向量	计算公式1	计算公式2
Queries	$q_1 = x_1 \cdot W_Q$	$q_2 = x_2 \cdot W_Q$
Keys	$k_1 = x_1 \cdot W_K$	$k_2 = x_2 \cdot W_K$
Values	$v_1 = x_1 \cdot W_V$	$v_2 = x_2 \cdot W_V$
Score	$s_1 = q_1 \cdot k_1 = 112$	$s_2 = q_2 \cdot k_2 = 96$
Divide by 8	$d_1 = s_1 / 8 = 14$	$d_2 = s_2 / 8 = 12$
Softmax	$sm_1 = e^{14} / (e^{14} + e^{12}) = 0.88$	$sm_2 = e^{12} / (e^{14} + e^{12}) = 0.12$
Softmax * value	$v_1 = sm_1 * v_1$	$v_2 = sm_2 * v_2$

## 一般attention 与 self-attention 的区别：

- self-attention 是一般attention 的特殊情况，在 self-attention 中， $Q=K=V$  每个序列中的单元和该序列中所有单元进行 attention 计算。Google 提出的 多头attention 通过计算多次来捕获不同子控件上的相关信息。
- self-attention 的特点在于无视词之间的距离直接计算依赖关系，能够学习一个句子的内部结构，实现也较为简单并且可以并行计算。从一些论文中看到，self-attention 可以当成一个层和 RNN, CNN, FNN等配合使用，成功应用于其他NLP任务。

上表中 attention 要除以8 (根号  $d_k$ ) 的原因是为了缩放，它具备分散注意力的作用；原始注意力值均聚集在得分最高的那个值，获得了权重为 1；而缩放后，注意力值就会分散一些。

attention 中除以根号  $d_k$  具备缩放的原因是因为原始表征  $x_1$  是符合均值为 0、方差为 1 的正态分布的，而与权重矩阵相乘后，结果符合均值为 0、方差为  $d_k$  的正态分布了，所以为了不改变原始表征的分布，需要除以根号  $d_k$ 。

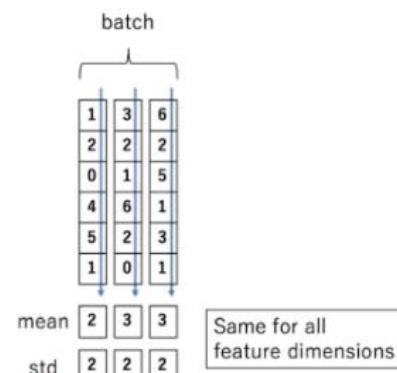
## Layer Normalization

### Layer Normalization模块

主要因为样本间长度不统一，不能直接用BN。

Normalize 层的目的就是对输入数据进行归一化，将其转化成均值为0、方差为1的数据。LN是在每一个样本上都计算均值和方差，如下右图所示：

LN的公式为：  $LN(x_i) = \alpha * (x_i - \mu_L / \sqrt(\sigma^2 L + \epsilon)) + \beta$



## 参考文献:

1. Transformer 模型详解: <https://devpress.csdn.net/hefei/63a5663eb878a54545946354.html>

## 正则化操作

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

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

## 参考文献:

1. 详解Transformer (Attention Is All You Need) : <https://zhuanlan.zhihu.com/p/48508221>

论文给出的编码公式如下：

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \quad (3)$$

$$PE(pos, 2i + 1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \quad (4)$$

在上式中， $pos$  表示单词的位置， $i$  表示单词的维度。关于位置编码的实现可在 Google 开源的算法中 `get_timing_signal_1d()` 函数找到对应的代码。

作者这么设计的原因是考虑到在 NLP 任务中，除了单词的绝对位置，单词的相对位置也非常重要。

根据公式  $\sin(\alpha + \beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta$  以及

$\cos(\alpha + \beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta$ ，这表明位置  $k + p$  的位置向量可以表示为位置  $k$  的特征向量的线性变化，这为模型捕捉单词之间的相对位置关系提供了非常大的便利。

## 总结

### 优点:

1. 虽然 Transformer 最终也没有逃脱传统学习的套路，Transformer 也是一个全连接（或者是一维卷积）加 Attention 的结合体。但是其设计已经足够有创新，因为其抛弃了在 NLP 中最根本的 RNN 或者 CNN 并且取得了非常不错的效果，算法的设计非常精彩，值得每个深度学习的相关人员仔细研究和品味。
2. Transformer 的设计最大的带来性能提升的关键是将任意两个单词的距离是 1，这对解决 NLP 中棘手的长期依赖问题是有效的。
3. Transformer 不仅仅可以应用在 NLP 的机器翻译领域，甚至可以不局限于 NLP 领域，是非常有科研潜力的一个方向。
4. 算法的并行性非常好，符合目前的硬件（主要指 GPU）环境。

### 缺点:

1. 粗暴的抛弃 RNN 和 CNN 虽然非常炫技，但它也使模型丧失了捕捉局部特征的能力，RNN + CNN + Transformer 的结合可能会带来更好的效果。
2. Transformer 失去的位置信息其实在 NLP 中非常重要，而论文中在特征向量中加入 Position Embedding 也只是一个权宜之计，并没有改变 Transformer 结构上的固有缺陷。