

## # Attention Mechanism

### 1. 引言

#### 1.1 基本概念

- Attention 是一种用于提升基于 RNN（LSTM 或 GRU）的 Encoder + Decoder 模型的效果的机制（Mechanism），一般称为 Attention Mechanism。Attention Mechanism 目前非常流行，广泛应用于机器翻译、语音识别、图像标注（Image Caption）等很多领域。Attention 给模型赋予了区分辨别的能力，例如，在机器翻译、语音识别应用中，为句子中的每个词赋予不同的权重，使神经网络模型的学习变得更加灵活（soft），同时 Attention 本身可以作为一种对齐关系，解释翻译输入/输出句子之间的对齐关系，解释模型到底学到了什么。

#### 1.2 直观理解

- 与人类对外界事物的观察机制很类似，在某种情景下（根据不同的任务可能关注点不同）我们注意力会集中在这张图片的一个区域内，而其他的信息受关注度会相应降低。Attention 机制让神经网络更聚焦于对那些对最终任务有帮助的部分。
- 翻译例子：

输入为英语序列“*They*” “*are*” “*watching*” “.”

输出为法语序列“*Ils*” “*regardent*” “.”。

可以想象，解码器在生成输出序列中的每一个词时可能只需利用输入序列某一部分的信息。例如，在输出序列的时间步 1，解码器可以主要依赖“*They*” “*are*”的信息来生成“*Ils*”，在时间步 2 则主要使用来自“*watching*”的编码信息生成“*regardent*”，最后在时间步 3 则直接映射句号“.”。这看上去就像是在解码器的每一时间步对输入序列中不同时步的表征或编码信息分配不同的注意力一样。

## 2. 注意力机制

### 2.1 基本原理

- Attention 的实质是加权求和。通过一个额外的神经网络层，用于选择输入的某些部分或者给输入的不同部分分配不同的权重。这个权是通过计算向量之间的相关性进行度量的。
- 仍然以循环神经网络为例，注意力机制通过对编码器所有时间步的隐藏状态做加权平均来得到背景变量。解码器在每一时间步调整这些权重，即注意力权重，从而能够在不同时间步分别关注输入序列中的不同部分并编码进相应时间步的背景变量。

### 2.2 问题解决

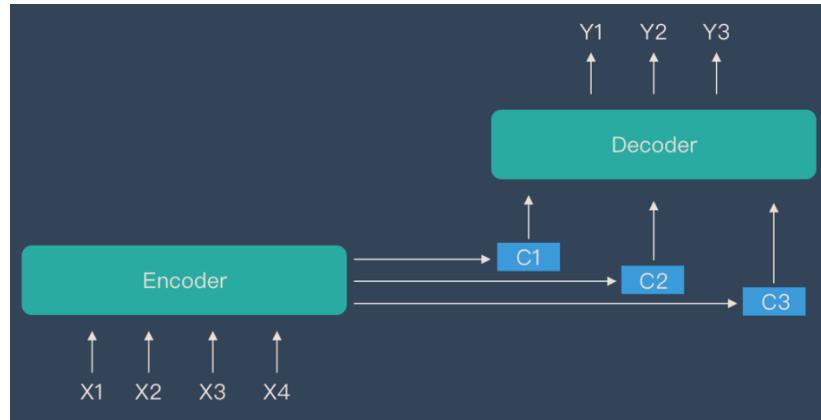
- 注意力机制主要解决了基础 Seq2Seq 所存在的缺陷，即：
  - a) 中间语义向量  $C$  无法完全表达整个输入序列的信息。
  - b) 中间语义向量  $C$  对输出  $y_1, y_2, \dots, y_m$  所产生的贡献都是一样的，即句子 source 中每个单词分配到的权重是相同的（对生成某个目标单词  $y_i$  来说影响力都是相同的）。
  - c) 随着输入信息长度的增加，先前编码好的信息会被后来的信息覆盖，丢失很多信息。

### 2.3 分类

- 按可微性分
  - a) Hard-Attention (硬注意力)  
0/1 问题，被 attention 的输入部分被选择，否则不关注不选择（不参与计算）。Hard-attention 是一个随机预测的过程，更关注点，强调动态变化。不可微，训练过程往往通过强化学习 (reinforcement learning) 完成。
  - b) Soft-Attention (软注意力)  
[0, 1] 连续分布，每个区域被关注的程度高低，用 0 ~ 1 的 score 表示。Soft-attention 更关注区域或者通道，是确定性的注意力，可以直接通过网络学习生成，可微。
- 按关注域分
  - a) Spatial Domain (空间域)
  - b) Channel Domain (通道域)
  - c) Layer Domain (层域)
  - d) Mixed Domain (混合域)
  - e) Time Domain (时间域)

### 3. 引入 Attention Mechanism 的 Seq2Seq 模型

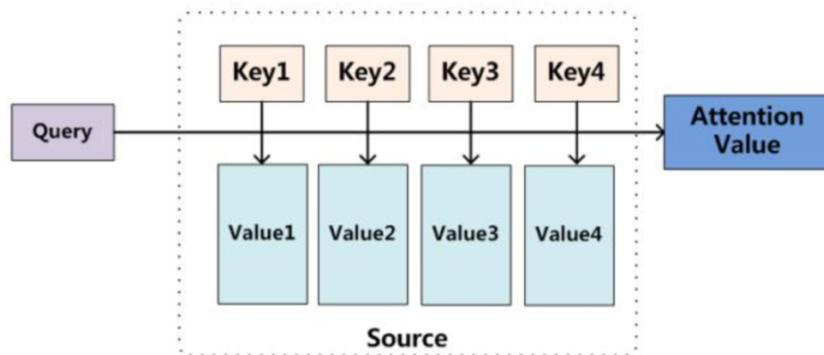
- 如果一个输出句有 N 个文字，就会产生 N 个 **context vector**，即下图中  $C_1, C_2, C_3$  和  $y_1, y_2, y_3$  一一对应。 $C_i$  考虑一整个 Sequence 的信息，且对于每个输出来说，会对输入的序列有不同的侧重（找到对于每个输出来说一整个 Sequence 中哪些部分更加重要）。



#### 3.1 计算本质理解

将 **Source** 中的构成元素想象成是由一系列的(**Key**, **Value**)数据对构成，此时给定 **Target** 中的某个元素 **Query**，通过计算 **Query** 和各个 **Key** 的相似性或者相关性，得到每个 **Key** 对应 **Value** 的权重系数，然后对 **Value** 进行加权求和，即得到了最终的 **Attention** 数值。所以本质上 **Attention** 机制是对 **Source** 中元素的 **Value** 值进行加权求和，而 **Query** 和 **Key** 用来计算对应 **Value** 的权重系数。即可以将其本质思想改写为如下公式：

$$\text{Attention}(\text{Query}_j, \text{Source}) = \sum_{i=1}^N \text{Similarity}(\text{Query}_j, \text{Key}_i) * \text{Value}_i$$



从本质上理解，**Attention** 是从大量信息中有选择地筛选出少量重要信息并聚焦到这些重要信息上，忽略大多不重要的信息。聚焦的过程体现在权重系数的计算上，权重越大越聚焦于其对应的 **Value** 值上，即权重代表了信息的重要性，而 **Value** 是其对应的信息。

#### 3.2 注意力机制流程

- 流程如下

- a) 信息输入。
- b) (核心步骤) 计算 **Attention Score**, 即注意力权重,  $\alpha$ , 衡量了输入句中的每个文字对目标句中的每个文字所带来重要性的程度。
- c) 一般可以接一个 softmax 对这些权重进行归一化, 也可以使用 ReLU 等激活函数。
- d) 计算输入信息的加权平均。

### 3.3 计算 Attention Score

- 也就是计算  $Similarity(Query_j, Key_i)$
- Attention 的实质是加权求和, 这个权是通过计算向量之间的相关性进行度量的。

根据不同的相关性计算方式, 可以有不同的 Attention Score 计算方式:

- a) 加性相关性(addictive)

$$\alpha_{i,j} = W \tanh(q^j + k^i)$$

- b) 点积相关性(dot product) [最常用], 比加性相关性计算更快

$$\alpha_{i,j} = q^j \cdot k^i$$

- c) 缩放点积相关性 [Transformer 中使用]

$$\alpha_{i,j} = \frac{(q^j \cdot k^i)}{\sqrt{d}}$$

其中,  $d$  是输入信息的维度。向量的点积结果会很大, 将 softmax 函数 push 到梯度很小的区域, scaled 会缓解这种现象。缩放点积相关性可以让 Attention Score 的均值和方差和  $q, k$  相同, 有利于缓解梯度消失。

可参考:

[transformer 中的 attention 为什么 scaled? - TniL 的回答](#)

- d) 双线性相关性

$$\alpha_{i,j} = q^j \cdot W \cdot k^i$$

### 3.4 计算 Attention 输出矩阵

- 根据不同相关性的计算方法, 我们利用  $Q$  和  $K$  计算 Attention Score, 此时只计算了相关性权重, 还需要对输入信息进行加权求和得到 Attention 输出矩阵:

$$att(q, V) = \sum_{i=1}^N \alpha_i V_i$$

- 整体来看:

- a) 当矩阵特征向量以列向量形式表示时, attention 的输出矩阵可以按照下述公式计算 (以缩放点积相关性 + softmax 为例)

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

- b) 当矩阵特征向量以行向量形式表示时, attention 的输出矩阵可以按照下述公式计算 (以缩放点积相关性+softmax 为例)

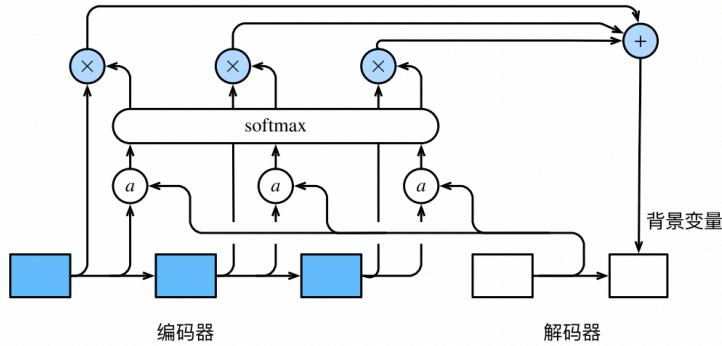
$$Attention(Q, K, V) = \text{softmax}\left(\frac{Q K^T}{\sqrt{d_k}}\right) V$$

### 3.5 计算详解

- 在注意力机制中，解码器的每一时间步将使用可变的语义向量。记 $\mathbf{c}_{t'}$ 是解码器在时间步 $t'$ 的语义向量。那么解码器在该时间步的隐藏状态可以改写为：

$$\mathbf{s}_{t'} = g(\mathbf{y}_{t'-1}, \mathbf{c}_{t'}, \mathbf{s}_{t'-1})$$

- 下图描绘了注意力机制如何为解码器在时间步 2 计算背景变量。首先，函数 $a$ 根据解码器在时间步 1 的隐藏状态和编码器在各个时间步的隐藏状态计算 softmax 运算的输入。softmax 运算输出概率分布并对编码器各个时间步的隐藏状态做加权平均，从而得到语义向量（背景变量）。



具体来说，令编码器在时间步 $t$ 的隐藏状态为 $\mathbf{h}_t$ ，且总时间步数为 $T$ 。那么解码器在时间步 $t'$ 的背景变量为所有编码器隐藏状态的加权平均：

$$\mathbf{c}_{t'} = \sum_{t=1}^T \alpha_{t't} \mathbf{h}_t$$

其中在给定  $t'$  时，权重  $\alpha_{t't}$  在  $t = 1, \dots, T$  的值是一个概率分布，为了得到概率分布，可以利用 softmax 运算：

$$\alpha_{t't} = \frac{\exp(e_{t't})}{\sum_{k=1}^T \exp(e_{t'k})}, \quad t = 1, \dots, T$$

现在，需要定义上述 softmax 运算中的输入  $e_{t't}$ （实际上就是 3.2 中的 Attention Score，此处的  $\alpha_{t't}$  是对 Attention Score 进行 softmax 处理后的结果）

由于  $e_{t't}$  同时取决于解码器的时间步  $t'$  和编码器的时间步  $t$ ，可以以解码器在时间步  $t' - 1$  的隐藏状态  $\mathbf{s}_{t'-1}$  与编码器在时间步  $t$  的隐藏状态  $\mathbf{h}_t$  为输入，通过函数  $a$  计算：

$$e_{t't} = a(\mathbf{s}_{t'-1}, \mathbf{h}_t)$$

这里函数  $a$  有多种选择，如果两个输入向量长度相同，一个简单的选择是计算内积  $a(\mathbf{s}, \mathbf{h}) = \mathbf{s}^T \mathbf{h}$ 。而最早提出注意力机制的论文则将输入连结后通过含单隐藏层的多层感知机变换：

$$a(\mathbf{s}, \mathbf{h}) = \mathbf{v}^T \tanh(\mathbf{W}_s \mathbf{s} + \mathbf{W}_h \mathbf{h})$$

其中， $\mathbf{v}, \mathbf{W}_s, \mathbf{W}_h$  都是可以学习的模型参数。

- 矢量化计算 ( $Q, K, V$  的来源)

- 可以对注意力机制采用更高效的矢量化计算。广义上，注意力机制的输入包括查询项 (Query) 以及一一对应的键项 (Key) 和值项 (Value)，其中值项

是需要加权平均的一组项。在加权平均中，值项的权重来自查询项以及与该值项对应的键项的计算。

- b) 在上面的例子中，**查询项为解码器的隐藏状态，键项和值项均为编码器的隐藏状态**。考虑一个常见的简单情形，即编码器和解码器的隐藏单元个数均为 $h$ ，且函数 $a(\mathbf{s}, \mathbf{h}) = \mathbf{s}^T \mathbf{h}$ 。假设我们希望根据解码器单个隐藏状态 $\mathbf{s}_{t'-1} \in \mathbb{R}^h$ 和编码器所有隐藏状态 $\mathbf{h}_t \in \mathbb{R}^h, t = 1, \dots, T$ 来计算语义向量 $\mathbf{C}_{t'} \in \mathbb{R}^h$ 。
- c) 可以将查询项矩阵设为 $\mathbf{Q} \in \mathbb{R}^{1 \times h}$ 设为 $\mathbf{s}_{t'-1}^T$ ，并令键项矩阵 $\mathbf{K} \in \mathbb{R}^{T \times h}$ 和值项矩阵 $\mathbf{V} \in \mathbb{R}^{T \times h}$ 相同且第 $t$ 行均为 $\mathbf{h}_t^T$ 。此时，只需要通过矢量化计算

$$\text{softmax}(\mathbf{Q}\mathbf{K}^T)\mathbf{V}$$

即可算出转置后的语义向量 $\mathbf{C}_{t'}^T$ 。当查询项矩阵 $\mathbf{Q}$ 的行数为 $n$ 时，上式将得到 $n$ 行的输出矩阵。输出矩阵与查询项矩阵在相同行上一一对应。

- 在明确需要获得什么对什么的 **Attention** 时，需要知道哪个向量作为 **Query**，哪个为 **Key**，哪个为 **Value**。

### 3.6 Seq2Seq 中 Attention 机制存在的问题

- 可以发现，Seq2Seq 中 attention 机制将输入句子和生成句子之间进行 attention，且下一个输出依赖输入句子和上一个输出的结果。这种形式存在两个问题：
  - a) 是串行的，存在无法并行化的问题，导致计算速度很慢。
  - b) 关注句子和句子间的特征，没有关注句子内部的信息（如语法特征、短语结构等）。

## 4. Transformer 中的 Self-Attention 机制

### 4.1 引言

- 翻译例子

“I arrived at the bank after crossing the river” 这里面的 bank 指的是银行还是河岸呢，这就需要联系上下文，当我们看到 river 之后就应该知道这里 bank 很大概率指的是河岸。在引入 Attention 机制的 Seq2Seq 中就需要一步一步的顺序处理从 bank 到 river 的所有词语，而当它们相距较远时模型的效果常常较差，且由于其顺序性处理效率也较低。Self-Attention 则利用了 Attention 机制，计算每个单词与其他所有单词之间的关联，在这句话里，当翻译 bank 一词时，river 一词就有较高的 Attention Score。

- 前面提到 Seq2Seq 中的 Attention 机制存在的两个问题可以用 Self-Attention 机制解决。
- Transformer 去掉了神经网络结构，仅仅依赖 Attention 机制。如下图所示。

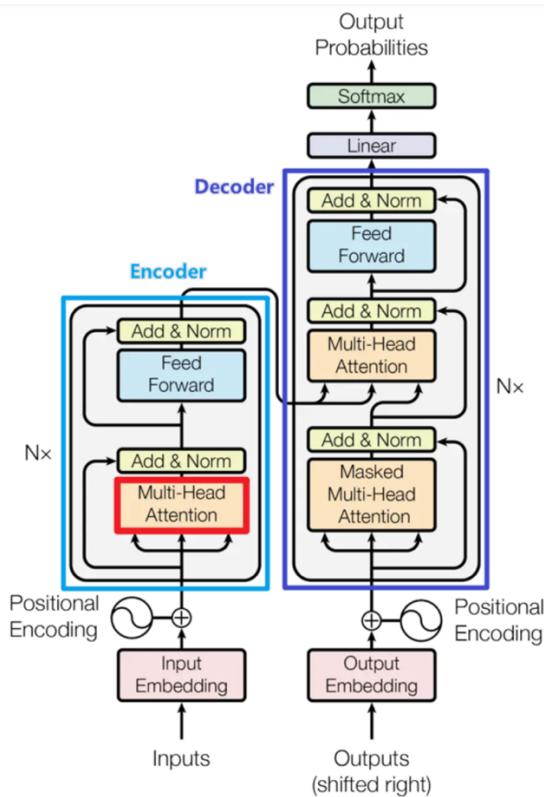


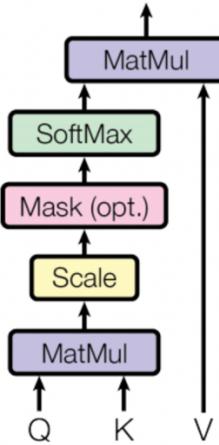
Figure 1: The Transformer - model architecture.

- Embedding:** 在机器学习领域，Embedding 指的是把某一个距离空间嵌入另一个距离空间比如把图像空间嵌入特征空间。简单来说 embedding 就是，**特征向量提取**。

### 4.2 Self-Attention 机制

- 基本结构（以缩放点积为例）

## Scaled Dot-Product Attention

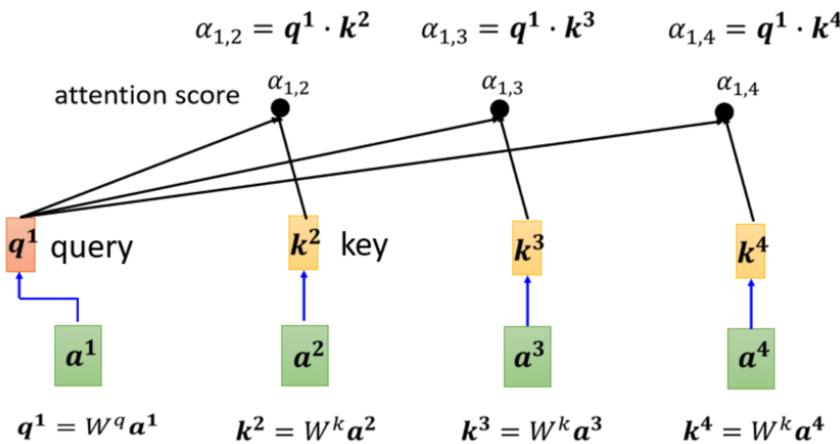


- 核心公式

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

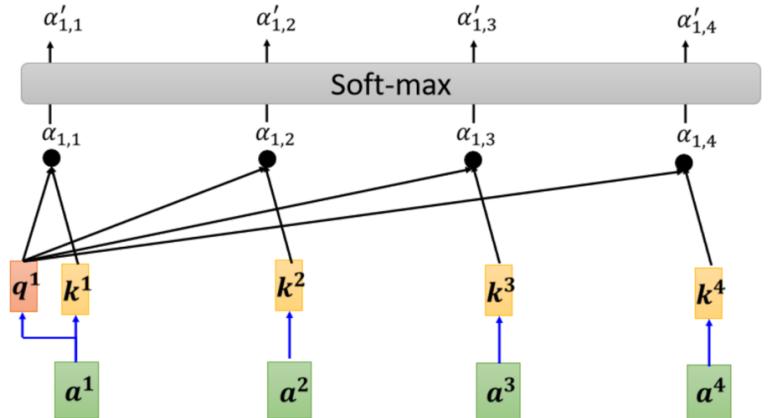
关于 $\text{softmax}(\mathbf{X}\mathbf{X}^T)\mathbf{X}$ 的实际意义可以参考: [超详细图解 Self-Attention](#)

- Attention 机制的通用形式, 其中 $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ 如何选择其实就影响了 Attention 输出矩阵的表达, Self-Attention 顾名思义就是对自己进行 Attention, 其 $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ 都是使用同一个输入矩阵 $\mathbf{X}$ , 然后分别用三个不同矩阵 $\mathbf{W}^q, \mathbf{W}^k, \mathbf{W}^v$ , 进一步线性变换得到的。
- $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ 的本质就是 $\mathbf{X}$ 的线性变换, 目的在于提升模型的拟合能力。  
 $\mathbf{W}^q, \mathbf{W}^k, \mathbf{W}^v$ 是训练过程中需要学习的参数。
- Self-Attention 的计算过程如下:



**Self-attention**

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

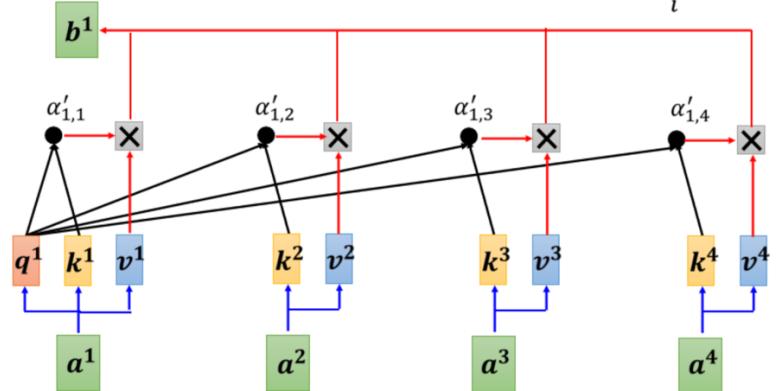


$$q^1 = W^q a^1 \quad k^2 = W^k a^2 \quad k^3 = W^k a^3 \quad k^4 = W^k a^4$$

$$k^1 = W^k a^1$$

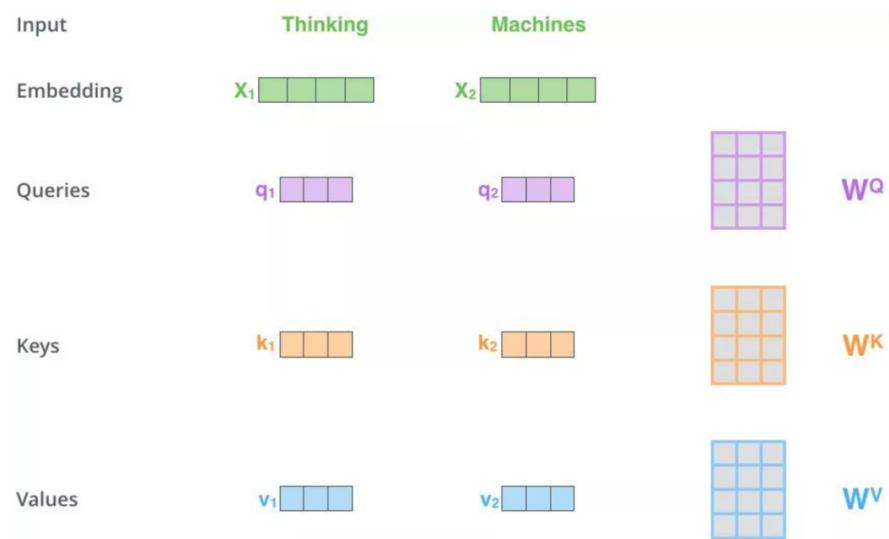
18

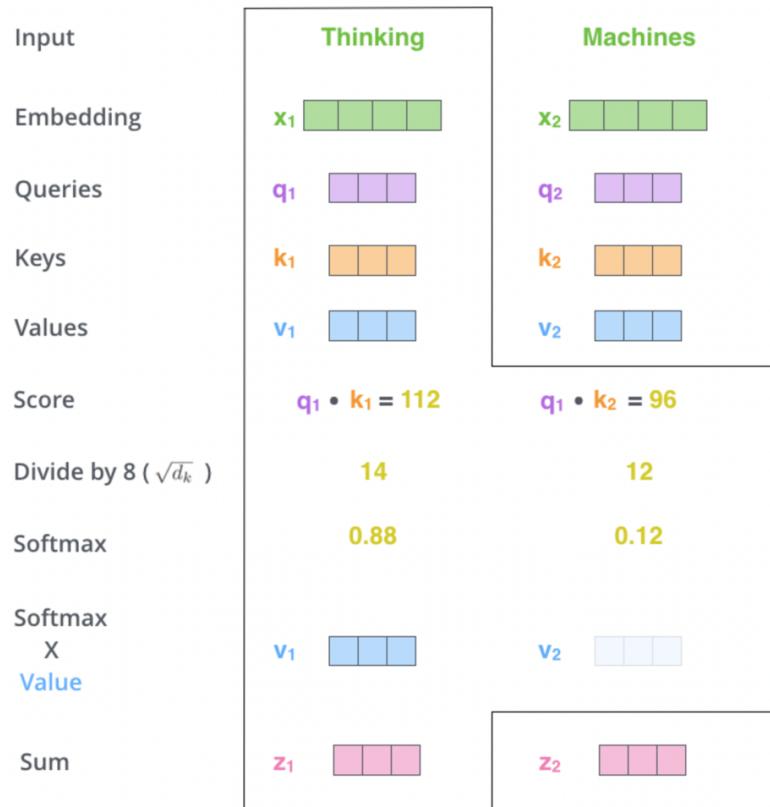
**Self-attention** Extract information based on attention scores  $b^1 = \sum_i \alpha'_{1,i} v^i$



$$v^1 = W^v a^1 \quad v^2 = W^v a^2 \quad v^3 = W^v a^3 \quad v^4 = W^v a^4$$

- 计算可视化 (Embedding 是行向量形式, 即  $q_1 = X_1 W^Q$ )





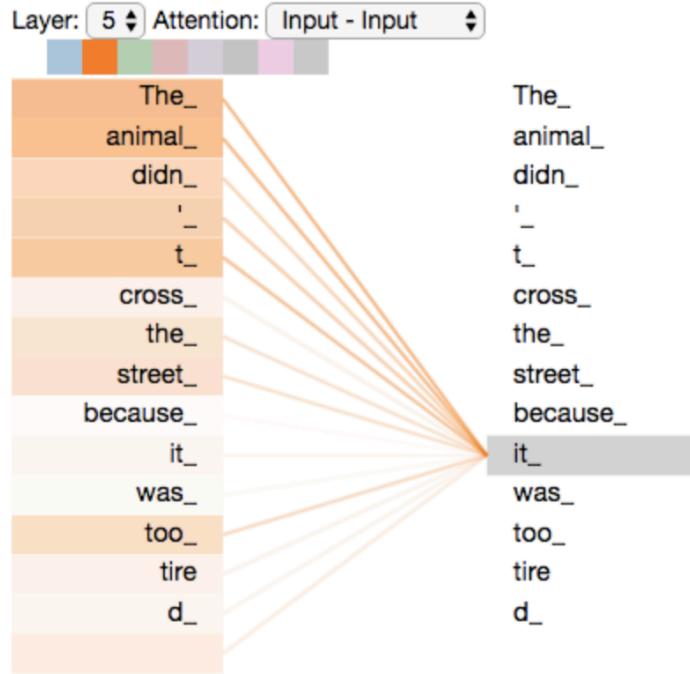
写成矩阵形式:

$$\begin{array}{ccc}
 X & W^Q & Q \\
 \begin{matrix} \text{---} \\ | \\ \text{---} \end{matrix} & \times & \begin{matrix} \text{---} \\ | \\ \text{---} \end{matrix} \\
 \begin{matrix} \text{---} \\ | \\ \text{---} \end{matrix} & & \begin{matrix} \text{---} \\ | \\ \text{---} \end{matrix} \\
 & & = \\
 & & \begin{matrix} \text{---} \\ | \\ \text{---} \end{matrix} \\
 \hline
 X & W^K & K \\
 \begin{matrix} \text{---} \\ | \\ \text{---} \end{matrix} & \times & \begin{matrix} \text{---} \\ | \\ \text{---} \end{matrix} \\
 \begin{matrix} \text{---} \\ | \\ \text{---} \end{matrix} & & \begin{matrix} \text{---} \\ | \\ \text{---} \end{matrix} \\
 & & = \\
 & & \begin{matrix} \text{---} \\ | \\ \text{---} \end{matrix} \\
 \hline
 X & W^V & V \\
 \begin{matrix} \text{---} \\ | \\ \text{---} \end{matrix} & \times & \begin{matrix} \text{---} \\ | \\ \text{---} \end{matrix} \\
 \begin{matrix} \text{---} \\ | \\ \text{---} \end{matrix} & & \begin{matrix} \text{---} \\ | \\ \text{---} \end{matrix} \\
 & & = \\
 & & \begin{matrix} \text{---} \\ | \\ \text{---} \end{matrix}
 \end{array}$$

转换成公式:

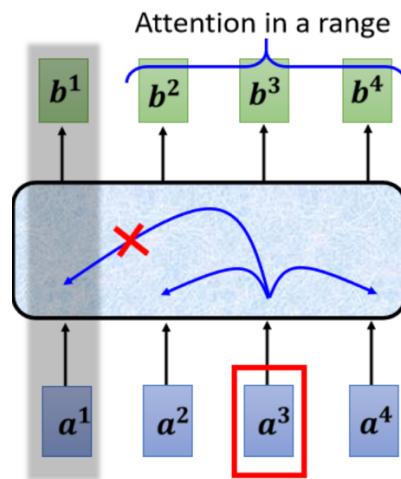
$$\begin{aligned}
 & \text{softmax} \left( \frac{Q \times K^T}{\sqrt{d_k}} \right) V \\
 & = Z
 \end{aligned}$$

- 举个 NLP 的例子来看，Self-Attention 机制建立了单词和整个句子的 Attention 关系，可以学习到句子内部的信息。此外，Self-Attention 是支持并行的，在计算每个单词的 Attention 时互不影响，但也因此损失了句子内部单词之间的位置信息。



#### 4.3 Truncated Self-Attention

- 输入序列的长度为  $N$ ，则 Self-Attention 计算得到的 Attention Score 大小为  $N \times N$ 。在面对超长文本或者语音时，Self-Attention 的计算量非常大，需要的内存也很大。这个时候可以使用 Truncated Self-Attention，相当于仅考虑一个窗口内的文本或者语音做 Attention，可以加快运算的速度。



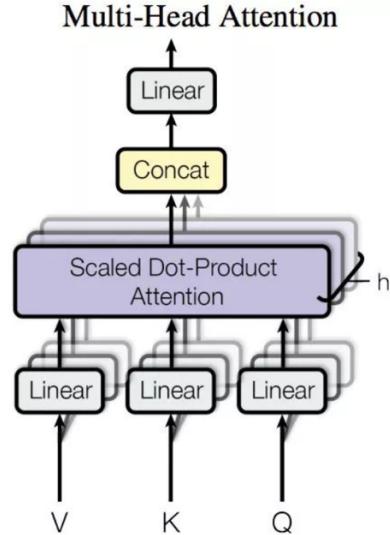
#### 4.4 Multi-head Self-Attention

- 事实上，在 Transformer 中使用的是 Self-Attention 的进阶版本，Multi-head Self-Attention 多头自注意力机制。
- 多头 Attention 注意力机制直观理解为，相关这件事情有很多种不同的形式，有很多种不同的定义的相关性，或许应该不同的  $Q$  负责不同种类的相关性，从而能

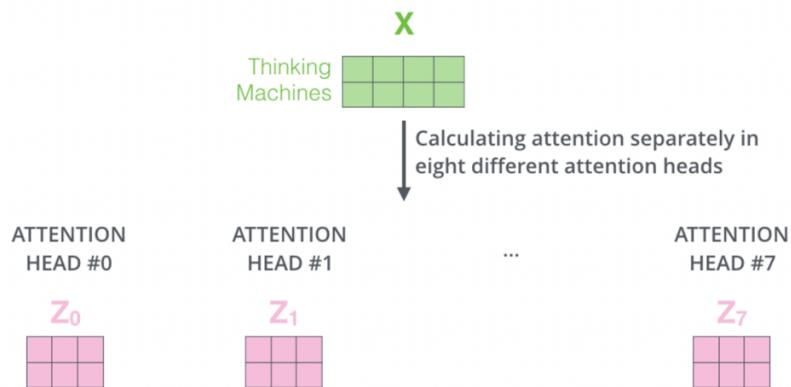
够关注来自不同位置的不同表示子空间的信息。但是，没有机制可以保证不同的 **Attention** 头一定可以捕捉到不同的特征。

- 或者说，把 **Scaled Dot-Product Attention** 的过程做  $h$  次，然后把输出  $Z$  合起来。

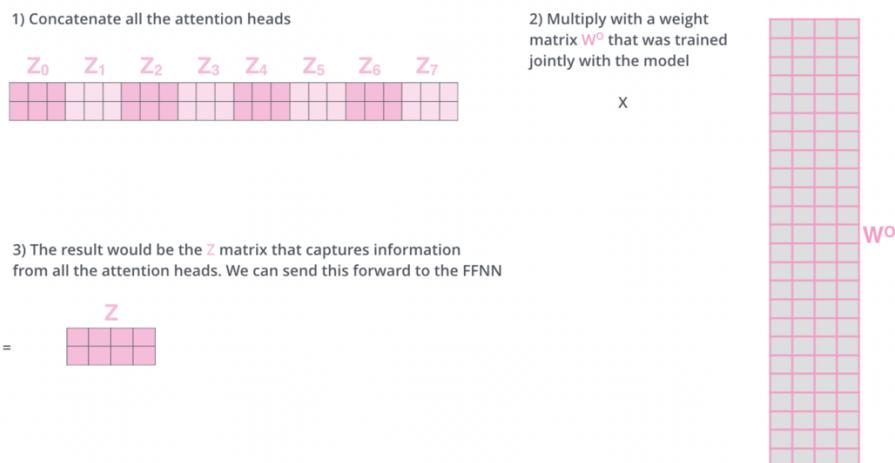
基本结构(以缩放点积为例)如下：

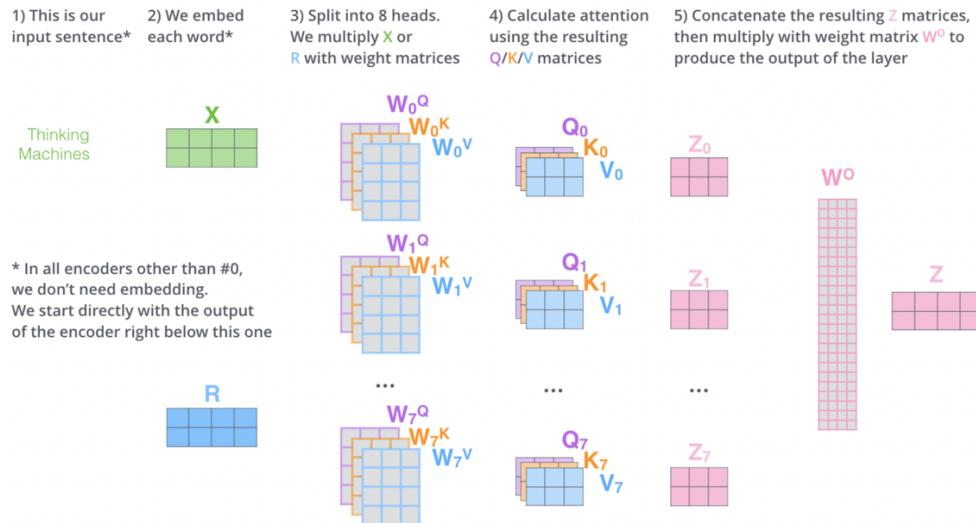


- 可视化过程 (假设采用八个 Attention Head)

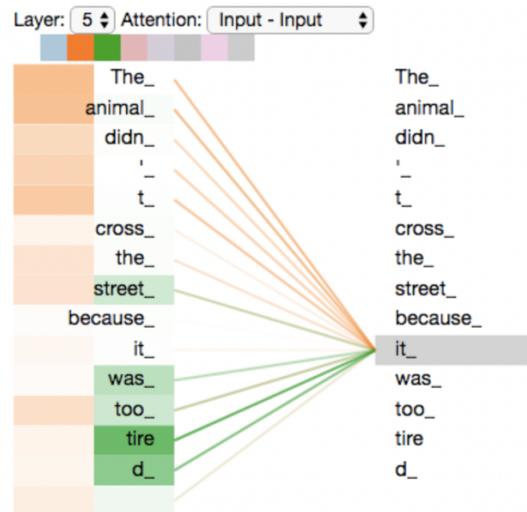


通过一个额外的权重矩阵  $W^0$  将八个  $Z$  矩阵合成为一个。

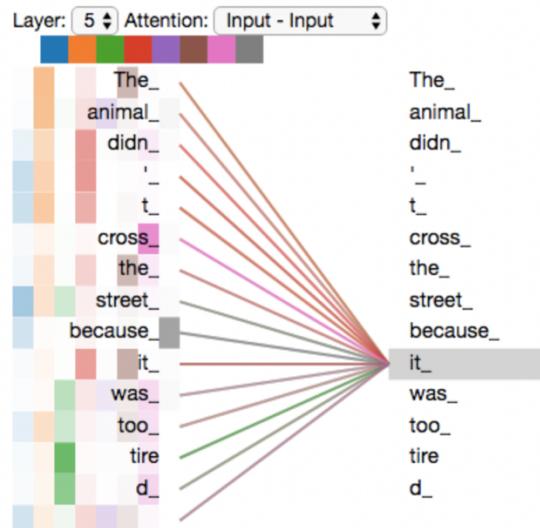




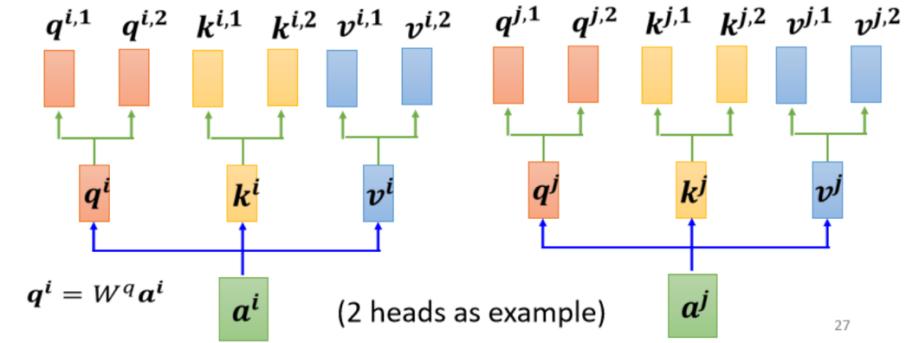
- 继续对之前的例子可视化，可以发现，对“it”这个词进行编码时，一个注意力头最关注“animal”，而另一个注意力头最关注“tire”。即从某种意义上说，模型对“it”这个词的表示包含了一些对“animal”和“tire”的表示。



将所有注意力都添加时：



- 另一种 Multi-head Attention 机制的计算方式（基本原理是一致的）：与上述不同的是，这种方式首先把  $X$  经过一个  $\mathbf{W}_q$  得到  $\mathbf{Q}$ ，然后再乘上另外  $n$  个（head 数）矩阵对  $\mathbf{Q}$  进行分解，再进行 Attention Score 和输出矩阵的计算。



27

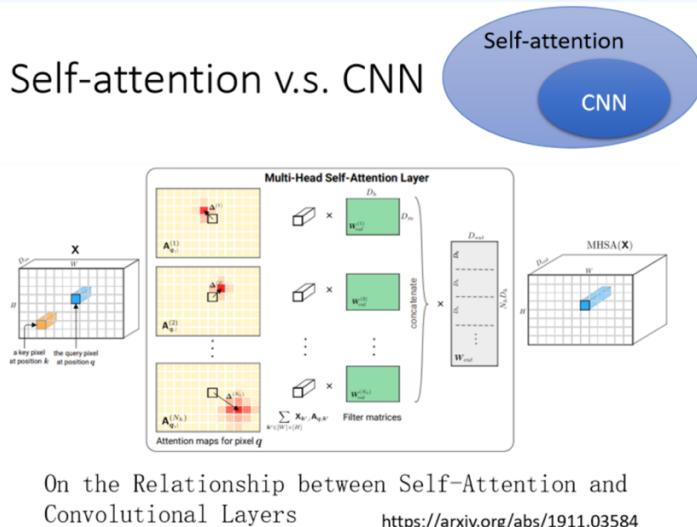
#### 4.5 Self-Attention 的优点和缺陷

- 优点
  - 并行: Multi-Head Attention 和 CNN 一样不依赖前一时刻的计算，可以很好的并行，优于 RNN。
  - 长距离依赖: 优于 Self-Attention 是每个词和所有词计算 Attention，所以不管他们中间有多长距离，最大路径长度都只是 1，可以捕获长距离依赖关系。
- 缺陷
  - 损失了句子内部单词之间的位置信息。
  - 运算量很大。

## 5. Self-Attention 和 CNN, RNN 的关系

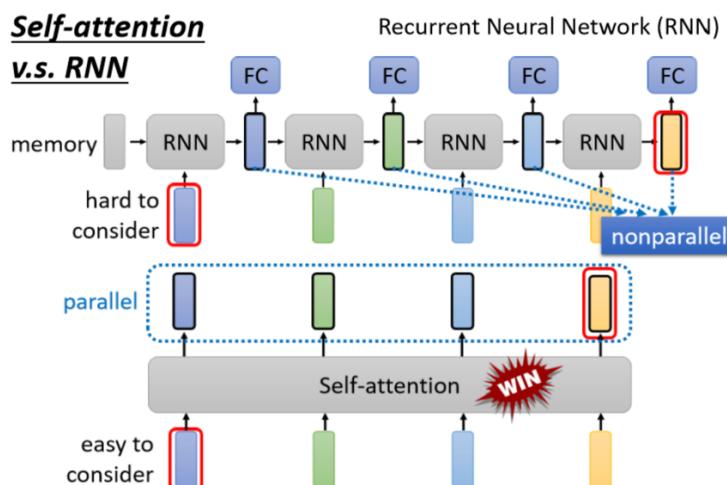
### 5.1 Self-Attention vs. CNN

- CNN 可以看作是一种简化版的 Self-Attention，是 Self-Attention 的特例，CNN 的卷积核对信息获取范围做了限制，只能获得卷积核内（receptive field，感受野）里面的信息，而在做 Self-Attention 的时候，考虑了整张图片的信息。receptive field 的范围跟大小，是人决定的。
- Self-Attention 是一个复杂化的 CNN，用 Attention 去找出重要的 pixel，就像 receptive field 是自动学习的，network 自己学习到 receptive field 的形状，以某个 pixel 为中心，哪些 pixel 是重要的需要被考虑的。



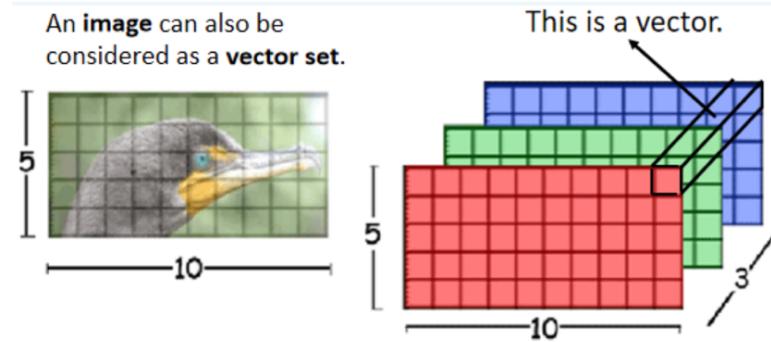
### 5.2 Self-Attention vs. RNN

- Self-Attention 是可并行化的，而 RNN 依赖于上一个输出，是串行机制，所以在运算速度上，Self-Attention 比 RNN 更有效率。
- 此外 RNN 模块，即使是双向的 RNN，或加上了门控机制的 RNN 模型，对很长对序列而言，距离当前 RNN 模块很远的 RNN 模块信息很可能被遗忘或稀释了，丢失很多信息。而 Self-Attention 可以从非常远的 vector，在整个序列上轻易地抽取信息。



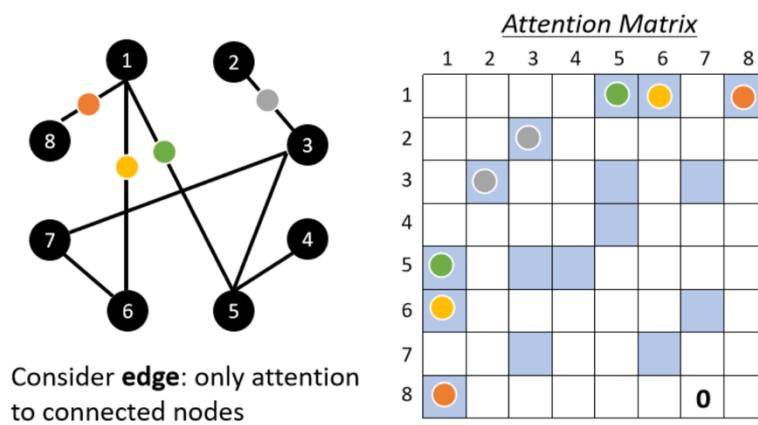
## 6. Self-Attention 对不同数据的应用场景

- 核心思想: 对 vector 集合就可以做 Self-Attention。
- 语音:  
将声音频谱编码成 vector
- 图像:  
可以将同一个像素不同 channel 的值看成一个 vector, 应用 attention 机制



Source of image: [https://www.researchgate.net/figure/Color-image-representation-and-RGB-matrix\\_fig15\\_282798184](https://www.researchgate.net/figure/Color-image-representation-and-RGB-matrix_fig15_282798184)

- Graph:  
node 有 node embedding, 组成 vector set, 可以用 Self-Attention。特别地, 在网络中存在 edge 信息, 指示了哪些 node 之间是相连的, 那么只计算有 edge 相连的 node 就可以, 即如果两个 node 之间没有关系, 则不需要再去计算它的 attention score, 直接设置为 0。



This is one type of **Graph Neural Network (GNN)**.