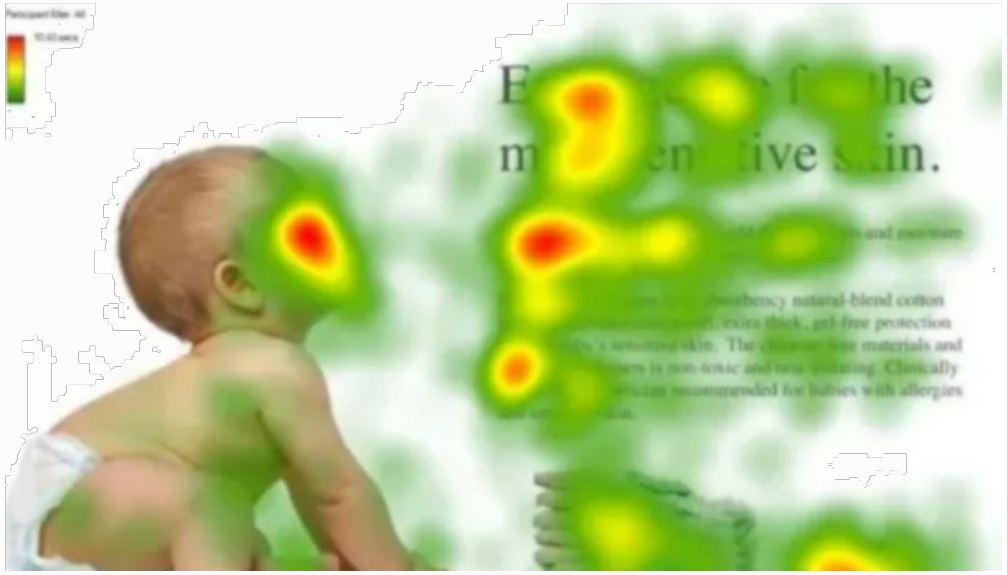
Attention Mechanism: An Overview

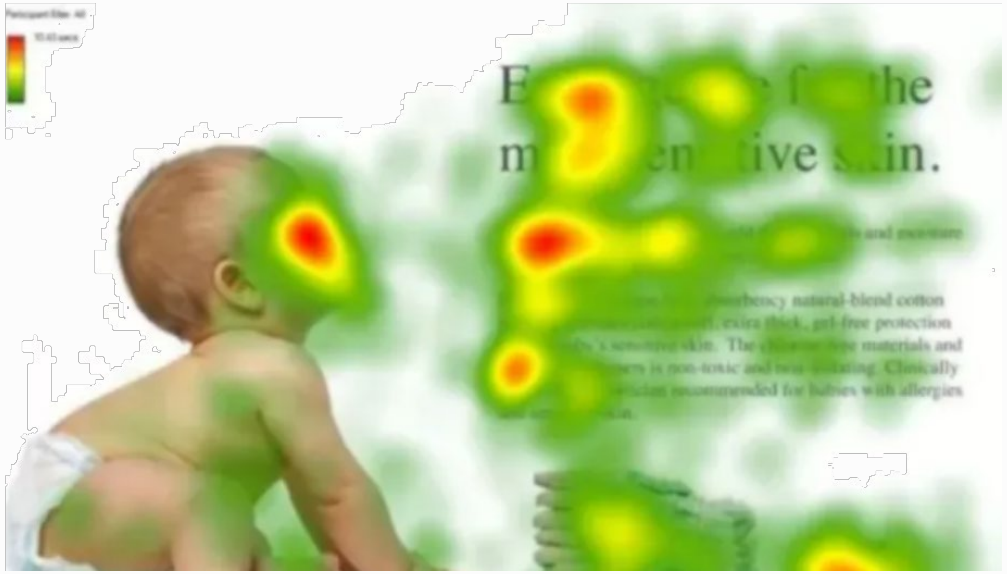
Attention Mechanism

- 注意力机制
- 多头注意力机制
- 自注意力机制
- 多头注意力机制
- Transformer
- Bert



Attention Mechanism

- 注意力机制
- 多头注意力机制
- 自注意力机制
- 多头注意力机制
- Transformer
- Bert



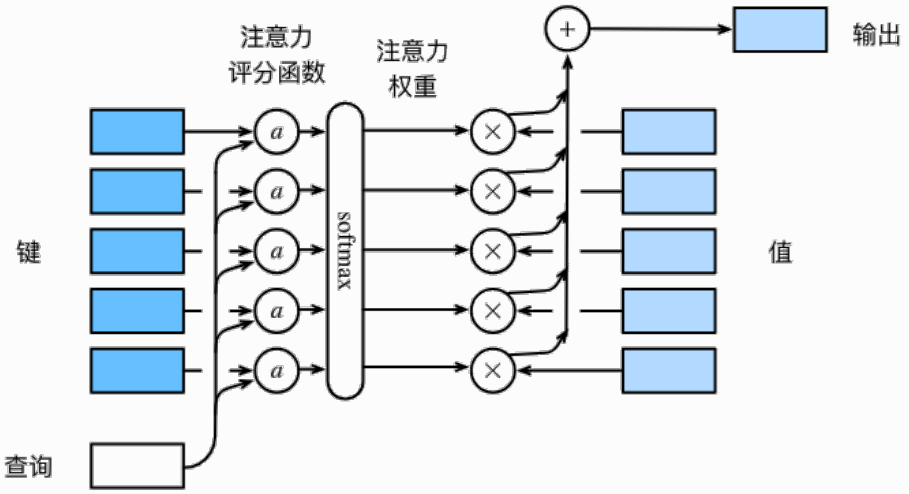
注意力机制

最初的注意力机制，也称为“**注意力分数**”。

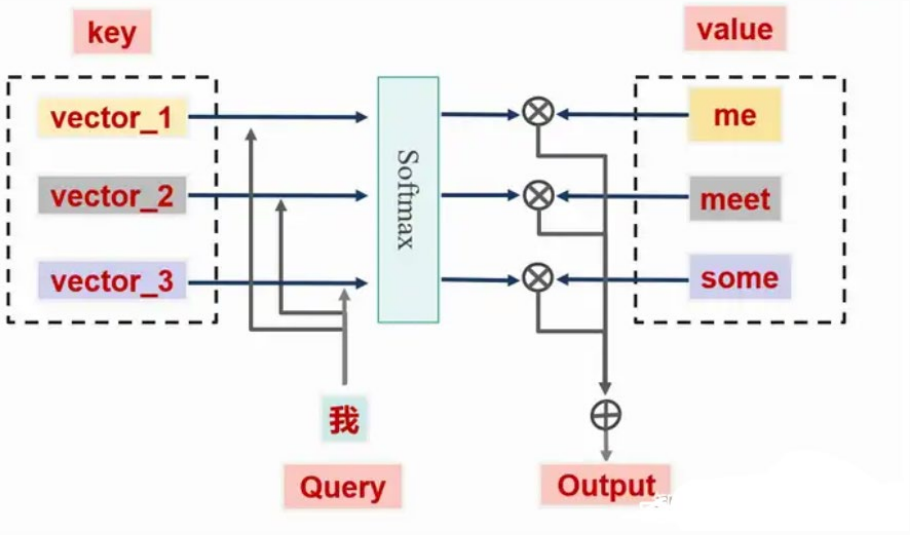
其主要是通过**Query与Key的注意力汇聚**（即，给定一个 Query，计算Query与 Key的相关性，然后根据Query与Key的相关性去和对应的Value进行相乘）；

实现对Value的注意力权重分配，生成最终的输出结果。

总而言是，就是将注意力汇聚的输出计算成为值的加权和。



注意力机制



我们先解释下 Query， Key，和Value 的含义：

Query：我们就可以将"我"看作成 Query，因为这就是我们当前需要查询的目标，即当前输入的特征表示。

Key：可以将每个单词的重要特征表示看作成 Key。

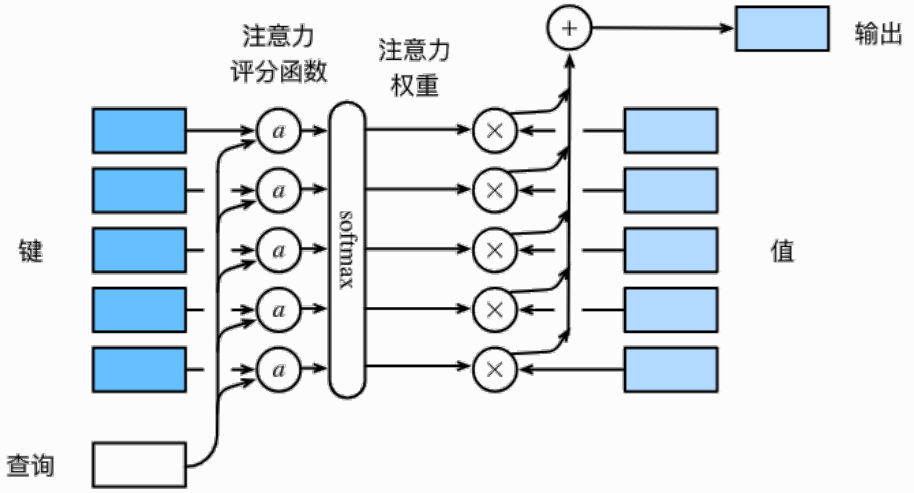
Value：每个单词本身的特征向量看作为 Value，一般和 Key成对出现，也就是我们常说的"键-值"对。

我们需要将中文的"我"翻译成英文的"me"，这就需要**"我"和"me"之间的注意力分数**相对于"我"和其他英文单词的**要高**。

注意力机制

具体操作如下：

（1）先根据 Query，Key计算两者的相关性，然后再通过 softmax 函数得到 注意力分数，使用 softmax 函数是为了使得所有的注意力分数在 [0,1] 之间，并且和为1。这里的重点在于如何计算 Query，Key的相关性，这也是很多论文一个小的创新点所在。Query，Key的相关性公式一般表示如下：



$$score(q,k_i) = softmax(\alpha(q,k_i)) = \frac{exp(\alpha(q,k_i))}{\sum_1^j exp(\alpha(q,k_j))}$$

$\alpha(q,k_i)$ 有很多变体，比如：**加性注意力**、**缩放点积注意力**等等。

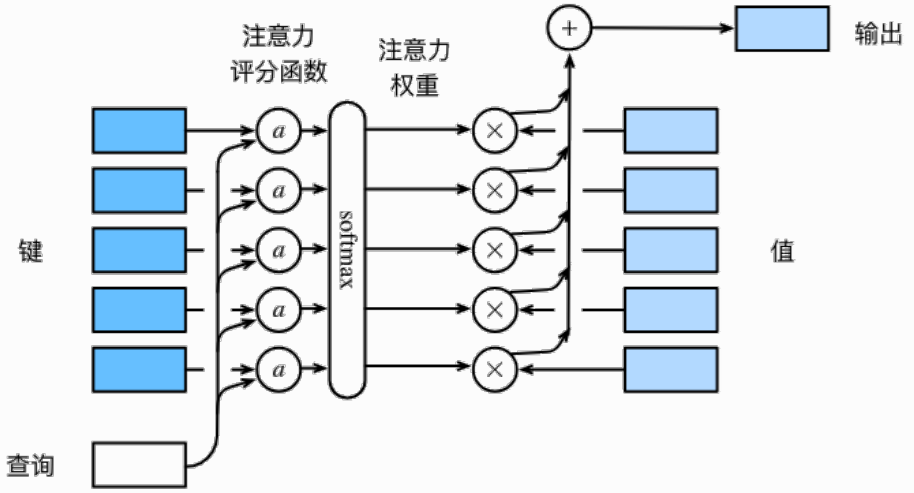
注意力机制

以加性注意力机制为例，

在加性注意力中，主要是将 Query，Key分别乘以对应的可训练矩阵，然后进行相加，具体如下：

$$\alpha(q, k_i) = w_v^T tanh(W_q q + W_k k)$$

其中，W_q，W_k 分别是是 Query，Key对应的可训练矩阵，w_v^T 是 Value对应的可训练矩阵，是为了后面方便和 Value 进行相乘。



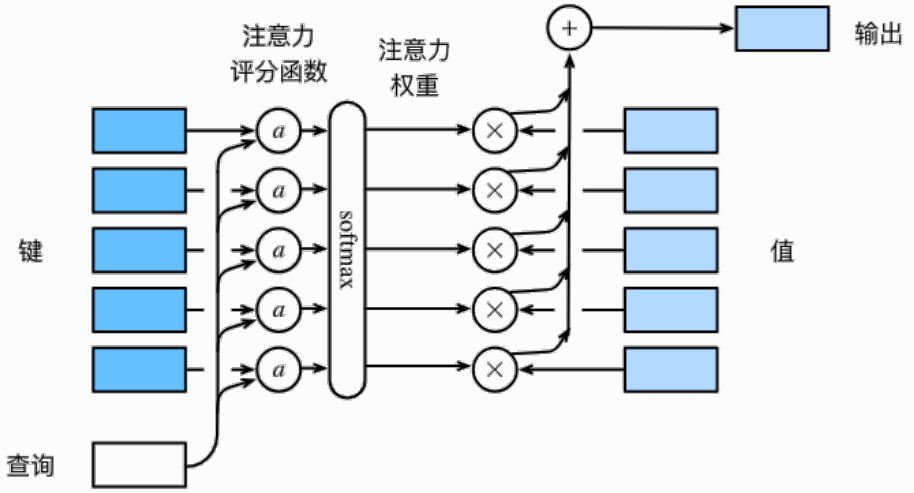
$$score(q, k_i) = softmax(\alpha(q, k_i)) = \frac{exp(\alpha(q,k_i))}{\sum_1^j exp(\alpha(q,k_j))}$$

$\alpha(q, k_i)$ 有很多变体，比如：**加性注意力**、**缩放点积注意力**等等。

注意力机制

以**缩放点积注意力机制**为例，

在加性注意力中，主要是将 Query，Key分别乘以对应的可训练矩阵，然后进行相加，具体如下：



$$\alpha(q, k_i) = \frac{QK^T}{\sqrt{d}}$$

$$score(q, k_i) = softmax(\alpha(q, k_i)) = \frac{exp(\alpha(q,k_i))}{\sum_1^j exp(\alpha(q,k_j))}$$

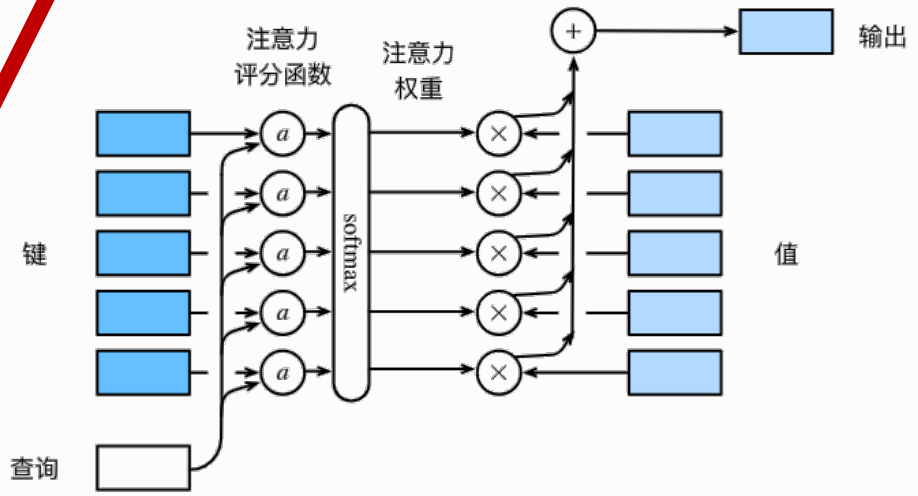
$\alpha(q, k_i)$ 有很多变体，比如：**加性注意力**、**缩放点积注意力**等等。

注意力机制

我们可以反过来想，如果直接将不带注意力分数的 V 进行输入到下游任务，下游任务可能会认为所有单词的重要性程度都是一样的，或者随机将不相关的单词与 Query 进行匹配。

（2）根据注意力分数进行加权求和，得到带注意力分数的 Value，以方便进行下游任务。

$$Output = score(Q, K)V$$



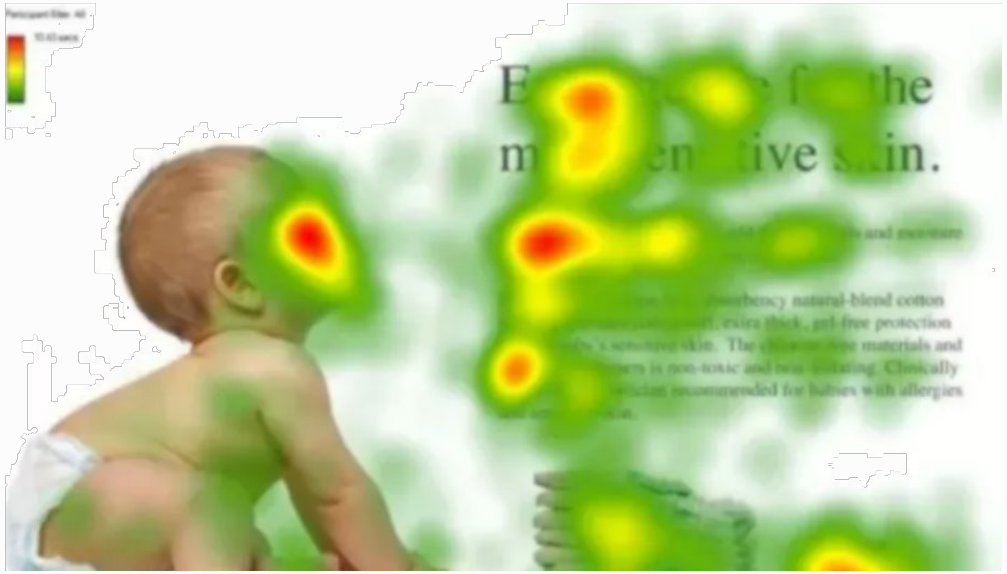
在（1）中，我们得到了Query，Key的相关性，如果相关性越大，注意力分数就越高，反之越低；然后将注意力分数乘以对应的 Value，再进行加权求和；就比如："我"和"me"的相关性较大，注意力分数就会越高；这样可以让下游任务理解"我"和"me"是匹配程度高。

$$score(q, k_i) = softmax(\alpha(q, k_i)) = \frac{exp(\alpha(q,k_i))}{\sum_1^j exp(\alpha(q,k_j))}$$

$\alpha(q, k_i)$ 有很多变体，比如：加性注意力[+]、缩放点积注意力等等。

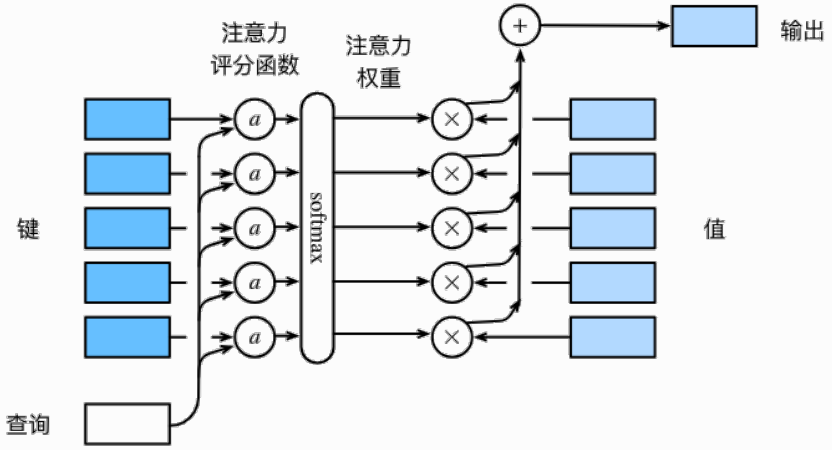
Attention Mechanism

- 注意力机制
- 多头注意力机制
- 自注意力机制
- 多头注意力机制
- Transformer
- Bert

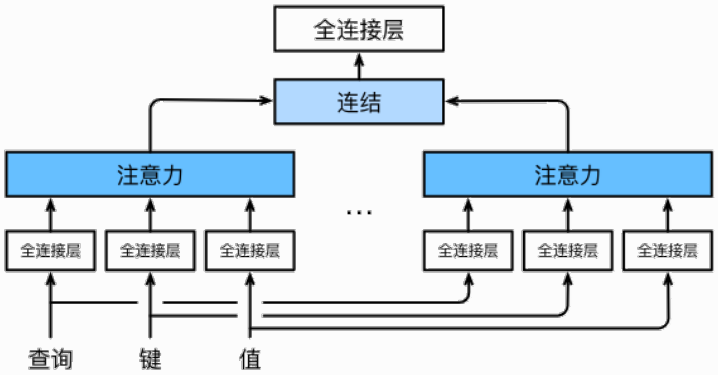


多头注意力机制

注意力机制



多头注意力机制



与其只使用单独一个注意力汇聚， 我们可以用独立学习得到的多组不同的线性投影（linear projections）来变换查询、键和值。

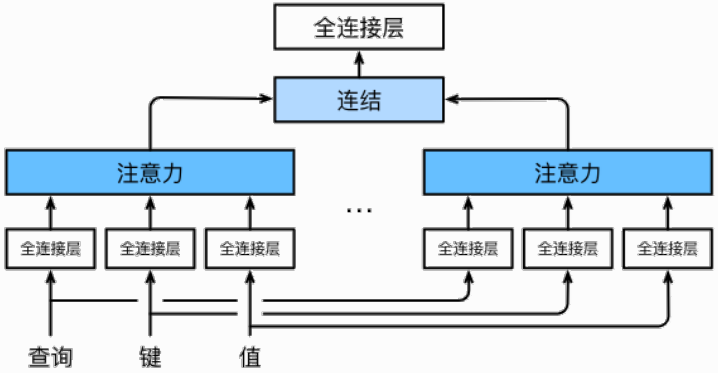
这也是为什么多头注意力为什么可以并行计算的原因！！

多头注意力机制

以加性注意力机制为例

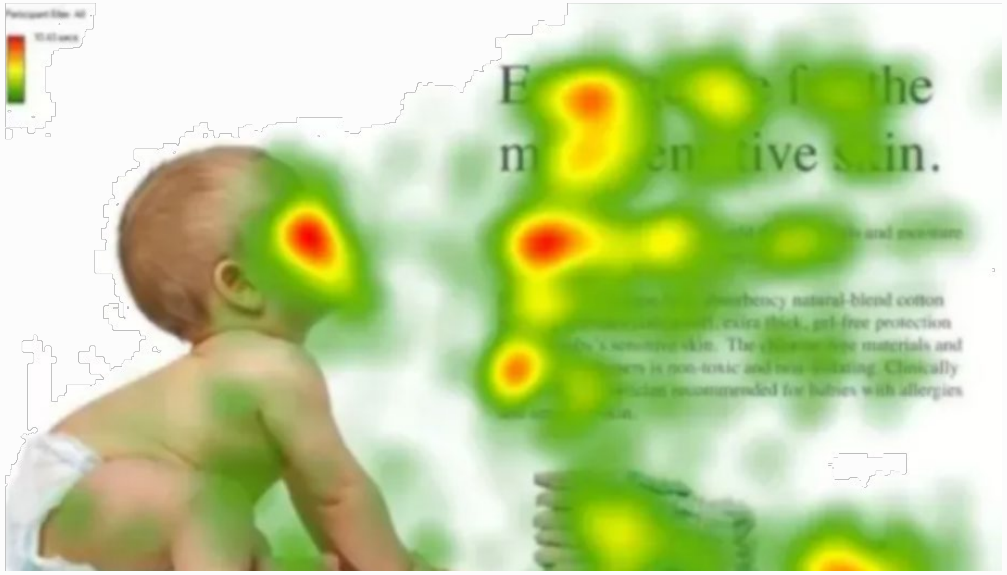
$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)}\mathbf{q}, \mathbf{W}_i^{(k)}\mathbf{k}, \mathbf{W}_i^{(v)}\mathbf{v}) \in \mathbb{R}^{p_v},$$

多头注意力机制



Attention Mechanism

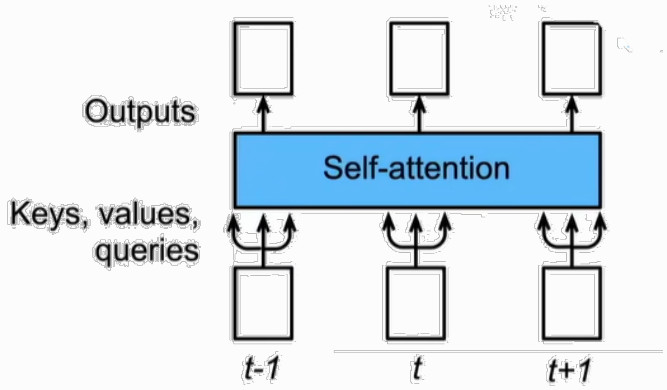
- 注意力机制
- 多头注意力机制
- **自注意力机制**
- 多头注意力机制
- 通道注意力机制
- 空间注意力机制



自注意力机制

同一组词元同时充当查询、键和值。

具体来说，每个查询都会关注所有的键 - 值对并生成一个注意力输出。 由于查询、键和值来自同一组输入，因此被称为 自注意力（self-attention）

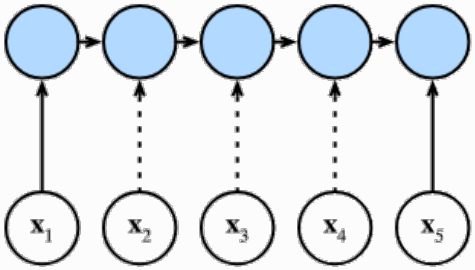


$$\mathbf{y}_i = f(\mathbf{x}_i, (\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_n, \mathbf{x}_n)) \in \mathbb{R}^d$$

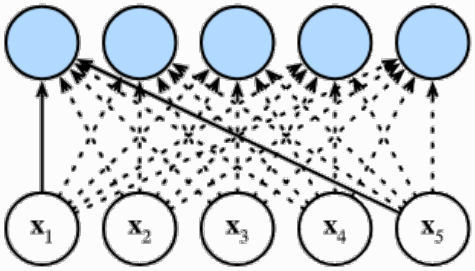
Query　Key　Value

自注意力机制

循环神经网络



自注意力



	RNN	自注意力
计算复杂度	O(nd^2)	O(n^2d)
并行度	O(1)	O(n)
最长路径	O(n)	O(1)

自注意力机制

$$\mathbf{y}_i = f(\mathbf{x}_i, (\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_n, \mathbf{x}_n)) \in \mathbb{R}^d$$

Query

Key

Value

跟CNN和RNN不一样的是，自注意力机制并没有记录位置信息。

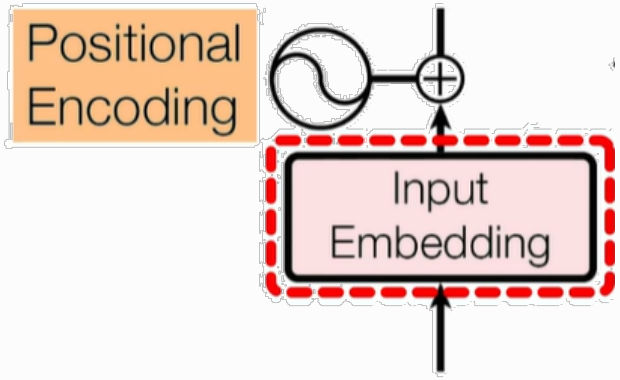
如何在自注意力机制中加入位置信息?

自注意力机制

位置编码

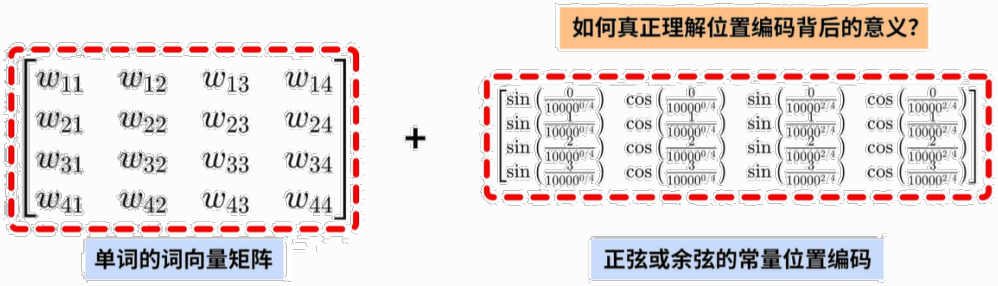
不同于RNN，位置编码或者位置信息并不是直接加载模型上或者把位置信息和数据分开；

而是**直接将位置信息放入输入中**！！



自注意力机制

位置编码



而是直接将位置信息放入输入中！！

会存在几个问题：

1. **将位置信息加到词向量上，是否会破坏词向量本身的信息?**
2. **加上位置信息的词向量，还能表达出原有词向量所表达的语意信息?**
3. **是否会破坏原有词向量的数据分布?**

自注意力机制

位置编码

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

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

位置编码的计算公式

维度0 维度1 维度2 维度3

$$\begin{bmatrix} \sin\left(\frac{0}{10000^{0/4}}\right) & \cos\left(\frac{0}{10000^{0/4}}\right) & \sin\left(\frac{0}{10000^{2/4}}\right) & \cos\left(\frac{0}{10000^{2/4}}\right) \\ \sin\left(\frac{1}{10000^{0/4}}\right) & \cos\left(\frac{1}{10000^{0/4}}\right) & \sin\left(\frac{1}{10000^{2/4}}\right) & \cos\left(\frac{1}{10000^{2/4}}\right) \\ \sin\left(\frac{2}{10000^{0/4}}\right) & \cos\left(\frac{2}{10000^{0/4}}\right) & \sin\left(\frac{2}{10000^{2/4}}\right) & \cos\left(\frac{2}{10000^{2/4}}\right) \\ \sin\left(\frac{3}{10000^{0/4}}\right) & \cos\left(\frac{3}{10000^{0/4}}\right) & \sin\left(\frac{3}{10000^{2/4}}\right) & \cos\left(\frac{3}{10000^{2/4}}\right) \end{bmatrix}$$

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

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

其中，d_{model} 是 input Embedding嵌入向量的维度，*pos* 是单词在序列中的位置，*i* 是嵌入向量中的维度索引。

位置编码和词向量具有相同的维度！！

自注意力机制

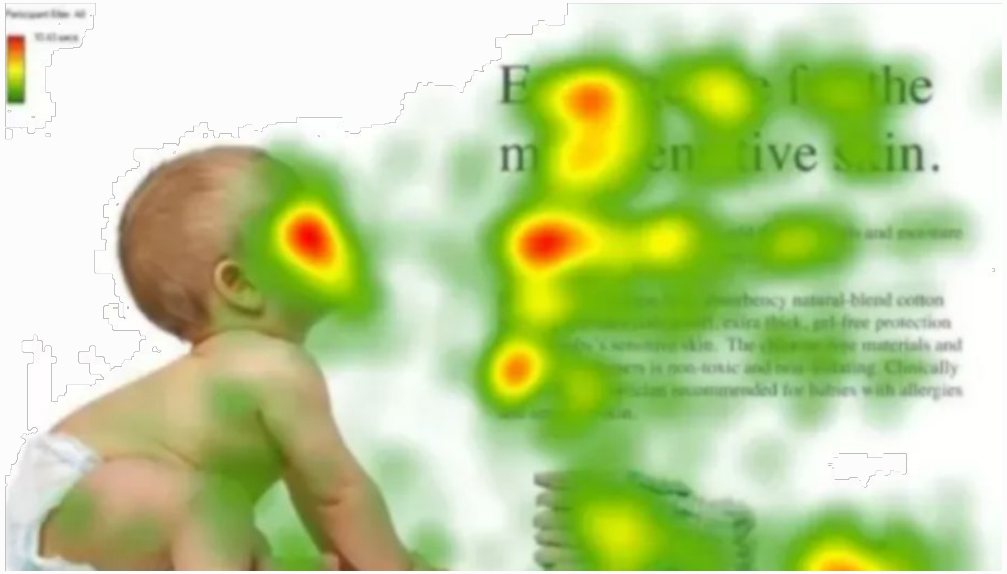
位置编码

将位置信息加到词向量上，是否会破坏词向量本身的信息？

不会的。一般NLP领域，是具有绝对充足的训练数据，训练数据的规模足以支撑对“词向量 + 位置编码”这种复合特征进行深层次的区分和理解。

Attention Mechanism

- 注意力机制
- 多头注意力机制
- 自注意力机制
- **多头注意力机制**
- Transformer
- Bert

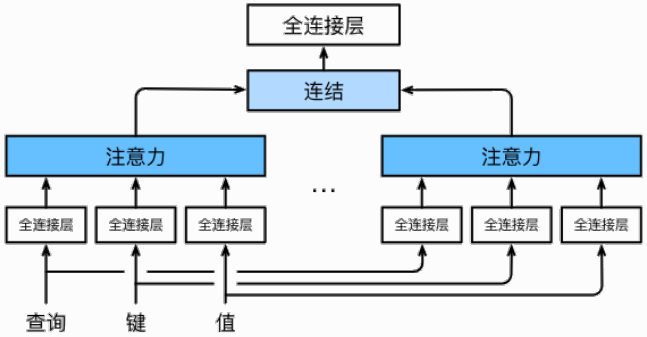


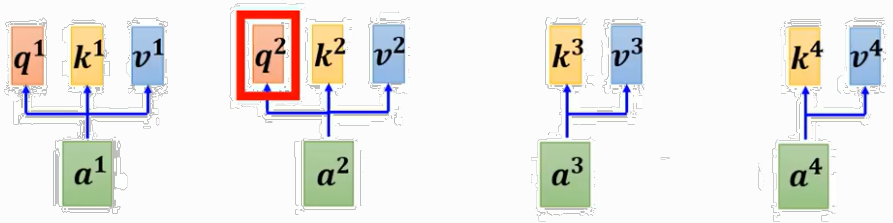
多头自注意力机制

多头注意力机制是在自注意力机制的基础上发展起来的，是自注意力机制的变体，旨在增强模型的表达能力和泛化能力。

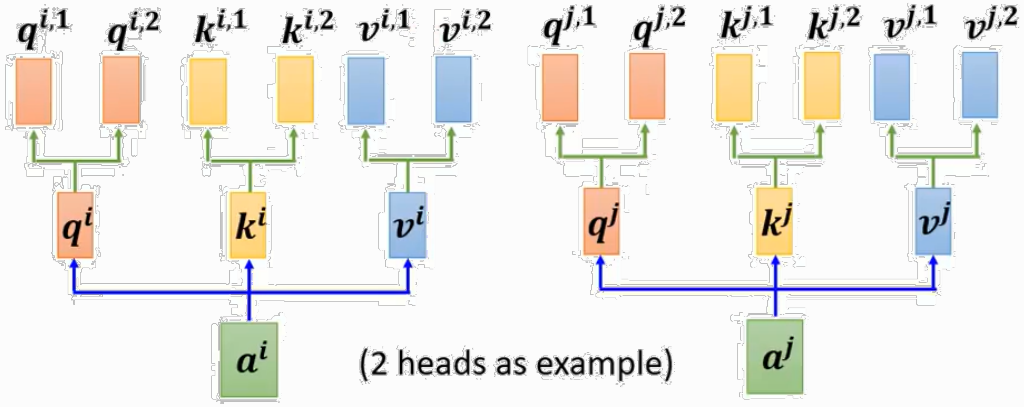
它通过使用多个独立的注意力头，分别计算注意力权重，并将它们的结果进行拼接或加权求和，从而获得更丰富的表示。

多头注意力机制

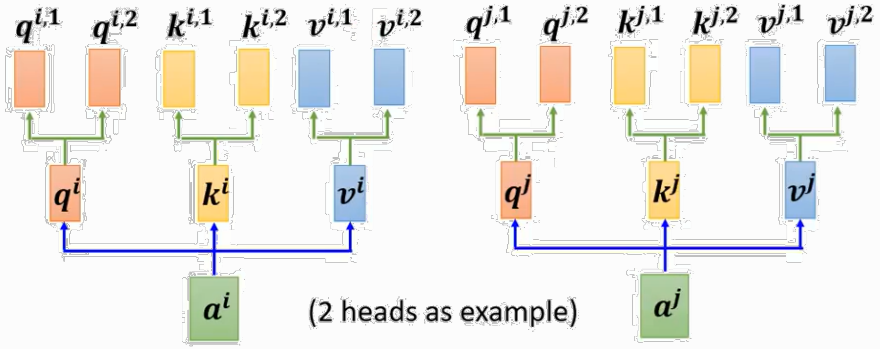




多头自注意力机制



多头自注意力机制



在多头注意力机制中，a^i 会先乘 q 矩阵，$q^i = W^q a^i$；

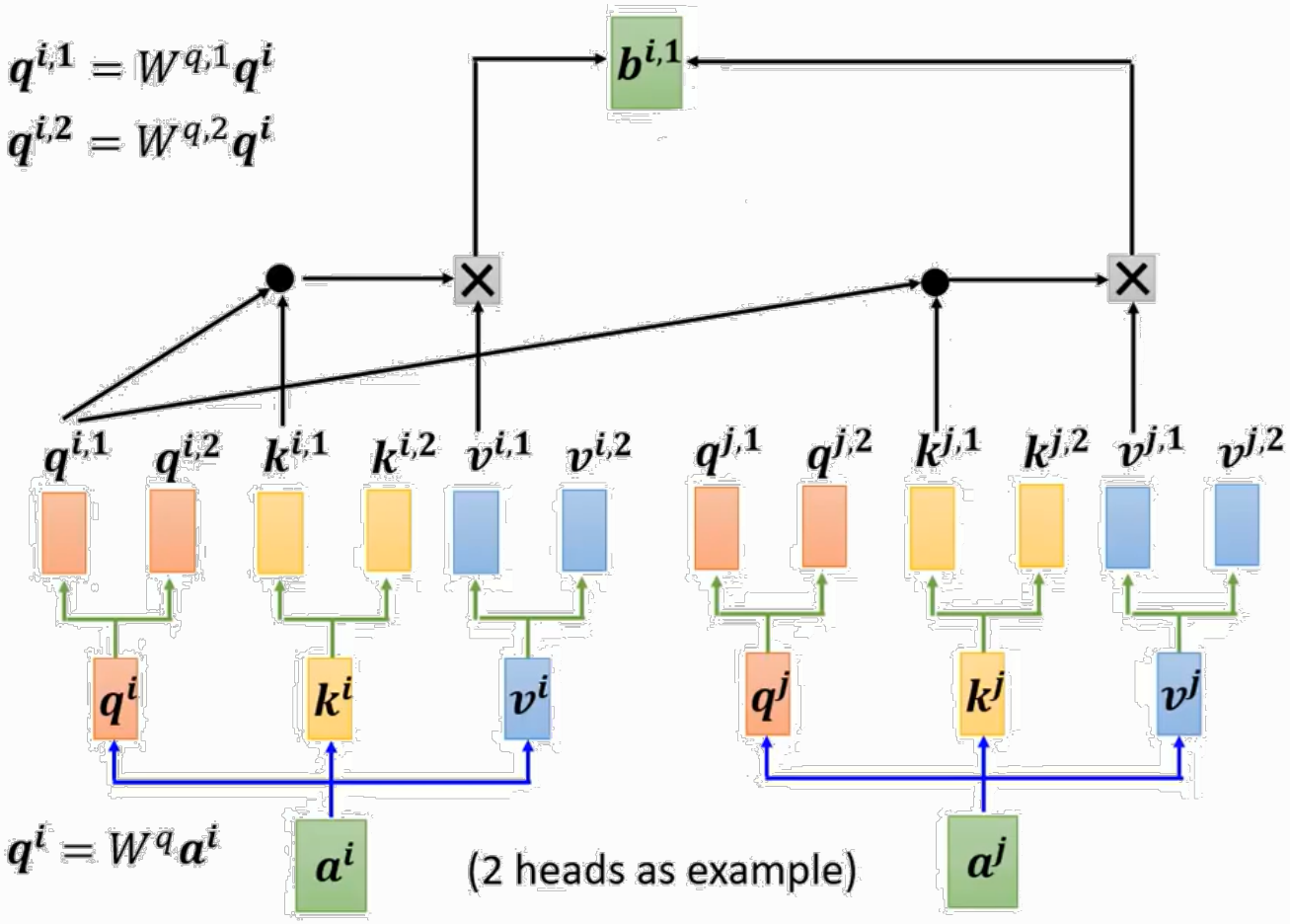
其次，会为其多分配两个head，以 q 为例，包括：$q^{i,1}, q^{i,2}$；

$$q^{i,1} = W^{q,1} q^i$$

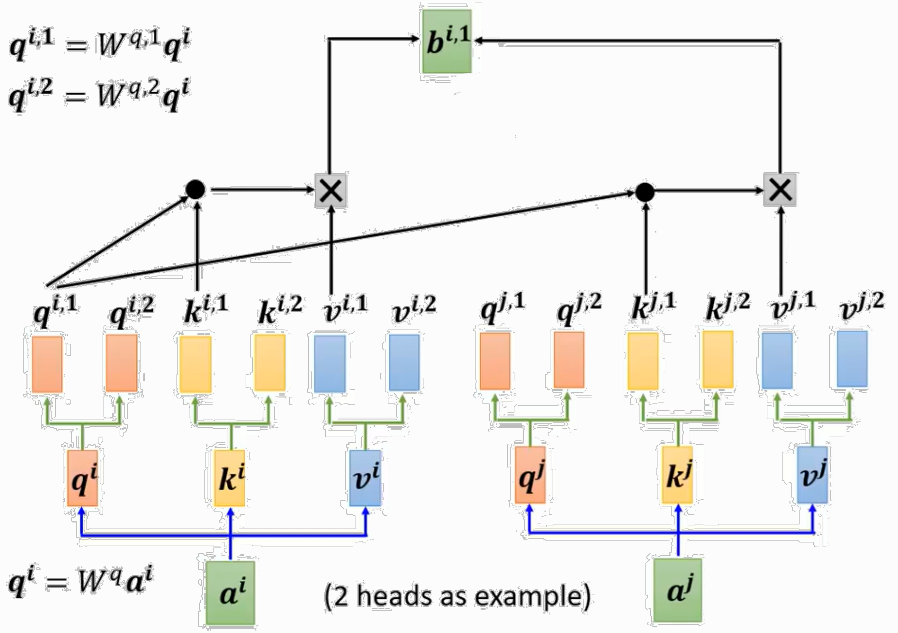
$$q^{i,2} = W^{q,2} q^i$$

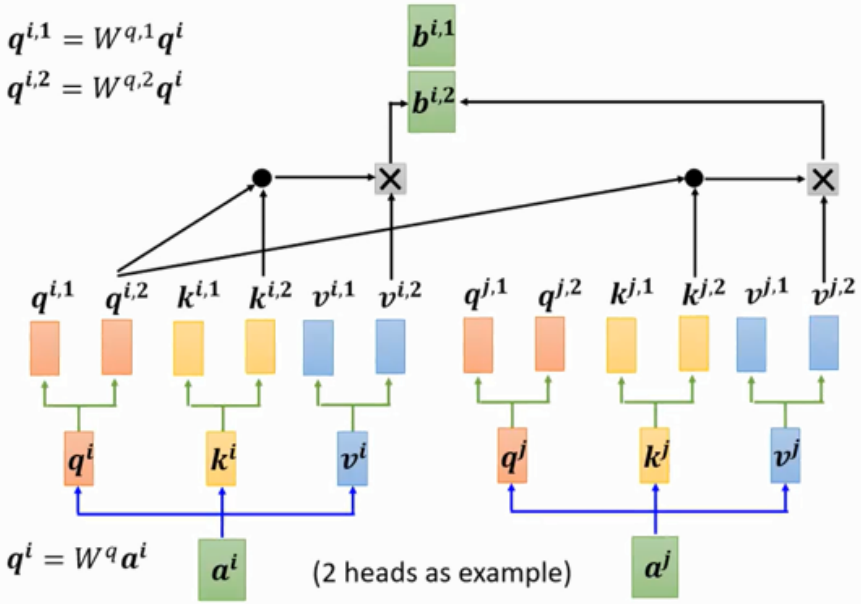
同样地，k 和 v 也是一样的操作。

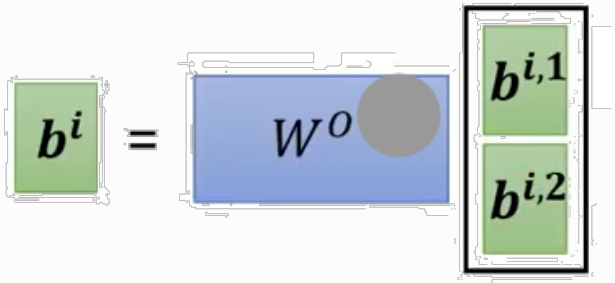
多头自注意力机制


$q^{i,1} = W^{q,1} q^i$
$q^{i,2} = W^{q,2} q^i$
$b^{i,1}$
$q^{i,1}$ $q^{i,2}$ $k^{i,1}$ $k^{i,2}$ $v^{i,1}$ $v^{i,2}$
$q^{j,1}$ $q^{j,2}$ $k^{j,1}$ $k^{j,2}$ $v^{j,1}$ $v^{j,2}$
q^i k^i v^i
q^j k^j v^j
$q^i = W^q a^i$
a^i
(2 heads as example)
a^j

多头自注意力机制

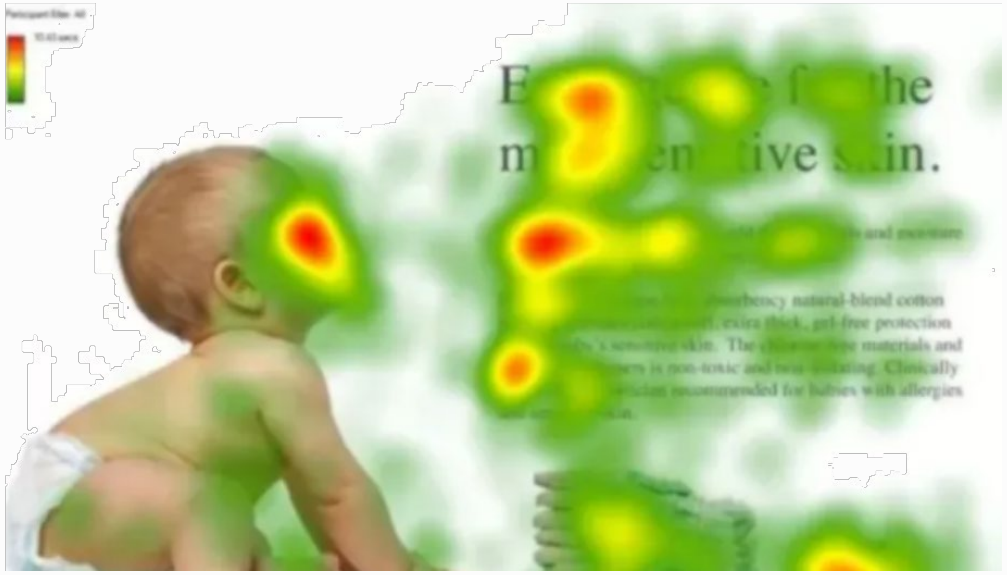

$q^{i,1} = W^{q,1} q^i$
$q^{i,2} = W^{q,2} q^i$
$b^{i,1}$
$q^{i,1}$ $q^{i,2}$ $k^{i,1}$ $k^{i,2}$ $v^{i,1}$ $v^{i,2}$ $q^{j,1}$ $q^{j,2}$ $k^{j,1}$ $k^{j,2}$ $v^{j,1}$ $v^{j,2}$
q^i k^i v^i q^j k^j v^j
$q^i = W^q a^i$
a^i
(2 heads as example)
a^j


$q^{i,1} = W^{q,1} q^i$
$q^{i,2} = W^{q,2} q^i$
$b^{i,1}$
$b^{i,2}$
$q^{i,1}$ $q^{i,2}$ $k^{i,1}$ $k^{i,2}$ $v^{i,1}$ $v^{i,2}$ $q^{j,1}$ $q^{j,2}$ $k^{j,1}$ $k^{j,2}$ $v^{j,1}$ $v^{j,2}$
q^i k^i v^i q^j k^j v^j
$q^i = W^q a^i$
a^i
(2 heads as example)
a^j


b^i
=
W^O
$b^{i,1}$
$b^{i,2}$

Attention Mechanism

- 注意力机制
- 多头注意力机制
- 自注意力机制
- 多头注意力机制
- Transformer
- Bert



Transformer

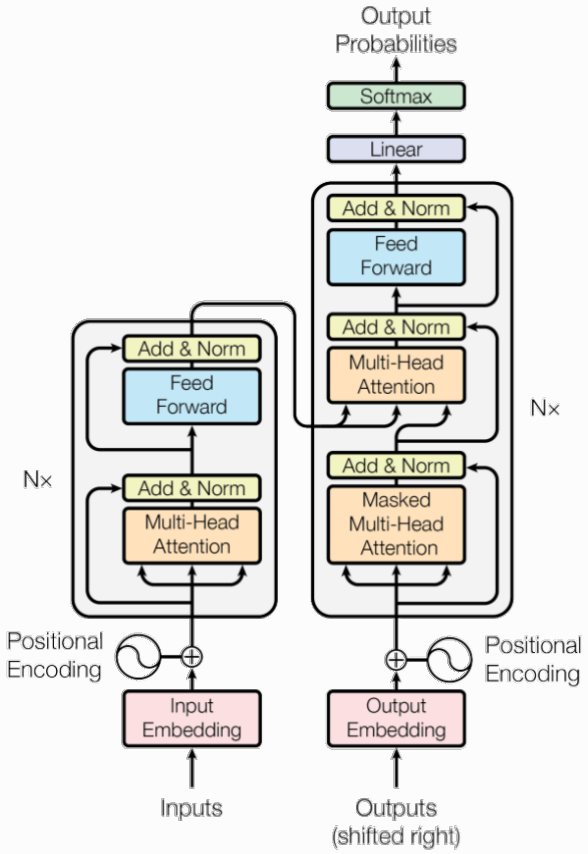
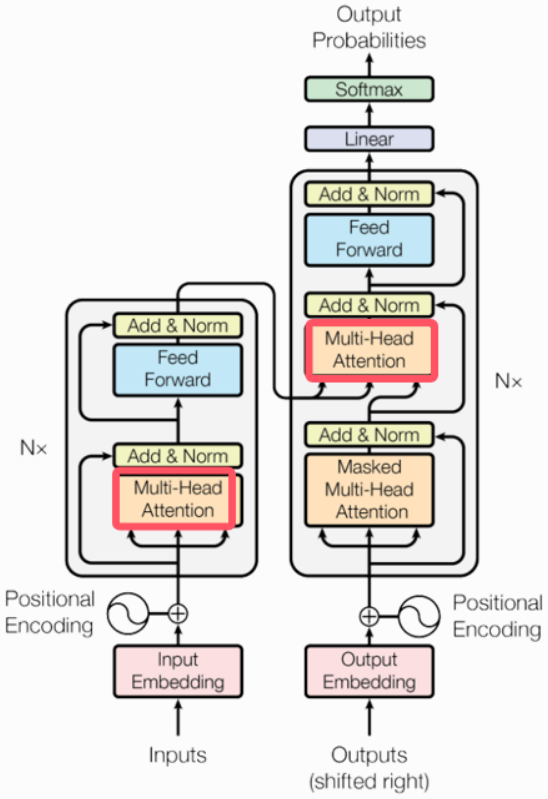
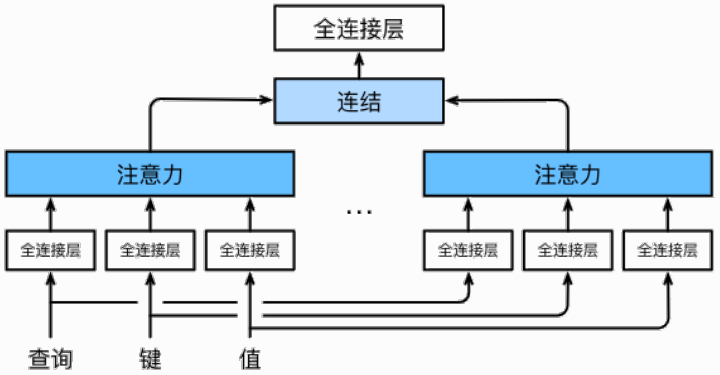


Figure 1: The Transformer - model architecture.

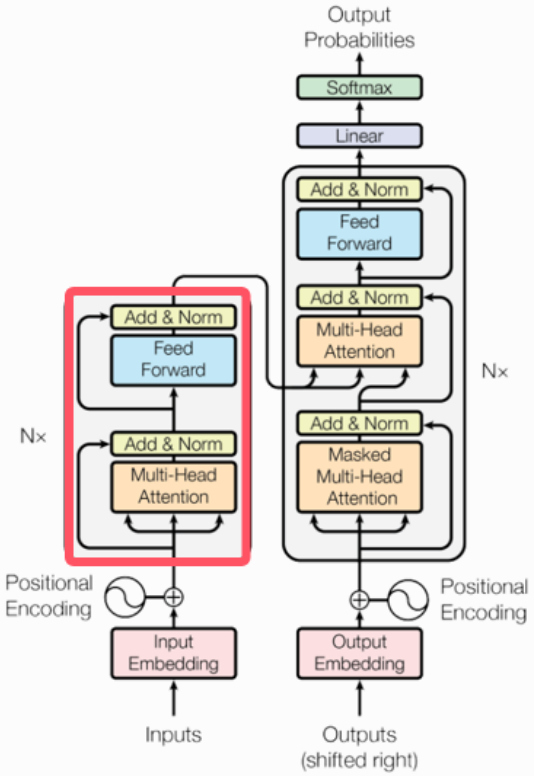
Transformer

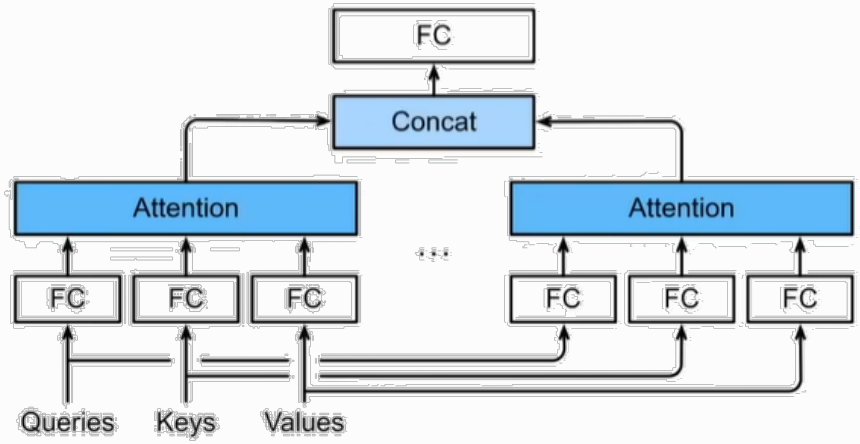




$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)}\mathbf{q}, \mathbf{W}_i^{(k)}\mathbf{k}, \mathbf{W}_i^{(v)}\mathbf{v}) \in \mathbb{R}^{p_v},$$

Transformer

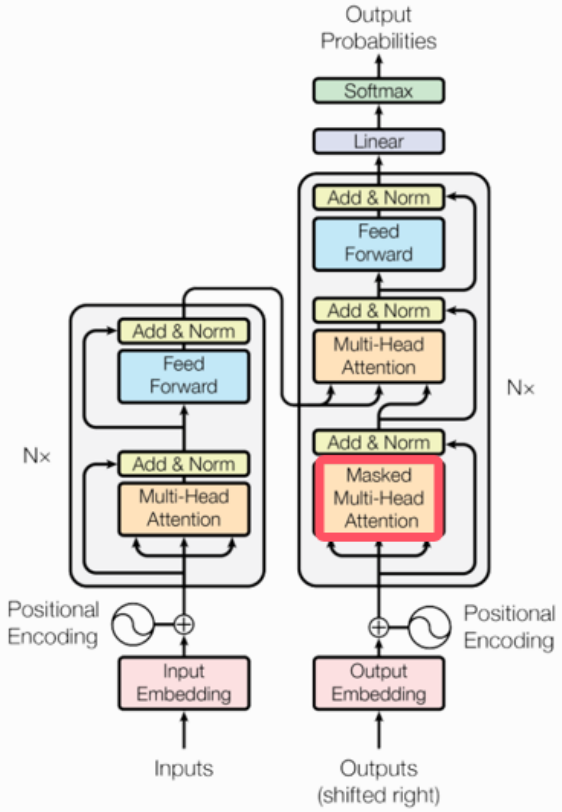




- query $\mathbf{q} \in \mathbb{R}^{d_q}$, key $\mathbf{k} \in \mathbb{R}^{d_k}$, value $\mathbf{v} \in \mathbb{R}^{d_v}$
- 头 i 的可学习参数 $\mathbf{W}_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$, $\mathbf{W}_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$, $\mathbf{W}_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$
- 头 i 的输出 $\mathbf{h}_i = f(\mathbf{W}_i^{(q)}\mathbf{q}, \mathbf{W}_i^{(k)}\mathbf{k}, \mathbf{W}_i^{(v)}\mathbf{v}) \in \mathbb{R}^{p_v}$
- 输出的可学习参数 $\mathbf{W}_o \in \mathbb{R}^{p_o \times hp_v}$
- 多头注意力的输出

$$\mathbf{W}_o \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_h \end{bmatrix} \in \mathbb{R}^{p_o}$$

Transformer

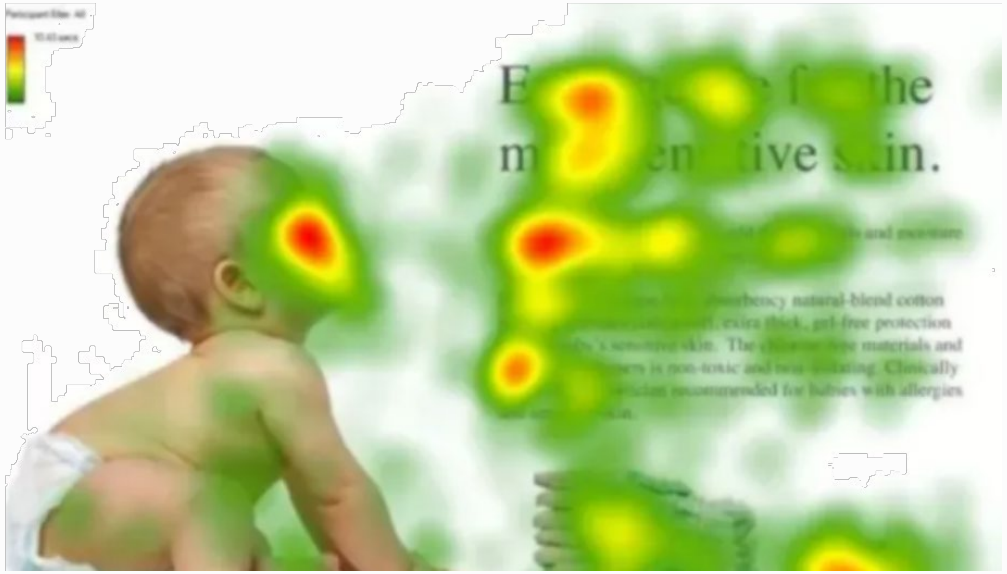


有掩码的多头注意力

- 解码器对序列中一个元素输出时，不应该考虑该元素之后的元素
- 可以通过掩码来实现
  - 也就是计算 $\mathbf{x}_i$ 输出时，假装当前序列长度为 i

Attention Mechanism

- 注意力机制
- 多头注意力机制
- 自注意力机制
- 多头注意力机制
- Transformer
- **Bert**



Bert

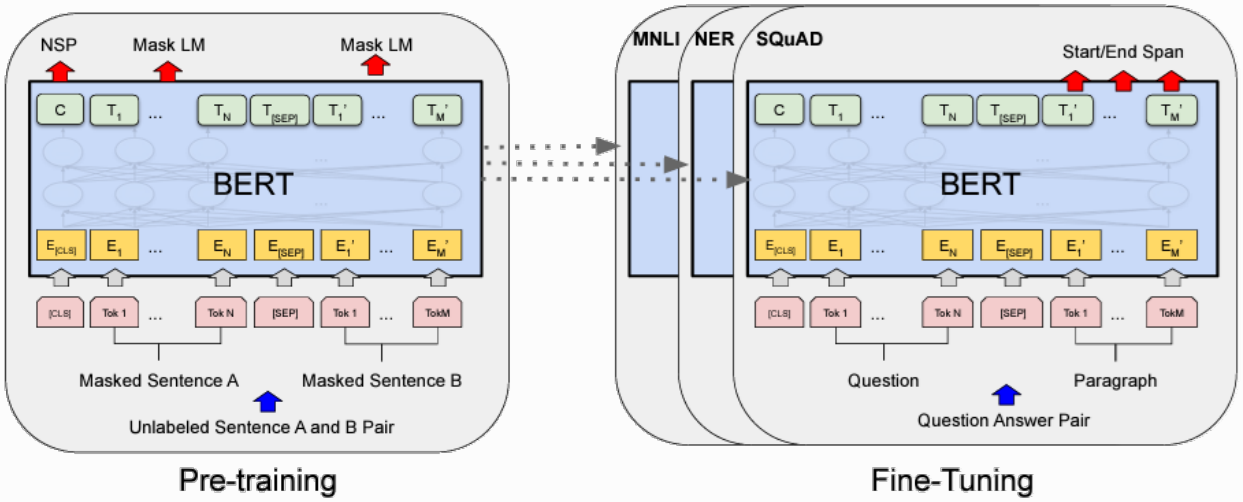


Figure 1: Overall pre-training and fine-tuning procedures for BERT. Apart from output layers, the same architectures are used in both pre-training and fine-tuning. The same pre-trained model parameters are used to initialize models for different down-stream tasks. During fine-tuning, all parameters are fine-tuned. [CLS] is a special symbol added in front of every input example, and [SEP] is a special separator token (e.g. separating questions/answers).

只有编码器的Transformer

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

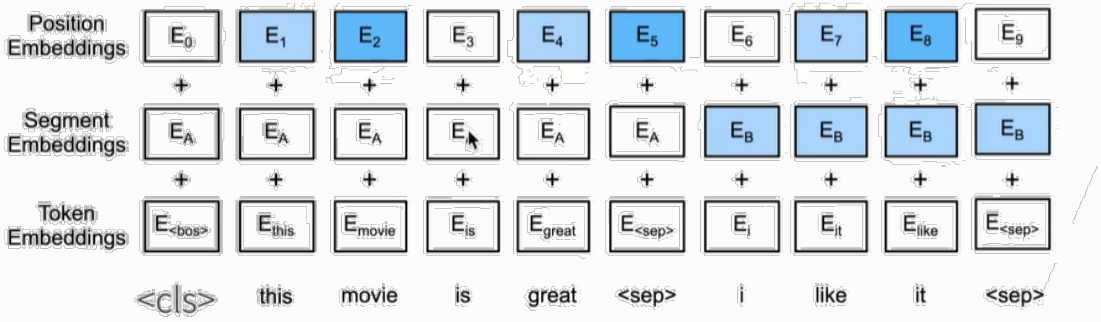
Bert

- 只有编码器的 Transformer
- 两个版本：
  - Base: #blocks = 12, hidden size = 768, #heads = 12, #parameters = 110M
  - Large: #blocks = 24, hidden size = 1024, #heads = 16, #parameter = 340M
- 在大规模数据上训练 > 3B 词

Bert

对输入的修改

- 每个样本是一个句子对
- 加入额外的片段嵌入
- 位置编码可学习



预训练任务1：带掩码的语言模型

- Transfomer的编码器是双向，标准语言模型要求单向
- 带掩码的语言模型每次随机（15%概率）将一些词元换成<mask>
- 因为微调任务中不出现<mask>
  - 80%概率下，将选中的词元变成<mask>
  - 10%概率下换成一个随机词元
  - 10%概率下保持原有的词元

预训练任务2：下一句子预测

- 预测一个句子对中两个句子是不是相邻
- 训练样本中：
  - 50%概率选择相邻句子对：<cls> this movie is great <sep> i like it <sep>
  - 50%概率选择随机句子对：<cls> this movie is gr hello world <sep>
- 将<cls>对应的输出放到一个全连接层来预测

总结

- BERT针对微调设计
- 基于Transformer的编码器做了如下修改
  - 模型更大，训练数据更多
  - 输入句子对，片段嵌入，可学习的位置编码
  - 训练时使用两个任务：
    - 带掩码的语言模型
    - 下一个句子预测