Transformer学习笔记（待改进）

1. 前言

本文档用于总结记录transformer架构学习过程中的关键点和笔记，加深理解避免遗忘。

原论文Attention is All You Need：https://arxiv.org/abs/1706.03762，由此可知，transfortmer是基于注意力机制上的突破而发展得来。

对新的架构进行学习前，首先和相关领域的其它架构进行对比分析优劣：

特性	RNN/LSTM	Transformer
并行能力	串行处理，后续的输入依赖于之前的输出	并行处理整个序列，训练速度快
远距离依赖	难以捕获长序列中的远距离依赖	通过自注意力机制能有效捕获长距离依赖，并且引入multi-head能够适应不同距离的依赖
计算量	计算量的增长和序列长度呈线性关系	计算量与序列长度呈平方关系
位置信息	按顺序串行处理，已经内置了位置信息	需要引入额外的位置编码

2. Transformer核心结构

Transformer和RNN的N2M结构的输入输出上类似，都是seq2seq结构的模型，但不同的地方在于，前者的模型中大量使用了"Self-Attention"这种特别的layer，即transformer的核心机制所在，后将续展开讲解(Transformer~~Seq2Seq Model with "Self-Attention")。

Self-Attention

计算步骤

步骤一：计算得到qkv

输入数据经过一系列操作进行编码和序列化后输入到Self-Attention Layer，如下图所示：

其中，a即为经过处理后的输入，其计算公式如下：

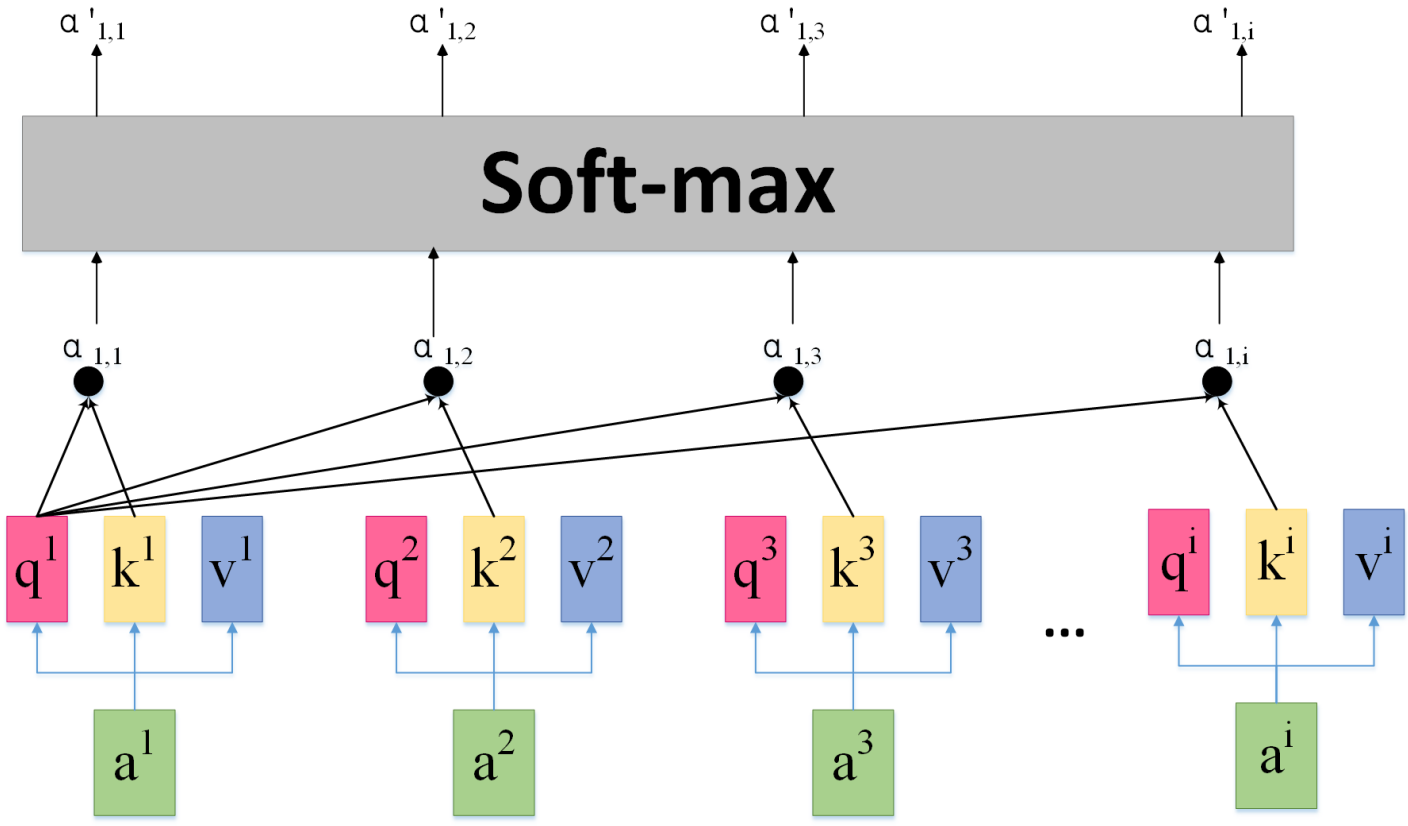
$$q^i = W^q a^i \qquad k^i = W^k a^i \qquad v^i = W^v a^i$$

q、k、v三个变量的解释如下：

- q：query(to match others)，用于匹配
- k：key(to be matched)，用于被匹配
- v：value(information to be extracted)，被抽取出来的信息

步骤二：每个q对每个k进行attention计算

得到qkv变量后，后续计算如下图所示：



拿步骤一计算出的每个q和每个k进行Scale Dot-Product Attention的计算，计算公式如下：

$$\alpha_{1,i} = q^1 \cdot k^i / \sqrt{d}$$

其中α表示q和k的匹配程度（相关性），·表示Dot Product运算，即数量积运算；Scale表示规模，对应上述公式中的d(dimention)，表示q和k二者的维度（二者维度相同才能进行数量积运算）。

softmax计算公式如下：

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

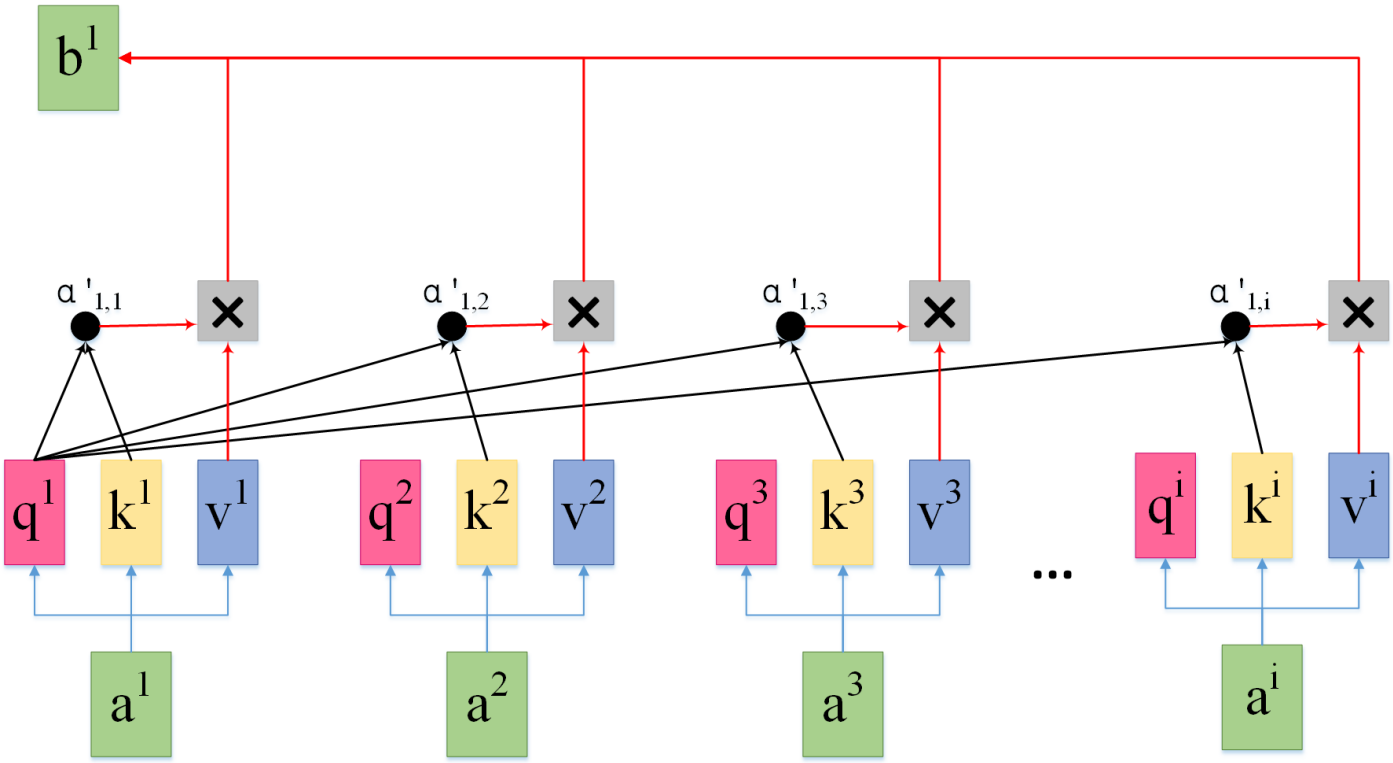
🎨 **为什么在Scale Dot-Product Attention计算中要除以 $\sqrt{d}$？**

参考原论文3.2.1解释如下：

首先，根据数量积计算公式（两向量对应位置的数值相乘后累加）可知，当维度d的值越大时，计算求出的q和k的variance（差异性数值）就越大，会将后续进行的softmax函数计算推入其梯度极小的区域，容易导致梯度消失的问题，影响模型的训练效果。**（精炼为：缩小点积范围，保证softmax梯度的稳定性）**

d越大variance越大的证明过程：https://zhuanlan.zhihu.com/p/436614439

步骤三：计算得到输出的vector/sequence

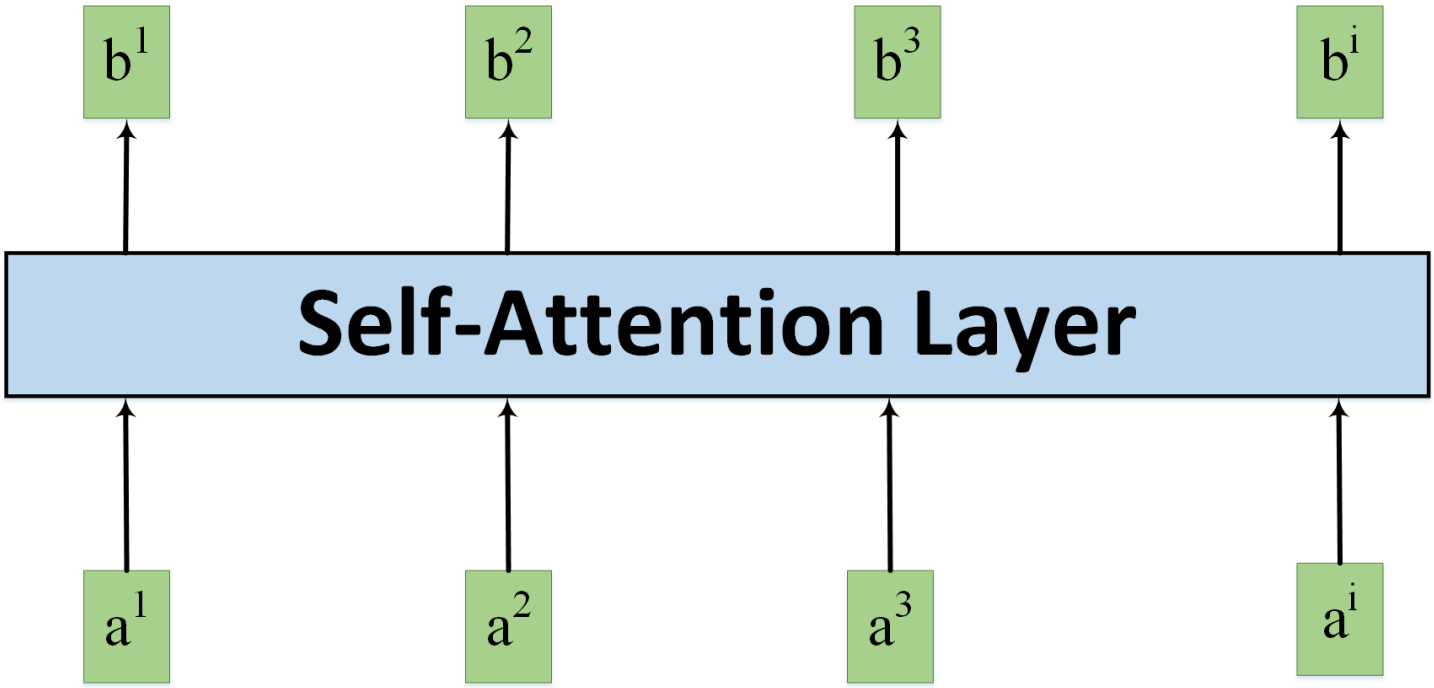


把经过Soft-max处理后的α'和v进行矩阵乘法后，进行加和得到b，即为输出序列的某个vector，其计算公式如下：

$$b^1 = \sum_i \alpha'_{1,i} v^i$$

由上述步骤可知，b_1 的产生来自 $\alpha'_{1,i}$，再往上进行探寻可知，来自所有的v，而v通过输入序列的每个vector和矩阵 W^v 进行乘法得到，进一步看到计算步骤一中我们对v这个变量所下的定义可知，b_1 已经考虑到了整个输入序列的信息，同理可知，所有的b都有能力考虑全局的信息；而如果只需要b考虑local imformation（即只考虑与其相近的vector的影响），只需要使得不需要纳入考虑位置的计算出来的α'结果为0即可达到。

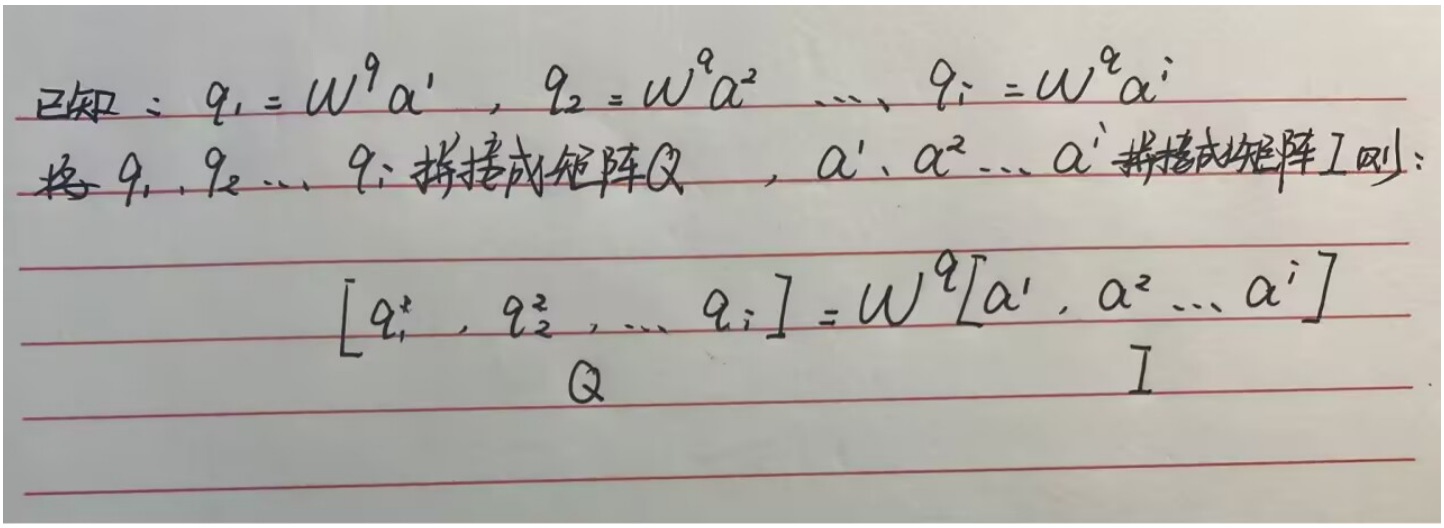
并行计算



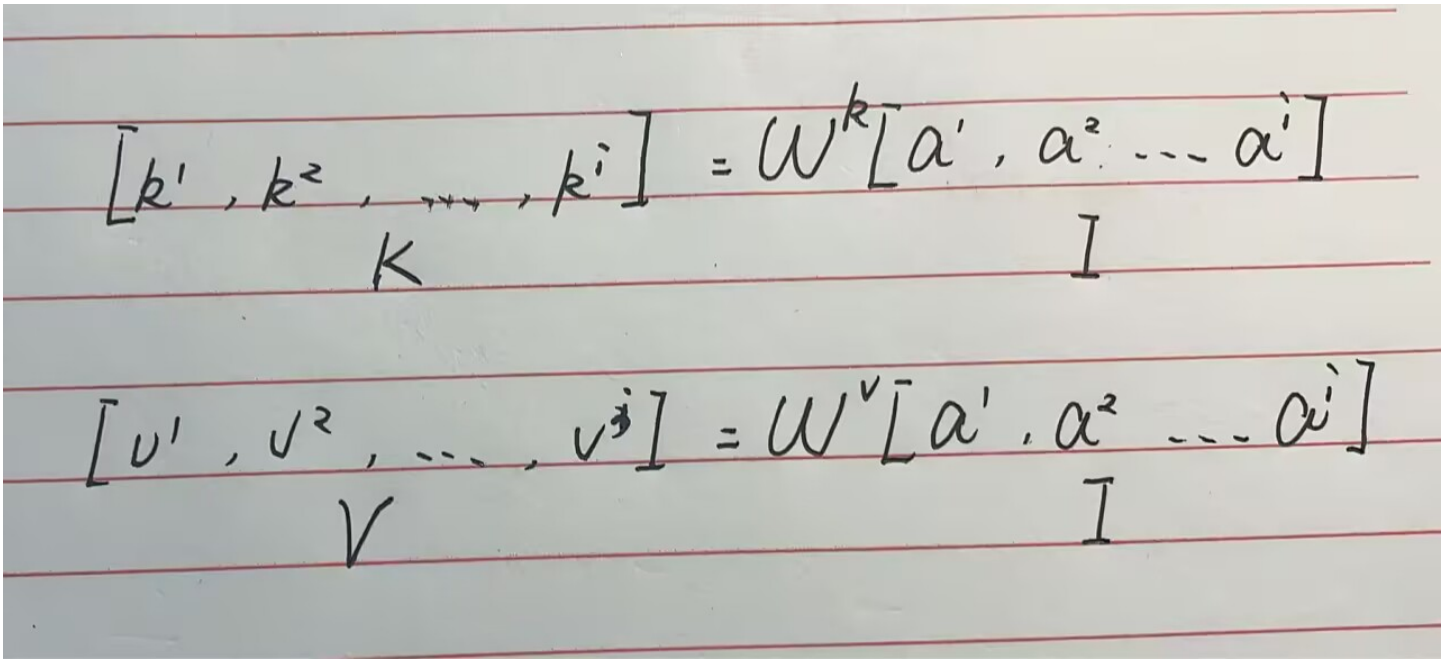
由计算步骤可知，Self-Attention的最终目的实际上和RNN几乎一致，即考虑需要纳入考虑的vector随后生成输出的vector组成sequence，但其中间计算步骤可由矩阵乘法的规则进行整合并行进行计算，其推导过程如下所示：

QKVI的定义

由计算公式 $q^i = W^q a^i$，将 α^i 根据矩阵乘法运算规则（参考分块矩阵知识点）进行拼接如下：



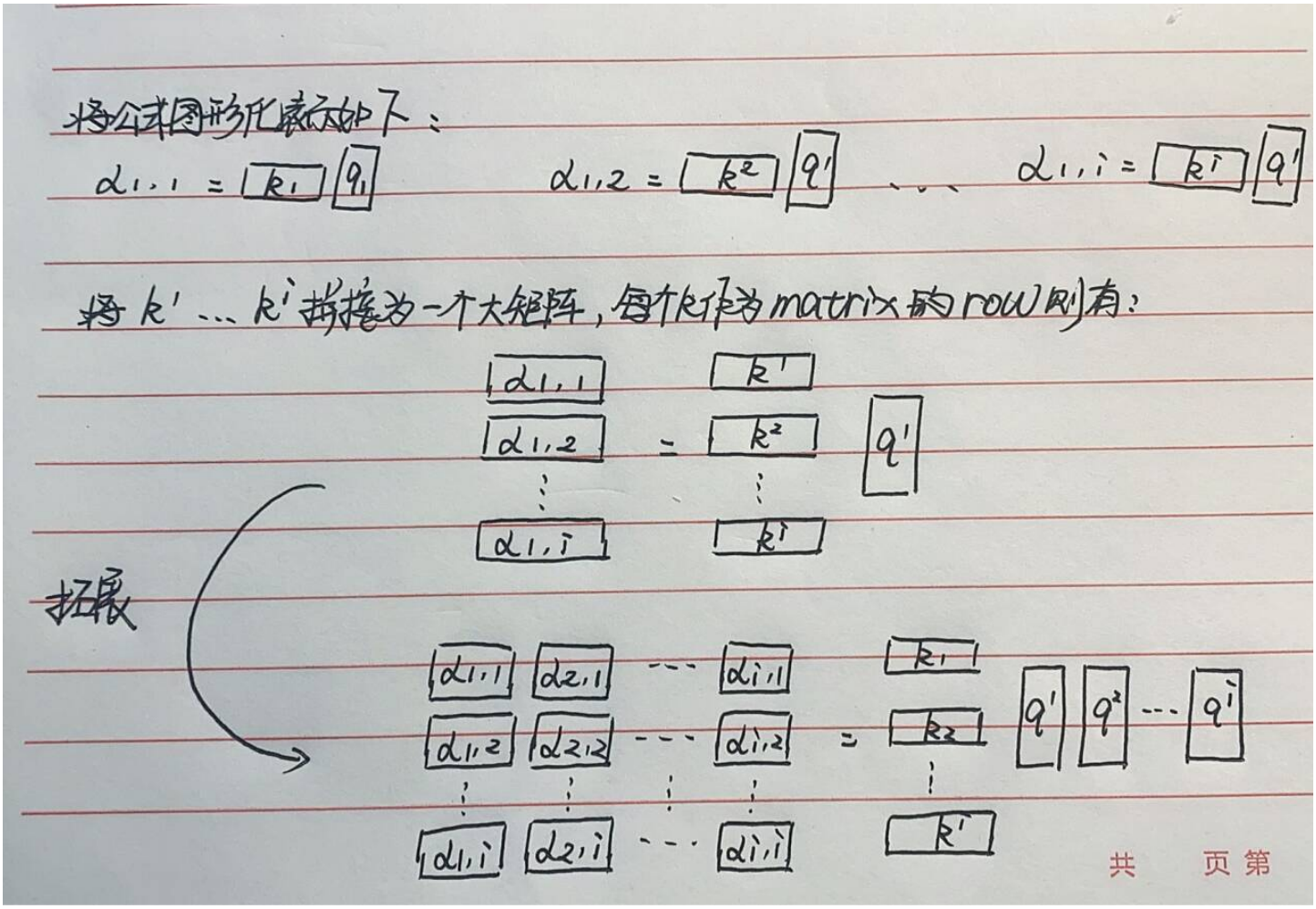
同理可知：



由上可知，QKV的每一个colum（列向量）即为对应的qkv。

注意力得分的并行

为了保证矩阵形式上的简单易识别，暂时忽略缩放因子 $\sqrt{d}$ 进行下述推导，已知 $\alpha_{1,i} = q^1 \cdot k^i / \sqrt{d}$：



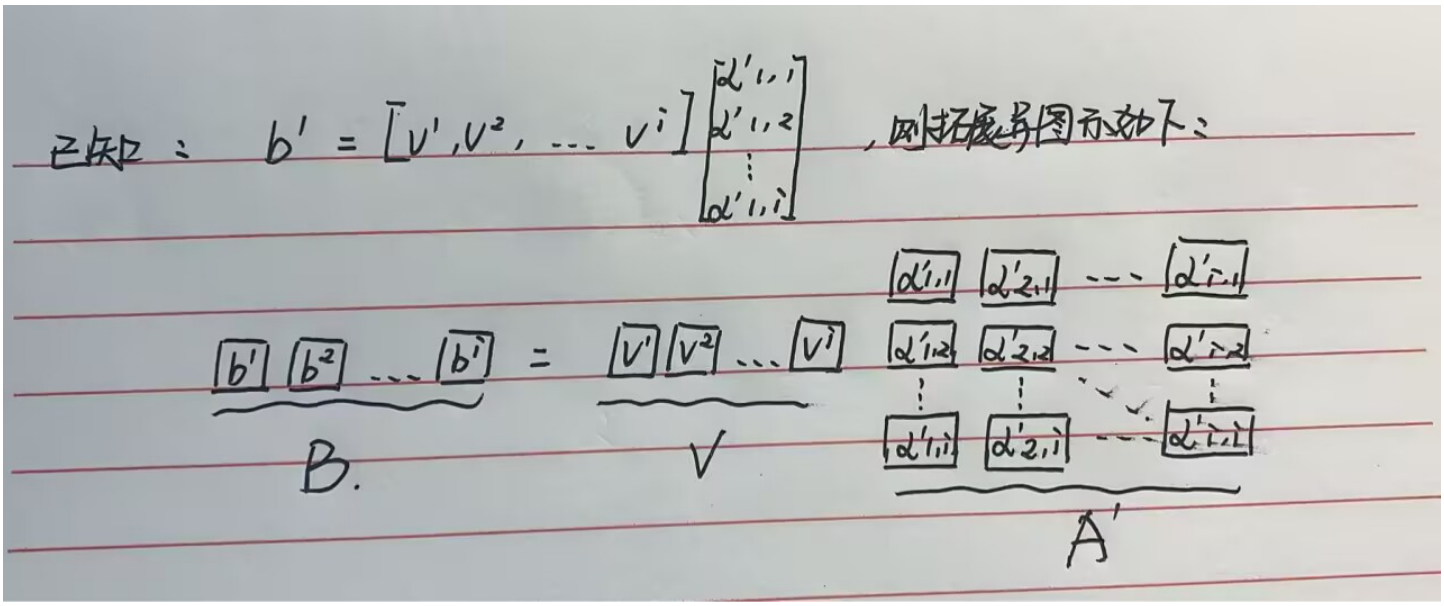
将等号左侧的α矩阵作为A，右侧即为K的转置与Q相乘（QKVI的定义部分有讲解），所以计算公式如下：

$$A = K^T Q \qquad\qquad A' = Softmax(A)$$

所以矩阵A所代表的含义即为注意力得分矩阵。

输出vector的并行计算

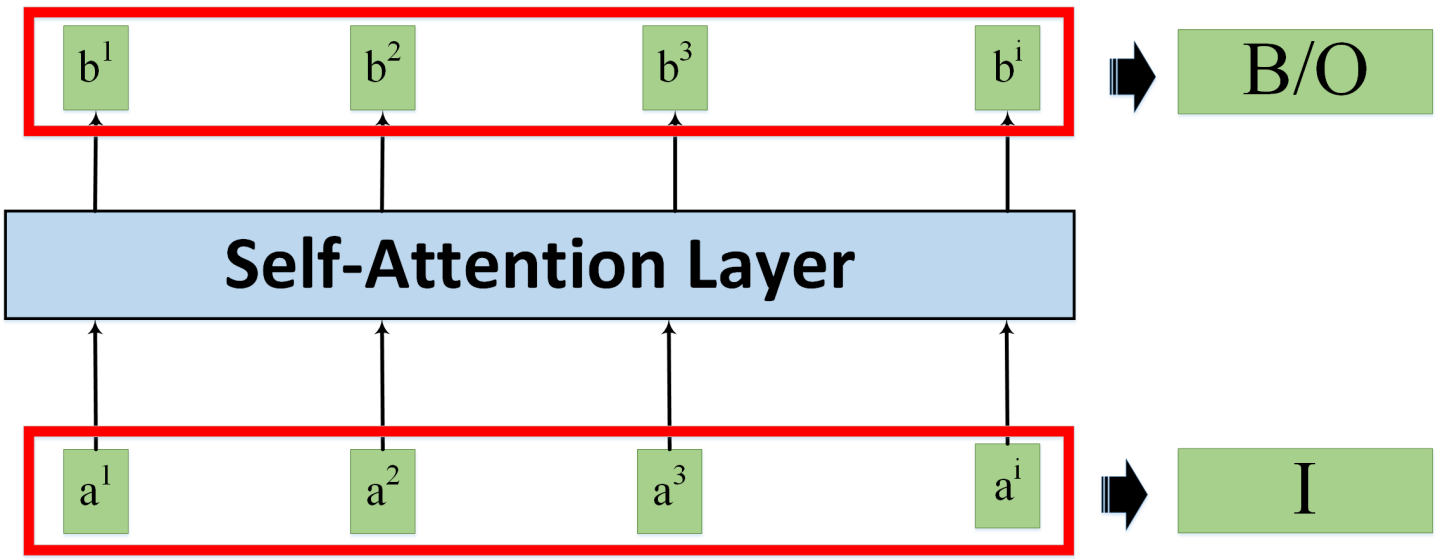
由公式 $b^1 = \sum_i \alpha'_{1,i} v^i$ 可进一步转换为 $b^j = \sum_{i,j} \alpha'_{j,i} v^i$ ，而 $\alpha'_{i,j}$ 组合起来就是上一步计算中所得到的A'，如下所示，按照线代矩阵乘法（第i行×第j列的sum为新矩阵的(i,j)位置元素）可以很好进行理解。



B矩阵实际意义为Self_Attention Layer的输出结果，所以同样记作矩阵O。

整体并行运算

首先，Self-Attention Layer可以整体化简如下：



则有公式：

$$B/O = Self - Attention(I)$$

接下来将Self-Attention Layer进行拆解公式化表达如下

1. 获取matrix Q，K，V

$$Q = W^q I \qquad K = W^k I \qquad V = W^v I$$

2. 获取matrix A

$$A = K^T Q$$

3. 获取输出O

$$O = V A' = V Softmax(A)$$

4. 原始变量带入

$$O = W^v I Softmax(A)$$

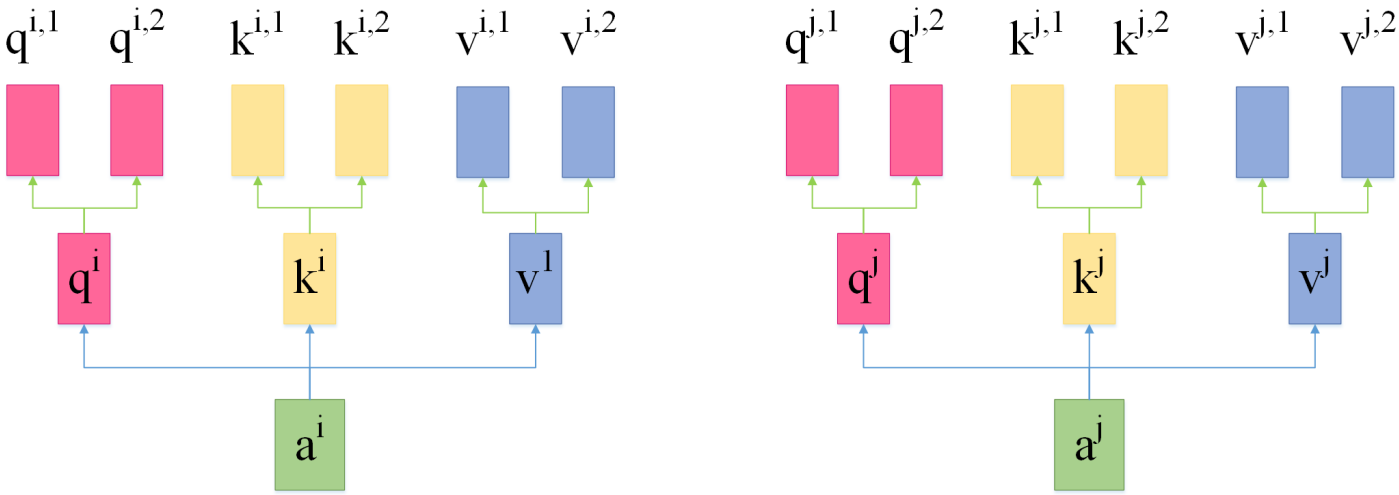
$$O = W^v I Softmax(K^T Q)$$

$$O = W^v I Softmax((W^k I)^T W^q I)$$

所以本质上，Self-Attention Layer只存在一个自变量I，在输入阶段就已经获取完成，而矩阵QKV则是随机初始化，需要在训练过程中进行学习和自动调参的变量，更能直观的体现出并行计算，并且均为矩阵相乘，能更好的利用GPU进行加速计算。

Muti-head Self-Attention

回想Self-Attention Layer的结果，可以很容易的推测出，多头自注意力机制的“多”，不可能多在Input和Output上，而是多在qkv三个矩阵上，也就意味着会出现多个 W^q, W^k, W^v 矩阵，其结构示意图如下所示（以两头自注意力机制为示例）：



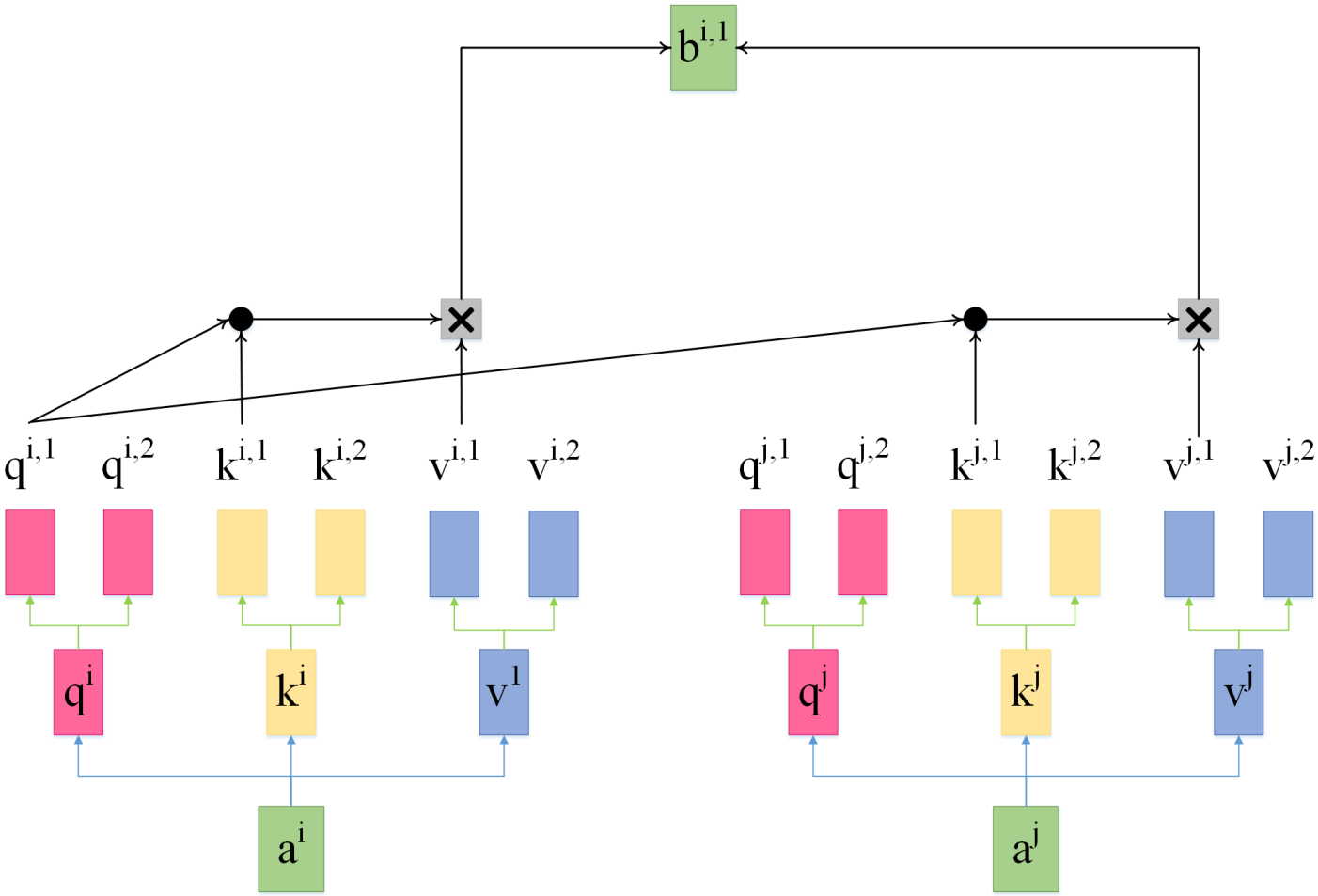
公式和Self-Attention中的类似，如下所示：

$$q^i = W^q a^i$$

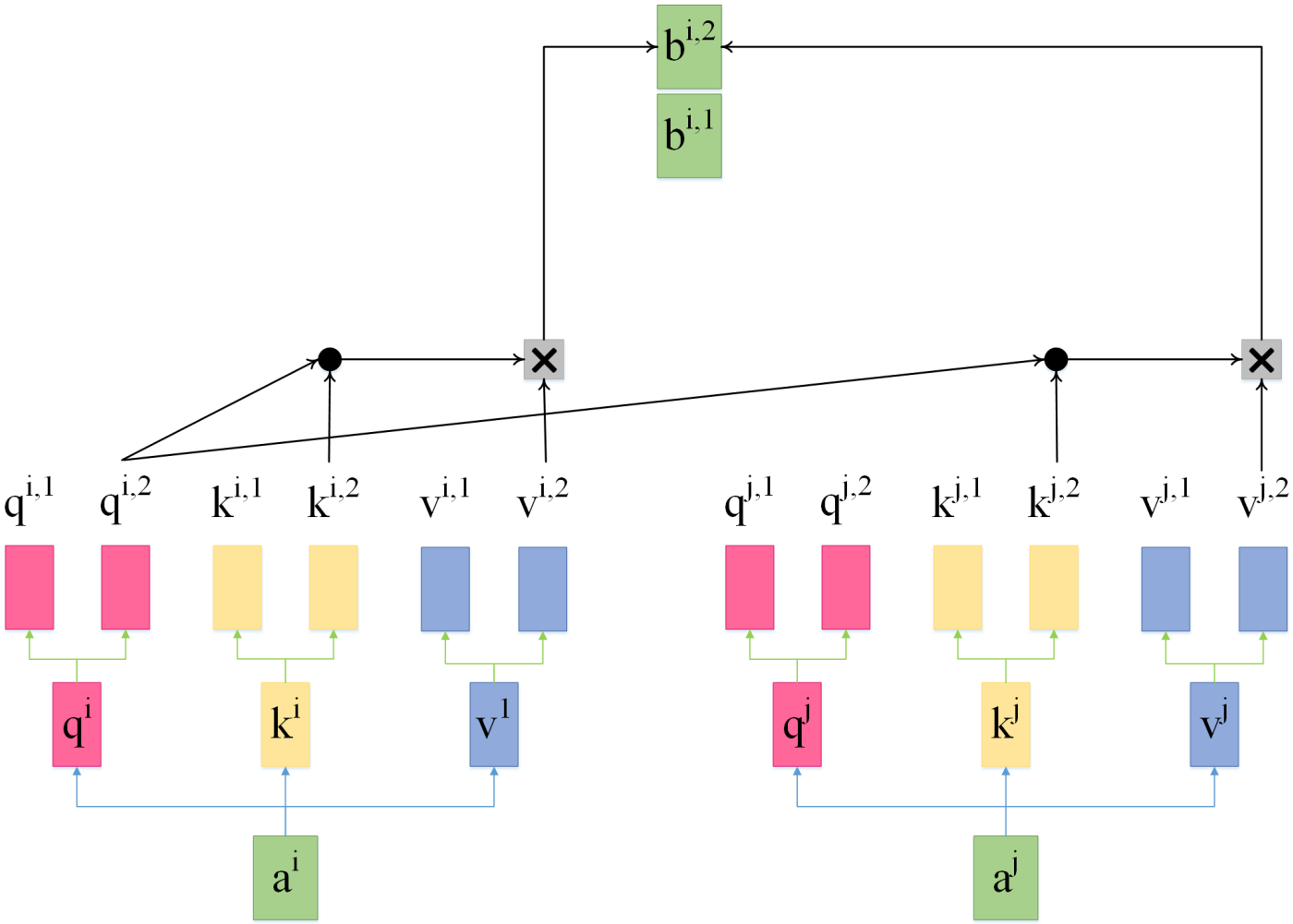
$$q^{i,1} = W^{q,1} q^i$$

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

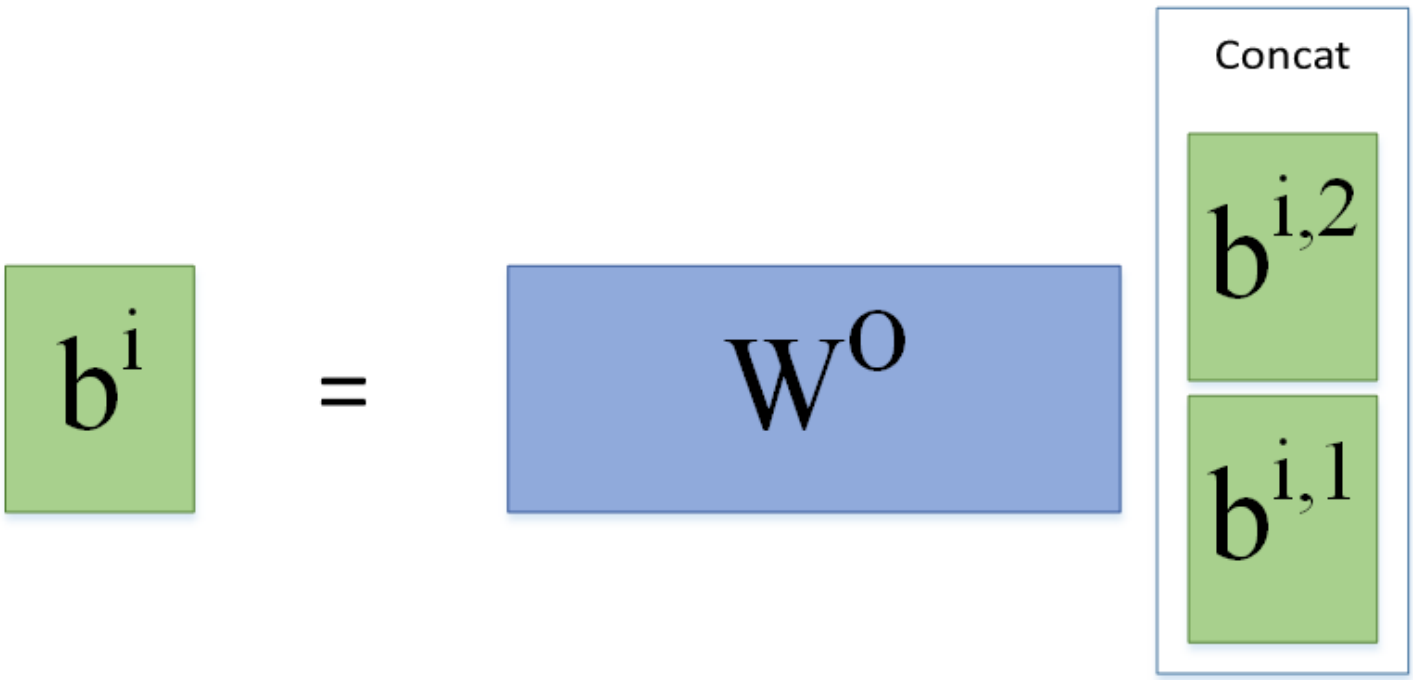
在得到不同的qkv后，将进行点击运算得到注意力分数b，此处需要注意的是，注意力分数和输出vector的计算不会跨“头”进行，即多头子注意力机制中，仅在相同类别的qk中进行计算，如下所示：



上图可以直观的看出，标号为1的q（标号即为q上标逗号后的数字）只会与标号为1的k进行点积后再和标号为1的v进行矩阵乘法求得b，不会受到其它的影响。那么，生成出来的vector维度将不止是一维，如下图所示，它会随着“head”的增加而增加：



将得到的多个b进行concat拼接操作后，再乘上一个变换矩阵 W^O 进行降维操作得到 b^i ，如下图所示：



将得到的多个b进行concat拼接操作后，再乘上一个变换矩阵 W^O 进行降维操作得到 b^i

Muti-head多个"head"的作用是什么?

首先，在Self-Attention中，我们了解到qkv三者间进行的计算本质上是和其它所有的输入vector进行相似度得分的计算，在训练过程中能够使得得到的矩阵B具备考虑全局信息的能力，也能使其具备考虑局部信息的能力，但显然二者是不能共存的，因为局部和全局是相互对立的，全局性好就意味着局部观测弱，局部能力强则意味着其缺乏考虑长久信息的能力**（Self-Attention/单头自注意力机制的弊端）**。

多头自注意力机制因为拥有不同的head，且相互之间在学习的过程中不受影响，那么就意味着，不同的head能够学习获取不同跨度的信息，从学习前后非常局部的信息，到学习全局联系的信息，**提高了模型的泛化能力和对复杂模式的信息捕获能力。**

其缺点也十分明显：

计算成本高：尽管并行处理有助于加速，但多头注意力依然增加了模型的参数量和计算复杂度，尤其是在资源受限的环境下可能成为负担。

解释性较差：多头注意力的内部工作机制较为复杂，每个头的具体功能往往难以直观理解，降低了模型的可解释性。

过拟合风险：过多的头可能会导致模型过度拟合训练数据，特别是在数据量有限的情况下。

3. Transformer的输入

在**“2. 模型结构中”**可以了解到，Self-Attention Layer中对输入序列的处理过程是不包含任何位置信息的，因为输入的向量a得到的qkv会和每一个输入向量进行attention的计算，所以两向量的“距离远近”或者是“逻辑远近”都是不重要的，但显然我们需要将输入序列的顺序纳入考虑范围内，因此则需要对输入序列进行处理使其本身能够包含位置信息。

Word Embedding

词向量直接从字面意思来看很好理解，即将单词转换为向量形式以便输入网络中进行后续计算，而具体的实现方式有以下几种。

Position Encoding

每个输入向量 a^i 都包含一个独特的位置编码向量 e^i 将二者进行加和则得到最终的输入向量。**注：** e^i **并不是从数据中学习得到的，而是人为给定的。**

📍 **为什么将向量直接进行相加能够表示位置信息呢？为什么不选择进行向量拼接？**

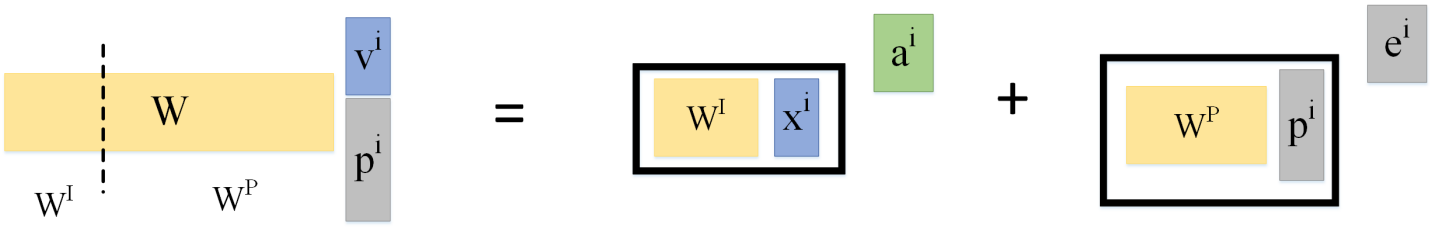
结论：二者最终结果等价，但进行Concat再计算会使网络进入深层后参数增加。

知识点：线性代数的分块矩阵

参考：

https://runwei.blog.csdn.net/article/details/124581338?fromshare=blogdetail&sharetype=blogdetail&sharerId=124581338&sharerefer=PC&shareresource=2301_76505819&sharefrom=from_link

图示如高亮块之后所示，其中矩阵p表示使用独热编码(one-hot)表示的位置关系



此处所说的人为给定肯定不是想设置多少就多少，而是指不通过训练过程进行学习，直接通过公式计算得到，详细信息如下所示。

首先对于理想的位置编码需要满足以下几个条件：

1. 单词在句子中的位置编码值唯一
2. 单词之间的间隔在不同长度的句子中的含义保持一致
3. 可以推广至更长的句子长度，且编码值有界

1和3都很好理解，那么2是个什么含义呢？

以往我们根据单词之间的间隔比例算距离，如果设置整个句子长度为1，如：Attention is all you need ，其中is和you之间的距离为0.5。而 To follow along you will first need to install PyTorch 较长文本中子里的0.5距离则会隔很多单词，这显然不合适。

其它编码方式的参考：

https://zhuanlan.zhihu.com/p/121126531

https://openreview.net/pdf?id=Hke-WTVtwr

所以采用如下公式：

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

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

当单词所处位置为偶数时，采用正弦函数进行计算，奇数时采用余弦函数。

pos：一个token在序列中的位置，假设句子长度为L，那么pos的取值则为0~L-1（通常为512）。

2i：偶数维度

2i+1：奇数维度

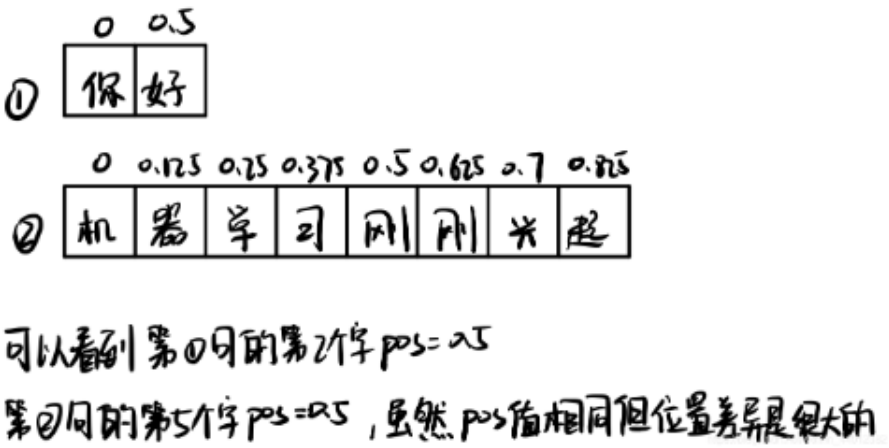
此处又会产生疑问了，我已经有了token在sequense中的位置，为什么又有一个变量i呢，是什么含义呢。其实很好理解，我们所输入的向量，是由词嵌入层和位置编码层加和后进行输入的，所以i即为

token中对应索引位置的元素，也就是说，一个token所计算出来的position encoding其实是一个向量，长度等价于token的向量长度而不是一个值。

原理部分

参考：https://blog.csdn.net/weixin_44012382/article/details/113059423

1. 不知道大家有没有想过为什么我们要用这么复杂的方法计算Positional Encoding，那让我们先来构思一个简单一点的Positional Encoding表达方法。首先，给定一个长为T的文本，最简单的位置编码就是计数，即使用$0, 1, ..., T-1$作为文本中每个字的位置编码(例如第2个字的Positional Encoding=[1,1,...,1])。但是这种编码有两个缺点：1. 如果一个句子的字数较多，则后面的字比第一个字的Positional Encoding大太多，和word embedding合并以后难免会出现特征在数值上的倾斜；2. 这种位置编码的数值比一般的word embedding的数值要大，对模型可能有一定的干扰。
2. 为了避免上述的问题，我们开始考虑把归一化，最简单的就是直接除以T，即使用$0, \frac{1}{T}, ..., \frac{T-1}{T}$(例如第2个字的Positional Encoding=[1/T,1/T,...,1/T])。这样固然使得所有位置编码都落入$[0, 1]$区间，但是问题也是显著的：不同长度文本的位置编码步长是不同的，在较短的文本中紧紧相邻的两个字的位置编码差异，会和长文本中相邻数个字的两个字的位置编码差异一致，如下图所示。



由于上面两种方法都行不通，于是谷歌的科学家们就想到了另外一种方法——使用三角函数$PE = sin(pos)$。优点是：1、可以使PE分布在$[0, 1]$区间。2、不同语句相同位置的字符PE值一样(如：当pos=0时，PE=0)。但是缺点在于三角函数具有周期性，可能出现pos值不同但是PE值相同的情况。于是我们可以在原始PE的基础上再增加一个维度：$PE = [sin(\frac{pos}{\alpha}), sin(\frac{pos}{\beta})]$，虽然还是可能出现pos值不同但是PE值相同的情况，但是整个PE的周期是不是明显变长了。那如果我们把PE的长度加长到和word embedding一样长呢？就像$PE = [sin(\frac{pos}{10000^{0/d_{model}}}), sin(\frac{pos}{10000^{2/d_{model}}}), ..., sin(\frac{pos}{10000^{2i/d_{model}}})]$（是不是有**Positional Encoding**计算公式内味了），PE的周期就可以看成是无限长的了，换句话说不论pos有多大都不会出现PE值相同的情况。然后谷歌的科学家们为了让PE值的周期更长，还交替使用sin/cos来计算PE的值，于是就得到了论文中的那个计算公式：

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

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

编码长度的拓展参考：https://blog.csdn.net/m0_37605642/article/details/132866365

从公式中可以看出，一个词语的位置编码是由不同频率的余弦函数组成的，从低位到高位，余弦函数对应的频率由1降低到 $\frac{1}{10000}$，波长从 2π 增加到 $10000 \cdot 2\pi$。这样设计的好处是：`pos+k` 位置的 `positional encoding` 可以被 `pos` 线性表示，体现其相对位置关系。

虽然 `Sinusoidal Position Encoding` 看起来很复杂，但是证明 `pos+k` 可以被 `pos` 线性表示，只需要用到高中的正弦余弦公式：

$$sin(\alpha + \beta) = sin\alpha \cdot cos\beta + cos\alpha \cdot sin\beta \quad (3)$$
$$cos(\alpha + \beta) = cos\alpha \cdot cos\beta - sin\alpha \cdot sin\beta \quad (4)$$

对于 `pos+k` 的 `positional encoding`：

$$PE_{(pos+k,2i)} = sin(w_i \cdot (pos + k)) = sin(w_i pos)cos(w_i k) + cos(w_i pos)sin(w_i k) \quad (5)$$
$$PE_{(pos+k;2i+1)} = cos(w_i \cdot (pos + k)) = cos(w_i pos)cos(w_i k) - sin(w_i pos)sin(w_i k) \quad (6)$$

其中 $w_i = \frac{1}{10000^{2i/d_{model}}}$。

将公式（5）（6）稍作调整，就有：

$$PE_{(pos+k,2i)} = cos(w_i k)PE_{(pos,2i)} + sin(w_i k)PE_{(pos,2i+1)} \quad (7)$$
$$PE_{(pos+k,2i+1)} = cos(w_i k)PE_{(pos,2i+1)} - sin(w_i k)PE_{(pos,2i)} \quad (8)$$

注意，`pos`和`pos+k`相对距离k是常数，所以有：

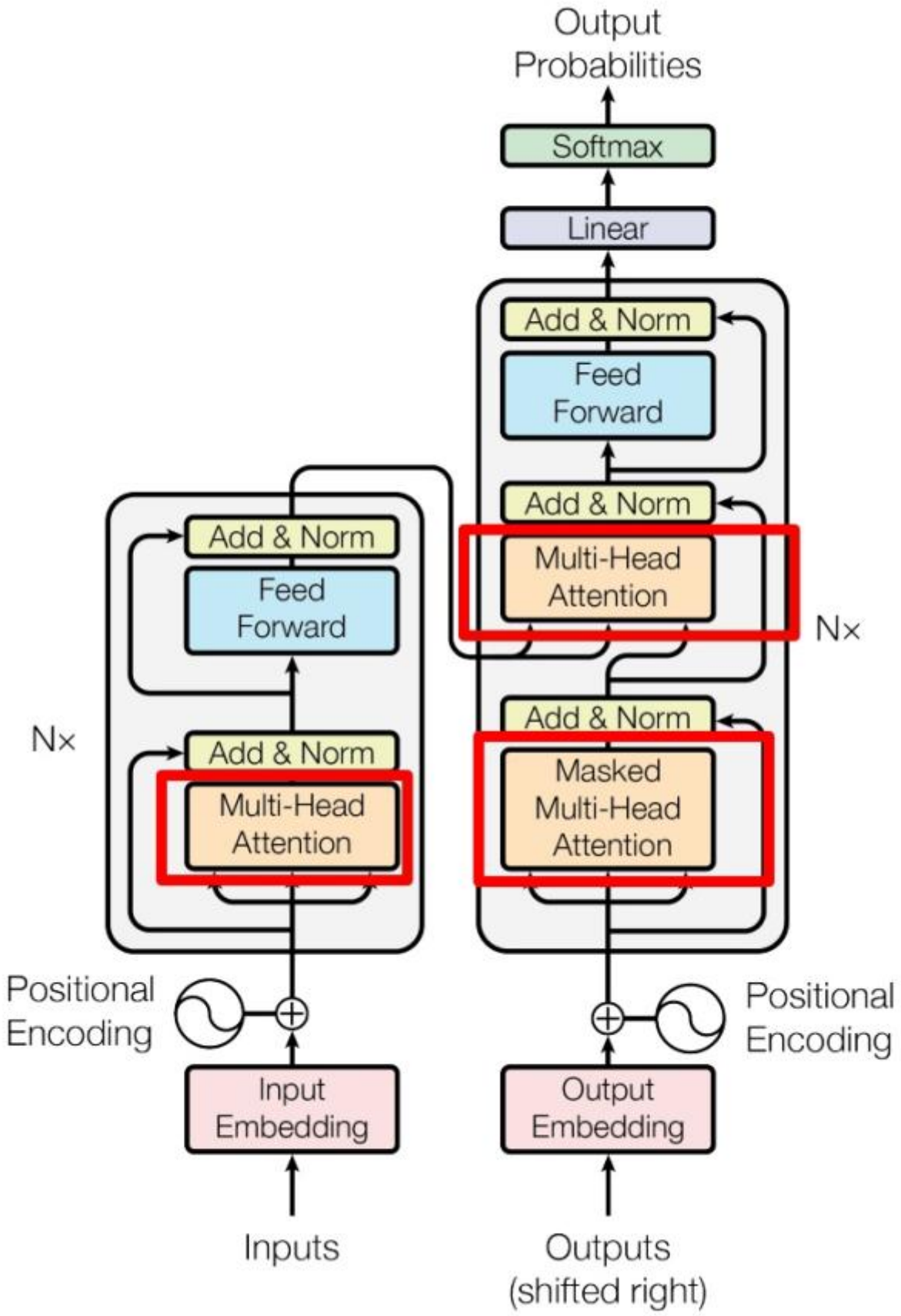
$$\begin{bmatrix} PE_{(pos+k,2i)} \\ PE_{(pos+k,2i+1)} \end{bmatrix} = \begin{bmatrix} u & v \\ -v & u \end{bmatrix} \times \begin{bmatrix} PE_{(pos,2i)} \\ PE_{(pos,2i+1)} \end{bmatrix} \quad (9)$$

其中，$u = cos(w_i \cdot k), v = sin(w_i \cdot k)$ 为常数。

可以看出，对于 $pos + k$ 位置的位置向量某一维 $2i$ 或 $2i + 1$ 而言，可以表示为：pos 位置与 k 位置的位置向量的 $2i$ 与 $2i + 1$ 维的线性组合，这样的线性组合意味着位置向量中蕴含了相对位置信息。所以 PE_{pos+k} 可以被 PE_{pos} 线性表示。

4. Seq2Seq模型中的Transformer

Transformer Layer在Seq2Seq Model中的使用如下图所示：



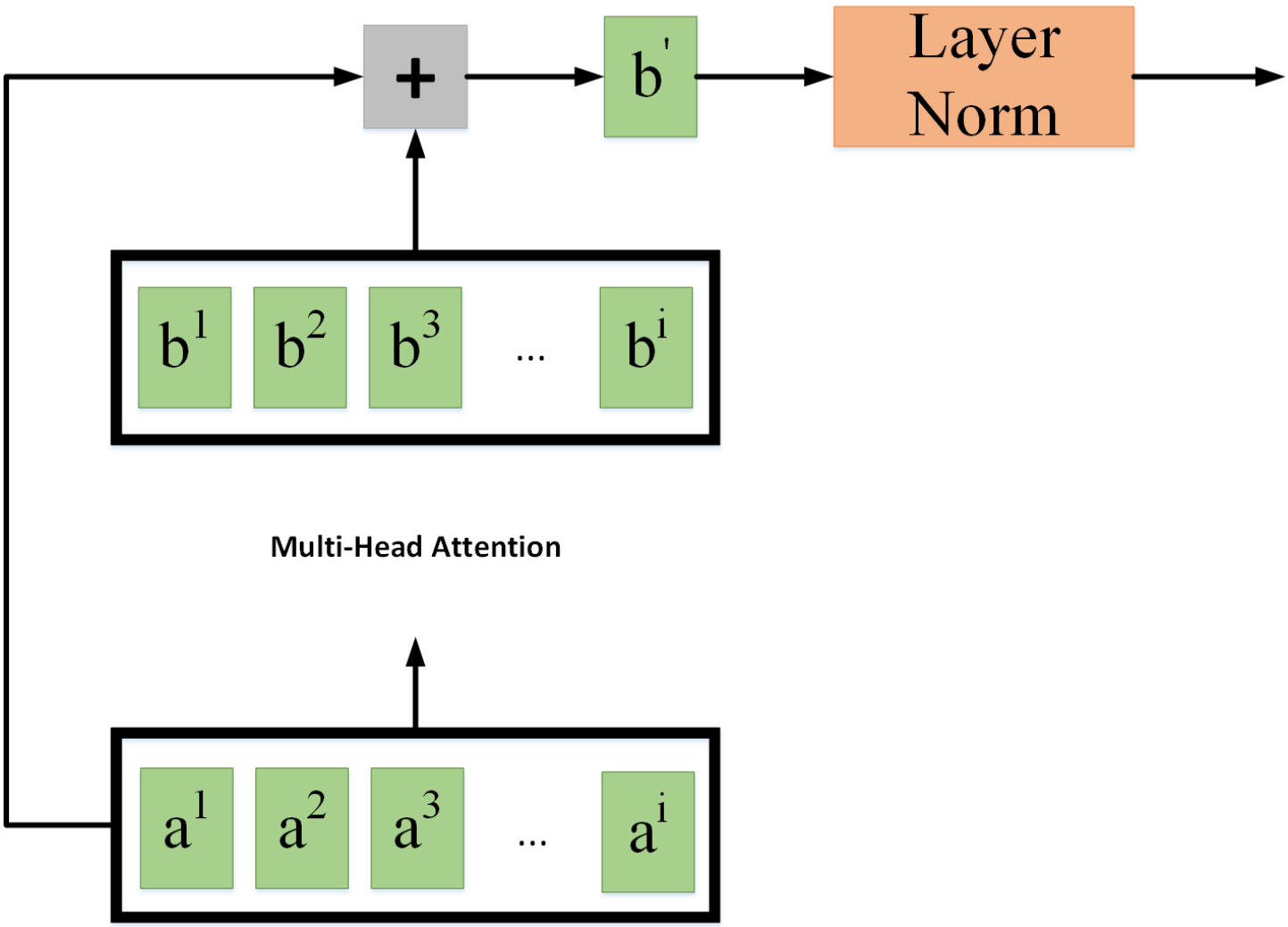
图像在竖直方向上呈现为左右两部分，其中左侧为编码器-Encoder，右侧为解码器-Decoder。

编码器Encoder

由图片所示能很直观的看出，输入的句子经过词嵌入层和位置编码处理后得到输入Sequence，传入Multi-Head Attention中，将输出序列传入后续模块进行计算。

Add & Norm Block

观察图中箭头指向可知，将Multi-Head Attention的输入和输出进行相加（有点类似于残差网络结构，只不过把简单的卷积层替换为了多头注意力模块）后再进行Norm操作（Layer Norm），如下图所示：



什么是Layer Normalization？它和Batch Normalization的区别是什么？

1. **为什么要进行归一化？**

Normalization 的作用很明显，把数据拉回标准正态分布，因为神经网络的Block大部分都是矩阵运算，一个向量经过矩阵运算后值会越来越大，为了网络的稳定性，我们需要及时把值拉回正态分布。

而根据标准化操作的维度不同可以区分为Batch和Layer两种。

2. **什么是Batch Normalization？**

Batch是批次的意思，那么就意味着该步骤的归一化是发生在一个批次内的，即：对同一批次数据中的相同特征维度（在卷积过程中会产生的通道，相同特征维度即代表相同的通道）进行归一化。

那么比较一个批次内不同数据的同一通道的作用或意义在哪？

首先理解深度学习中通道的含义：每个通道能提取到的特征信息有所不同，各司其职，所以同一维度的特征图往往对应着同类型的特征信息，例如第一层代表纹理，第二层代表亮度等……，所以也就说明了，不同信息的同纬度特征比较是存在意义的，而同一信息的不同特征如自身的纹理和亮度之间则失去其可比性。

3. 什么是Layer Normalization?

与Batch相对的维度在于，Layer是对每个样本/信息/层的所有特征进行归一化的处理

二者对比：

从上述描述可以得知，Batch关注于不同事物间同一特征区域的异同进行学习，而Layer则关注于自身的联系，所以前者用于CV任务中的归一化，后者用于nlp任务中的归一化，以更好的发现Sequence中每个token之间的联系及语义信息。

参考：

深度学习中通道含义的理解：

https://blog.csdn.net/qq_54641516/article/details/127079382?fromshare=blogdetail&sharetype=blogdetail&sharerId=127079382&sharerefer=PC&shareresource=2301_76505819&sharefrom=from_link

Batch和Layer：

https://zhuanlan.zhihu.com/p/647813604

https://blog.csdn.net/Little_White_9/article/details/123345062?fromshare=blogdetail&sharetype=blogdetail&sharerId=123345062&sharerefer=PC&shareresource=2301_76505819&sharefrom=from_link

Feed Forward

前馈全连接层原文链接：https://towardsdatascience.com/simplifying-transformers-state-of-the-art-nlp-using-words-you-understand-part-4-feed-foward-264bfee06d9

参考：https://zhuanlan.zhihu.com/p/665269977

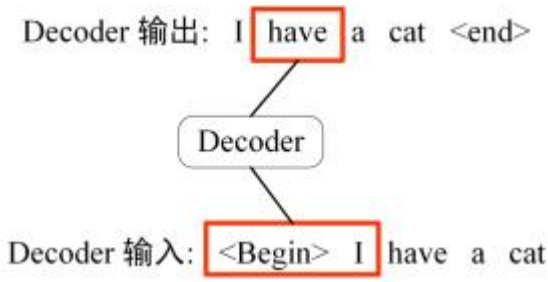
解码器Decoder

此处仅进行和Encoder中不同组件的解释

Masked Multi-Head Attention

Masked的意思在于，此处进行Self-Attention计算时，只关注于已经产生的序列，解释如下。解码器的作用在于拿到我们所需要的输出，以翻译举例，只有在完成这个单词的翻译后才能继续翻译下一个单词，所以在翻译第i个单词

时，需要掩盖i+1及之后的所有信息，如下图所示：



第二个Multi-Head Attention

观察图中可知，输入此部分的信息有两个箭头来自Encoder部分的输出，我们称为**编码信息矩阵**（此矩阵是由Encoder处理完输入序列所得到，包含输入序列的高级表示形式、元素间关系以及上下文信息），那么根据Self-Attention的机制我们可以得知，Encoder输入到Decoder部分的应该是k和v，而Decoder输入到后续步骤的则是q，用，这种操作使得Decoder能够获取到Encoder对输入序列的所有理解，从而根据当前的元素的q进行查询生成下一元素。