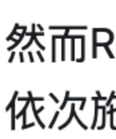
解剖Transformer 第一部分：你能默写出Transformer注意力机制的网络结构吗?


大方
轻舟智航 联合创始人

历史背景

Transformer的提出是在2017年，由Google Brain的Vaswani等学者发表的论文《Attention is All You Need》。当时自然语言处理（NLP）的各种任务（像machine translation, question answering, reading comprehension等）上，性能最好的网络结构基本都是循环神经网络（RNN✦），比如LSTM✦或者GRU✦。很多任务都符合序列转换（sequence transduction）的形式，即从一个任意长度的源序列生成一另个（可能不同长度）的目标序列，因此都广泛使用编码-解码（encoder-decoder）框架；在对序列的编码和解码中，RNN都是很自然的选择。注意力机制（attention）在当时并不是什么新鲜事，也已经作为源序列和目标序列之间进行对齐（alignment）的方法，在RNN的框架之内广泛使用。

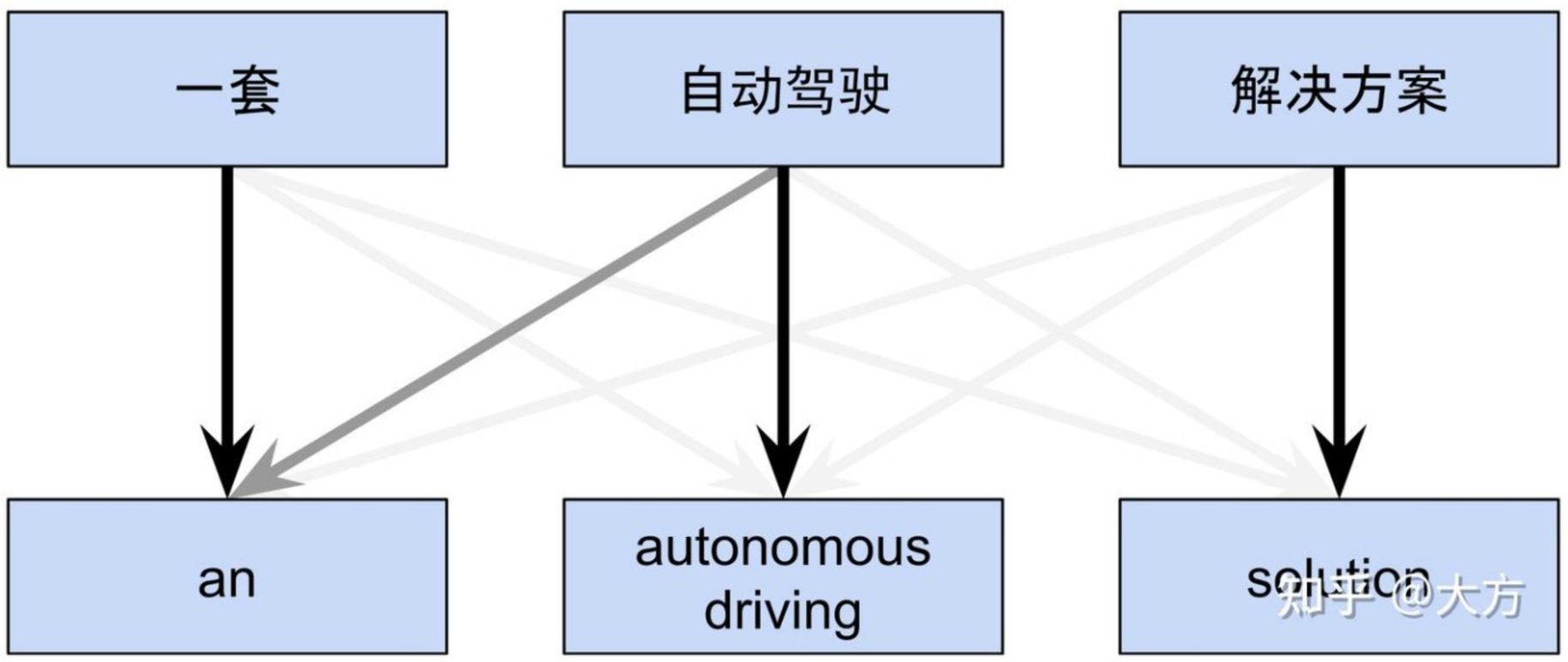
然而RNN有一个重要的缺点，就是计算的序贯性：用RNN处理一个输入的句子，需要将RNN单元依次施加于句子中的各个token（近似于单词），前一个token的计算不结束，下一个token的计算就不能开始，因此对并行计算非常不利，训练和推理效率都比较有限。从这个角度出发，上述论文的作者之一Uszkoreit构想，RNN对序列的编解码过程可能可以换成用注意力机制来实现。由此得到的是一种破天荒不依赖RNN结构、完全由注意力机制组成的架构，也就是“Transformer”，因为注意力计算的并行性提高了训练效率，在机器翻译等任务上的表现超越了以往的RNN方法，更重要的是大大提高了模型能力的天花板，打开了通向后来大规模预训练模型（像BERT，GPT-3）的大门。

在Transformer架构中，源到目标序列之间的对齐仍然使用以前的注意力机制，但对源序列的编码和目标序列的解码中，替换掉RNN的是一种叫做自注意力✦（self-attention）的网络结构，与用于对齐的注意力在结构上大同小异，这也是整个Transformer架构的核心所在。所以，我们先来看看注意力机制是怎么回事。我们从最根本的机制讲起，因为这个机制本身很灵活普适，所以在此基础上很多学者提出了大量的变体、改进，但都是万变不离其宗。理解了注意力的核心思想之后，那些衍生的概念就都会迎刃而解，几句话就能讲清楚。

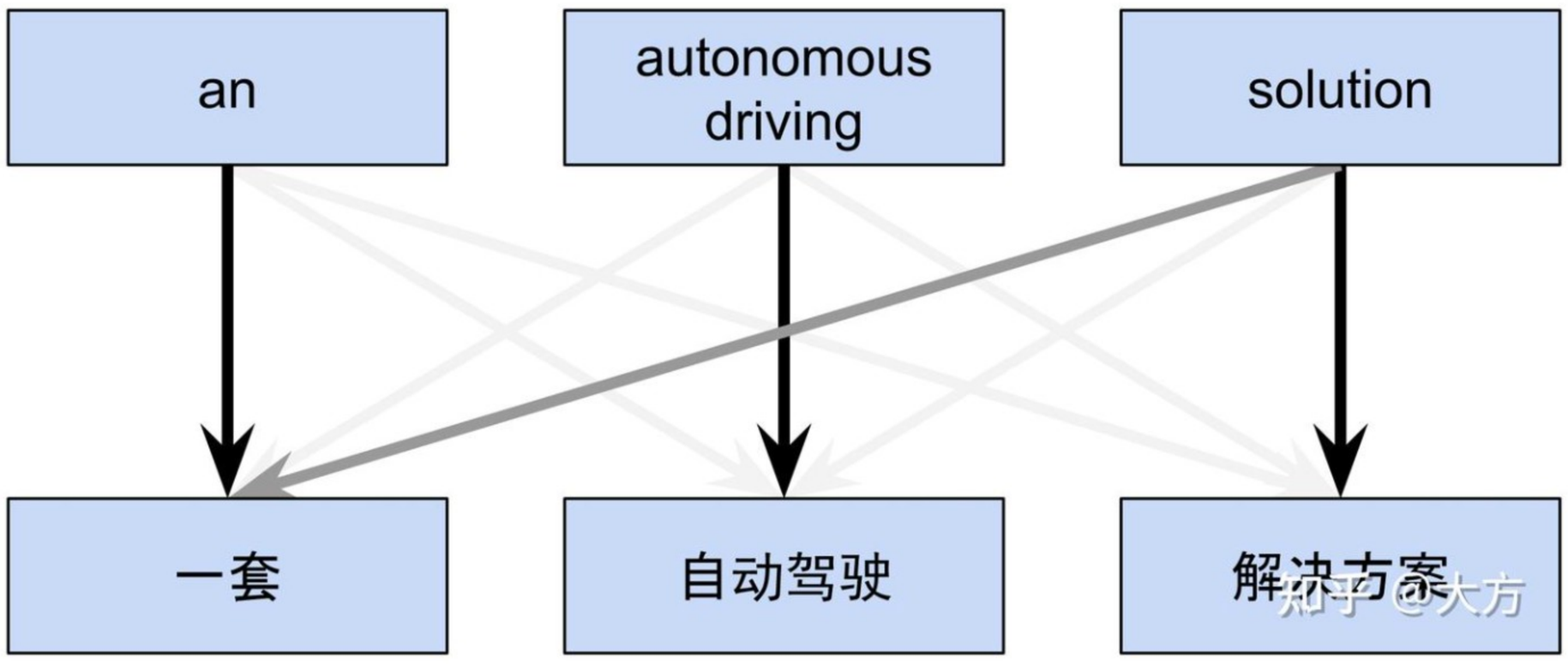
注意力机制

注意力机制是一种从一系列输入中选择性地提取信息的结构。不论应用场景是机器翻译（从一种语言的token序列生成另一种语言的token序列）还是图像分类（从一堆像素或者图像块识别出物体种类），如果输入的形式是一个某种单位元素的集合或者序列，就可以使用注意力机制来让元素之间进行信息交换。将注意力应用在序列转换的源序列和目标序列之间，就是Transformer（包括Transformer以前的方法）中进行对齐的机制。为了避免歧义，这里我们把序列转换的形式定义如下：序列转换有两个输入序列，一个叫做源序列，一个叫做目标序列；两个序列不一定有相同的长度。每个序列中每个元素都由一个一定维度的特征向量表示，源序列和目标序列的特征向量维度也不一定相同。序列转换操作的输出是对目标序列中的每个元素输出一个新的特征向量，叫做输出序列。虽然像机器翻译这样的任务中要求模型以一个序列（源语言的语句）为基础从头生成另一个序列（目标语言的语句），形式跟上面我们定义的序列转换不太一样，但机器翻译（以及其他类似的“序列转换类”的任务）的模型通常是以上述序列转换操作为基础构建的，通常以源语言语句为源序列，以已生成的部分为目标序列来生成下一个目标语言token（在第二部分关于Transformer训练的部分会详细讨论）。

我们通过一个中译英的例子来讨论注意力机制：从中文“一套自动驾驶解决方案”翻译到英文“an autonomous driving solution”。虽然“自动驾驶”翻译成“autonomous driving”和“解决方案”翻译成“solution”都是比较直接了当的[1]，从中文token自身的embedding就可以生成对应的英文，但“一套”应该翻译成“a”还是“an”就不是看这个单词本身就能决定的了。进行翻译的模型在生成“an”这个输出的时候，不但需要从对应成份的源元素“一套”中提取语义特征，还需要参考其相关的源元素“自动驾驶”的特征，判断出其对应的英文的首字母是个元音，由此才能确定不定冠词“an”。注意力就是这样一种能够让源序列的多个元素上的特征有选择地进入目标序列的特征的网络结构。每个源元素都有可能对输出有影响，因此每个源元素的特征上都存在到输出特征的通路，而每个通路上都有一个开关，根据不同情况来决定每个开关的打开与否（或者打开程度）；目标序列中的每一个元素，都应该能够根据自身需要来独立控制自己的每个开关。如下图所示，从源序列（上方）每个元素到目标序列（下方）每个元素都有一条连线表示一个特征通路（这个目标元素的特征依赖这个源元素的特征），而每条线的颜色深浅则表示这个通路的打开程度，或者叫做“注意力分布”（比如不但存在从“一套”到“an”有直接的语义对应，也存在从“自动驾驶”到“an”的因为拼写造成的间接影响）。



这个例子反过来也有类似的情况，从英文“an autonomous driving solution”翻译到中文“一套自动驾驶解决方案”，在输出“一套”时，同样不但要参考对应的输入“an”，还要参考后面短语的中心语“解决方案”，因为它决定了量词是“套”（另如，从“an autonomous driving car”翻译成“一辆自动驾驶汽车”，这个量词就会变成“辆”）。



那么这个“注意力分布”是如何计算的，或者说，每个目标元素怎样决定打开哪些源元素的开关，让它们的特征参与输出特征的计算呢？这里不同的注意力机制类型有不同的算法。最常见的方式是Transformer所使用的注意力机制，称作“点积注意力✦”[2]。具体的计算方法是，源序列的每个元素根据自身特征计算一个“键”（key）向量和一个“值”（value）向量，而目标序列的每个元素根据自身特征计算一个“查询”（query）向量，然后通过这个query与每个源元素的key的相似度来决定这

个源元素的value有多少会进入到输出特征中；key和query越相似，对应的开关就越接近100%完全打开：

$$\mathbf{y} = \sum_{i=1}^{N} S(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i$$

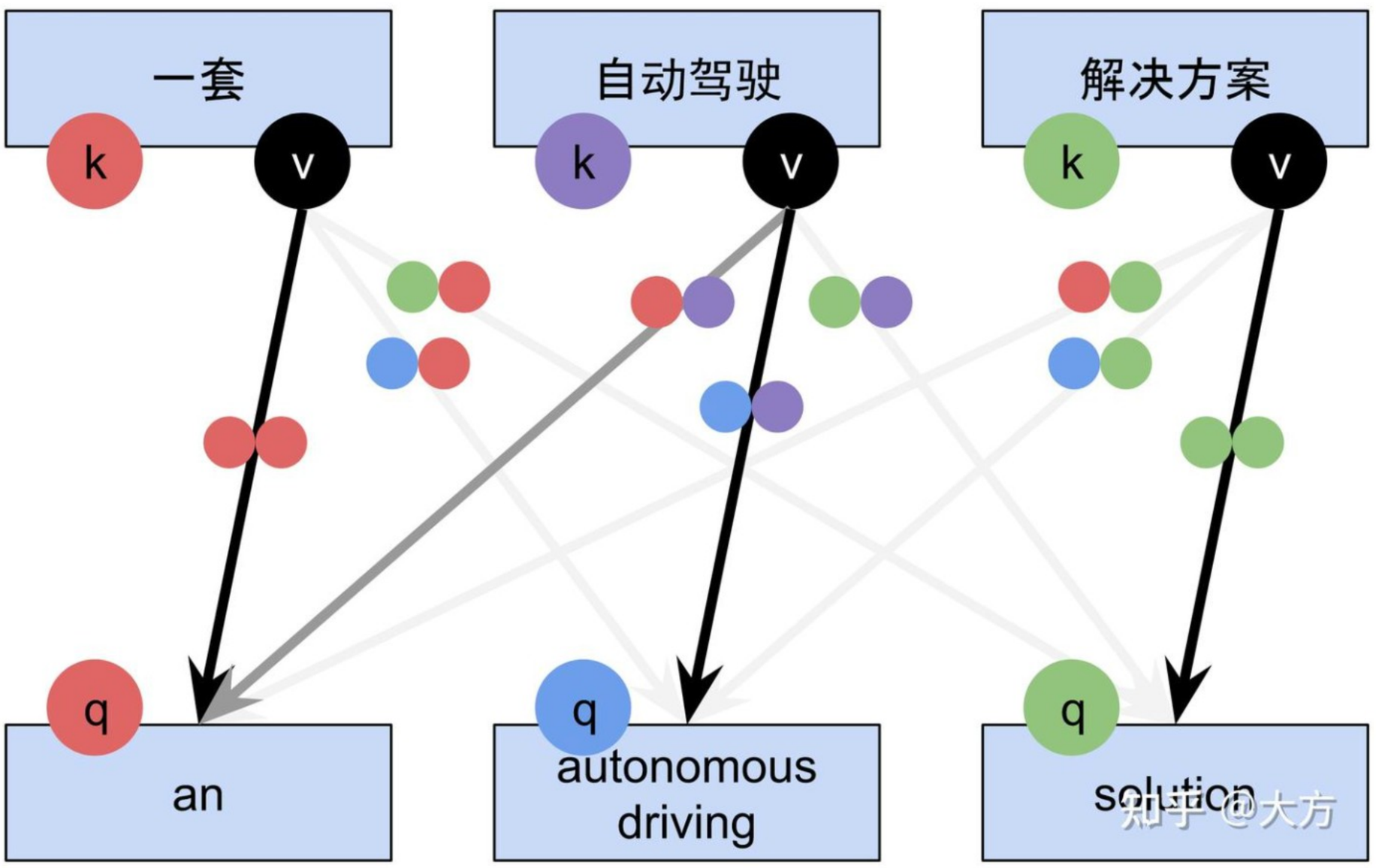
这里 $\mathbf{y}$ 和 $\mathbf{q}$ 分别为最终输出和query向量， $\mathbf{k}_i$ 和 $\mathbf{v}_i$ 分别为第i个源元素的key和value向量，N为源元素的个数，S是一个向量相似度的量度。最常用的向量相似度就是向量点积，原理上近似于用我们熟悉的余弦距离（cosine distance）作为相似度量度S，这也是“点积注意力”这个名字的来源：

$$\mathbf{y} = \sum_{i=1}^{N} \operatorname{softmax}\left(\mathbf{q} \cdot \mathbf{k}_i\right) \mathbf{v}_i$$

$$S(\mathbf{q}, \mathbf{k}) = \operatorname{softmax}\left(\mathbf{q} \cdot \mathbf{k}\right)$$

在点积之外有个在所有源元素间的softmax操作，使得所有value的权重总和为1，这是为了保证所有源元素贡献的特征总量保持一定；如果有多个key都与query高度相似，那么它们各自的通道都会只打开一部分（好像“注意力分散在这几个源元素上”）。从这个角度来看，可以理解为输出y是在value之间根据key-query的相似度进行内插值，或者加权平均。

如果目标序列有多个元素，每个元素都有各自的q，并且单独计算各自的输出y。所以，上述中译英的例子里共有九个点乘，对应于上图中九条潜在的信息通路，点积各自决定对应通路的打开程度。



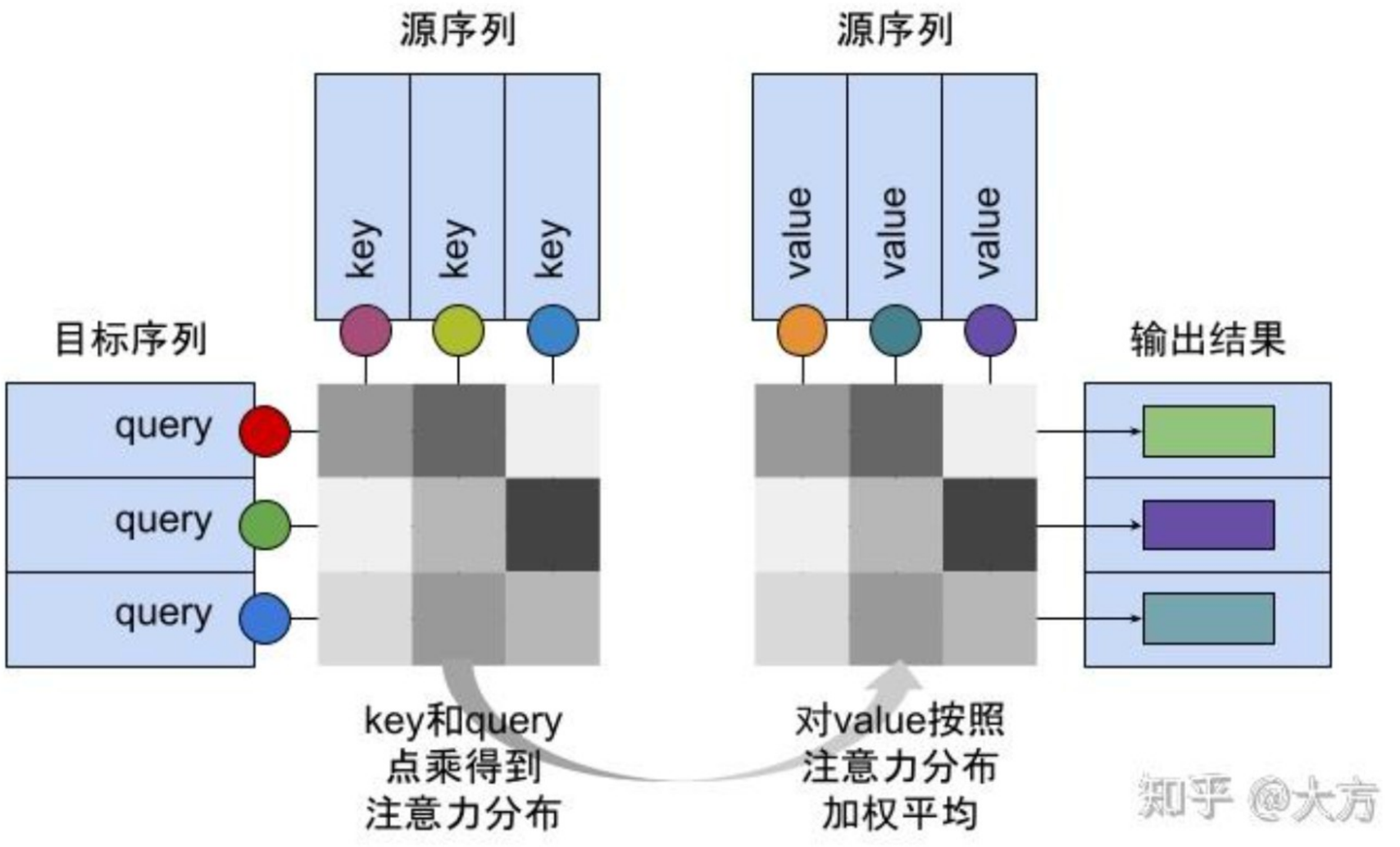
$$\mathbf{y}_{an} = \frac{\exp\left(\mathbf{q}_{an} \cdot \mathbf{k}_{\text{一套}}\right)\mathbf{v}_{\text{一套}} + \exp\left(\mathbf{q}_{an} \cdot \mathbf{k}_{\text{自动驾驶}}\right)\mathbf{v}_{\text{自动驾驶}} + \exp\left(\mathbf{q}_{an} \cdot \mathbf{k}_{\text{解决方案}}\right)\mathbf{v}_{\text{解决方案}}}{\exp\left(\mathbf{q}_{an} \cdot \mathbf{k}_{\text{一套}}\right) + \exp\left(\mathbf{q}_{an} \cdot \mathbf{k}_{\text{自动驾驶}}\right) + \exp\left(\mathbf{q}_{an} \cdot \mathbf{k}_{\text{解决方案}}\right)}$$

$$\mathbf{y}_{AD} = \frac{\exp\left(\mathbf{q}_{AD} \cdot \mathbf{k}_{\text{一套}}\right)\mathbf{v}_{\text{一套}} + \exp\left(\mathbf{q}_{AD} \cdot \mathbf{k}_{\text{自动驾驶}}\right)\mathbf{v}_{\text{自动驾驶}} + \exp\left(\mathbf{q}_{AD} \cdot \mathbf{k}_{\text{解决方案}}\right)\mathbf{v}_{\text{解决方案}}}{\exp\left(\mathbf{q}_{AD} \cdot \mathbf{k}_{\text{一套}}\right) + \exp\left(\mathbf{q}_{AD} \cdot \mathbf{k}_{\text{自动驾驶}}\right) + \exp\left(\mathbf{q}_{AD} \cdot \mathbf{k}_{\text{解决方案}}\right)}$$ [3]

$$\mathbf{y}_{solution} = \frac{\exp\left(\mathbf{q}_{solution} \cdot \mathbf{k}_{\text{一套}}\right)\mathbf{v}_{\text{一套}} + \exp\left(\mathbf{q}_{solution} \cdot \mathbf{k}_{\text{自动驾驶}}\right)\mathbf{v}_{\text{自动驾驶}} + \exp\left(\mathbf{q}_{solution} \cdot \mathbf{k}_{\text{解决方案}}\right)\mathbf{v}_{\text{解决方案}}}{\exp\left(\mathbf{q}_{solution} \cdot \mathbf{k}_{\text{一套}}\right) + \exp\left(\mathbf{q}_{solution} \cdot \mathbf{k}_{\text{自动驾驶}}\right) + \exp\left(\mathbf{q}_{solution} \cdot \mathbf{k}_{\text{解决方案}}\right)}$$

源-目标的注意力分布（颜色越深表示关注度越高）	an	AD	solution
一套			
自动驾驶			
解决方案			@大方

由于从每个源、目标元素自身的特征生成query、key和value向量的计算都是可学习的（通常这个计算就是一个全连接层，分别称作query映射，key映射和value映射），因此经过大量训练之后，每个元素都会找到完成各自任务所需的最合适的query、key和value。下图对注意力机制的计算过程（点乘，加权平均）作一直观总结。



到这里，想必你已经可以用自己熟悉的机器学习框架（比如pytorch）写出注意力机制的源代码了吧?

模型输入：

- $\mathbf{x}$： 所有源序列的元素的特征，维度为 $N \times D_x$， N 为源元素的个数， D_x 为特征维度
- $\mathbf{z}$： 所有目标序列元素的特征，维度为 $M \times D_z$， M 为目标元素个数， D_z 为特征维度

模型参数：

- $w_k, w_v, w_q, b_k, b_v, b_q$： key，value和query的全连接层权重和偏置，维度分别为：
  - w_k： $D_k \times D_x$
  - b_k： D_k
  - w_v： $D_v \times D_x$
  - b_v： D_v
  - w_q： $D_k \times D_z$
  - b_q： D_k

模型输出：

- $\mathbf{y}$： 所有目标序列元素上的输出，维度为 $M \times D_v$

```
def dot_product_attention(x, z, w_k, w_v, w_q, b_k, b_v, b_q):
    # Try filling in the code here by yourself!


    return y
```

试试在上面按你自己的理解填入代码。你可以试试跟pytorch自带的Transformer参考实现的源代码比较一下（multi_head_attention_forward()），但参考实现中包含大量定制化和优化，以及下文会讲到的“多头注意力✦”等改进，所以代码读起来可能不太直观。

自注意力（self-attention）

自注意力其实就是上述注意力机制施加于同一个序列上，也就是说，源序列和目标序列的元素相同（$\mathbf{x} = \mathbf{z}$）。从形式上来看，输入是一个序列的元素特征，输出是同一个序列的元素经过解析后的特征，形式跟RNN是一样的，符合编码器（encoder）的接口，因此Transformer架构直接用自注意力替换掉RNN，从而避免了RNN的序贯计算的弱点。

直观来讲，自注意力是让这个序列中每一个元素都有机会根据自身特征选择性地吸取整个序列中每一个其他元素的信息，这样达到的效果就是我们常说的元素间的“互动”。比如，在机器翻译中，在把源语言的单词翻译到目标语言之前，模型需要先理解每个词的语义，而有些词的语义需要上下文（context）才能确定。在上述英译中的例子里，输入的最后一个元素是“solution”。单看这个词本身，它可能表示“解决方案”，也可能表示方程的“解”，还可能表示“溶液”，那么在这里它的含义应该是哪个呢？这就需要观察这个词周围的上下文，从前面的“autonomous driving”推断出它表示的含义应该是“解决方案”而不是“解”或者“溶液”。这个过程就是序列中元素间交换信息的过程，也就是“互动”的过程。信息交换发挥着替代RNN中的记忆能力（memory）的功能，使得自注意力编码器也能像RNN一样对一个复杂的语句进行理解和解析。

把点积注意力写成矩阵乘法

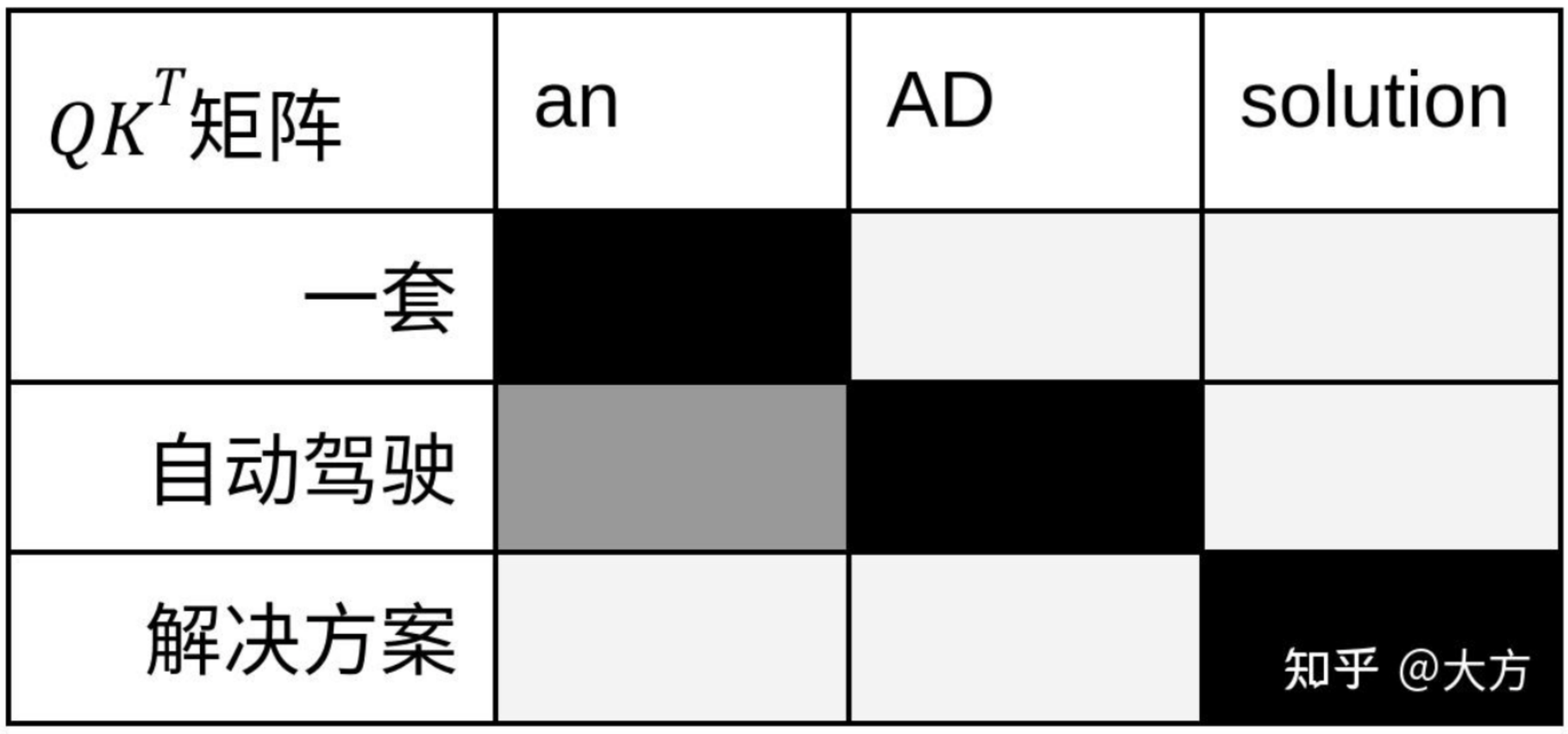
上文中我们把从3个元素的源序列到3个元素的目标序列的注意力计算写成了3个方程，每个方程包含3个点积运算。用矩阵乘法记法来写，可以大幅度简化，也让运算的过程更一目了然。假定源序列和目标序列元素个数分别为 N 和 M，query和key的维度为 D_k（因为点乘需要，两者维度必须向同），value维度为 D_v（可以不等于 D_k），我们可以把所有元素的query，key和value分别堆叠在一起：

- M 个目标序列元素的query形成一个 $M \times D_k$ 矩阵 Q，
- N 个源序列元素的key形成一个 $N \times D_k$ 矩阵 K，
- N 个源序列元素的value形成一个 $N \times D_v$ 矩阵 V，

那么，对所有输出 $\mathbf{y}$（维度 $M \times D_v$）的计算可以直接写成：

$$\mathbf{y} = \text{softmax}\left(QK^T\right)V$$

这也不难理解，毕竟矩阵乘法的直观解读之一就是左侧每一行（query）分别和右侧每一列（key）的点积。虽然含义没变，但是使用这一记法对理解点积注意力的工作机制非常有帮助。其中，矩阵 QK^T 取softmax之后就是所谓的“注意力值”或者注意力分布，作为用源元素value合成输出值时所用的权重，它直接反映了每个目标元素对每个源元素的关注程度。在源序列、目标序列比较长的时候，观察这个矩阵可以帮助我们理解模型到底学到了什么。



多头注意力（multi-headed attention）

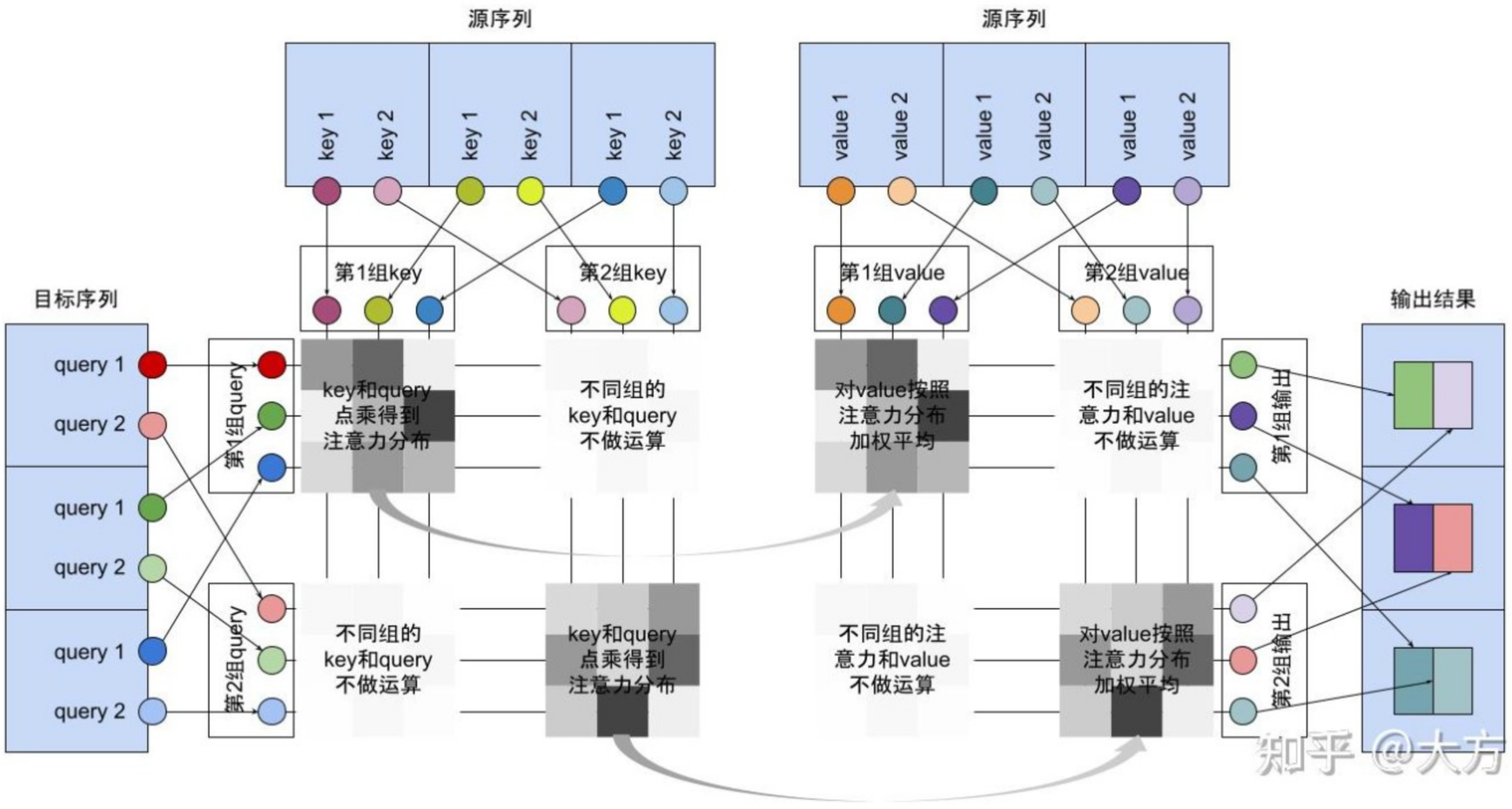
多头注意力是在上述注意力机制的基础上进行的一个简单的改进。“多头”指的是把每个query，key和value沿个各自的维度分割成若干段，形成若干独立的query-key-value分组，每个分组内进行点积运算和value加权平均，之后再把各组产生的结果连接起来。比如，如果组数量为 G ，那么每个query和key的维度只有 D_k/G ，每个value的维度只有 D_v/G ；query的第一段只与其他元素的key的第一段做点乘，其结果也只作为value第一段的权重；query的第二段只与其他元素的key的第二段做点乘，其结果也只作为value第二段的权重，以此类推。

$$\mathbf{q} = [\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_G], \mathbf{k}_i = [\mathbf{k}_{i,1}, \mathbf{k}_{i,2}, \ldots, \mathbf{k}_{i,G}], \mathbf{v}_i = [\mathbf{v}_{i,1}, \mathbf{v}_{i,2}, \ldots, \mathbf{v}_{i,G}]$$

$$\mathbf{y}_j = \sum_{i=1}^{N} S\left(\mathbf{q}_j, \mathbf{k}_{i,j}\right) \mathbf{v}_{i,j}, \quad j = 1, 2, \ldots, G$$

$$\mathbf{y} = [\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_G]$$

这样设计的好处是，允许目标元素使用不止一种选择源元素的标准，因为不同组之间的注意力分布可以不同。比如，上文中译英的例子里，输出“an”的注意力分布中不但关注输入中的“一套”，也关注“自动驾驶”，但这两个通路的含义是完全不同的，一个是提取语义，一个是通过首字母决定不定冠词的变体，因此与其强迫用同一个query同时匹配到“一套”的key和“自动驾驶”的key，不如学习两个不同的query，一个去寻找源序列中的语义对应（注意力在“一套”），一个去定位下文的拼写信息（注意力在“自动驾驶”），然后从各自value中提取相应的信息。多头注意力就提供了这样的支持，使得每个目标元素可以根据多种选择标准（query，key）来灵活地选择源元素中的各类特征信息。下图展示了分组数量 $G = 2$ 的多头注意力的计算过程。



尺度补偿

在Transformer之前，点积注意力在高维度下表现不太好，Transformer作者之一Shazeer认为可能跟query和key的尺度缩放问题有关。由于query和key都是独立学习的，query和key向量中的每个系数都有大致固定的尺度（这取决于计算query和key的全连接层参数，因此是不太可控的；作为向量，query和key都没有特意做归一化），因此每个对应系数的积也有固定的尺度（也就是固定的方差），也就是说，维度越高，点积 $\mathbf{q} \cdot \mathbf{k}$ 值的尺度就会越大（点积方差正比于维度数量）。表示复杂的语义概念需要很高的query和key维度，造成很大的点积绝对值，会在softmax中造成问题，因为其中用到指数运算，绝对值很大的点积在训练中会收到几乎为0的梯度，导致训练进展缓慢。对此，Transformer的解决方法是在点积 $\mathbf{q} \cdot \mathbf{k}$ 上除以 $\sqrt{D_k}$ ，这就正好抵消了维度增加造成的点积尺度放大效应，保证了不论维度多高，点积的方差都是恒定的。这是一个简单的小技巧，但背后反映了在对底层计算的细致观察，在深度学习领域，这样的小技巧造就性能卓越的模型的例子比比皆是。

系列链接：大方：「QBlog 08」解剖Transformer系列