第6章 注意力机制

简单介绍:
在RNN，LSTM中，我们丢进去一句话，输出一句话，可以看成一个seq2seq，进一步，我们还将这句话求和平均，
得到了这句话的聚合表示，可以看成是这句话的编码向量。
注意力其实也是在做同样的事，就是丢进去一句话，出来一个编码表示，这个编码可以是seq，也可以是一个vector

本章主要涉及到的注意力：

这里主要介绍最基本的3种注意力机制的原理及实现，分别为：加法注意力，乘积注意力，自注意力。
这3种注意力都可以统一为q，k，v模式，什么是qkv模式呢？就是统一的计算形式，就是q=query是一个
或多个查询向量，从k=key中提取信息，s(q,k)就是q从k提取信息的函数，也叫打分函数，接着把分数
归一化，这步叫做计算注意力分布，最后利用注意力对v=value的多个向量作加权求和。
qkv模式计算过程比较抽象，因为此时q,k,v只是三个随便抽象出来的向量，没有讲具体应用场景。
这里假如k就是一个句子的多个单词=[k1,k2,k3,k4]，q是一个随便初始化的可学习向量，然后q分别与
k1,k2,k3,k4运算，得到4个分数s1,s2,s3,s4，然后归一化变成$\alpha_1,\alpha_2,\alpha_3,\alpha_4$,
然后v还是取这句话[k1,k2,k3,k4]，我们利用注意力对这句话加权求和：$\alpha_1*k_1,\alpha_2*k_2,\alpha_3*k_3,\alpha_4*k_4$
就得到了这句话最终的一个表示，由于这里q是可学习参数，它能根据任务自动调整为合适的值，然后就能
合理的从句子中提取到有用的信息,算出来的$\alpha_1,\alpha_2,\alpha_3,\alpha_4$正是对这4个单词的不同关注程度。
为了提高模型的表示能力,学习能力,复杂度,我们这里的q,k,v可能需要稍作改动,后面我们就以这3种注意力的实现为例进行讲解

计算模式(后面的讲解顺序): 注意力打分计算-->注意力计算-->加权平均 其中,注意力打分计算是区分不同注意力的主要部分

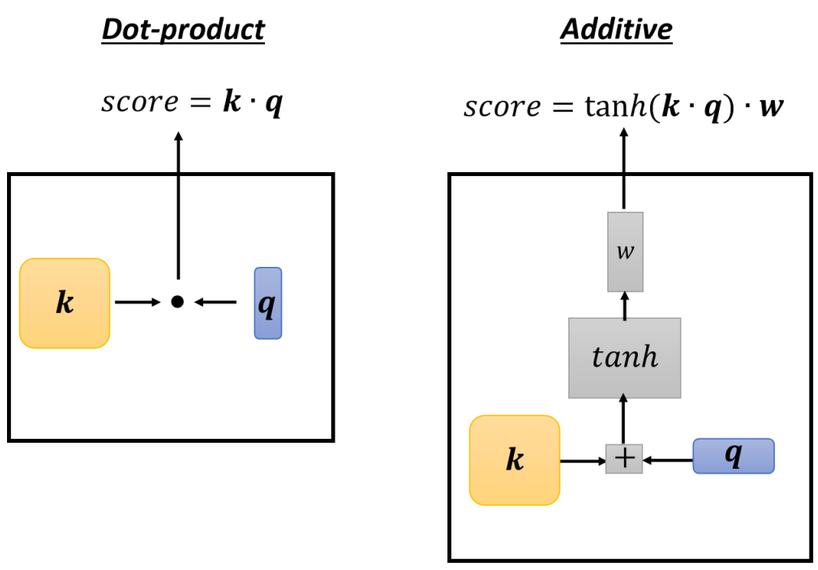
In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
torch.manual_seed(102)
```

Out[1]:
```
<torch._C.Generator at 0x1d10f57dcf0>
```

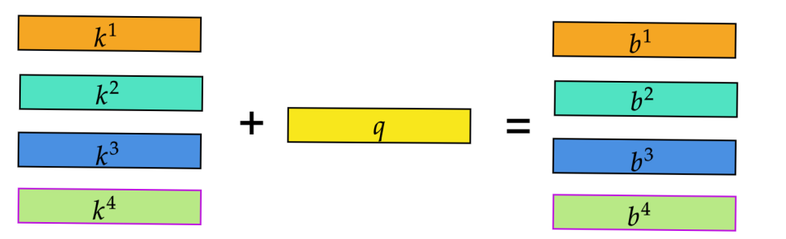
一、注意力打分

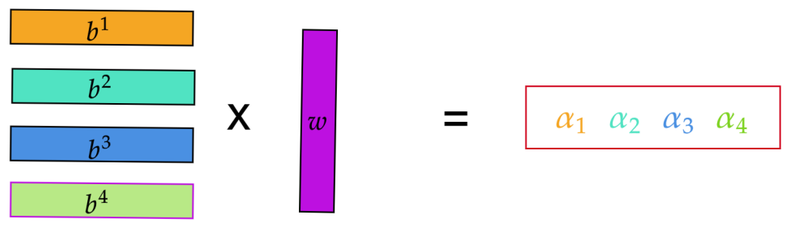
① 加法注意力与乘积注意力

加法注意力: q分别与k每一行相加,然后降维得到打分值
乘积注意力: q分别与k每一行作内积,直接得到q对4行的打分值



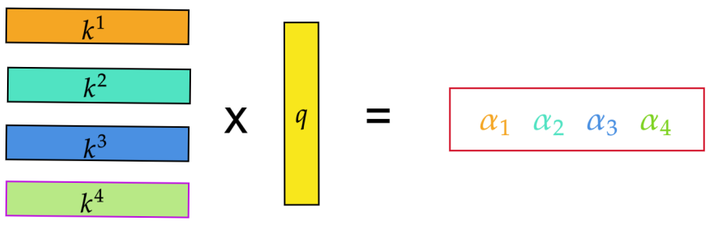
加法注意力：
通过加法,q已经从每个k中提取了信息,但都是向量,不能作为打分的权值,
所以需要变成一个数值,简单的在dim=1维求和即可,
也可以乘以一个列向量w，就是经过一个线性层降维成一个数值



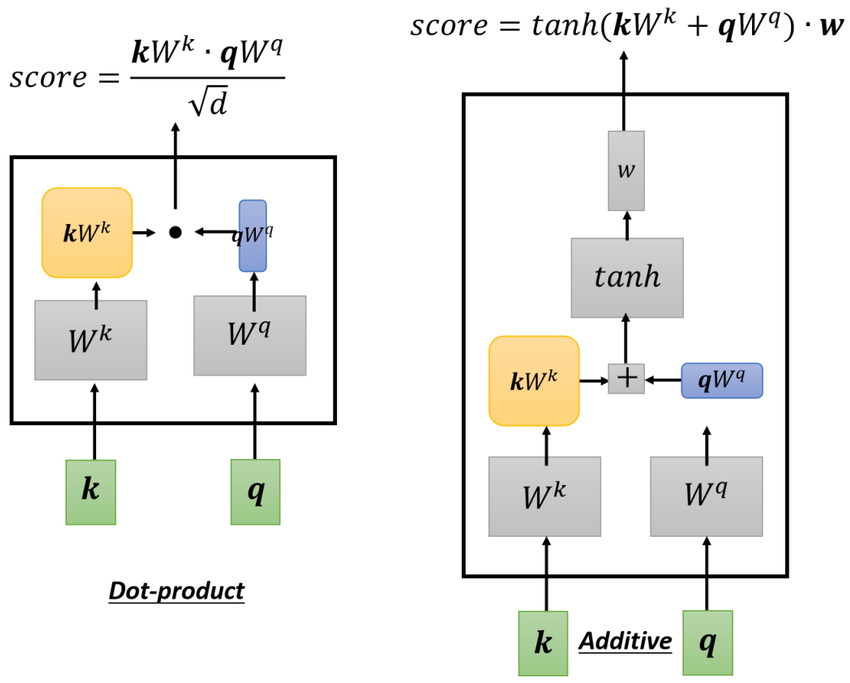


乘积注意力：
这里是通过乘法计算的注意力分数,很好理解,说到这里有人会发出疑问,
为什么加法,乘法就能提取到有用的注意力,难道别的运算不行吗?
答案是都可以,只要是运算都行,实际上通过加法乘法就能计算出合适的
注意力分数,这是神经网络通过学习一步一步调整的,至于它学到什么
管他呢,反正通过学到的q,它就能通过跟k相加或者相乘,算出合适的值,
然后这个值刚好能让你的loss很小,loss变小,你的分类任务算的就是正确的.
实际上卷积核也是类似的东西,他们都只是设置网络参数与计算的一种,
只不过到了不同的地方解释这种运算的说法不一样,这种解释的说法也都是
在近似去解释它的工作原理,工作机制.
统一的工作机制就是:待定系数法,求导优化减少loss,ok,任务完成

不同的网络就是待定系数法设的不同函数,他们的拟合性能不同。



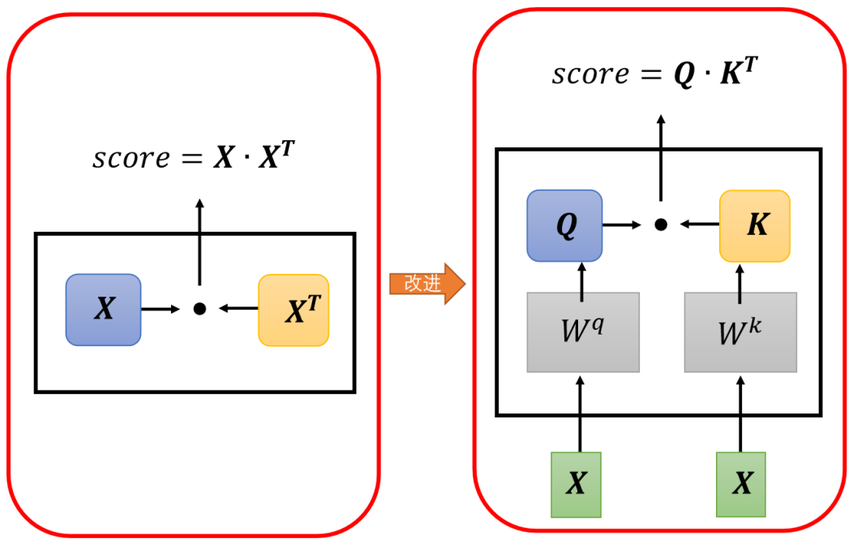
改进: 为了增加网络的表示学习能力,q和k在运算之前,先乘以w做个变换
注意k的多个向量乘以w后,每个k流淌的血液还是分开的,不会混在一起,这里就不细写了



② 自注意力

前面的加法和乘积注意力,它们的key是一句话各个单词的编码向量,而query是随机初始化的查询向量,或者根据任务的需要设置,自注意力与它们的区别主要是这个q取的还是这句话的各个单词,这样就相当于自己从自己身上提取信息,所以叫自注意力,另外,由于q是自己的多个单词向量,算出来的注意力打分就有好几组,分别是各个单词分别提取的注意力分数.每组分数后面都能得到这个句子的一个聚合表示,前面的加法乘法注意力只有一组注意力,所以最后加权求和后只能得到一个聚合表示向量h,而自注意力有了多组自注意力,就能得到这组句子的多个编码向量,这个是打分后面的计算,后面会详细讲解.
由于自注意力的主要不同是query的选取,所以计算方式按理来说也分为加法和乘法,但是这里后面提到的自注意力都是乘法

Self-attention



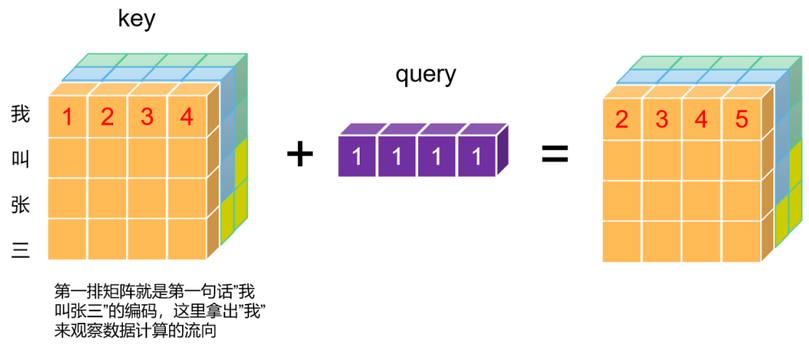
③ 注意力打分模块的实现

这里加法注意力实现的是改进版，乘积注意力实现的是基础版，自注意力实现改进版

加法注意力

In [2]:
```
class AdditiveScore(nn.Module):
    def __init__(self,hidden_size):
        super(AdditiveScore,self).__init__()
        self.W_q=nn.Linear(hidden_size,hidden_size,bias=False)      # DxD
        self.W_k=nn.Linear(hidden_size,hidden_size,bias=False)      # DxD
        self.w=nn.Linear(hidden_size,1,bias=False)                  # Dx1
        self.q=nn.Parameter(torch.randn(size=[1,hidden_size]))      # 1xD
    def forward(self,X):
        B,L,d=X.shape                                               # 注意，d=D
        score=torch.tanh(self.W_k(X)+self.W_q(self.q))              # BxLxD
        score=self.w(score).squeeze(-1)                            # BxLx1 --> BxL  每个句子注意力分值各占一行
        return score
```

加法注意力的批计算数据流



注：下方的q，k是已经乘过W的q，k了

丢进去BxLxd,表示B个句子，每个句子长L，即有L个单词，每个单词维度d
出来BxL，第一行对应第一句话L个词的注意力分数，第二行对应第二句话的分值
由于句子多了，我们这里的颜色只用于区分句子（黄色例外，表示pad）
注意这里pad字符的注意力也被我们算了出来，这里暂时不处理

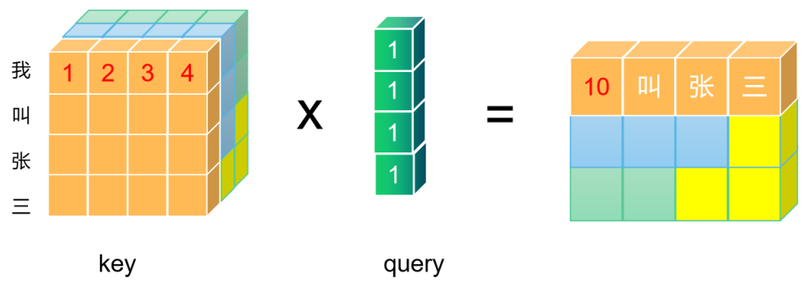
In [3]:
```
## test
add_score=AdditiveScore(4)
X=torch.randn([3,4,4])  # 3句话，每句话4个词，每个词dim=4
add_score(X)            # 输出是：3句话，每句话4个词的注意力
```

Out[3]:
```
tensor([[-0.1606, -0.7549, -0.7688, -0.3724],
        [-0.6060, -0.6850, -0.3164, -0.5156],
        [-0.2841, -0.2565, -0.4438, -0.8402]], grad_fn=<SqueezeBackward1>)
```

乘积注意力

In [4]:
```
class DotProductScore(nn.Module):
    def __init__(self,hidden_size):
        super(DotProductScore,self).__init__()
        self.q=nn.Parameter(torch.randn(size=[hidden_size,1]))      # Dx1
    def forward(self,X):
        B,L,d=X.shape
        score=self.q(X).squeeze(-1)                          # BxLx1 --> BxL
        return score
```

乘积注意力的批计算数据流



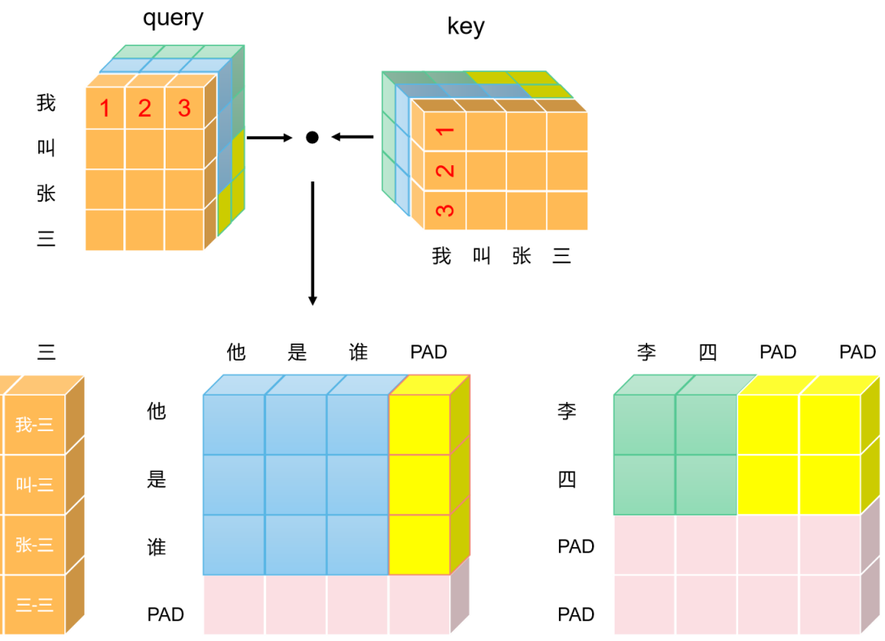
In [5]:
```
## test
dot_score=DotProductScore(4)
X=torch.randn([3,4,4])
add_score(X)
```

Out[5]:
```
tensor([[-0.7574, -0.8108, -0.7375, -0.4197],
        [-0.7355, -0.5016, -0.3604, -0.5905],
        [-0.5023, -0.6090, -0.6071, -0.6563]], grad_fn=<SqueezeBackward1>)
```

自注意力

In [6]:
```
class SelfAttentionScore(nn.Module):
    def __init__(self,hidden_size):
        super(SelfAttentionScore,self).__init__()
        self.W_q=nn.Linear(hidden_size,hidden_size,bias=False)      # DxD
        self.W_k=nn.Linear(hidden_size,hidden_size,bias=False)      # DxD
    def forward(self,X):
        B,L,d=X.shape
        score=torch.matmul(self.W_q(X),self.W_k(X).transpose(1,2))  # BxLxL
        return score
```

自注意力的批计算数据流



注：下方的q，k是已经乘过W的q，k了

丢进去BxLxd,3个句子，每个句子4个单词，每个单词维度3，出来3xLxL，
表示3个句子的注意力分数矩阵，这里直接画出3张矩阵了，没有用魔方，
不然每个方块是怎么计算的不好观察。

先观察第一个句子的注意力矩阵，也就是"我叫张三"的注意力矩阵：
第1行是"我"分别与"我"，"叫"，"张"，"三"作内积，
第2行是"叫"分别与"我"，"叫"，"张"，"三"作内积，
第3行是"张"分别与"我"，"叫"，"张"，"三"作内积，
第4行是"三"分别与"我"，"叫"，"张"，"三"作内积，

再观察第二个句子"他是谁PAD"的注意力矩阵：
第1行是"他"分别与"他"，"是"，"谁"，"PAD" 作内积
第2行是"是"分别与"他"，"是"，"谁"，"PAD" 作内积
第3行是"谁"分别与"他"，"是"，"谁"，"PAD" 作内积
第4行是"PAD"分别与"他"，"是"，"谁"，"PAD" 作内积

从上面可以看出来，当执行批计算的时候，本来我们只需要计算他是谁
这3个词的自注意力打分，但是多计算了一部分，这部分我在图中用别的
颜色进行区分，好在目前为止，pad并没有对我本该计算的东西造成影响，
比如他是谁这句话，出来的4x4矩阵，蓝色区域的结果就是他是谁的自注意力，
pad对这部分并没有任何影响，单独算也会得到蓝色区域同样的结果。
至于其他部分对后续计算有什么影响，怎么处理来消除这种影响，下面就来分析。

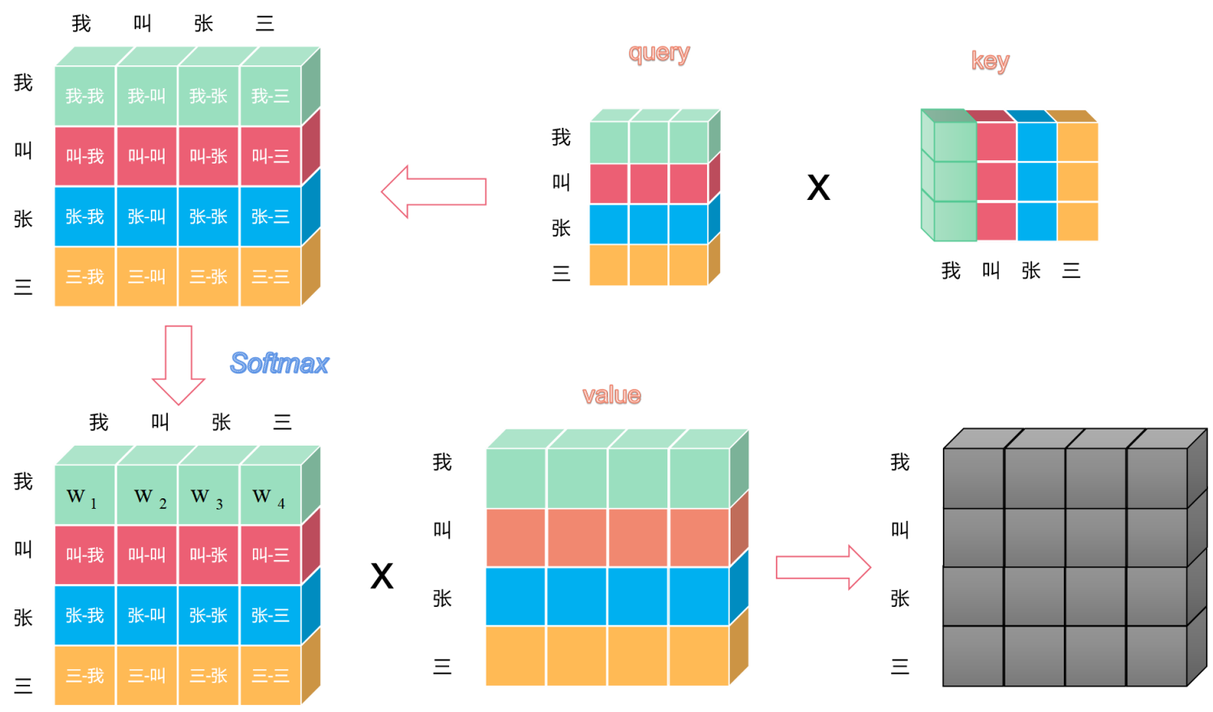
In [7]:
```
## test
self_attn_score=SelfAttentionScore(4)
X=torch.randn([3,4,4])
self_attn_score(X)
```

Out[7]:
```
tensor([[[-1.0250, -1.4075,  1.5705, -0.3260],
         [ 0.6683,  1.2705, -1.4100, -0.8153],
         [ 0.2808,  0.8430, -0.5327, -0.0906],
         [-0.4378, -0.8837,  0.6344,  0.6777]],

        [[ 0.7714, -0.3381, -0.2805,  0.2527],
         [ 0.1213, -0.1705, -0.1485, -0.0239],
         [-0.7505,  0.5079,  0.1091, -0.3481],
         [ 0.1738, -0.1089,  0.0558,  0.0988]],

        [[-0.0215,  0.3536, -0.8061,  0.0252],
         [ 0.7424,  0.0893,  0.5168, -0.5696],
         [-0.2335,  0.2555, -0.8253,  0.3144],
         [-0.3693, -0.4938,  0.6376,  0.2589]]], grad_fn=<UnsafeViewBackward0>)
```

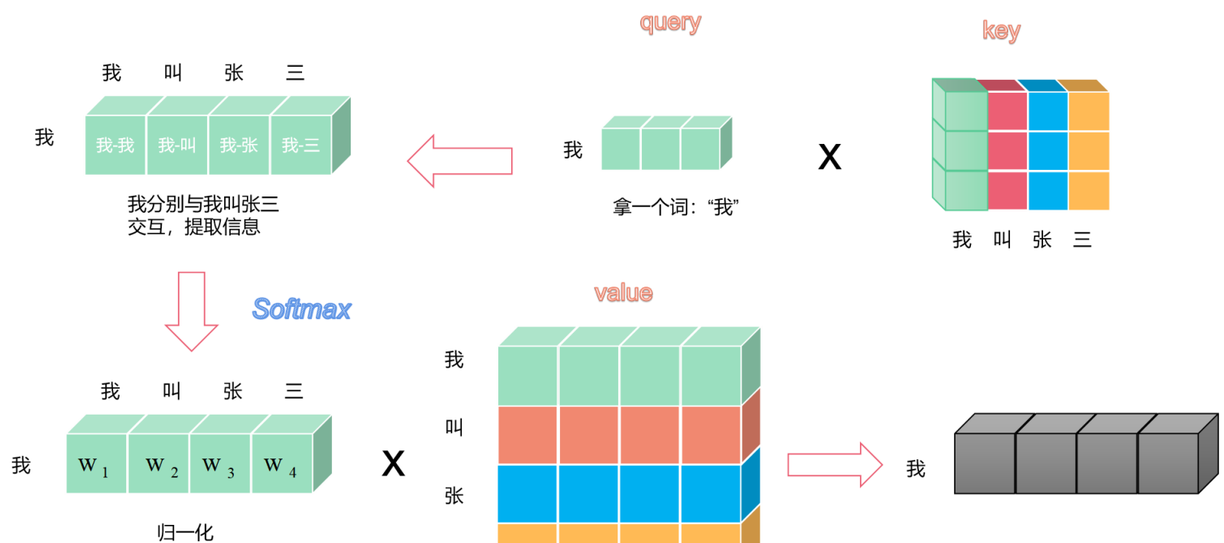
二、注意力分布计算与加权求和

承接上部分，当我们算完注意力分数，紧接着就是算注意力分布，然后根据注意力分布
对原始句子(或者先对它进行线性映射增强模型的表示能力)进行加权求和，这里先拿一句话
为例完整把这个过程演示一遍

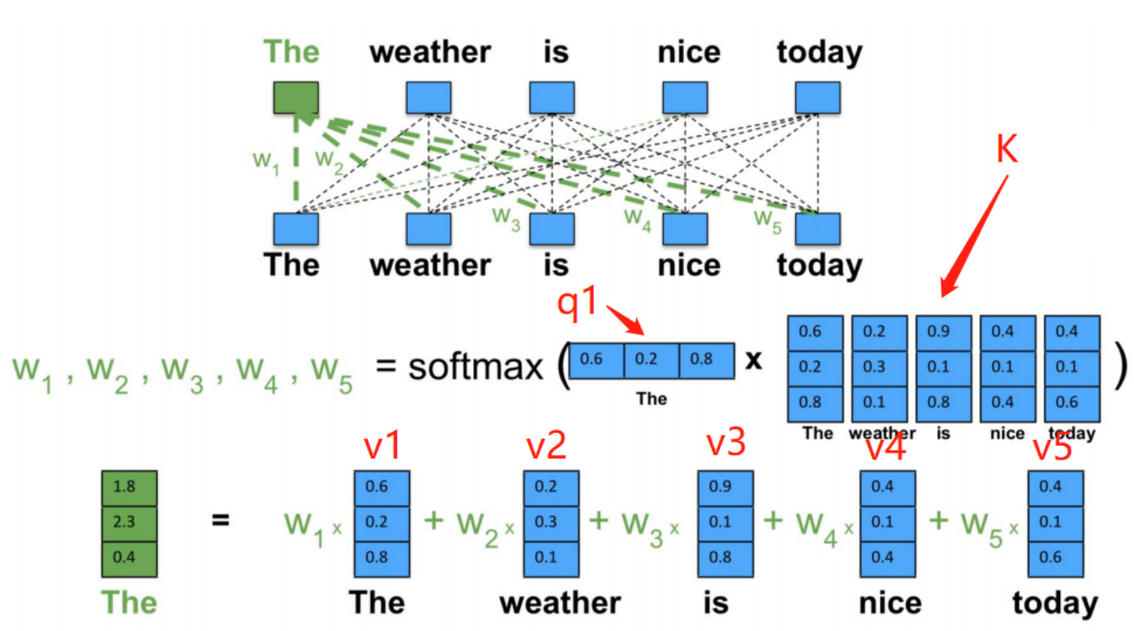
注意力打分--> Softmax归一化--> 加权求和



单独分析其中一个单词的计算动向



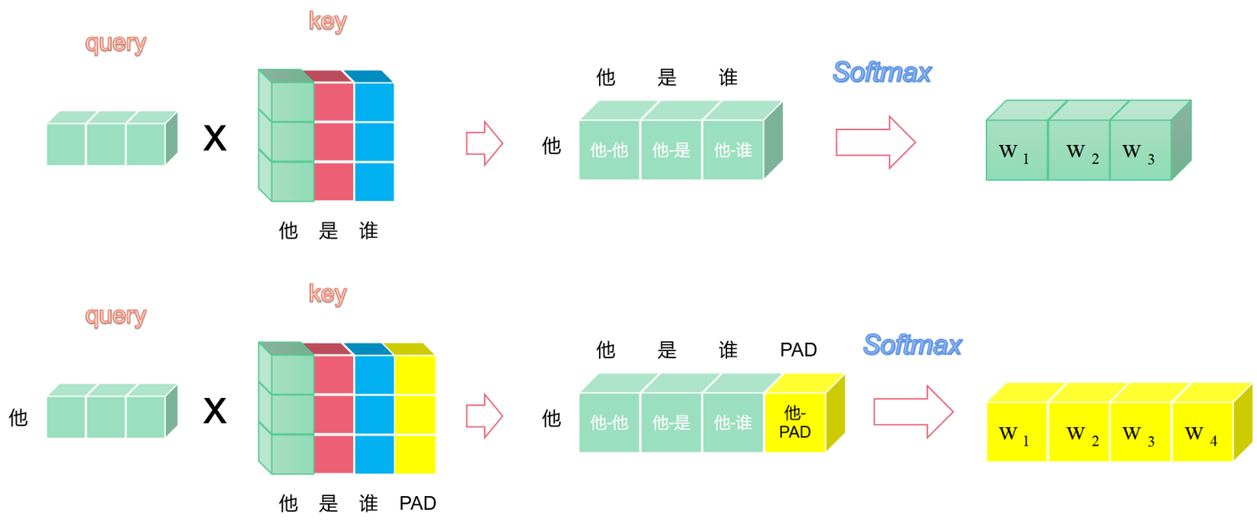
另一种解释



前面拿我叫张三为例，用我分别与我叫张三计算注意力，然后归一化得出注意力分布，
最后用分布对我叫张三进行加权求和。这里使用之前用过的一张图，改成另一句话，
按照同样的思路分析单词The的计算过程，同样也是分别跟所有词交互得到打分，然后
归一化得到权值，最后加权求和，从最后加权求和我们可以看出，最后算出来的向量是所有词
的线性组合了，其组合系数是The与整句话交互得到的，所以最后这个加权求和的向量可以看成
The考虑了整句话(上下文)后的重新编码表示，整句话看成基向量(坐标轴)，注意力分布权值看成坐标。

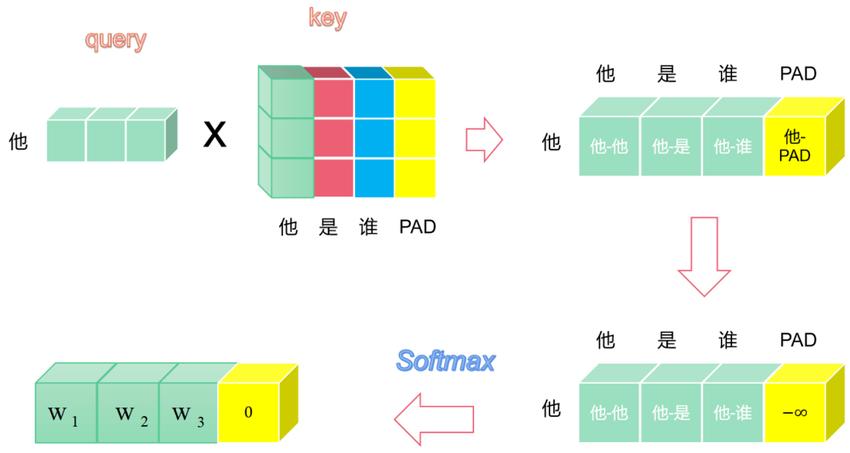
处理pad

通过上面的计算流程，接下来讨论如何处理pad比较合理



从上图可以看出，他与整句话交互，分别作内积提取信息的时候，有没有pad都不影响，
但是通过softmax归一化的时候，pad就影响了其他3个权值的计算。在有pad的情况下，
如果什么都不做，算出来的东西跟我原本想算的东西不相等，不是一个东西。

进行修正



简单证明如下：
 不使用pad时：

$$Softmax([x_1,x_2,x_3]) = [\frac{exp(x_1)}{exp(x_1)+exp(x_2)+exp(x_3)}, \frac{exp(x_2)}{exp(x_1)+exp(x_2)+exp(x_3)}, \frac{exp(x_3)}{exp(x_1)+exp(x_2)+exp(x_3)}] = [w_1,w_2,w_3]$$

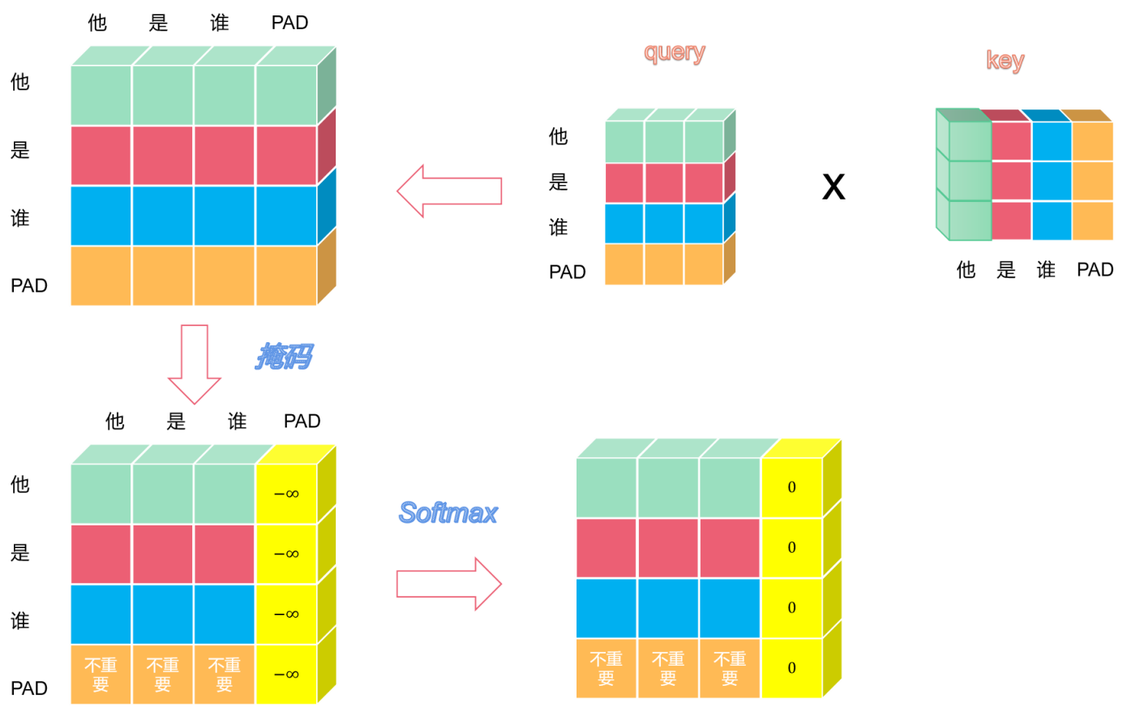
使用pad并按上图掩码时：

$$Softmax([x_1,x_2,x_3,-\infty]) = [\frac{exp(x_1)}{exp(x_1)+exp(x_2)+exp(x_3)+exp(-\infty)}, \frac{exp(x_2)}{exp(x_1)+exp(x_2)+exp(x_3)+exp(-\infty)}, \frac{exp(x_3)}{exp(x_1)+exp(x_2)+exp(x_3)+exp(-\infty)}, \frac{exp(-\infty)}{exp(x_1)+exp(x_2)+exp(x_3)+exp(-\infty)}]$$
$$= [\frac{exp(x_1)}{exp(x_1)+exp(x_2)+exp(x_3)+0}, \frac{exp(x_2)}{exp(x_1)+exp(x_2)+exp(x_3)+0}, \frac{exp(x_3)}{exp(x_1)+exp(x_2)+exp(x_3)+0}, \frac{0}{exp(x_1)+exp(x_2)+exp(x_3)+0}] = [w_1,w_2,w_3,0]$$

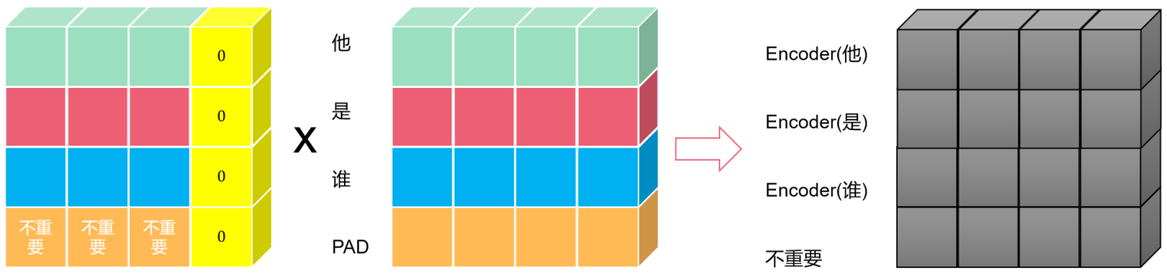
从上面的证明要意识到这些非常简单的事实：
1.pad的出现是因为句子不等长，无法把句子组装起来进行矩阵并行运算，有了pad就能组装进行并行计算。
2.当pad参与进来计算的时候，一定要保证和一句话单独计算的结果一样，因为单独计算一句话的过程才是我设计的模型。

处理一句话的掩码

通过上面可知，一个词在与其他词交互后，在进行softmax之前，需要把这个词与pad交互计算的内容掩码变成−∞，
然而处理一句话的时候，需要如何处理呢？下面这张图回答了这一问题

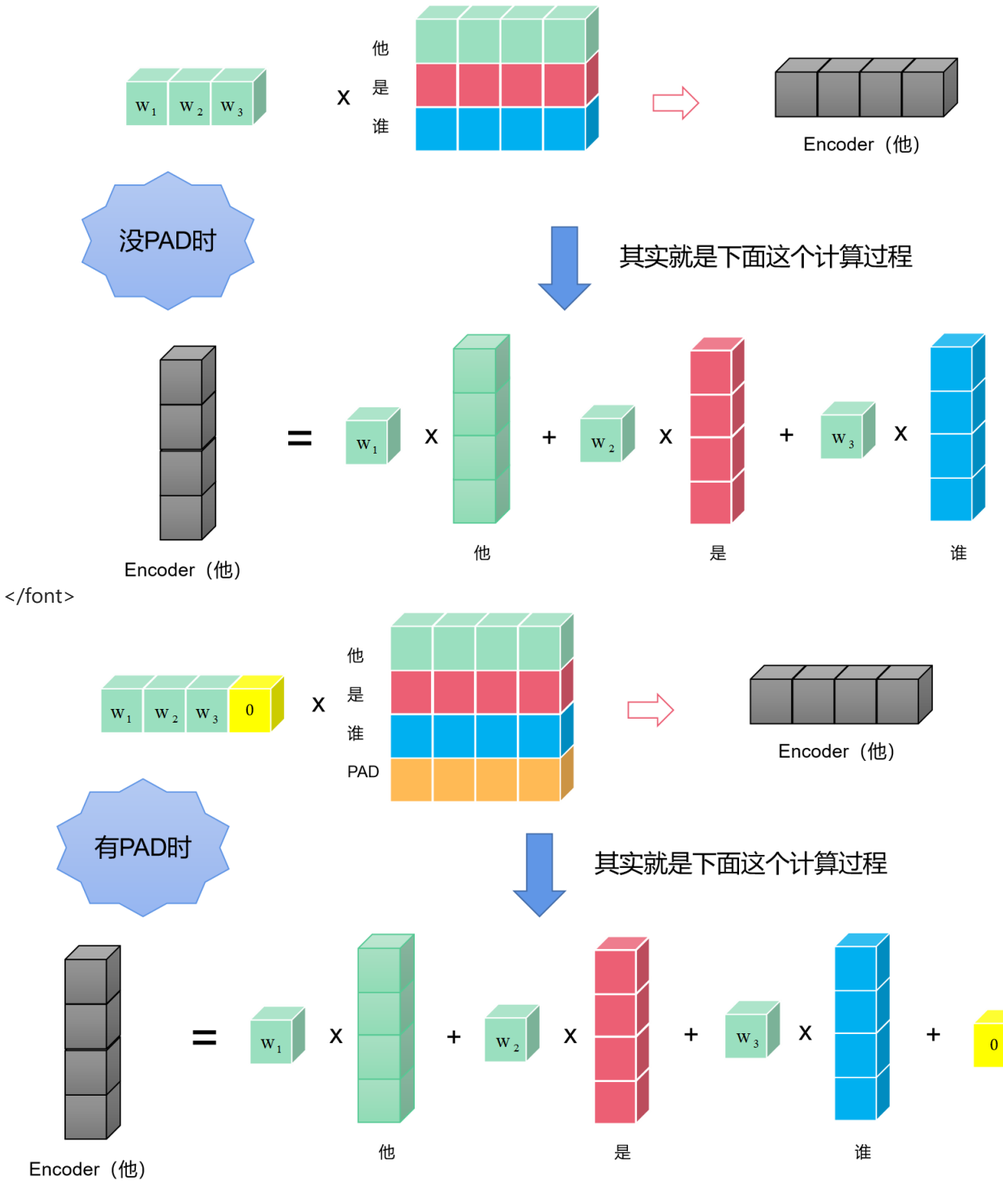


计算注意力打分-->掩码(令每个单词与pad作内积的那列为−∞)-->Softmax计算注意力分布



加权求和

我们单独看一个词“他”的计算过程，验证经过上面这么处理的计算跟单独计算结果完全一样
首先掩码使得前3个注意力分布计算跟没pad的计算相等前面已经证明，现在证明后面加权求和
得到的向量Encoder(他)也跟没pad时的计算结果相等

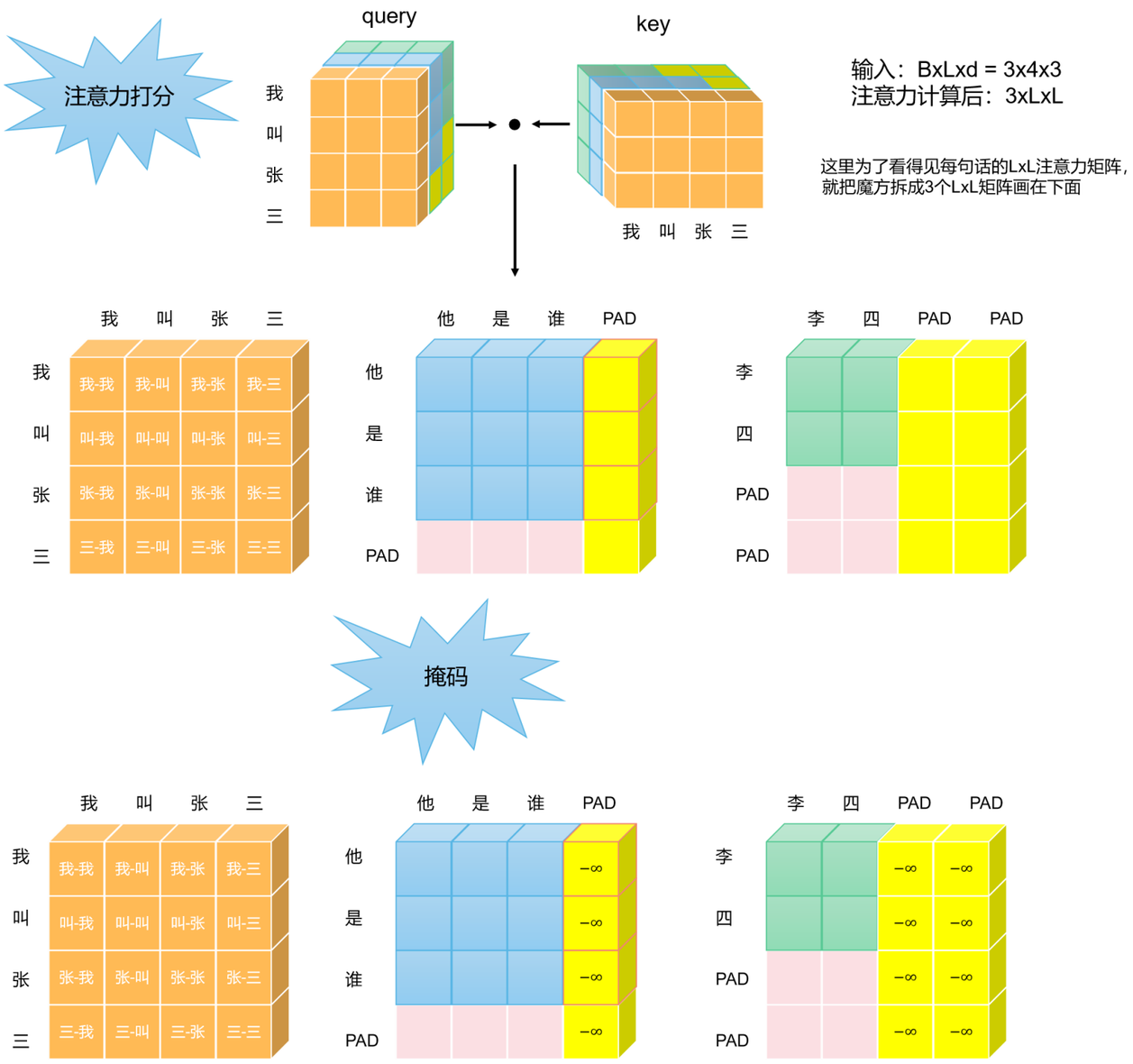


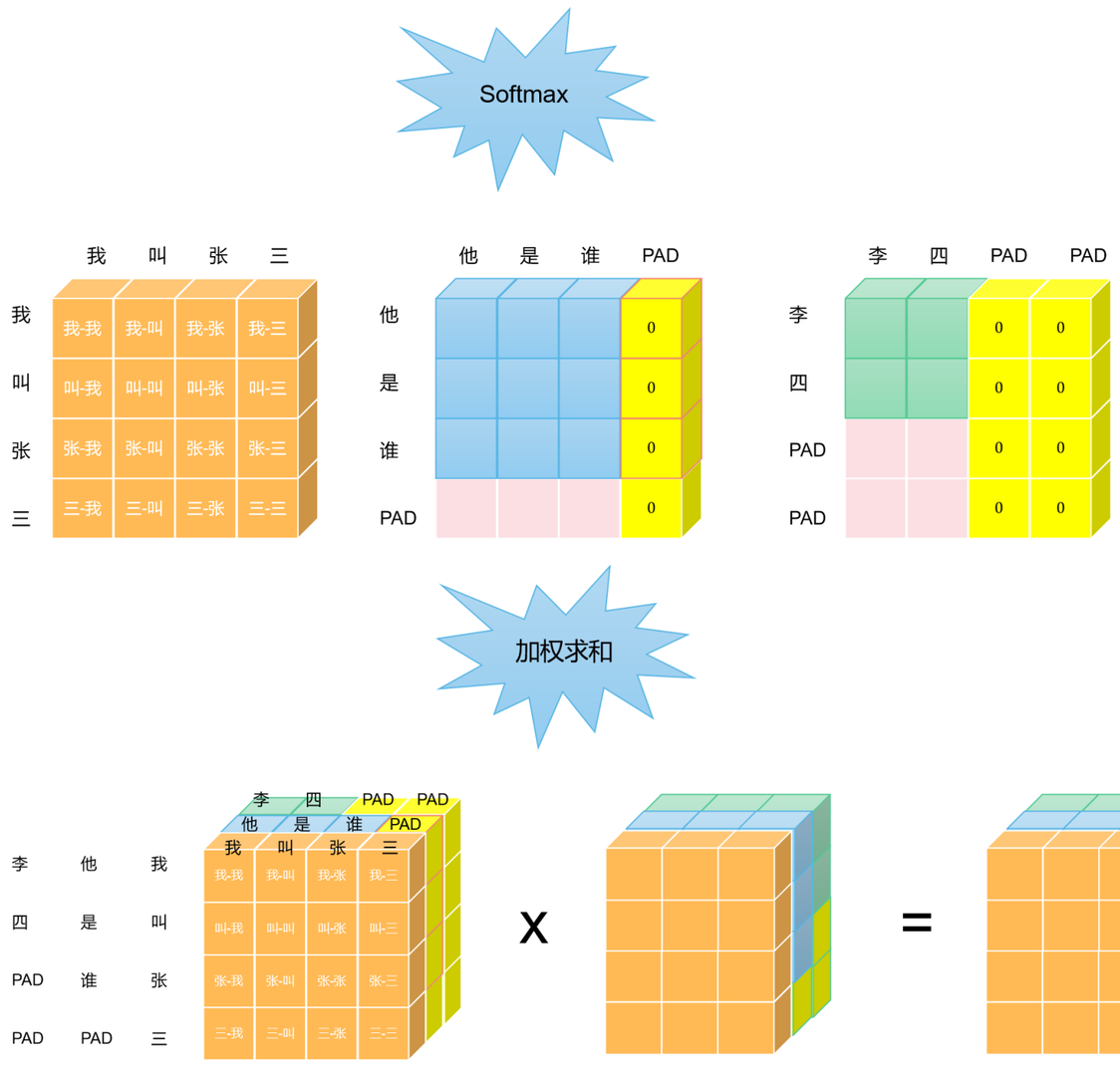
由此可见，在softmax之前掩码后，最后加权求和的结果也完全正确，所以pad后，这样掩码算出来的结果完全正确

处理一个批次(多句话)的掩码

回顾一句话的处理流程：
打分得到LxL矩阵（L个词分别与L个词交互计算）-->掩码(令每个单词与PAD作内积的那列为$-\infty$)-->Softmax计算注意力分布-->加权求和

那么B句话的处理流程：
打分得到BxLxL矩阵-->掩码(每句话按照各自的PAD进行掩码)-->Softmax计算注意力分布-->加权求和





三、自注意力模块设计

前面第一部分已经把三种注意力打分的类写了出来，第二部分着重对自注意力的后续计算进行了详细讲解，
由于讲清楚一批句子在pad+掩码的操作下整个过程的计算比较繁琐，所以花了大量篇幅详细讲解这个过程，
进而忽略了代码实现，因为中间再穿插代码实现，全都混在一起，就更复杂了，索性代码放在这部分实现。

接口说明：

首先我们要实现的接口如下：

$$Attention(q,k,v) = Softmax(\frac{Q \cdot K^T}{\sqrt{D}}) \cdot V$$

其中，

$$Q = XW_q \quad , \quad K = XW_k \quad , \quad V = XW_v$$

输入：q,k,v,简单起见，它三个都是X，形状BxLxD
输出：BxLxD的编码立方体，这就是Encoder(X)，并且PAD的编码部分会变成0
其中打分后除以$\sqrt{D}$是因为向量内积后(很多随机变量相乘再累加方差变大)
数值很大，直接参与后面的计算会导致这步梯度很大，不利于优化

修正后的接口：

$$Masked\_Score = Mask(\frac{Q \cdot K^T}{\sqrt{D}})$$

$$Attention(Q,K,V) = Softmax(Masked\_Score) \cdot V$$

其中mask的作用是让score的pad列变负无穷

在直接写这个类之前，为了使得整个过程清晰，还是现在外面过一遍计算流程

第1步：

初始化一批句子的编码立方体，并变换得到Q,K,V.(保持词编码维度不变)
假设3句话，第一句话4词，第二句话3词，第三句话2词，每个词维度3

In [8]:
```
# 这两个参数是从后面官方api接口随机初始化的参数里面复制过来的，就拿这个参数进行计算，方便比较结果
W=torch.tensor([[ 0.4926, -0.7041, -0.4945],
        [ 0.6348, -0.2868, -0.4850],
        [-0.2386,  0.1559, -0.5426],
        [-0.0833,  0.3115,  0.6994],
        [ 0.2863, -0.1040,  0.5505],
        [-0.0437,  0.2370,  0.1977],
        [-0.4363, -0.4253,  0.3783],
        [ 0.2342,  0.2743,  0.5620],
        [-0.5765,  0.6763,  0.6810]], requires_grad=True)

W_out=torch.tensor([[ 0.4881,  0.4734, -0.3643],
        [-0.4775,  0.4050,  0.0622],
        [-0.5061,  0.0370, -0.5362]], requires_grad=True)
```

In [9]:
```
x=torch.randn(size=[3,4,3])
seq_lens=[4,3,2]
W_q=W[:3,:]
W_k=W[3:6,:]
W_v=W[6:,:]
Q=torch.matmul(x,W_q.T)
K=torch.matmul(x,W_k.T)
V=torch.matmul(x,W_v.T)
Q.shape,K.shape,V.shape
```

Out[9]:
```
(torch.Size([3, 4, 3]), torch.Size([3, 4, 3]), torch.Size([3, 4, 3]))
```

第2步：

计算注意力分数，并掩码

In [10]:
```
def score_mask(score,seq_lens):
    max_len=score.shape[1]
    seq_lens=torch.tensor(seq_lens).unsqueeze(-1)  # Bx1
    mask=torch.arange(max_len)<seq_lens            # BxL
    mask=mask.unsqueeze(1)                         # Bx1xL , 注意后面是对列掩码
    masked_score=score.masked_fill(~mask,-1e9)
    return masked_score
```

In [11]:
```
attn_score=torch.matmul(Q,K.transpose(1,2))/torch.sqrt(torch.tensor(3.))
attn_score
```

Out[11]:
```
tensor([[[-0.0170, -0.1186,  0.1650, -0.0501],
         [-0.4638, -0.3019, -0.5509, -0.3744],
         [ 0.1350,  0.0521,  0.3532,  0.1233],
         [-0.2920, -0.2106, -0.2852, -0.2374]],

        [[ 0.0859, -0.6261,  0.4080,  0.0228],
         [-0.0114, -1.2626,  0.6670, -0.1373],
         [ 0.1461, -0.1723,  0.2271,  0.1562],
         [ 0.2529, -1.5483,  1.0716,  0.1674]],

        [[-0.0907, -0.3286, -0.2191, -0.1593],
         [-0.4125, -1.7913, -0.6960, -0.3861],
         [-0.2193, -1.1654, -0.4594, -0.3091],
         [-0.1604, -1.0057, -0.3043, -0.1917]]], grad_fn=<DivBackward0>)
```

In [12]:
```
masked_score=score_mask(attn_score,seq_lens)
masked_score
```

Out[12]:
```
tensor([[[-1.7017e-02, -1.1863e-01,  1.6498e-01, -5.0083e-02],
         [-4.6383e-01, -3.0191e-01, -5.5091e-01, -3.7439e-01],
         [ 1.3504e-01,  5.2138e-02,  3.5321e-01,  1.2326e-01],
         [-2.9205e-01, -2.1059e-01, -2.8524e-01, -2.3737e-01]],

        [[ 8.5884e-02, -6.2605e-01,  4.0801e-01, -1.0000e+09],
         [-1.1372e-02, -1.2626e+00,  6.6700e-01, -1.0000e+09],
         [ 1.4613e-01, -1.7234e-01,  2.2708e-01, -1.0000e+09],
         [ 2.5290e-01, -1.5483e+00,  1.0716e+00, -1.0000e+09]],

        [[-9.0675e-02, -3.2860e-01, -1.0000e+09, -1.0000e+09],
         [-4.1246e-01, -1.7913e+00, -1.0000e+09, -1.0000e+09],
         [-2.1926e-01, -1.1654e+00, -1.0000e+09, -1.0000e+09],
         [-1.6037e-01, -1.0057e+00, -1.0000e+09, -1.0000e+09]]],
       grad_fn=<MaskedFillBackward0>)
```

第3步：

计算注意力分布

In [13]:
```
atten=F.softmax(masked_score,dim=2)
atten
```

Out[13]:
```
tensor([[[0.2457, 0.2219, 0.2947, 0.2377],
         [0.2389, 0.2809, 0.2190, 0.2612],
         [0.2408, 0.2217, 0.2995, 0.2380],
         [0.2411, 0.2615, 0.2427, 0.2546]],

        [[0.3483, 0.1709, 0.4807, 0.0000],
         [0.3070, 0.0879, 0.6051, 0.0000],
         [0.3557, 0.2587, 0.3857, 0.0000],
         [0.2913, 0.0481, 0.6606, 0.0000]],

        [[0.5592, 0.4408, 0.0000, 0.0000],
         [0.7988, 0.2012, 0.0000, 0.0000],
         [0.7203, 0.2797, 0.0000, 0.0000],
         [0.6996, 0.3004, 0.0000, 0.0000]]], grad_fn=<SoftmaxBackward0>)
```

第4步：加权求和

本来加权求和这个注意力计算的部分应该就算算完了才对，但是官方接口最后还接了一个线性层， 为了对比计算结果，这里也做个线性层计算吧

In [14]:
```
out=torch.matmul(atten,V)
out
```

Out[14]:
```
tensor([[[ 3.1098e-02,  5.5693e-01,  2.5170e-01],
         [-1.3614e-02,  5.4281e-01,  3.5157e-01],
         [ 3.7487e-02,  5.5560e-01,  2.4869e-01],
         [ 4.1097e-04,  5.4734e-01,  3.1991e-01]],

        [[ 6.8953e-02,  1.9005e-01, -5.0370e-01],
         [ 6.4709e-02,  3.8920e-01, -4.0146e-01],
         [ 6.2444e-02,  7.4336e-03, -5.9470e-01],
         [ 6.3902e-02,  4.8145e-01, -3.5440e-01]],

        [[-2.5525e-01,  6.8594e-01,  1.0380e+00],
         [-8.9557e-01,  6.7747e-01,  7.1594e-01],
         [-6.8588e-01,  6.8024e-01,  8.2142e-01],
         [-6.3042e-01,  6.8097e-01,  8.4932e-01]]],
       grad_fn=<UnsafeViewBackward0>)
```

In [15]:
```
torch.matmul(out,W_out.T)
```

Out[15]:
```
tensor([[[ 0.1871,  0.2264, -0.1301],
         [ 0.1222,  0.2482, -0.1615],
         [ 0.1907,  0.2226, -0.1318],
         [ 0.1428,  0.2414, -0.1515]],

        [[ 0.3071,  0.0127,  0.2422],
         [ 0.3621,  0.1018,  0.1969],
         [ 0.2506, -0.0638,  0.2876],
         [ 0.3882,  0.1424,  0.1755]],

        [[-0.1780,  0.4642, -0.4020],
         [-0.3772,  0.7465,  0.0944],
         [-0.3120,  0.6541, -0.0682],
         [-0.2947,  0.6296, -0.1112]]], grad_fn=<UnsafeViewBackward0>)
```

使用torch的API计算

初始化模型，选择一个头，并固定随机种子打印出参数，复制到上面来手动计算并对比结果

In [16]:
```
multihead_attention=nn.MultiheadAttention(embed_dim=3,num_heads=1,batch_first=True)
```

In [17]:
```
for param in multihead_attention.parameters():
    print(param)
```

```
Parameter containing:
tensor([[ 0.4926, -0.7041, -0.4945],
        [ 0.6348, -0.2868, -0.4850],
        [-0.2386,  0.1559, -0.5426],
        [-0.0833,  0.3115,  0.6994],
        [ 0.2863, -0.1040,  0.5505],
        [-0.0437,  0.2370,  0.1977],
        [-0.4363, -0.4253,  0.3783],
        [ 0.2342,  0.2743,  0.5620],
        [-0.5765,  0.6763,  0.6810]], requires_grad=True)
Parameter containing:
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0.], requires_grad=True)
Parameter containing:
tensor([[ 0.4881,  0.4734, -0.3643],
        [-0.4775,  0.4050,  0.0622],
        [-0.5061,  0.0370, -0.5362]], requires_grad=True)
Parameter containing:
tensor([0., 0., 0.], requires_grad=True)
```

掩码使用atten_mask实现

In [18]:
```
max_len=4
seq_lens_=torch.tensor(seq_lens).unsqueeze(-1)  # Bx1
mask=torch.arange(max_len)<seq_lens_            # BxL
mask=mask.unsqueeze(1)                          # Bx1xL , 注意后面是对列掩码
mask=(mask+torch.zeros([3,4,4])).bool()         # 广播成BxLxL
mask
```

Out[18]:
```
tensor([[[ True,  True,  True,  True],
         [ True,  True,  True,  True],
         [ True,  True,  True,  True],
         [ True,  True,  True,  True]],

        [[ True,  True,  True, False],
         [ True,  True,  True, False],
         [ True,  True,  True, False],
         [ True,  True,  True, False]],

        [[ True,  True, False, False],
         [ True,  True, False, False],
         [ True,  True, False, False],
         [ True,  True, False, False]]])
```

In [19]:
```
out_and_attn=multihead_attention(x,x,x,attn_mask=~mask)
out_and_attn
```

Out[19]:
```
(tensor([[[ 0.1871,  0.2263, -0.1301],
          [ 0.1222,  0.2481, -0.1615],
          [ 0.1907,  0.2225, -0.1318],
          [ 0.1428,  0.2413, -0.1515]],

         [[ 0.3071,  0.0127,  0.2422],
          [ 0.3621,  0.1017,  0.1969],
          [ 0.2506, -0.0638,  0.2875],
          [ 0.3882,  0.1424,  0.1755]],

         [[-0.1780,  0.4642, -0.4020],
          [-0.3773,  0.7465,  0.0944],
          [-0.3120,  0.6540, -0.0682],
          [-0.2948,  0.6296, -0.1112]]], grad_fn=<TransposeBackward0>),
 tensor([[[0.2457, 0.2219, 0.2947, 0.2377],
          [0.2389, 0.2809, 0.2190, 0.2612],
          [0.2408, 0.2217, 0.2995, 0.2380],
          [0.2411, 0.2615, 0.2427, 0.2546]],

         [[0.3483, 0.1709, 0.4807, 0.0000],
          [0.3071, 0.0879, 0.6051, 0.0000],
          [0.3557, 0.2586, 0.3857, 0.0000],
          [0.2913, 0.0481, 0.6606, 0.0000]],

         [[0.5592, 0.4408, 0.0000, 0.0000],
          [0.7988, 0.2012, 0.0000, 0.0000],
          [0.7203, 0.2797, 0.0000, 0.0000],
          [0.6996, 0.3004, 0.0000, 0.0000]]], grad_fn=<MeanBackward1>))
```

掩码使用key_padding_mask实现

In [20]:
```
max_len=4
seq_lens_=torch.tensor(seq_lens).unsqueeze(-1)  # Bx1
mask=torch.arange(max_len)<seq_lens_            # BxL
mask
```

Out[20]:
```
tensor([[ True,  True,  True,  True],
        [ True,  True,  True, False],
        [ True,  True, False, False]])
```

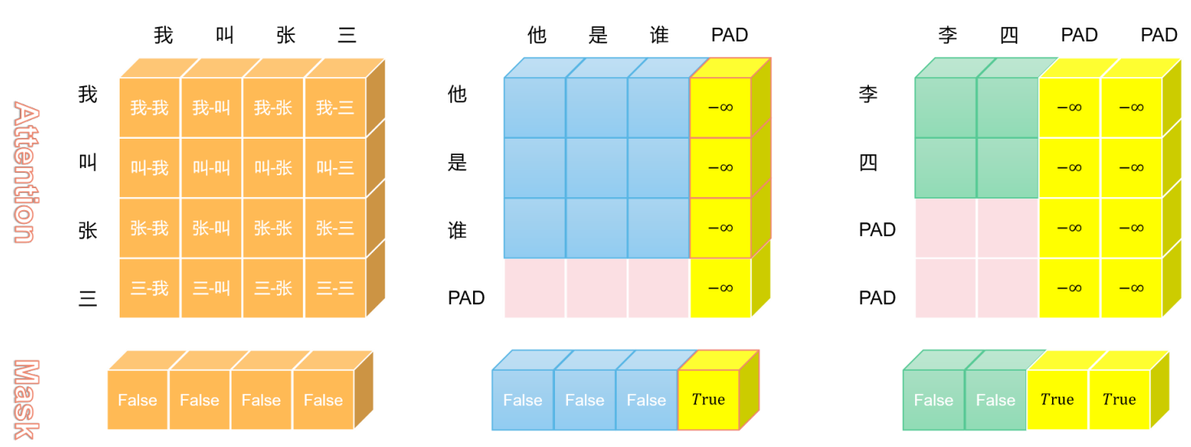
In [21]:
```
out_and_attn=multihead_attention(x,x,x,key_padding_mask=~mask)
out_and_attn
```

Out[21]:
```
(tensor([[[ 0.1871,  0.2263, -0.1301],
          [ 0.1222,  0.2481, -0.1615],
          [ 0.1907,  0.2225, -0.1318],
          [ 0.1428,  0.2413, -0.1515]],

         [[ 0.3071,  0.0127,  0.2422],
          [ 0.3621,  0.1017,  0.1969],
          [ 0.2506, -0.0638,  0.2875],
          [ 0.3882,  0.1424,  0.1755]],

         [[-0.1780,  0.4642, -0.4020],
          [-0.3773,  0.7465,  0.0944],
          [-0.3120,  0.6540, -0.0682],
          [-0.2948,  0.6296, -0.1112]]], grad_fn=<TransposeBackward0>),
 tensor([[[0.2457, 0.2219, 0.2947, 0.2377],
          [0.2389, 0.2809, 0.2190, 0.2612],
          [0.2408, 0.2217, 0.2995, 0.2380],
          [0.2411, 0.2615, 0.2427, 0.2546]],

         [[0.3483, 0.1709, 0.4807, 0.0000],
          [0.3071, 0.0879, 0.6051, 0.0000],
          [0.3557, 0.2586, 0.3857, 0.0000],
          [0.2913, 0.0481, 0.6606, 0.0000]],

         [[0.5592, 0.4408, 0.0000, 0.0000],
          [0.7988, 0.2012, 0.0000, 0.0000],
          [0.7203, 0.2797, 0.0000, 0.0000],
          [0.6996, 0.3004, 0.0000, 0.0000]]], grad_fn=<MeanBackward1>))
```

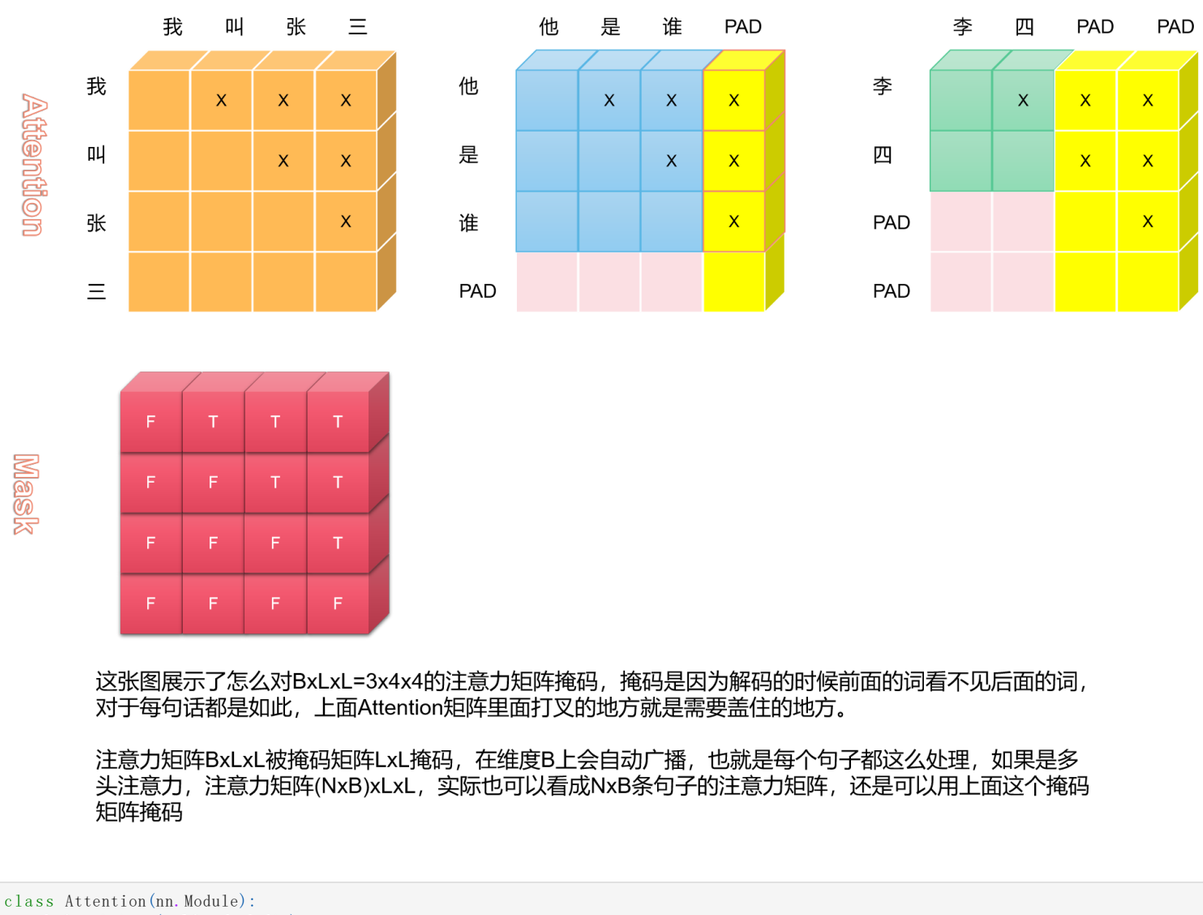
整合上面变成一个Attention类

参考官方接口实现的计算：
step1：从q,k,v出发，计算Q,K,V
step2：计算注意力打分，并根据attn_mask矩阵进行掩码
step3：通过softmax层计算注意力分布
step4：加权求和
step5：对上面的结果再通过一个线性层
输出：最后通过线性层的那个编码立方体，中间计算的注意力分布
因为那个分布后面其他地方有可能需要用到，所以也输出出来

关于两个掩码参数(这里针对单头注意力讲解)：
1.key_padding_mask：提供一个BxL二维矩阵，告诉pad的位置在哪，后面会自动unsqueeze(1)，然后给attention魔方的每一排attention矩阵按列掩码
2.attn_mask:如果是二维矩阵，形状为LxL，对矩阵逐个元素掩码，会自动广播到Batch维，如果我们每个句子都使用相同的掩码方式，比如decoder；
但是上面对于pad的掩码，必须提供一个完整的BxLxL掩码矩阵，因为我们每个句子长度不一样，pad不同，需要掩码的位置不一样，不能用同一个掩码矩阵自动广播到所有的句子
另外，我们在外面算出来的掩码矩阵，pad的位置是false，上面掩码需要pad的位置是true，所以需要取反
由于经常要掩码，并且根据不同的目的掩码，这里还是画张图演示一下吧



这张图展示了怎么对BxLxL=3x4x4的注意力矩阵掩码，掩码目的是为了计算注意力分布时，不能让pad影响到别的词，这个批次第一句话没pad，不需要掩码，第二句话最后一列需要掩码，第三句话最后两列需要掩码，所以掩码矩阵应该是这样的：
mask=[mask1，mask2，mask3]，形状 Bx1xL
mask1=[[False，False，False]]
mask2=[[False，False，True]]
mask3=[[False， True， True]]
然后注意力张量BxLxL和mask掩码张量Bx1xL运算时，在中间这个维度会自动广播，比如
mask2会自动广播成[[False，False，True] ，[False，False，True] ，[False，False，True]]
从1xL广播成LxL，那么mask掩码张量Bx1xL会自动广播成BxLxL



这张图展示了怎么对BxLxL=3x4x4的注意力矩阵掩码，掩码是因为解码的时候前面的词看不见后面的词，对于每句话都是如此，上面Attention矩阵里面打叉的地方就是需要盖住的地方。

注意力矩阵BxLxL被掩码矩阵LxL掩码，在维度B上会自动广播，也就是每个句子都这么处理，如果是多头注意力，注意力矩阵(NxB)xLxL，实际也可以看成NxB条句子的注意力矩阵，还是可以用上面这个掩码矩阵掩码

In [22]:
```
class Attention(nn.Module):
    def __init__(self,embed_dim):
        super(Attention,self).__init__()
        self.W_q=nn.Linear(embed_dim,embed_dim,bias=False)
        self.W_k=nn.Linear(embed_dim,embed_dim,bias=False)
        self.W_v=nn.Linear(embed_dim,embed_dim,bias=False)
        self.W_out=nn.Linear(embed_dim,embed_dim,bias=False)
        self.softmax=nn.Softmax(dim=2)
    def forward(self,q,k,v,key_padding_mask=None,attn_mask=None):
        B,L,D=q.shape
        Q=self.W_q(q)
        K=self.W_k(k)
        V=self.W_v(v)
        attn_score=torch.matmul(Q,K.transpose(1,2))/torch.sqrt(torch.tensor(D))
        if key_padding_mask is not None:
            assert key_padding_mask.shape==torch.Size([B,L])
            mask=key_padding_mask.unsqueeze(1)
            attn_score=attn_score.masked_fill(mask,-1e9)
        if attn_mask is not None:
            assert attn_mask.shape==torch.Size([B,L,L])
            mask=key_padding_mask.unsqueeze(1)
            attn_score=attn_score.masked_fill(mask,-1e9)
        atten=self.softmax(attn_score)
```

```
        out=torch.matmul(atten,V)
        out=self.W_out(out)
        return out,atten
```

测试： 注意全连接层初始化的参数和你脑子里想的是颠倒的的

In [23]:
```
myattention=Attention(3)
myattention.W_q.weight.data=W_q
myattention.W_k.weight.data=W_k
myattention.W_v.weight.data=W_v
myattention.W_out.weight.data=W_out
```

In [24]:
```
myattention(x,x,x,key_padding_mask=~mask)
```

Out[24]:
```
(tensor([[[ 0.1871,  0.2264, -0.1301],
          [ 0.1222,  0.2482, -0.1615],
          [ 0.1907,  0.2226, -0.1318],
          [ 0.1428,  0.2414, -0.1515]],

         [[ 0.3071,  0.0127,  0.2422],
          [ 0.3621,  0.1018,  0.1969],
          [ 0.2506, -0.0638,  0.2876],
          [ 0.3882,  0.1424,  0.1755]],

         [[-0.1780,  0.4642, -0.4020],
          [-0.3772,  0.7465,  0.0944],
          [-0.3120,  0.6541, -0.0682],
          [-0.2947,  0.6296, -0.1112]]], grad_fn=<UnsafeViewBackward0>),
 tensor([[[0.2457, 0.2219, 0.2947, 0.2377],
          [0.2389, 0.2809, 0.2190, 0.2612],
          [0.2408, 0.2217, 0.2995, 0.2380],
          [0.2411, 0.2615, 0.2427, 0.2546]],

         [[0.3483, 0.1709, 0.4807, 0.0000],
          [0.3070, 0.0879, 0.6051, 0.0000],
          [0.3557, 0.2587, 0.3857, 0.0000],
          [0.2913, 0.0481, 0.6606, 0.0000]],

         [[0.5592, 0.4408, 0.0000, 0.0000],
          [0.7988, 0.2012, 0.0000, 0.0000],
          [0.7203, 0.2797, 0.0000, 0.0000],
          [0.6996, 0.3004, 0.0000, 0.0000]]], grad_fn=<SoftmaxBackward0>))
```

四、多头自注意力

什么是多头自注意力

我们之前定义的attention，丢进去x，形状BxLxd，出来BxLxD，并且d=D，这个出来的
几句话各个词的编码立方体，就叫一个注意力头，并且这个输出的每个单词的维数D是由
W_q,W_k,W_v,W_out控制的，从现在开始，我们要区分d和D，具体形状变化如下：
1.计算Q,K,V，它们的形状都是BxLxD，让输入d变成了D
2.计算打分，并归一化，注意力分布矩阵形状为BxLxL
3.注意力分布矩阵乘以V，形状变成了BxLxD ——> 这个东西就叫做一个头
4.最后给这个头接个不改变维数的线性层，输出形状就是BxLxD
上面的步骤1，2，3就是计算一个头的过程，不包括步骤4，这个过程只涉及到3个参数：W_q,W_k,W_v

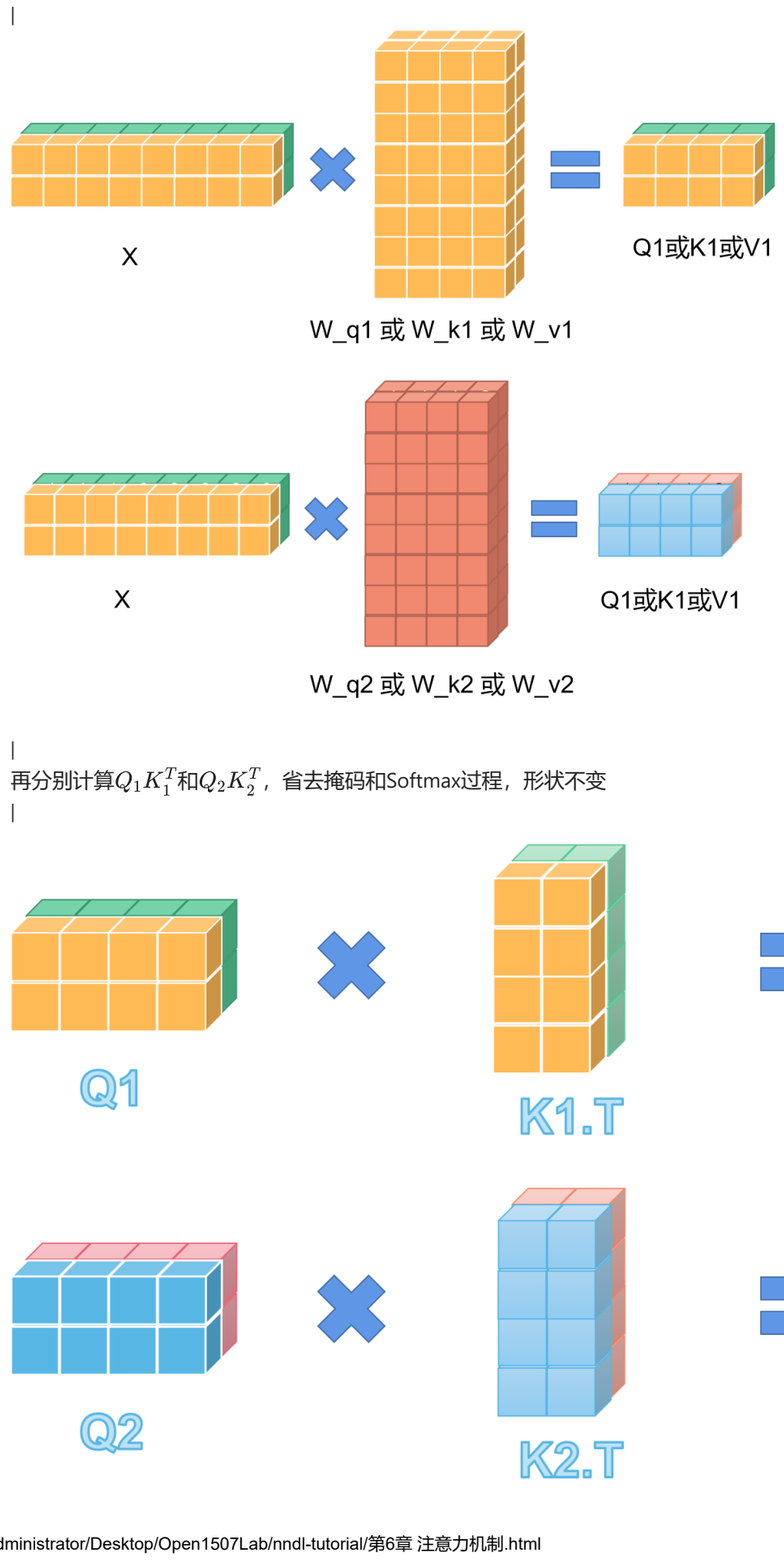
计算2头注意力的过程如下：

1.初始化2组参数：W_q1,W_k1,W_v1和W_q2,W_k2,W_v2
2.分别计算两组的QKV：Q1,K1,V1和Q2,K2,V2
3.分别计算打分，注意力分布：attention1，attention2
4.分别计算两个头：head1=attetion1*V1*，*head2=attetion2*V2
前面的过程其实就是分别计算两个不同注意力的过程，十分容易理解，接下来就是不一样的地方了
假设head1形状为BxLxD1，head2形状为BxLxD2，它们分别表示B句话，每句话L个词的编码，但是它们
的编码维数不一样，head1用D1维表示每个单词，head2用D2维表示每个单词，现在我们要做的一件事
就是把这两个头对应的每个单词拼接起来，得到BxLx(D1+D2)立方体，表示B句话，每句话L个词，每个词
用一个D1+D2维的向量表示，这么一来，通过不同立方体过来的单词编码特征就拼接融合了。
不过这里的D1和D2维数不是随便选取的，比如这里是2个头，那么把输入的维度d/2就是D，也就是每个头
的维数是d/2，最后两个头拼接起来就是d，这么一来输入维数和输出维数就一样了。
ok，把上面讲的东西归纳为第5步：
5.拼接两个头：all_head=concat(head1,head2)
最后别忘了给头加个线性层作为最后的输出
6.out=linear(all_head)

计算两个头的注意力的图形演示

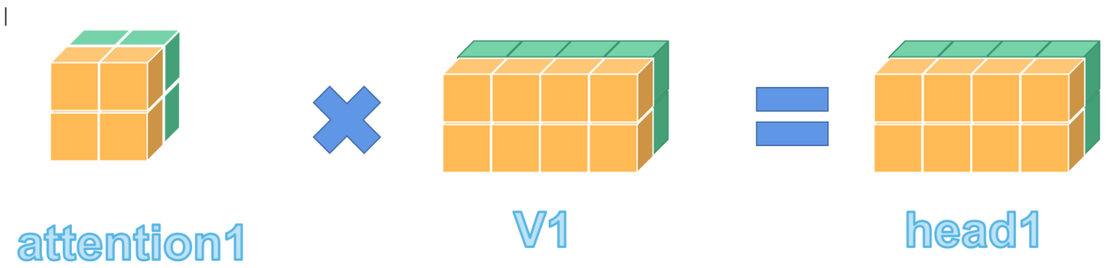
简单起见，这里省略掩码过程，只展示涉及到形状变化的关键步骤

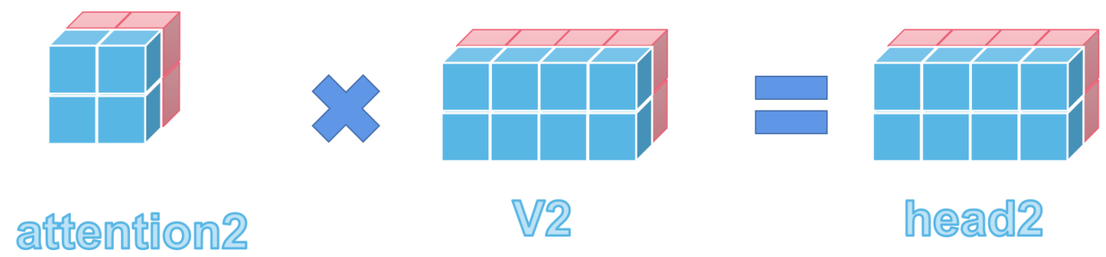
|
先计算两组QKV
|



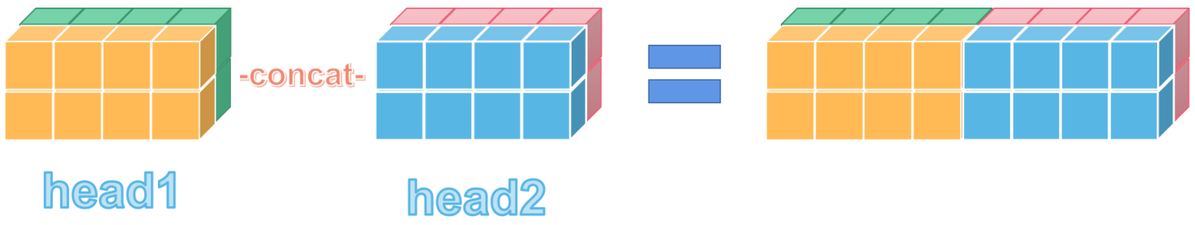
|
再分别计算$Q_1K_1^T$和$Q_2K_2^T$，省去掩码和Softmax过程，形状不变
|

|
再分别计算head1和head2
|



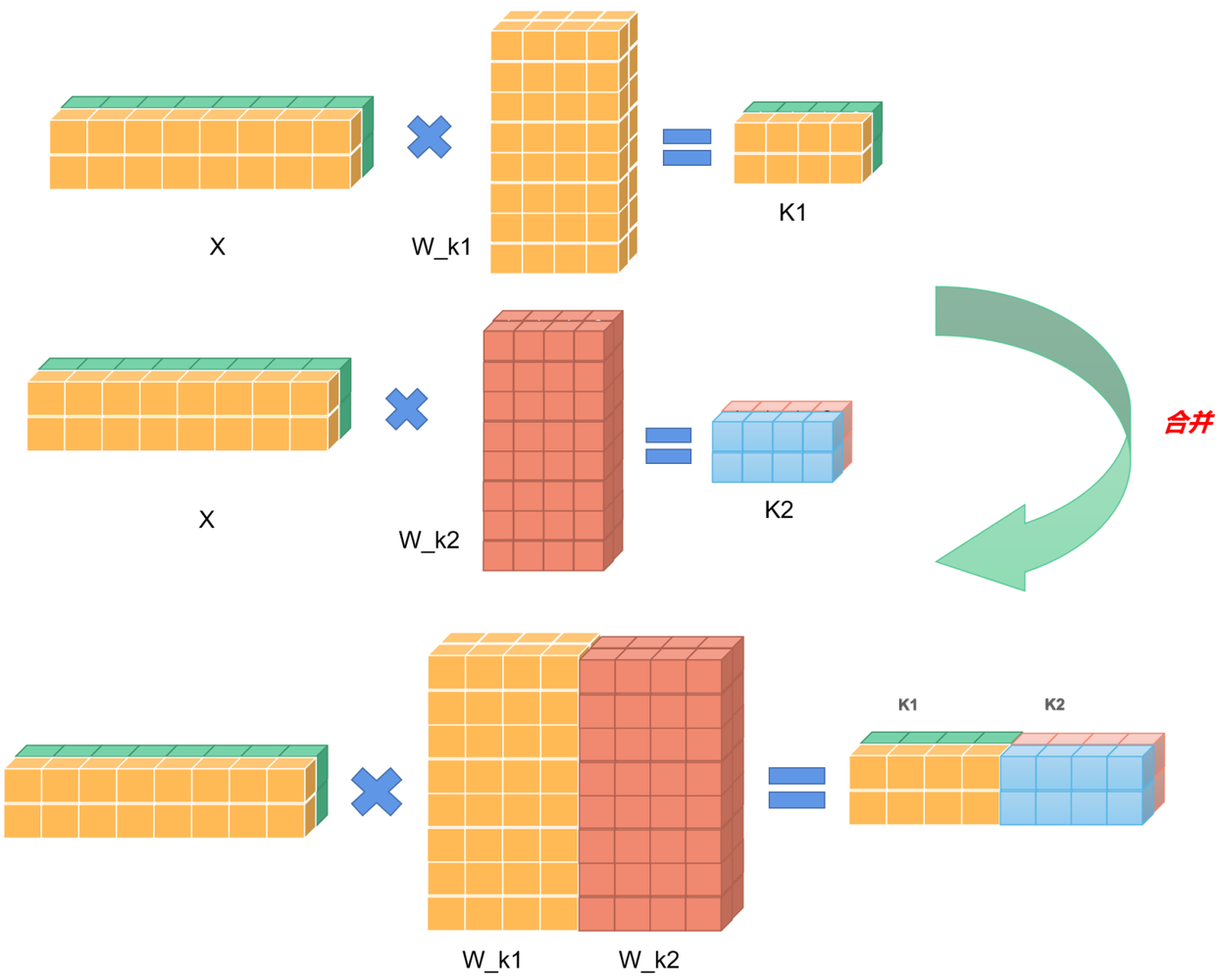


|
拼接head1和head2
|



并行化计算处理

|
Step1：计算每组Q,K,V的过程可以合并为一次性计算，以计算K为例，看下图是怎么合并的
|



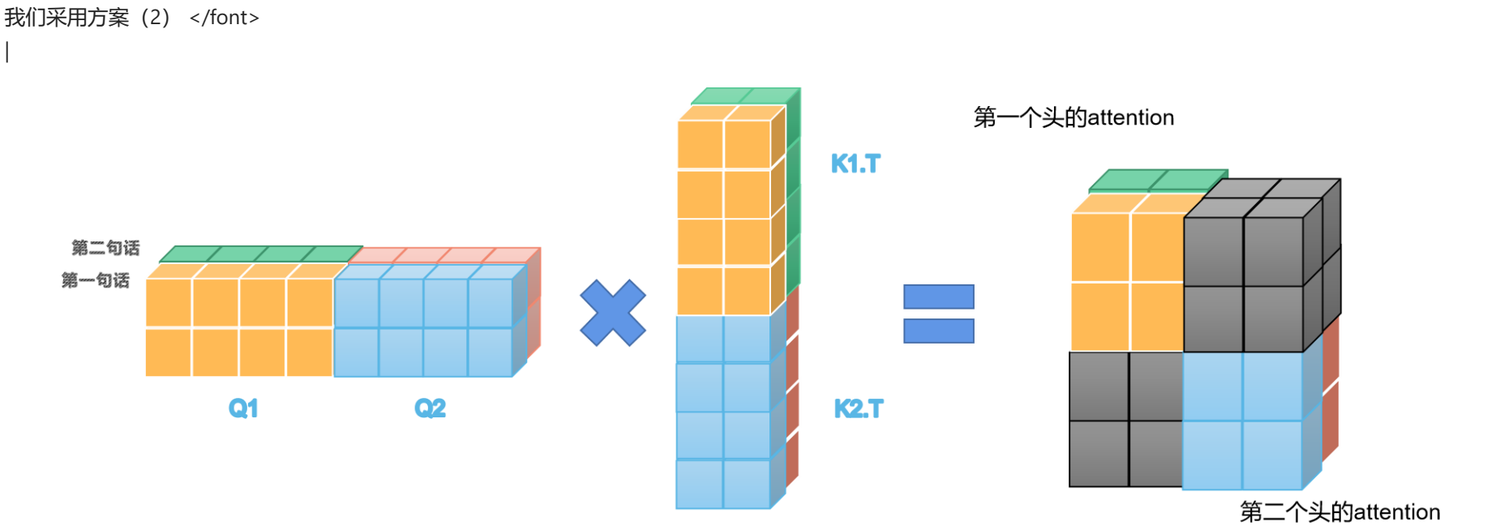
因此计算多个头的K的时候，初始化参数W_k本来需要分别初始化为多个的， 通过上图可以知道只要初始化一个超长的即可

|
Step2：注意力打分的并行处理：

接下来我们就需要考虑怎么让一个矩阵乘法同时获得$Q_1K_1^T$和$Q_2K_2^T$，下面是可考虑的一些实现方案：

（1）. 直接相乘，获得分块矩阵，取出对角上的分块就是我要的结果，但是计算复杂度太高，多计算了一堆东西
（2）. 分别取出第一句话的多个头，第二句话的多个头，前后依次排列，利用在前后方向上的多组矩阵并行计算实现
比如B=4句话，head=3，那么前后排列了12个矩阵，相当于对12句话并行计算

我们采用方案（2）
|



|

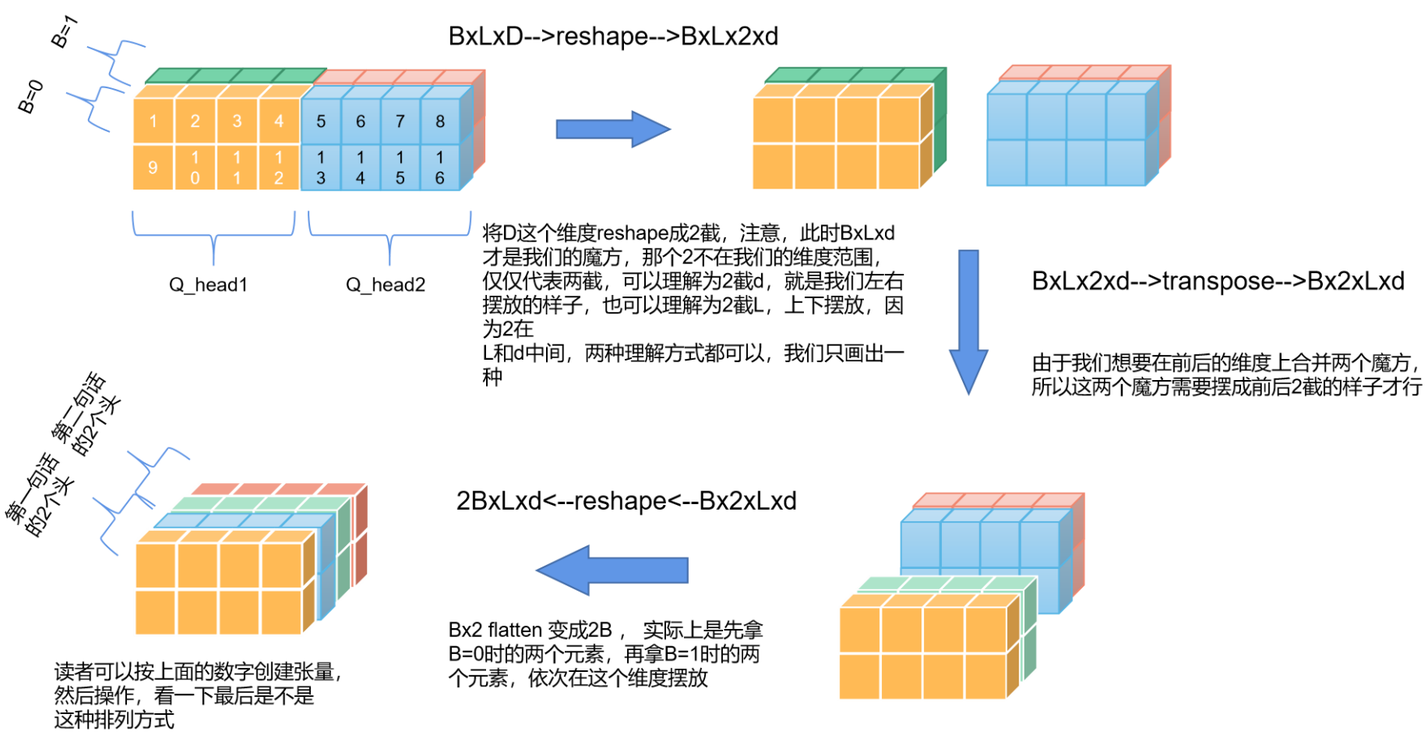
Step3：加权求和，拼接（省略掩码和softmax的过程，这两个过程不影响注意力形状）

|

|

注意力打分并行处理的编程实现： 重点就是把我们上一步得到的合体Q，合体K，合体V变成上面展示的前后排 这么一个形状，下图以Q的变形为例进行解释（K和V也是一样的）

|



In [25]:
```
x=torch.arange(1,33).reshape(2,2,8)
x
```

Out[25]:
```
tensor([[[ 1,  2,  3,  4,  5,  6,  7,  8],
         [ 9, 10, 11, 12, 13, 14, 15, 16]],

        [[17, 18, 19, 20, 21, 22, 23, 24],
         [25, 26, 27, 28, 29, 30, 31, 32]]])
```

In [26]:
```
x.reshape(2,2,2,4).transpose(1,2).reshape(4,2,4)
```

Out[26]:
```
tensor([[[ 1,  2,  3,  4],
         [ 9, 10, 11, 12]],

        [[ 5,  6,  7,  8],
         [13, 14, 15, 16]],

        [[17, 18, 19, 20],
         [25, 26, 27, 28]],

        [[21, 22, 23, 24],
         [29, 30, 31, 32]]])
```

五、多头自注意力的复现

代码参考：

http://nlp.seas.harvard.edu/2018/04/03/attention.html

https://finisky.github.io/2020/05/25/multiheadattention/

https://github.com/pytorch/pytorch/blob/main/torch/nn/functional.py

https://stackoverflow.com/questions/62629644/what-the-difference-between-att-mask-and-key-padding-mask-in-multiheadattnetion

In [27]:
```
class MultiheadAttention(nn.Module):
    def __init__(self,embed_dim,num_heads):
        super(MultiheadAttention,self).__init__()

        assert embed_dim % num_heads==0
        self.embed_dim=embed_dim
        self.num_heads=num_heads
        self.head_size=torch.tensor(embed_dim/num_heads).int()

        self.W_q=nn.Linear(embed_dim,embed_dim,bias=False)
        self.W_k=nn.Linear(embed_dim,embed_dim,bias=False)
        self.W_v=nn.Linear(embed_dim,embed_dim,bias=False)
        self.W_out=nn.Linear(embed_dim,embed_dim,bias=False)
        self.softmax=nn.Softmax(dim=-1)

    def forward(self,q,k,v,key_padding_mask=None,attn_mask=None,average_attn_weights=True):
        B,L,D=q.shape
        Q,K,V=self.W_q(q),self.W_k(k),self.W_v(v)                                                  # [B,L,D]
        Q,K,V=[self.split_head_reshape(item,self.num_heads,self.head_size) for item in [Q,K,V]]   # [B*num_heads,L,head_size]
        attn_score=torch.matmul(Q,K.transpose(1,2))/torch.sqrt(torch.tensor(self.head_size))      # [B*num_heads,L,L]

        # 分别掩码，官方代码是合并到attn_mask再掩码
        if key_padding_mask is not None:
            assert key_padding_mask.shape==torch.Size([B,L])
            key_padding_mask = key_padding_mask.reshape(B, 1, 1, L).expand([-1, self.num_heads, -1, -1]).reshape([B * self.num_heads, 1, L])
            attn_score=attn_score.masked_fill(key_padding_mask,-1e9)
        if attn_mask is not None:
            if attn_mask.dim()==2:
                assert attn_mask.shape==torch.Size([L,L])
                attn_score=attn_score.masked_fill(attn_mask,-1e9)
            elif attn_mask.dim()==3:
                assert attn_mask.shape==torch.Size([B*self.num_heads,L,L])
                attn_score=attn_score.masked_fill(attn_mask,-1e9)

        attention=self.softmax(attn_score)              # [B*num_heads,L,L]
        out=torch.matmul(attention,V)                   # [B*num_heads,L,head_size]
        out=self.head_rebuild(out,B,L,self.num_heads,self.head_size)
        out=self.W_out(out)

        attn_weights=attention.reshape([B,self.num_heads,L,L])
        if average_attn_weights:
            attn_weights=attn_weights.mean(dim=1)

        return out,attn_weights

    def split_head_reshape(self,X,num_heads,head_size):
        B,L,D=X.shape
        X=X.reshape([B,L,num_heads,head_size])
```

```
        X=X.transpose(1,2)
        X=X.reshape([B*num_heads,L,head_size])
        return X
    def head_rebuild(self,X,B,L,num_heads,head_size):
        X=X.reshape([B,num_heads,L,head_size])
        X=X.transpose(1,2)
        X=X.reshape([B,L,num_heads*head_size])
        return X
```

输入假设：

我们的输入是BxLxD=3x4x6，句子的真实长度分别为[4,3,2]

然后我们打算分成num_heads=N=2个头来计算

In [28]:
```
input_x=torch.randn([3,4,6])
seq_lens=torch.tensor([4,3,2])
```

In [29]:
```
def creat_key_padding_mask(L,seq_lens):
    seq_lens=seq_lens.unsqueeze(-1)  # Bx1
    mask=torch.arange(L)<seq_lens    # BxL
    return ~mask
```

In [30]:
```
multi_head_attention_1=nn.MultiheadAttention(embed_dim=6,num_heads=2,batch_first=True)
multi_head_attention_2=MultiheadAttention(embed_dim=6,num_heads=2)
```

In [31]:
```
out_1,attn_1=multi_head_attention_1(input_x,input_x,input_x,key_padding_mask=creat_key_padding_mask(4,seq_lens))
out_2,attn_2=multi_head_attention_2(input_x,input_x,input_x,key_padding_mask=creat_key_padding_mask(4,seq_lens))
```

```
C:\Users\Administrator\AppData\Local\Temp\ipykernel_16140\805210490.py:20: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.tensor(sourceTensor).
  attn_score=torch.matmul(Q,K.transpose(1,2))/torch.sqrt(torch.tensor(self.head_size))     # [B*num_heads,L,L]
```

In [32]:
```
print("注意力矩阵：")
print(attn_1)
print(attn_2)
```

```
注意力矩阵：
tensor([[[0.2479, 0.1741, 0.3751, 0.2029],
         [0.2193, 0.4230, 0.1476, 0.2101],
         [0.1511, 0.4550, 0.2110, 0.1830],
         [0.2817, 0.1808, 0.2378, 0.2998]],

        [[0.3857, 0.3750, 0.2392, 0.0000],
         [0.3933, 0.4271, 0.1797, 0.0000],
         [0.3328, 0.2895, 0.3777, 0.0000],
         [0.4923, 0.3538, 0.1539, 0.0000]],

        [[0.5787, 0.4213, 0.0000, 0.0000],
         [0.3919, 0.6081, 0.0000, 0.0000],
         [0.4242, 0.5758, 0.0000, 0.0000],
         [0.5342, 0.4658, 0.0000, 0.0000]]], grad_fn=<MeanBackward1>)
tensor([[[0.3454, 0.1231, 0.2477, 0.2839],
         [0.1779, 0.2290, 0.3747, 0.2185],
         [0.2698, 0.3188, 0.1916, 0.2199],
         [0.2322, 0.3266, 0.1959, 0.2453]],

        [[0.2258, 0.2972, 0.4771, 0.0000],
         [0.3383, 0.3273, 0.3344, 0.0000],
         [0.2856, 0.2579, 0.4565, 0.0000],
         [0.4272, 0.3563, 0.2164, 0.0000]],

        [[0.4538, 0.5462, 0.0000, 0.0000],
         [0.4375, 0.5625, 0.0000, 0.0000],
         [0.3620, 0.6380, 0.0000, 0.0000],
         [0.4154, 0.5846, 0.0000, 0.0000]]], grad_fn=<MeanBackward1>)
```

In [33]:
```
print("输出：")
print(out_1)
print(out_2)
```

```
输出：
tensor([[[-0.2368, -0.1400,  0.2927, -0.0476, -0.2835,  0.1501],
         [ 0.0071,  0.1493, -0.0158,  0.0378,  0.0341, -0.0022],
         [ 0.0641,  0.2962, -0.5384,  0.0922,  0.4851, -0.3735],
         [-0.2394, -0.1035,  0.1142, -0.1185, -0.2374,  0.0224]],

        [[-0.2772, -0.0514,  0.2960, -0.0411, -0.1393,  0.1720],
         [-0.2930, -0.1058,  0.3133, -0.0605, -0.1990,  0.1812],
         [-0.1530, -0.0343,  0.3695,  0.0329, -0.1102,  0.2418],
         [-0.2110, -0.2073,  0.4253, -0.0081, -0.2480,  0.2559]],

        [[-0.2653, -0.3826,  0.2606, -0.3051, -0.5485,  0.0902],
         [-0.2334, -0.3845,  0.2232, -0.3489, -0.5798,  0.0601],
         [-0.2389, -0.3840,  0.2299, -0.3409, -0.5739,  0.0652],
         [-0.2589, -0.3895,  0.2441, -0.3341, -0.5748,  0.0880]]],
       grad_fn=<TransposeBackward0>)
tensor([[[ 0.0739, -0.1913, -0.0865,  0.0426, -0.0242, -0.0651],
         [ 0.0586, -0.2609, -0.2395,  0.0826, -0.1521,  0.0088],
         [ 0.0021, -0.0936, -0.1135, -0.0105, -0.0858,  0.0302],
         [-0.0456, -0.1154, -0.1666,  0.0281, -0.0817,  0.0858]],

        [[-0.0605,  0.0991,  0.2972, -0.4862,  0.2363, -0.2258],
         [ 0.0327,  0.0762,  0.3396, -0.4736,  0.2286, -0.3287],
         [-0.0721,  0.0952,  0.2880, -0.4897,  0.2361, -0.2175],
         [ 0.0725,  0.0566,  0.3548, -0.4514,  0.2361, -0.3779]],

        [[ 0.0900,  0.1377,  0.2951,  0.0668,  0.2110, -0.1114],
         [ 0.0862,  0.1438,  0.2892,  0.0573,  0.1970, -0.1031],
         [ 0.0252,  0.1511,  0.2649,  0.0670,  0.2165, -0.0335],
         [ 0.0691,  0.1463,  0.2820,  0.0592,  0.2013, -0.0833]]],
       grad_fn=<UnsafeViewBackward0>)
```

和官方API对比：

上面那个不叫对比，因为初始化的参数不一样，下面我们强行把官方初始化的参数

和我们的参数变成一样的，然后对比结果，看看是不是一切都完全一样

In [34]:
```
all_w=[]
for param in multi_head_attention_1.parameters():
    all_w.append(param)
all_w[0].shape,all_w[2].shape
```

Out[34]:
```
(torch.Size([18, 6]), torch.Size([6, 6]))
```

In [35]:
```
multi_head_attention_2.W_q.weight.data=all_w[0][:6,:]
multi_head_attention_2.W_k.weight.data=all_w[0][6:12,:]
multi_head_attention_2.W_v.weight.data=all_w[0][12:,:]
multi_head_attention_2.W_out.weight.data=all_w[2]
```

In [36]:
```
out_3,attn_3=multi_head_attention_2(input_x,input_x,input_x,key_padding_mask=creat_key_padding_mask(4,seq_lens))
```

```
C:\Users\Administrator\AppData\Local\Temp\ipykernel_16140\805210490.py:20: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.tensor(sourceTensor).
  attn_score=torch.matmul(Q,K.transpose(1,2))/torch.sqrt(torch.tensor(self.head_size))     # [B*num_heads,L,L]
```

In [37]:
```
print("注意力矩阵：")
print(attn_1)
print(attn_3)
```

```
注意力矩阵：
tensor([[[0.2479, 0.1741, 0.3751, 0.2029],
         [0.2193, 0.4230, 0.1476, 0.2101],
         [0.1511, 0.4550, 0.2110, 0.1830],
         [0.2817, 0.1808, 0.2378, 0.2998]],

        [[0.3857, 0.3750, 0.2392, 0.0000],
         [0.3933, 0.4271, 0.1797, 0.0000],
         [0.3328, 0.2895, 0.3777, 0.0000],
         [0.4923, 0.3538, 0.1539, 0.0000]],

        [[0.5787, 0.4213, 0.0000, 0.0000],
         [0.3919, 0.6081, 0.0000, 0.0000],
         [0.4242, 0.5758, 0.0000, 0.0000],
         [0.5342, 0.4658, 0.0000, 0.0000]]], grad_fn=<MeanBackward1>)
tensor([[[0.2479, 0.1741, 0.3751, 0.2029],
         [0.2193, 0.4230, 0.1476, 0.2101],
         [0.1511, 0.4550, 0.2110, 0.1830],
         [0.2817, 0.1808, 0.2378, 0.2998]],

        [[0.3857, 0.3750, 0.2392, 0.0000],
         [0.3933, 0.4271, 0.1797, 0.0000],
         [0.3328, 0.2895, 0.3777, 0.0000],
         [0.4923, 0.3538, 0.1539, 0.0000]],

        [[0.5787, 0.4213, 0.0000, 0.0000],
         [0.3919, 0.6081, 0.0000, 0.0000],
         [0.4242, 0.5758, 0.0000, 0.0000],
         [0.5342, 0.4658, 0.0000, 0.0000]]], grad_fn=<MeanBackward1>)
```

In [38]:
```
print("输出：")
print(out_1)
print(out_3)
```

```
输出：
tensor([[[-0.2368, -0.1400,  0.2927, -0.0476, -0.2835,  0.1501],
         [ 0.0071,  0.1493, -0.0158,  0.0378,  0.0341, -0.0022],
         [ 0.0641,  0.2962, -0.5384,  0.0922,  0.4851, -0.3735],
         [-0.2394, -0.1035,  0.1142, -0.1185, -0.2374,  0.0224]],

        [[-0.2772, -0.0514,  0.2960, -0.0411, -0.1393,  0.1720],
         [-0.2930, -0.1058,  0.3133, -0.0605, -0.1990,  0.1812],
         [-0.1530, -0.0343,  0.3695,  0.0329, -0.1102,  0.2418],
         [-0.2110, -0.2073,  0.4253, -0.0081, -0.2480,  0.2559]],

        [[-0.2653, -0.3826,  0.2606, -0.3051, -0.5485,  0.0902],
         [-0.2334, -0.3845,  0.2232, -0.3489, -0.5798,  0.0601],
         [-0.2389, -0.3840,  0.2299, -0.3409, -0.5739,  0.0652],
         [-0.2589, -0.3895,  0.2441, -0.3341, -0.5748,  0.0880]]],
       grad_fn=<TransposeBackward0>)
tensor([[[-0.2368, -0.1400,  0.2927, -0.0476, -0.2835,  0.1501],
         [ 0.0071,  0.1493, -0.0158,  0.0378,  0.0341, -0.0022],
         [ 0.0641,  0.2962, -0.5384,  0.0922,  0.4851, -0.3735],
         [-0.2394, -0.1035,  0.1142, -0.1185, -0.2374,  0.0224]],

        [[-0.2772, -0.0514,  0.2960, -0.0411, -0.1393,  0.1720],
         [-0.2930, -0.1058,  0.3133, -0.0605, -0.1990,  0.1812],
         [-0.1530, -0.0343,  0.3695,  0.0329, -0.1102,  0.2418],
         [-0.2110, -0.2073,  0.4253, -0.0081, -0.2480,  0.2559]],

        [[-0.2653, -0.3826,  0.2606, -0.3051, -0.5485,  0.0902],
         [-0.2334, -0.3845,  0.2232, -0.3489, -0.5798,  0.0601],
         [-0.2389, -0.3840,  0.2299, -0.3409, -0.5739,  0.0652],
         [-0.2589, -0.3895,  0.2441, -0.3341, -0.5748,  0.0880]]],
       grad_fn=<UnsafeViewBackward0>)
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: