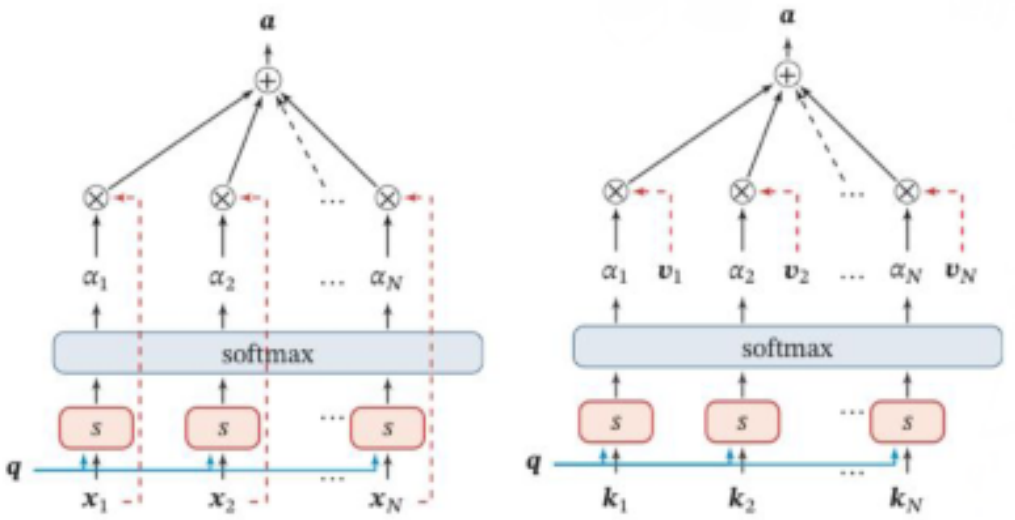


图 8.1 注意力机制

超详细图解Self-Attention

ConquerJ
伟大是熬出来的

一年之前，初次接触 `Transformer` 。当时只觉得模型复杂，步骤繁复，苦读论文多日也没有完全理解其中道理，只是泛泛地记住了一些名词，于其内部机理完全不通，相关公式更是过目便忘。

`Self-Attention` 是 `Transformer` 最核心的思想，最近几日重读论文，有了一些新的感想。由此写下本文与读者共勉。

笔者刚开始接触 `Self-Attention` 时，最大的不理解的地方就是 `Q K V` 三个矩阵以及我们常提起的Query查询向量等等，现在究其原因，应当是被高维繁复的矩阵运算难住了，没有真正理解矩阵运算的核心意义。因此，在本文开始之前，笔者首先总结一些基础知识，文中会重新提及这些知识蕴含的思想是怎样体现在模型中的。

一些基础知识

1. 向量的内积是什么，如何计算，最重要的，其几何意义是什么？
2. 一个矩阵 W 与其自身的转置相乘，得到的结果有什么意义？

1. 键值对注意力

这一节我们首先分析 `Transformer` 中最核心的部分，我们从公式开始，将每一步都绘制成图，方便读者理解。

键值对Attention最核心的公式如下图。其实这一个公式中蕴含了很多个点，我们一个一个来讲。请读者跟随我的思路，从最核心的部分入手，细枝末节的部分会豁然开朗。

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

假如上面的公式很难理解，那么下面的公式读者能否知道其意义是什么呢？

$$Softmax(XX^T)X$$

我们先抛开 `Q K V` 三个矩阵不谈，self-Attention最原始的形态其实长上面这样。那么这个公式到底是什么意思呢？

我们一步一步讲

XX^T 代表什么？

一个矩阵乘以它自己的转置，会得到什么结果，有什么意义？

我们知道，矩阵可以看作由一些向量组成，一个矩阵乘以它自己转置的运算，其实可以看成这些向量分别与其他向量计算内积。（此时脑海里想起矩阵乘法的口诀，第一行乘以第一列、第一行乘以第二列.....嗯哼，矩阵转置以后第一行不就是第一列吗？这是在计算**第一个行向量与自己**的内积，第一行乘以第二列是计算**第一个行向量与第二个行向量的内积**第一行乘以第三列是计算**第一个行向量与第三个行向量的内积**.....）

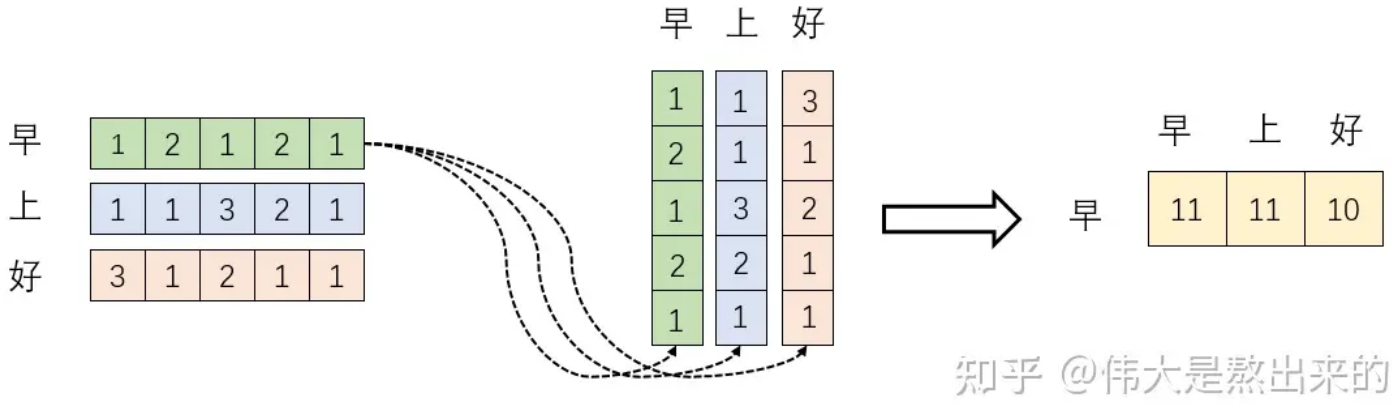
回想我们文章开头提出的问题，向量的内积，其几何意义是什么？

答：表征两个向量的夹角，表征一个向量在另一个向量上的投影

记住这个知识点，我们进入一个超级详细的实例：

我们假设 $X = [x_1^T; x_2^T; x_3^T]$ ，其中 X 为一个二维矩阵， x_i^T 为一个行向量（其实很多教材都默认向量是列向量，为了方便举例请读者理解笔者使用行向量）。对应下面的图， x_1^T 对应"早"字embedding之后的结果，以此类推。

下面的运算模拟了一个过程，即 XX^T 。我们来看看其结果究竟有什么意义



首先，行向量 x_i^T 分别与自己和其他两个行向量做内积（"早"分别与"上""好"计算内积），得到了一个新的向量。我们回想前文提到的**向量的内积表征两个向量的夹角，表征一个向量在另一个向量上的投影**。那么新的向量向量有什么意义的？是行向量 x_i^T 在自己和其他两个行向量上的投影。我们思考，投影的值大有什么意思？投影的值小又如何？

投影的值大，说明两个向量相关度高。

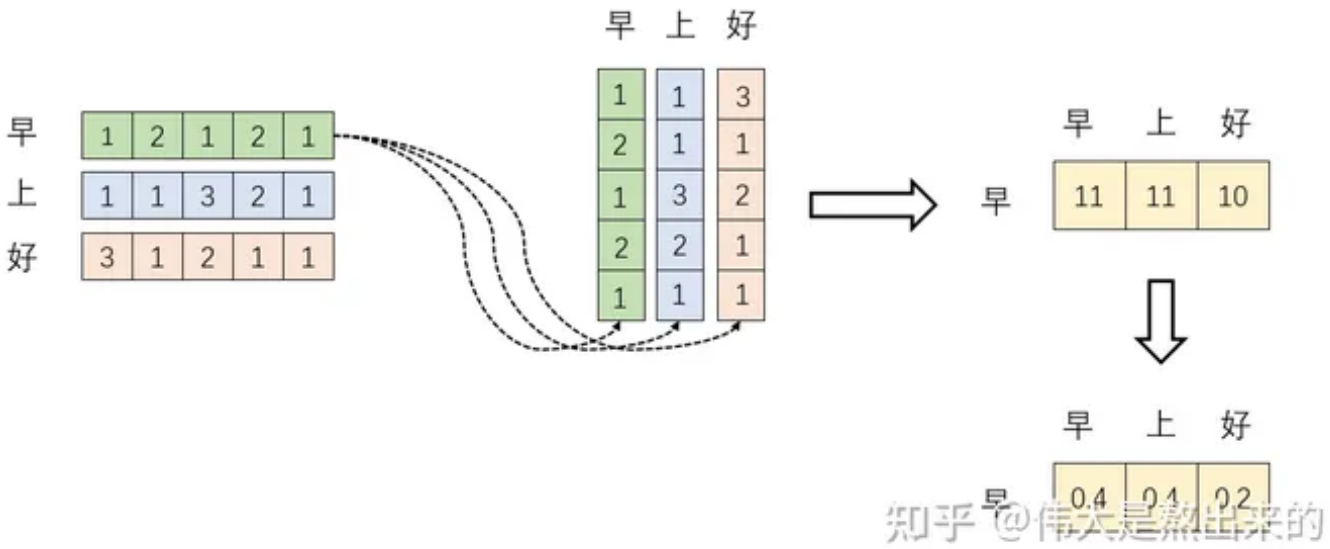
我们考虑，如果两个向量夹角是九十度，那么这两个向量线性无关，完全没有相关性！

更进一步，这个向量是词向量，是词在高维空间的数值映射。词向量之间相关度高表示什么？是不是**在一定程度上**（不是完全）表示，在关注词A的时候，应当给予词B更多的关注？

上图展示了一个行向量运算的结果，那么矩阵 XX^T 的意义是什么呢？

矩阵 XX^T 是一个方阵，我们以行向量的角度理解，里面保存了每个向量与自己和其他向量进行内积运算的结果。

至此，我们理解了公式 $Softmax(XX^T)X$ 中， XX^T 的意义。我们进一步，Softmax的意义何在呢？请看下图



我们回想Softmax的公式，Softmax操作的意义是什么呢？

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_{c=1}^{C} e^{z_c}}$$

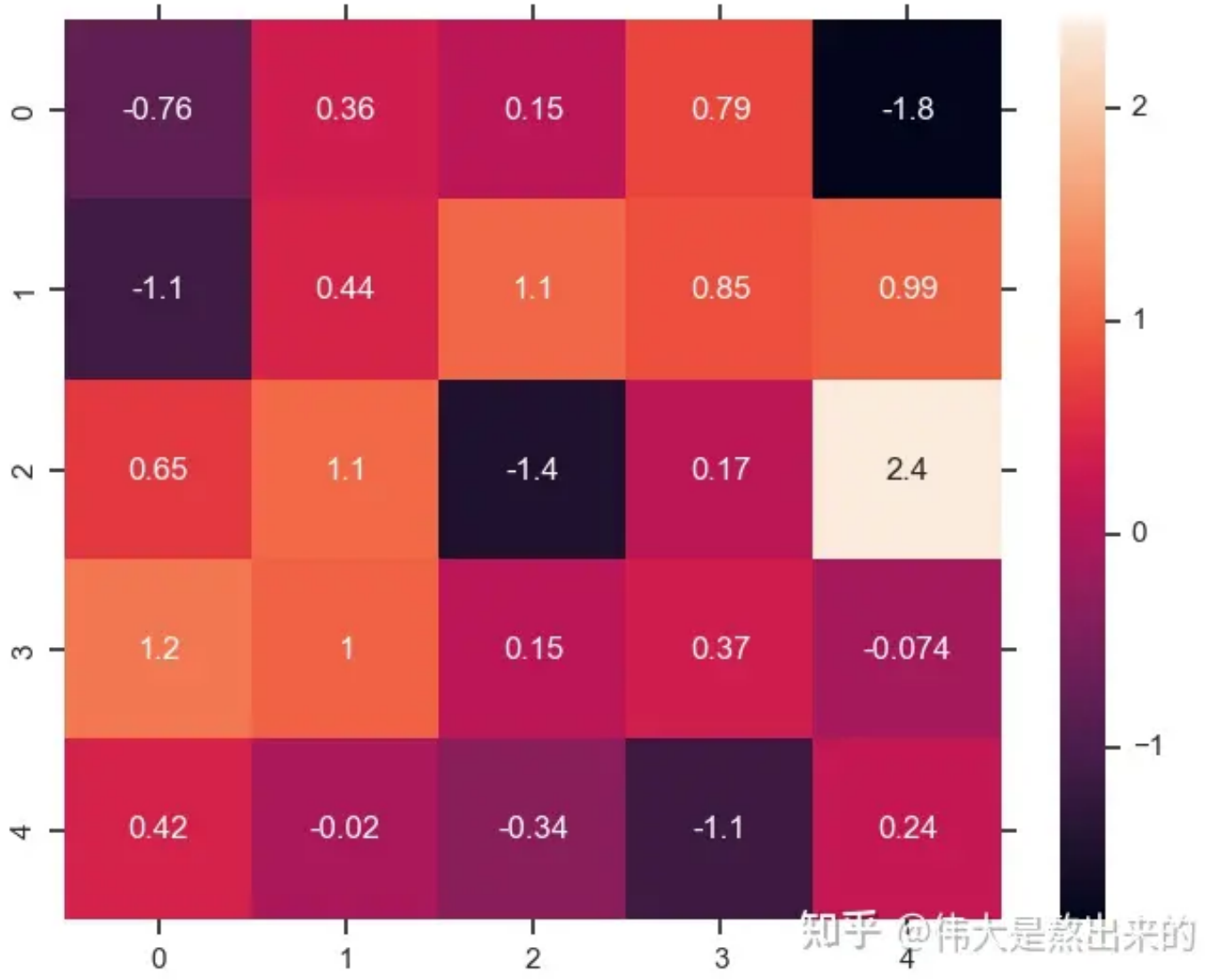
答：归一化

我们结合上面图理解，Softmax之后，这些数字的和为1了。我们再想，Attention机制的核心是什么？

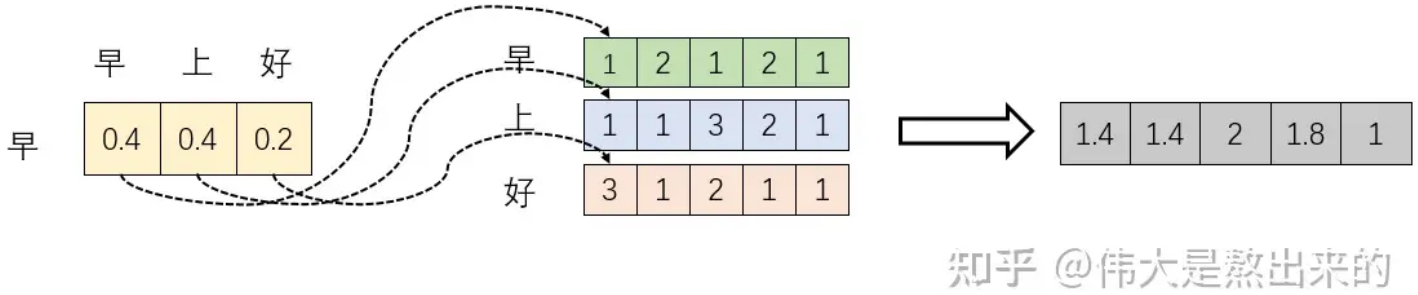
加权求和

那么权重从何而来呢？就是这些归一化之后的数字。当我们关注"早"这个字的时候，我们应当分配0.4的注意力给它本身，剩下0.4关注"上"，0.2关注"好"。当然具体到我们的Transformer，就是对应向量的运算了，这是后话。

行文至此，我们对这个东西是不是有点熟悉？Python中的热力图Heatmap，其中的矩阵是不是也保存了相似度的结果？



我们仿佛已经拨开了一些迷雾，公式 $Softmax(XX^T)X$ 已经理解了其中的一半。最后一个 X 有什么意义？完整的公式究竟表示什么？我们继续之前的计算，请看下图

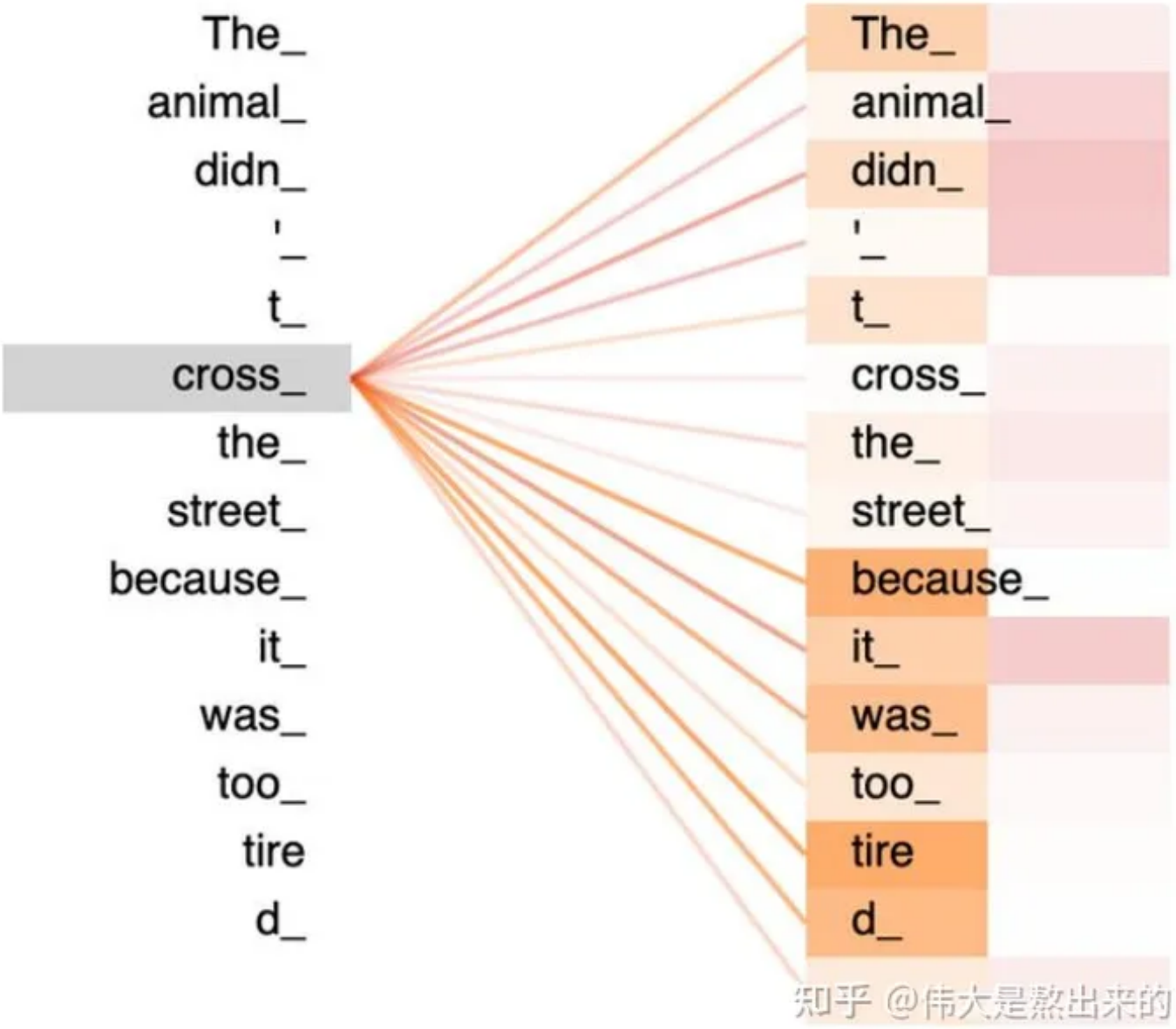


我们取 $Softmax(XX^T)$ 的一个行向量举例。这一行向量与 X 的一个列向量相乘，表示什么？

观察上图，行向量与 X 的第一个列向量相乘，得到了一个新的行向量，且这个行向量与 X 的维度相同。

在新的向量中，每一个维度的数值都是由三个词向量在这一维度的数值加权求和得来的，**这个新的行向量就是"早"字词向量经过注意力机制加权求和之后的表示。**

一张更形象的图是这样的，图中右半部分的颜色深浅，其实就是我们上图中黄色向量中数值的大小，意义就是单词之间的相关度（**回想之前的内容，相关度其本质是由向量的内积度量的**）！



如果您坚持阅读到这里，相信对公式 $Softmax(XX^T)X$ 已经有了更深刻的理解。

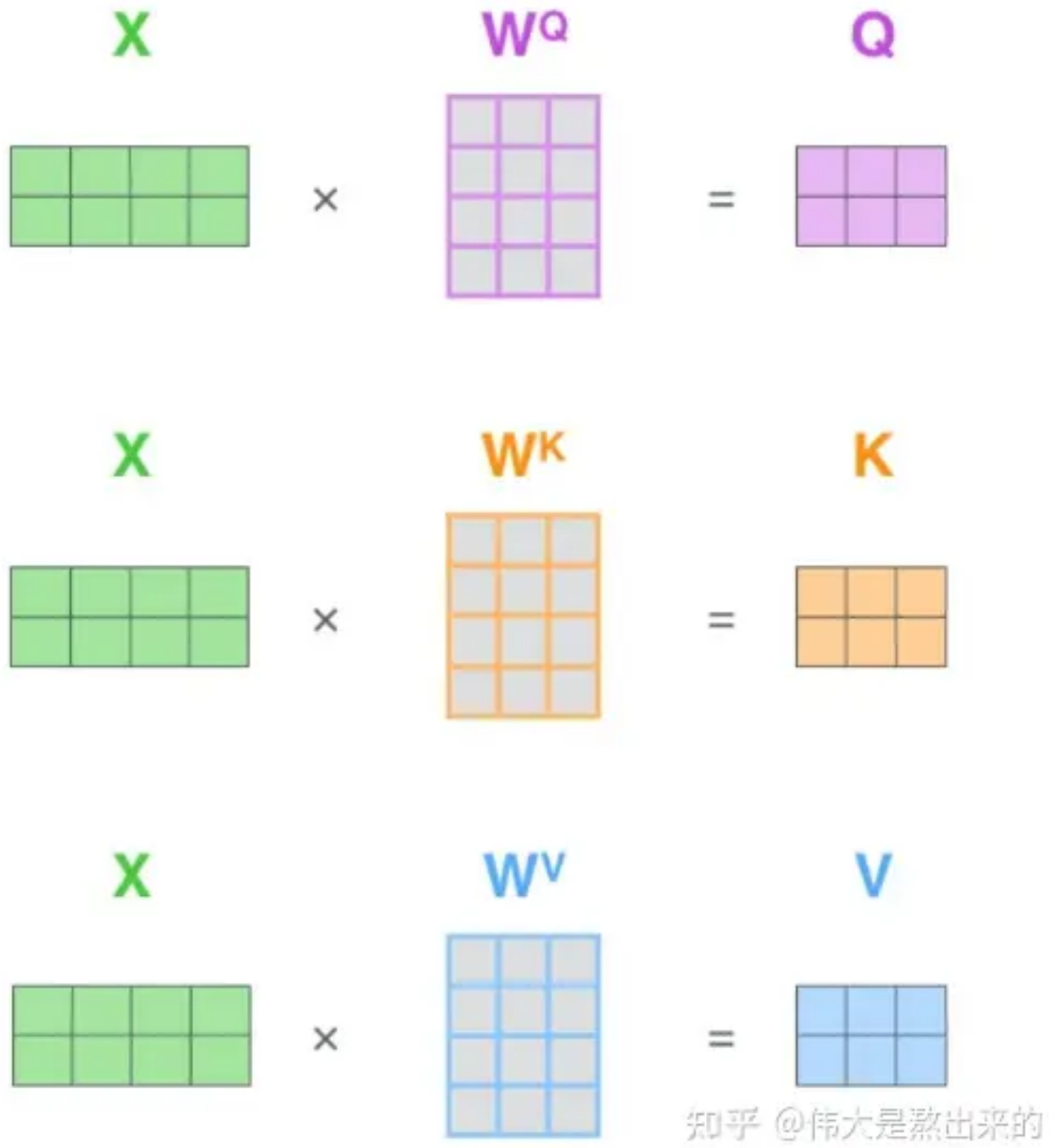
我们接下来解释原始公式中一些细枝末节的问题

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

2. Q K V 矩阵

在我们之前的例子中并没有出现 Q K V 的字眼，因为其并不是公式中最本质的内容。

Q K V 究竟是什么？我们看下面的图



其实，许多文章中所谓的 Q K V 矩阵、查询向量之类的字眼，其来源是 X 与矩阵的乘积，**本质上都是 X 的线性变换。**

为什么不直接使用 X 而要对其进行线性变换？

当然是为了提升模型的拟合能力，矩阵 W 都是可以训练的，起到一个缓冲的效果。

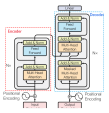
如果你真正读懂了前文的内容，读懂了 $Softmax(XX^T)$ 这个矩阵的意义，相信你也理解了所谓查询向量一类字眼的含义。

3. $\sqrt{d_k}$ 的意义

假设 Q, K 里的元素的均值为0，方差为1，那么 $A^T = Q^T K$ 中元素的均值为0，方差为d. 当d变得很大时， A 中的元素的方差也会变得很大，如果 A 中的元素方差很大，那么 $Softmax(A)$ 的分布会趋于陡峭(分布的方差大，分布集中在绝对值大的区域)。总结一下就是 $Softmax(A)$ 的分布会和d有关。因此 A 中每一个元素除以$\sqrt{d_k}$ 后，方差又变为1。这使得 $Softmax(A)$ 的分布“陡峭”程度与d解耦，从而使得训练过程中梯度值保持稳定。

至此Self-Attention中最核心的内容已经讲解完毕，关于Transformer的更多细节可以参考我的这篇回答：

Transformer - Attention is all you need
720 赞同 · 44 评论 文章



最后再补充一点，**对self-attention来说，它跟每一个input vector都做attention，所以没有考虑到input sequence的顺序。**更通俗来讲，大家可以发现我们前文的计算每一个词向量都与其他词向量计算内积，得到的结果丢失了我们原来文本的顺序信息。对比来说，LSTM是对于文本顺序信息的解释是输出词向量的先后顺序，而我们上文的计算对sequence的顺序这一部分则完全没有提及，你打乱词向量的顺序，得到的结果仍然是相同的。

这就牵扯到Transformer的位置编码了，我们按住不表。

Self-Attention的代码实现

```
# Muti-head Attention 机制的实现
from math import sqrt
import torch
import torch.nn


class Self_Attention(nn.Module):
    # input : batch_size * seq_len * input_dim
    # q : batch_size * input_dim * dim_k
    # k : batch_size * input_dim * dim_k
    # v : batch_size * input_dim * dim_v
    def __init__(self,input_dim,dim_k,dim_v):
        super(Self_Attention,self).__init__()
        self.q = nn.Linear(input_dim,dim_k)
        self.k = nn.Linear(input_dim,dim_k)
        self.v = nn.Linear(input_dim,dim_v)
        self._norm_fact = 1 / sqrt(dim_k)
```

```
    def forward(self,x):
        Q = self.q(x) # Q: batch_size * seq_len * dim_k
        K = self.k(x) # K: batch_size * seq_len * dim_k
        V = self.v(x) # V: batch_size * seq_len * dim_v

        atten = nn.Softmax(dim=-1)(torch.bmm(Q,K.permute(0,2,1))) * self._norm_fact # Q * K.T() # batch_s

        output = torch.bmm(atten,V) # Q * K.T() * V # batch_size * seq_len * dim_v

        return output
```

In [53]:
```
X = torch.randn(4, 3, 2)
print(X)
print(X.size())
```

```
tensor([[[ 3.3768, -0.4754],
         [ 0.1517, -0.0778],
         [ 1.6661,  1.5392]],

        [[-0.6465, -1.4073],
         [ 0.4545, -0.2833],
         [-1.1694, -1.3225]],

        [[-0.2970, -0.6370],
         [ 0.3744,  1.6434],
         [ 1.4549,  0.8723]],

        [[ 1.2980,  1.1185],
         [-0.4053,  1.5160],
         [ 0.7803,  0.8244]]])
torch.Size([4, 3, 2])
```

```
self_attention = Self_Attention(2,4,5)
res = self_attention(X)
print(res)
print(res.size())
```

```
tensor([[[ 0.2783, -1.0532,  0.2659,  0.6301, -0.2441],
         [ 0.4821, -0.5387, -0.0054,  0.5873, -0.0874],
         [ 0.4203, -0.6309,  0.1364,  0.6058, -0.2530]],

        [[ 0.8067,  0.2657, -0.4515,  0.5179,  0.1900],
         [ 0.7905,  0.2304, -0.4248,  0.5218,  0.1672],
         [ 0.8212,  0.2975, -0.4754,  0.5144,  0.2101]],

        [[ 0.6072, -0.3891, -0.3278,  0.5466,  0.3173],
         [ 0.6489, -0.2342, -0.3365,  0.5422,  0.2569],
         [ 0.5459, -0.5959, -0.2949,  0.5549,  0.3667]],

        [[ 0.5059, -0.8927, -0.4250,  0.5463,  0.6990],
         [ 0.5257, -0.8837, -0.4897,  0.5386,  0.7902],
         [ 0.5115, -0.8910, -0.4440,  0.5441,  0.7263]]],
       grad_fn=<BmmBackward0>)
torch.Size([4, 3, 5])
```



```
# Muti-head Attention 机制的实现
from math import sqrt
import torch
import torch.nn


class Self_Attention_Muti_Head(nn.Module):
    # input : batch_size * seq_len * input_dim
    # q : batch_size * input_dim * dim_k
    # k : batch_size * input_dim * dim_k
    # v : batch_size * input_dim * dim_v
    def __init__(self,input_dim,dim_k,dim_v,nums_head):
        super(Self_Attention_Muti_Head,self).__init__()
        assert dim_k % nums_head == 0
        assert dim_v % nums_head == 0
        self.q = nn.Linear(input_dim,dim_k)
        self.k = nn.Linear(input_dim,dim_k)
```

```
        self.v = nn.Linear(input_dim,dim_v)

        self.nums_head = nums_head
        self.dim_k = dim_k
        self.dim_v = dim_v
        self._norm_fact = 1 / sqrt(dim_k)


    def forward(self,x):
        Q = self.q(x).reshape(-1,x.shape[0],x.shape[1],self.dim_k // self.nums_head)
        K = self.k(x).reshape(-1,x.shape[0],x.shape[1],self.dim_k // self.nums_head)
        V = self.v(x).reshape(-1,x.shape[0],x.shape[1],self.dim_v // self.nums_head)
        print(x.shape)
        print(Q.size())

        atten = nn.Softmax(dim=-1)(torch.matmul(Q,K.permute(0,1,3,2))) # Q * K.T() # batch_size * seq_le

        output = torch.matmul(atten,V).reshape(x.shape[0],x.shape[1],-1) # Q * K.T() * V # batch_size *

        return output
```

在本文的基础上，笔者从零实现了Transformer模型，感兴趣的读者欢迎看一看呀~

熬了一晚上，我从零实现了Transformer模型，把代码讲给你听
2460 赞同 · 115 评论 文章

一个小广告

今年做的比较满意的一件事就是拉了一个信息浓度蛮高的交流群，主要面向 24 届秋招后端的同学（其他同学想进来瞅瞅也欢迎，群内还有多个社招大厂ssp大佬），大佬云集并且一直在保持活跃。这里不多说了，感兴趣的同学可以传送进去看看~

拿个offer知识星球 X 本群福利

24届秋招 | 高密度信息交流环境 | 高质量交流群

ConquerJ：剑指秋招 | 高质量学习交流环境?

拿个offer知识星球介绍及常见QA

docs.qq.com/doc/DWUtqcX...

简历修改 | 拿个offer知识星球

docs.qq.com/doc/DWXJ0Sl...

本群特殊福利大额优惠券直接领：

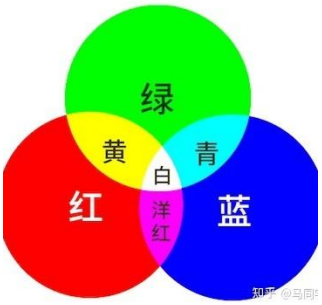
https://t.zsxq.com/0a2frxDEK

ConquerJ：24 届秋招 | 高质量学习交流环境
29 赞同 · 20 评论 文章

ConquerJ
5 次咨询
5.0
10969 次赞同
去咨询 ›
编辑于 2023-01-30 10:50 · IP 属地上海
注意力机制 自然语言处理 Transformer
▲赞同 5673 ▼ 299 条评论
分享
喜欢 收藏 申请转载
…

写下你的评论...

299 条评论
默认
最新

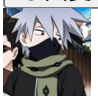


chris

漂亮的解读，早餐时间读，饭都不想吃了！

2021-09-17

回复 135



ConquerJ

作者

嘿嘿嘿，感谢！

2021-09-17

回复 14

月夜

废寝忘食了

2022-04-27

回复 3

展开其他 2 条回复 ❯

芋头
很赞啊，看着这篇，感觉其他关于self-attention的博文写的都是写啥，晦涩难懂。理解深入了很多
2021-09-30
回复 74

Gladis
▸
实践检验真理
李沐的动手学深度学习有讲，而且他还有逐段读transformer论文原文的视频，这个视频收获也挺大的
2022-04-21
回复 17

ConquerJ
作者
感谢
2021-09-30
回复 7
展开其他 3 条回复 ❯

菜菜
为什么我今天才读到这篇文章啊，以前看了多少文章都没看明白，谢谢作者，爱你一万年

2021-11-25
回复 21

北门大官人
除以根号dk这个地方应该需要用中心极限定理去证明吧？比如两个方差为1的数，相加就是方差为2了，但是相乘的话，还是1。多个这样的数 积的和，就变成了1*k了。这个地方我记得是正态分布那里的，两个分布相加相减，得来的
2022-03-28
回复 16

ljz17

第一个Self_Attention的的实现中atten = nn.Softmax(dim=-1)(torch.bmm(Q,K.permute(0,2,1))) * self._norm_fact, 这个写法是对Q * K.T()求取softmax()然后压缩了，应该是先压缩再求取softmax(),是不是应该改成atten = nn.Softmax(dim=-1)((torch.bmm(Q,K.permute(0,2,1))) * self._norm_fact)

2021-11-21

回复 15

nn.Dropout

我也发现了这个问题，感觉那个dk应该除在softmax函数里面

2021-12-07

回复 5

喵喵爱吃鱼

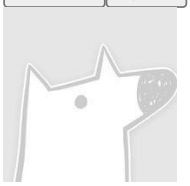
第一个scale顺序写错了，第二个scale计算写错了，而且forward里也没有scale

2022-07-05

回复 1

胡高杰

作者这边的多头注意力的代码应该有点问题，这样直接reshape虽然得到的尺寸是不变的但是，多头的含义体现错了
2021-11-13
●回复 ♥15



Sinph
写的很好，受益匪浅，就是有个问题我纠结了挺久，想请教一下：
MultiheadAttention - PyTorch 1.9.1 documentation
上面是pytorch最新版doc里面关于MULTIHEADATTENTION的源码，它里面写的QKV（似乎和博主对应的不一致，但是好像和原论文一致，姑且按他的走）的维度分别是
Q - (L,N,E)
K - (S,N,E)
V - (S,N,E)
在这个设定下Attention-score是Q×K的转制应该是A-(L,N,S),之后乘以V输出维度是（L,N,E），其中L是seqlen，E是embedding维度。
博主的最后输出是batch_size * seq_len * dimv，也就是（N,L,dimV）但是这个dimV的设定好像应该是输入的长度S吧，也就是（L,N,S）而S和E显然是两个含义。
我自己手推结果和博文写的完全一致，但是我不明白为什么我的最后结果维度和torch文档不一致，难道是torch文档写错了？纠结了好几天了
2021-10-02
●回复 ♥9

喵喵爱吃鱼

这是两种方法：第一种是直接embed_size/num_heads去拆解多头，第二种是embed_size先映射到dim_v的空间后，再dim_v/num_heads去拆解多头

2022-07-05
●回复 ♥3

赵爱梅
你应该是对的，作者写错了

2022-03-23

回复 喜欢

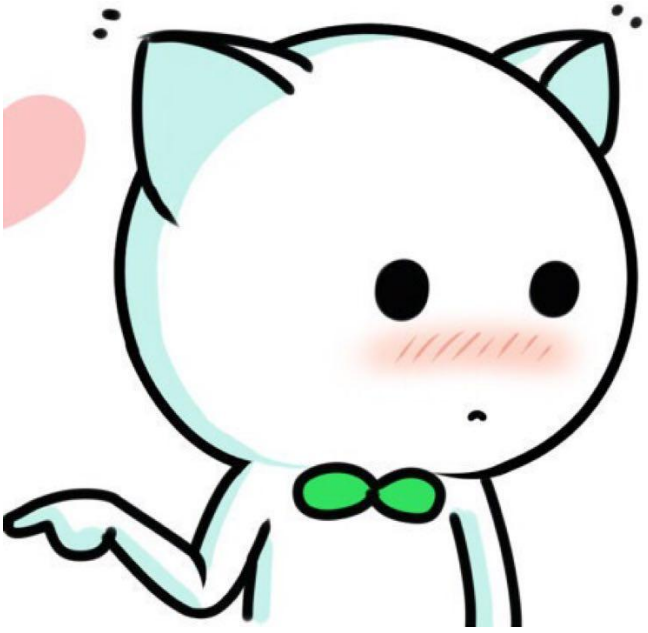
展开其他 1 条回复 ❯

努力学习的CC

有一个问题，在一堆样本上训练出的qkv，为什么能够泛化呢？

2021-11-22

回复 8



朱珂锐

因为这个“一堆”是非常大一堆 

2023-01-29

回复 7

何方圆之能周兮

深度学习的泛化性能至今只能当个黑盒去看，没有科学的解释

2023-12-04

回复 1

Happyplayer

两个向量x, y内积的定义是<x,y> = cos(x,y)|x||y|. 这里x和y的模都被学为1了，所以内积实际上就是测量x和y的夹角（或者相似度）。

2023-02-19

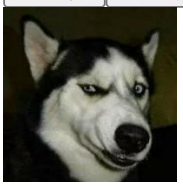
回复 4

xsyfor

必须是这个量纲一致的假设下，后边的说明才合理

2023-05-10

回复 喜欢



躺平咸鱼

读完恍然大悟，很赞很赞

2021-11-25

回复 3

点击查看全部评论

›

写下你的评论...

文章被以下专栏收录

NLP学习笔记

tensorflow人工智能