位置编码

LLaMA官方实现的旋转位置编码代码(内含详细注释及演示案例)

基本概念

为什么要使用位置编码？

嵌入层将词元(的编码)转换为一个固定大小的嵌入向量，相应地，一段文本就被转换为一个嵌入向量序列(矩阵)。例如，一段文本被分词器划分为 N 个词元，嵌入层将每个词元转换为一个 d 维的向量。那么，这段文本就被转换为一个 $N \times d$ 的矩阵(N 行 d 列)。这种更高维的表示可以捕捉词元之间的语义关系，例如同作为宠物，“猫”和“狗”可能会被嵌入层映射到相近的向量空间中。

然而，仅仅使用嵌入层并不能很好地捕捉词元在句子中的位置信息。例如，“狗在追猫”和“猫在追狗”两句话的词元构成是一样的(只有顺序差别)，对应的嵌入向量序列也只有顺序的差别，但描述的场景有很大差别。也就是说，仅仅使用嵌入层捕捉语义信息，在词元顺序的表征上有些力不从心。

为了解决这个问题，可以使用位置编码(Positional Encoding)来为每个词元添加其位置表示。

位置编码如何工作？

对于一个含有 N 个词元的文本序列：

$$S_N = \{w_i\}_{i=1}^{N}$$

其中，w_i 表示序列中的第 i 个词元。而序列 S_N 对应的嵌入向量表示为：

$$E_N = \{x_i\}_{i=1}^{N}$$

其中，x_i 表示序列中的第 i 个词元对应的 d 维嵌入向量(每个 x_i 都是一个 d 维向量)。

在自注意力机制中，我们会首先使用嵌入向量计算 q, k, v (这一点暂时不需要特别深入理解，只要知道会有这样的一步计算即可，后面会详细讲到)。就在这个过程中，我们会通过位置编码的方式加入位置信息，使计算得到的 q, k, v 是包含词元位置信息的。即：

$$q_m = f_q(x_m, m)$$

$$k_n = f_k(x_n, n)$$

$$v_n = f_v(x_n, n)$$

其中，q_m 是文本序列中第 m 个词元对应的嵌入向量 x_m 集成了位置信息 m 之后生成的 query (查询)向量；k_n 和 v_n 则分别是第 n 个词元对应的嵌入向量 x_n 集成位置信息 n 之后生成的 key (键)和 value (值)向量。

位置编码方法的关键点就是构造一个合适的函数 f，将位置信息集成到 q、k 和 v 中。

在自注意力机制中，比如要计算第 m 个词元的嵌入向量 x_m 对应的自注意力输出结果，即是对 q_m 和所有 k_n 进行点积运算，然后通过 softmax 函数归一化，得到 n 个注意力分数，最后乘以对应的 v_n 并求和：

$$a_{m,n} = \frac{\exp\left(\frac{q_m^T k_n}{\sqrt{d}}\right)}{\sum_{j=1}^{N} \exp\left(\frac{q_m^T k_j}{\sqrt{d}}\right)}, n = 1, ..., N.$$

$$o_m = \sum_{n=1}^{N} a_{m,n} v_n.$$

绝对位置编码

绝对位置编码直接为序列中的每个位置分配一个固定的编码向量，表示该位置在序列中的绝对索引。其核心特点是：每个位置都有一个独立的编码，模型依赖这些编码来区分不同位置的元素。绝对位置编码的缺点是：绝对位置编码通常难以很好地捕捉序列中元素之间的相对位置关系。

原始Transformer论文中提到的基于 Sinusoidal 函数生成位置编码的方式，是一种典型的绝对位置编码方法。绝对位置编码有如下的特点：

- 固定性：位置编码在整个训练和推理过程中都是固定的，不会被更新；
- 可扩展性：由于 Sinusoidal 函数的周期性质，这种方法可以潜在地泛化到比训练期间遇到的更长的序列长度上，这对于处理“变长”序列特别有用；
- 加法方式：通常情况下，绝对位置编码会直接加到词嵌入上，作为输入的一部分传递给模型的第1层。

简言之，Sinusoidal 位置编码是在计算 q、k 和 v 之前，预先计算一个位置编码向量 p_i（p_i 与嵌入向量 x_i 的维度一样，都是 d 维的)与嵌入向量 x_i 直接相加，再乘以相应的变换矩阵 $W_{\{q,k,v\}}$ 即得到 q、k 和 v：

$$q_m = f_q(x_m, m) = W_q(x_m + p_m)$$

$$k_n = f_k(x_n, n) = W_k(x_n + p_n)$$

$$v_n = f_v(x_n, n) = W_v(x_n + p_n)$$

而其中位置编码向量 p_i 的计算方法如下：

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

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

其中，$p_{i,2t}$ 对应于第 i 个词元的嵌入向量 x_i 的第 $2t$ 位置(偶数索引位置)的位置编码；$p_{i,2t+1}$ 对应于第 i 个词元的嵌入向量 x_i 的第 $2t+1$ 位置(奇数索引位置)的位置编码；d 是嵌入向量的维度；$\{2t, 2t+1\} \subset \{1, 2, ..., d\}$。

这种编码方式确保了对于任何固定的偏移量(后面一个词元相对于前面某个词元的位置偏移距离) k，后一个词元的位置编码 p_{i+k} 都可以表示为前面第 i 个词元的位置编码 p_i 的线性函数。

旋转位置编码

与绝对位置编码相对的是相对位置编码。从上面的绝对位置编码的介绍中，我们可以看出，绝对位置编码的缺点是难以捕捉序列中元素之间的相对位置关系。而相对位置编码关注的是序列中元素之间的相对距离，而不是它们的绝对位置。这种方法更符合人类语言理解的方式，因为在自然语言中，相对位置往往比绝对位置更重要。例如，“狗在追猫”和“猫被狗追”，重要的不是单词的绝对位置，而是它们之间的相对顺序。

旋转位置编码(Rotary Position Embedding, RoPE)是当前最常见的相对位置编码方法。它通过将位置信息整合到自注意力机制中，克服了传统绝对位置编码的一些缺点，在多个方面提升了Transformer模型的性能表现。

前面已经了解到，在自注意力机制中，我们会用到 q_m 和 k_n 的内积运算(参见 $a_{m,n}$ 的计算公式)。为了能够利用词元之间的相对位置信息，我们首先假设 q_m 和 k_n 的内积运算可以用一个函

数 g 来表示。而该函数 g 的输入是嵌入向量 x_m 、x_n 和它们之间相对位置 $m-n$ ，而与它们的绝对位置 m 和 n 无直接关系。即：

$$q_m^T k_n = \langle f_q(x_m, m), f_k(x_n, n) \rangle = g(x_m, x_n, m-n)$$

剩下的问题就是：找到一种位置编码方式(即 f 和 g 的函数形式)，使得上式成立(记作**问题(1)**)。

为了简化问题但不失一般性，我们暂且假定嵌入向量的维度 $d=2$ 。基于二维平面上的向量几何性质，可以很容易找到满足上述关系的 f 和 g ，形式如下：

$$f_q(x_m, m) = (W_q x_m) e^{im\theta}$$

$$f_k(x_n, n) = (W_k x_n) e^{in\theta}$$

$$g(x_m, x_n, m-n) = \mathrm{Re}\left[(W_q x_m)(W_k x_n)^* e^{i(m-n)\theta}\right]$$

其中，Re 表示取复数的实部；$(W_k x_n)^*$ 表示复数 $W_k x_n$ 的共轭($a+ib$ 的共轭为 $a-ib$)。

证明过程

已知欧拉公式：

$$e^{ix} = \cos x + i \sin x$$

其中，x 表示任意实数，e 是自然对数的底数，i 是复数中的虚数单位。也就是说，指数函数可以表示为实部为 $\cos x$ 、虚部为 $\sin x$ 的复数。欧拉函数建立了指数函数、三角函数和复数之间的关系。根据欧拉公式，可知：

$$e^{im\theta} = \cos(m\theta) + i \sin(m\theta)$$

$$e^{in\theta} = \cos(n\theta) + i \sin(n\theta)$$

$$e^{i(m-n)\theta} = \cos[(m-n)\theta] + i \sin[(m-n)\theta]$$

对于前面的公式 $q_m = f_q(x_m, m) = (W_q x_m) e^{im\theta}$ ，已知 W_q 是一个 2×2 的矩阵，x_m 是一个2维向量。那么 W_q 与 x_m 相乘的结果(记作 q_m^*)也是一个2维向量，即：

$$q_m^* = \begin{pmatrix} q_m^{*(1)} \\ q_m^{*(2)} \end{pmatrix} = W_q x_m = \begin{pmatrix} W_q^{(11)} & W_q^{(12)} \\ W_q^{(21)} & W_q^{(22)} \end{pmatrix} \begin{pmatrix} x_m^{(1)} \\ x_m^{(2)} \end{pmatrix}$$

而 q_m^* 作为一个2维向量，还可以表示成复数形式：

$$q_m^* = \begin{pmatrix} q_m^{*(1)} & q_m^{*(2)} \end{pmatrix} = q_m^{*(1)} + iq_m^{*(2)}$$

因此可以得到：

$$\begin{aligned} q_m &= f_q(x_m, m) \\ &= (W_q x_m) e^{im\theta} \\ &= q_m^* e^{im\theta} \\ &= \left(q_m^{*(1)} + iq_m^{*(2)}\right) \times (\cos(m\theta) + i\sin(m\theta)) \\ &= \left(q_m^{*(1)}\cos(m\theta) - q_m^{*(2)}\sin(m\theta)\right) + i\left(q_m^{*(2)}\cos(m\theta) + q_m^{*(1)}\sin(m\theta)\right) \\ &= \left(q_m^{*(1)}\cos(m\theta) - q_m^{*(2)}\sin(m\theta), \quad q_m^{*(2)}\cos(m\theta) + q_m^{*(1)}\sin(m\theta)\right) \\ &= \begin{pmatrix} \cos(m\theta) & -\sin(m\theta) \\ \sin(m\theta) & \cos(m\theta) \end{pmatrix} \begin{pmatrix} q_m^{*(1)} \\ q_m^{*(2)} \end{pmatrix}. \end{aligned}$$

由此可见，q_m 实际上 query 向量乘以一个旋转矩阵。这也是为什么这种方法被称为“旋转位置编码”的原因。

同理，可以得到 k_n 的表达式：

$$k_n = f_k(x_n, n) = \begin{pmatrix} \cos(n\theta) & -\sin(n\theta) \\ \sin(n\theta) & \cos(n\theta) \end{pmatrix} \begin{pmatrix} k_n^{*(1)} \\ k_n^{*(2)} \end{pmatrix}$$

最后，对 $g(x_m, x_n, m-n)$ 已知：

$$g(x_m, x_n, m-n) = \mathrm{Re}\left[(W_q x_m)(W_k x_n)^* e^{i(m-n)\theta}\right]$$

其中，Re 表示取复数的实部；$(W_k x_n)^*$ 表示复数 $W_k x_n$ 的共轭。由此可得到 $g(x_m, x_n, m-n)$ 的各个部分：

$$W_q x_m = q_m^* = q_m^{*(1)} + iq_m^{*(2)}$$

$$W_k x_n = k_n^* = k_n^{*(1)} + ik_n^{*(2)}$$

$$e^{i(m-n)\theta} = \cos((m-n)\theta) + i\sin((m-n)\theta)$$

将上述三个部分代入 $g(x_m, x_n, m-n)$，可以得到：

$$
\begin{aligned}
g(x_m, x_n, m-n) &= \mathrm{Re}\Big[(W_q x_m)(W_k x_n)^* e^{i(m-n)\theta}\Big] \\
&= \mathrm{Re}\Big\{\left(q_m^{*(1)} + i q_m^{*(2)}\right)\left(k_n^{*(1)} - i k_n^{*(2)}\right)[\cos((m-n)\theta) + i\sin((m-n)\theta)]\Big\} \\
&= \mathrm{Re}\Big\{\Big[\left(q_m^{*(1)} k_n^{*(1)} + q_m^{*(2)} k_n^{*(2)}\right) + i\left(q_m^{*(2)} k_n^{*(1)} - q_m^{*(1)} k_n^{*(2)}\right)\Big][\cos((m-n)\theta) + i\sin((m-n)\theta)]\Big\} \\
&= \left(q_m^{*(1)} k_n^{*(1)} + q_m^{*(2)} k_n^{*(2)}\right)\cos((m-n)\theta) - \left(q_m^{*(2)} k_n^{*(1)} - q_m^{*(1)} k_n^{*(2)}\right)\sin((m-n)\theta)
\end{aligned}
$$

而将之前推导得到的 q_m 和 k_n 的表达式代入 $q_m^T k_n = \langle f_q(x_m, m), f_k(x_n, n)\rangle$ ，可以得到：

$$
\begin{aligned}
q_m^T k_n &= \langle f_q(x_m, m), f_k(x_n, n)\rangle \\
&= \left(q_m^{*(1)} k_n^{*(1)} + q_m^{*(2)} k_n^{*(2)}\right)\cos((m-n)\theta) - \left(q_m^{*(2)} k_n^{*(1)} - q_m^{*(1)} k_n^{*(2)}\right)\sin((m-n)\theta) \\
&= g(x_m, x_n, m-n)
\end{aligned}
$$

显然，在二维平面找到的 f 和 g：

$$
f_q(x_m, m) = (W_q x_m) e^{im\theta}
$$

$$
f_k(x_n, n) = (W_k x_n) e^{in\theta}
$$

$$
g(x_m, x_n, m-n) = \mathrm{Re}\left[(W_q x_m)(W_k x_n)^* e^{i(m-n)\theta}\right]
$$

解决了**问题(1)**。

且可以得到 $q_m^T k_n = \langle f_q(x_m, m), f_k(x_n, n)\rangle$ 的矩阵向量乘积表达式为：

$$
\begin{aligned}
q_m^T k_n &= g(x_m, x_n, m-n) \\
&= \begin{pmatrix} q_m^{*(1)} & q_m^{*(2)} \end{pmatrix} \begin{pmatrix} \cos((m-n)\theta) & -\sin((m-n)\theta) \\ \sin((m-n)\theta) & \cos((m-n)\theta) \end{pmatrix} \begin{pmatrix} k_n^{*(1)} \\ k_n^{*(2)} \end{pmatrix}
\end{aligned}
$$

拓展到 $d > 2$

上面的证明和推导过程是基于 $d = 2$ 的假设，对于 $d > 2$ 的情况，则需要将词嵌入向量的元素按两两一组进行分组，每组分别应用上述的二维旋转编码操作方法。由此可得到一般化的 f ：

$$
\begin{aligned}
f_q(\boldsymbol{x}_m, m) &= \boldsymbol{R}_{\Theta,m}^d \boldsymbol{W}_q \boldsymbol{x}_m \\
f_k(\boldsymbol{x}_n, n) &= \boldsymbol{R}_{\Theta,n}^d \boldsymbol{W}_k \boldsymbol{x}_n
\end{aligned}
$$

其中，

$$\boldsymbol{R}_{\Theta,m}^{d}=\begin{pmatrix}\cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0\\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0\\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0\\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0\\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots\\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2}\\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2}\end{pmatrix}$$

$$\boldsymbol{R}_{\Theta,n}^{d}=\begin{pmatrix}\cos n\theta_1 & -\sin n\theta_1 & 0 & 0 & \cdots & 0 & 0\\ \sin n\theta_1 & \cos n\theta_1 & 0 & 0 & \cdots & 0 & 0\\ 0 & 0 & \cos n\theta_2 & -\sin n\theta_2 & \cdots & 0 & 0\\ 0 & 0 & \sin n\theta_2 & \cos n\theta_2 & \cdots & 0 & 0\\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots\\ 0 & 0 & 0 & 0 & \cdots & \cos n\theta_{d/2} & -\sin n\theta_{d/2}\\ 0 & 0 & 0 & 0 & \cdots & \sin n\theta_{d/2} & \cos n\theta_{d/2}\end{pmatrix}$$

其中，$\Theta=\{\theta_t=10000^{-2(t-1)/d},\quad i\in\{1,2,...,d\}\}$ 是预设的旋转角度。

在自注意力机制中，我们近一步有：

$$\boldsymbol{q}_m^{\boldsymbol{T}}\boldsymbol{k}_n=\left(\boldsymbol{R}_{\Theta,m}^{d}\boldsymbol{W}_q\boldsymbol{x}_m\right)^{\boldsymbol{T}}\left(\boldsymbol{R}_{\Theta,n}^{d}\boldsymbol{W}_k\boldsymbol{x}_n\right)=\boldsymbol{x}^{\boldsymbol{T}}\boldsymbol{W}_q\boldsymbol{R}_{\Theta,n-m}^{d}\boldsymbol{W}_k\boldsymbol{x}$$

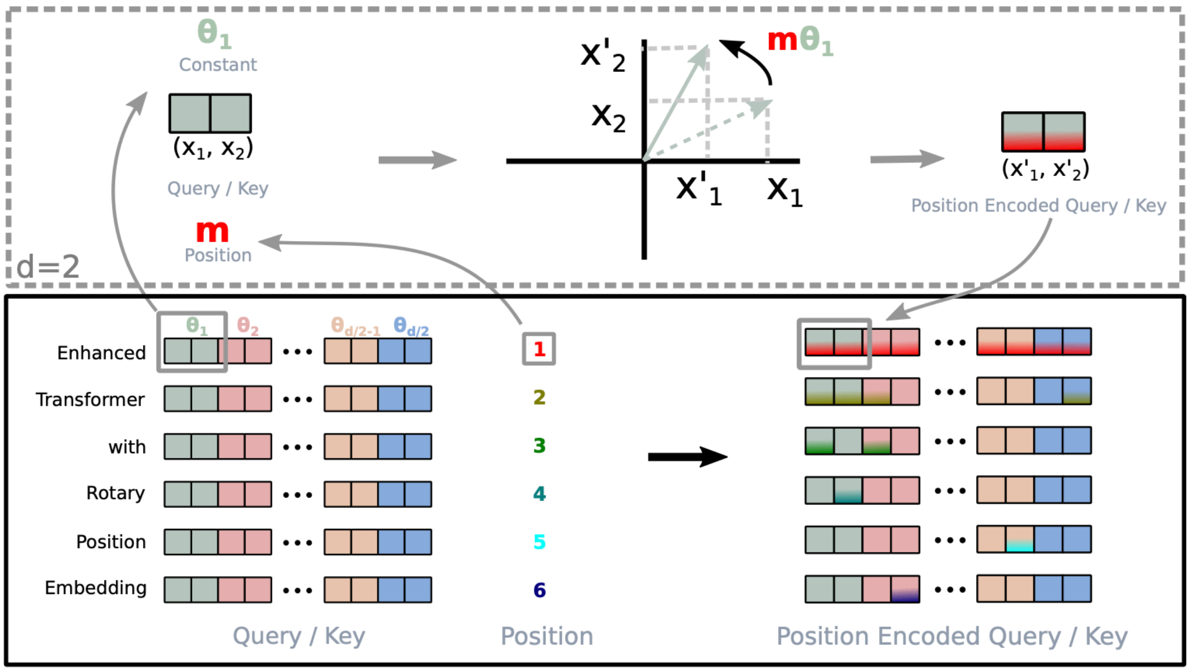
其中，

$$\boldsymbol{R}_{\Theta,n-m}^{d}=\left(\boldsymbol{R}_{\Theta,m}^{d}\right)^{\boldsymbol{T}}\boldsymbol{R}_{\Theta,n}^{d}.$$

实现方法

综上所述，旋转位置编码在自注意力机制中的具体实现流程如下：

- 对文本序列中的每个词元对应的嵌入向量计算其对应的 query 和 key 向量；
- 对每个词元所在位置都计算对应的旋转位置编码；
- 对 query 和 key 向量的元素按照两两一组应用旋转变换；
- 计算 query 和 key 向量的内积，得到注意力分数。



具体案例

假设每次处理的文本为2条 ’即批量大小=2 ’处理的最大文本块长度=5 ’模型的嵌入维度=16 ’多头自注意力机制中头的个数(即Query的头的个数)=2 ’多头自注意力机制中的分组数量(Query被分为query_group_num个组，每组对应1个Key和Value)=2 ’旋转位置编码中使用的theta值=10000.0 。

(1) 假设我们有如下的 query 张量 xq 和 key 张量 xk:

其中 ’xq是一个形状为(2, 5, 2, 8)的张量 ’代表2个样本(批量大小) ’每个样本有5个token(文本块长度) ’每个token有2个头 ’每个头的嵌入维度为8.

```
xq = tensor([[[[  0.,   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.,  33.,  34.,  35.,  36.,  37.,  38.,  39.],                  |-- 1个样本(有5个token)
               [ 40.,  41.,  42.,  43.,  44.,  45.,  46.,  47.]],                 /
              [[ 48.,  49.,  50.,  51.,  52.,  53.,  54.,  55.],                 /
               [ 56.,  57.,  58.,  59.,  60.,  61.,  62.,  63.]],               /
              [[ 64.,  65.,  66.,  67.,  68.,  69.,  70.,  71.],               /
               [ 72.,  73.,  74.,  75.,  76.,  77.,  78.,  79.]]],            /
             [[[ 80.,  81.,  82.,  83.,  84.,  85.,  86.,  87.],    1 \ 1个token   \
               [ 88.,  89.,  90.,  91.,  92.,  93.,  94.,  95.]],   1 / 有2个头      \
              [[ 96.,  97.,  98.,  99., 100., 101., 102., 103.],    2                \
               [104., 105., 106., 107., 108., 109., 110., 111.]],   2                 \
              [[112., 113., 114., 115., 116., 117., 118., 119.],    3 -1个头, 有8个维度  |-- 1个样本(有5个token)
               [120., 121., 122., 123., 124., 125., 126., 127.]],   3                 /
              [[128., 129., 130., 131., 132., 133., 134., 135.],    4                /
               [136., 137., 138., 139., 140., 141., 142., 143.]],   4               /
              [[144., 145., 146., 147., 148., 149., 150., 151.],    5              /
               [152., 153., 154., 155., 156., 157., 158., 159.]]]]) 5             /
```

xk是一个形状为(2, 5, 1, 8)的张量，代表2个样本(批量大小)，每个样本有5个token(文本块长度)，每个token有1个头，每个头的嵌入维度为8.

```
xk = tensor([[[[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.]],  -- 1个token有1个头 \
              [[ 8.,  9., 10., 11., 12., 13., 14., 15.]],  -- 1个头有8个维度   \
              [[16., 17., 18., 19., 20., 21., 22., 23.]],                      \-- 2个样本
              [[16., 17., 18., 19., 20., 21., 22., 23.]],                      /-- 有5个token
              [[24., 25., 26., 27., 28., 29., 30., 31.]],                     /
              [[32., 33., 34., 35., 36., 37., 38., 39.]]],                   /
             [[[40., 41., 42., 43., 44., 45., 46., 47.]],
              [[48., 49., 50., 51., 52., 53., 54., 55.]],
              [[56., 57., 58., 59., 60., 61., 62., 63.]],
              [[64., 65., 66., 67., 68., 69., 70., 71.]],
              [[72., 73., 74., 75., 76., 77., 78., 79.]]]])
```

(2) 预先计算位置编码的复数形式笛卡尔坐标

- 计算旋转角度 freqs：
  其中的第 i 个元素的计算方法为: $1/(10000.0^{(2(i-1)/8)})$。i 的取值从 1 到 4.
  得到 freqs = tensor([1.0000, 0.1000, 0.0100, 0.0010]).
- 创建一个长度为 5 (也可以是长度大于 5 的，但在后续使用中只截取前面 5 个)的张量 t, 其中的每个元素为其所在位置序号, 即从0到4.
  得到的 t = tensor([0, 1, 2, 3, 4]).
- 计算位置序号 t 与旋转角度 freqs 的乘积(外积, 其中的元素是t中的每个元素与freqs中的每个元素相乘的结果)

```
freqs = tensor([[0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00],
                [1.0000e+00, 1.0000e-01, 1.0000e-02, 1.0000e-03],
                [2.0000e+00, 2.0000e-01, 2.0000e-02, 2.0000e-03],
                [3.0000e+00, 3.0000e-01, 3.0000e-02, 3.0000e-03],
                [4.0000e+00, 4.0000e-01, 4.0000e-02, 4.0000e-03]])
```

- 将上一步计算的极坐标 freqs 转换为笛卡尔坐标(用复数表示，实部为cos(freq)，虚部为sin(freq))

```
freqs_cis = tensor([[ 1.0000+0.0000j,  1.0000+0.0000j,  1.0000+0.0000j,  1.0000+0.0000j],
                    [ 0.5403+0.8415j,  0.9950+0.0998j,  0.9999+0.0100j,  1.0000+0.0010j],
                    [-0.4161+0.9093j,  0.9801+0.1987j,  0.9998+0.0200j,  1.0000+0.0020j],
                    [-0.9900+0.1411j,  0.9553+0.2955j,  0.9996+0.0300j,  1.0000+0.0030j],
                    [-0.6536-0.7568j,  0.9211+0.3894j,  0.9992+0.0400j,  1.0000+0.0040j]])
```

(3) 用预先计算的角度 freqs_cis 来对 xq 和 xk 进行旋转位置嵌入 (因为这里提供的 freqs_cis 与 xq 和 xk 长度一样，所以不需要进行裁剪，否则应该裁剪为与 xq 和 xk 长度一样)

- 将 xq 和 xk 转换为复数形式，以 xq 的转换为例
  - a. 将 xq 的形状(2, 5, 2, 8)除了最后一个维度展开成一系列参数, 即得到 2, 5, 2
  - b. 将 xq 的形状从(2, 5, 2, 8)变为(2, 5, 2, 4, 2), 其中-1表示自动计算该维度的大小, 即4. 也即是说, xq 的每个头的嵌入维度 8 被拆分为 4 组, 每组 2 个维度.
  - c. 将实数张量转换为复数张量, 如果 xq 最后一个维度的大小是2, 如[a, b] 则对应输出 a + bi.
  - d. 最终的输出结果形状是(2, 5, 2, 4).

```
xq_ = tensor([[[[  0.+1.j,    2.+3.j,    4.+5.j,    6.+7.j],
                [  8.+9.j,   10.+11.j,  12.+13.j,  14.+15.j]],
               [[ 16.+17.j,  18.+19.j,  20.+21.j,  22.+23.j],
                [ 24.+25.j,  26.+27.j,  28.+29.j,  30.+31.j]],
               [[ 32.+33.j,  34.+35.j,  36.+37.j,  38.+39.j],
                [ 40.+41.j,  42.+43.j,  44.+45.j,  46.+47.j]],
               [[ 48.+49.j,  50.+51.j,  52.+53.j,  54.+55.j],
                [ 56.+57.j,  58.+59.j,  60.+61.j,  62.+63.j]],
               [[ 64.+65.j,  66.+67.j,  68.+69.j,  70.+71.j],
                [ 72.+73.j,  74.+75.j,  76.+77.j,  78.+79.j]]],
              [[[ 80.+81.j,  82.+83.j,  84.+85.j,  86.+87.j],
                [ 88.+89.j,  90.+91.j,  92.+93.j,  94.+95.j]],
               [[ 96.+97.j,  98.+99.j,  100.+101.j, 102.+103.j],
                [104.+105.j, 106.+107.j, 108.+109.j, 110.+111.j]],
               [[112.+113.j, 114.+115.j, 116.+117.j, 118.+119.j],
                [120.+121.j, 122.+123.j, 124.+125.j, 126.+127.j]],
               [[128.+129.j, 130.+131.j, 132.+133.j, 134.+135.j],
                [136.+137.j, 138.+139.j, 140.+141.j, 142.+143.j]],
               [[144.+145.j, 146.+147.j, 148.+149.j, 150.+151.j],
                [152.+153.j, 154.+155.j, 156.+157.j, 158.+159.j]]]])
xk_ = tensor([[[[ 0.+1.j,  2.+3.j,  4.+5.j,  6.+7.j]],
               [[ 8.+9.j, 10.+11.j, 12.+13.j, 14.+15.j]],
               [[16.+17.j, 18.+19.j, 20.+21.j, 22.+23.j]],
               [[24.+25.j, 26.+27.j, 28.+29.j, 30.+31.j]],
               [[32.+33.j, 34.+35.j, 36.+37.j, 38.+39.j]]],
              [[[40.+41.j, 42.+43.j, 44.+45.j, 46.+47.j]],
               [[48.+49.j, 50.+51.j, 52.+53.j, 54.+55.j]],
               [[56.+57.j, 58.+59.j, 60.+61.j, 62.+63.j]],
               [[64.+65.j, 66.+67.j, 68.+69.j, 70.+71.j]],
               [[72.+73.j, 74.+75.j, 76.+77.j, 78.+79.j]]]])
```

- 根据 xq_ 将预先计算好的旋转角度 freqs_cis 修改成特定形状(与xq_中每个批次﹑每个头对应的张量形状一样).
  这里 xq_ 的形状为 (2, 5, 2, 4), 则 freqs_cis 的形状为 (1, 5, 1, 4).

```
freqs_cis = tensor([[[[ 1.0000+0.0000j,  1.0000+0.0000j,  1.0000+0.0000j,  1.0000+0.0000j]],
                     [[ 0.5403+0.8415j,  0.9950+0.0998j,  0.9999+0.0100j,  1.0000+0.0010j]],
                     [[-0.4161+0.9093j,  0.9801+0.1987j,  0.9998+0.0200j,  1.0000+0.0020j]],
                     [[-0.9900+0.1411j,  0.9553+0.2955j,  0.9996+0.0300j,  1.0000+0.0030j]],
                     [[-0.6536-0.7568j,  0.9211+0.3894j,  0.9992+0.0400j,  1.0000+0.0040j]]]])
```

- 计算 xq 和 xk 的旋转嵌入﹐以 xq 的计算为例
  - a. 计算 xq_ 与 freqs_cis 的复数乘法, 它将 xq_ 中的每个元素与 freqs_cis 中的对应元素相乘.
    例如, xq_ 中位置为 (0, 1, 1, 2) 的元素为 28.+29.j, 需要与 freqs_cis 中位于 (0, 1, 0, 2)

的元素 0.9999+0.0100j 相乘(由于小数位数原因, 实际上这个元素是 0.99995+0.009999833j)它们的乘积为 27.7086+29.2785j.

  - b. 将复数张量转换为实数张量。如果复数张量中的元素是 a + bi, 那转换后的实数张量是一个形如 [a, b] 的向量。
  - c. 将张量在第 3 维上进行展平, 即把第 3 维及其之后的所有维度展平成一个维度. 例如, 待展平的张量形状是 (2, 5, 2, 4, 2), 第 3 维上进行展平操作之后的张量形状是 (2, 5, 2, 8). 根据上面的计算, 得到的 xq_out 中位置在 (0, 1, 1, 4:6) 的元素应该是 [27.7086, 29.2785].

```
xq_out = tensor([[[[   0.0000,    1.0000,    2.0000,    3.0000,    4.0000,    5.0000,    6.0000,    7.0000],
                   [   8.0000,    9.0000,   10.0000,   11.0000,   12.0000,   13.0000,   14.0000,   15.0000]],
                  [[  -5.6602,   22.6487,   16.0132,   20.7021,   19.7890,   21.1989,   21.9770,   23.0220],
                   [  -8.0695,   33.7029,   23.1746,   29.4608,   27.7086,   29.2785,   29.9690,   31.0300]],
                  [[ -43.3235,   15.3647,   26.3688,   41.0571,   35.2528,   37.7126,   37.9219,   39.0759],
                   [ -53.9271,   19.3099,   32.6200,   50.4870,   43.0913,   45.8709,   45.9059,   47.0919]],
                  [[ -54.4345,  -41.7359,   32.6953,   63.4982,   50.3868,   54.5359,   53.8348,   55.1618],
                   [ -63.4834,  -48.5269,   37.9738,   73.5050,   58.1433,   62.7723,   61.8107,   63.1857]],
                  [[   7.3590,  -90.9222,   34.6990,   87.4127,   65.1863,   71.6641,   69.7154,   71.2794],
                   [   8.1842, -102.2058,   38.9521,   97.8965,   72.8600,   79.9776,   77.6834,   79.3114]]],
                 [[[  80.0000,   81.0000,   82.0000,   83.0000,   84.0000,   85.0000,   86.0000,   87.0000],
                   [  88.0000,   89.0000,   90.0000,   91.0000,   92.0000,   93.0000,   94.0000,   95.0000]],
                  [[ -29.7537,  133.1905,   87.6269,  108.2891,   98.9850,  101.9949,  101.8969,  103.1020],
                   [ -32.1630,  144.2447,   94.7883,  117.0478,  106.9046,  110.0745,  109.8889,  111.1099]],
                  [[-149.3591,   54.8167,   88.8806,  135.3560,  113.6370,  119.2964,  117.7618,  119.2358],
                   [-159.9626,   58.7619,   95.1318,  144.7859,  121.4754,  127.4548,  125.7457,  127.2517]],
                  [[-144.9235, -109.6457,   85.4806,  163.5667,  127.9512,  136.8996,  133.5944,  135.4014],
                   [-153.9724, -116.4366,   90.7591,  173.5736,  135.7076,  145.1359,  141.5704,  143.4254]],
                  [[  15.6117, -203.7579,   77.2304,  192.2510,  141.9232,  154.7992,  149.3948,  151.5988],
                   [  16.4370, -215.0414,   81.4836,  202.7349,  149.5969,  163.1128,  157.3627,  159.6307]]]])
xk_out = tensor([[[[   0.0000,    1.0000,    2.0000,    3.0000,    4.0000,    5.0000,    6.0000,    7.0000]],
                  [[  -3.2508,   11.5945,    8.8519,   11.9434,   11.8694,   13.1193,   13.9850,   15.0140]],
                  [[ -22.1164,    7.4743,   13.8665,   22.1973,   19.5760,   21.3958,   21.9540,   23.0440]],
                  [[ -27.2878,  -21.3629,   16.8597,   33.4776,   27.1175,   29.8268,   29.9069,   31.0899]],
                  [[   4.0579,  -45.7879,   17.6864,   45.4774,   34.4916,   38.4100,   37.8437,   39.1517]]],
                 [[[  40.0000,   41.0000,   42.0000,   43.0000,   44.0000,   45.0000,   46.0000,   47.0000]],
                  [[ -15.2976,   66.8654,   44.6587,   55.7369,   51.4674,   53.5173,   53.9450,   55.0540]],
                  [[ -75.1342,   27.2003,   45.1224,   69.3467,   58.7681,   62.1877,   61.8739,   63.1239]],
                  [[ -72.5323,  -55.3178,   43.2524,   83.5119,   65.8997,   71.0087,   69.7867,   71.2097]],
                  [[   8.1842, -102.2058,   38.9521,   97.8965,   72.8600,   79.9776,   77.6834,   79.3114]]]])
```

对比 xk 和 xk_out, 可以看出：

- 旋转位置编码的嵌入, 在第1个token(即位置为0)上, 与原始的嵌入是一致的;
- 在后面的token上, 嵌入发生了变化. 其中嵌入向量中越靠前的元素变化程度越大，而越靠后的元素变化程度越小.
- 不同样本同一个位置的token，旋转角度是相同的.

参考文献

ROFORMER: ENHANCED TRANSFORMER WITH ROTARY POSITION EMBEDDING.

一文看懂 LLaMA 中的旋转式位置编码(Rotary Position Embedding)

LLaMA 官方实现代码