# P1 【迪哥谈AI】还不懂Llama3的有救了00:36

P1 【迪哥谈AI】还不懂Llama3的有救了？迪哥终于把从零到一实现Llama3大模型讲清楚了！一个非常落地的AI大模型项目！

<https://github.com/naklecha/llama3-from-scratch>

llama3 从头开始实现

在这个文件中，我从头开始实现了 llama3 ，一次一个张量和矩阵乘法。

此外，我将直接从 Meta 为 llama3 提供的模型文件加载张量，您需要在运行此文件之前下载权重。 以下是下载砝码的官方链接: <https://llama.meta.com/lama-downloads/>

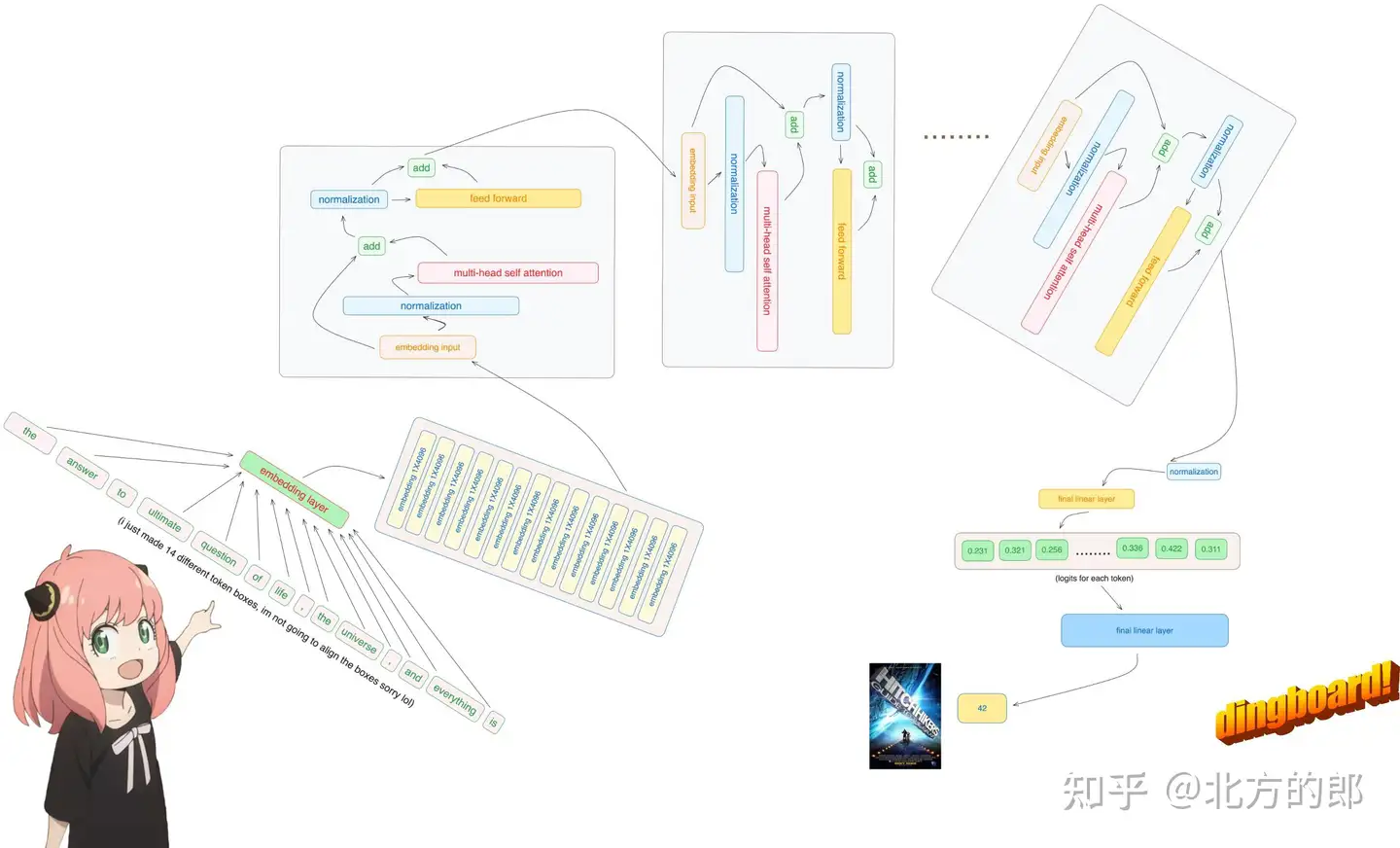
# P1+详解llama3模型结构，从头构建llama3

**[北方的郎](https://www.zhihu.com/people/albert-chen-4" \t "https://zhuanlan.zhihu.com/p/_blank)** <https://zhuanlan.zhihu.com/p/698899728>

翻译自naklecha的llama3-from-scratch，翻译的过程中有调整。原文地址：

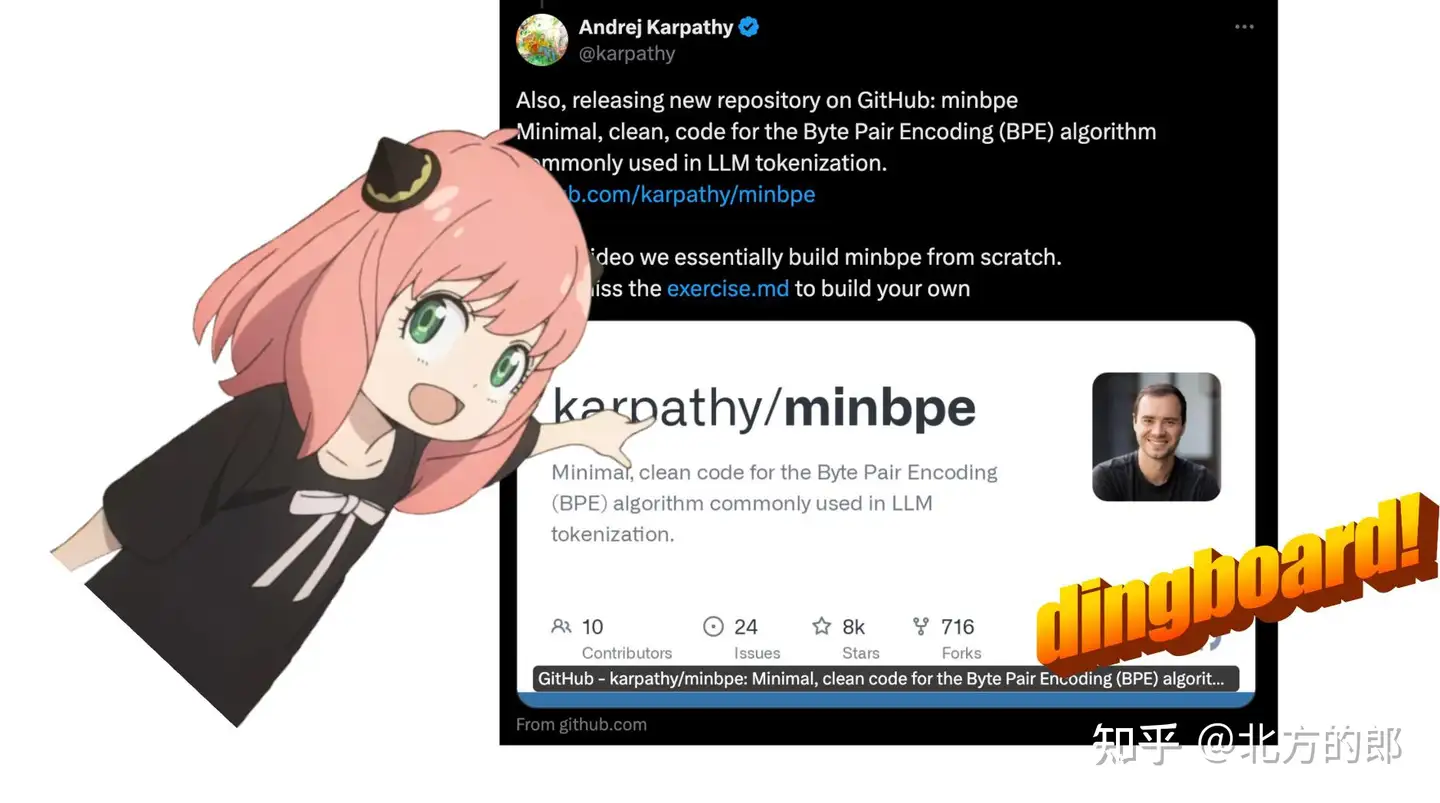
<https://github.com/naklecha/llama3-from-scratch>

在这个文件中，我从头开始实现了 llama3，一次一个张量和矩阵乘法。另外，我将直接从Meta为 llama3 提供的模型文件加载张量，您需要在运行此文件之前下载权重。这是下载权重的官方链接：https://llama.meta.com/llama-downloads/



## 1.分词器（tokenizer）

我不会实现 bpe tokenizer（但 andrej karpathy 有一个非常干净的实现）  
他的实现链接：[https://github.com/karpathy/minbpe](https://link.zhihu.com/?target=https://github.com/karpathy/minbpe" \t "https://zhuanlan.zhihu.com/p/_blank)



from pathlib import Path

import tiktoken

from tiktoken.load import load\_tiktoken\_bpe

import torch

import json

import matplotlib.pyplot as plt

tokenizer\_path = "Meta-Llama-3-8B/tokenizer.model"

special\_tokens = [

            "<|begin\_of\_text|>",

            "<|end\_of\_text|>",

            "<|reserved\_special\_token\_0|>",

            "<|reserved\_special\_token\_1|>",

            "<|reserved\_special\_token\_2|>",

            "<|reserved\_special\_token\_3|>",

            "<|start\_header\_id|>",

            "<|end\_header\_id|>",

            "<|reserved\_special\_token\_4|>",

            "<|eot\_id|>",  # end of turn

        ] + [f"<|reserved\_special\_token\_{i}|>" for i in range(5, 256 - 5)]

mergeable\_ranks = load\_tiktoken\_bpe(tokenizer\_path)

tokenizer = tiktoken.Encoding(

    name=Path(tokenizer\_path).name,

    pat\_str=r"(?i:'s|'t|'re|'ve|'m|'ll|'d)|[^\r\n\p{L}\p{N}]?\p{L}+|\p{N}{1,3}| ?[^\s\p{L}\p{N}]+[\r\n]\*|\s\*[\r\n]+|\s+(?!\S)|\s+",

    mergeable\_ranks=mergeable\_ranks,

    special\_tokens={token: len(mergeable\_ranks) + i for i, token in enumerate(special\_tokens)},

)

tokenizer.decode(tokenizer.encode("hello world!"))

'hello world!'

## 2.读取模型文件

通常，阅读本文取决于模型类的编写方式以及其中的变量名称。  
但由于我们是从头开始实现 llama3，因此我们将一次读取一个张量该文件。



model = torch.load("Meta-Llama-3-8B/consolidated.00.pth")

print(json.dumps(list(model.keys())[:20], indent=4))

[

    "tok\_embeddings.weight",

    "layers.0.attention.wq.weight",

    "layers.0.attention.wk.weight",

    "layers.0.attention.wv.weight",

    "layers.0.attention.wo.weight",

    "layers.0.feed\_forward.w1.weight",

    "layers.0.feed\_forward.w3.weight",

    "layers.0.feed\_forward.w2.weight",

    "layers.0.attention\_norm.weight",

    "layers.0.ffn\_norm.weight",

    "layers.1.attention.wq.weight",

    "layers.1.attention.wk.weight",

    "layers.1.attention.wv.weight",

    "layers.1.attention.wo.weight",

    "layers.1.feed\_forward.w1.weight",

    "layers.1.feed\_forward.w3.weight",

    "layers.1.feed\_forward.w2.weight",

    "layers.1.attention\_norm.weight",

    "layers.1.ffn\_norm.weight",

    "layers.2.attention.wq.weight"

]

with open("Meta-Llama-3-8B/params.json", "r") as f:

    config = json.load(f)

config

{'dim': 4096,

 'n\_layers': 32,

 'n\_heads': 32,

 'n\_kv\_heads': 8,

 'vocab\_size': 128256,

 'multiple\_of': 1024,

 'ffn\_dim\_multiplier': 1.3,

 'norm\_eps': 1e-05,

 'rope\_theta': 500000.0}

我们使用此配置来推断有关模型的详细信息，例如

* 该模型有 32 个transformer layers
* 每个多头注意力块有 32 个头
* 词汇大小等等

dim = config["dim"]

n\_layers = config["n\_layers"]

n\_heads = config["n\_heads"]

n\_kv\_heads = config["n\_kv\_heads"]

vocab\_size = config["vocab\_size"]

multiple\_of = config["multiple\_of"]

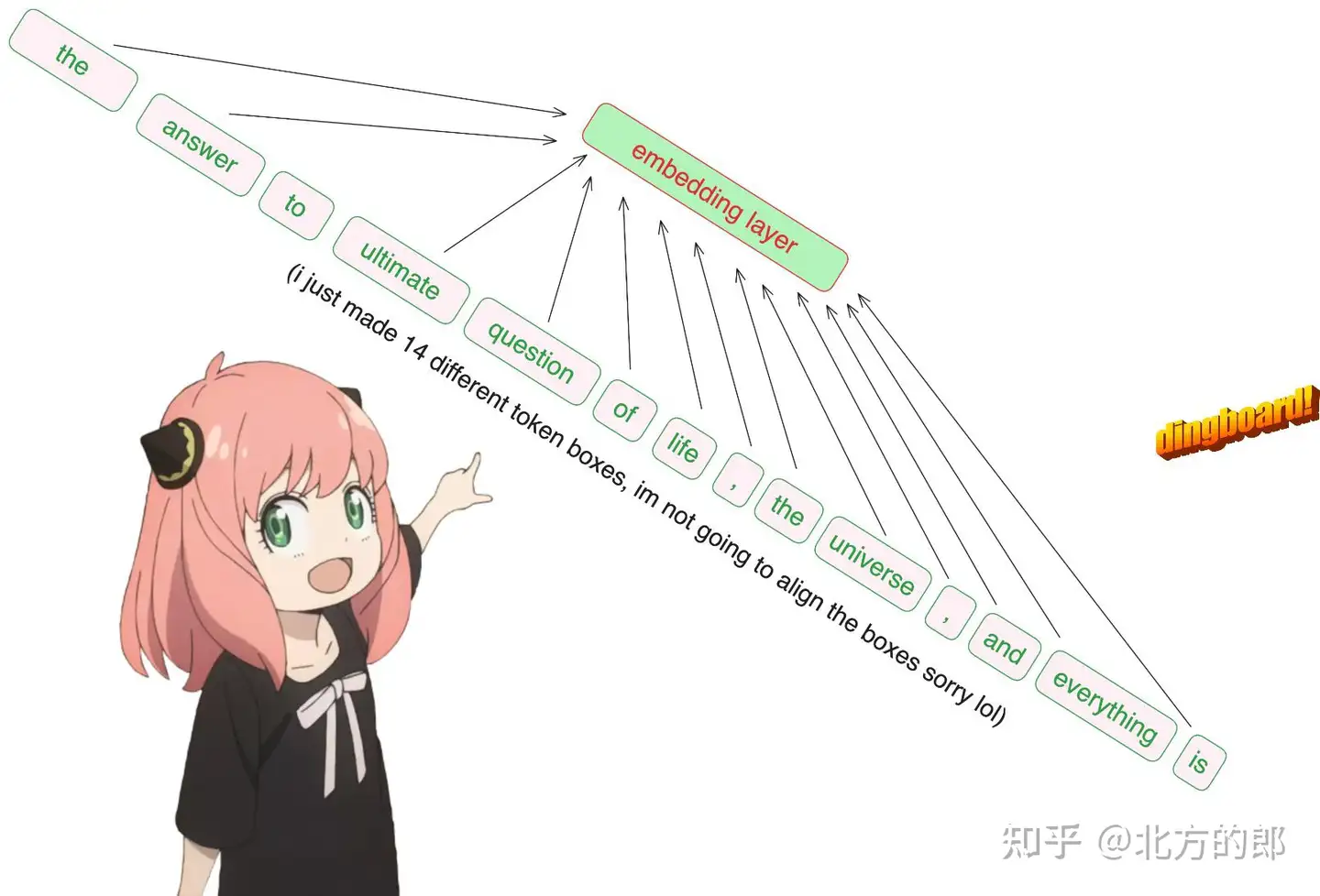
ffn\_dim\_multiplier = config["ffn\_dim\_multiplier"]

norm\_eps = config["norm\_eps"]

rope\_theta = torch.tensor(config["rope\_theta"])

## 3.将文本转换为标记（tokens）

这里我们使用 tiktoken （我认为是一个 openai 库）作为 tokenizer



prompt = "the answer to the ultimate question of life, the universe, and everything is "

tokens = [128000] + tokenizer.encode(prompt)

print(tokens)

tokens = torch.tensor(tokens)

prompt\_split\_as\_tokens = [tokenizer.decode([token.item()]) for token in tokens]

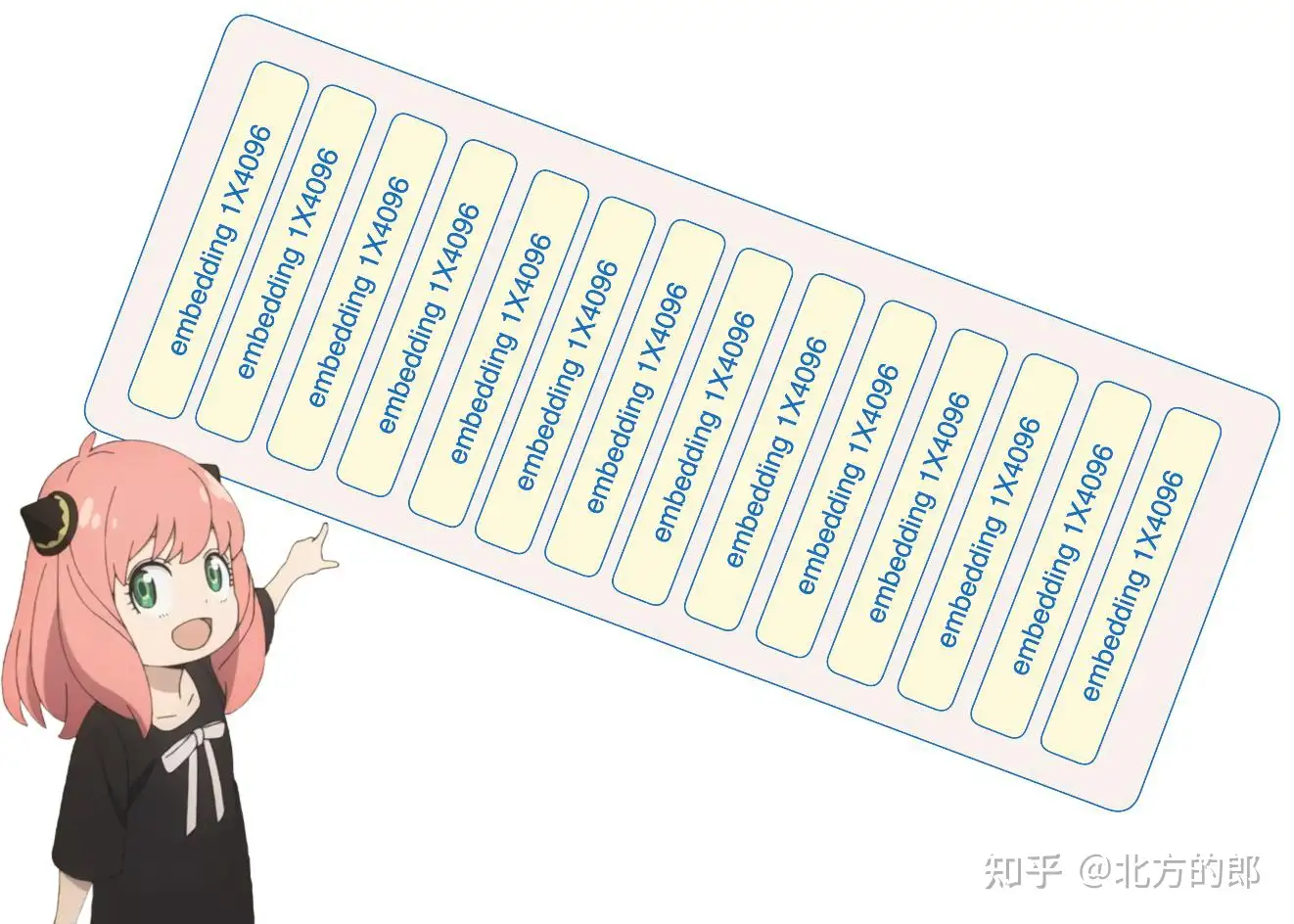
print(prompt\_split\_as\_tokens)

[128000, 1820, 4320, 311, 279, 17139, 3488, 315, 2324, 11, 279, 15861, 11, 323, 4395, 374, 220]

['<|begin\_of\_text|>', 'the', ' answer', ' to', ' the', ' ultimate', ' question', ' of', ' life', ',', ' the', ' universe', ',', ' and', ' everything', ' is', ' ']

## 4.将标记转换为嵌入（embedding）

抱歉，但无论如何，这是代码库中我使用内置神经网络模块的唯一部分，因此我们的 [17x1] 标记现在是 [17x4096]，即长度为 4096 的 17 个嵌入（每个标记一个）  
注意：跟踪形状，它让你更容易理解一切



embedding\_layer = torch.nn.Embedding(vocab\_size, dim)

embedding\_layer.weight.data.copy\_(model["tok\_embeddings.weight"])

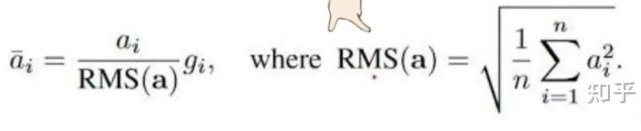
token\_embeddings\_unnormalized = embedding\_layer(tokens).to(torch.bfloat16)

token\_embeddings\_unnormalized.shape

torch.Size([17, 4096])

## **5.然后我们使用 rms 归一化对嵌入进行归一化**

请注意，在这一步之后，形状不会改变，这些值只是需要记住的标准化内容，我们需要一个norm\_eps（来自配置），因为我们不想意外地将rms设置为0并除以0，这里是公式：



# def rms\_norm(tensor, norm\_weights):

#     rms = (tensor.pow(2).mean(-1, keepdim=True) + norm\_eps)\*\*0.5

#     return tensor \* (norm\_weights / rms)

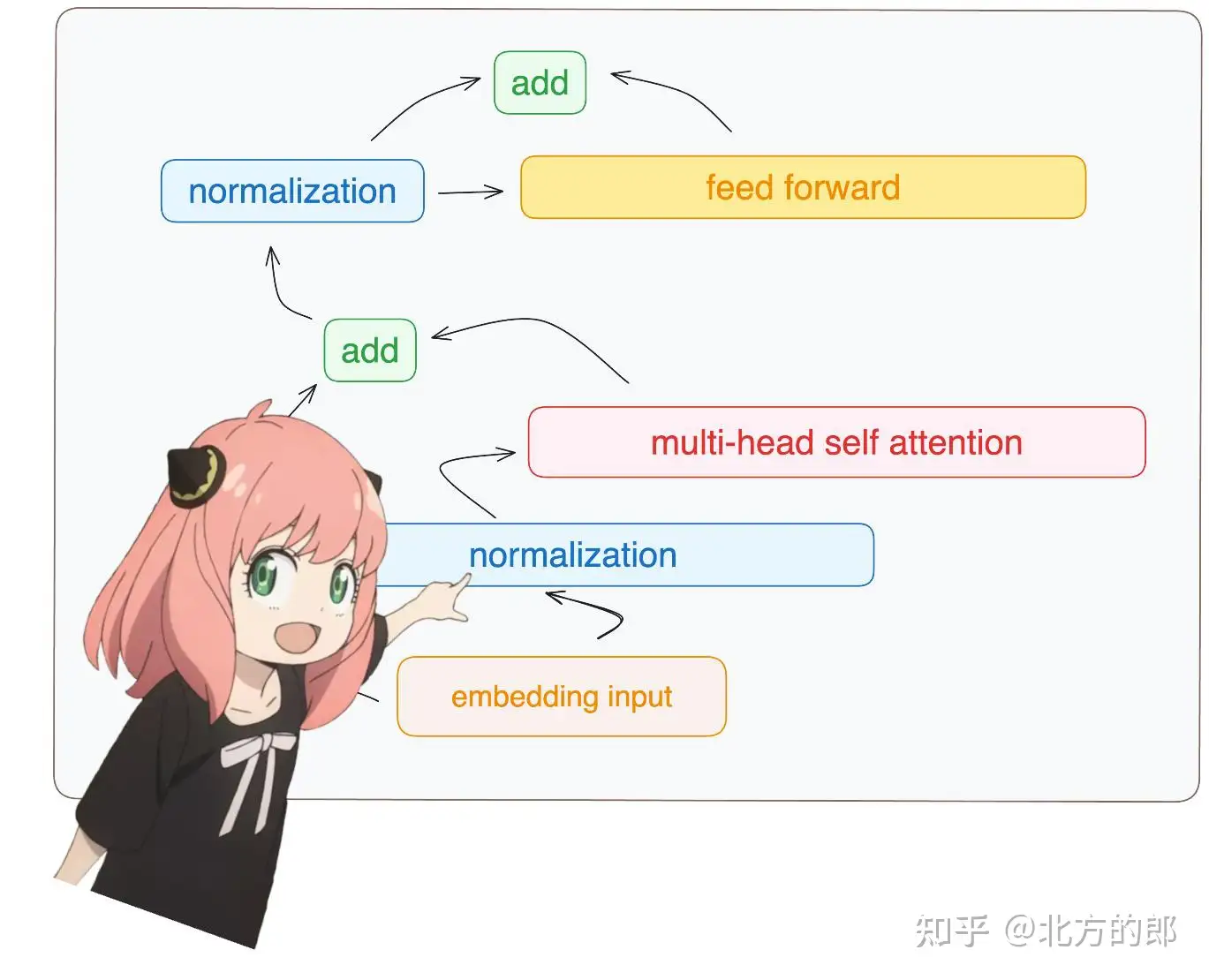
def rms\_norm(tensor, norm\_weights):

    return (tensor \* torch.rsqrt(tensor.pow(2).mean(-1, keepdim=True) + norm\_eps)) \* norm\_weights

## 6.构建transformer的第一层

### 6.1正常化

无论如何，你会看到我从模型字典访问layer.0（这是第一层），所以在标准化之后我们的形状仍然[17x4096]与嵌入相同但标准化



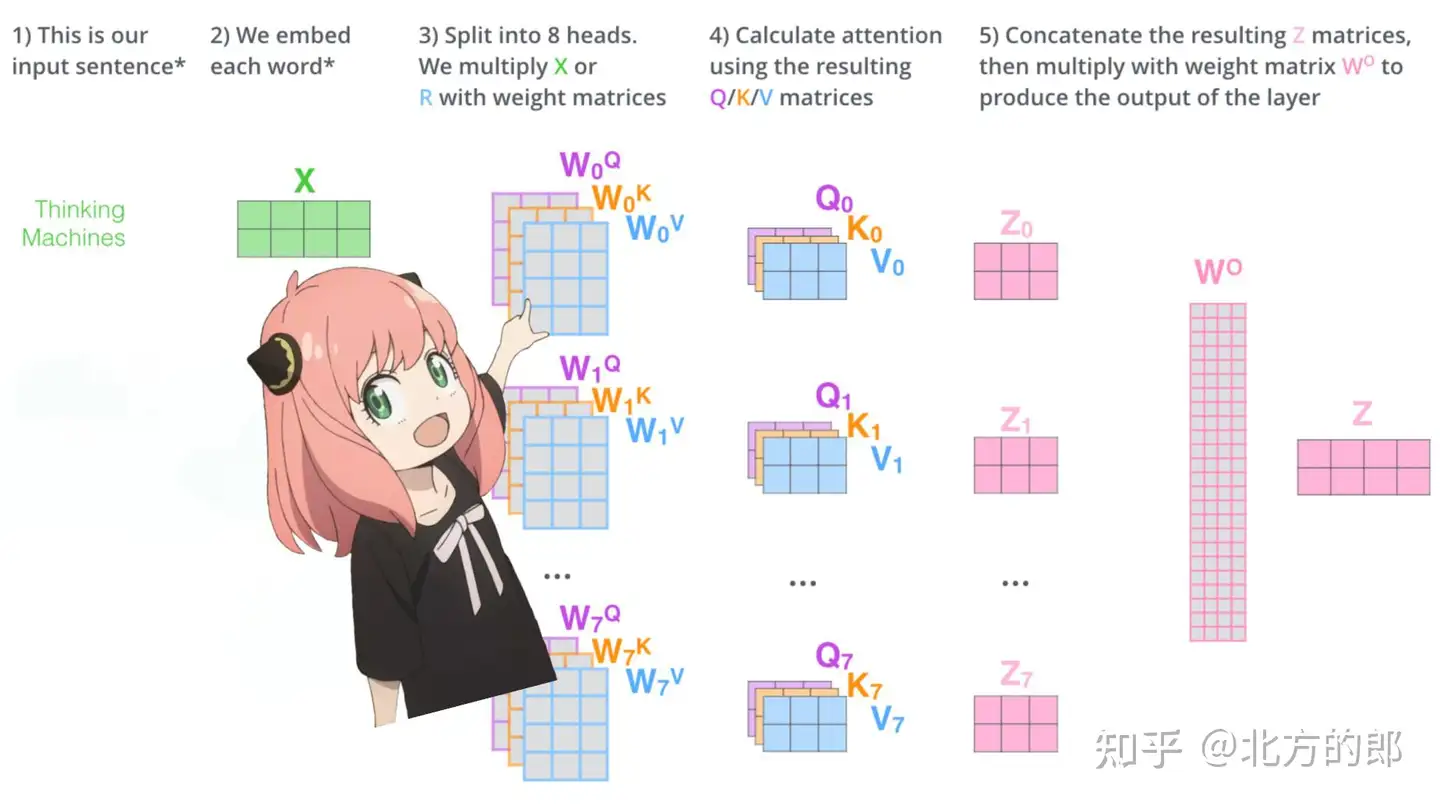
token\_embeddings = rms\_norm(token\_embeddings\_unnormalized, model["layers.0.attention\_norm.weight"])

token\_embeddings.shape

torch.Size([17, 4096])

### 6.2从头开始实施注意力

让我们加载transformer第一层的注意力头



> 当我们从模型加载查询、键、值和输出向量时，我们注意到形状为 [4096x4096]、[1024x4096]、[1024x4096]、[4096x4096]  
> 乍一看这很奇怪，因为理想情况下我们想要每个 q ,k,v 和 o 分别代表每个头  
> 代码的作者将它们捆绑在一起，因为它很容易，有助于并行化注意力头乘法。  
> 我要打开所有东西...

print(

    model["layers.0.attention.wq.weight"].shape,

    model["layers.0.attention.wk.weight"].shape,

    model["layers.0.attention.wv.weight"].shape,

    model["layers.0.attention.wo.weight"].shape

)

torch.Size([4096, 4096]) torch.Size([1024, 4096]) torch.Size([1024, 4096]) torch.Size([4096, 4096])

## **展开查询**

在下一节中，我们将从多个注意力头中解开查询，生成的形状为 [32x128x4096]，其中 32 是 llama3 中注意力头的数量，128 是查询向量的大小，4096 是令牌嵌入的大小

q\_layer0 = model["layers.0.attention.wq.weight"]

head\_dim = q\_layer0.shape[0] // n\_heads

q\_layer0 = q\_layer0.view(n\_heads, head\_dim, dim)

q\_layer0.shape

torch.Size([32, 128, 4096])

我要实现第一层的第一个头

这里我访问第一层的查询权重矩阵第一个头，这个查询权重矩阵的大小是[128x4096]

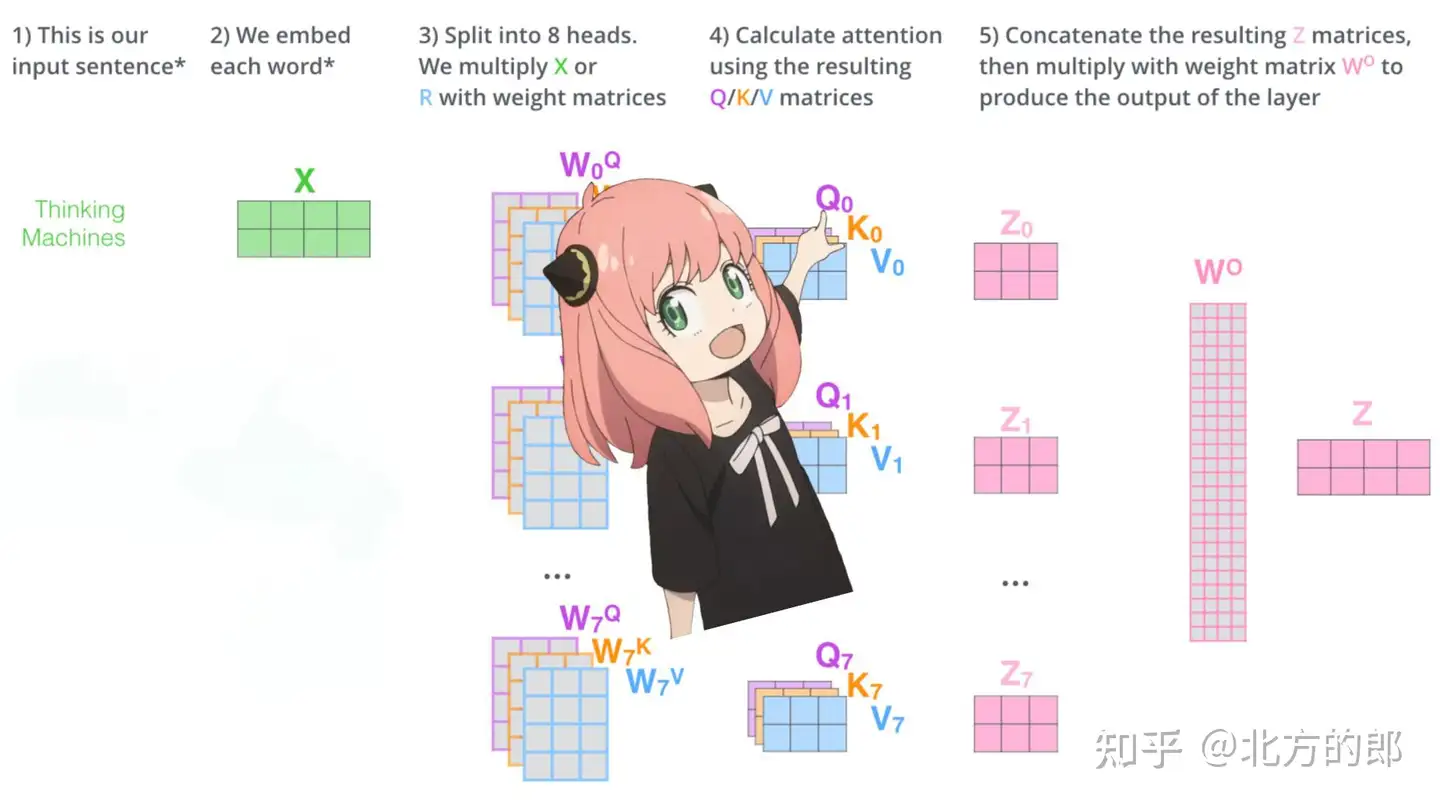
q\_layer0\_head0 = q\_layer0[0]

q\_layer0\_head0.shape

torch.Size([128, 4096])

我们现在将查询权重与令牌嵌入相乘，以接收令牌的查询

在这里你可以看到结果的形状是 [17x128]，这是因为我们有 17 个标记，每个标记都有一个 128 长度的查询。



q\_per\_token = torch.matmul(token\_embeddings, q\_layer0\_head0.T)

q\_per\_token.shape

torch.Size([17, 128])

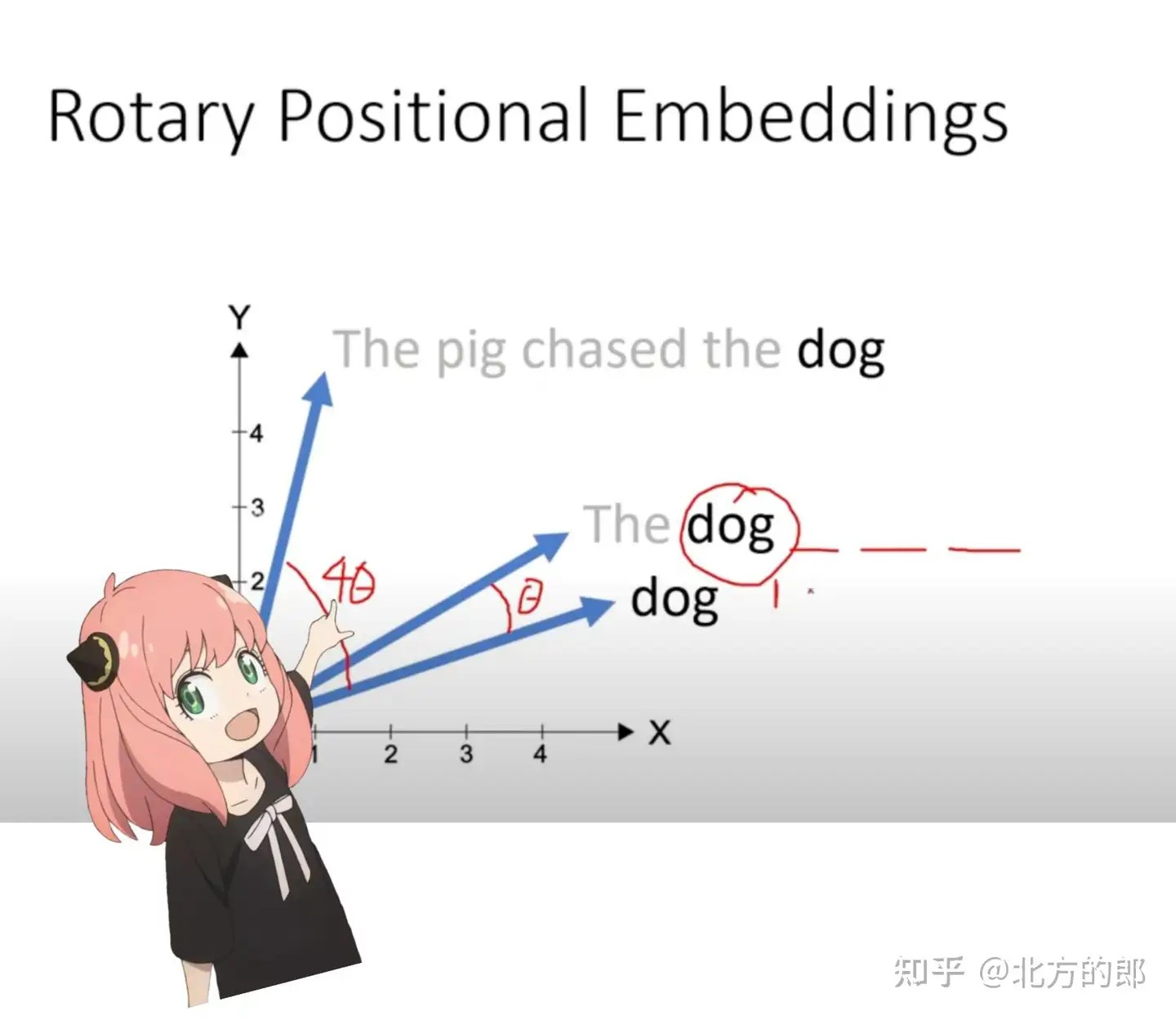
## **定位编码**

我们现在处于这样一个阶段：提示中的每个标记都有一个查询向量，但如果你仔细想想——单独的查询向量不知道提示中的位置。  
查询：“生命、宇宙和一切的终极问题的答案是”  
在我们的提示中我们已经使用了“the”三次，我们需要所有 3 个“the”标记的查询向量具有不同的查询向量（每个大小 [1x128]）基于它们在查询中的位置。我们使用 RoPE（旋转位置嵌入）执行这些旋转。

## **RoPE**

观看此视频（这就是我观看的）以理解数学。

[https://www.youtube.com/watch?v=o29P0Kpobz0&t=530s](https://link.zhihu.com/?target=https://www.youtube.com/watch?v=o29P0Kpobz0&t=530s" \t "https://zhuanlan.zhihu.com/p/_blank)

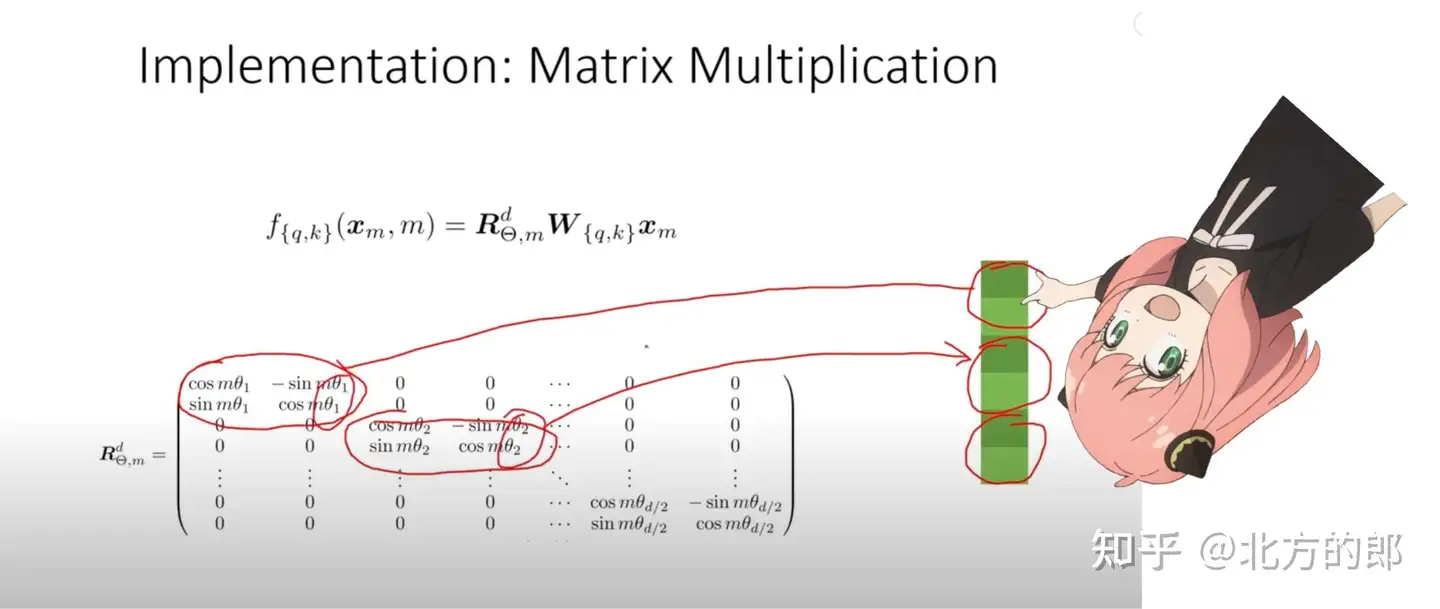


q\_per\_token\_split\_into\_pairs = q\_per\_token.float().view(q\_per\_token.shape[0], -1, 2)

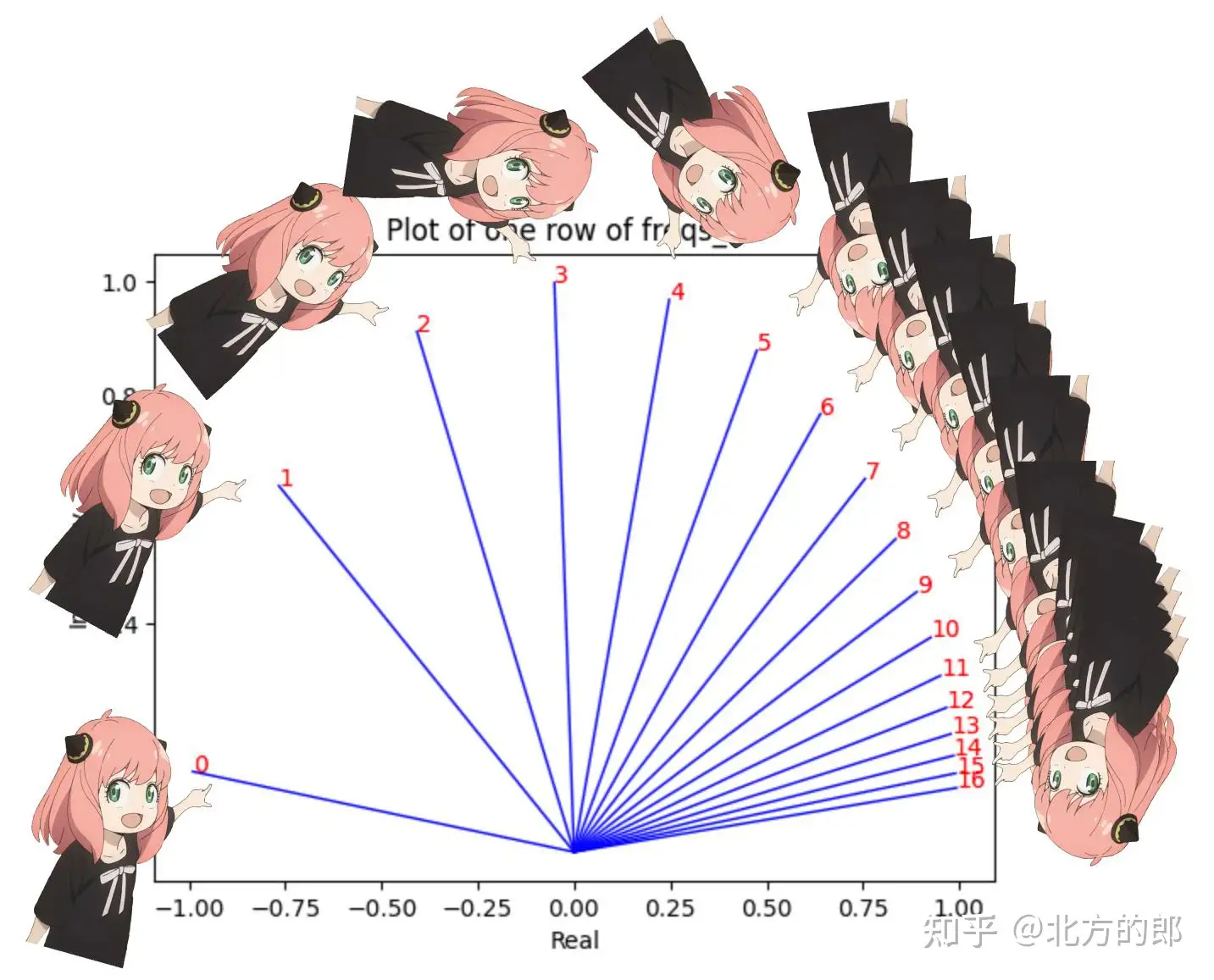
q\_per\_token\_split\_into\_pairs.shape

torch.Size([17, 64, 2])

在上面的步骤中，我们将查询向量分成对，我们对每对应用旋转角度偏移！  
我们现在有一个大小为 [17x64x2] 的向量，这是针对提示中的每个标记将 128 个长度的查询分为 64 对！这 64 对中的每一对都将旋转 m\*(theta)，其中 m 是我们旋转查询的标记的位置！



## **使用复数的点积来旋转向量**



zero\_to\_one\_split\_into\_64\_parts = torch.tensor(range(64))/64

zero\_to\_one\_split\_into\_64\_parts

tensor([0.0000, 0.0156, 0.0312, 0.0469, 0.0625, 0.0781, 0.0938, 0.1094, 0.1250,

        0.1406, 0.1562, 0.1719, 0.1875, 0.2031, 0.2188, 0.2344, 0.2500, 0.2656,

        0.2812, 0.2969, 0.3125, 0.3281, 0.3438, 0.3594, 0.3750, 0.3906, 0.4062,

        0.4219, 0.4375, 0.4531, 0.4688, 0.4844, 0.5000, 0.5156, 0.5312, 0.5469,

        0.5625, 0.5781, 0.5938, 0.6094, 0.6250, 0.6406, 0.6562, 0.6719, 0.6875,

        0.7031, 0.7188, 0.7344, 0.7500, 0.7656, 0.7812, 0.7969, 0.8125, 0.8281,

        0.8438, 0.8594, 0.8750, 0.8906, 0.9062, 0.9219, 0.9375, 0.9531, 0.9688,

        0.9844])

freqs = 1.0 / (rope\_theta \*\* zero\_to\_one\_split\_into\_64\_parts)

freqs

tensor([1.0000e+00, 8.1462e-01, 6.6360e-01, 5.4058e-01, 4.4037e-01, 3.5873e-01,

        2.9223e-01, 2.3805e-01, 1.9392e-01, 1.5797e-01, 1.2869e-01, 1.0483e-01,

        8.5397e-02, 6.9566e-02, 5.6670e-02, 4.6164e-02, 3.7606e-02, 3.0635e-02,

        2.4955e-02, 2.0329e-02, 1.6560e-02, 1.3490e-02, 1.0990e-02, 8.9523e-03,

        7.2927e-03, 5.9407e-03, 4.8394e-03, 3.9423e-03, 3.2114e-03, 2.6161e-03,

        2.1311e-03, 1.7360e-03, 1.4142e-03, 1.1520e-03, 9.3847e-04, 7.6450e-04,

        6.2277e-04, 5.0732e-04, 4.1327e-04, 3.3666e-04, 2.7425e-04, 2.2341e-04,

        1.8199e-04, 1.4825e-04, 1.2077e-04, 9.8381e-05, 8.0143e-05, 6.5286e-05,

        5.3183e-05, 4.3324e-05, 3.5292e-05, 2.8750e-05, 2.3420e-05, 1.9078e-05,

        1.5542e-05, 1.2660e-05, 1.0313e-05, 8.4015e-06, 6.8440e-06, 5.5752e-06,

        4.5417e-06, 3.6997e-06, 3.0139e-06, 2.4551e-06])

freqs\_for\_each\_token = torch.outer(torch.arange(17), freqs)

freqs\_cis = torch.polar(torch.ones\_like(freqs\_for\_each\_token), freqs\_for\_each\_token)

freqs\_cis.shape

# viewing tjhe third row of freqs\_cis

value = freqs\_cis[3]

plt.figure()

for i, element in enumerate(value[:17]):

    plt.plot([0, element.real], [0, element.imag], color='blue', linewidth=1, label=f"Index: {i}")

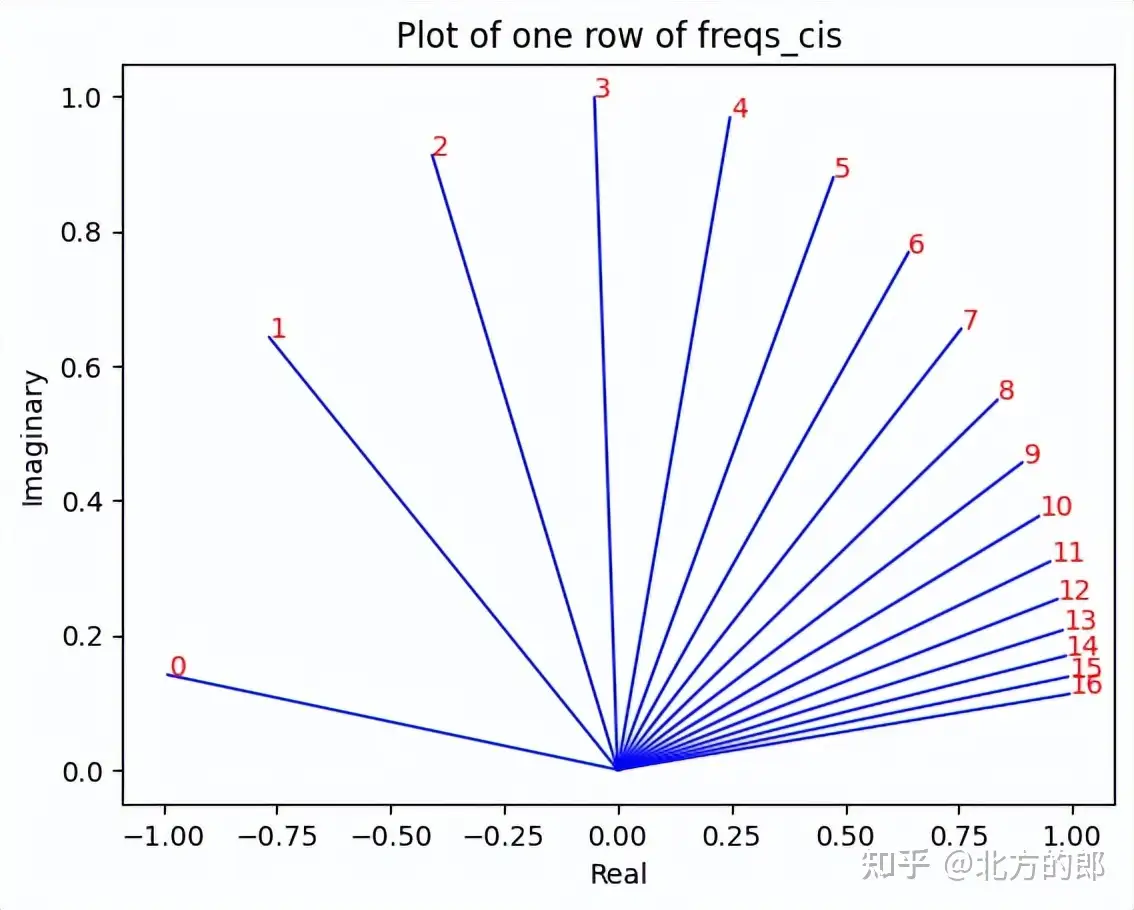
    plt.annotate(f"{i}", xy=(element.real, element.imag), color='red')

plt.xlabel('Real')

plt.ylabel('Imaginary')

plt.title('Plot of one row of freqs\_cis')

plt.show()



## **现在我们对每个标记（token）的查询元素都有一个复数（角度变化向量）**

我们可以将查询（我们分成对的查询）转换为复数，然后进行点积以根据位置诚实旋转查询，这想想就很美好:)

q\_per\_token\_as\_complex\_numbers = torch.view\_as\_complex(q\_per\_token\_split\_into\_pairs)

q\_per\_token\_as\_complex\_numbers.shape

torch.Size([17, 64])

q\_per\_token\_as\_complex\_numbers\_rotated = q\_per\_token\_as\_complex\_numbers \* freqs\_cis

q\_per\_token\_as\_complex\_numbers\_rotated.shape

torch.Size([17, 64])

## **得到旋转向量后**

我们可以通过再次将复数视为实数来返回成对的查询

q\_per\_token\_split\_into\_pairs\_rotated = torch.view\_as\_real(q\_per\_token\_as\_complex\_numbers\_rotated)

q\_per\_token\_split\_into\_pairs\_rotated.shape

torch.Size([17, 64, 2])

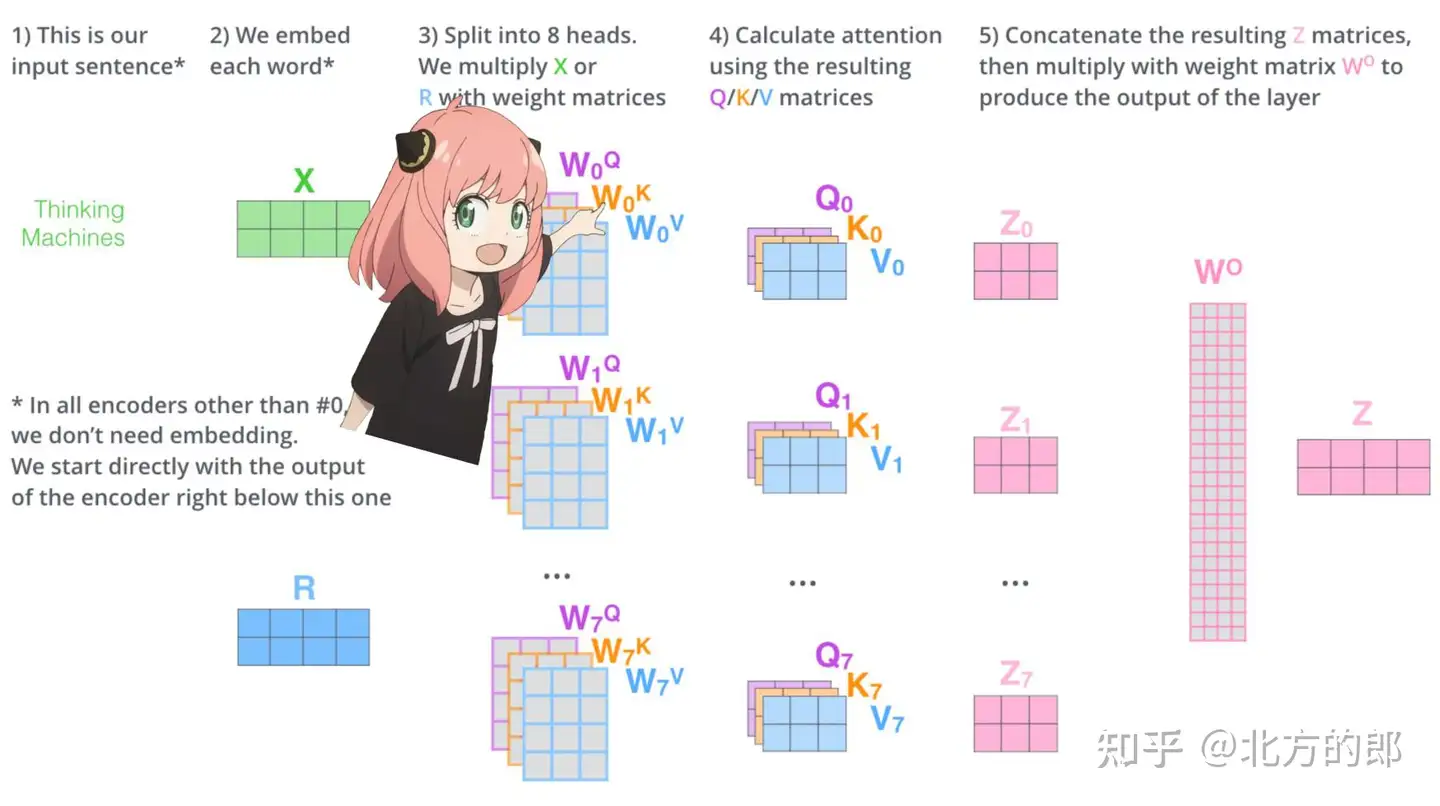
旋转的对现在被合并，我们现在有一个新的查询向量（旋转查询向量），其形状为 [17x128]，其中 17 是标记的数量，128 是查询向量的暗度

q\_per\_token\_rotated = q\_per\_token\_split\_into\_pairs\_rotated.view(q\_per\_token.shape)

q\_per\_token\_rotated.shape

torch.Size([17, 128])

## **键（几乎与查询相同）**



我太懒了，所以我不会对键进行数学计算，你需要记住的唯一事情是：  
> 键也生成维度 128 的键向量  
> 键的权重数量只有 1/4查询，这是因为键的权重一次在 4 个头之间共享，为了减少需要的计算数量  
> 键也会旋转以添加位置信息，就像查询一样，原因相同

k\_layer0 = model["layers.0.attention.wk.weight"]

k\_layer0 = k\_layer0.view(n\_kv\_heads, k\_layer0.shape[0] // n\_kv\_heads, dim)

k\_layer0.shape

torch.Size([8, 128, 4096])

k\_layer0\_head0 = k\_layer0[0]

k\_layer0\_head0.shape

torch.Size([128, 4096])

k\_per\_token = torch.matmul(token\_embeddings, k\_layer0\_head0.T)

k\_per\_token.shape

torch.Size([17, 128])

k\_per\_token\_split\_into\_pairs = k\_per\_token.float().view(k\_per\_token.shape[0], -1, 2)

k\_per\_token\_split\_into\_pairs.shape

torch.Size([17, 64, 2])

k\_per\_token\_as\_complex\_numbers = torch.view\_as\_complex(k\_per\_token\_split\_into\_pairs)

k\_per\_token\_as\_complex\_numbers.shape

torch.Size([17, 64])

k\_per\_token\_split\_into\_pairs\_rotated = torch.view\_as\_real(k\_per\_token\_as\_complex\_numbers \* freqs\_cis)

k\_per\_token\_split\_into\_pairs\_rotated.shape

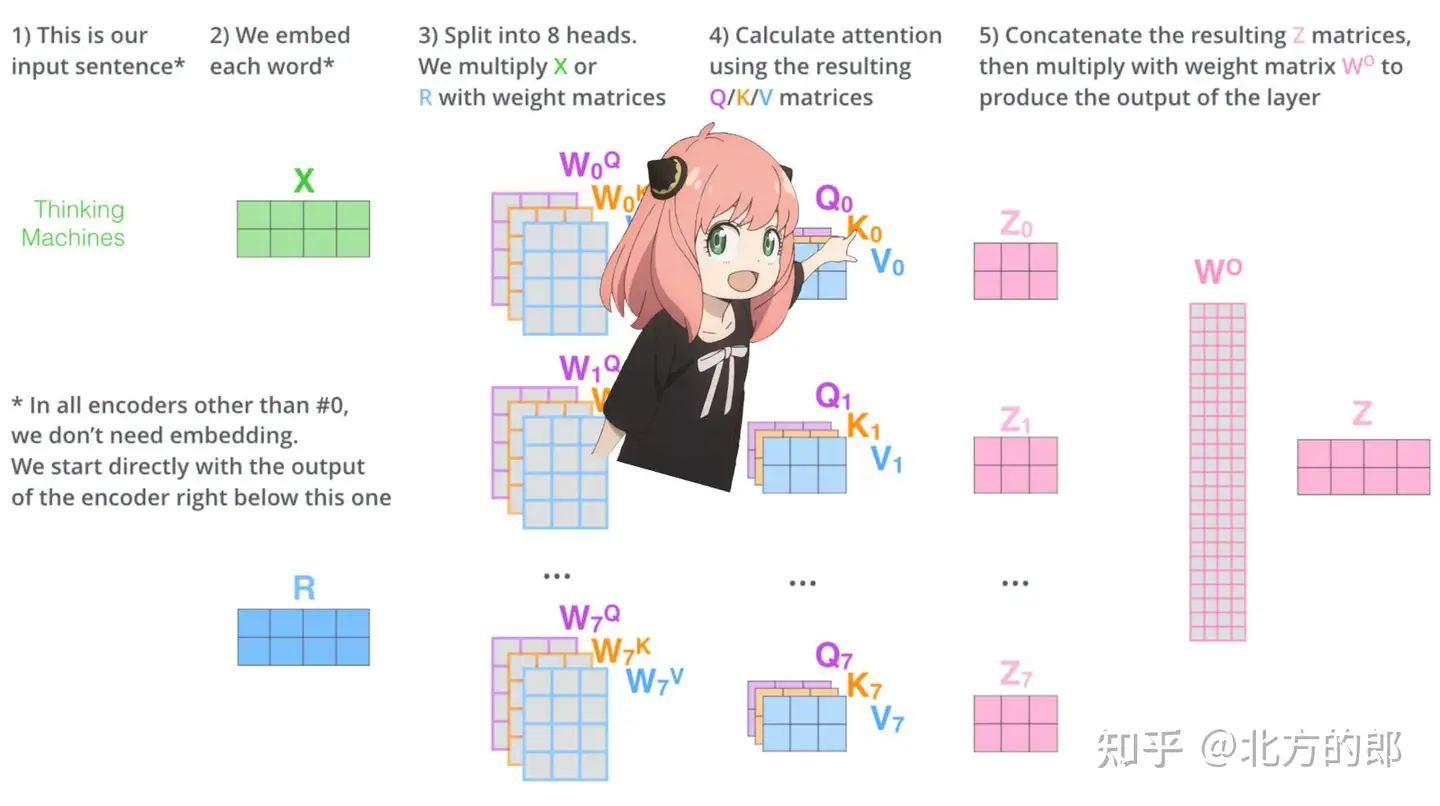
torch.Size([17, 64, 2])

k\_per\_token\_rotated = k\_per\_token\_split\_into\_pairs\_rotated.view(k\_per\_token.shape)

k\_per\_token\_rotated.shape

torch.Size([17, 128])

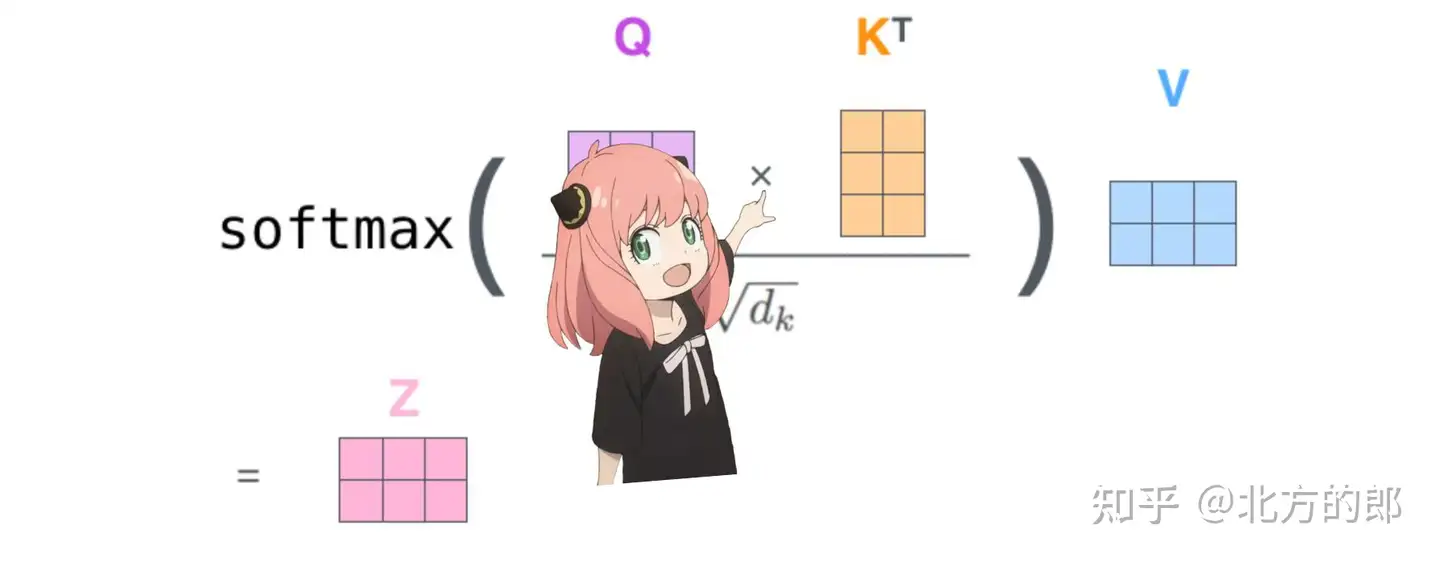
## **在此阶段，现在每个标记都有查询和键的旋转值。**



现在每个查询和键的形状都是 [17x128]。

## **在下一步中，我们将乘以查询和关键矩阵**

这样做将为我们提供一个将每个标记相互映射的分数，该分数描述了每个标记的查询与每个标记的密钥的相关程度。这是自我注意力 :)  
注意力分数矩阵 (qk\_per\_token) 的形状是 [17x17]，其中 17 是提示中的标记数量



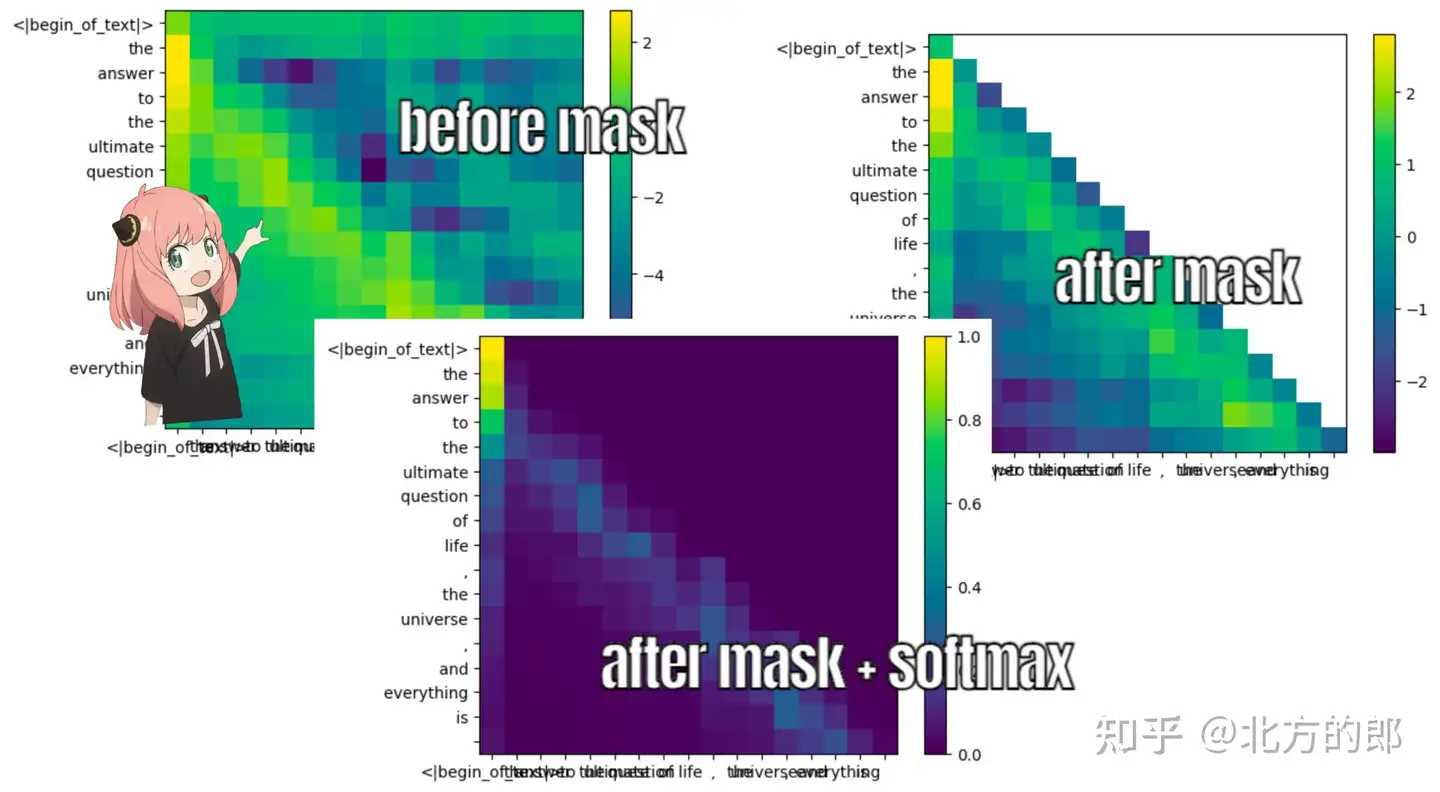
qk\_per\_token = torch.matmul(q\_per\_token\_rotated, k\_per\_token\_rotated.T)/(head\_dim)\*\*0.5

qk\_per\_token.shape

torch.Size([17, 17])

## **我们现在必须屏蔽查询关键分数**

在 llama3 的训练过程中，未来的 token qk 分数被屏蔽。为什么？因为在训练期间我们只学习使用过去的标记来预测标记。  
因此，在推理过程中，我们将未来的标记设置为零。



def display\_qk\_heatmap(qk\_per\_token):

\_, ax = plt.subplots()

im = ax.imshow(qk\_per\_token.to(float).detach(), cmap='viridis')

ax.set\_xticks(range(len(prompt\_split\_as\_tokens)))

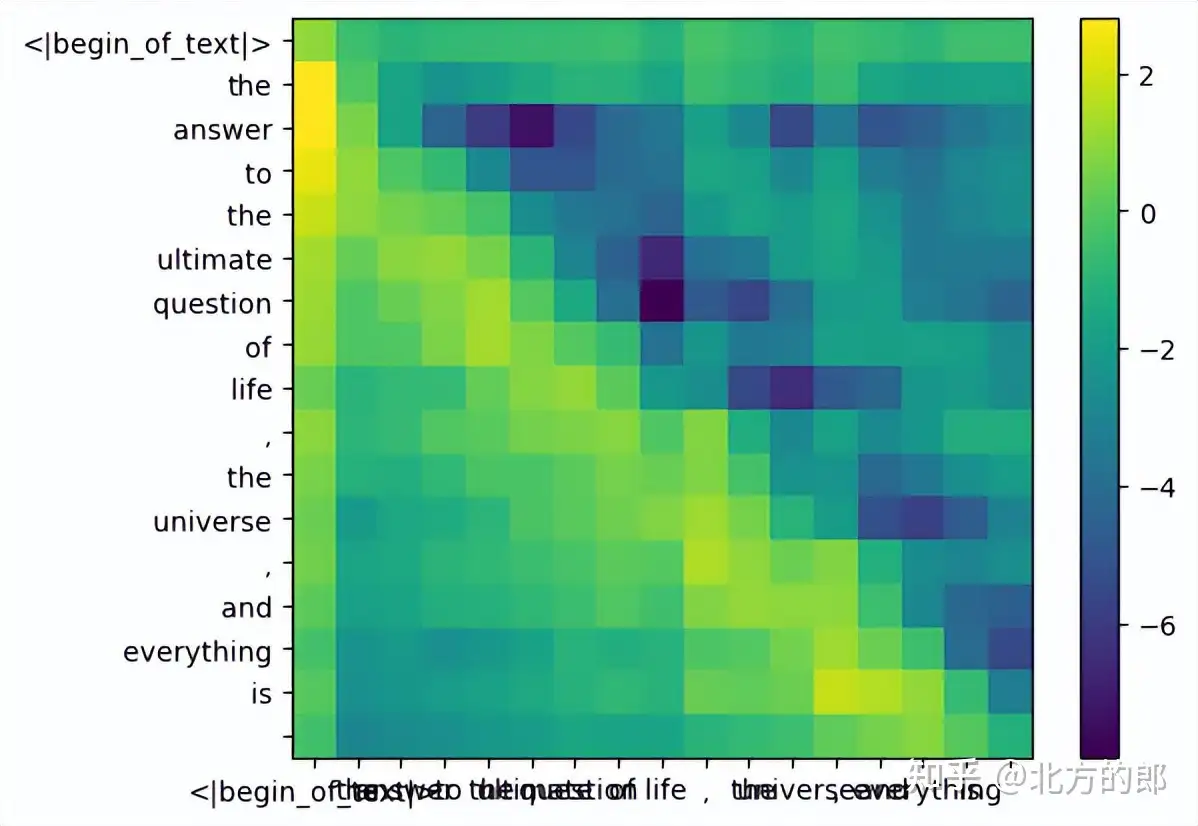
ax.set\_yticks(range(len(prompt\_split\_as\_tokens)))

ax.set\_xticklabels(prompt\_split\_as\_tokens)

ax.set\_yticklabels(prompt\_split\_as\_tokens)

ax.figure.colorbar(im, ax=ax)

display\_qk\_heatmap(qk\_per\_token)



mask = torch.full((len(tokens), len(tokens)), float("-inf"), device=tokens.device)

mask = torch.triu(mask, diagonal=1)

mask

tensor([[0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],

[0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],

[0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],

[0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],

[0., 0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],

[0., 0., 0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],

[0., 0., 0., 0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],

[0., 0., 0., 0., 0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],

[0., 0., 0., 0., 0., 0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., -inf, -inf, -inf, -inf],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., -inf, -inf, -inf],

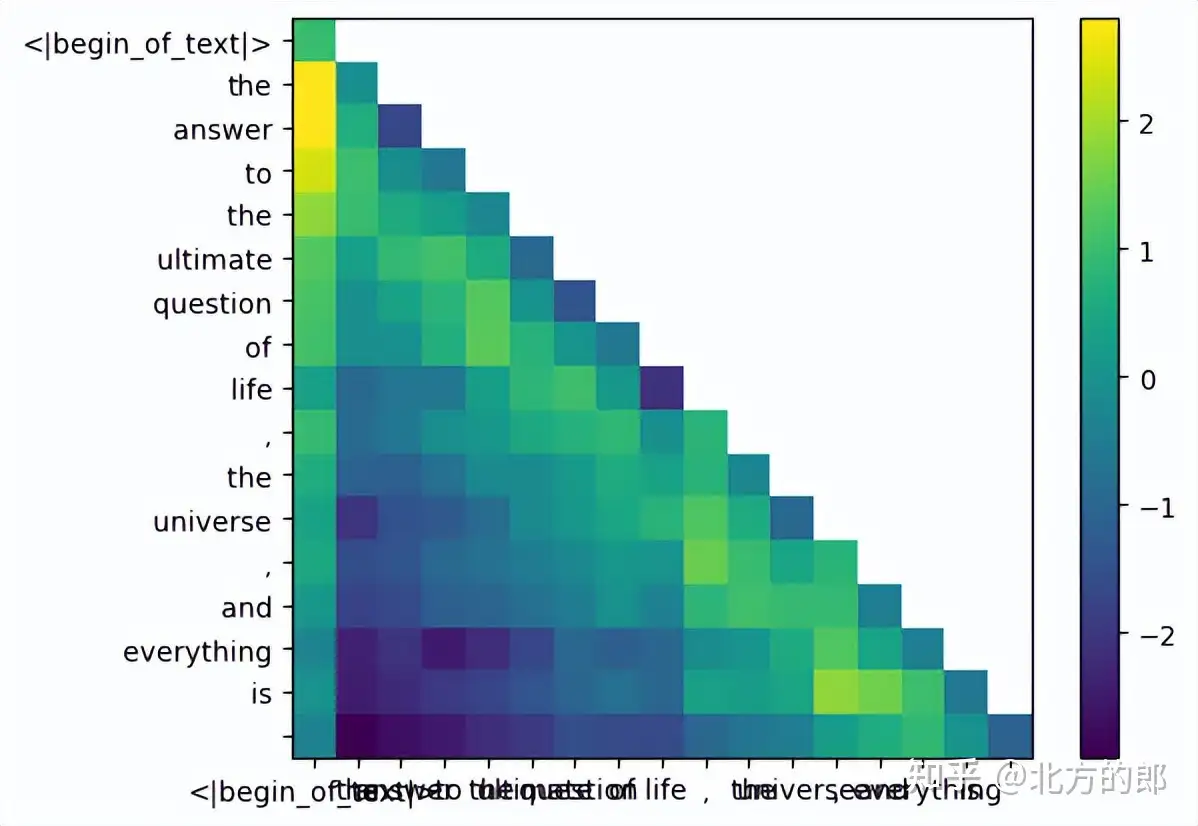
[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., -inf, -inf],

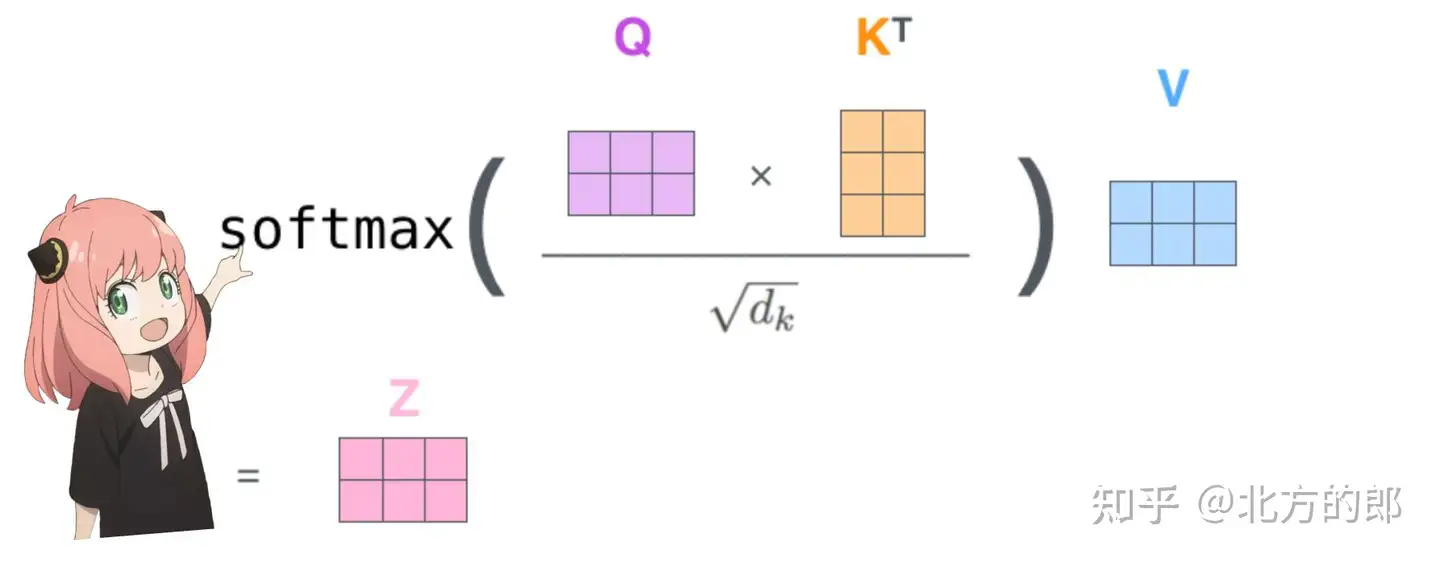
[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., -inf],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])

qk\_per\_token\_after\_masking = qk\_per\_token + mask

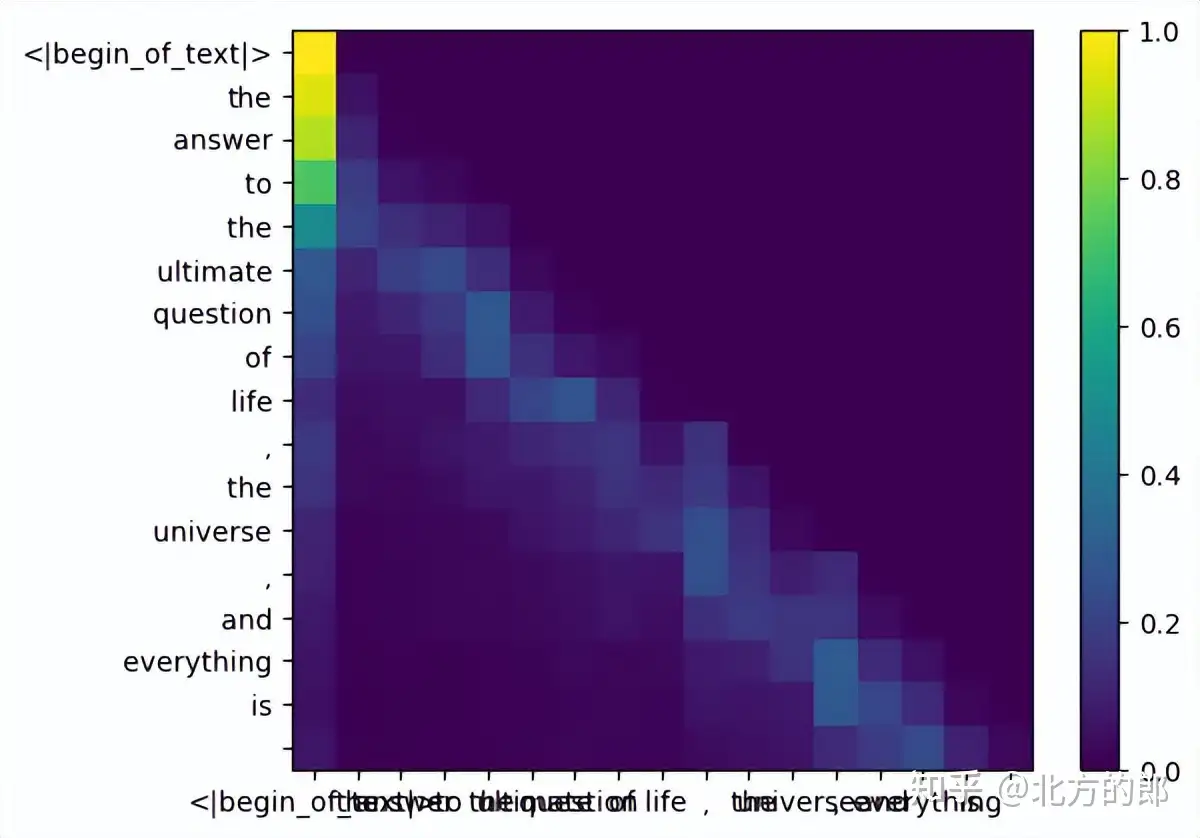
display\_qk\_heatmap(qk\_per\_token\_after\_masking)



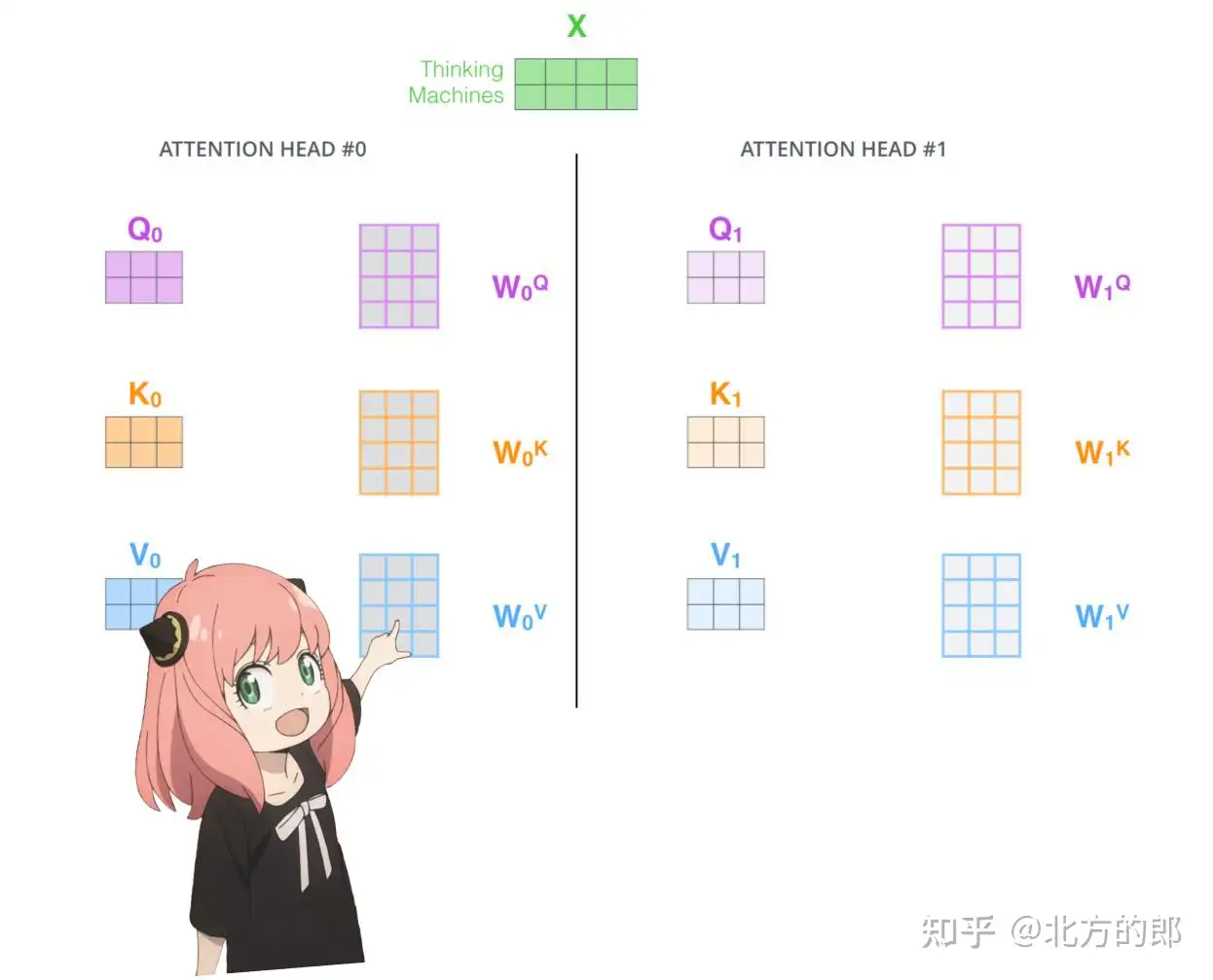


qk\_per\_token\_after\_masking\_after\_softmax = torch.nn.functional.softmax(qk\_per\_token\_after\_masking, dim=1).to(torch.bfloat16)

display\_qk\_heatmap(qk\_per\_token\_after\_masking\_after\_softmax)



## **值（values）（注意力几乎结束）**



这些分数（0-1）用于确定每个标记使用了多少值矩阵  
>就像键一样，值权重也每4个注意力头共享（以节省计算）  
>因此，值的形状下面的权重矩阵是[8x128x4096]

v\_layer0 = model["layers.0.attention.wv.weight"]

v\_layer0 = v\_layer0.view(n\_kv\_heads, v\_layer0.shape[0] // n\_kv\_heads, dim)

v\_layer0.shape

torch.Size([8, 128, 4096])

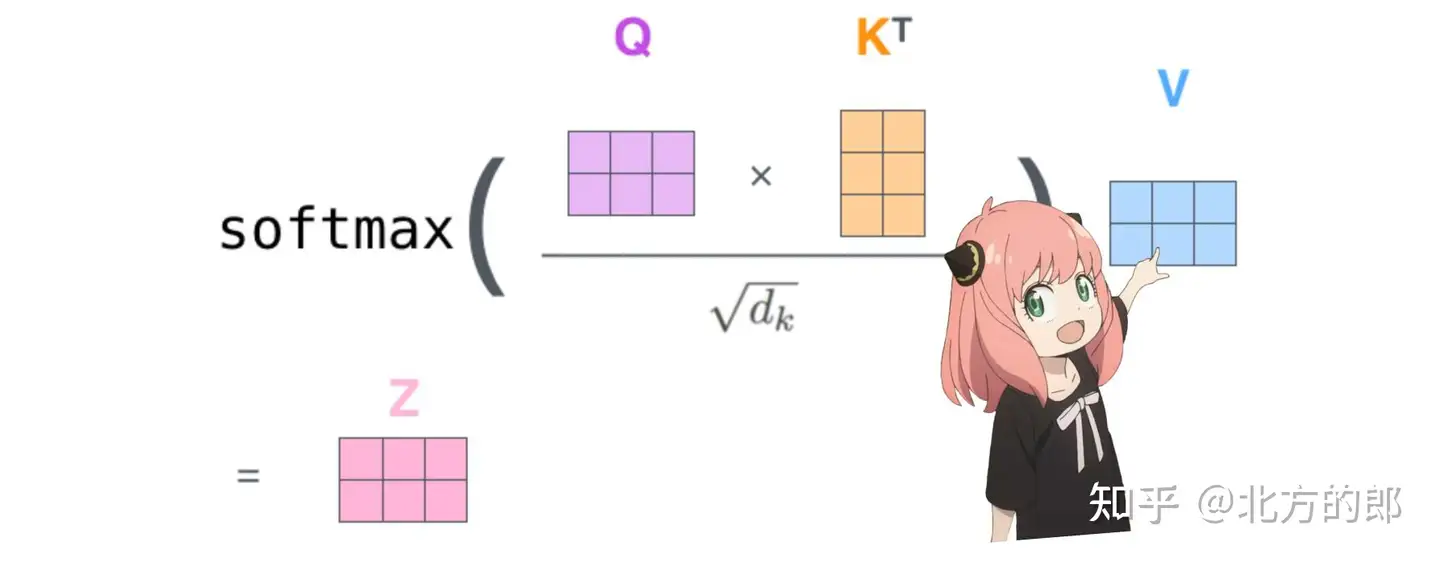
下面给出第一层第一头值权重矩阵

v\_layer0\_head0 = v\_layer0[0]

v\_layer0\_head0.shape

torch.Size([128, 4096])

## **值向量（value vectors）**



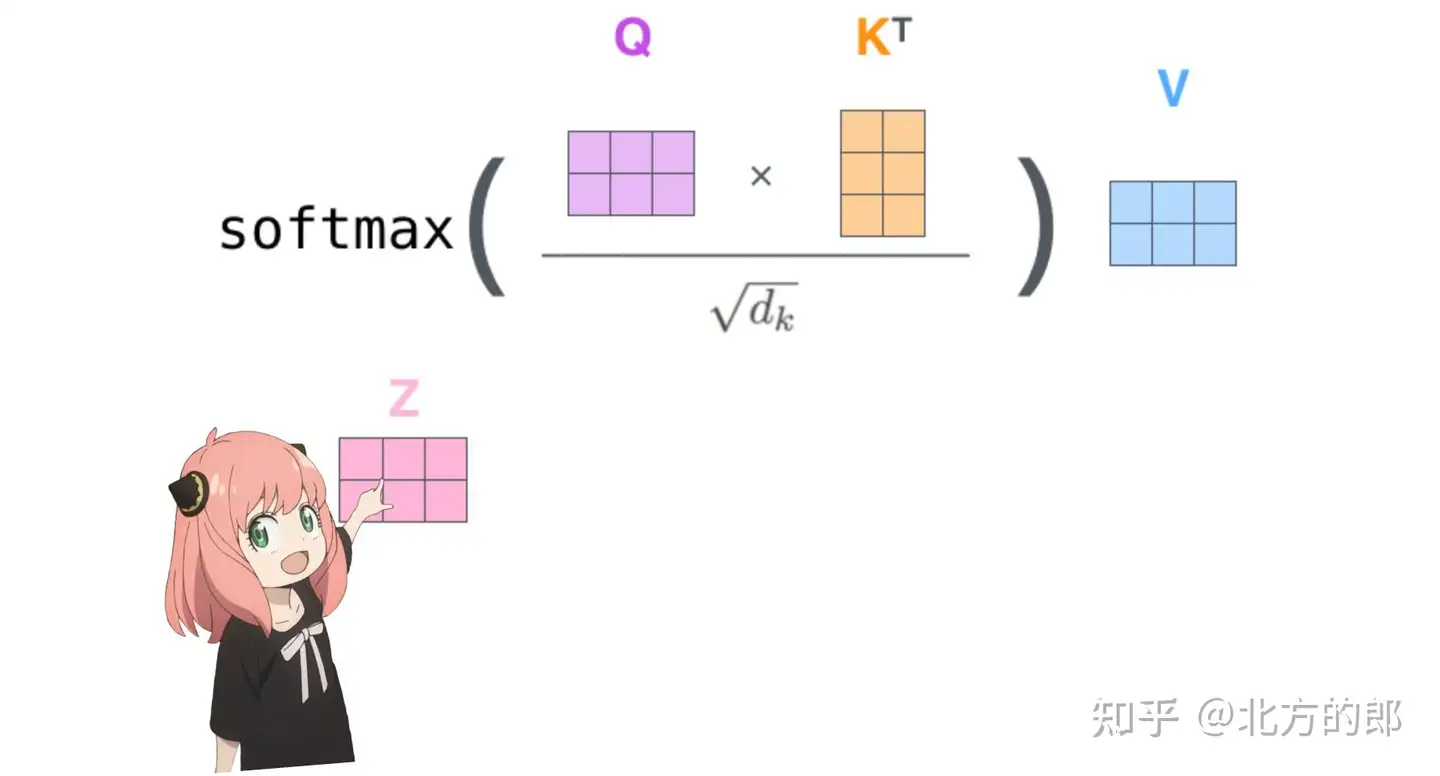
我们现在使用值权重来获取每个标记的注意力值，其大小为 [17x128]，其中 17 是提示中标记的数量，128 是每个标记的值向量的暗度

v\_per\_token = torch.matmul(token\_embeddings, v\_layer0\_head0.T)

v\_per\_token.shape

torch.Size([17, 128])

## **注意力**



与每个标记的值相乘后得到的注意力向量的形状为 [17\*128]

qkv\_attention = torch.matmul(qk\_per\_token\_after\_masking\_after\_softmax, v\_per\_token)

qkv\_attention.shape

torch.Size([17, 128])

## **多头注意力**

我们现在有了第一层和第一个头的注意力值，  
现在我将运行一个循环并执行与上面的单元完全相同的数学运算，但对于第一层中的每个头

qkv\_attention\_store = []

for head in range(n\_heads):

q\_layer0\_head = q\_layer0[head]

k\_layer0\_head = k\_layer0[head//4] # key weights are shared across 4 heads

v\_layer0\_head = v\_layer0[head//4] # value weights are shared across 4 heads

q\_per\_token = torch.matmul(token\_embeddings, q\_layer0\_head.T)

k\_per\_token = torch.matmul(token\_embeddings, k\_layer0\_head.T)

v\_per\_token = torch.matmul(token\_embeddings, v\_layer0\_head.T)

q\_per\_token\_split\_into\_pairs = q\_per\_token.float().view(q\_per\_token.shape[0], -1, 2)

q\_per\_token\_as\_complex\_numbers = torch.view\_as\_complex(q\_per\_token\_split\_into\_pairs)

q\_per\_token\_split\_into\_pairs\_rotated = torch.view\_as\_real(q\_per\_token\_as\_complex\_numbers \* freqs\_cis[:len(tokens)])

q\_per\_token\_rotated = q\_per\_token\_split\_into\_pairs\_rotated.view(q\_per\_token.shape)

k\_per\_token\_split\_into\_pairs = k\_per\_token.float().view(k\_per\_token.shape[0], -1, 2)

k\_per\_token\_as\_complex\_numbers = torch.view\_as\_complex(k\_per\_token\_split\_into\_pairs)

k\_per\_token\_split\_into\_pairs\_rotated = torch.view\_as\_real(k\_per\_token\_as\_complex\_numbers \* freqs\_cis[:len(tokens)])

k\_per\_token\_rotated = k\_per\_token\_split\_into\_pairs\_rotated.view(k\_per\_token.shape)

qk\_per\_token = torch.matmul(q\_per\_token\_rotated, k\_per\_token\_rotated.T)/(128)\*\*0.5

mask = torch.full((len(tokens), len(tokens)), float("-inf"), device=tokens.device)

mask = torch.triu(mask, diagonal=1)

qk\_per\_token\_after\_masking = qk\_per\_token + mask

qk\_per\_token\_after\_masking\_after\_softmax = torch.nn.functional.softmax(qk\_per\_token\_after\_masking, dim=1).to(torch.bfloat16)

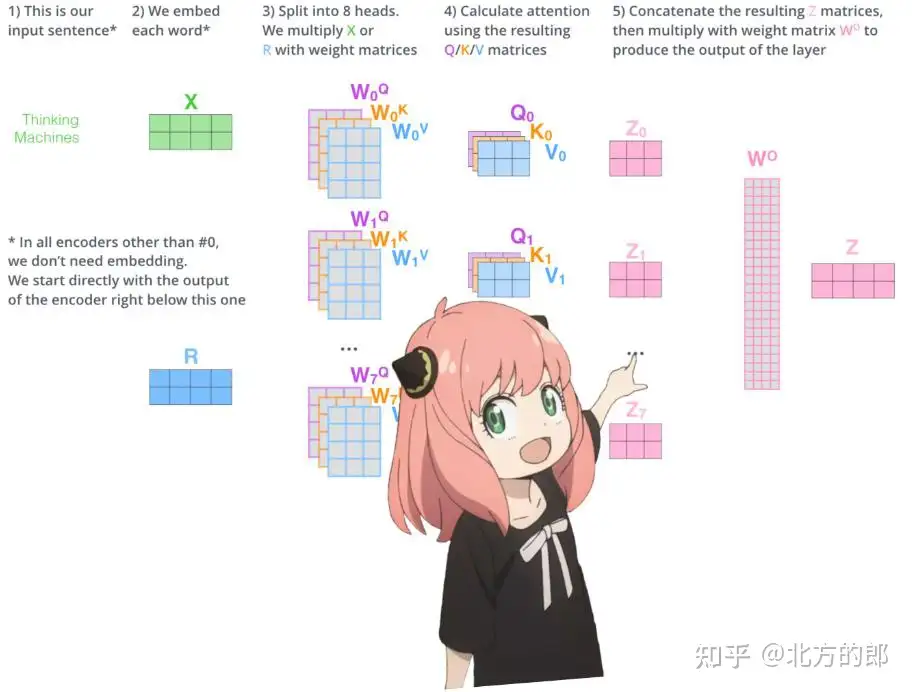
qkv\_attention = torch.matmul(qk\_per\_token\_after\_masking\_after\_softmax, v\_per\_token)

qkv\_attention = torch.matmul(qk\_per\_token\_after\_masking\_after\_softmax, v\_per\_token)

qkv\_attention\_store.append(qkv\_attention)

len(qkv\_attention\_store)

32



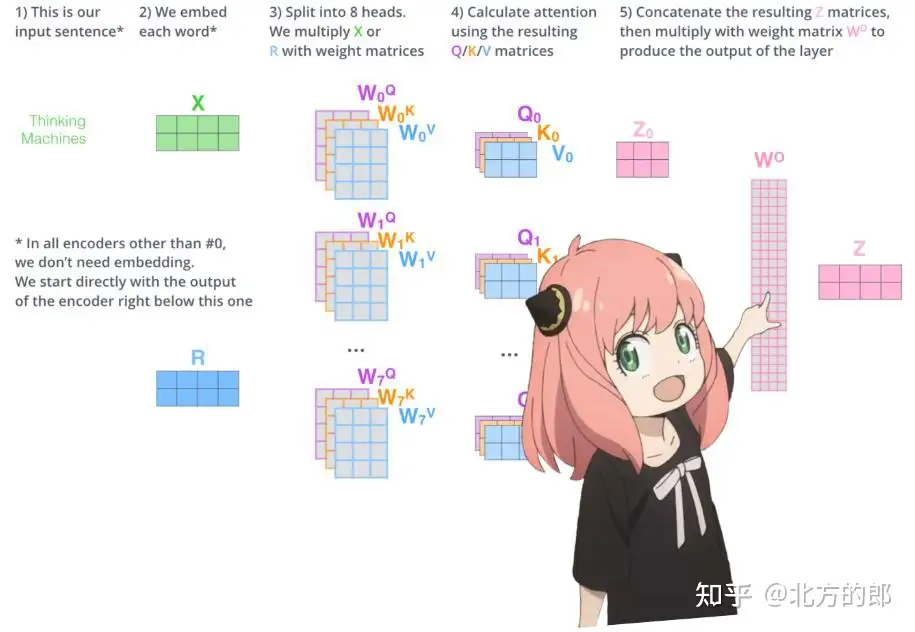
我们现在有了第一层所有 32 个头的 qkv\_attention 矩阵，接下来我将把所有注意力分数合并到一个大小为 [17x4096] 的大矩阵中，  
我们即将结束:)

stacked\_qkv\_attention = torch.cat(qkv\_attention\_store, dim=-1)

stacked\_qkv\_attention.shape

torch.Size([17, 4096])

## **权重矩阵，最后步骤之一**



对于第 0 层注意力要做的最后一件事是乘以

w\_layer0 = model["layers.0.attention.wo.weight"]

w\_layer0.shape

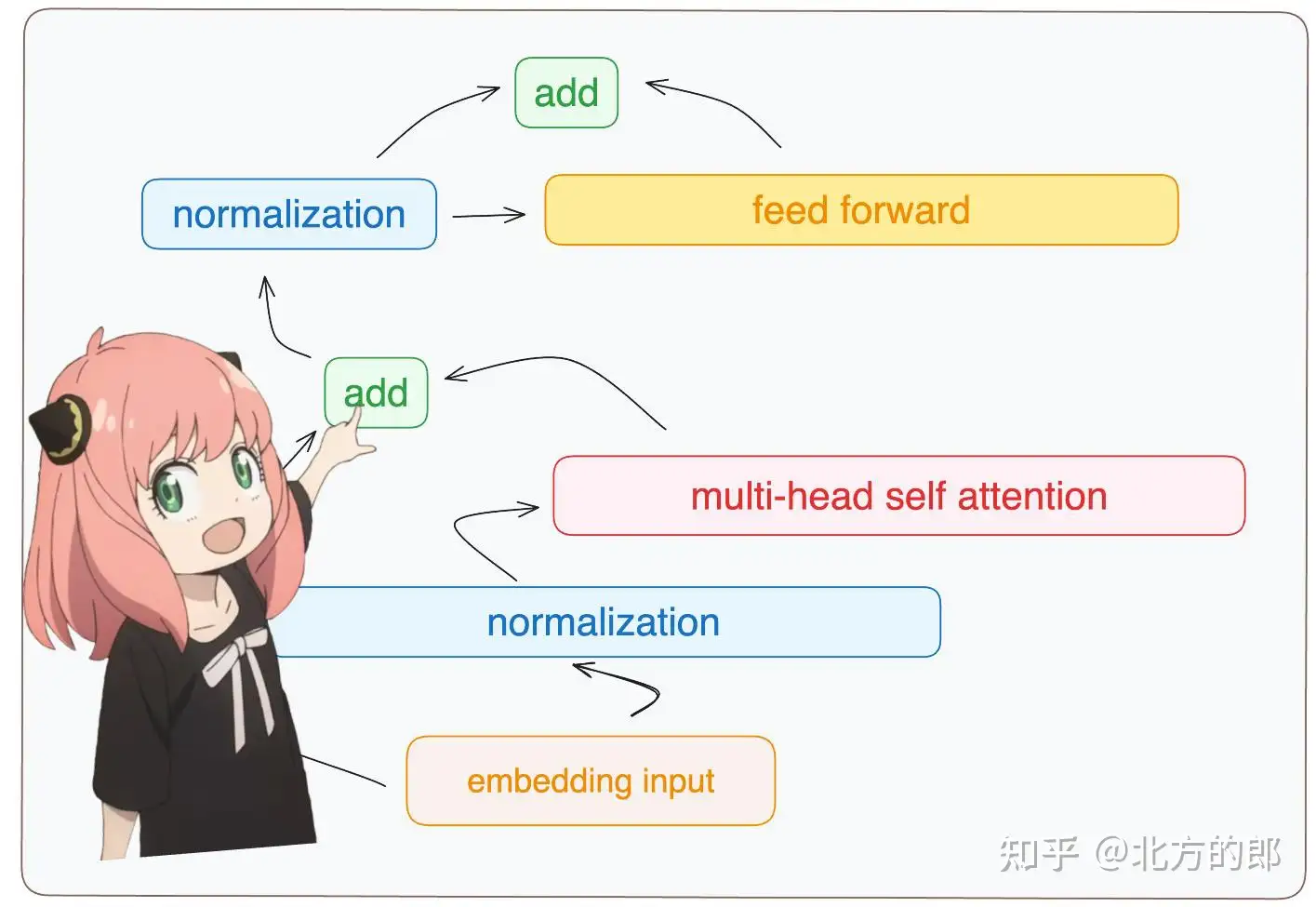
torch.Size([4096, 4096])

## **这是一个简单的线性层，所以我们只需 matmul**

embedding\_delta = torch.matmul(stacked\_qkv\_attention, w\_layer0.T)

embedding\_delta.shape

torch.Size([17, 4096])



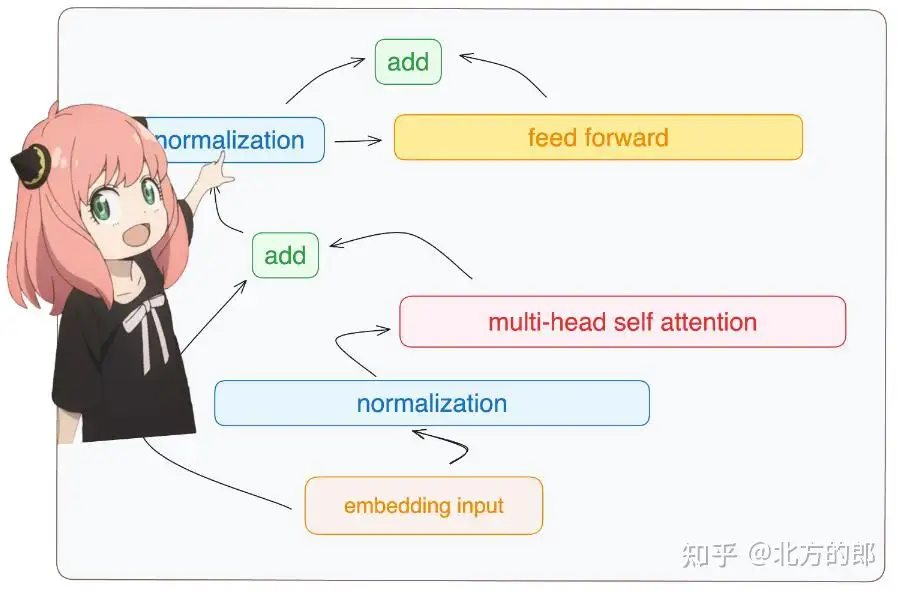
我们现在在注意力之后嵌入值发生了变化，这应该添加到原始令牌嵌入中

embedding\_after\_edit = token\_embeddings\_unnormalized + embedding\_delta

embedding\_after\_edit.shape

torch.Size([17, 4096])

我们进行标准化，然后通过嵌入增量运行**[前馈神经网络](https://zhida.zhihu.com/search?q=%E5%89%8D%E9%A6%88%E7%A5%9E%E7%BB%8F%E7%BD%91%E7%BB%9C&zhida_source=entity&is_preview=1" \t "https://zhuanlan.zhihu.com/p/_blank)**

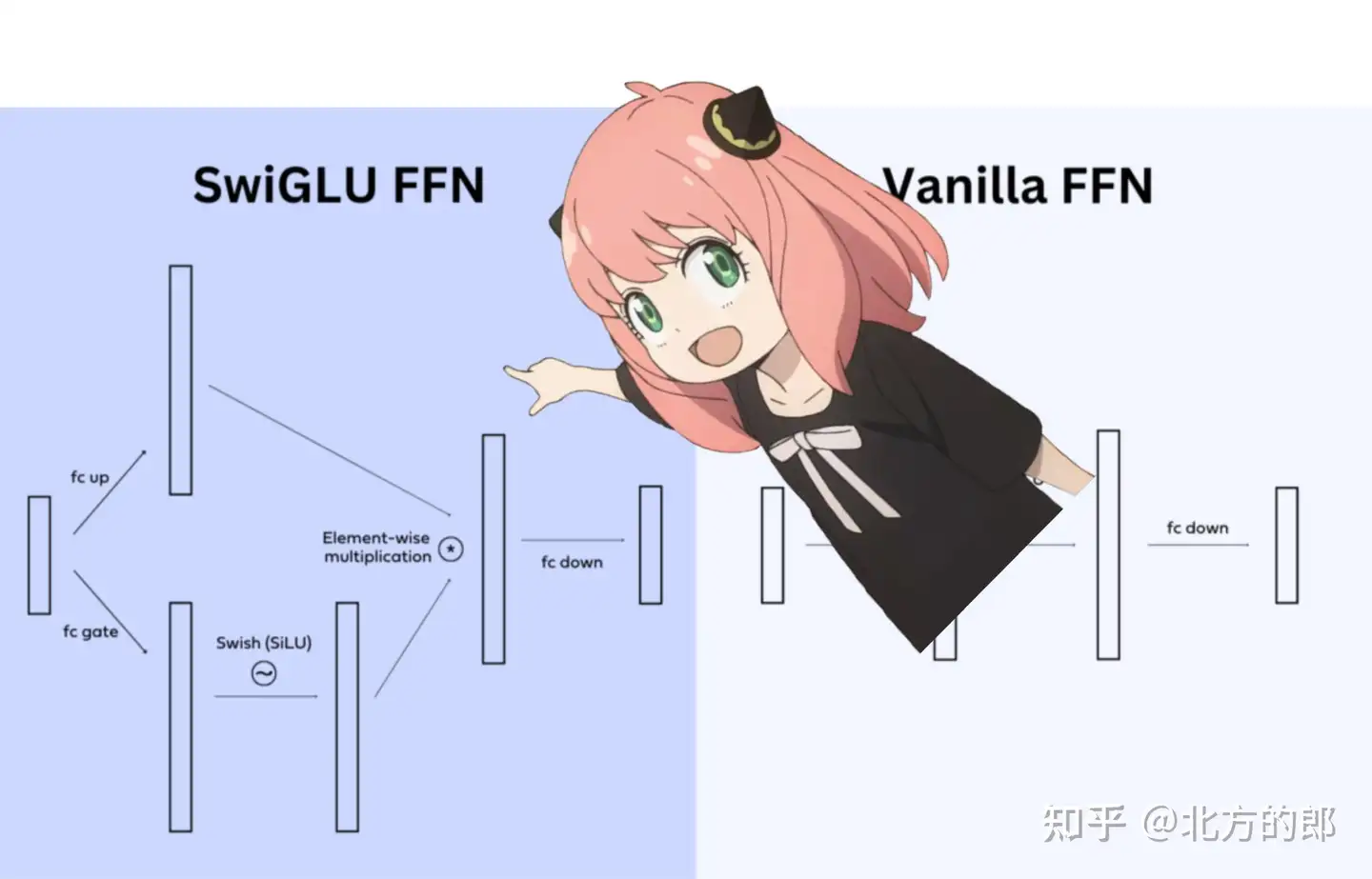


embedding\_after\_edit\_normalized = rms\_norm(embedding\_after\_edit, model["layers.0.ffn\_norm.weight"])

embedding\_after\_edit\_normalized.shape

torch.Size([17, 4096])

## **加载 ff 权重并实现前馈网络**



在 llama3 中，他们使用了 SwiGLU 前馈网络，这种网络架构非常擅长在模型需要时添加非线性。  
如今在 llms 中使用这种前馈网络架构是相当标准的

w1 = model["layers.0.feed\_forward.w1.weight"]

w2 = model["layers.0.feed\_forward.w2.weight"]

w3 = model["layers.0.feed\_forward.w3.weight"]

output\_after\_feedforward = torch.matmul(torch.functional.F.silu(torch.matmul(embedding\_after\_edit\_normalized, w1.T)) \* torch.matmul(embedding\_after\_edit\_normalized, w3.T), w2.T)

output\_after\_feedforward.shape

torch.Size([17, 4096])

## **我们终于在第一层之后为每个令牌有了新编辑的嵌入**

在我们完成之前还需要 31 层（一个 for 循环），  
您可以想象这个编辑后的嵌入包含有关第一层上提出的所有查询的信息，  
现在每一层都会对所提出的问题编码越来越复杂的查询，直到我们有一个嵌入知道我们需要的下一个令牌的所有信息。

layer\_0\_embedding = embedding\_after\_edit+output\_after\_feedforward

layer\_0\_embedding.shape

torch.Size([17, 4096])

## **天哪，一切都同时发生**



是的，就是这样。我们之前为每一层所做的一切都是一次性完成的。

## **祝阅读愉快:)**

final\_embedding = token\_embeddings\_unnormalized

for layer in range(n\_layers):

qkv\_attention\_store = []

layer\_embedding\_norm = rms\_norm(final\_embedding, model[f"layers.{layer}.attention\_norm.weight"])

q\_layer = model[f"layers.{layer}.attention.wq.weight"]

q\_layer = q\_layer.view(n\_heads, q\_layer.shape[0] // n\_heads, dim)

k\_layer = model[f"layers.{layer}.attention.wk.weight"]

k\_layer = k\_layer.view(n\_kv\_heads, k\_layer.shape[0] // n\_kv\_heads, dim)

v\_layer = model[f"layers.{layer}.attention.wv.weight"]

v\_layer = v\_layer.view(n\_kv\_heads, v\_layer.shape[0] // n\_kv\_heads, dim)

w\_layer = model[f"layers.{layer}.attention.wo.weight"]

for head in range(n\_heads):

q\_layer\_head = q\_layer[head]

k\_layer\_head = k\_layer[head//4]

v\_layer\_head = v\_layer[head//4]

q\_per\_token = torch.matmul(layer\_embedding\_norm, q\_layer\_head.T)

k\_per\_token = torch.matmul(layer\_embedding\_norm, k\_layer\_head.T)

v\_per\_token = torch.matmul(layer\_embedding\_norm, v\_layer\_head.T)

q\_per\_token\_split\_into\_pairs = q\_per\_token.float().view(q\_per\_token.shape[0], -1, 2)

q\_per\_token\_as\_complex\_numbers = torch.view\_as\_complex(q\_per\_token\_split\_into\_pairs)

q\_per\_token\_split\_into\_pairs\_rotated = torch.view\_as\_real(q\_per\_token\_as\_complex\_numbers \* freqs\_cis)

q\_per\_token\_rotated = q\_per\_token\_split\_into\_pairs\_rotated.view(q\_per\_token.shape)

k\_per\_token\_split\_into\_pairs = k\_per\_token.float().view(k\_per\_token.shape[0], -1, 2)

k\_per\_token\_as\_complex\_numbers = torch.view\_as\_complex(k\_per\_token\_split\_into\_pairs)

k\_per\_token\_split\_into\_pairs\_rotated = torch.view\_as\_real(k\_per\_token\_as\_complex\_numbers \* freqs\_cis)

k\_per\_token\_rotated = k\_per\_token\_split\_into\_pairs\_rotated.view(k\_per\_token.shape)

qk\_per\_token = torch.matmul(q\_per\_token\_rotated, k\_per\_token\_rotated.T)/(128)\*\*0.5

mask = torch.full((len(token\_embeddings\_unnormalized), len(token\_embeddings\_unnormalized)), float("-inf"))

mask = torch.triu(mask, diagonal=1)

qk\_per\_token\_after\_masking = qk\_per\_token + mask

qk\_per\_token\_after\_masking\_after\_softmax = torch.nn.functional.softmax(qk\_per\_token\_after\_masking, dim=1).to(torch.bfloat16)

qkv\_attention = torch.matmul(qk\_per\_token\_after\_masking\_after\_softmax, v\_per\_token)

qkv\_attention\_store.append(qkv\_attention)

stacked\_qkv\_attention = torch.cat(qkv\_attention\_store, dim=-1)

w\_layer = model[f"layers.{layer}.attention.wo.weight"]

embedding\_delta = torch.matmul(stacked\_qkv\_attention, w\_layer.T)

embedding\_after\_edit = final\_embedding + embedding\_delta

embedding\_after\_edit\_normalized = rms\_norm(embedding\_after\_edit, model[f"layers.{layer}.ffn\_norm.weight"])

w1 = model[f"layers.{layer}.feed\_forward.w1.weight"]

w2 = model[f"layers.{layer}.feed\_forward.w2.weight"]

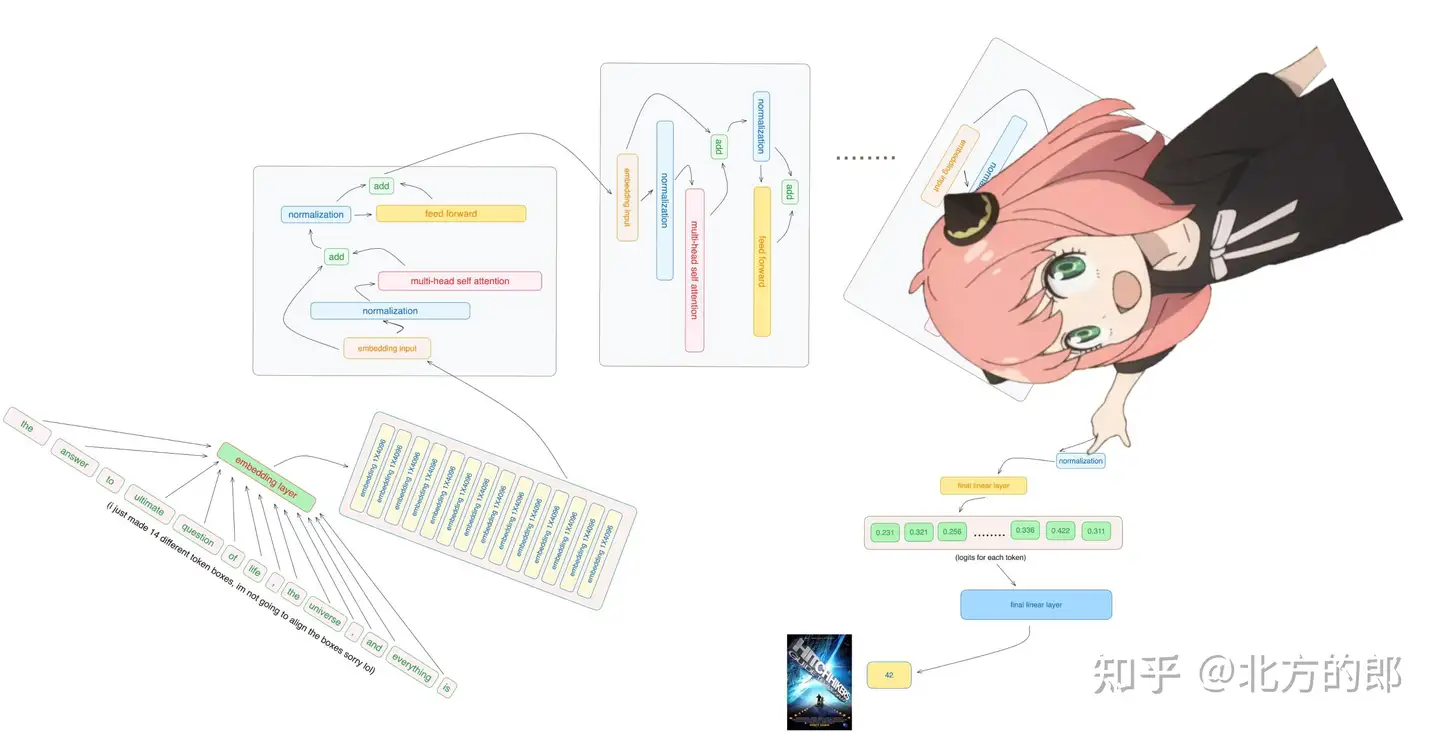
w3 = model[f"layers.{layer}.feed\_forward.w3.weight"]

output\_after\_feedforward = torch.matmul(torch.functional.F.silu(torch.matmul(embedding\_after\_edit\_normalized, w1.T)) \* torch.matmul(embedding\_after\_edit\_normalized, w3.T), w2.T)

final\_embedding = embedding\_after\_edit+output\_after\_feedforward

## **我们现在有了最终的嵌入，模型可以对下一个标记做出的最佳猜测**

嵌入的形状与常规令牌嵌入 [17x4096] 相同，其中 17 是令牌数量，4096 是嵌入暗淡

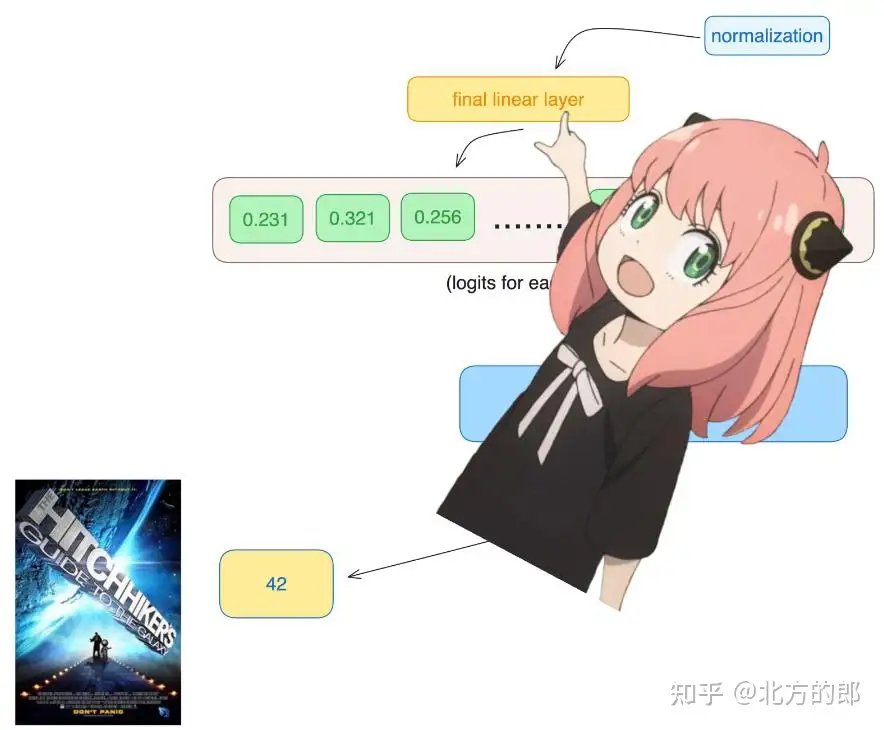


final\_embedding = rms\_norm(final\_embedding, model["norm.weight"])

final\_embedding.shape

torch.Size([17, 4096])

## **最后，让我们将嵌入解码到令牌值中**



我们将使用输出解码器将最终的嵌入转换为令牌

model["output.weight"].shape

torch.Size([128256, 4096])

## **我们使用最后一个标记的嵌入来预测下一个值**

希望在我们的例子中，42 :) 注意：42 是“生命、宇宙和一切的终极问题的答案”的答案，根据《[银河系漫游指南](https://zhida.zhihu.com/search?q=%E9%93%B6%E6%B2%B3%E7%B3%BB%E6%BC%AB%E6%B8%B8%E6%8C%87%E5%8D%97&zhida_source=entity&is_preview=1" \t "https://zhuanlan.zhihu.com/p/_blank)》一书，大多数现代 llms 都会回答这里有 42，这应该验证我们的整个代码！祝我好运 ：）

logits = torch.matmul(final\_embedding[-1], model["output.weight"].T)

logits.shape

torch.Size([128256])

## **模型预测令牌编号 2983 作为下一个令牌，这是 42 的令牌编号吗？**

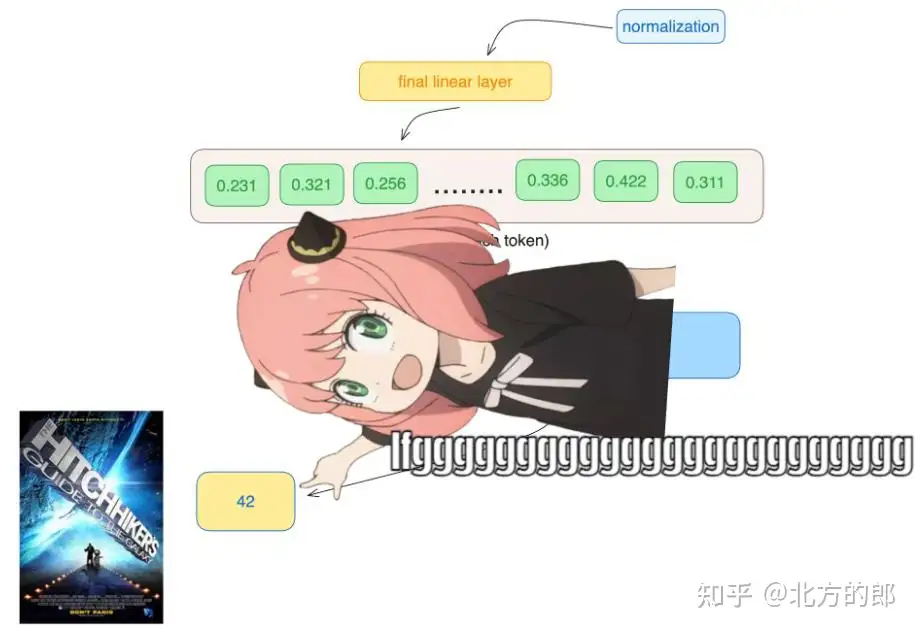
我正在向您宣传，这是代码的最后一个单元格，希望您玩得开心:)

next\_token = torch.argmax(logits, dim=-1)

next\_token

tensor(2983)

## **Let's go**



tokenizer.decode([next\_token.item()])

编辑于 2024-05-24 15:15・IP 属地黑龙江