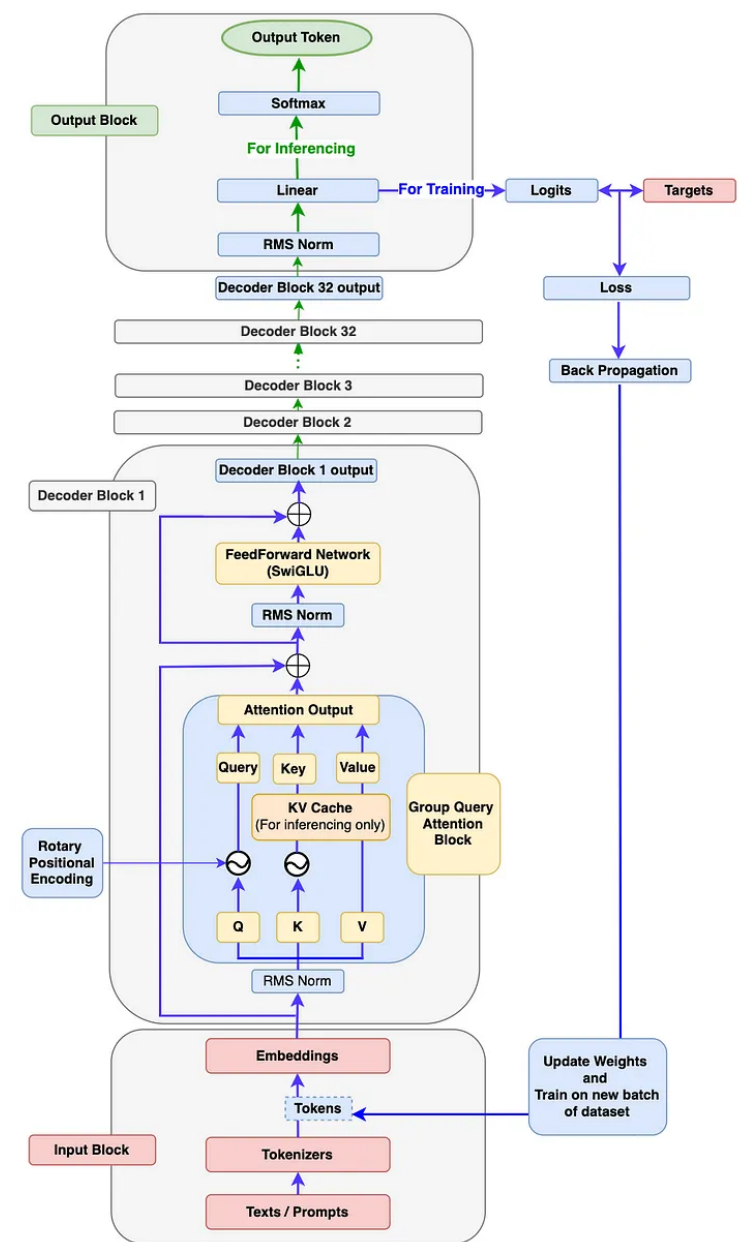


LLaMA3



📌 本节课旨在以LLaMA3为案例，讲解现代大语言模型通用最佳实践，截止2024年12月11日，市场上主流的开源大语言模型都采用类似架构，非常具有代表性。

基于Transformer-Decoder的改进

RMS-Norm

📌 LLaMA3采用RMS-Norm来进行归一化，从而取代了Layer-Norm。

RMS-Norm相较于Layer-Norm大大节省了计算开销，并且经过试验验证，与Layer-Norm取得相似的效果

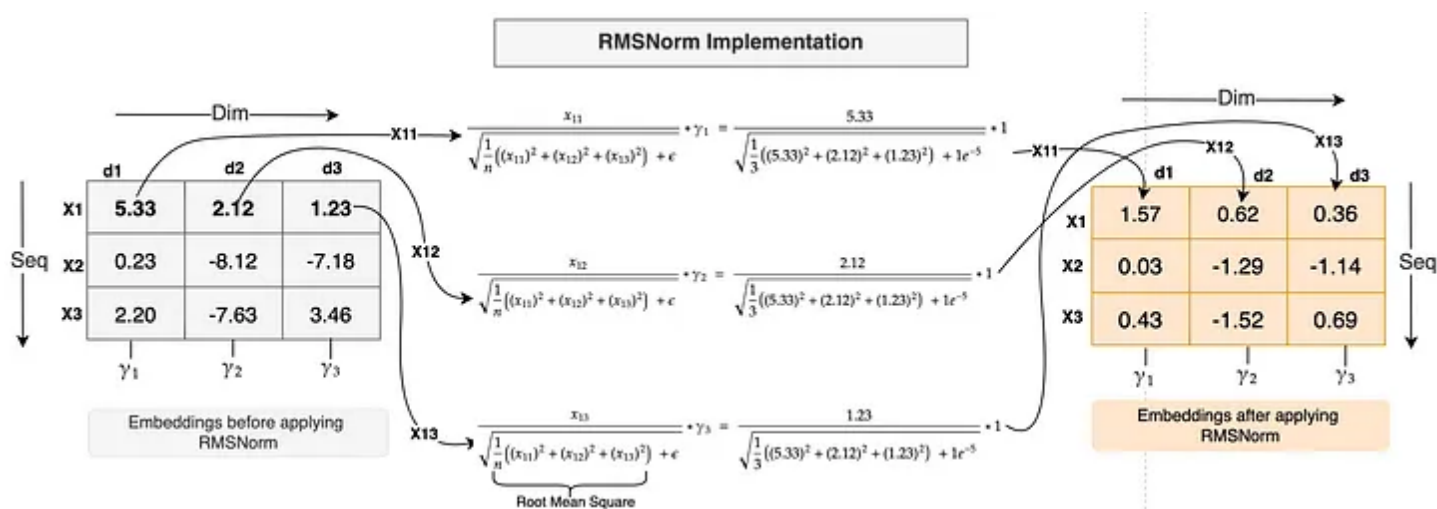
$$\bar{a}_i = \frac{a_i}{\text{RMS}(\mathbf{a})} g_i, \quad \text{where } \text{RMS}(\mathbf{a}) = \sqrt{\frac{1}{n} \sum_{i=1}^n a_i^2}.$$

RMS-Norm计算公式

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sigma} \cdot \gamma + \beta$$

$$\text{其中 } \mu = \frac{1}{d} \sum_{i=1}^d x_i, \quad \sigma = \sqrt{\frac{1}{d} \sum_{i=1}^d (x_i - \mu)^2}.$$

Layer-Norm计算公式



RMS-Norm流程

RoPE位置编码（Rotary Positional Encoding）



RoPE位置编码取代了绝对位置编码（Transformer）/可学习的位置编码（BERT）

核心思想

RoPE 将位置编码直接应用于自注意力机制中的**查询向量（query）**和**键向量（key）**，通过旋转变换的方式为它们引入相对位置信息。这种方法基于以下核心步骤：

输入向量分解为二维空间：

- 假设输入向量 $x \in \mathbb{R}^d$ 的维度为 d ，我们将它的每两个连续维度配对，形成 $\frac{d}{2}$ 个二维向量。
 - 例如， $x = [x_1, x_2, x_3, x_4]$ 会被分为两对： (x_1, x_2) 和 (x_3, x_4) 。

为每个二维向量引入旋转变换：

- 对于每对维度，定义一个旋转角度 θ 来编码位置信息。
- 具体旋转公式为：

- $$\text{RoPE}(x) = \begin{bmatrix} x_1 \cos(\theta) - x_2 \sin(\theta) \\ x_1 \sin(\theta) + x_2 \cos(\theta) \end{bmatrix}$$

- 对应到所有向量维度，利用每个位置的特定角度 θ 完成变换。

与位置相关的旋转角度：

- 旋转角度基于位置索引 p 和频率分量 ω 来定义： $\theta_{p,k} = p \cdot \omega_k$
 - p 是位置信息。
 - ω_k 是与维度 k 相关的频率分量。
 - $\omega_k = 10000^{-\frac{2k}{d}}$

将旋转编码引入自注意力机制：

- 对查询 q 和键 k 应用旋转编码，使得点积 $q \cdot k$ 自然包含相对位置信息：

$$q_p = \text{RoPE}(q), \quad k_p = \text{RoPE}(k)$$

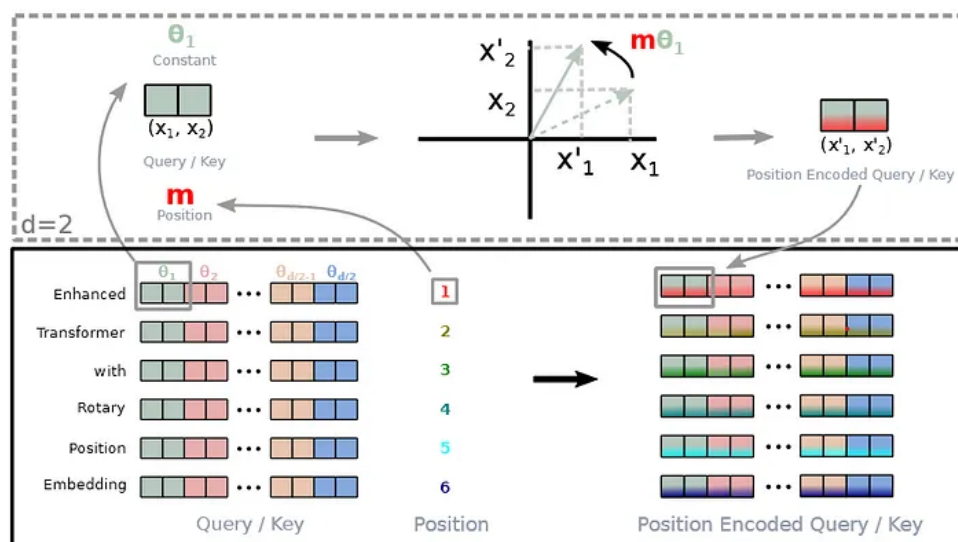


Figure 1: Implementation of Rotary Position Embedding(RoPE).

- 具体计算示例：

问题设置

1. 输入向量：

$$x = [x_1, x_2, x_3, x_4]$$

2. 位置信息：假设此向量位于序列中的第 $p = 2$ 个位置。

3. 频率分量：对于 RoPE，频率通常根据公式定义为：

$$\omega_k = 10000^{-\frac{2k}{d}}$$

对于 $d = 4$ 的情况：

- $k = 0: \omega_0 = 10000^{-\frac{2 \cdot 0}{4}} = 1$
- $k = 1: \omega_1 = 10000^{-\frac{2 \cdot 1}{4}} \approx 0.01$

4. 旋转角度：对应每个位置 p ，第 k 维的旋转角度为：

$$\theta_{p,k} = p \cdot \omega_k$$

- $\theta_{2,0} = 2 \cdot 1 = 2$
- $\theta_{2,1} = 2 \cdot 0.01 = 0.02$

计算步骤

1. 将向量分解为二维对：RoPE 对输入向量中的每两个连续维度（如 x_1, x_2 和 x_3, x_4 ）进行处理。

- 第 1 对：(x_1, x_2)
- 第 2 对：(x_3, x_4)

2. 对每对维度应用旋转变换：每对维度根据公式：

$$\text{RoPE}(x) = \begin{bmatrix} x_1 \cos(\theta) - x_2 \sin(\theta) \\ x_1 \sin(\theta) + x_2 \cos(\theta) \end{bmatrix}$$

对第 1 对 (x_1, x_2) 应用旋转：

$$\begin{aligned} y_1 &= x_1 \cos(\theta_{2,0}) - x_2 \sin(\theta_{2,0}) \\ y_2 &= x_1 \sin(\theta_{2,0}) + x_2 \cos(\theta_{2,0}) \end{aligned}$$

插入 $\theta_{2,0} = 2$ ：

$$\begin{aligned} y_1 &= x_1 \cos(2) - x_2 \sin(2) \\ y_2 &= x_1 \sin(2) + x_2 \cos(2) \end{aligned}$$

对第 2 对 (x_3, x_4) 应用旋转：

$$\begin{aligned} y_3 &= x_3 \cos(\theta_{2,1}) - x_4 \sin(\theta_{2,1}) \\ y_4 &= x_3 \sin(\theta_{2,1}) + x_4 \cos(\theta_{2,1}) \end{aligned}$$

插入 $\theta_{2,1} = 0.02$ ：

$$\begin{aligned} y_3 &= x_3 \cos(0.02) - x_4 \sin(0.02) \\ y_4 &= x_3 \sin(0.02) + x_4 \cos(0.02) \end{aligned}$$

3. 合并结果：将结果向量合并：

$$y = [y_1, y_2, y_3, y_4]$$

数值计算示例

假设输入向量为：

$$x = [1, 2, 3, 4]$$

1. 第 1 对 (x_1, x_2)：插入 $\theta_{2,0} = 2$ ，并使用近似值：

- $\cos(2) \approx -0.416, \sin(2) \approx 0.909$

$$y_1 = 1 \cdot (-0.416) - 2 \cdot 0.909 = -2.234$$

$$y_2 = 1 \cdot 0.909 + 2 \cdot (-0.416) = 0.077$$

2. 第 2 对 (x_3, x_4)：插入 $\theta_{2,1} = 0.02$ ，并使用近似值：

- $\cos(0.02) \approx 0.9998, \sin(0.02) \approx 0.02$

$$y_3 = 3 \cdot 0.9998 - 4 \cdot 0.02 = 2.879$$

$$y_4 = 3 \cdot 0.02 + 4 \cdot 0.9998 = 4.079$$

3. 最终结果向量：

$$y = [-2.234, 0.077, 2.879, 4.079]$$

为什么用RoPE替换绝对位置编码？

绝对位置编码的局限性：

- 位置依赖性：**绝对位置编码（如 sinusoidal 编码或 learnable 编码）将固定的位置索引编码到每个词的位置上，这使得模型对序列长度非常敏感，无法很好地推广到未见过的长序列。
- 缺乏相对位置信息：**绝对位置编码不能捕捉词之间的相对位置信息，而相对位置通常更重要（例如，在长序列中，关键关系常表现为词之间的相对距离）。

RoPE 的相对性：

- RoPE 的旋转编码使得位置信息直接内嵌到查询和键的点积中，并且自然而然地包含了相对位置关系。

- RoPE作用后的q与k在做点积时，其结果会包含一项：

$$q'_{p1} \cdot k'_{p2} = f(q, k) \cdot g(p_1 - p_2)$$

- $f(q, k)$ ：与原始查询和键相关的内容。
 - $g(p_1 - p_2)$ ：与位置差值相关的因子。
- 在自注意力机制中，这种相对性使得模型可以灵活地处理不同长度的序列，而无需重新调整编码。

RoPE 的高效性：

- RoPE 计算简单，只需要在查询和键上进行矩阵变换，计算开销比大多数复杂的相对位置编码方法（如 Transformer-XL 的相对位置编码）要低，一般通过复数乘法来做。

$$\mathbf{R}_{\Theta, m}^d \mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{d-2} \\ x_{d-1} \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_0 \\ \cos m\theta_0 \\ \cos m\theta_1 \\ \cos m\theta_1 \\ \vdots \\ \cos m\theta_{d/2-1} \\ \cos m\theta_{d/2-1} \end{pmatrix} + \begin{pmatrix} -x_1 \\ x_0 \\ -x_3 \\ x_2 \\ \vdots \\ -x_{d-1} \\ x_{d-2} \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_0 \\ \sin m\theta_0 \\ \sin m\theta_1 \\ \sin m\theta_1 \\ \vdots \\ \sin m\theta_{d/2-1} \\ \sin m\theta_{d/2-1} \end{pmatrix} \quad (15)$$

RoPE

SwiGLU

 用SwiGLU替换掉了FFN

SwiGLU 的公式如下：

$$\text{SwiGLU}(x) = (\text{swish}(xW_1 + b_1)) \odot (xW_2 + b_2)$$

 **Swish激活函数**

Swish激活函数如下

$$f(x) = x \cdot \sigma(x)$$

$$\sigma(x) = (1 + \exp(-x))^{-1}$$

SWISH激活函数是光滑且非单调，在x大于0时f(x)无上限，在x小于0时f(x)有下限，图如下

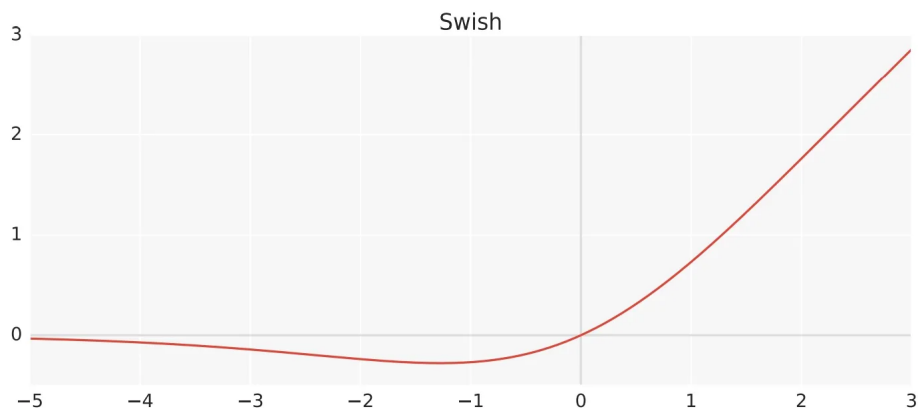


Figure 1: The Swish activation function.

Swish激活函数

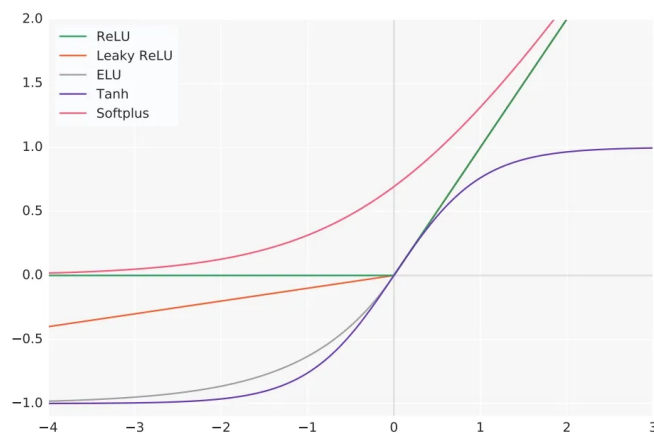


Figure 3: Common baseline activation functions. Best viewed in color.

其他激活函数

$$\begin{aligned}
 f'(x) &= \sigma(x) + x \cdot \sigma(x)(1 - \sigma(x)) \\
 &= \sigma(x) + x \cdot \sigma(x) - x \cdot \sigma^2(x) \\
 &= x \cdot \sigma(x) + \sigma(x)(1 - x \cdot \sigma(x)) \\
 &= f(x) + \sigma(x)(1 - f(x))
 \end{aligned}$$

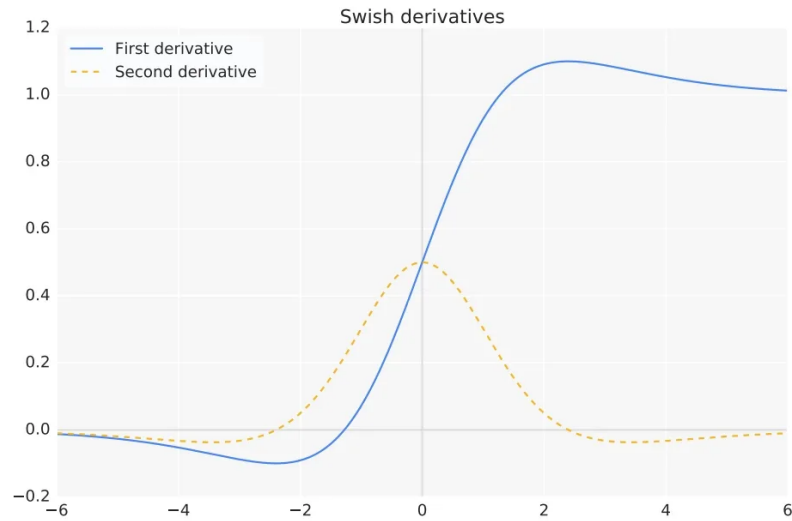


Figure 2: First and second derivatives of Swish.

Swish一阶导数和二阶导数


GLU门控单元 (Gated Linear Unit)

GLU 的公式为：

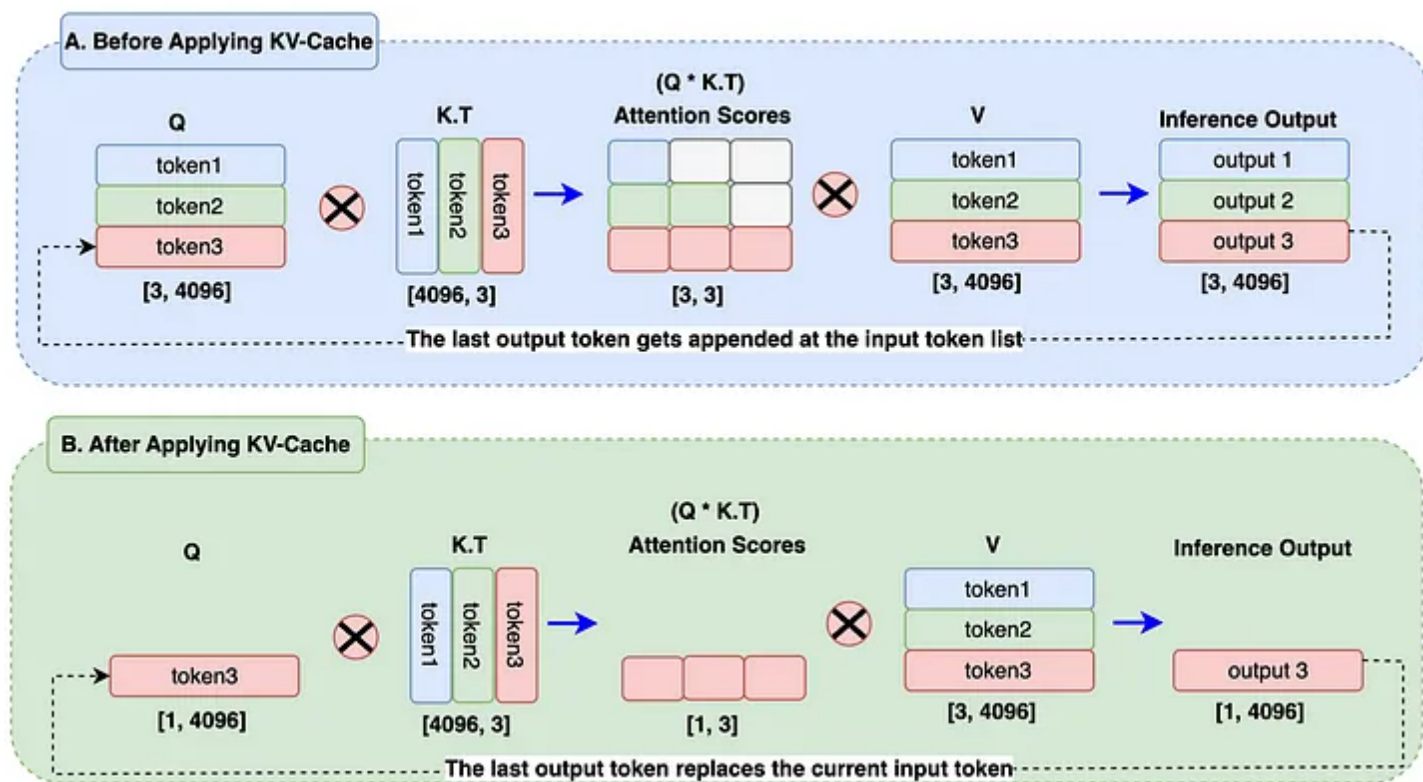
$$\text{GLU}(x) = (xW_1 + b_1) \odot \sigma(xW_2 + b_2)$$

- GLU 使用 Sigmoid 作为激活函数。
- SwiGLU 使用 Swish 替代 Sigmoid，提供更平滑的梯度和增强的表达能力。
- SwiGLU在性能上优于ReLU和GELU

KV 缓存：

 **什么是 KV-Cache?** 在 Llama 3 架构中，在推理时引入了 KV-Cache 的概念，以 Key 和 Value 缓存的形式存储先前生成的 token。这些缓存将用于计算自注意力以生成下一个 token。只有 key 和 value token 会被缓存，而查询 token 不会被缓存，因此称为 KV Cache。

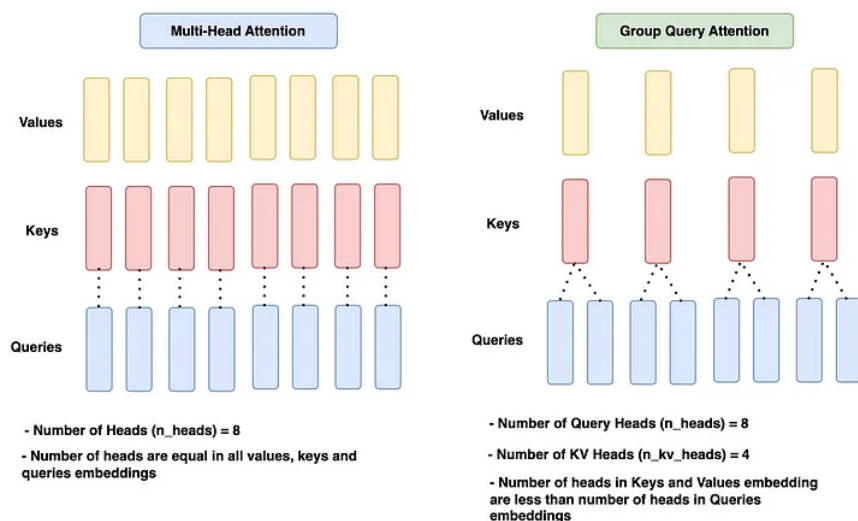
为什么我们需要KV Cache? 让我们看下面的图来阐明。



- 在图中的 A 块中，在生成 output3 token 时，之前的输出 token (output1, output2) 还在计算，这是完全没有必要的。这导致在计算注意力时需要进行额外的矩阵乘法，因此计算资源增加了很多。
- 在图中的块 B 中，输出标记替换了查询嵌入中的输入标记。KV Cache 存储了先前生成的标记。在注意力得分计算期间，我们只需使用查询中的 1 个标记并使用键和值缓存中的先前标记。它将从块 A 到块 B 的矩阵乘法从 3x3 减少到 1x3，减少了近 66%。在现实世界中，由于序列长度和批量大小巨大，这将有助于显著降低计算能力。最后，将始终只生成一个最新的输出标记。这是引入 KV-Cache 的主要原因。

分组注意力查询

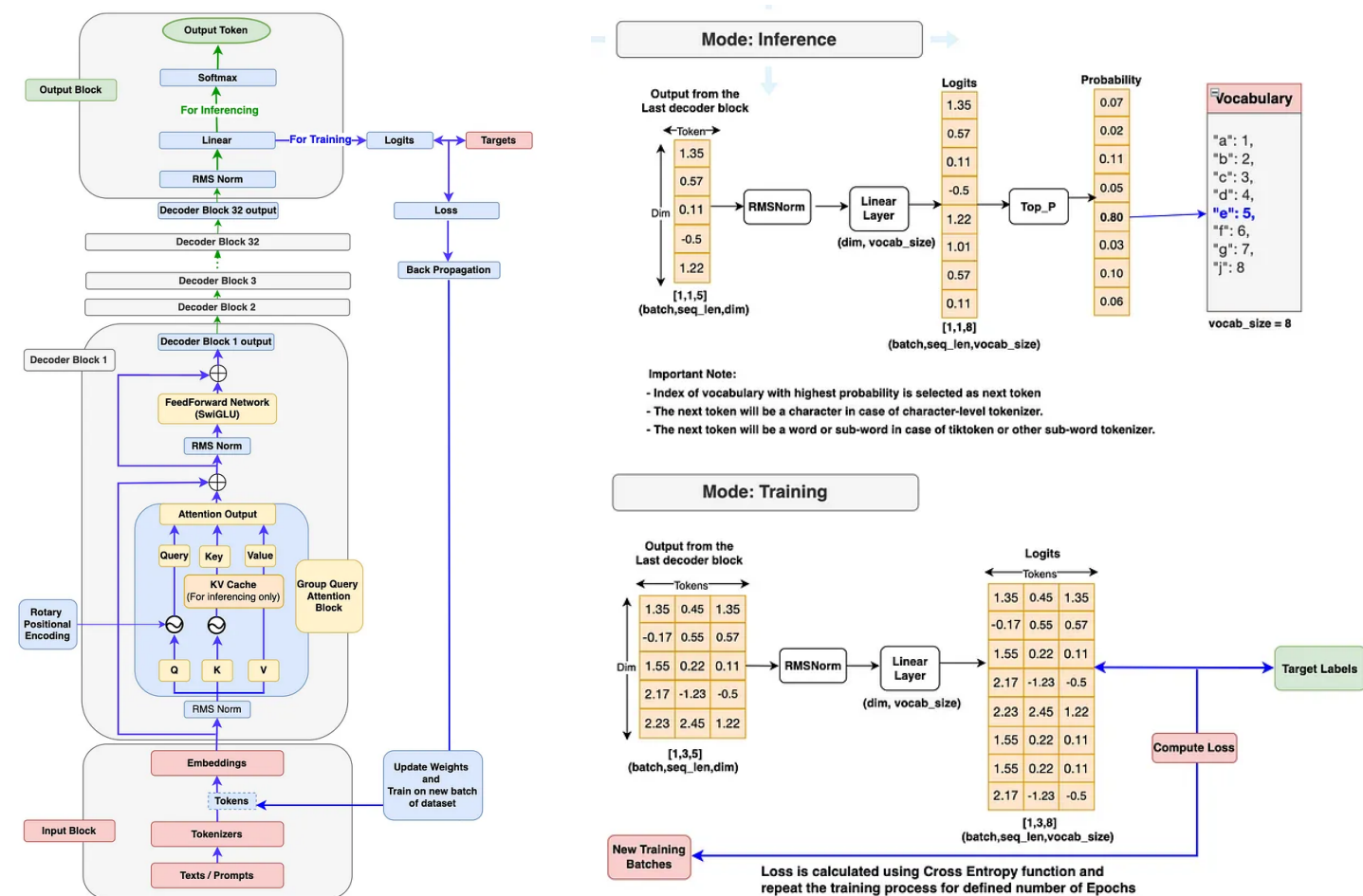
- 组查询注意力机制与之前的模型（例如 Llama 1）中使用的多头注意力机制相同，唯一的区别在于查询使用单独的 head，键/值使用单独的 head。通常，分配给查询的 head 的数量是键和值 head 数量的 n 倍。让我们看一下图表以进一步加深理解。



📌 在给定的图表中，多头注意力在所有查询、键和值中具有相同数量的头，即 $n_heads = 8$ 。组查询注意块有 8 个用于查询的头 (n_heads) 和 4 个用于键和值的头 (n_kv_heads)，比查询头少 2 倍。

既然 **MultiHead Attention** 已经这么好了，为什么还需要 **Group query Attention**? 要回答这个问题，我们需要先回顾一下 KV Cache。KV Cache 有助于大大减少计算资源。然而，随着 KV Cache 存储越来越多的先前 token，内存资源将显著增加。无论从模型性能角度还是从财务角度来看，这都不是一件好事。因此，引入了 **Group query Attention**。减少 K 和 V 的 head 数量会减少要存储的参数数量，因此使用的内存更少。各种测试结果证明，采用这种方法，模型准确率仍保持在同一范围内。

全貌



从0实现LLaMA3

📌 我们从0实现一个LLaMA3的模型结构并且加载LLaMA3-8B-Instruct的参数，当然，tokenizer会使用预训练好的，这是基于BBPE训练出来的。

```

1 from pathlib import Path
2 import tiktoken
3 from tiktoken.load import load_tiktoken_bpe
4 import torch
5 import json
6 import matplotlib.pyplot as plt
7
8 # 加载分词器模型路径
9 tokenizer_path = "Meta-Llama-3-8B-Instruct/tokenizer.model"
10 special_tokens = [
11     "<|begin_of_text|>",
12     "<|end_of_text|>",
13     "<|reserved_special_token_0|>",
14     "<|reserved_special_token_1|>",
15     "<|reserved_special_token_2|>",
16     "<|reserved_special_token_3|>",

```

```

17         "<|start_header_id|>",
18         "<|end_header_id|>",
19         "<|reserved_special_token_4|>",
20         "<|eot_id|>", # end of turn
21     ] + [f"<|reserved_special_token_{i}|>" for i in range(5, 256 - 5)]
22 mergeable_ranks = load_tiktoken_bpe(tokenizer_path)
23 tokenizer = tiktoken.Encoding(
24     name=Path(tokenizer_path).name,
25     pat_str=r"([i:'s|'t|'re|'ve|'m|'ll|'d]|^[^r\n\\p{L}\\p{N}}?\\p{L}+|\\p{N}
26 {1,3}| ?[^\s\\p{L}\\p{N}]+[\\r\\n]*|\\s*[\\r\\n]+|\\s+(?!\\S)|\\s+)",
27     mergeable_ranks=mergeable_ranks,
28     special_tokens={token: len(mergeable_ranks) + i for i, token in
29 enumerate(special_tokens)}),
30 )
31 # 测试分词器编码和解码功能
32 tokenizer.decode(tokenizer.encode("hello world!"))

```

输出：

hello world!

取读模型文件

通常，读取模型文件，往往取决于模型类的编写方式以及其中的变量名。

但由于要从零实现 Llama3，直接将模型全重加载到model变量

```
1 # 加载模型权重model = torch.load("Meta-Llama-3-8B-Instruct/consolidated.00.pth")
2 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"  
]
```

```
1  
2 # 获取模型配置参数  
3 with open("Meta-Llama-3-8B-Instruct/params.json", "r") as f:  
4     config = json.load(f)  
5 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  
}
```

使用这些配置推理模型的细节

1. 模型有 32 个 Transformer 层
2. 每个多头注意力块有 32 个头
3. 词汇表大小等

```
1 # 从配置文件中提取模型参数
2 dim = config["dim"]
3 n_layers = config["n_layers"]
4 n_heads = config["n_heads"]
5 n_kv_heads = config["n_kv_heads"]
6 vocab_size = config["vocab_size"]
7 multiple_of = config["multiple_of"]
8 ffn_dim_multiplier = config["ffn_dim_multiplier"]
9 norm_eps = config["norm_eps"]
10 rope_theta = torch.tensor(config["rope_theta"])
```

这里使用 tiktoken (OpenAI 的库) 作为分词器

```
1 prompt = "the answer to the ultimate question of life, the universe, and
  everything is "
2
3 # 编码为 token
4 tokens = [128000] + tokenizer.encode(prompt)
5 print(tokens)
6 tokens = torch.tensor(tokens)
7
8 # 将每个 token 解码为对应的文本
9 prompt_split_as_tokens = [tokenizer.decode([token.item()]) for token in tokens]
10 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', ' ']
```

将 token 转换为 embedding

这里使用内置的神经网络模块

无论如何, `[17x1]` token 现在是 `[17x4096]`, 即每个 token 的长度为 4096 的 embeddings

注意: 跟踪 shapes, 这样一切将变得理解更容易

```
1
2 # 加载嵌入层并复制权重
3 embedding_layer = torch.nn.Embedding(vocab_size, dim)
4 embedding_layer.weight.data.copy_(model["tok_embeddings.weight"])
5
6 # 获取未归一化的 token 嵌入
7 token_embeddings_unnormalized = embedding_layer(tokens).to(torch.bfloat16)
8 token_embeddings_unnormalized.shape
```

`torch.Size([17, 4096])`

接下来使用 RMS 归一化嵌入

请注意, 经过此步骤后 shapes 不变, 只是值被归一化

需要注意的是, 需要一个 `norm_eps` (来自配置) 以避免不小心将 RMS 设置为 0 并导致除以 0 的情况

$$\bar{a}_i = \frac{a_i}{\text{RMS}(\mathbf{a})} g_i, \quad \text{where } \text{RMS}(\mathbf{a}) = \sqrt{\frac{1}{n} \sum_{i=1}^n a_i^2}.$$

RMS-Norm计算公式

```
1 # rms 归一化函数
2
3 def rms_norm(tensor, norm_weights):
4     return (tensor * torch.rsqrt(tensor.pow(2).mean(-1, keepdim=True) +
5         norm_eps)) * norm_weights
```

构建第一个 Transformer 层

归一化

从模型字典中访问 `layer.0` (这是第一层)

归一化后 shapes 仍然是 `[17x4096]`，与嵌入相同但已归一化

```
1 # 归一化token嵌入
2 token_embeddings = rms_norm(token_embeddings_unnormalized,
    model["layers.0.attention_norm.weight"])
3 token_embeddings.shape
```

`torch.Size([17, 4096])`

从头实现注意力机制

加载第一个 Transformer 层的注意力头

- 当我们从模型中加载 `query`，`key`，`value` 和 `output` 权重矩阵时，注意到 shapes 分别为 `[4096x4096]`，`[1024x4096]`，`[1024x4096]`，`[4096x4096]`
- 乍一看这有些奇怪，因为在理想情况下我们希望每个头单独拥有各自的 q, k, v 和 o
- 这里作者将其捆绑在一起，为什么会这样呢？因为这样有助于并行化注意力头的计算

```
1 # 打印第一个层的注意力权重 shapes
2 print(
3     model["layers.0.attention.wq.weight"].shape,
4     model["layers.0.attention.wk.weight"].shape,
5     model["layers.0.attention.wv.weight"].shape,
6     model["layers.0.attention.wo.weight"].shape
7 )
```

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

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

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

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

展开 query

在下一步中，将展开多个注意力头的 query，得到的 shapes 为 `[32x128x4096]`

这里的 32 是 Llama3 的注意力头数量，128 是 query 向量的大小，4096 是 token 嵌入的大小

```
1 # reshape query 权重为[头数, 头维度, 嵌入维度]
2
3 q_layer0 = model["layers.0.attention.wq.weight"]
4 head_dim = q_layer0.shape[0] // n_heads
5 q_layer0 = q_layer0.view(n_heads, head_dim, dim)
6 q_layer0.shape
```

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

实现第一层的第一个头

这里查询了第一个层的第一个头的 `query` 权重矩阵，其大小为 `[128x4096]`

```
1 q_layer0_head0 = q_layer0[0]
2 q_layer0_head0.shape
```

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

现在将 `query` 权重与 `token` 嵌入相乘，以获得每个 `token` 的 `query`

这里可以看到得到的 `shape` 是 `[17x128]`，这是因为有 17 个 `token`，每个 `token` 有一个长度为 128 的 `query`

```
1 q_per_token = torch.matmul(token_embeddings, q_layer0_head0.T)
2 q_per_token.shape
```

```
torch.Size([17, 128])
```

位置编码

当前，每个 `token` 都有一个 `query` 向量，但如果你想一想 -- 其实各个 `query` 向量并不知道它们在 `prompt` 中的位置。

```
query: "the answer to the ultimate question of life, the
universe, and everything is "
```

在我示例 `prompt` 中，使用了三次 `"the"`，需要根据它们在 `prompt` 中的位置为每个 `"the"` `token` 生成不同的 `query` 向量（每个长度为128）。可以使用 RoPE（旋转位置编

码) 来实现这一点。

```
1 q_per_token_split_into_pairs = q_per_token.float().view(q_per_token.shape[0],  
-1, 2)  
2 q_per_token_split_into_pairs.shape
```

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



- 这里为 prompt 中每个位置生成了旋转位置编码。可以看到，这些编码是正弦和余弦函数的组合。
- 在上的步骤里, 将 `query` 向量分成对, 并对每对应用旋转角度移位!
- 现在有一个大小为 `[17x64x2]` 的向量, 这是针对 prompt 中的每个 token 将 128 个长度的 query 分为 64 对! 这 64 对中的每一对都将旋转 `m * θ` , 其中 `m` 是旋转查询的 token 的位置!

为每个二维向量引入旋转变换:

- 对于每对维度, 定义一个旋转角度 θ 来编码位置信息。
- 具体旋转公式为:
 - $$\text{RoPE}(x) = \begin{bmatrix} x_1 \cos(\theta) - x_2 \sin(\theta) \\ x_1 \sin(\theta) + x_2 \cos(\theta) \end{bmatrix}$$
 - 对应到所有向量维度, 利用每个位置的特定角度 θ 完成变换。

与位置相关的旋转角度:

- 旋转角度基于位置索引 p 和频率分量 ω 来定义: $\theta_{p,k} = p \cdot \omega_k$
 - p 是位置信息。
 - ω_k 是与维度 k 相关的频率分量。
 - $$\omega_k = 10000^{-\frac{2k}{d}}$$

```
1 zero_to_one_split_into_64_parts = torch.tensor(range(64))/64  
2 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])
```

```
1 freqs = 1.0 / (rope_theta ** zero_to_one_split_into_64_parts)  
2 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])
```

```
1 freqs_for_each_token = torch.outer(torch.arange(17), freqs)  
2 freqs_cis = torch.polar(torch.ones_like(freqs_for_each_token),  
freqs_for_each_token)
```

```
1 q_per_token_as_complex_numbers =  
torch.view_as_complex(q_per_token_split_into_pairs)  
2 q_per_token_as_complex_numbers.shape
```

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

```
1 q_per_token_as_complex_numbers_rotated = q_per_token_as_complex_numbers *
  freqs_cis
2 q_per_token_as_complex_numbers_rotated.shape
```


| torch.Size([17, 64])

得到旋转向量后

可以通过再次将复数看作实数来返回成对的 query

```
1 q_per_token_split_into_pairs_rotated =
  torch.view_as_real(q_per_token_as_complex_numbers_rotated)
2 q_per_token_split_into_pairs_rotated.shape
```

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

 旋转对现在已合并，现在有了一个新的 query 向量（旋转 query 向量），其 shape 为 [17x128]，其中 17 是 token 的数量，128 是 query 向量的维度

```
1 q_per_token_rotated =
  q_per_token_split_into_pairs_rotated.view(q_per_token.shape)
2 q_per_token_rotated.shape
```

| torch.Size([17, 128])

keys（几乎与 query 一模一样）

- keys 生成的 key 向量的维度也是 128
- **keys 的权重只有 query 的 1/4，因为 keys 的权重在 4 个头之间共享，以减少计算量**
- keys 也像 query 一样被旋转以添加位置信息，其原因相同

```
1 k_layer0 = model["layers.0.attention.wk.weight"]
2 k_layer0 = k_layer0.view(n_kv_heads, k_layer0.shape[0] // n_kv_heads, dim)
3 k_layer0.shape
```

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

```
1 k_layer0_head0 = k_layer0[0]
2 k_layer0_head0.shape
```

torch.Size([128, 4096])

```
1 k_per_token = torch.matmul(token_embeddings, k_layer0_head0.T)
2 k_per_token.shape
```

torch.Size([17, 128])

```
1 k_per_token_split_into_pairs = k_per_token.float().view(k_per_token.shape[0],
-1, 2)
2 k_per_token_split_into_pairs.shape
```

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

```
1 k_per_token_as_complex_numbers =
  torch.view_as_complex(k_per_token_split_into_pairs)
2 k_per_token_as_complex_numbers.shape
```

torch.Size([17, 64])

```
1 k_per_token_split_into_pairs_rotated =
  torch.view_as_real(k_per_token_as_complex_numbers * freqs_cis)
2 k_per_token_split_into_pairs_rotated.shape
```

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

```
1 k_per_token_rotated =
  k_per_token_split_into_pairs_rotated.view(k_per_token.shape)
2 k_per_token_rotated.shape
```

torch.Size([17, 128])

现在，已经有了每个 token 的旋转后的 query 和 key

每个 query 和 key 的 shape 都是 `[17x128]`

接下来，将 query 和 key 的矩阵相乘

- 这样做会得到每一个 token 相互映射的分数
- 这个分数描述了每个 token 的 query 与每个 token 的 key 的相关度。这就是自注意力！)
- 注意力得分矩阵（qk_per_token）的 shape 是 `[17x17]`，其中 17 是 prompt 中的 token 数量

```
1 qk_per_token = torch.matmul(q_per_token_rotated,  
    k_per_token_rotated.T)/(head_dim)**0.5  
2 qk_per_token.shape
```

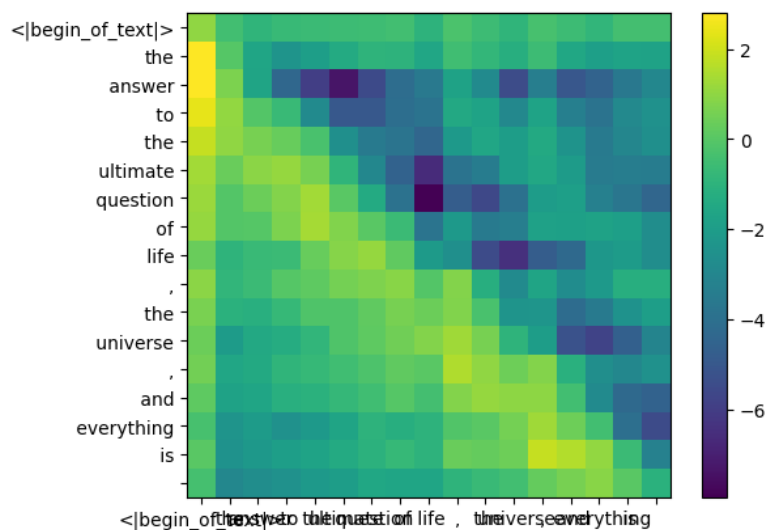
```
torch.Size([17, 17])
```

现在必须屏蔽 QK 分数

- 在 llama3 的训练过程中，未来的 token qk 分数被屏蔽。
- 为什么？因为在训练过程中，只学习使用过去的 token 来预测 token。

因此，在推理过程中，将未来的 token 设置为零。

```
1 def display_qk_heatmap(qk_per_token):  
2     _, ax = plt.subplots()  
3     im = ax.imshow(qk_per_token.to(float).detach(), cmap='viridis')  
4     ax.set_xticks(range(len(prompt_split_as_tokens)))  
5     ax.set_yticks(range(len(prompt_split_as_tokens)))  
6     ax.set_xticklabels(prompt_split_as_tokens)  
7     ax.set_yticklabels(prompt_split_as_tokens)  
8     ax.figure.colorbar(im, ax=ax)  
9  
10 display_qk_heatmap(qk_per_token)
```

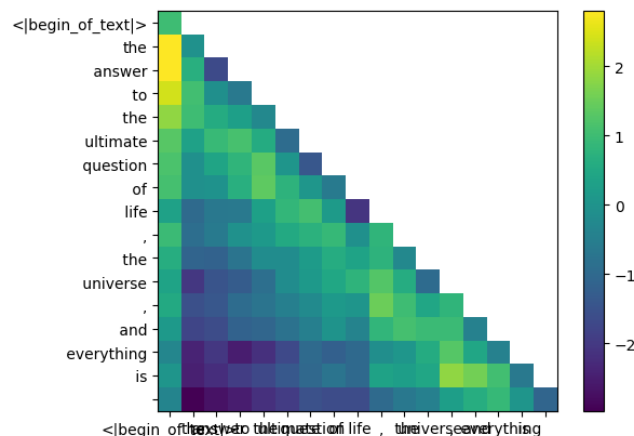


attention热力图

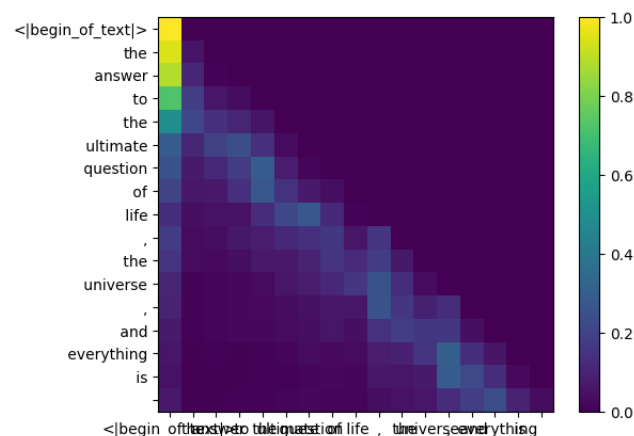
```
1 mask = torch.full((len(tokens), len(tokens)), float("-inf"),
  device=tokens.device)
2 mask = torch.triu(mask, diagonal=1)
3 mask
```

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

```
1 qk_per_token_after_masking = qk_per_token + mask
2 display_qk_heatmap(qk_per_token_after_masking)
```



```
1 qk_per_token_after_masking_after_softmax =
  torch.nn.functional.softmax(qk_per_token_after_masking,
    dim=1).to(torch.bfloat16)
2 display_qk_heatmap(qk_per_token_after_masking_after_softmax)
```




📌 这些分数（0-1）用于确定每个 token 中融合 value 矩阵的比例

和 key 一样，value 权重也在每 4 个注意力头之间进行共享（以节省计算量）

因此，下面的 value 权重矩阵的 shape 为 `[8x128x4096]`

```
1 v_layer0 = model["layers.0.attention.wv.weight"]
2 v_layer0 = v_layer0.view(n_kv_heads, v_layer0.shape[0] // n_kv_heads, dim)
3 v_layer0.shape
```

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

 llama3的第一层，第一个头的权值矩阵如下所示：

```
1 v_layer0_head0 = v_layer0[0]
2 v_layer0_head0.shape
```

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

value 向量

现在使用 value 权重来获取每个 token 的注意力值，其大小为 `[17x128]`，其中 17 是 prompt 中的 token 数，128 是每个 token 的 value 向量的维度

```
1 v_per_token = torch.matmul(token_embeddings, v_layer0_head0.T)
2 v_per_token.shape
```

`torch.Size([17, 128])`

注意力(attention)

和每个 token 的 value 相乘后得到的注意力向量的 shape 为 `[17*128]`

```
1 qkv_attention = torch.matmul(qk_per_token_after_masking_after_softmax,
                                v_per_token)
2 qkv_attention.shape
```

`torch.Size([17, 128])`

多头注意力 (multi head attention)


现在已经有了第一层和第一个头的注意力值

现在将运行一个循环，并执行与上面单元格中相同的数学运算，但只针对第一层中的每个头


```


1 qkv_attention_store = []
2
3 for head in range(n_heads):
4     q_layer0_head = q_layer0[head]
5     k_layer0_head = k_layer0[head//4] # key weights are shared across 4 heads
6     v_layer0_head = v_layer0[head//4] # value weights are shared across 4 heads
7     q_per_token = torch.matmul(token_embeddings, q_layer0_head.T)
8     k_per_token = torch.matmul(token_embeddings, k_layer0_head.T)
9     v_per_token = torch.matmul(token_embeddings, v_layer0_head.T)
10
11     q_per_token_split_into_pairs =
12     q_per_token.float().view(q_per_token.shape[0], -1, 2)
13     q_per_token_as_complex_numbers =
14     torch.view_as_complex(q_per_token_split_into_pairs)
15     q_per_token_split_into_pairs_rotated =
16     torch.view_as_real(q_per_token_as_complex_numbers * freqs_cis[:len(tokens)])
17     q_per_token_rotated =
18     q_per_token_split_into_pairs_rotated.view(q_per_token.shape)
19
20     k_per_token_split_into_pairs =
21     k_per_token.float().view(k_per_token.shape[0], -1, 2)
22     k_per_token_as_complex_numbers =
23     torch.view_as_complex(k_per_token_split_into_pairs)
24     k_per_token_split_into_pairs_rotated =
25     torch.view_as_real(k_per_token_as_complex_numbers * freqs_cis[:len(tokens)])
26     k_per_token_rotated =
27     k_per_token_split_into_pairs_rotated.view(k_per_token.shape)
28
29     qk_per_token = torch.matmul(q_per_token_rotated,
30     k_per_token_rotated.T)/(128)**0.5
31     mask = torch.full((len(tokens), len(tokens)), float("-inf"),
32     device=tokens.device)
33     mask = torch.triu(mask, diagonal=1)
34     qk_per_token_after_masking = qk_per_token + mask
35     qk_per_token_after_masking_after_softmax =
36     torch.nn.functional.softmax(qk_per_token_after_masking,
37     dim=1).to(torch.bfloat16)
38     qkv_attention = torch.matmul(qk_per_token_after_masking_after_softmax,
39     v_per_token)
40     qkv_attention = torch.matmul(qk_per_token_after_masking_after_softmax,
41     v_per_token)
42     qkv_attention_store.append(qkv_attention)
43
44 len(qkv_attention_store)

```

 现在有了第一个层的 32 个头的 qkv_attention 矩阵，接下来将把所有注意力分数合并成一个大矩阵，大小为 [17x4096]

```
1 stacked_qkv_attention = torch.cat(qkv_attention_store, dim=-1)
2 stacked_qkv_attention.shape
```


| torch.Size([17, 4096])

 **权重矩阵，最后几步之一**

对于第0层，最后要做的一件事是，将权重矩阵相乘


```
1 w_layer0 = model["layers.0.attention.wo.weight"]
2 w_layer0.shape
```

| torch.Size([4096, 4096])

 **这是一个简单的线性层，所以只需要进行乘法运算**


```
1 embedding_delta = torch.matmul(stacked_qkv_attention, w_layer0.T)
2 embedding_delta.shape
```

| torch.Size([17, 4096])

 **注意之后，现在有了嵌入值的变化，应该将其添加到原始的 token embeddings 中（残差）**

```
1 embedding_after_edit = token_embeddings_unnormalized + embedding_delta
2 embedding_after_edit.shape
```

| torch.Size([17, 4096])

 **将其归一化**

```
1 embedding_after_edit_normalized = rms_norm(embedding_after_edit,
    model["layers.0.ffn_norm.weight"])
2 embedding_after_edit_normalized.shape
```

```
torch.Size([17, 4096])
```

加载 FFN 权重并实现前馈网络

在 llama3 中，使用了 `SwiGLU` 前馈网络，这种网络架构非常擅长非线性计算。

如今，在 LLMS 中使用这种前馈网络架构是相当常见的

```
1 w1 = model["layers.0.feed_forward.w1.weight"]
2 w2 = model["layers.0.feed_forward.w2.weight"]
3 w3 = model["layers.0.feed_forward.w3.weight"]
4 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)
5 output_after_feedforward.shape
```

```
torch.Size([17, 4096])
```

在第一层之后，终于为每个 token 生成了新的 EMBEDDINGS（残差）

离结束还剩 31 层（一层 for 循环）

可以将经过编辑的 embedding 想象为包含有关第一层上提出的所有 query 的信息

现在，每一层都会对 query 进行越来越复杂的编码，直到得到一个 embedding，其中包含了需要的下一个 token 的所有信息。

```
1 layer_0_embedding = embedding_after_edit+output_after_feedforward
2 layer_0_embedding.shape
```

```
torch.Size([17, 4096])
```

整合

```

1 final_embedding = token_embeddings_unnormalized
2 for layer in range(n_layers):
3     qkv_attention_store = []
4     layer_embedding_norm = rms_norm(final_embedding, model[f"layers.
{layer}.attention_norm.weight"])
5     q_layer = model[f"layers.{layer}.attention.wq.weight"]
6     q_layer = q_layer.view(n_heads, q_layer.shape[0] // n_heads, dim)
7     k_layer = model[f"layers.{layer}.attention.wk.weight"]
8     k_layer = k_layer.view(n_kv_heads, k_layer.shape[0] // n_kv_heads, dim)
9     v_layer = model[f"layers.{layer}.attention.wv.weight"]
10    v_layer = v_layer.view(n_kv_heads, v_layer.shape[0] // n_kv_heads, dim)
11    w_layer = model[f"layers.{layer}.attention.wo.weight"]
12    for head in range(n_heads):
13        q_layer_head = q_layer[head]
14        k_layer_head = k_layer[head//4]
15        v_layer_head = v_layer[head//4]
16        q_per_token = torch.matmul(layer_embedding_norm, q_layer_head.T)
17        k_per_token = torch.matmul(layer_embedding_norm, k_layer_head.T)
18        v_per_token = torch.matmul(layer_embedding_norm, v_layer_head.T)
19        q_per_token_split_into_pairs =
q_per_token.float().view(q_per_token.shape[0], -1, 2)
20        q_per_token_as_complex_numbers =
torch.view_as_complex(q_per_token_split_into_pairs)
21        q_per_token_split_into_pairs_rotated =
torch.view_as_real(q_per_token_as_complex_numbers * freqs_cis)
22        q_per_token_rotated =
q_per_token_split_into_pairs_rotated.view(q_per_token.shape)
23        k_per_token_split_into_pairs =
k_per_token.float().view(k_per_token.shape[0], -1, 2)
24        k_per_token_as_complex_numbers =
torch.view_as_complex(k_per_token_split_into_pairs)
25        k_per_token_split_into_pairs_rotated =
torch.view_as_real(k_per_token_as_complex_numbers * freqs_cis)
26        k_per_token_rotated =
k_per_token_split_into_pairs_rotated.view(k_per_token.shape)
27        qk_per_token = torch.matmul(q_per_token_rotated,
k_per_token_rotated.T)/(128)**0.5
28        mask = torch.full((len(token_embeddings_unnormalized),
len(token_embeddings_unnormalized)), float("-inf"))
29        mask = torch.triu(mask, diagonal=1)
30        qk_per_token_after_masking = qk_per_token + mask
31        qk_per_token_after_masking_after_softmax =
torch.nn.functional.softmax(qk_per_token_after_masking,
dim=1).to(torch.bfloat16)
32        qkv_attention = torch.matmul(qk_per_token_after_masking_after_softmax,
v_per_token)
33        qkv_attention_store.append(qkv_attention)

```

```

34
35     stacked_qkv_attention = torch.cat(qkv_attention_store, dim=-1)
36     w_layer = model[f"layers.{layer}.attention.wo.weight"]
37     embedding_delta = torch.matmul(stacked_qkv_attention, w_layer.T)
38     embedding_after_edit = final_embedding + embedding_delta
39     embedding_after_edit_normalized = rms_norm(embedding_after_edit,
model[f"layers.{layer}.ffn_norm.weight"])
40     w1 = model[f"layers.{layer}.feed_forward.w1.weight"]
41     w2 = model[f"layers.{layer}.feed_forward.w2.weight"]
42     w3 = model[f"layers.{layer}.feed_forward.w3.weight"]
43     output_after_feedforward =
torch.matmul(torch.functional.F.silu(torch.matmul(embedding_after_edit_normaliz
ed, w1.T)) * torch.matmul(embedding_after_edit_normalized, w3.T), w2.T)
44     final_embedding = embedding_after_edit+output_after_feedforward

```

得到最终 Embedding，对下一个 token 做预测

embedding 的 shape 与常规 token embedding shape `[17x4096]` 相同，其中 17 是 token 数量，4096 是 embedding 维度

```

1 final_embedding = rms_norm(final_embedding, model["norm.weight"])
2 final_embedding.shape

```

```
torch.Size([17, 4096])
```

最后，将 embedding 解码为 token value

将使用输出解码器将最终 embedding 转换为 token。

```
1 model["output.weight"].shape
```

```
torch.Size([128256, 4096])
```

使用最后一个 token 的 embedding 来预测下一个值

希望在我们预料之内, 42 :)

注意：根据《银河系漫游指南》书中提到，“生命、宇宙和一切的终极问题的答案是 42”。大多数现代语言模型在这里应该会回答 42，这应该能验证我们的整个代码！祝我好运！

```
1 logits = torch.matmul(final_embedding[-1], model["output.weight"].T)
2 logits.shape
```

| torch.Size([128256])

 模型预测的 token 编号是 2983，这是否代表 42 的 token 编号？

```
1 next_token = torch.argmax(logits, dim=-1)
2 next_token
```

| tensor(2983)

```
1 tokenizer.decode([next_token.item()])
```

| 42

 OHHHHHHHHHHH!

参考资料

- RoPE

https://www.zhihu.com/tardis/zm/art/647109286?source_id=1003

www.zhihu.com