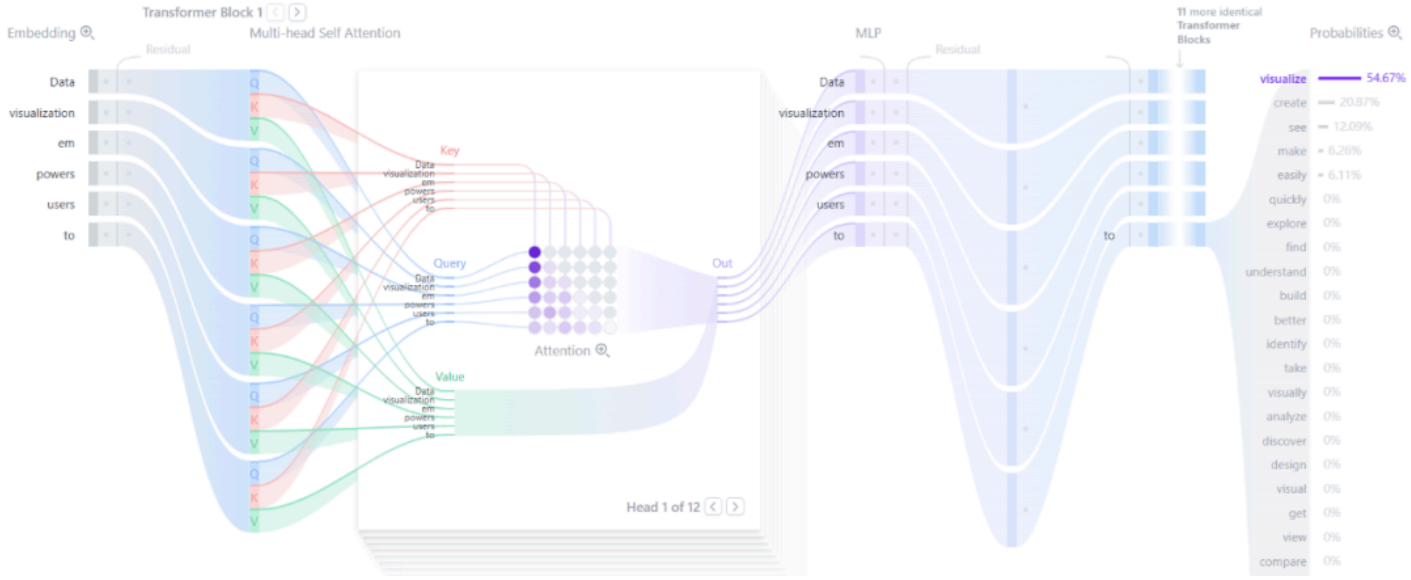
LLM 基础 - 让 Transformer 模型可视化

Original Accela推箱子 Accela推箱子 2025年12月28日 20:29

大模型（Large Language Model ，LLM）技术的迭代犹如技术爆炸，半年前的先进技术如今已成基础的基础。Polo Club 做了 Transformer 模型的动态图解，对可视化地理解 Transformer 的工作原理非常友好。另外，DeepSeek-V3 的论文犹如嘉年华大放送，详细介绍了从模型到基础设施的方方面面，并且还开源了，甚至讲解了生产级的 MoE 实现。

Polo Club 可视化：https://poloclub.github.io/transformer-explainer/



引用

DeepSeek-V3 的论文：

- [1] DeepSeek-V3 Technical Report: https://arxiv.org/pdf/2412.19437
- [2] DeepSeek infra: https://arxiv.org/html/2408.14158v1
- [3] DeepSeek-R1: https://arxiv.org/pdf/2501.12948

DeepSeek 开源：

- [4] 模型: https://github.com/deepseek-ai/DeepSeek-V3
- [5] 3FS: https://github.com/deepseek-ai
- [6] 更多组件: https://github.com/orgs/deepseek-ai/repositories?type=all

总体架构

从上图可以看到，Transformer 模型本质上是一打 Transformer Block（上十个或上百个）。今天的 Transformer 模型基本是 **Decoder-only** 的，因为更易训练，更易优化性能，以利用 **Scaling Laws** 。很早就有研究表明，神经网络（Neuro Network, NN）增加深度比增加宽度更有效，因为更深的神经网络类似组合函数，拟合表达能力更强，下一层可以利用上一层的中间结果。

下面讲解上图中的组件：

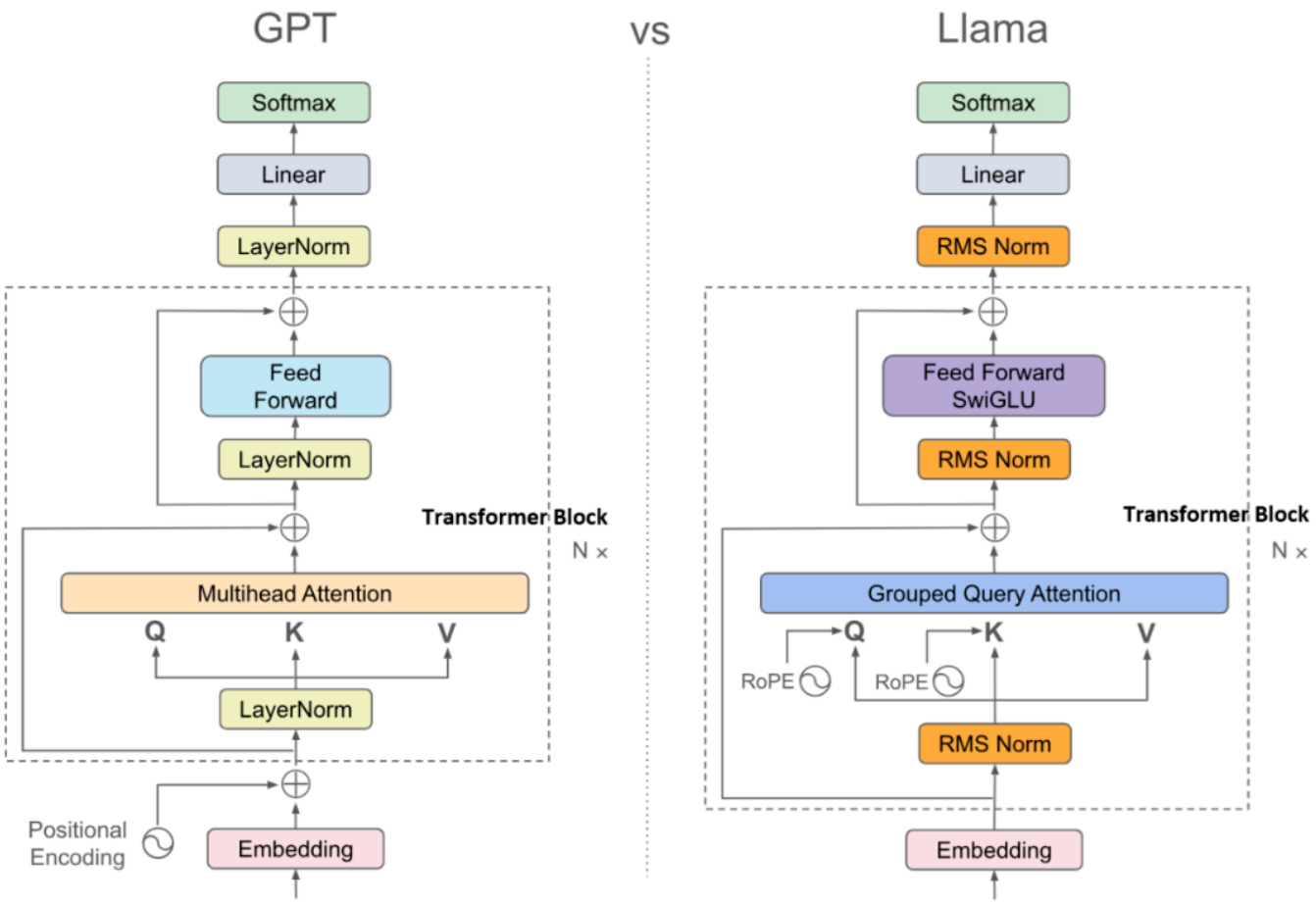
- Transformer 模型入口处有一个 **Embedding 层**，它做了分词（Tokenization），Token Embedding，**位置编码**（Positional Encoding）三件事。分词基于频率识别词缀和词组，例如“un”、“ing”、“able” 也被识别为单独的 Token 。Token Embedding 将 Token ID 映射为向量，其参数与 Transformer 模型一起训练得到。位置编码，例如 Sinusoidal Encoding ，将 Token 在句子中的相对位置编码，融合进 Token 输入。

- 中间是一打堆叠的 **Transformer Block** ，这个架构和几年前的卷积神经网络 (Convolutional Neural Network，CNN) 有相通之处。CNN 用于图像视觉处理。其较低层的神经网络学习局部和句法模式，而较高层学习全局语义和推理，像一座“抽象之塔”。Attention 在 Transformer Block 内部，后文详解。

- 最后是 **输出层**。最后一个 Transformer Block 输出的向量会被乘以一个大矩阵，up-project（见后文）到一个和词汇库一样大的向量，称作 Logits 。随后，Logits 被选择 Top-K 个值最大的 Token 。最后，用 Softmax 将 Logits 值映射成概率，从概率中抽样选择一个 Token 输出。可选地，在进入 Softmax 前，可以将 Logits 除以 Temperature 值。Temperature 越高，则概率被拉平，模型的最终输出具有更高的随机性（高温高熵）。

Embedding 在 Transformer 前已有多年的应用，它将词语转换成具有几何距离的高维向量，这样模型才能处理。而位置编码与 Transformer 一起被广为人知，它的数学即简单又精妙。输出层的 Softmax 方法是神经网络常用的。而 **Transformer Block** 中包含着大模型的关键，见下文。

（图片来源：https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/te_llama/tutorial_accelerate_hf_llama_with_te.html）



引用

- [7] 为什么现在的LLM都是Decoder only的架构: https://www.zhihu.com/question/588325646
- [8] 为什么神经网络更深比更宽有效: https://www.reddit.com/r/MachineLearning/comments/h0g83p/d_why_are_deeper_networks_better_than_wider/
- [9] 卷积神经网络 CNN: https://learnopencv.com/understanding-convolutional-neural-networks-cnn/
- [10] 位置编码详解: https://medium.com/thedeephub/positional-encoding-explained-a-deep-dive-into-transformer-pe-65cfe8cfe10b

Transformer Block

Transformer 模型的核心是 Transformer Block ，其中具体有什么呢?

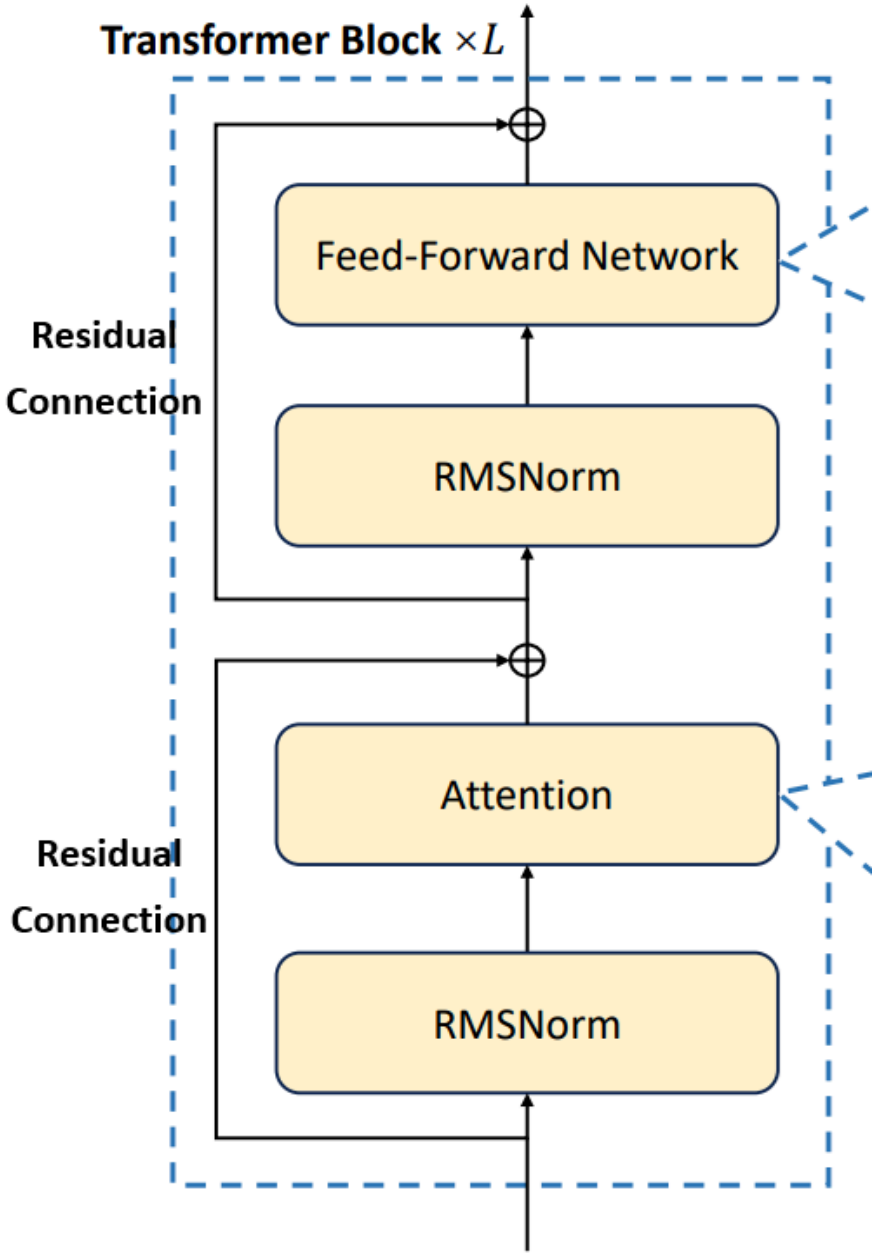
- 首先是 **Attention** 模块，Transformer 的魔力来源于此，后文章节将详述。剩下的部分是神经网络，以及一些技巧。

- 其次是 **前馈神经网络**（Feed-forward network，FFN）。前馈神经网络只是最基础的神经网络的夸张名字。网络结构简单直接：输入层 -> 隐藏层 -> 输出层，隐藏层可以有多层。被称为“前馈”是因为数据只能向前流动。FFN 也常常被称作：多层感知机（Multilayer Perceptron, MLP）、全连接神经网络（Fully Connected Network，FCN）。

- **RMSNorm**（Root Mean Square Normalization），直译为用均方根来做标准化。神经网络各层间常作标准化（Normalization），以规范激活的取值范围，避免梯度爆炸或消失。RMSNorm 逐渐取代 LayerNorm 成为大模型的主流。RMSNorm 参数更少，更易训练，并且能够保留向量的方向（而不是像 LayerNorm 一样减去平均值）。

- 下一个关键是 **残差连接**（Residual Connection），出自 ResNet 。在深度神经网络中增加一个跨层的连接，可以缓解“深”带来危害，避免信息消失、错误积累，帮助梯度在深层次中传递。

(图片出自 DeepSeek-V3 论文。)



FFN 是神经网络的基础，早已有数十年的应用。RMSNorm 也很常见，神经网络各层见总是需要标准化，Scaling、Normalization、Standardization 是处理特征的标准方法。残差连接简单有效，在几年前的 ResNet 推出后被广泛采用。而 **Attention** 才是真正的原创，Attention is all you need 。

引用

- [11] FFN: https://learnopencv.com/understanding-feedforward-neural-networks/
- [12] Normalization: https://2020machinelearning.medium.com/deep-dive-into-deep-learning-layers-rmsnorm-and-batch-normalization-b2423552be9f
- [13] ResNet: https://en.wikipedia.org/wiki/Residual_neural_network

Attention

Attention 是 Transformer 的关键。下图是注意力（Attention）的公式。

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

原公式的信息密度很高，下面用代码把它展开成更普通的形式，然后逐条讲解每个组件。

```
// InputMatrix 的每一行是一个输入 Token 的 embedding 向量
// 行数是 Token 的总数
Q = InputMatrix * WeightMatrix_Q
K = InputMatrix * WeightMatrix_K
V = InputMatrix * WeightMatrix_V

// Attention 的公式
Attention = softmax((InputMatrix * WeightMatrix_Q
      * WeightMatrix_K^T * InputMatrix^T)
    / sqrt(d_k))
  * (InputMatrix * WeightMatrix_V)
```

下面逐项讲解公式中的组件：

- **输入矩阵**（InputMatrix）：Transformer 以 Token 作为输入，每个 Token 表示为一个向量（Embedding）。一次文本输入可达数十至百万 Token，Token 向量有数百至千维。Token 向量以行表示，一行行叠起来组成输入矩阵。（有的文章中，Token 向量是以列表示的，你会看到 Attention 公式变成 Q^T * K，Q = WeightMatrix_Q * InputMatrix，不影响结果。）

- **Q = InputMatrix * WeightMatrix_Q**：Q 缩写 Query。在计算 Attention 前，输入（InputMatrix）被事先乘以一个权重 WeightMatrix_Q，得到 Q。用权重处理输入是常用手法，K、V 也由此得到。K 缩写 Key，V 缩写 Value。

- **QK^T**：方形矩阵，维度是 Token 的总数（非常巨大），矩阵的元素是浮点数值（不是向量）。展开后其形式是 InputMatrix * M * InputMatrix^T。矩阵 (i,j) 位置是 Token i 和 Token j 的点积（加权

重）。形象的含义是文本输入（一串 Token）自己看向自己（相乘）。这是 “Self-attention” 的由来，“自注意力”。

- **Sqrt(d_k)**：d_k 是 Token 向量的维数。Sqrt(d_k) 常被称作 “Scaling factor”。假设 Q 和 K 中的元素符合正态分布 N(0,1)，QK^T 会将方差增加到 d_k。为了使结果的方差回到 1，需要除以 Scaling factor —— sqrt(d_k)。

- **Softmax**：它是常用于概率的标准化（Normalization）公式，按行作用于输入矩阵。经过 softmax，一行中的元素之和变为 1，这一行变得像一组概率。QK^T 经过 softmax 处理，变成了 Token i 看向 Token j 的概率（Self-attention）。Softmax 公式用 e 求指数，因为概率与指数有天然联系，可以看指数分布下的 MTBF。

- ***V**：经过上面，softmax(QK^T/sqrt(d_k)) 得到了 Self-attention 的概率。概率还需要乘以输入的值，才能得到同一量纲的输出。V 就是输入的值（加权重），所以叫 Value。

总结起来，Attention 可看作给 Token 输入（InputMatrix）加了权，权重是 Self-attention。注意 Q、K、V 的权重是右乘，加权作用于单个 Token 向量内。而 Self-attention 的权重是左乘，加权作用于多个 Token 间，使 Token **混合**。

此外，Attention 还有一些额外要点：

- **多头**（Multi-head）：Transformer Block 中的 Attention 层是 Multi-head Attention。即这层中同时在跑多个 Attention 模型，每个被称为一个“头”，各自拥有独立的参数，总头数有十几到几十。“多头”的术语可能源自图像处理（如 CNN），处理 RGB 像素至少需要三个通道三个头。

- **Masking**（掩码）：如果 Transformer 支持 1M Token 的上下文，那么 Attention 理论上在处理 1M * 1M 维度的矩阵。但实际上，见 KV Cache 一节，Attention 的计算实现是一个 Token 一个 Token 进行的。如果用户输入是 1000 个 Token，那么在计算到第 900 个 Token 时（Prefill），后面的 100 个 Token 需要填零。这是 Masking 的第一重含义。用户输入加输出大概填不满 1M 长度的上下文，那么这些填不满的部分也需要填零。这是 Masking 的第二重含义。零在 softmax 计算时，并不能转换成零概率（e^0=1）。为了概率零，需要将输入从零改成负无穷。这是 Masking 的第三重含义。

引用

- [14] 深入理解 Attention 机制 P2: https://medium.com/@funcry/in-depth-understanding-of-attention-mechanism-part-ii-scaled-dot-product-attention-and-its-7743804e610e

Attention 的几何形象

Attention 其实有非常生动且符合直觉的几何形象。在有了上面的基础之后，下面更深入地解释 Attention 在做什么：

1. **Embedding**：这是 Attention 的输入。Embedding 将 Token 转换为向量，开始进行 Attention 计算。这些向量是有几何含义的。两个向量的夹角小、点积（Dot product）大，表示对应的 Token 的含义接近。向量相加，能够混合含义生成新的 Token，如 king + woman = queen 。

2. **Q = InputMatrix * WeightMatrix_Q**：Embedding 的 Token 向量没有直接参与 Attention 计算，而是先乘以 Q 的权重矩阵。把 WeightMatrix_Q 的每一列想象成另一组 Embedding 向量，它们是尺子。其与 Token 向量相乘，是在测量 Token 与尺子有多相近。输出 Q ，则是以这组尺子为基准坐标，重建出来的 Token 。WeightMatrix_K 同理。

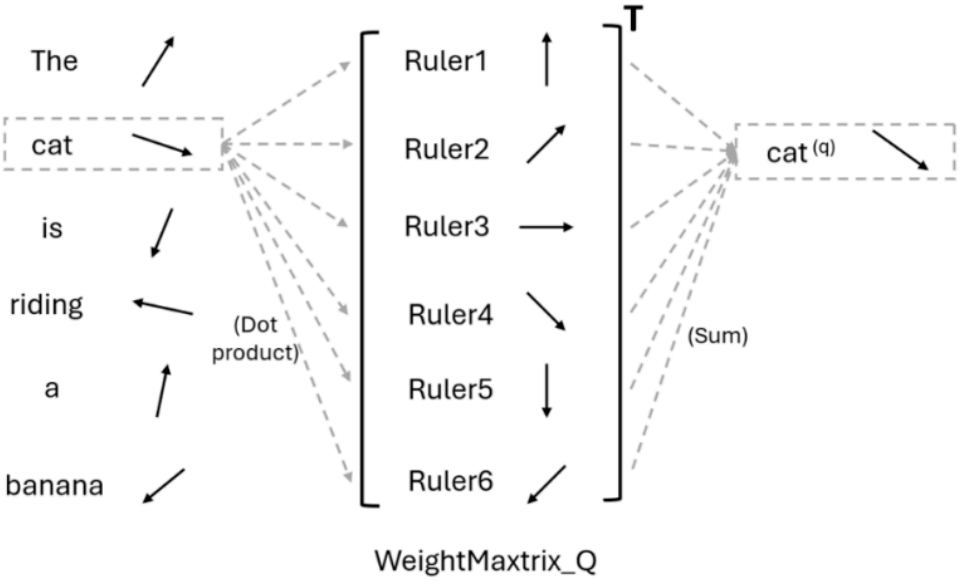
3. **位置编码**：这里以 RoPE 举例，简化来讲，它将 Token 向量旋转了一个角度。Token 在句子中的位置越靠后，旋转角度越大。两个 Token 的距离越远，旋转角度之差越大。在句子中相隔同样距离的两个 Token，无论位置，旋转角度之差总是相同的。角度差越小，点积越大。这样，位置编码使相邻的 Token 得到更高的 Attention 分数。

4. **QK^T**：这是把 Token 两两互相求点积，点积就是 Attention 分数（之后通过 sqrt(d) 和 softmax 做标准化）。那么，两个 Token 向量的夹角越近，含义越近，在句子中的位置越近，就会拥有更大的 Attention 分数。

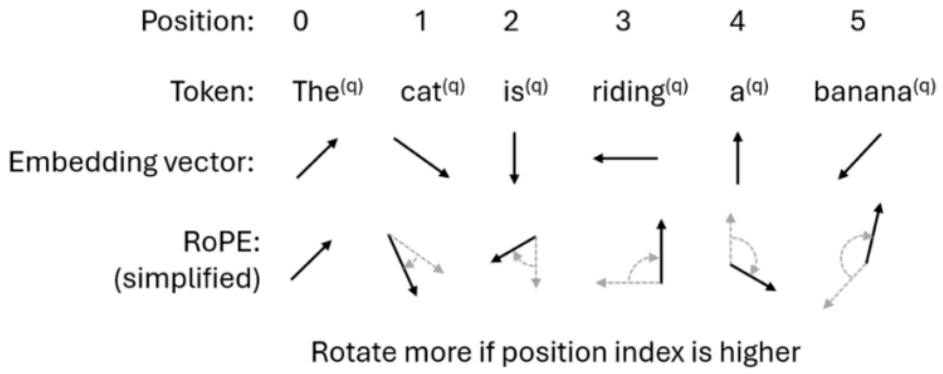
5. ***V**：Attention 计算最后将 Attention 分数乘以 V ，这是在将 V 各行所代表的 Token 向量进行混合。如果这一行的 Token 对另一 Token 有更高的 Attention 分数，那么后者混合时的权重更高。Attention 最终的输出中，原 Token 向量变成了——自己 + 它所“关注”的句子上下文。

(图中忽略了 sqrt(d)、softmax 等用于 Normalization 的部分。)

Q = InputMatrix * WeightMatrix_Q



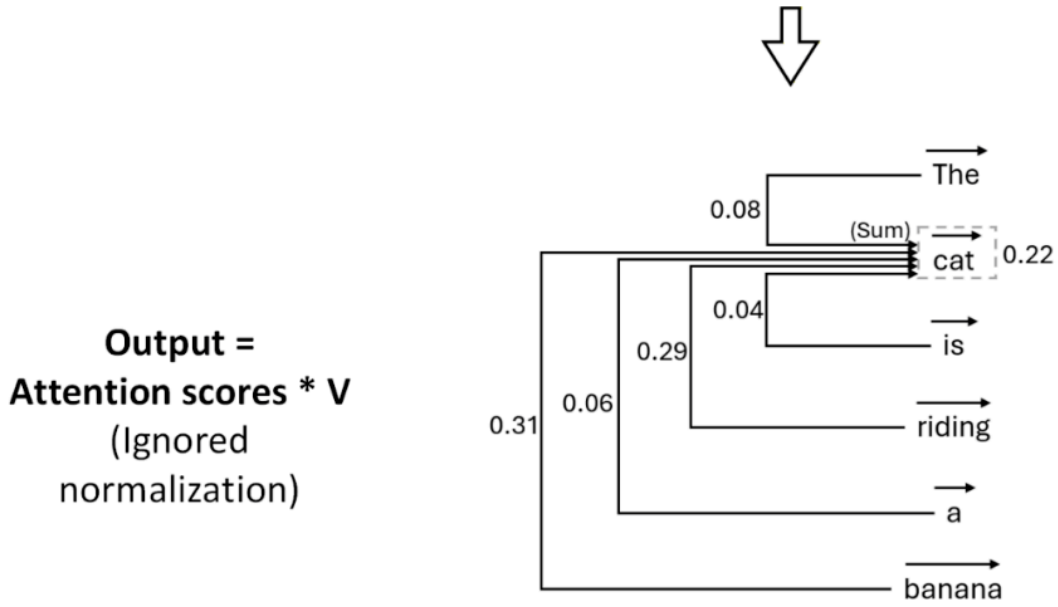
Apply positional encoding: Q = RoPE(pos)*Q



Calculate attention scores: QK^T

	The(k)	cat(k)	is(k)	riding(k)	a(k)	banana(k)
The(q)	0.27	0.05	0.14	0.08	0.22	0.24
cat(q)	0.08	0.22	0.04	0.29	0.06	0.31
is(q)	0.17	0.03	0.31	0.41	0.03	0.03
riding(q)	0.10	0.28	0.05	0.21	0.03	0.33
a(q)	0.14	0.12	0.07	0.02	0.36	0.29
banana(q)	0.03	0.05	0.18	0.45	0.08	0.21

(Dot product)



有了几何解释，可以发现，Attention 分数本质是在计算 Token 向量间的两两相似性。如果 **没有** 位置编码，那么 Attention 将变成对 Token 是 **顺序无关的**（Permutation Invariant）。因此，位置编码才必须要被引入。

If without positional encoding:

$$\text{Attention}(X) = \text{Attention}(\pi(X))$$

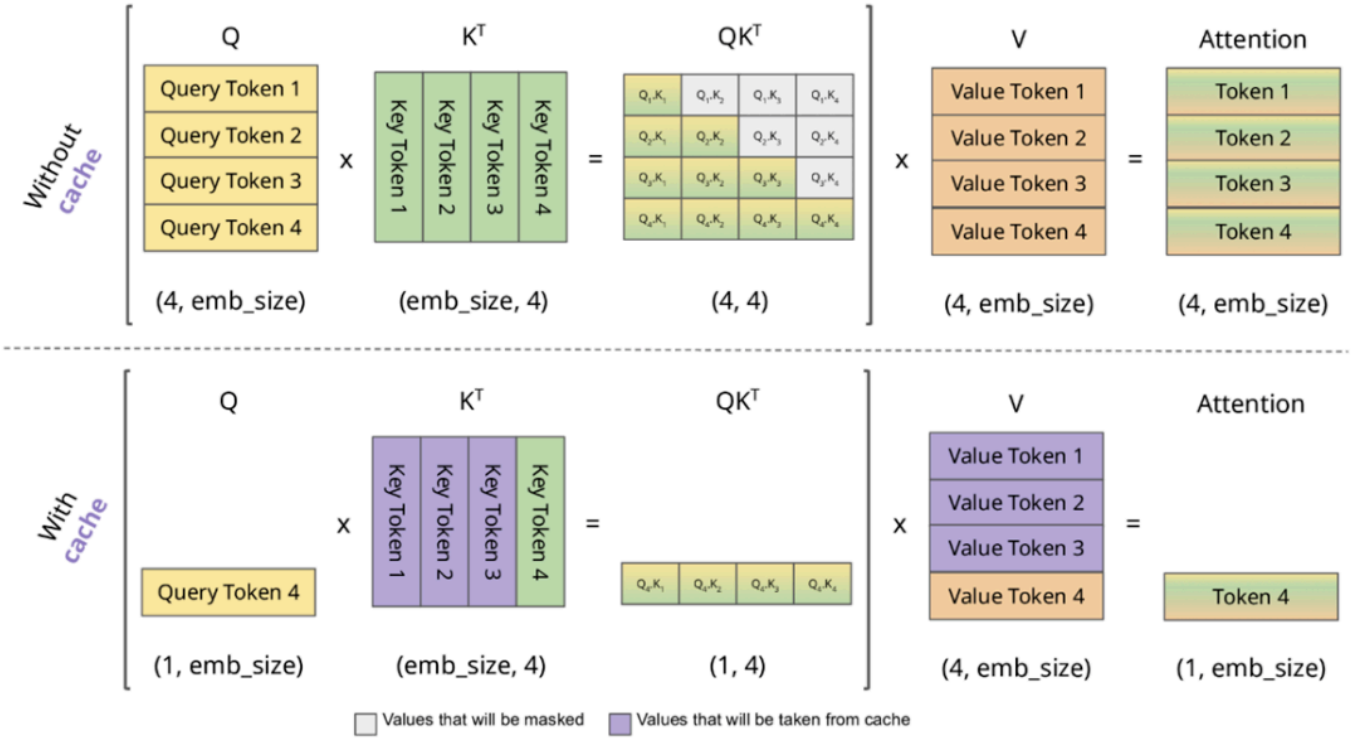
KV Cache

Attention 的另一巨大优势在于计算。如果 Transformer 支持 1M Token 的上下文，那么 Attention 理论上在处理 1M * 1M 维度的矩阵相乘。但实际上，Attention 可以实现成一个 Token 一个 Token 增量、递进地计算。下一个 Token 利用上一个 Token 的计算结果，即 KV Cache 。

下图链接中，作者制作了可视化的动态图，非常有助于理解 KV Cache 的计算过程。可以看到：

1. 计算 Token 2 时，需要用到 Value Token 1~2，Key Token 1~2，Query Token 2 。
2. 计算 Token 3 时，需要用到 Value Token 1~3，Key Token 1~3，Query Token 3 。Value Token 1~2 和 Key Token 1~2 可复用 Token 2 的计算。
3. 计算 Token 4 时，需要用到 Value Token 1~4，Key Token 1~4，Query Token 4 。Value Token 1~3 和 Key Token 1~3 可复用 Token 3 的计算。

（KV Cache 动画：https://medium.com/@joaolages/kv-caching-explained-276520203249 ）



计算下一个 Token 时，总是需要全量的 Key Token 和 Value Token，它们可以 **递进地复用** 上一轮 Token 的计算结果，因而适合存入 Cache ，这是 KV Cache 的由来。而 Query 仅需当前 Token 的，不需要全量，因而没有 Q Cache 。

- 可以看到，用户输入的 **上下文越长**，Token 越多，所需的 KV Cache 成比例地越大，成本越高。这解释了为什么 ChatGPT 或同类产品在 UI 设计上，倾向于引导用户开始全新的对话。

注意，KV Cache **依赖于之前计算过的 Token** 。如果上下文中间的 Token 换了一个，后面的 KV Cache 就不能复用了。这是因为 Transformer 有很多中间层，中间层的 Transformer Block 也有自己的 Attention 和 KV Cache 。中间层的 Token 来自上一层的 Attention 。Attention 的概率加权是跨 Token 的，历史 Token 的值会造成当前 Token 的值的不同。

- 因此，KV Cache 常作为 **Prefix Cache** 实现。Prefix（之前的对话历史）一致，KV Cache 才可复用。

- 在 Agent 设计中，**System Prompt** 总是放在对话开头，内容总是一样。它们的 KV Cache 可以跨对话、跨用户地被复用。

由于 KV Cache ，在使用 Transformer 进行推理（而非训练）时，常常将其分为 **Prefll 阶段** 和 **Decode 阶段**。Prefill 阶段将用户输入的文本输入 Transformer，旨在生成 KV Cache ，因而被称作“预填充”（Prefill）。Decode 阶段发生在 Prefill 完成后，Transformer 开始根据用户输入的文本预测后面的文字，产生输出。

- Prefill 和 Decode 阶段有不用的计算特性。Prefill 阶段，用户输入已知，系统设计旨在最大化并行，瓶颈常在于 **GPU 计算能力**。而 Decode 阶段只能线性地一个一个 Token 地预测，但 KV Cache 需要全部缓存，系统的瓶颈常在于 **内存（显存）**。

引用

- [15] Mooncake / Kimi 的 Prefill / Decode 分离设计：https://arxiv.org/pdf/2407.00079
- [16] Cursor 关于 Prompt processing（Prefill阶段）和 Generation（Decode阶段）的成本计算：https://cursor.com/blog/llama-inference#a-primer-in-transformer-math

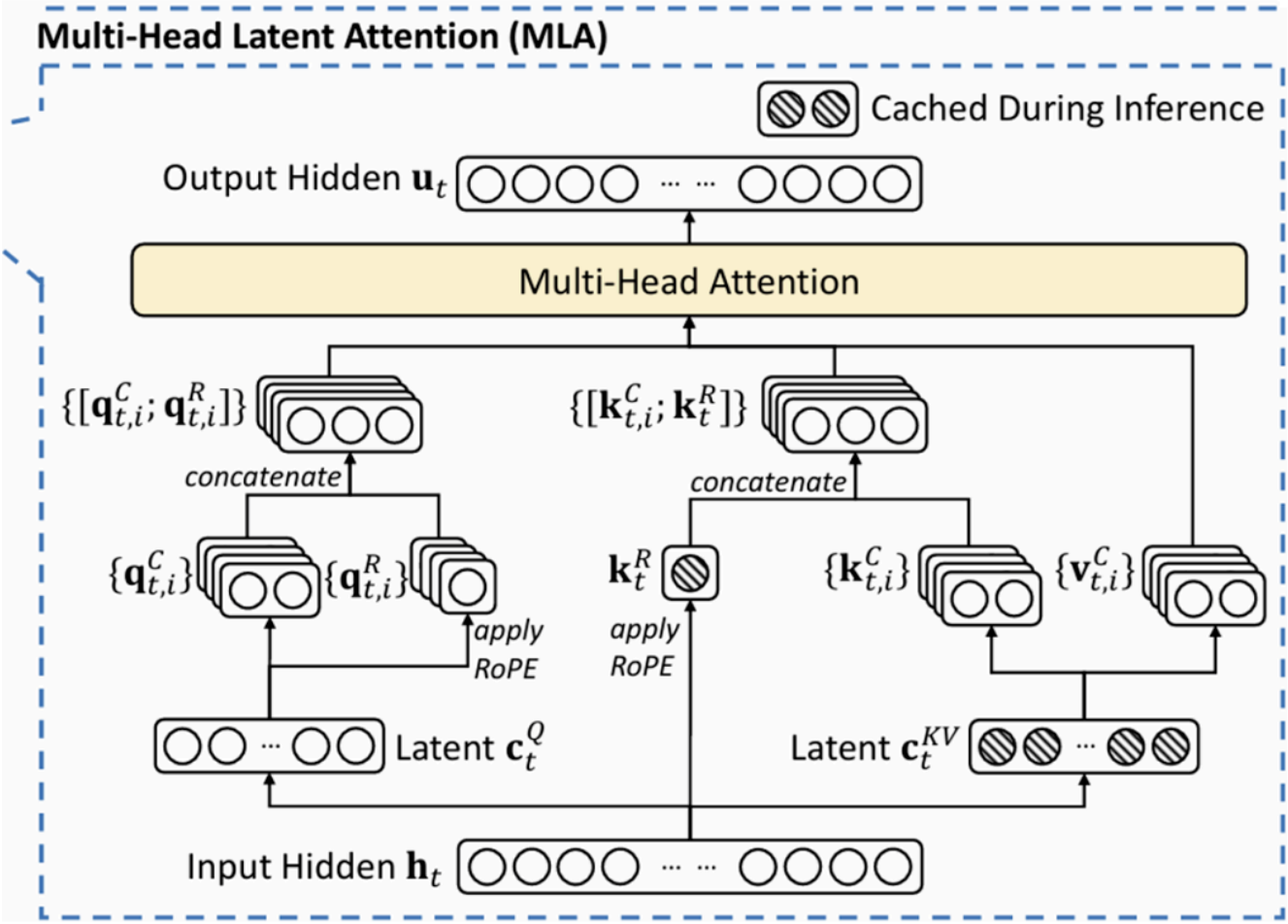
Multi-Head Latent Attention (MLA)

下面介绍一下 DeepSeek-V3 论文中的 Multi-Head Latent Attention（MLA）。MLA 的“魔法”是通过 down-projection 压缩来减少 KV Cache 占用，从而提升推理性能。而测试表明这种压缩对推理表现的影响不大。

下图公式中，

- 蓝字方框中的向量被 KV Cache 缓存。
- h_t 表示 Token 向量（embedding），它是 Attention 的输入。
- W^D* 矩阵表示 down-projection 矩阵，W^U* 矩阵表示 up-projection 矩阵。
- k、q、v 分别指 Attention 中 K、Q、V 的向量。
- k_t,i 中，t 表示第 t 个 Token，i 表示第 i 个头（Multi-head）。
- c* 是 MLA 的引入的 latent 向量。即经过压缩的、隐藏在中间步骤的向量。
- *R 表示 RoPE 编码，用于编码 Token 在句子中的位置信息。

(图片出自 DeepSeek-V3 论文。)



$$\boxed{\mathbf{c}_t^{KV}} = W^{DKV}\mathbf{h}_t,$$

$$[\mathbf{k}_{t,1}^C; \mathbf{k}_{t,2}^C; ...; \mathbf{k}_{t,n_h}^C] = \mathbf{k}_t^C = W^{UK}\mathbf{c}_t^{KV},$$

$$\boxed{\mathbf{k}_t^R} = \mathrm{RoPE}(W^{KR}\mathbf{h}_t),$$

$$\mathbf{k}_{t,i} = [\mathbf{k}_{t,i}^C; \mathbf{k}_t^R],$$

$$[\mathbf{v}_{t,1}^C; \mathbf{v}_{t,2}^C; ...; \mathbf{v}_{t,n_h}^C] = \mathbf{v}_t^C = W^{UV}\mathbf{c}_t^{KV},$$

$$\mathbf{c}_t^Q = W^{DQ}\mathbf{h}_t,$$

$$[\mathbf{q}_{t,1}^C; \mathbf{q}_{t,2}^C; ...; \mathbf{q}_{t,n_h}^C] = \mathbf{q}_t^C = W^{UQ}\mathbf{c}_t^Q,$$

$$[\mathbf{q}_{t,1}^R; \mathbf{q}_{t,2}^R; ...; \mathbf{q}_{t,n_h}^R] = \mathbf{q}_t^R = \mathrm{RoPE}(W^{QR}\mathbf{c}_t^Q),$$

$$\mathbf{q}_{t,i} = [\mathbf{q}_{t,i}^C; \mathbf{q}_{t,i}^R],$$

$$\mathbf{o}_{t,i} = \sum_{j=1}^{t} \text{Softmax}_j\left(\frac{\mathbf{q}_{t,i}^T \mathbf{k}_{j,i}}{\sqrt{d_h + d_h^R}}\right)\mathbf{v}_{j,i}^C,$$

$$\mathbf{u}_t = W^O[\mathbf{o}_{t,1}; \mathbf{o}_{t,2}; \ldots; \mathbf{o}_{t,n_h}],$$

下面逐条讲解 MLA 的各组件：

1. 先看 **图中右半边**，不同于普通 Attention 直接用 h_t 计算 K、V，MLA 首先将 h_t 给 **down-project** 到 c_t^KV 。c_t^KV 被缓存，而不是分别缓存 K、V ，也不用为每个头缓存一份。这样减少了 KV Cache 的使用。随后，c_t^KV 被 up-project 还原出 Attention 计算所需的 K、V ，并且多头。

2. 再看 **图中左半边**，Q 的处理类似，也是先将 h_t 给 **down-project** 到 c_t^Q ，然后再 up-project 还原出 Attention 计算所需的 Q 向量。注意 Q 是不需要 KV Cache 缓存的。

3. 观察图中 **带有 R 的部分**，MLA 还为 K、Q 向量增加了 **RoPE** 位置编码，k_t^R 、q_t,i^R 。最终输入 Attention 的向量由 up-project 出的向量和 RoPE 编码后的向量拼接而成。注意 RoPE 编码的向量长度只有前者的一半。而且 k_t^R 没有多头，减少 KV Cache 使用。其技巧类似 MQA ，所有头共享 K 。

4. 最终 K、Q、V 输入标准的 **Multi-Head Attention** 进行计算。

可以看到，节省 KV Cache 的关键在于 down-projection 压缩。而 RoPE 位置编码被划分到独立路径特殊处理。

上文的 down-projection 压缩的技巧出现在大模型领域的多个方面，也被称作 **Low-rank Compression** ，与存储领域的“压缩”很不一样。例如：

- 应用1：**LoRA** 是大模型常用的微调技术，微调后的模型可以表示为 h = W0 * x + ΔW * x = W0 * x + BA * x 。模型本身的矩阵 W0 和参数更新 ΔW 维数巨大，更新成本高昂。但 ΔW 可以表达为两个较小的矩阵相乘 B * A 。B 的列数和 A 的行数可取任意较小的维数，使得 B 和 A 的参数量之和远小于 ΔW 。这是微调比训练成本更低的原因。

- 应用2：**MLP Expansion/Compression** 指多层神经网络的中间层升高维度或降低维度。例如，输入向量的维数是 768 ，但中间层的维数是 3072 ，最后输出又回到 768 维。将神经网络表达成矩阵形式，相当于输入向量首先乘以一个 up-projection 矩阵升维，进行计算，然后在输出时乘以一个

down-projection 矩阵降低还原到原维数。神经网络内部的高维数为存储知识、表达非线性提供了容量。这个技巧被用于 Transformer Block 的 FFN 层。

引用

- [17] Deepseek技术解读(1)-彻底理解 MLA: https://zhuanlan.zhihu.com/p/16730036197
- [18] LoRA 论文: https://arxiv.org/abs/2106.09685

关于 MLA 更深入的问题

上面讲解了 MLA 的工作原理，随后，可以引出一些关于 MLA 的更深入的问题：

为什么 RoPE 与 K、Q down-projection 需要 **分开计算**?

- 可以这样理解，回到 Attention 章节的公式，QK^T = InputMatrix * WeightMatrix_Q * WeightMatrix_K^T * InputMatrix^T 。其中 WeightMatrix_Q * WeightMatrix_K^T 可以事先 **融合** 成一个矩阵 M 以简化计算。并且 M 无论对第几个 Token，都是不变的。但如果中间插入了 RoPE，其对位置敏感，就导致 M 没有固定的值了，无法简化计算。

- 用 **论文中的解释** 为：RoPE 与低秩 KV 不兼容，RoPE 对 Q 和 K 都是位置敏感的，如果为 k_t^C 应用 RoPE ，那么 W^UK 将与位置敏感的 RoPE 矩阵耦合，导致 W^UK 无法被吸收到 W^UQ 中。

Q 不需要 KV Cache，但为什么 Q 也需要 **down-project** 到 c_t^Q ?

- 可以这样理解，DeepSeek-V3 的 h_t 向量维度是 7168，参与 Attention 计算的 Q 向量维度是 1536 。Naive 的方案是用矩阵 W 乘以 h_t 以降维，W 需要 7168 * 1536 维度，非常大。而 DeepSeek 的方案中，W 被替换为 W^DQ * W^UQ ，先降维再升维，中间 c 向量的维度是 512 。那么 W^DQ 矩阵的维度是 7168 * 512 ， W^UQ 矩阵的维度是 512 * 1536 。两者维度之和远小于 W 的 7168 * 1536 ，参数量大幅减小。这是经典的 **Low-rank Compression** 技巧。

- **论文中的解释** 很精简：使用 Low-rank Compression 减少训练中所需的 activation memory 。

普通 Transformer 只需要在开头做一次位置编码，但为什么 MLA 需要在 **每个 Attention 层** 都做 RoPE ?

- 老式的 Transformer 使用的是 Absolute Positional Encoding（Sinusoidal Encoding），这里称其为 APE 。APE 在 Transformer 开头编码一次，随着残差连接流过整个模型。而更新的模型如 Llama

则使用 RoPE 编码，**RoPE** 需要在每个 Attention 计算前都应用一遍，是 RoPE 论文如此设计。MLA 也是如此。

- 另一方面，如前文提到，RoPE 与 down-projection 变换 **不兼容**。因而也需要在计算 Attention 前反复重新应用 RoPE ，且 RoPE 的计算与 K、V 的 down-projection 路径是解耦的。

输入 InputMatrix 应该 **先应用 RoPE** 位置编码，还是应该先应用 WeightMatrix_Q 得到 Q？（同理 WeightMatrix_K）

- 与 APE 应用于 Transformer 开头、WeightMatrix_Q 之前不同，RoPE 要求应用在 **WeightMatrix_Q 之后**，RoPE 论文如此设计。前者是对 Token 向量加上一个 Position 向量，而 RoPE 是乘上旋转矩阵。如果 RoPE 不应用在 WeightMatrix_Q 之后，旋转所带来的 Q、K 间的 **几何角度** 就会被 WeightMatrix_Q 破坏。

为什么 Q、K 需要应用 RoPE 位置编码，而 **V 不需要**？

- 使用 RoPE 是为了让 Q、K 按照句子中的位置 **形成角度差**，以计算 Attention 分数。而 V 用于保存 Token 原始信息，它是被用来乘以 Attention 分数的，没有必要再用 RoPE 进行旋转。

- APE 与 RoPE 不同，APE 在 Transformer 开头应用位置编码，导致所有 K、Q、V 都含有位置编码。而 RoPE 则 **只应用到** Q、K 上，见 RoPE 论文。

RoPE 中，旋转角度具有周期性，会导致距离较远的 Token 向量反而得到相近的角度吗，即“**撞周期**”？

- 这需要看 RoPE 的公式，见引用部分。RoPE 并不是简单地把 Token 向量旋转了一个角度，而是 Token 向量首先被分成一系列 2D 分量，**每个 2D 分量都按不同的速度旋转**。对于 Token 位置，低维分量旋转快，高维分量旋转慢，犹如时钟的秒、分、小时指针。单个分量的旋转有周期性，但完整的 Token 就很难撞周期了。

- **低维分量旋转快，高维分量旋转慢**。这使得 Token 向量的低维分量关注句子的局部信息，而高维分量关注句子的全局语义。Embedding 本身并不要求高低维分量有不同，但 Transformer 的 Embedding 是和 Transformer 一起训练出来的。这使得 Token 向量可以分化对句子远近的学习。

(RoPE 的公式，图片出自 RoPE 论文。)

$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \boldsymbol{R}^d_{\Theta,m}\boldsymbol{W}_{\{q,k\}}\boldsymbol{x}_m$$

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

$$\boldsymbol{q}_m^\intercal \boldsymbol{k}_n = (\boldsymbol{R}^d_{\Theta,m}\boldsymbol{W}_q\boldsymbol{x}_m)^\intercal(\boldsymbol{R}^d_{\Theta,n}\boldsymbol{W}_k\boldsymbol{x}_n) = \boldsymbol{x}^\intercal\boldsymbol{W}_q R^d_{\Theta,n-m}\boldsymbol{W}_k\boldsymbol{x}_n$$

$$\Theta = \theta_i = 10000^{-2i/d}, i \in [1, 2, 3, 4..\frac{d}{2}]$$

引用

- [4] 位置编码详解: https://medium.com/thedeephub/positional-encoding-explained-a-deep-dive-into-transformer-pe-65cfe8cfe10b
- [19] RoPE 论文：https://arxiv.org/abs/2104.09864

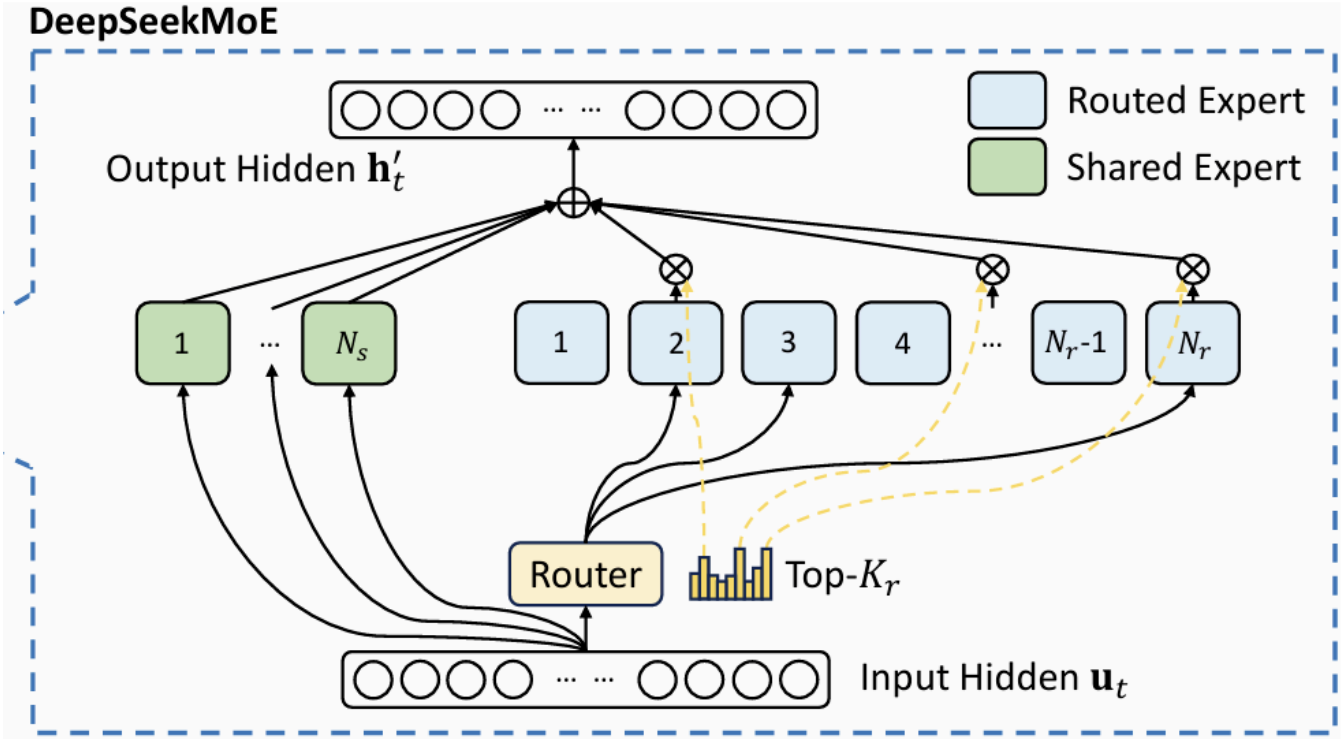
Mixture of Experts (MoE)

DeepSeek 论文中还详细讲解了生产级的 Mixture-of-Experts (MoE) 如何实现。MoE 已经是新一代大模型的标配，能够支持更加巨大的参数和稀疏激活。在 MoE 中，Transformer Block 的 FFE 层被替换为 MoE 层：

- 不同于单个稠密的 FFE 层，MoE 将 FFE **拆分** 成许多更小的 FFE 模块，称作 “**专家**” 。专家可以同时激活，计算时专家并行（Expert Parallelism）。
- 对于一个输入 Token，只有部分专家被激活。这是“**稀疏**”性的所在，相比巨大的总体参数量，训练和推理的成本大幅减少。
- 一些专家总是被激活，它们被称为 **共享专家**。
- MoE 引入了 **路由**（Router），它是 MoE 的关键。路由需要保证负载均衡，为 Token 选择亲和的专家，甚至考虑设备和网络开销。

DeepSeek-V3 的每层 MoE 拥有 256 个专家，输入 Token 将激活 8 个专家以及共享专家，各专家的处理结果被合并输出。

(DeepSeek MoE 架构 ，出自 DeepSeek-V3 论文。)



MoE 是如何进行路由的呢？见下图公式：

- u_t 表示输入 Token ， h_t' 表示 MoE 的输出。
- 每个专家拥有一个 **质心向量**（Centroid），记作 e_i 。质心向量与输入 Token 的相似度代表输入与专家的亲和成都。亲和度最高的 Top-K 个专家被激活，称作门控 g_i,t 。
- 质心向量是和专家被一起训练出来的。实际的门控还需要应对负载偏斜，添修正值和对 **负载不均衡** 的惩罚。见 DeepSeek-V3 论文的公式 (16) ~ (20)。
- 由上可以看到，路由能够被表达成矩阵运算加上激活函数（Sigmoid）。也就是说，路由模块也是一个（非常小的）**神经网络**，专家质心是其可以训练的参数，出口可以用 Softmax 来抽样激活专家。
- 在 h_t' 的公式中，专家的输出被直接相加获得结果。FFN_i^(s) 是共享专家，它们没有门控。此外，输入 u_t 也被相加，这是 **残差连接**。

MoE 在如何使专家负载均衡上常常是难点。Expert Collapse（少数专家获得大部分令牌）、Expert Starvation（某些专家训练不足），是路由训练的常见问题。另外也可以发现，MoE 有利于减少激活数量、减少计算、专家并行，但并不能直接减少内存消耗，因为专家都需要被加载。

(DeepSeek MoE 的公式，出自 DeepSeek-V3 论文。)

$$\mathbf{h}_t' = \mathbf{u}_t + \sum_{i=1}^{N_s} \mathrm{FFN}_i^{(s)}(\mathbf{u}_t) + \sum_{i=1}^{N_r} g_{i,t}\,\mathrm{FFN}_i^{(r)}(\mathbf{u}_t),$$

$$g_{i,t} = \frac{g'_{i,t}}{\sum_{j=1}^{N_r} g'_{j,t}},$$

$$g'_{i,t} = \begin{cases} s_{i,t}, & s_{i,t} \in \mathrm{Topk}(\{s_{j,t} | 1 \leqslant j \leqslant N_r\}, K_r), \\ 0, & \text{otherwise}, \end{cases}$$

$$s_{i,t} = \mathrm{Sigmoid}\left(\mathbf{u}_t{}^T \mathbf{e}_i\right),$$

为什么对 FFN 层使用 MoE，而不是去切分 Attention 层呢？一方面，FFN 层一个一个处理 Token，更易切分，而 Attention 在处理 Token 对。另一方面，FFN 层占据了 Transformer 的大部分参数，远比 Attention 多。

(DeepSeek MoE 的参数数量，链接：https://mp.weixin.qq.com/s/WXrgxV3LgYYvRLPTVzLkjw)

Block	单层参数量	层数	累计参数
MLA	187.17M	61	11.41B
DenseMLP	396.41M	3	1.19B
Expert	44.05Mx(256_routed+1_shared)	58	656.6B
Gate	1.83M	58	106.14M
Embedding	926.67M	1	926.67M
Output	926.67M	1	926.67M
SUM	-	-	671.16B

引用

- [20] Deepseek技术解读(3)-MoE的演进之路: https://zhuanlan.zhihu.com/p/18565423596
- [21] DeepSeek-V3/R1推理效率分析: https://mp.weixin.qq.com/s/WXrgxV3LgYYvRLPTVzLkjw

总结

全文完。到这里，我们介绍了 Transformer 的基础知识，从模型架构、神经网络层、Attention、到 KV Cache ，也介绍了更深入的 MLA 和 MoE 。DeepSeek 论文详细介绍了大模型从模型到基础设施的方方面面，非常值得阅读。而 Polo Club 的可视化 Transformer 可以动态交互，非常方便学习。

Accela推箱子

Like the Author