

Attention Is All You Need

目录

1. 整体架构	5
2. 问题背景：为什么需要 Transformer?	6
2.1. 序列建模在解决什么问题?	6
2.2. RNN 的问题	7
2.3. Attention 的想法	7
2.4. Attention 具体体现在哪里?	7
2.5. 相比以前，Transformer 去掉了什么?	8
2.6. attention 其实不是全部	9
3. Attention 的基本直觉	9
3.1. 从查字典理解 Query、Key、Value	9
3.2. 一个小例子	9
4. Scaled Dot-Product Attention	10
4.1. 公式	10
4.2. 一个具体矩阵例子	11
4.3. 为什么用点积?	12
4.4. 为什么要除以 $\sqrt{d_k}$?	12
4.5. 为什么有些模型还要对 Q 和 K 单独做 Norm?	15
4.5.1. input_layernorm 为什么不够?	16
4.5.2. QK Norm 和除以 $\sqrt{d_h}$ 是不是重复了?	16
4.5.3. 为什么只归一化 Q/K，不归一化 V?	17
4.5.4. 为什么 q_norm 和 k_norm 是两套参数?	17
4.5.5. 对应到 nano-vllm 的 Qwen3 实现	17
4.6. 为什么要 softmax?	18
4.7. 每一层的 Q/K/V 权重是否相同?	18
4.8. 为什么 Q 和 K 不用同一个权重矩阵?	19
5. Multi-Head Attention	20
5.1. 为什么需要多个 Head?	20
5.2. 公式	21
5.3. 一个 head 到底做了什么?	21
5.4. 多头不是把原始输入机械切片	22
5.5. 为什么不直接用一个更宽的 Head?	22
5.6. Head 越多，计算量就按倍数增加吗?	22
5.7. Multi-Head 的工程实现	22
5.8. 一个小尺寸例子	25
5.9. 为什么要有输出投影 W^O ?	26
6. 残差连接	26
6.1. 残差连接在 Transformer 里长什么样?	26
6.2. 为什么需要残差连接?	26
6.3. 一个直观例子	27
6.4. Post-LN 和 Pre-LN	27
7. Norm	28
7.0.1. 为什么 Transformer 需要 Norm?	28

7.1.	BatchNorm	28
7.2.	LayerNorm	29
7.2.1.	LayerNorm 在归一化什么?	29
7.2.2.	从 Post-Norm 到 Pre-Norm	29
7.2.3.	LayerNorm 的数值和工程代价	30
7.3.	RMSNorm	30
7.3.1.	RMSNorm 在做什么?	30
7.3.2.	RMSNorm 为什么适合 LLM?	31
7.3.3.	为什么很多 LLM 也去掉 Linear bias?	31
8.	FFN (MLP)	31
8.1.	传统 FFN 实现	31
8.2.	SwiGLU MLP: 门控前馈网络	32
8.3.	FFN 和 Attention 分工有什么不同?	33
8.4.	为什么 FFN 要先升维再降维?	33
8.5.	FFN 的位置为什么在 Attention 后面?	34
8.6.	FFN 也参与预测下一个词	34
9.	从 Attention 输出到预测下一个词	34
9.1.	Temperature 和采样	37
9.2.	Gumbel-Max Trick	37
9.3.	LLM 为什么知道何时停下来?	38
10.	Self-Attention	38
10.1.	什么是 Self-Attention?	38
10.2.	Self-Attention 在学习什么?	39
10.3.	Self-Attention 的优点	40
11.	位置编码	40
11.1.	为什么需要位置编码?	40
11.2.	正弦位置编码	40
11.2.1.	为什么 Embedding 要乘以 $\sqrt{d_{\text{model}}}$?	41
11.2.2.	为什么用正弦和余弦?	42
11.3.	RoPE: 把位置写进 Q/K 的旋转角度	42
11.3.1.	为什么是二维一组地旋转?	43
11.3.2.	为什么点积会自然得到相对位置?	43
11.3.3.	用复数理解 RoPE	44
11.3.4.	RoPE 在一层 Attention 中放在哪里?	44
11.3.5.	RoPE、causal mask 和 KV Cache 的关系	44
11.3.6.	与正弦位置编码的关系	45
11.3.7.	RoPE 不等于无限长度外推	45
11.4.	RoPE 小结	45
12.	Transformer Encoder	45
12.1.	Encoder 在整个模型里的角色	45
12.2.	Encoder 层结构	46
12.3.	残差连接有什么用?	48
12.4.	FFN 做什么?	48
12.5.	Encoder 输出给谁用?	48
13.	Transformer Decoder	48
13.1.	Decoder 在整个模型里的角色	48

13.2. 原论文 Decoder 层结构	49
13.3. Masked Self-Attention	49
13.4. Encoder-Decoder Attention	51
13.5. Decoder-only 模型和原论文 Decoder 的区别	51
14. 训练过程	52
14.1. Q、K、V 是如何训练出来的?	52
14.2. 数据切分后, 权重矩阵如何合并?	53
14.3. 为什么训练可以并行?	55
15. 高效 Attention 机制	56
15.1. Linear Attention	57
15.2. FlashAttention	58
15.3. FlashAttention 和 CUTLASS 矩阵分块是什么关系?	60
15.4. Linear Attention 和 FlashAttention 的区别	61
16. Multi-Head Latent Attention (MLA)	62
16.1. 动机: KV Cache 为什么是瓶颈?	62
16.2. MQA/GQA 的局限: 共享不等于压缩	62
16.3. MLA 的核心思路: 压缩而非共享	62
16.4. 公式推导	63
16.4.1. 输入和符号约定	63
16.4.2. 第一步: 压缩 (Down-Projection)	63
16.4.3. 第二步: 解压 (Up-Projection)	63
16.4.4. 第三步: Query 投影和注意力计算	64
16.4.5. 推理时的关键优化: 矩阵吸收	64
16.4.6. 位置编码的难题: Decoupled RoPE	65
16.5. MLA 的 KV Cache 如何工作?	66
16.5.1. Prefill: 一次生成并写入整段 Prompt 的 Cache	67
16.5.2. Decode: 每步只追加一条 Cache Record	67
16.5.3. 推理 Backend 不一定采用同一种计算路径	68
16.5.4. Cache 的分配、分页和释放	69
16.5.5. 每一步读写多少数据?	69
16.6. KV Cache 大小对比	69
16.7. 训练时如何运作?	70
16.8. 总结	70
17. 从 vLLM 的角度分析	71
17.1. 为什么 KV 可以 Cache?	71
17.2. 为什么不 Cache Q?	71
17.2.1. QK^T 会不会占用 $O(T^2)$ 的 KV Cache?	72
17.2.2. Prefill 阶段的临时 Attention 矩阵	72
17.2.3. Decode 阶段只计算新的一行	72
17.2.4. 为什么 QK^T 也不值得缓存?	73
17.2.5. 存储复杂度和计算复杂度	73
17.3. 推理为什么要分成 Prefill 和 Decode?	75
17.4. Prefill 阶段到底在做什么?	77
17.5. Decode 阶段到底在做什么?	77
17.6. Prefill 和 Decode 的计算量差异	78
17.7. vLLM 的 PagedAttention 解决什么?	78

17.8. KV cache 会不会不命中?	80
18. Softmax	81
18.1. Softmax 为什么需要 safe 写法?	83
18.2. Online Softmax	83
18.3. LSE 为什么有用?	86
19. AWQ: 激活感知的权重量化	87
19.1. 动机: 为什么需要量化?	87
19.2. 量化基础	87
19.2.1. 最朴素的量化 (RTN)	87
19.2.2. GPTQ 的思路	87
19.3. AWQ 的核心观察	88
19.4. AWQ 的方法: 通道级缩放	88
19.5. AWQ 和 GPTQ 的对比	89
19.6. 为什么 AWQ 有效?	89
19.7. 小结	90
20. 常见误区	90
21. 总结	90
21.1. Infra 优化的共同模式	91

1. 整体架构

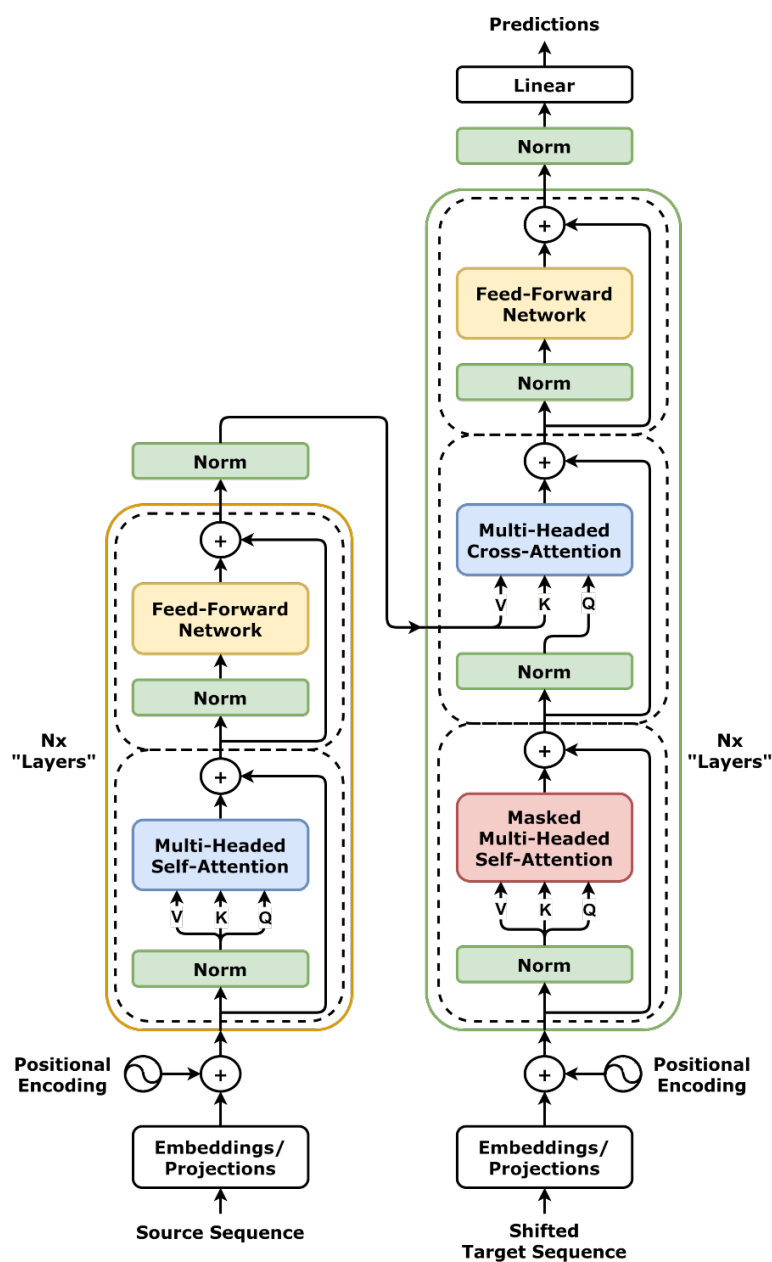


Figure 1: transformer 架构

上图展示的是原始 Transformer 的 encoder-decoder 架构；下面先以主流 decoder-only LLM 为例说明单层结构。

以主流 decoder-only LLM（如 LLaMA、GPT 类）为例，一层 Transformer 通常长这样：

输入隐藏状态 x

- ├─ RMSNorm / LayerNorm
- ├─ 多头因果自注意力 (Causal Self-Attention)
- ├─ 残差连接: $x \leftarrow x + \text{Attention}(\dots)$
- ├─ RMSNorm / LayerNorm

```

|
├─ 前馈网络 (MLP / FFN)
└─ 残差连接:  $x \leftarrow x + \text{MLP}(\dots)$ 
输出隐藏状态  $x$ 

```

用公式写得更紧凑一些：

$$h = x + \text{Attention}(\text{Norm}(x)) \quad (1)$$

$$y = h + \text{MLP}(\text{Norm}(h)) \quad (2)$$

这就是最常见的 **Pre-Norm Transformer block**。

注意力部分负责 “token 之间交流”。每个 token 会生成 Query、Key、Value：

$$Q = xW_Q, \quad K = xW_K, \quad V = xW_V \quad (3)$$

然后计算：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + \text{mask}\right)V \quad (4)$$

其中 causal mask 是加到 attention 分数上的掩码：允许的位置取 0，未来位置取负无穷，从而保证当前位置只能看自己和前面的 token，不能偷看未来文本。多头注意力则是并行做多组这样的注意力，让不同头学习不同关系，例如指代、句法、长距离依赖等。现代模型还常用 GQA/MQA 来减少 KV cache 和推理成本。

此外，进入第一层前，token 会先经过：

```

文本 → tokenizer → token ID → embedding
      → 将位置信息编码到表示中（现代 LLM 常在 Q/K 上使用 RoPE）
      → Transformer 第 1 层 ... 第 N 层
      → 最终 Norm → 词表线性层 → 下一个 token 概率

```

直观地说：

- Attention：决定 “该从上下文里参考什么”；
- MLP：将参考到的信息做更复杂的特征转换；
- Norm：稳定每层的数据尺度；
- Residual：保留旧信息并让深层网络容易训练。

2. 问题背景：为什么需要 Transformer？

2.1. 序列建模在解决什么问题？

自然语言是序列。一个词的含义通常依赖前后文。例如：

1. “苹果发布了新手机”
2. “我吃了一个苹果”

同一个 “苹果” 在两个句子中含义不同。模型必须根据上下文判断它到底指公司还是水果。

机器翻译也是典型序列到序列任务。输入是源语言句子：

I love machine learning

输出是目标语言句子：

我 喜欢 机器 学习

模型要做的事情不是逐词查字典，而是理解整个输入序列，然后逐步生成输出序列。

2.2. RNN 的问题

Transformer 之前，序列建模常用 RNN、LSTM、GRU。它们的思想是从左到右读序列：

$$h_t = f(h_{t-1}, x_t) \quad (5)$$

其中 x_t 是第 t 个输入， h_t 是读到当前位置后的隐藏状态。

这个结构有一个自然问题：它必须按顺序计算。要得到 h_t ，必须先算完 h_{t-1} ；要算 h_{t-1} ，又必须先算 h_{t-2} 。因此 RNN 很难并行。

另一个问题是长距离依赖。假设句子很长，第 100 个词要依赖第 3 个词的信息，那么信息必须经过很多步状态传递。即使用 LSTM 缓解，长距离信息仍然容易衰减或混杂。

例 2.2.1: 句子：

The book that the professor recommended yesterday is very useful.

“is” 的主语是 “book”，不是离它更近的 “professor”。模型需要跨过中间从句建立远距离关系。

2.3. Attention 的想法

Attention 的思想是：不要强迫信息一步一步传递，而是让每个位置直接查看序列中的所有位置。

如果当前词需要理解上下文，它可以直接问：

1. 哪些词和我最相关？
2. 每个相关词应该占多大权重？
3. 我应该从这些词那里取回哪些信息？

这就是 attention 的核心。

RNN 像是在一条链上逐站传话；attention 像是在一个会议室里，每个人可以直接看向所有人，并决定听谁说话。

2.4. Attention 具体体现在哪里？

Attention 具体体现在这个矩阵：

$$A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \quad (6)$$

这里的 A 就是 attention 权重矩阵。如果序列有 n 个 token，那么：

$$A \in \mathbb{R}^{n \times n} \quad (7)$$

其中 $A_{i,j}$ 表示：

$$\text{第 } i \text{ 个 token 应该从第 } j \text{ 个 token 那里拿多少信息} \quad (8)$$

所以 attention 的本质是：每个 token 动态决定自己要关注哪些 token，以及关注多少。

最终输出是：

$$O = AV \quad (9)$$

也就是说，模型先用 QK^T 算出“谁应该关注谁”，再用这个权重矩阵去加权 V ，得到每个 token 融合上下文之后的新表示。

例如句子：

The animal didn't cross the street because it was tired.

当模型处理“it”时，attention 权重可能让它更多关注“animal”，少关注“street”。这个“更多关注谁、少关注谁”的权重分配，就是 attention 在模型里的具体体现。

2.5. 相比以前，Transformer 去掉了什么？

Transformer 之前，序列建模常见方案是 RNN、LSTM、GRU，或者 CNN。

RNN 类模型依赖循环递推：

$$h_t = f(h_{t-1}, x_t) \quad (10)$$

这意味着第 t 个位置的表示依赖第 $t-1$ 个位置的隐藏状态。信息需要沿着时间步一步一步传递。要处理第 100 个 token，通常必须先处理完前 99 个 token。

Transformer 去掉了这种循环结构。它不再要求：

先处理第 1 个 token
再处理第 2 个 token
再处理第 3 个 token
...

而是把整句话表示成矩阵 X ，一次性计算所有 token 之间的关系：

$$Q = XW^Q \quad (11)$$

$$K = XW^K \quad (12)$$

$$V = XW^V \quad (13)$$

$$A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \quad (14)$$

这样任意两个 token 可以在一层 attention 中直接交互。

CNN 类模型依赖局部卷积窗口。一个 token 先看到附近几个 token，堆很多层之后，远处 token 的信息才能逐渐传过来。

Transformer 不再依赖卷积作为序列信息传递的核心机制。它用 attention 直接构造全局的 token-to-token 关系矩阵 A 。只要 mask 允许，第 i 个 token 可以直接从第 j 个 token 取信息，不需要通过很多层局部卷积慢慢传递。

所以论文标题可以更直白地理解为：

做序列建模时，token 之间的信息交换不一定需要循环，也不一定需要卷积；attention 就可以承担这个核心角色。

因此，Transformer 不是去掉了所有神经网络组件，而是去掉了 RNN/CNN 作为序列建模主干的地位：

1. 去掉 RNN/LSTM/GRU 的循环递推结构

2. 去掉 CNN 的局部卷积作为核心序列交互机制
3. 用 self-attention 作为 token 之间信息传递的主机制

这也是为什么 Transformer 的训练并行性更好：它不需要沿时间步一个一个递推，而是把序列内部的大量关系变成矩阵运算。

2.6. attention 其实不是全部

这个标题的意思不是模型里真的只有 attention。Transformer 还有 embedding、位置编码、FFN、残差连接、LayerNorm 等组件。

它真正想强调的是：序列中 token 之间的信息交互，不再需要 RNN 的循环结构，也不再依赖 CNN 的局部卷积结构，仅靠 attention 就可以完成。

Attention 替代了 RNN/CNN 作为序列建模的核心机制。

3. Attention 的基本直觉

3.1. 从查字典理解 Query、Key、Value

Attention 中最容易困惑的是 Query、Key、Value。可以把它理解成查字典或检索系统。

假设你在图书馆找资料：

1. **Query**：你提出的问题，比如“我要找和机器翻译有关的资料”
2. **Key**：每本书的索引标签，比如“自然语言处理”“神经网络”“机器翻译”
3. **Value**：书的真实内容

检索过程是：

1. 用 Query 和每个 Key 比较相似度
2. 相似度越高，对应 Value 权重越大
3. 把所有 Value 按权重加权求和，得到最终结果

Transformer 中每个 token 都会生成自己的 Query、Key、Value。一个 token 想更新自己表示时，会拿自己的 Query 去和所有 token 的 Key 比较，然后从所有 token 的 Value 中加权取信息。

定义 3.1.1：在 self-attention 中，同一组输入向量会被线性变换成三组向量：

1. Query：当前位置想找什么信息
2. Key：每个位置提供什么索引
3. Value：每个位置真正携带的信息内容

3.2. 一个小例子

考虑句子：

The animal didn't cross the street because it was tired.

这里“it”指的是“animal”，不是“street”。当模型处理“it”时，它应该更多关注“animal”。

在 attention 中，“it” 的 Query 会和所有词的 Key 做相似度计算。如果训练得好，“animal” 的 Key 会和 “it” 的 Query 更匹配，于是 “animal” 的 Value 会获得更高权重。最终 “it” 的新表示就会融入 “animal” 的信息。

这就是 attention 解决指代、依赖、上下文关系的方式。

4. Scaled Dot-Product Attention

4.1. 公式

Transformer 使用的 attention 形式叫 Scaled Dot-Product Attention：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (16)$$

这里：

1. Q 是 Query 矩阵
2. K 是 Key 矩阵
3. V 是 Value 矩阵
4. d_k 是 Key 向量的维度

这些矩阵不是手工构造的，而是从输入 token 表示线性投影出来的。设输入矩阵为 $X \in \mathbb{R}^{n \times d_{\text{model}}}$ ，其中每一行是一个 token 的向量表示，则：

$$Q = XW^Q \quad (17)$$

$$K = XW^K \quad (18)$$

$$V = XW^V \quad (19)$$

这里的 X 表示进入当前 attention 层之前，整句话中所有 token 的当前表示。原始 Transformer 把词向量和绝对位置编码相加：

$$X = \sqrt{d_{\text{model}}} \text{TokenEmbedding} + \text{PositionEncoding} \quad (20)$$

现代 decoder-only LLM 常采用 RoPE。此时通常不把位置向量直接加到 X 上，而是在每一层 attention 内，把位置编码为 Query 和 Key 的旋转；Value 通常不旋转。后面的“RoPE”小节会详细说明。

如果一句话有 n 个 token，每个 token 的隐藏维度是 d_{model} ，那么 X 就有 n 行、 d_{model} 列。第 i 行就是第 i 个 token 当前携带的信息。

这里的“隐藏维度”可以理解为：模型内部用多少个数字表示一个 token。比如一句话有 3 个 token，每个 token 用 4 个数字表示，那么：

$$X \in \mathbb{R}^{3 \times 4} \quad (21)$$

这里 $n = 3$ ， $d_{\text{model}} = 4$ 。真实 Transformer 中 d_{model} 往往更大，例如原始 Transformer base 使用 $d_{\text{model}} = 512$ ，现代大模型中可能是几千维。

之所以叫“隐藏”，不是因为模型故意藏起了某个具体东西，而是因为它属于模型内部的中间表示：它既不是用户直接输入的 token，也不是模型最终输出的词表概率、分类结果或翻译结果。

这些维度也不是人工规定的特征。它们不是明确的“是否名词”“是否动词”这类标签，而是模型在训练中自己学出来的内部表示。某一维可能混合了词义、语法、位置和上下文关系等信息，通常不能简单对应到一个人类可命名的概念。

为什么要乘以 W^Q ？因为模型需要把同一个 token 表示变成适合做 Query 的表示。设：

$$W^Q \in \mathbb{R}^{d_{\text{model}} \times d_k} \quad (22)$$

$$W^K \in \mathbb{R}^{d_{\text{model}} \times d_k} \quad (23)$$

$$W^V \in \mathbb{R}^{d_{\text{model}} \times d_v} \quad (24)$$

因此 W^Q 和 W^K 的形状通常一样，因为 Query 和 Key 后面要做点积，二者必须有相同维度。 W^V 的形状可以不同，因为 Value 不参与 QK^T 的相似度计算，它是在 attention 权重算出来之后被加权求和。

很多实现会取 $d_k = d_v$ ，这时 W^Q, W^K, W^V 的形状都一样。例如原始 Transformer base 中 $d_{\text{model}} = 512$ ，head 数量 $h = 8$ ，每个 head 通常使用 $d_k = d_v = 64$ ，所以每个 head 里的三个投影矩阵形状都是 512×64 。

但形状一样不代表它们是同一个矩阵。 W^Q, W^K, W^V 是三组独立训练的参数，分别负责生成 Query、Key、Value。

4.2. 一个具体矩阵例子

假设有 3 个 token，Query/Key/Value 都用二维向量表示：

$$Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \quad K = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \quad V = \begin{pmatrix} 10 & 0 \\ 0 & 10 \\ 10 & 10 \end{pmatrix} \quad (25)$$

先计算 Query 和 Key 的匹配分数：

$$QK^T = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 2 \end{pmatrix} \quad (26)$$

第一行 $[1, 0, 1]$ 的意思是：第 1 个 token 和第 1、3 个 token 更匹配，和第 2 个 token 不太匹配。

因为 $d_k = 2$ ，所以要除以 $\sqrt{2}$ 。对第一行做 softmax：

$$\text{softmax}\left(\left[\frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}\right]\right) \approx [0.40, 0.20, 0.40] \quad (27)$$

最后用这些权重加权 Value：

$$0.40[10, 0] + 0.20[0, 10] + 0.40[10, 10] = [8, 6] \quad (28)$$

所以第 1 个 token 的输出向量大约是 $[8, 6]$ 。它保留了自己 Value 的信息，也吸收了第 3 个 token 的信息，并少量吸收第 2 个 token 的信息。

这个 $[8, 6]$ 不是最终预测结果，而是第 1 个 token 更新后的中间表示。Attention 的作用就是把一个 token 原本只包含自身信息的表示，改造成融合上下文信息的新表示。

换句话说，第 1 个 token 原来的 Value 是 $[10, 0]$ ，经过 attention 后变成 $[8, 6]$ 。这个新向量会继续传给后面的模块，例如输出投影、残差连接、LayerNorm、FFN，或者下一层

Transformer。经过多层处理后，模型才会用最终表示去做翻译、分类、预测下一个 token 等任务。

所以在这个例子中，[8, 6] 可以理解为：

第 1 个 token 看完整个序列之后得到的上下文增强版表示 (29)

真实模型中的向量通常不是二维，而可能是几十维、几百维甚至几千维。这里用二维只是为了方便手算。

4.3. 为什么用点积？

两个向量点积越大，通常表示方向越接近。在线性代数里：

$$\vec{q} \cdot \vec{k} = \|\vec{q}\| \|\vec{k}\| \cos \theta \quad (30)$$

如果两个向量方向接近， $\cos \theta$ 大，点积就大。Attention 用点积作为 Query 和 Key 的匹配分数，本质上是在问：“当前问题和这个索引有多匹配？”

4.4. 为什么要除以 $\sqrt{d_k}$ ？

严格说，公式里除的是 $\sqrt{d_k}$ ，不是 d_k ：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (31)$$

原因是：当 Query 和 Key 的维度变大时，点积的数值尺度会自然变大。

假设一个 Query 向量和一个 Key 向量都是 d_k 维：

$$q = [q_1, q_2, \dots, q_{d_k}] \quad (32)$$

$$k = [k_1, k_2, \dots, k_{d_k}] \quad (33)$$

它们的点积是：

$$q \cdot k = \sum_{r=1}^{d_k} q_r k_r \quad (34)$$

下面给一个更具体的推导。假设每一维的数值大致独立，并且：

$$E(q_r) = E(k_r) = 0 \quad (35)$$

$$\text{Var}(q_r) = \text{Var}(k_r) = 1 \quad (36)$$

因为均值为 0、方差为 1，所以：

$$E(q_r^2) = 1, \quad E(k_r^2) = 1 \quad (37)$$

先看点积中的一项 $q_r k_r$ 。由于 q_r 和 k_r 独立：

$$E(q_r k_r) = E(q_r) E(k_r) = 0 \quad (38)$$

它的方差是：

$$\text{Var}(q_r k_r) = E((q_r k_r)^2) - E(q_r k_r)^2 \quad (39)$$

$$= E(q_r^2 k_r^2) - 0 \quad (40)$$

$$= E(q_r^2) E(k_r^2) \quad (41)$$

$$= 1 \quad (42)$$

也就是说，每一项 $q_r k_r$ 的方差大约是 1。

点积是 d_k 项相加：

$$q \cdot k = \sum_{r=1}^{d_k} q_r k_r \quad (43)$$

如果这些项之间也近似独立，那么和的方差等于方差之和：

$$\text{Var}(q \cdot k) \approx d_k \quad (44)$$

标准差就是：

$$\text{Std}(q \cdot k) \approx \sqrt{d_k} \quad (45)$$

这就是为什么维度越大，点积的典型波动范围大约按 $\sqrt{d_k}$ 变大。

现在把点积除以 $\sqrt{d_k}$ ：

$$\text{Var}\left(\frac{q \cdot k}{\sqrt{d_k}}\right) = \frac{\text{Var}(q \cdot k)}{d_k} \quad (46)$$

$$\approx \frac{d_k}{d_k} \quad (47)$$

$$= 1 \quad (48)$$

因此除以 $\sqrt{d_k}$ 可以把 attention 分数的方差重新拉回到 1 附近，让 softmax 前的数值尺度比较稳定。

如果不缩放， d_k 大时点积分数会变得很大。分数太大时，softmax 会变得非常尖锐：最大的那个位置接近 1，其他位置接近 0。

这个现象可以直接从 softmax 公式看出来。先看只有两个候选位置的情况，分数为 s_1 和 s_2 。

第 1 个位置的 softmax 权重是：

$$p_1 = \frac{\exp(s_1)}{\exp(s_1) + \exp(s_2)} \quad (49)$$

分子分母同时除以 $\exp(s_1)$ ：

$$p_1 = \frac{1}{1 + \exp(s_2 - s_1)} \quad (50)$$

令两者差值为：

$$\Delta = s_1 - s_2 \quad (51)$$

则：

$$p_1 = \frac{1}{1 + \exp(-\Delta)} \quad (52)$$

这就是 sigmoid 形式。它说明 softmax 真正关心的是分数差 Δ 。

如果不除以 $\sqrt{d_k}$ ，点积分数的典型尺度会变大。可以把它理解成把原本稳定尺度下的分数差放大了系数 α ：

$$\Delta' = \alpha \Delta \quad (53)$$

其中 α 大约和 $\sqrt{d_k}$ 同级。此时：

$$p'_1 = \frac{1}{1 + \exp(-\alpha \Delta)} \quad (54)$$

当 $\Delta > 0$ 时，如果 α 变大：

$$\exp(-\alpha \Delta) \rightarrow 0 \quad (55)$$

所以：

$$p'_1 \rightarrow 1 \quad (56)$$

当 $\Delta < 0$ 时，如果 α 变大：

$$\exp(-\alpha \Delta) \rightarrow \infty \quad (57)$$

所以：

$$p'_1 \rightarrow 0 \quad (58)$$

这说明：只要分数有一点差距，放大以后 softmax 就会把概率快速推向 0 或 1。

多项 softmax 也是同样道理。设最大分数是 s_m ，第 m 项的权重是：

$$p_m = \frac{\exp(s_m)}{\sum_j \exp(s_j)} \quad (59)$$

分子分母同时除以 $\exp(s_m)$ ：

$$p_m = \frac{1}{\sum_j \exp(s_j - s_m)} \quad (60)$$

因为 $s_j - s_m \leq 0$ 。如果所有分数整体被放大 α 倍：

$$p'_m = \frac{1}{\sum_j \exp(\alpha(s_j - s_m))} \quad (61)$$

对于非最大项， $s_j - s_m < 0$ ，当 α 很大时：

$$\exp(\alpha(s_j - s_m)) \rightarrow 0 \quad (62)$$

所以分母会越来越接近 1，最大项概率 p'_m 越来越接近 1。其他项的概率则越来越接近 0。

这会带来两个问题：

1. 模型过早只关注少数位置
2. softmax 梯度变小，训练变困难

看一个简单例子。假设某个 token 对三个位置的未缩放 attention 分数是：

$$[8, 4, 0] \quad (63)$$

直接做 softmax：

$$\text{softmax}([8, 4, 0]) \approx [0.982, 0.018, 0.000] \quad (64)$$

这个分布非常尖锐，几乎只看第一个位置。

如果这是 $d_k = 64$ 的 attention head，那么：

$$\sqrt{d_k} = 8 \quad (65)$$

缩放后分数变成：

$$\frac{[8, 4, 0]}{8} = [1, 0.5, 0] \quad (66)$$

再做 softmax：

$$\text{softmax}([1, 0.5, 0]) \approx [0.506, 0.307, 0.186] \quad (67)$$

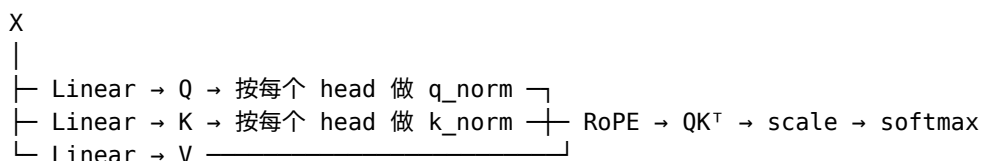
这时模型仍然更关注第一个位置，但不会一开始就把其他位置几乎全部压成 0。

所以除以 $\sqrt{d_k}$ 的作用不是改变谁更相关，而是控制 softmax 前分数的尺度，让权重分布不要因为维度变大而过度尖锐。

可以把它理解成归一化：维度越大，点积自然越容易变大，所以需要按维度规模缩小。

4.5. 为什么有些模型还要对 Q 和 K 单独做 Norm？

input_layernorm 已经归一化了 attention 的输入，有些现代模型（例如 Qwen3）仍会在 Q/K 投影之后分别增加 q_norm 和 k_norm。这通常称为 **QK Norm**：



设第 t 个 token、单个 attention head 的投影结果为：

$$q_t = x_t W^Q \in \mathbb{R}^{d_h}, \quad k_t = x_t W^K \in \mathbb{R}^{d_h} \quad (68)$$

QK Norm 对每个 token 的每个 head 分别沿 d_h 个特征维度做 RMSNorm：

$$|(q)_t = \gamma_Q \odot \frac{q_t}{\sqrt{\frac{1}{d_h} \sum_{r=1}^{d_h} q_{t,r}^2 + \varepsilon}} \quad (69)$$

$$|(k)_t = \gamma_K \odot \frac{k_t}{\sqrt{\frac{1}{d_h} \sum_{r=1}^{d_h} k_{t,r}^2 + \varepsilon}} \quad (70)$$

然后才计算 attention 分数。带 RoPE 时，对应顺序通常是：

$$\tilde{q}_t = R_t |(q)_t, \quad \tilde{k}_s = R_s |(k)_s \quad (71)$$

$$\text{score}(t, s) = \frac{\tilde{q}_t^T \tilde{k}_s}{\sqrt{d_h}} \quad (72)$$

这里 γ_Q 和 γ_K 是两套独立的可训练缩放参数。忽略 ε 且暂时令 $\gamma_Q = \gamma_K = 1$ 时，归一化后的向量满足：

$$\|(q)_t\|_2 \approx \sqrt{d_h}, \quad \|(k)_s\|_2 \approx \sqrt{d_h} \quad (73)$$

所以 Q/K 向量本身任意变大，不再能无约束地放大 attention logits；匹配结果会更主要地由向量方向决定。严格说它并不是纯 cosine attention，因为仍有可学习的 γ_Q, γ_K ，模型可以按维度重新调节重要性。

4.5.1. input_layernorm 为什么不够？

input_layernorm 归一化的是投影之前的整个 token 隐藏向量：

$$x_t \in \mathbb{R}^{d_{\text{model}}} \quad (74)$$

但接下来还要乘以独立学习的 W^Q 和 W^K ：

$$q_t = \text{Norm}(x_t)W^Q, \quad k_t = \text{Norm}(x_t)W^K \quad (75)$$

即使 x_t 的尺度稳定，也不能保证投影后的 q_t 和 k_t 尺度稳定。原因包括：

1. W^Q 和 W^K 可以沿某些方向产生很大的放大倍数。
2. 不同 token 落在投影矩阵的不同方向上，投影后的范数仍可能差很多。
3. 不同 head 有各自的投影子空间，某些 head 的 Q/K 尺度可能比其他 head 更大。
4. 训练过程中权重持续更新，Q/K 的尺度可能逐渐漂移。

而 softmax 的直接输入恰好是 Q/K 点积。若某个 Query 整体放大 10 倍、某个 Key 也放大 10 倍，那么它们的点积会放大 100 倍：

$$(10q)^T(10k) = 100q^T k \quad (76)$$

这会让 softmax 变得极尖，容易出现接近 one-hot 的 attention 和很小的梯度。

input_layernorm 只能稳定投影的输入，QK Norm 则直接在点积发生之前稳定两个操作数。

因此三者作用层级不同：

Norm	归一化位置与维度	主要目的
input_layernorm	QKV 投影前； d_{model}	稳定 attention 子层的输入和深层残差训练
q_norm / k_norm	Q/K 投影后；每个 head 的 d_h	直接稳定 softmax 前的 QK 点积分
post_attention_layernorm	attention 残差之后； d_{model}	稳定送入 MLP 的隐藏状态

Table 1: Transformer block 中三类 Norm 的作用层级

4.5.2. QK Norm 和除以 $\sqrt{d_h}$ 是不是重复了？

不是。两者处理的尺度来源不同：

1. 除以 $\sqrt{d_h}$ 是一个固定缩放，修正“ d_h 项随机乘积相加”导致的点积方差随维度增长。
2. QK Norm 是逐 token、逐 head 的动态归一化，修正投影权重和当前输入导致的 Q/K 范数漂移。

即使使用 QK Norm，随机方向的两个 d_h 维向量做点积时，其典型波动仍随 $\sqrt{d_h}$ 变化，所以 attention 通常仍保留 $\frac{1}{\sqrt{d_h}}$ 。可以把二者分别理解为：

QK Norm：控制参加比赛的两支队伍各自有多大

$1/\text{sqrt}(d_h)$ ：控制比赛项目数量增加带来的总分自然增长

4.5.3. 为什么只归一化 Q/K，不归一化 V？

Q 和 K 决定 softmax 前的 logits：

$$S = \frac{QK^T}{\sqrt{d_h}} \quad (77)$$

它们的尺度会通过指数函数改变“关注谁”和 attention 分布的尖锐程度。V 不参与这一步；它只在权重确定后被线性加权：

$$O = \text{softmax}(S)V \quad (78)$$

因此 QK Norm 首先针对最敏感的 softmax 输入。强制归一化 V 还可能限制 Value 用向量幅度表达内容强弱；attention 输出的尺度可以由输出投影、残差结构和后续 Norm 继续处理。这不是说 V 在任何模型中都绝不能归一化，而是标准 QK Norm 要解决的问题主要出现在 Q/K 点积上。

4.5.4. 为什么 q_norm 和 k_norm 是两套参数？

Query 表示“当前 token 想找什么”，Key 表示“当前 token 如何被别人匹配”。二者由不同投影产生，分布和角色都不相同，因此通常各自拥有可训练缩放参数：

$$\gamma_Q \neq \gamma_K \quad (79)$$

分别归一化不会让 Q 和 K 变成相同向量，也不会让不同 head 互相混合。它只是分别控制 Q 空间和 K 空间的数值尺度；真正的 Query-Key 交互仍然发生在 QK^T 中。

4.5.5. 对应到 nano-vllm 的 Qwen3 实现

Qwen3Attention.forward() 中的顺序是：

```
q = q.view(-1, self.num_heads, self.head_dim)
k = k.view(-1, self.num_kv_heads, self.head_dim)
v = v.view(-1, self.num_kv_heads, self.head_dim)

q = self.q_norm(q)
k = self.k_norm(k)
q, k = self.rotary_emb(positions, q, k)
o = self.attn(q, k, v)
```

例如 `q.shape = [token_num, num_heads, head_dim]`，`RMSNorm(self.head_dim)` 总是归一化最后一个维度。因此它会独立处理每个 token 的每个 Q head；`k_norm` 同样独立处理每个 KV head。它们既不会跨 token 计算统计量，也不会把不同 head 混在一起。这里“逐 head 归一化”不表示每个 head 都有一套独立参数：当前实现只有一个长度为 `head_dim` 的 `q_norm.weight` 供所有 Q head 复用，所有 K head 则复用另一套 `k_norm.weight`。

代码目前用 `if not self.qkv_bias` 同时选择“无 QKV bias”和“启用 QK Norm”的 Qwen3 路径。需要注意，无 bias 和 QK Norm 在数学上不是因果关系；这是该实现区分所支持模型配置的工程写法，不应理解成“只要 Linear 没有 bias，就必须增加 QK Norm”。

4.6. 为什么要 softmax?

点积得到的是原始分数，不一定为正，也不一定和为 1。softmax 把这些分数变成权重：

1. 每个权重非负
2. 所有权重加起来等于 1
3. 分数越大，权重越大

所以 attention 输出其实是 Value 的加权平均：

$$\text{output}_i = \sum_j \alpha_{ij} \vec{v}_j \quad (80)$$

其中 α_{ij} 表示第 i 个 token 对第 j 个 token 的关注权重。

例 4.6.1: 当模型处理 “it” 时，假设它对三个词的权重是：

1. animal: 0.70
2. street: 0.10
3. tired: 0.20

那么 “it” 的新表示主要来自 “animal”，同时也吸收少量其他上下文信息。

4.7. 每一层的 Q/K/V 权重是否相同?

标准 Transformer 中，不同层的 Q/K/V 权重矩阵通常不是同一套参数。第 l 层有自己的投影矩阵：

$$W_l^Q, W_l^K, W_l^V \quad (81)$$

如果第 l 层 attention 的输入是 X_l ，那么这一层计算：

$$Q_l = X_l W_l^Q \quad (82)$$

$$K_l = X_l W_l^K \quad (83)$$

$$V_l = X_l W_l^V \quad (84)$$

下一层会有另一组参数：

$$W_{l+1}^Q, W_{l+1}^K, W_{l+1}^V \quad (85)$$

对应计算：

$$Q_{l+1} = X_{l+1} W_{l+1}^Q \quad (86)$$

$$K_{l+1} = X_{l+1} W_{l+1}^K \quad (87)$$

$$V_{l+1} = X_{l+1} W_{l+1}^V \quad (88)$$

所以，通常应该理解为：

1. 同一层内， W_l^Q, W_l^K, W_l^V 是三组不同参数。
2. 不同层之间，Q/K/V 投影矩阵也各自独立。
3. 多头注意力中，不同 head 通常也有各自的 Q/K/V 投影。

工程实现里经常把同一层、所有 head 的 Q/K/V 投影合并成一个大矩阵来一次性计算。例如把 W_l^Q, W_l^K, W_l^V 拼成一个矩阵，得到：

$$[Q_l, K_l, V_l] = X_l W_l^{\text{QKV}} \quad (89)$$

这只是为了计算效率，并不表示 Q、K、V 或不同层之间共享同一套权重。

少数模型会做跨层参数共享，例如 ALBERT 会共享部分 Transformer 层参数。但这属于特殊设计，不是标准 Transformer 的默认做法。

直观地说，每一层处理的信息抽象层次不同。底层可能更偏局部词形和短距离关系，高层可能更偏语义、指代和任务相关关系。因此每层通常需要自己的 Q/K/V 投影方式。

那么：

$$Q = XW^Q \in \mathbb{R}^{n \times d_k} \quad (90)$$

$$K = XW^K \in \mathbb{R}^{n \times d_k} \quad (91)$$

$$V = XW^V \in \mathbb{R}^{n \times d_v} \quad (92)$$

也就是说， W^Q 把每个 token 的 d_{model} 维表示变换成 d_k 维的 Query 表示， W^K 生成 Key 表示， W^V 生成 Value 表示。这样矩阵乘法的形状也能对上：

$$(n \times d_{\text{model}})(d_{\text{model}} \times d_k) = n \times d_k \quad (93)$$

由于 $Q \in \mathbb{R}^{n \times d_k}$ ， $K \in \mathbb{R}^{n \times d_k}$ ，所以：

$$QK^T \in \mathbb{R}^{n \times n} \quad (94)$$

矩阵 QK^T 的第 i, j 个元素表示：第 i 个 token 的 Query 和第 j 个 token 的 Key 有多相似。

“Attention” (Q, K, V) 的计算结果仍然是一个矩阵，形状为 $n \times d_v$ 。它的第 i 行是第 i 个 token 更新后的上下文表示：这个表示不是只来自自己，而是根据 attention 权重从所有 token 的 Value 中加权合成出来的。

其中 W^Q, W^K, W^V 是模型训练出来的参数。上标 Q, K, V 只是标签，表示这三个矩阵分别用于生成 Query、Key、Value，并不是数学里的幂运算。也可以把它们写成 W_Q, W_K, W_V 。

为什么不直接用 X 同时当 Query、Key、Value？因为同一个 token 在 attention 中有三种不同角色：

1. Query：当前位置想找什么信息
2. Key：当前位置提供什么索引，方便别人匹配自己
3. Value：当前位置真正提供给别人的内容

这三种角色不一定应该使用同一个向量空间，所以 Transformer 用三组不同的可学习线性变换，让模型自己学会如何从同一个 token 表示中抽取“提问特征”“索引特征”和“内容特征”。

4.8. 为什么 Q 和 K 不用同一个权重矩阵？

理论上可以让 Query 和 Key 共用同一个权重矩阵，甚至可以直接用 XX^T 做自身点乘。但标准 Transformer 通常让 W^Q 和 W^K 独立，因为 Query 和 Key 扮演的是两种不同角色：

1. Query：当前位置想找什么信息
2. Key：当前位置如何暴露自己，方便别人匹配到自己

这两个角色需要能互相匹配，但不一定应该是同一种表示。

例如在句子 “The animal didn’t cross the street because it was tired” 中，“it” 的 Query 可能表达的是：“我需要找一个可以被代词指代的实体”；而 “animal” 的 Key 可能表达的是：“我是一个可以被代词指代的实体”。这两种向量应该容易匹配，但它们表达的方向并不相同。

如果强行共用同一个矩阵，就等于要求 “提问方式” 和 “被检索方式” 使用完全相同的投影方式，表达能力会更弱。

还有一个更重要的原因：直接用自身点乘会让原始匹配分数天然对称。

$$A = XX^T \quad (95)$$

这时：

$$\text{score}(i, j) = \text{score}(j, i) \quad (96)$$

也就是说，第 i 个 token 看第 j 个 token 的匹配分数，和第 j 个 token 看第 i 个 token 的匹配分数一样。

但语言中的关系经常不是对称的。“it” 应该关注 “animal”，不等于 “animal” 也应该同样关注 “it”。动词选择宾语、形容词修饰名词、当前位置读取前文，这些关系都有方向性。

使用不同的 W^Q 和 W^K 后：

$$\text{score}(i, j) = (x_i W^Q) \cdot (x_j W^K) \quad (97)$$

$$\text{score}(j, i) = (x_j W^Q) \cdot (x_i W^K) \quad (98)$$

这两个值一般不相等。因此 attention 可以学到更灵活、更有方向性的 token 关系。

所以总结起来：

1. W^Q 学的是如何提出问题
2. W^K 学的是如何暴露索引
3. 二者分开可以避免把 attention 限制成对称匹配

5. Multi-Head Attention

5.1. 为什么需要多个 Head?

单个 attention 头只能在一个表示空间里计算关系。但语言关系是多面的。

同一个句子中，一个词可能同时需要关注：

1. 它的主语
2. 它修饰的名词
3. 它所在短语的边界
4. 它和远处词的指代关系

一个 head 可能关注局部短语结构，另一个 head 可能关注主谓关系，还有一个 head 可能关注代词指代。最后模型把这些视角合并，形成更丰富的 token 表示。

如果只有一个 attention 头，这些关系都挤在同一个注意力分布里，表达能力有限。Multi-Head Attention 的想法是：让多个 attention 头在不同子空间中并行观察序列。

这类似多人讨论：一个人只从一个角度看问题，多个专家从不同角度，再把结论合并，表达能力更强。

5.2. 公式

先把进入 MHA 模块的 Query、Key、Value 输入记作 X_Q, X_K, X_V 。每个 head 都有自己独立的投影矩阵：

$$\text{head}_i = \text{Attention}(X_Q W_i^Q, X_K W_i^K, X_V W_i^V) \quad (99)$$

然后把所有 head 拼接起来，再做一次线性变换：

$$\text{MultiHead}(X_Q, X_K, X_V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) W^O \quad (100)$$

其中 h 是 head 数量。在 self-attention 中：

$$X_Q = X_K = X_V = X \quad (101)$$

也就是 Query、Key、Value 都来自同一份序列表示，只是经过不同的线性投影。在 encoder-decoder cross-attention 中， X_Q 来自 Decoder，而 X_K, X_V 来自 Encoder。

5.3. 一个 head 到底做了什么？

对第 i 个 head，令：

$$Q_i = X_Q W_i^Q, \quad K_i = X_K W_i^K, \quad V_i = X_V W_i^V \quad (102)$$

它先计算自己的 attention 权重：

$$A_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_{\text{head}}}}\right) \quad (103)$$

再用这组权重读取 Value：

$$H_i = A_i V_i \quad (104)$$

所以每个 head 都会产生两样东西：

1. 一张独立的 $n \times n$ attention 权重矩阵 A_i
2. 一组根据这张权重矩阵汇总出来的 token 表示 H_i

MHA 的关键不只是把向量切成多份，而是让同一个 token 同时拥有多张不同的 attention 分布。对于第 t 个 token，单头 attention 只有一行权重：

$$[\alpha_{t1}, \alpha_{t2}, \dots, \alpha_{tn}] \quad (105)$$

它必须用这一组权重决定从所有位置读取多少信息。MHA 则有 h 组权重：

$$[\alpha_{t1}^1, \dots, \alpha_{tn}^1], \quad \dots, \quad [\alpha_{t1}^h, \dots, \alpha_{tn}^h] \quad (106)$$

因此，不同 head 可以在同一层里用不同方式汇总上下文。例如处理代词时，一个 head 可以更多读取可能的指代对象，另一个 head 可以关注附近的谓语，还有一个 head 可以关注句子边界。这里描述的是模型具备的表达能力，不代表训练后每个 head 一定能被赋予单一、固定且容易命名的语言学功能。

5.4. 多头不是把原始输入机械切片

容易产生的误解是：如果 $d_{\text{model}} = 512$ 、 $h = 8$ ，是不是先把输入 X 的 512 维机械切成 8 段，每个 head 只能看到其中固定的 64 维？

不是。以 Query 为例，每个 head 的投影矩阵通常是：

$$W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}} \quad (107)$$

因此：

$$Q_i = XW_i^Q \quad (108)$$

计算 Q_i 的每一个输出维度时，都可以使用 X 的全部 d_{model} 个输入维度。Key 和 Value 也是同理。所谓每个 head 位于不同的子空间，是指各个 head 通过自己学习到的投影矩阵，把完整输入映射到不同的 d_{head} 维表示；不是人为规定第 1 个 head 只能看输入的前 64 维、第 2 个 head 只能看接下来的 64 维。

工程实现中，通常先用一个大矩阵一次性算出所有 head 的投影结果，再 reshape 出 head 这一维。这里的“拆 head”发生在投影之后，逻辑上仍然等价于每个 head 分别读取完整输入并做独立投影。

5.5. 为什么不直接用一个更宽的 Head？

一个更宽的单头可以拥有更大的 Query、Key、Value 向量，但对每个 Query 位置，它仍然只生成一组经过 softmax 的 attention 权重。多个 head 则会分别执行 softmax，得到多组可以同时成立的读取模式。

例如当前位置既需要从主语读取“谁执行动作”，又需要从时间状语读取“动作何时发生”。单头必须把这两类需求混合在一张权重分布中；多头可以让不同 head 分别形成自己的权重分布，再通过 concat 和 W^O 合并结果。这是“一个宽 head”和“多个窄 head”之间最核心的结构差别。

5.6. Head 越多，计算量就按倍数增加吗？

在常见设置中：

$$d_{\text{head}} = \frac{d_{\text{model}}}{h} \quad (109)$$

因此所有 head 的总宽度仍然是：

$$hd_{\text{head}} = d_{\text{model}} \quad (110)$$

只看 attention 分数矩阵乘法，单个 head 的主要计算量与 $n^2 d_{\text{head}}$ 同阶， h 个 head 合起来是：

$$hn^2 d_{\text{head}} = n^2 d_{\text{model}} \quad (111)$$

所以在总隐藏维度固定时，把一个表示拆成更多 head，并不会让这部分矩阵乘法直接变成原来的 h 倍。Q/K/V 投影的总输出宽度通常也保持不变。

不过 head 数量仍然会影响实际性能：每个 head 都有独立的 attention 分数和 softmax，head 太多会带来更多调度、访存和中间状态开销；同时 d_{head} 会变小，单个 head 的表达容量也会下降。因此 h 是表示能力、训练效果和硬件效率之间的设计选择，并不是越多越好。

5.7. Multi-Head 的工程实现

论文公式把每个 head 写成单独的投影：

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \quad (112)$$

但实际代码通常不会真的写一个循环，为每个 head 单独做三次小矩阵乘法。工程实现一般把所有 head 的 Q/K/V 投影合并成一次大矩阵乘法。

假设输入是：

$$X \in \mathbb{R}^{B \times n \times d_{\text{model}}} \quad (113)$$

其中：

1. B 是 batch size
2. n 是序列长度
3. d_{model} 是隐藏维度
4. h 是 head 数量
5. $d_{\text{head}} = \frac{d_{\text{model}}}{h}$

这里的 batch 可以理解为“这一轮一起送进模型计算的样本数量”。例如一次同时处理 4 句话，那么 $B = 4$ 。每句话都有自己的 token 序列，每个 token 又有自己的隐藏向量，所以输入张量需要三维：

$$\text{batch 维} \times \text{sequence 维} \times \text{hidden 维} \quad (114)$$

也就是：

$$B \times n \times d_{\text{model}} \quad (115)$$

如果只处理一句话，可以认为 $B = 1$ 。这时输入形状是：

$$1 \times n \times d_{\text{model}} \quad (116)$$

batch 维不会改变 attention 的数学含义。attention 仍然是在每个样本内部，让这句话的 token 互相看。不同 batch 样本之间通常不会互相 attention，它们只是被打包在一起并行计算，以提高 GPU 利用率。

常见实现会定义一个合并后的权重矩阵：

$$W^{\text{QKV}} \in \mathbb{R}^{d_{\text{model}} \times 3d_{\text{model}}} \quad (117)$$

一次性计算：

$$\text{QKV} = XW^{\text{QKV}} \quad (118)$$

得到：

$$\text{QKV} \in \mathbb{R}^{B \times n \times 3d_{\text{model}}} \quad (119)$$

然后把最后一维拆成三份：

$$Q, K, V \in \mathbb{R}^{B \times n \times d_{\text{model}}} \quad (120)$$

接着把每个 d_{model} 维向量拆成 h 个 head：

$$Q \rightarrow \mathbb{R}^{B \times n \times h \times d_{\text{head}}} \quad (121)$$

$$K \rightarrow \mathbb{R}^{B \times n \times h \times d_{\text{head}}} \quad (122)$$

$$V \rightarrow \mathbb{R}^{B \times n \times h \times d_{\text{head}}} \quad (123)$$

为了方便矩阵乘法，通常会转置成：

$$Q \rightarrow \mathbb{R}^{B \times h \times n \times d_{\text{head}}} \quad (124)$$

$$K \rightarrow \mathbb{R}^{B \times h \times n \times d_{\text{head}}} \quad (125)$$

$$V \rightarrow \mathbb{R}^{B \times h \times n \times d_{\text{head}}} \quad (126)$$

这时每个 batch、每个 head 都有自己的一组 Q, K, V 。attention 分数为：

$$S = \frac{QK^T}{\sqrt{d_{\text{head}}}} \quad (127)$$

形状是：

$$S \in \mathbb{R}^{B \times h \times n \times n} \quad (128)$$

这里第一个 n 表示 query 位置，第二个 n 表示 key 位置。对最后一维做 softmax：

$$A = \text{softmax}(S) \quad (129)$$

再加权 Value：

$$O = AV \quad (130)$$

得到：

$$O \in \mathbb{R}^{B \times h \times n \times d_{\text{head}}} \quad (131)$$

然后把 head 维度转回 token 维度旁边：

$$O \rightarrow \mathbb{R}^{B \times n \times h \times d_{\text{head}}} \quad (132)$$

再把所有 head 拼接回 d_{model} 维：

$$O \rightarrow \mathbb{R}^{B \times n \times d_{\text{model}}} \quad (133)$$

最后经过输出投影矩阵：

$$W^O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}} \quad (134)$$

$$Y = OW^O \quad (135)$$

输出仍然是：

$$Y \in \mathbb{R}^{B \times n \times d_{\text{model}}} \quad (136)$$

所以 Multi-Head Attention 的实现主线是：

1. 输入 X
2. 一次大矩阵乘法得到 Q, K, V
3. 把 Q, K, V reshape 成多个 head
4. 每个 head 独立计算 attention
5. 把所有 head 的输出 concat 回来
6. 经过 W^O 输出投影

需要注意，合并成 W^{QKV} 只是计算上的优化。概念上它仍然等价于每一层、每个 head 都有自己的 W_i^Q, W_i^K, W_i^V 。只是这些小矩阵在内存里被拼成了一个大矩阵，方便 GPU 一次性完成更大的矩阵乘法。

5.8. 一个小尺寸例子

假设：

$$d_{\text{model}} = 8, \quad h = 2 \quad (137)$$

那么：

$$d_{\text{head}} = 4 \quad (138)$$

每个 token 原本用 8 个数字表示。每个 head 都会从完整的 8 维输入中独立投影出一个 4 维表示；实现上再把合并计算出的 8 维结果 reshape 成 2 个 head。于是每个 head 在自己学习到的 4 维子空间中计算 attention。

如果输入形状是：

$$X \in \mathbb{R}^{B \times n \times 8} \quad (139)$$

概念上，每个 head 都有自己的 Q/K/V 投影矩阵。因为每个 head 输出 4 维，所以第 1 个 head 的矩阵形状可以写成：

$$W_1^Q \in \mathbb{R}^{8 \times 4}, \quad W_1^K \in \mathbb{R}^{8 \times 4}, \quad W_1^V \in \mathbb{R}^{8 \times 4} \quad (140)$$

第 2 个 head 也是：

$$W_2^Q \in \mathbb{R}^{8 \times 4}, \quad W_2^K \in \mathbb{R}^{8 \times 4}, \quad W_2^V \in \mathbb{R}^{8 \times 4} \quad (141)$$

如果把两个 head 的 Query 投影拼起来，就得到整层的 Query 投影：

$$W^Q = [W_1^Q, W_2^Q] \in \mathbb{R}^{8 \times 8} \quad (142)$$

同理：

$$W^K \in \mathbb{R}^{8 \times 8}, \quad W^V \in \mathbb{R}^{8 \times 8} \quad (143)$$

如果再把 Q/K/V 三个投影拼成一个大矩阵：

$$W^{\text{QKV}} = [W^Q, W^K, W^V] \in \mathbb{R}^{8 \times 24} \quad (144)$$

合并 QKV 投影后：

$$\text{QKV} \in \mathbb{R}^{B \times n \times 24} \quad (145)$$

拆成 Q/K/V 后：

$$Q, K, V \in \mathbb{R}^{B \times n \times 8} \quad (146)$$

再拆 head：

$$Q, K, V \in \mathbb{R}^{B \times 2 \times n \times 4} \quad (147)$$

每个 head 独立算 attention，输出仍然是：

$$O \in \mathbb{R}^{B \times 2 \times n \times 4} \quad (148)$$

最后 concat 两个 head：

$$O \in \mathbb{R}^{B \times n \times 8} \quad (149)$$

再乘以输出投影 W^O 。因此，多头注意力虽然内部拆成多个 head，但整个模块输入和输出的最后一维通常都保持 d_{model} 不变，这样才能接残差连接：

$$Y = X + \text{MultiHeadAttention}(X) \quad (150)$$

5.9. 为什么要有输出投影 W^O ？

concat 之后，多个 head 的结果只是被简单拼在一起。第 1 个 head 的信息在前一段维度，第 2 个 head 的信息在后一段维度，它们还没有充分混合。

输出投影：

$$Y = OW^O \quad (151)$$

作用是把不同 head 的信息重新线性组合，让模型可以学习：

1. 哪些 head 更重要
2. 不同 head 的信息如何混合
3. 而且经过 attention 计算 + 多头的合并，维度已经不再是 hidden dimension，需要对于维度也做转换

因此 W^O 不是可有可无的收尾操作，而是 multi-head 结果融合的一部分。

6. 残差连接

6.1. 残差连接在 Transformer 里长什么样？

Transformer 不是把 attention 或 FFN 的输出直接作为下一层输入，而是在每个子层外面加一个残差连接。

如果某个子层记作：

$$\text{SubLayer}(x) \quad (152)$$

那么带残差连接的形式是：

$$y = x + \text{SubLayer}(x) \quad (153)$$

在 Encoder 中，常见写法是：

$$X' = \text{LayerNorm}(X + \text{MultiHeadSelfAttention}(X)) \quad (154)$$

$$Y = \text{LayerNorm}(X' + \text{FFN}(X')) \quad (155)$$

这里有两个残差连接：

1. attention 子层外面有一次： $X + \text{MultiHeadSelfAttention}(X)$
2. FFN 子层外面有一次： $X' + \text{FFN}(X')$

也就是说，attention 和 FFN 都不是完全替换原来的表示，而是在原表示上增加一个修正量。

6.2. 为什么需要残差连接？

如果没有残差连接，一层的输出完全由当前子层决定：

$$y = \text{SubLayer}(x) \quad (156)$$

这意味着每一层都必须重新生成一个可用表示。网络变深以后，训练会变困难：后面的损失信号要穿过很多层非线性变换才能传回前面层，梯度更容易变弱、变乱，模型也更难学到“某些信息应该原样保留”。

残差连接提供了一条更直接的信息通路：

$$y = x + \text{SubLayer}(x) \quad (157)$$

它至少带来三个好处。

第一，模型可以保留原信息。假设某一层暂时没有必要大幅修改 token 表示，那么它可以让 $\text{SubLayer}(x)$ 接近 0，于是：

$$y \approx x \quad (158)$$

这相当于允许这一层近似跳过自己。深层网络因此更容易训练，因为不是每一层都被迫做复杂变换。

第二，模型学习的是增量。可以把 $\text{SubLayer}(x)$ 理解成“在原表示基础上需要补充或修正什么”。比如一个 token 原本已经有词义信息，attention 子层只需要把相关上下文信息加进来；FFN 子层只需要进一步加工其中某些特征。这样比每层从头构造完整表示更容易。

第三，梯度更容易传回前面层。因为 y 中直接包含 x ，反向传播时梯度可以沿着加法路径传回去，不必完全依赖子层内部的复杂变换。这是深层 Transformer 能稳定训练的重要原因之一。

6.3. 一个直观例子

假设某个 token 的表示 x 已经包含了“机器”这个词的基本语义。经过 attention 后，模型从前文“我喜欢”中取回一些上下文信息：

$$\text{Attention}(x) = \text{和喜欢有关的上下文增量} \quad (159)$$

残差之后得到：

$$x + \text{Attention}(x) \quad (160)$$

这可以理解为：

$$\text{机器本身的表示} + \text{从上下文取回来的信息} \quad (161)$$

如果直接用 $\text{Attention}(x)$ 替代 x ，原始 token 信息可能被破坏；如果使用残差，模型可以在保留原信息的同时叠加新信息。

FFN 外面的残差连接也是同理。FFN 对每个 token 的表示做非线性加工，但加工结果仍然和输入相加，避免每个子层都把已有信息完全重写。

6.4. Post-LN 和 Pre-LN

原始 Transformer 论文常写成：

$$y = \text{LayerNorm}(x + \text{SubLayer}(x)) \quad (162)$$

这叫 Post-LN，因为 LayerNorm 放在残差加法之后。

很多现代大模型更常使用 Pre-LN：

$$y = x + \text{SubLayer}(\text{LayerNorm}(x)) \quad (163)$$

也就是先归一化，再进入子层，最后做残差相加。Pre-LN 往往让深层模型训练更稳定，所以在大型语言模型中很常见。

不过无论是 Post-LN 还是 Pre-LN，残差连接的核心作用没有变：让原表示有一条直接传递路径，让每个子层主要学习对表示的增量修正。

更深一层看，Pre-LN 和 Post-LN 的差别主要体现在梯度路径上。Post-LN 可以展开成：

$$x_{l+1} = \text{LayerNorm}(x_l + f_l(x_l)) \quad (164)$$

残差主干每经过一层都要穿过一次归一化。层数很深时，归一化的导数会持续改变梯度尺度，浅层梯度更容易衰减，深层梯度也可能在训练早期变得很不均衡。

Pre-LN 则写成：

$$x_{l+1} = x_l + f_l(\text{LayerNorm}(x_l)) \quad (165)$$

此时残差主干上有一条更直接的恒等路径。子层输入被归一化，子层输出作为增量加回原表示。现代 decoder-only LLM 往往使用 Pre-LN 或它的变体，本质原因就是深层堆叠时更容易训练稳定。

7. Norm

参考资料: <https://zhuanlan.zhihu.com/p/1985480142802412942>

7.0.1. 为什么 Transformer 需要 Norm?

Transformer 每层都在不断做矩阵乘法、attention 加权、残差相加和 FFN 非线性变换。如果没有归一化，隐藏状态的数值尺度可能在层与层之间不断漂移。

数值尺度漂移会带来几个问题：

1. 某些层输出过大，后面的 softmax 或激活函数进入极端区域
2. 某些层输出过小，信号和梯度都变弱
3. 不同层的输入分布变化太大，训练更难稳定

LayerNorm 的作用就是把每个 token 表示拉回到比较稳定的数值范围。它不改变序列长度，也不直接决定关注谁，但它让 attention 和 FFN 的输入更规整，训练更稳定。

可以把 Norm 理解成每个 token 在进入下一步计算前做一次“尺度整理”。这个整理不是把所有信息抹掉，因为归一化之后还有可学习的 γ 和 β 。模型可以自己学会哪些维度需要放大，哪些维度需要平移。

7.1. BatchNorm

BatchNorm 常见于 CNN，它通常依赖一个 batch 内多个样本的统计量。LayerNorm 不依赖 batch 内其他样本，而是在单个样本、单个 token 的隐藏维度内计算统计量。

这对语言模型很重要。语言模型推理时经常一次只生成一个 token，也可能 batch 大小变化很大。如果归一化依赖 batch 统计量，推理行为会不稳定。LayerNorm 只看当前 token 自己的隐藏向量，所以训练和推理更一致。

简单说：

1. BatchNorm：更关心 batch 维度上的统计
2. LayerNorm：更关心单个 token 内部隐藏维度的统计
3. Transformer 里通常使用 LayerNorm，因为它适合变长序列和自回归生成

7.2. LayerNorm

7.2.1. LayerNorm 在归一化什么？

LayerNorm 的全称是 Layer Normalization。它的作用是把一个 token 的隐藏向量归一化，让这个向量内部各维度的数值尺度更稳定。

假设某个 token 的隐藏表示是：

$$\vec{x} = [x_1, x_2, \dots, x_d] \quad (166)$$

这里 $d = d_{\text{model}}$ 。LayerNorm 会先计算这个 token 向量内部所有维度的均值和方差：

$$\mu = \frac{1}{d} \sum_{i=1}^d x_i \quad (167)$$

$$\sigma^2 = \frac{1}{d} \sum_{i=1}^d (x_i - \mu)^2 \quad (168)$$

然后做归一化：

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}} \quad (169)$$

最后再通过可学习参数缩放和平移：

$$y_i = \gamma_i \hat{x}_i + \beta_i \quad (170)$$

其中：

1. μ 和 σ^2 来自当前 token 的隐藏维度
2. ε 是很小的数，防止除以 0
3. γ_i 和 β_i 是可训练参数
4. 输出 $\vec{y}$ 的维度仍然是 d_{model}

注意 LayerNorm 不是在不同 token 之间求均值。它通常是对每个 token 自己的隐藏维度做归一化。输入有 n 个 token 时，每一行都单独做一次 LayerNorm：

$$X \in \mathbb{R}^{n \times d_{\text{model}}} \quad (171)$$

第 1 行算第 1 行的均值和方差，第 2 行算第 2 行的均值和方差。不同 token 不会在 LayerNorm 里互相混合。token 之间的信息交换主要发生在 attention，不发生在 LayerNorm。

7.2.2. 从 Post-Norm 到 Pre-Norm

LaayerNorm 放在哪里？

在原论文的 Encoder 层中，常见公式是：

$$X' = \text{LayerNorm}(X + \text{MultiHeadSelfAttention}(X)) \quad (172)$$

$$Y = \text{LayerNorm}(X' + \text{FFN}(X')) \quad (173)$$

这里 LayerNorm 放在残差相加之后。

在现代 decoder-only 大模型中，更常见的是：

$$X' = X + \text{Attention}(\text{LayerNorm}(X)) \quad (174)$$

$$Y = X' + \text{FFN}(\text{LayerNorm}(X')) \quad (175)$$

这两种结构都在解决同一个问题：让深层网络的中间表示尺度稳定，让训练更容易。区别主要是工程和优化稳定性上的取舍。

7.2.3. LayerNorm 的数值和工程代价

归一化之所以重要，不只是为了让训练更快，也是在保护低精度计算。LLM 训练和推理经常使用 FP16、BF16、FP8 等格式。FP16 的动态范围有限，隐藏状态如果在层间持续放大，容易出现 "Inf" 或 "NaN"；BF16 动态范围更大，但尾数位更少，数值尺度差异过大时更容易出现舍入误差，典型现象就是“大数吃小数”。

LayerNorm 把每个 token 的隐藏向量拉回稳定尺度，可以缓解这种问题。但从 kernel 的角度看，LayerNorm 不是一个普通的逐元素操作。它需要先对隐藏维度做规约，得到：

$$\sum_i x_i \quad \text{和} \quad \sum_i x_i^2 \quad (176)$$

再计算均值、方差，最后把每个元素标准化并乘上 γ 、加上 β 。

朴素实现可能需要多次读写 HBM。高性能 CUDA/Triton kernel 通常会把一个 token 的隐藏维度放进一个 block，在片上寄存器或 shared memory 中同时累计 $\sum x$ 和 $\sum x^2$ ，利用：

$$\text{Var}(X) = E[X^2] - E[X]^2 \quad (177)$$

把 HBM 访问压到尽量少。但即使这样，LayerNorm 仍然需要维护均值相关统计量，并执行额外的减均值操作。相比纯逐元素算子，它更容易受规约、同步、寄存器压力和访存带宽影响。

需要注意， $E[X^2] - E[X]^2$ 在低精度下可能有数值抵消问题。如果两个数很接近，相减会丢失有效位，甚至因为舍入得到负方差。因此工程实现中常会用 FP32 累加，或者使用 Welford 这类更稳定的一遍统计方法。

7.3. RMSNorm

7.3.1. RMSNorm 在做什么？

RMSNorm 是 Root Mean Square Layer Normalization。它保留了 LayerNorm 中“按尺度缩放”的部分，但去掉了“减均值”这一步。

对一个 token 的隐藏向量：

$$\vec{x} = [x_1, x_2, \dots, x_d] \quad (178)$$

先计算均方根：

$$\text{RMS}(\vec{x}) = \sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2 + \varepsilon} \quad (179)$$

然后归一化：

$$y_i = \frac{\gamma_i x_i}{\text{RMS}(\vec{x})} \quad (180)$$

很多现代 LLM 中的 RMSNorm 只有可学习缩放参数 γ ，没有平移参数 β 。它的直觉是：模型主要需要稳定激活值的尺度，而不一定需要强制把每个 token 的隐藏向量均值归零。

所以 LayerNorm 和 RMSNorm 可以放在一起对比：

1. LayerNorm: 减均值, 再除以标准差, 最后乘 γ 加 β
2. RMSNorm: 不减均值, 只除以 RMS, 通常只乘 γ

7.3.2. RMSNorm 为什么适合 LLM?

从数学上看, RMSNorm 假设“尺度稳定”比“均值归零”更关键。神经网络真正使用的往往是特征之间的相对关系, 而不是隐藏向量的绝对数值大小。只要把尺度约束住, 很多层继续堆叠时就不容易因为方差膨胀而崩掉。

从工程上看, RMSNorm 更简单:

1. 不需要计算均值 μ
2. 不需要对每个元素执行 $x_i - \mu$
3. 规约状态更少, 只需要累计 $\sum x_i^2$
4. 通常省掉 β , 减少一次参数读取和一次逐元素加法

在 GPU kernel 中, 它可以写成更直接的流水:

```
load x -> sum(x^2) -> rsqrt -> x * scale * gamma -> store y
```

这类算子通常是 memory-bound。少读一个参数、少做一轮逐元素操作、少维护一组规约统计量, 都会影响吞吐和寄存器占用。

7.3.3. 为什么很多 LLM 也去掉 Linear bias?

现代 decoder-only LLM 常见组合是:

1. Pre-Norm
2. RMSNorm
3. attention 和 MLP 的 Linear 层不带 bias
4. SwiGLU 或类似门控 FFN

这不是单一数学定理推出的必然结果, 而是效果、稳定性和硬件效率共同选择出来的经验结构。

一个直观解释是: 如果每个子层前面都有归一化, 后面还会接残差和下一层归一化, 那么很多固定平移项会被后续结构吸收或削弱。保留 bias 当然也能训练, 但它会多一份参数加载和逐元素加法。大模型里这类操作会在每层、每个 token、每个投影上反复出现, 累积起来就是实际成本。

所以 RMSNorm 和无 bias Linear 的意义不只是“参数更少”, 更重要的是让主干计算更规整, 方便融合 kernel, 并减少访存压力。

8. FFN (MLP)

FFN 是 Feed-Forward Network, 也叫前馈网络。在 Transformer 里, 它通常作用在每个 token 的表示上。

基本上 FFN 的具体实现基本上 MLP

8.1. 传统 FFN 实现

一个传统 FFN 可以写成:

$$\text{FFN}(x) = \varphi(xW_1 + b_1)W_2 + b_2 \quad (181)$$

其中 φ 是非线性激活函数。原始 Transformer 使用 ReLU:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (182)$$

很多现代模型会使用 GELU、SwiGLU 等激活形式，但核心思想类似：先把隐藏维度升高，经过非线性激活，再投影回模型隐藏维度。

假设：

$$x \in \mathbb{R}^{d_{\text{model}}} \quad (183)$$

常见 FFN 的两个线性层形状是：

$$W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}} \quad (184)$$

$$W_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}} \quad (185)$$

其中 d_{ff} 通常比 d_{model} 大。例如原始 Transformer base 中：

$$d_{\text{model}} = 512, \quad d_{\text{ff}} = 2048 \quad (186)$$

也就是先从 512 维升到 2048 维，做非线性变换，再降回 512 维。

8.2. SwiGLU MLP：门控前馈网络

以常见的 **SwiGLU MLP** 为例，输入是某个 token 的隐藏向量：

$$x \in \mathbb{R}^{d_{\text{model}}} \quad (187)$$

例如可以取 $d_{\text{model}} = 4096$ 。

先用两组矩阵把它投影到更大的中间维度 d_{ff} ：

$$u = xW_{\text{up}} + b_{\text{up}} \quad (188)$$

$$g = xW_{\text{gate}} + b_{\text{gate}} \quad (189)$$

其中：

$$W_{\text{up}}, W_{\text{gate}} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}} \quad (190)$$

因此：

$$u, g \in \mathbb{R}^{d_{\text{ff}}} \quad (191)$$

接着，对 gate 分支做 SiLU 激活：

$$\text{SiLU}(g) = g\sigma(g) \quad (192)$$

$$\sigma(g) = \frac{1}{1 + \exp(-g)} \quad (193)$$

这里的 SiLU 和 sigmoid 都逐元素作用于向量 g 。

然后将两个分支逐元素相乘：

$$z = \text{SiLU}(g) \odot u \quad (194)$$

这里的 $\odot$ 不是矩阵乘法，而是对应元素相乘。直观上， g 决定每个中间特征“放行多少”， u 提供待传递的特征内容。

最后投影回模型隐藏维度：

$$\text{MLP}(x) = zW_{\text{down}} + b_{\text{down}} \quad (195)$$

其中：

$$W_{\text{down}} \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}} \quad (196)$$

合起来：

$$\text{MLP}(x) = (\text{SiLU}(xW_{\text{gate}} + b_{\text{gate}}) \odot (xW_{\text{up}} + b_{\text{up}}))W_{\text{down}} + b_{\text{down}} \quad (197)$$

在 LLaMA 一类模型中，线性层通常没有 bias，因此可简化为：

$$\text{MLP}(x) = (\text{SiLU}(xW_{\text{gate}}) \odot (xW_{\text{up}}))W_{\text{down}} \quad (198)$$

若输入是一整段 token：

$$X \in \mathbb{R}^{B \times T \times d_{\text{model}}} \quad (199)$$

同一套权重会独立应用到每个 batch 和每个 token 位置；MLP 不混合 T 这个序列维度。它只变换每个 token 内部的特征，token 与 token 的信息交换是在 Attention 中完成的。

8.3. FFN 和 Attention 分工有什么不同？

Attention 负责 token 之间的信息交换。它回答的问题是：

$$\text{当前位置应该从其他哪些位置取信息？} \quad (200)$$

FFN 负责每个 token 内部的特征加工。它回答的问题更像是：

$$\text{这个 token 已经拿到上下文信息后，应该怎样重新组织自己的内部特征？} \quad (201)$$

所以可以这样理解：

1. attention 是横向的：沿着序列维度，让 token 之间互相看
2. FFN 是纵向的：沿着隐藏维度，对单个 token 的特征做变换

给定输入：

$$X \in \mathbb{R}^{n \times d_{\text{model}}} \quad (202)$$

FFN 会对每一行独立应用同一个函数：

$$Y_i = \text{FFN}(X_i) \quad (203)$$

这里 i 表示 token 位置。第 1 个 token 的 FFN 计算不会直接读取第 2 个 token；第 2 个 token 的 FFN 计算也不会直接读取第 1 个 token。它们共享同一套 FFN 参数，但每个位置独立计算。

这就是为什么 Transformer block 通常需要 attention 和 FFN 搭配使用。只有 attention，没有 FFN，token 之间可以交换信息，但每个位置内部的非线性加工能力不足；只有 FFN，没有 attention，每个 token 可以加工自己，但无法根据上下文动态吸收其他 token 的信息。

8.4. 为什么 FFN 要先升维再降维？

如果 FFN 只是一个线性变换：

$$y = xW \quad (204)$$

那么多层线性变换叠在一起，本质上仍然可以合并成一个线性变换，表达能力有限。

FFN 中间加入更高维空间和非线性激活：

$$x \rightarrow xW_1 + b_1 \rightarrow \varphi(\dots) \rightarrow W_2 \rightarrow y \quad (205)$$

这样模型可以学习更复杂的特征组合。升维可以提供更大的中间表示空间，非线性激活可以让模型表达“某些特征组合出现时才激活”这类模式。

例如某个 token 经过 attention 后已经融合了上下文：

$$\text{机器} + \text{喜欢} + \text{后面可能接技术名词} \quad (206)$$

FFN 可以进一步把这种混合信息加工成更适合下一层使用的特征，例如增强“机器学习短语正在形成”这种内部信号。下一层 attention 再基于这个更丰富的表示去和其他 token 交互。

8.5. FFN 的位置为什么在 Attention 后面？

一个 Transformer block 通常先做 attention，再做 FFN：

1. attention：先让 token 收集上下文
2. FFN：再对收集完上下文的每个 token 做内部加工

如果用一句话概括：

$$\text{Attention 负责交流，FFN 负责思考。} \quad (207)$$

这里的“思考”不是说 FFN 有单独意识，而是说它对每个位置的隐藏特征做非线性重组。经过很多层重复之后，模型就形成了复杂的上下文表示。

8.6. FFN 也参与预测下一个词

预测下一个 token 时，最后一个位置的隐藏状态不是只由 attention 决定的。它经过了多层：

1. attention 从历史 token 取信息
2. FFN 加工当前 token 的内部特征
3. 残差连接保留已有信息
4. LayerNorm 稳定数值尺度

所以最终的 $\vec{h}_t$ 是多种模块共同加工出来的结果。输出层把 $\vec{h}_t$ 投影到词表时，里面已经包含了 attention 取回的上下文信息，也包含了 FFN 反复加工后的非线性特征。

9. 从 Attention 输出到预测下一个词

现在回到最核心的问题：当我们已经算出了

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (208)$$

为什么它最后就能预测下一个词？

关键是要先区分两件事：

1. attention 的输出不是词表概率
2. attention 的输出是每个位置的上下文表示

也就是说，attention 本身并没有直接说“下一个词是苹果”或者“下一个词是学习”。它只负责把当前位置的表示更新成一个更懂上下文的向量。真正把这个向量变成词表概率的，是 Transformer 后面的若干层、最后的线性输出层和 softmax。

假设当前输入 token 序列是：

我 喜欢 机器

模型要预测下一个 token。经过 embedding 和位置编码之后，每个 token 会变成一个向量：

$$X = \begin{pmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vec{x}_3 \end{pmatrix} \quad (209)$$

其中 $\vec{x}_1$ 对应“我”， $\vec{x}_2$ 对应“喜欢”， $\vec{x}_3$ 对应“机器”。

在 decoder-only 语言模型里，因为使用 causal mask，第 3 个位置只能看见第 1、2、3 个位置，不能看见未来。于是第 3 个位置的 attention 输出可以写成：

$$\vec{o}_3 = \sum_{j \leq 3} \alpha_{3j} \vec{v}_j \quad (210)$$

这里的 $\vec{o}_3$ 就是“机器”这个位置看完前文之后得到的新表示。它不再只是“机器”这个词本身的向量，而是融合了“我 喜欢 机器”这个上下文的信息。模型真正用来预测下一个 token 的，通常就是最后一个可见位置的最终隐藏状态。

注意这里说的是“最终隐藏状态”，不是第一层 attention 的输出。一个 Transformer block 里通常还有残差连接、LayerNorm、FFN：

$$X' = \text{LayerNorm}(X + \text{Attention}(Q, K, V)) \quad (211)$$

$$X'' = \text{LayerNorm}(X' + \text{FFN}(X')) \quad (212)$$

多层 Transformer 会把这个过程重复很多次。第 1 层可能只形成比较浅的局部关系，例如“机器”前面有“喜欢”；更高层可能形成更抽象的语义关系，例如“我 喜欢 机器”后面很可能接“学习”、“翻译”、“视觉”这类词。经过最后一层后，我们得到：

$$H \in \mathbb{R}^{n \times d_{\text{model}}} \quad (213)$$

第 t 行 $\vec{h}_t$ 表示第 t 个位置的最终上下文表示。如果输入长度是 t ，那么预测下一个 token 时，取最后一行：

$$\vec{h}_t \quad (214)$$

接下来需要把这个 d_{model} 维向量映射到整个词表。假设词表大小是 N_{vocab} ，输出层有一个可学习矩阵：

$$W_{\text{out}} \in \mathbb{R}^{d_{\text{model}} \times N_{\text{vocab}}} \quad (215)$$

再加上偏置：

$$\vec{z}_t = \vec{h}_t W_{\text{out}} + \vec{b} \quad (216)$$

这里 $\vec{z}_t$ 的维度是 N_{vocab} 。它里面的每一个数字对应词表中一个 token 的原始分数，通常叫 logits。例如：

1. “学习” 对应一个 logit
2. “苹果” 对应一个 logit
3. “天气” 对应一个 logit
4. 词表里其他 token 也各有一个 logit

logit 还不是概率，它可以是任意实数。模型最后再对这些 logits 做 softmax：

$$\vec{p}_t = \text{softmax}(\vec{z}_t) \quad (217)$$

这样就得到一个词表上的概率分布：

$$P(x_{t+1} = w \mid x_1, \dots, x_t) \quad (218)$$

也就是说，对词表里的每个候选 token w ，模型都会给出一个概率。概率最大的 token 就是模型最倾向的下一个词。如果使用贪心解码，就直接选概率最大的 token；如果使用采样，就按照这个概率分布随机抽一个 token；如果使用 top-k、top-p、temperature，它们也都是在这个概率分布上做不同的选择策略。

所以完整链路可以概括为：

1. 输入 token 变成 embedding
2. embedding 加上位置信息，得到 X
3. 每层通过 Q, K, V 计算 attention，让 token 之间交换信息
4. 残差、LayerNorm、FFN 继续加工每个位置的表示
5. 最后一层得到最终隐藏状态 H
6. 取最后一个位置的向量 $\vec{h}_t$
7. 用 W_{out} 投影到词表维度，得到 logits
8. 对 logits 做 softmax，得到下一个 token 的概率分布
9. 根据解码策略选出下一个 token

这里最容易误解的是第 3 步和第 7 步的关系。Attention 的输出维度通常还是模型隐藏维度，例如 $d_{\text{model}} = 4096$ 。但词表大小可能是几万、十几万。attention 输出向量本身并不对应某个词，它只是一个语义状态；输出层 W_{out} 才负责把这个语义状态翻译成“词表中每个 token 有多合适”。

可以把 W_{out} 理解为一个大型打分器。它的每一列对应词表里的一个 token。拿 $\vec{h}_t$ 和某个 token 对应的输出向量做点积，就得到这个 token 的分数：

$$z_{t,w} = \vec{h}_t \cdot \vec{w}_{\text{out}} + b_w \quad (219)$$

如果 $\vec{h}_t$ 表示的上下文语义接近“接下来应该说学习”，那么“学习”对应的输出向量就应该和 $\vec{h}_t$ 点积更大，logit 更高，softmax 后概率也更高。

这个能力是训练出来的。训练语言模型时，样本会被构造成“根据前文预测下一个 token”。例如训练句子是：

我 喜欢 机器 学习

模型输入可以是：

我 喜欢 机器

正确答案是：

学习

前向计算得到概率分布以后，模型会看正确 token “学习”的概率有多大。如果它给“学习”的概率低，而给“苹果”的概率高，交叉熵损失就会很大：

$$L = -\log P(x_{t+1} = \text{学习} \mid x_1, \dots, x_t) \quad (220)$$

反向传播会从这个损失一路传回输出层、最后一层 Transformer、前面的 attention、再到 W^Q, W^K, W^V 。所以虽然训练数据没有直接告诉模型“第 3 个 token 应该关注第 2 个 token”，但如果某种 attention 权重能让最终正确 token 的概率变大，那么对应参数就会被更新到更容易产生这种权重。

因此，attention 能帮助预测下一个词，不是因为公式本身直接生成词，而是因为它生成了更好的上下文表示。这个上下文表示经过多层加工后，被输出层映射成词表分数，再通过 softmax 变成概率。预测下一个词的本质是：

用最后一个位置的上下文表示，对整个词表做一次分类。 (221)

每生成一个 token 后，这个 token 会被追加到上下文末尾。下一轮模型再把新的完整前缀作为条件，预测再下一个 token。于是语言生成就变成了不断重复同一个过程：

$$P(x_{t+1} \mid x_1, \dots, x_t) \quad (222)$$

生成出 x_{t+1} 后，再计算：

$$P(x_{t+2} \mid x_1, \dots, x_t, x_{t+1}) \quad (223)$$

这就是从 attention 公式走到自回归文本生成的关键路径。

9.1. Temperature 和采样

模型最后输出的是 logits：

$$\vec{z} \in \mathbb{R}^{N_{\text{vocab}}} \quad (224)$$

如果直接取最大 logit 对应的 token，这叫贪心解码：

$$x_{t+1} = \operatorname{argmax}_i z_i \quad (225)$$

它确定性强，但容易变得重复和保守。实际生成常会先对 logits 做 temperature 缩放，再 softmax：

$$p_i = \frac{\exp(\frac{z_i}{T})}{\sum_j \exp(\frac{z_j}{T})} \quad (226)$$

这里 T 是 temperature：

1. $T < 1$ ：logits 差距被放大，分布更尖锐，模型更倾向于高概率 token
2. $T = 1$ ：使用原始 softmax 分布
3. $T > 1$ ：logits 差距被压缩，分布更平坦，低概率 token 更容易被采到

例如两个 token 的 logit 差是 Δ 。经过 temperature 后，softmax 实际看到的是：

$$\frac{\Delta}{T} \quad (227)$$

所以低温会放大差距，高温会缩小差距。

得到概率分布以后，采样的目标是：如果 $p_i = 0.8$ ，那么 token i 应该以 80% 的概率被抽中。最直接的做法是 multinomial sampling：生成一个 $[0, 1]$ 的随机数 u ，然后对概率做前缀和，找到第一个超过 u 的位置。

这个方法直观，但对大词表不一定友好。现代 LLM 词表可能有十几万甚至二十多万个 token。前缀和有数据依赖，还可能涉及跨 GPU 分片通信，不如纯逐元素计算和规约适合 GPU。

9.2. Gumbel-Max Trick

Gumbel-Max Trick 提供了另一种等价采样方式。设 g_i 是标准 Gumbel 噪声：

$$g_i = -\log(-\log u_i), \quad u_i \sim \text{Uniform}(0, 1) \quad (228)$$

那么：

$$\operatorname{argmax}_i(\log p_i + g_i) \quad (229)$$

得到的索引服从原来的离散分布 p 。也就是说，给每个 token 的对数概率加一份 Gumbel 噪声，然后取最大值，等价于按概率做 multinomial sampling。

由于 $p_i = \operatorname{softmax}(z)_i$ ，而 $\log p_i$ 和 z_i 只差一个所有 token 共享的常数：

$$\log p_i = z_i - \log \sum_j \exp(z_j) \quad (230)$$

取 argmax 时这个公共常数不影响结果，所以也可以理解为：

$$\operatorname{argmax}_i(z_i + g_i) \quad (231)$$

工程上还有一个常见变体。若 $q_i \sim \operatorname{Exp}(1)$ ，则：

$$\operatorname{argmax}_i \frac{p_i}{q_i} \quad (232)$$

和 Gumbel-Max 是等价的，因为：

$$\operatorname{argmax}_i \frac{p_i}{q_i} = \operatorname{argmax}_i(\log p_i - \log q_i) \quad (233)$$

而 $-\log q_i$ 可以对应到 Gumbel 噪声形式。

这个转换的 Infra 价值很大：采样从“前缀和 + 查找”变成了：

1. 每个词表位置独立生成随机噪声
2. 对概率或 logit 做逐元素变换
3. 做一次并行 argmax 规约

如果词表按 tensor parallel 切到多张 GPU 上，每张卡可以先在本地 shard 上找局部最大值和索引，再通过带索引的全局 max-reduce 得到最终 token。通信对象从整张词表概率降成少量最大值元数据。

9.3. LLM 为什么知道何时停下来？

简单来说，结束 token 也是候选类别之一，也就是：

训练数据会把文本边界也编码进序列。例如一段训练文本可以抽象成：

$$x_1, x_2, \dots, x_t, x_{\text{EOS}} \quad (234)$$

训练时，模型不仅要学习：

$$P(x_{i+1} \mid x_1, \dots, x_i) \quad (235)$$

也要学习在内容结束时预测：

$$P(x_{\text{EOS}} \mid x_1, \dots, x_t) \quad (236)$$

10. Self-Attention

10.1. 什么是 Self-Attention？

Self-attention 的意思是：Query、Key、Value 都来自同一个序列。

给定输入矩阵 X ，通过三组可学习参数得到：

$$Q = XW^Q \quad (237)$$

$$K = XW^K \quad (238)$$

$$V = XW^V \quad (239)$$

然后计算：

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (240)$$

因此，每个 token 的新表示都由整个序列的信息加权组合而成。

这里的 **self** 不是说每个 token 只关注自己，而是说 Query、Key、Value 都来自同一个序列本身。也就是说，这个序列内部的 token 互相做 attention。

如果第 i 个 token 想更新自己的表示，它会用自己的 Query q_i 去匹配同一句话中所有 token 的 Key k_j ：

$$\text{score}(i, j) = q_i \cdot k_j \quad (241)$$

然后根据这些匹配分数，从同一句话中所有 token 的 Value 里加权取信息。

例如在句子 “The animal didn’t cross the street because it was tired” 中，“it” 的 Query 可以去匹配同一句话里的 “animal”、“street” 等 token 的 Key。如果模型学得好，“animal” 的 Key 会和 “it” 的 Query 更匹配，于是 “it” 更新后的表示会更多融合 “animal” 的 Value 信息。

所以 self-attention 的重点是：

1. 不是只看自己这个位置
2. 而是在同一个序列内部，所有 token 互相查看、互相取信息

和 cross-attention 对比会更清楚：

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V \quad \text{self-attention} \quad (242)$$

$$Q = ZW^Q, \quad K = HW^K, \quad V = HW^V \quad \text{cross-attention} \quad (243)$$

Self-attention 是一个序列内部自己问自己；cross-attention 是一个序列用自己的 Query 去问另一个序列提供的 Key 和 Value。

10.2. Self-Attention 在学习什么？

Self-attention 学到的是 token 之间的关系。不同 attention 头可能学到不同类型关系：

1. 主语和谓语的关系
2. 代词和先行词的关系
3. 形容词和名词的关系
4. 长距离依赖关系
5. 局部短语结构

它不是手工指定规则，而是通过训练数据自动学出这些关系。

10.3. Self-Attention 的优点

和 RNN 相比，self-attention 有两个重要优点：

1. **并行性好**：所有 token 的 Query、Key、Value 可以同时计算
2. **路径短**：任意两个 token 之间只需要一次 attention 就能直接交互

RNN 中第 1 个 token 的信息传到第 100 个 token，需要经过 99 步。Self-attention 中，第 100 个 token 可以直接关注第 1 个 token。

这就是 Transformer 能高效建模长距离依赖的重要原因。

11. 位置编码

11.1. 为什么需要位置编码？

Self-attention 本身不关心顺序。它看到的是一组 token 向量，计算的是两两相似度。如果打乱输入 token 的顺序，只要不加入位置信息，attention 机制本身无法知道顺序变化。

Attention is all you need 原论文中，使用的是正弦位置编码，后来的模型也常使用可学习位置编码、相对位置编码、RoPE 等方式，但核心问题不变：attention 需要额外机制知道 token 顺序。

但自然语言强烈依赖顺序：

1. “猫追狗”
2. “狗追猫”

这两个句子词相同，但意义完全不同。因此 Transformer 必须额外加入位置信息。

11.2. 正弦位置编码

原论文使用正弦和余弦函数构造位置编码：

$$\text{PE}_{p,2i} = \sin\left(\frac{p}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \quad (244)$$

$$\text{PE}_{p,2i+1} = \cos\left(\frac{p}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \quad (245)$$

其中：

1. p 是 token 的位置
2. i 是维度索引
3. d_{model} 是模型隐藏维度

最终输入不是单纯的词向量，而是：

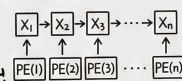
$$X = \sqrt{d_{\text{model}}} \text{TokenEmbedding} + \text{PositionEncoding} \quad (246)$$

正弦位置编码 (Sinusoidal Positional Embeddings)

Transformer模型中编码位置信息的关键技术

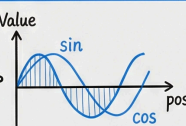
一、介绍

- 它是用于Transformer模型的方法,用来编码位置信息。
- 它使得模型能够理解序列元素的顺序。
- 在“Attention is All You Need”论文中首次提出。



二、原理

- 核心思想: 使用不同频率的正弦和余弦函数。
- 利用周期性变化来表示相对位置关系。



- 对于偶数维度:

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

- 对于奇数维度:

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

pos: 位置
i: 维度索引 ($0 \leq i < d_{model}/2$)
 d_{model} : 嵌入维度

三、计算过程

- 确定位置(pos)和嵌入维度(d_{model}): e.g., $pos = 5$, $d_{model} = 512$
- 对于每个维度*i*, 计算波长: $Wavelength = 10000^{2i/d_{model}}$
- 对于偶数索引, 应用正弦函数
 $PE(pos, 2i) = \sin\left(\frac{pos}{Wavelength}\right)$
- 对于奇数索引, 应用余弦函数
 $PE(pos, 2i+1) = \cos\left(\frac{pos}{Wavelength}\right)$
- 组合形成完整的位置编码向量
 $PE(pos) = [PE(pos, 0), PE(pos, 1), \dots, PE(pos, d_{model}-1)]$

$$PE(pos) = \begin{bmatrix} \sin(\omega_0 \cdot pos) \\ \cos(\omega_0 \cdot pos) \\ \sin(\omega_1 \cdot pos) \\ \cos(\omega_1 \cdot pos) \\ \vdots \end{bmatrix}$$

Figure 2: 正弦位置编码的公式、频率与向量组装过程

11.2.1. 为什么 Embedding 要乘以 $\sqrt{d_{model}}$?

原论文在把词向量和位置编码相加之前, 会先把词向量乘以 $\sqrt{d_{model}}$:

$$x_i = \sqrt{d_{model}}e_i + p_i \quad (247)$$

其中:

- e_i 是第 i 个 token 的 embedding
- p_i 是第 i 个位置的位置编码
- d_{model} 是 embedding size, 也就是隐藏维度

这样做的主要目的, 是让 token embedding 的数值尺度和 position encoding 的数值尺度更匹配。

原论文使用的正弦位置编码由 sin 和 cos 构造, 所以每一维大致落在 $[-1, 1]$ 。而 embedding 参数在初始化时通常比较小。如果直接相加:

$$x_i = e_i + p_i \quad (248)$$

可能出现位置编码的数值幅度比词向量更大, 导致输入表示里“位置信息”压过“词语语义信息”。

乘以 $\sqrt{d_{\text{model}}}$ 后, token embedding 会被放大到更合适的尺度。例如原始 Transformer base 中:

$$d_{\text{model}} = 512 \quad (249)$$

$$\sqrt{d_{\text{model}}} \approx 22.6 \quad (250)$$

如果某个 embedding 分量大约是 0.02, 缩放后变成:

$$0.02 \times 22.6 \approx 0.45 \quad (251)$$

这就和正弦位置编码的 $[-1, 1]$ 范围更接近。

从向量尺度上也可以这样理解: embedding 不是单个数字, 而是 d_{model} 维向量。乘以 $\sqrt{d_{\text{model}}}$ 是在把 token embedding 的整体能量拉到和模型隐藏维度更匹配的量级, 避免它在和位置编码相加时过弱。

注意这和 attention 里的缩放方向相反:

1. embedding 乘以 $\sqrt{d_{\text{model}}}$: 放大词向量, 避免被位置编码淹没
2. attention score 除以 $\sqrt{d_k}$: 缩小点积分数, 避免 softmax 过尖

这个缩放不是所有现代模型都必须保留的设计。后来的模型可能使用可学习位置编码、RoPE、不同初始化方式或不同 LayerNorm 结构, 输入缩放方式也可能不同。但在原始 Transformer 中, 它是为了配合正弦位置编码和当时的 embedding 初始化尺度。

11.2.2. 为什么用正弦和余弦?

正弦位置编码有两个好处:

1. 每个位置都有独特编码
2. 相对位移可以通过线性关系表达

不同维度使用不同频率: 低频维度变化慢, 适合表示长距离位置; 高频维度变化快, 适合区分近邻位置。

参考视频: [算法魔法师: 为什么不用 1, 2, 3? 深度解密 Transformer 位置编码, 带公式推导!](#)

11.3. RoPE: 把位置写进 Q/K 的旋转角度

RoPE (Rotary Position Embedding, 旋转位置编码) 的核心不是“给 token embedding 加一个位置向量”, 而是:

1. 先像普通 attention 一样得到 Query 和 Key
2. 按 token 所在位置旋转 Query 和 Key
3. 再用旋转后的 Query、Key 计算点积

参考视频: [手撕 RoPE 旋转位置编码推导](#)

设第 t 个 token 在某个 attention head 中的 Query 和 Key 为:

$$q_t = x_t W^Q, \quad k_t = x_t W^K \quad (252)$$

RoPE 对它们做位置相关的旋转:

$$\tilde{q}_t = R_t q_t, \quad \tilde{k}_t = R_t k_t \quad (253)$$

然后 attention 分数变成:

$$s_{t,s} = \frac{\tilde{q}_t^T \tilde{k}_s}{\sqrt{d_h}} \quad (254)$$

其中 d_h 是单个 attention head 的维度。causal mask、padding mask 等仍然照常加到 $s_{t,s}$ 上；RoPE 本身不负责阻止当前位置看到未来。

11.3.1. 为什么是二维一组地旋转？

旋转天然发生在二维平面中。因此，RoPE 把一个 head 的维度两两分组：

$$(q_{t,0}, q_{t,1}), (q_{t,2}, q_{t,3}), \dots \quad (255)$$

对第 m 个二维分组，定义一个频率：

$$\theta_m = B^{-\frac{2m}{d_h}}, \quad m = 0, 1, \dots, \frac{d_h}{2} - 1 \quad (256)$$

B 是频率基数，经典取值是 10000，具体模型可以使用其他值。位置 t 对应的旋转角度是 $t\theta_m$ 。二维旋转矩阵为：

$$R(\varphi) = \begin{pmatrix} \cos \varphi & -\sin \varphi \\ \sin \varphi & \cos \varphi \end{pmatrix} \quad (257)$$

于是：

$$\begin{pmatrix} \tilde{q}_{t,2m} \\ \tilde{q}_{t,2m+1} \end{pmatrix} = R(t\theta_m) \begin{pmatrix} q_{t,2m} \\ q_{t,2m+1} \end{pmatrix} \quad (258)$$

Key 完全同理：

$$\begin{pmatrix} \tilde{k}_{t,2m} \\ \tilde{k}_{t,2m+1} \end{pmatrix} = R(t\theta_m) \begin{pmatrix} k_{t,2m} \\ k_{t,2m+1} \end{pmatrix} \quad (259)$$

不同维度对使用不同频率：前面的维度对旋转得快，擅长区分相近位置；后面的维度对旋转得慢，可以在更长距离上平滑变化。多个频率组合起来，为模型提供多尺度的位置信息。

11.3.2. 为什么点积会自然得到相对位置？

RoPE 最关键的性质来自旋转矩阵：

$$R_t^T R_s = R_{s-t} \quad (260)$$

因此，位置 t 的 Query 与位置 s 的 Key 的点积可以写成：

$$\tilde{q}_t^T \tilde{k}_s = (R_t q_t)^T (R_s k_s) = q_t^T R_t^T R_s k_s = q_t^T R_{s-t} k_s \quad (261)$$

虽然 Query 和 Key 分别使用了绝对位置 t 和 s 进行旋转，但二者做点积后，位置部分只通过相对距离 $s - t$ 出现。

定义 11.3.2.1: RoPE 的核心性质：先用绝对位置分别旋转 Query 和 Key，再通过点积自动得到依赖相对位置差的 attention 分数。

这并不表示 attention 分数只由距离决定。 q_t 和 k_s 仍然携带 token 内容；RoPE 只是让内容匹配同时受到相对距离影响：

$$\text{attention score} = \text{内容相关性} + \text{由旋转引入的位置关系} \quad (262)$$

这里的加号是直觉描述，不表示实现中真的单独计算两个标量再相加。实际实现仍然只是对旋转后的 Query 和 Key 做一次点积。

11.3.3. 用复数理解 RoPE

每个二维向量也可以看成一个复数：

$$z_{t,m} = q_{t,2m} + iq_{t,2m+1} \quad (263)$$

二维旋转等价于乘以单位复数：

$$\tilde{z}_{t,m} = z_{t,m} e^{it\theta_m} \quad (264)$$

Query 在位置 t 乘 $e^{it\theta_m}$ ，Key 在位置 s 乘 $e^{is\theta_m}$ ；二者计算内积时留下的相位差与：

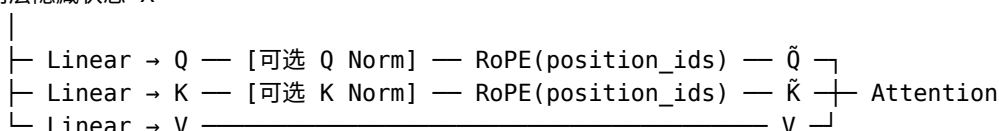
$$e^{i(s-t)\theta_m} \quad (265)$$

有关。这就是“绝对位置进去，相对位置出来”的另一种理解。

11.3.4. RoPE 在一层 Attention 中放在哪里？

典型顺序是：

当前层隐藏状态 X



对应公式为：

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V \quad (266)$$

$$\tilde{Q} = \text{RoPE}(Q, \text{position_ids}), \quad \tilde{K} = \text{RoPE}(K, \text{position_ids}) \quad (267)$$

$$\text{Attention}(X) = \text{softmax}\left(\frac{\tilde{Q}\tilde{K}^T}{\sqrt{d_h}} + \text{mask}\right)V \quad (268)$$

需要特别注意：

1. RoPE 通常每一层都作用在该层新计算出的 $\frac{Q}{K}$ 上，而不是只在进入第一层前做一次。
2. RoPE 通常只旋转 $\frac{Q}{K}$ ，不旋转 V 。
3. 有些模型只旋转 head 的部分维度，此时旋转维度 $d_r \leq d_h$ ；其余维度保持不变。
4. 代码可能把相邻维度配对，也可能先把向量分成前后两半再配对。只要频率和维度排列保持一致，这些是等价的布局约定。
5. 使用 QK Norm 的模型通常在 RoPE 之前分别归一化每个 head 的 Q/K；RoPE 是正交旋转，不会额外放大向量的欧氏范数。

11.3.5. RoPE、causal mask 和 KV Cache 的关系

RoPE 与 causal mask 解决的是两个不同问题：

- RoPE 回答：“两个可见 token 相隔多远、顺序如何？”
- causal mask 回答：“当前位置允许看哪些 token？”

在 decoder-only LLM 的 Decode 阶段，历史 token 的 Key 通常已经按各自位置完成 RoPE，然后作为 $\tilde{K}$ 写入 KV Cache；Value 不需要 RoPE，也一同缓存。生成新 token 时，只需：

- 1. 用新 token 的绝对位置旋转当前 Query 和 Key
- 2. 把旋转后的新 Key 和新 Value 追加到 KV Cache
- 3. 用当前旋转后的 Query 与缓存中的历史旋转 Key 计算分数

因此无需在每一步重新旋转全部历史 Key。

11.3.6. 与正弦位置编码的关系

RoPE 和原始正弦位置编码都使用多频率的 $\frac{\sin}{\cos}$ ，但注入位置的方式不同：

对比项	原始正弦位置编码	RoPE
作用位置	加到输入 embedding	旋转每层 attention 的 Q/K
主要形式	$X = E + PE$	$\tilde{Q} = RQ, \tilde{K} = RK$
位置信息	直接提供绝对位置向量	点积中自然表现为相对位置差
是否改变 V	输入变化会继续影响 Q/K/V	通常不直接旋转 V

Table 2: 正弦位置编码与 RoPE 的区别

11.3.7. RoPE 不等于无限长度外推

RoPE 的相对位置结构有利于长度泛化，但并不保证模型可以无损处理任意长的上下文。超过训练长度后，模型会遇到没有充分训练过的旋转相位和距离分布。长上下文模型常通过调整频率或缩放 position，例如线性缩放、动态缩放、NTK-aware scaling、YaRN 等方法扩展上下文。

这些方法的共同目标是：让更长位置范围内的旋转角度仍处于模型能够利用的分布中。但它们会改变不同频率上的距离分辨率，因此上下文长度、近距离精度和远距离稳定性之间仍然存在权衡。

11.4. RoPE 小结

RoPE 可以压缩成四句话：

- 1. 它不把位置向量加到 token embedding，而是旋转每层 attention 的 Query 和 Key。
- 2. 每两个维度组成一个旋转平面，不同维度对使用不同频率。
- 3. $R_t^T R_s = R_{s-t}$ ，所以 Q/K 点积自然携带相对位置差。
- 4. 在自回归推理中，缓存已经旋转过的历史 Key，可以直接与当前旋转后的 Query 计算 attention。

12. Transformer Encoder

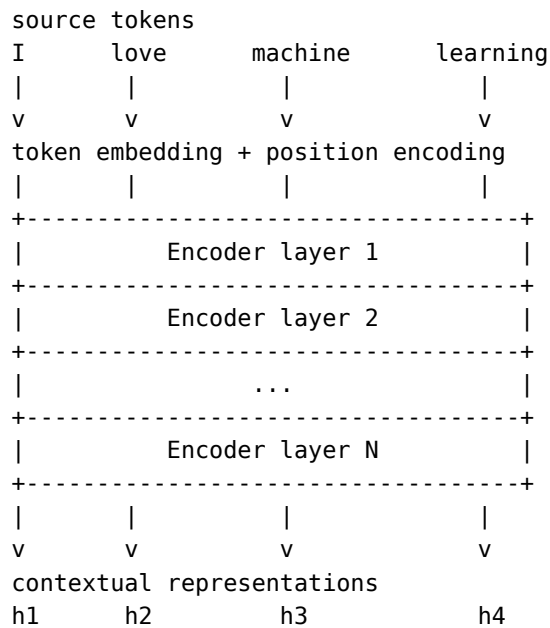
12.1. Encoder 在整个模型里的角色

Encoder 的任务是：把输入序列读完，并为每个输入 token 生成一个融合上下文后的表示。

在机器翻译里，输入可能是英文句子：

I love machine learning

Encoder 不直接生成中文。它先把每个英文 token 变成一个上下文表示。这个表示不只是该词本身的 embedding，而是“这个词在整句话中的含义”。



Listing 1: Encoder 把源序列编码成一组上下文表示

注意 Encoder 的输出仍然是一串向量，而不是一个单独向量。输入有 n 个 token，Encoder 输出通常也有 n 个位置：

$$H = \begin{pmatrix} h_1 \\ h_2 \\ \dots \\ h_n \end{pmatrix} \in \mathbb{R}^{n \times d_{\text{model}}} \quad (269)$$

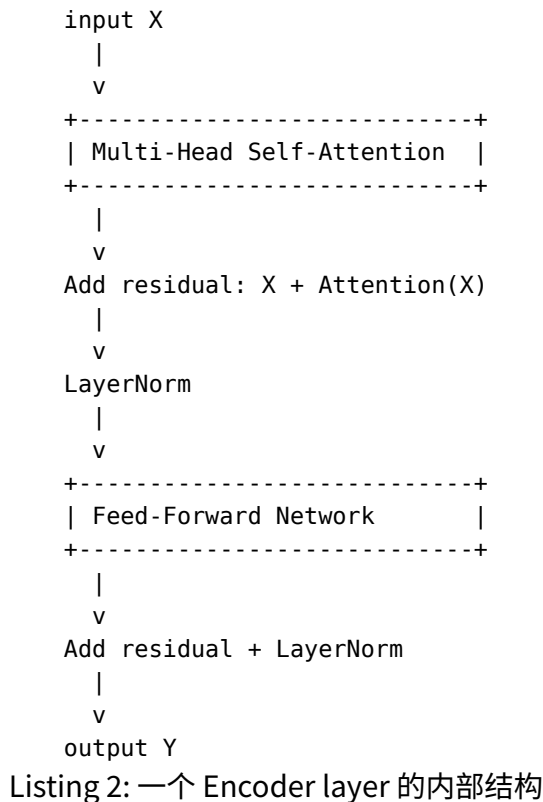
每个 h_i 都已经看过整个输入序列，所以它包含全局上下文。比如在句子 “I saw a bat” 中，“bat” 的最终表示会根据上下文更偏向 “蝙蝠” 还是 “球棒”。

12.2. Encoder 层结构

Transformer Encoder 由多个相同层堆叠而成。每层包含两个主要模块：

1. Multi-Head Self-Attention
2. Feed-Forward Network

每个模块外面都有残差连接和 LayerNorm。



用公式可以概括为：

$$X' = \text{LayerNorm}(X + \text{MultiHeadSelfAttention}(X)) \quad (270)$$

$$Y = \text{LayerNorm}(X' + \text{FFN}(X')) \quad (271)$$

其中 FFN 是逐位置前馈网络：

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (272)$$

这里的 X 是进入当前 Encoder 层的一整段序列表示，形状是 $n \times d_{\text{model}}$ 。Self-attention 会让每个位置查看所有输入位置：

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V \quad (273)$$

$$\text{SelfAttention}(X) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (274)$$

因为 Encoder 是在“理解输入”，不是在生成未来 token，所以它通常不需要 causal mask。第 1 个词可以看第 4 个词，第 4 个词也可以看第 1 个词。

Encoder self-attention visibility				
query \ key	x1	x2	x3	x4
x1	yes	yes	yes	yes
x2	yes	yes	yes	yes
x3	yes	yes	yes	yes
x4	yes	yes	yes	yes

Listing 3: Encoder 中每个位置通常都能看见完整输入序列

12.3. 残差连接有什么用？

残差连接形式是：

$$x + \text{SubLayer}(x) \quad (275)$$

它让模型可以在原表示基础上学习增量，而不是每层都完全重写表示。这样深层网络更容易训练，梯度也更容易传回前面层。

12.4. FFN 做什么？

Attention 负责 token 之间的信息交换，FFN 负责对每个 token 的表示做非线性变换。

注意 FFN 是逐位置应用的：同一个 FFN 作用到每个 token 上，但不同 token 之间不在 FFN 里交互。token 之间的交互主要发生在 attention 中。

可以粗略理解：

1. Attention：横向交流，token 之间互相看
2. FFN：纵向加工，每个 token 自己做特征变换

12.5. Encoder 输出给谁用？

Encoder 输出的 H 可以有不同用途：

1. 在机器翻译中， H 会交给 Decoder，作为源语言信息
2. 在文本分类中，可以取特殊 token 或池化后的表示做分类
3. 在抽取任务中，可以对每个 token 的输出做标注
4. 在检索和匹配任务中，可以把句子编码成向量用于相似度计算

所以 Encoder 更像一个“理解器”。它适合输入完整可见、需要全局理解的任务。

13. Transformer Decoder

13.1. Decoder 在整个模型里的角色

Decoder 的任务是：根据已经生成的前缀，预测下一个 token。

在机器翻译中，如果目标句子是：

我 喜欢 机器 学习

Decoder 不是一次凭空吐出整句话，而是按自回归方式生成：

1. 看到 <bos>，预测“我”
2. 看到 <bos> 我，预测“喜欢”
3. 看到 <bos> 我 喜欢，预测“机器”
4. 看到 <bos> 我 喜欢 机器，预测“学习”

generated prefix	next token
<bos>	--> 我
<bos> 我	--> 喜欢
<bos> 我 喜欢	--> 机器
<bos> 我 喜欢 机器	--> 学习

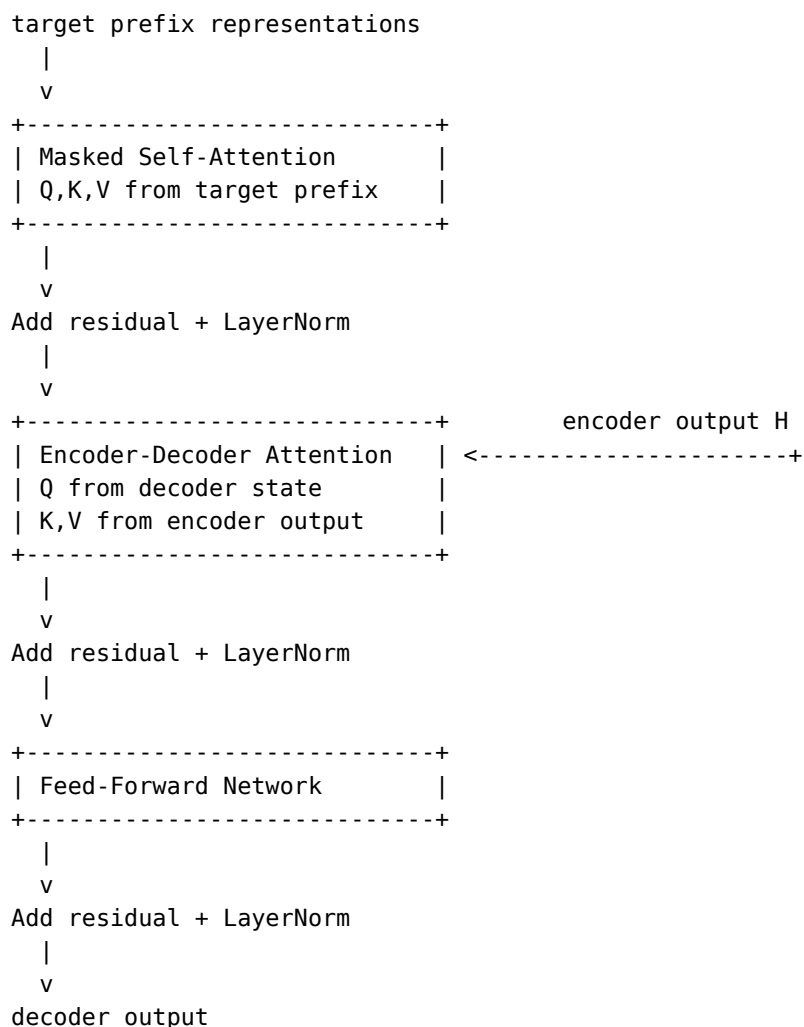
Listing 4: Decoder 以自回归方式逐步生成目标序列

Decoder 和 Encoder 的核心区别是：Decoder 不能看未来。预测“喜欢”时，它不能偷看后面的“机器学习”。这就是 masked self-attention 的原因。

13.2. 原论文 Decoder 层结构

原始 Transformer 用于机器翻译，所以 Decoder 层有三个主要模块：

1. Masked Self-Attention
2. Encoder-Decoder Attention
3. Feed-Forward Network



Listing 5: 原论文中的 Decoder layer 结构

用公式表示，可以粗略写成：

$$Z' = \text{LayerNorm}(Z + \text{MaskedSelfAttention}(Z)) \quad (276)$$

$$Z'' = \text{LayerNorm}(Z' + \text{CrossAttention}(Z', H)) \quad (277)$$

$$Y = \text{LayerNorm}(Z'' + \text{FFN}(Z'')) \quad (278)$$

这里 Z 是目标端已经生成 token 的表示， H 是 Encoder 输出的源序列表示。

13.3. Masked Self-Attention

生成任务必须从左到右生成。预测第 t 个 token 时，模型不能偷看未来的第 $t + 1$ 、 $t + 2$ 个 token。

因此 Decoder 的 self-attention 需要 mask。mask 会把未来位置的 attention 分数设为负无穷，使 softmax 后权重为 0。

这样第 t 个位置只能关注：

1. 已经生成的 token
2. 当前 token

不能关注未来 token。

Decoder masked self-attention visibility

query \ key	y1	y2	y3	y4
y1	yes	no	no	no
y2	yes	yes	no	no
y3	yes	yes	yes	no
y4	yes	yes	yes	yes

Listing 6: Decoder 的 causal mask 只允许当前位置看见自己和过去

这张表的含义是：

1. 第 1 个位置只能看第 1 个位置
2. 第 2 个位置可以看第 1、2 个位置
3. 第 3 个位置可以看第 1、2、3 个位置
4. 任何位置都不能看自己右侧的未来位置

所以 Decoder 的 self-attention 仍然是 self-attention，只是多了一个三角形 mask。

用公式写，masked self-attention 通常是：

$$A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right) \quad (279)$$

$$O = AV \quad (280)$$

其中 M 是 causal mask：

$$M_{i,j} = \begin{cases} 0 & j \leq i \\ -\infty & j > i \end{cases} \quad (281)$$

当 $j > i$ 时，表示 key 位置在 query 位置右侧，是未来 token。分数加上 $-\infty$ 后：

$$\exp(-\infty) = 0 \quad (282)$$

所以 softmax 后对应权重就是 0。

实现上不一定真的在显存里构造一个完整的 $n \times n$ mask 矩阵。长上下文下，显式 mask 本身就是一张大矩阵，会带来额外的 HBM 读写。高性能 attention kernel 通常根据 tile 的行列坐标判断哪些位置越过了因果边界，在片上计算时直接把这些分数置为极小值，或者跳过未来区域。

因此 causal mask 有两层含义：

1. 数学上，它保证自回归模型不能看见未来 token
2. 工程上，它应该尽量融合进 attention kernel，而不是作为一张单独的大矩阵反复读写

13.4. Encoder-Decoder Attention

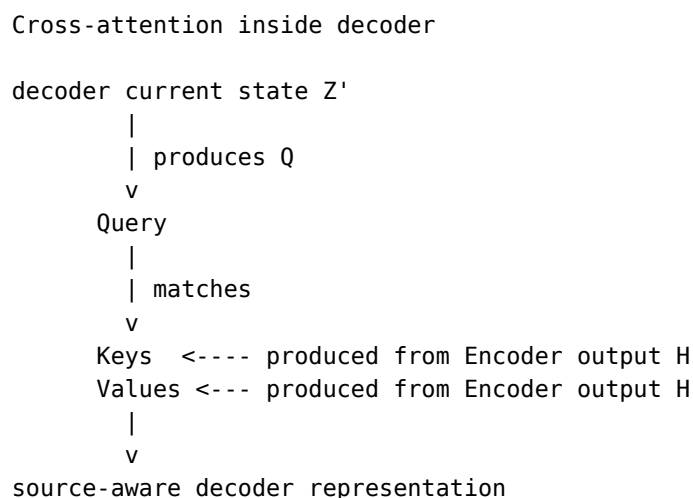
在机器翻译中，Decoder 生成目标语言时还需要查看源语言信息。Encoder-Decoder Attention 就负责这件事。

这里：

1. Query 来自 Decoder 当前状态
2. Key 和 Value 来自 Encoder 输出

也就是说，Decoder 用自己的 Query 去问：“源语言句子中哪些位置和我当前要生成的词最相关？”

这 and 传统 seq2seq 中的 attention 思想一致，只是 Transformer 把它系统化、并行化、多头化了。



Listing 7: Encoder-Decoder Attention 中 Q 来自 Decoder，K/V 来自 Encoder

对应公式是：

$$Q = Z'W^Q \tag{283}$$

$$K = HW^K \tag{284}$$

$$V = HW^V \tag{285}$$

$$\text{CrossAttention}(Z', H) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{286}$$

这里最重要的是 Q/K/V 的来源不同：

1. Decoder 当前状态生成 Query，表示“我现在要生成目标语言的某个位置，需要源语言里的什么信息”
2. Encoder 输出生成 Key，表示“源语言每个位置可以被什么问题匹配到”
3. Encoder 输出生成 Value，表示“源语言每个位置真正提供给 Decoder 的内容”

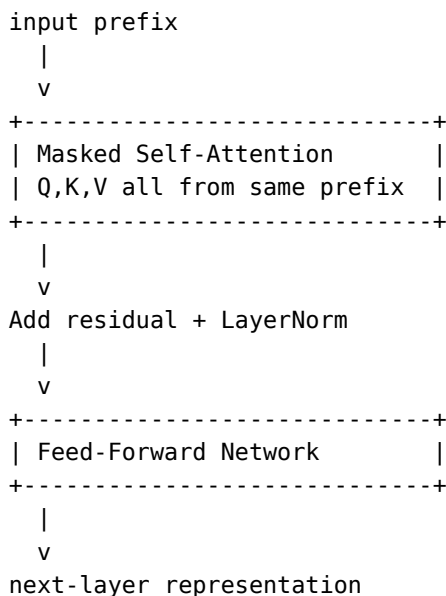
例如翻译到中文词“喜欢”时，Decoder 的 Query 可能会更多匹配英文中的“love”。翻译到“机器”时，Query 可能会更多匹配“machine”。

13.5. Decoder-only 模型和原论文 Decoder 的区别

GPT、Llama、Qwen 这类大语言模型通常是 decoder-only Transformer。它们名字里也叫 Decoder，但和原论文机器翻译里的 Decoder 不完全一样。

原论文 Decoder 有 cross-attention，因为它要一边看目标语言前缀，一边看 Encoder 给出的源语言句子。Decoder-only 模型没有单独的 Encoder，也就没有 Encoder-Decoder Attention。它只保留 masked self-attention 和 FFN：

Decoder-only language model layer



Listing 8: Decoder-only 模型删除了 cross-attention

这就是为什么 GPT 类模型可以把所有内容都放进同一个上下文里：

请把下面英文翻译成中文：I love machine learning

对 decoder-only 模型来说，“任务说明”和“待翻译文本”都只是同一个 token 序列中的前缀。模型通过 masked self-attention 看前文，然后预测后续 token。

因此可以这样区分：

1. **Encoder**：适合理解完整输入，所有位置互相可见
2. **原论文 Decoder**：适合 seq2seq 生成，目标端看过去，同时通过 cross-attention 看源端
3. **Decoder-only**：适合语言模型生成，所有信息都放在同一个自回归上下文里

14. 训练过程

14.1. Q、K、V 是如何训练出来的？

严格说，训练的不是 Q, K, V 本身，而是生成它们的参数矩阵 W^Q, W^K, W^V 。

Q, K, V 是一次前向计算中的中间结果。输入 X 变了，算出来的 Q, K, V 也会变：

$$Q = XW^Q \quad (287)$$

$$K = XW^K \quad (288)$$

$$V = XW^V \quad (289)$$

但 W^Q, W^K, W^V 是模型参数。训练的目标就是不断调整这些参数，让最终任务做得更好。

这里很容易误解：训练数据里并没有“正确的 Query 矩阵”“正确的 Key 矩阵”“正确的 Value 矩阵”。模型不会被直接监督说：

1. 这个 token 的 Query 应该是多少
2. 那个 token 的 Key 应该是多少
3. 某个 Value 向量应该长什么样

训练时，模型通常只知道最终任务答案。例如：

1. 语言模型：给定前文，预测下一个 token
2. 机器翻译：给定源语言句子，生成目标语言句子
3. 分类任务：给定文本，预测类别

所以 W^Q, W^K, W^V 是通过最终任务的损失函数间接学出来的。

训练过程可以概括为：

1. 随机初始化 W^Q, W^K, W^V ，一开始它们只是一些小随机数
2. 输入一批训练样本，得到 token 表示 X
3. 计算 $Q = XW^Q, K = XW^K, V = XW^V$
4. 计算 attention 分数 $\frac{QK^T}{\sqrt{d_k}}$
5. 经过 softmax 得到 attention 权重
6. 用这些权重加权 V ，得到新的 token 表示
7. 后续 Transformer 层继续计算，最后得到预测结果
8. 把预测结果和正确答案比较，得到损失函数 L
9. 通过反向传播计算 L 对所有参数的梯度，包括 W^Q, W^K, W^V
10. 用优化器更新参数，让下一次预测更接近正确答案

可以把它理解成：模型并不知道一开始应该关注谁，但如果某种关注方式能让最终答案更正确，这种关注方式对应的参数就会被强化；如果某种关注方式让最终答案更差，对应参数就会被削弱。

例如句子：

The animal didn't cross the street because it was tired.

如果任务要求模型理解“it”指向“animal”，那么让“it”的 Query 更容易匹配“animal”的 Key，通常会帮助模型降低损失。训练过程不会显式写下规则：

it 应该关注 animal

而是在大量类似样本中，反向传播不断调整 W^Q 和 W^K ，使得这种有用的匹配更容易发生。

再看 Value。即使 attention 权重找对了位置，如果被取回来的 Value 信息没有用，最终预测仍然可能错。所以 W^V 也会被训练：它要学会把 token 中对后续任务有用的信息放进 Value 表示里。

因此三组矩阵学到的是不同能力：

1. W^Q ：当前位置应该如何提出“我要找什么信息”
2. W^K ：当前位置应该如何暴露“我可以被什么问题匹配到”
3. W^V ：当前位置应该提供什么内容给其他位置使用

这些能力不是人工写死的，而是通过最终任务损失自动形成的。

14.2. 数据切分后，权重矩阵如何合并？

如果有非常大的训练数据，比如几十 TB 或上百 TB 文本，训练时当然不会一次性把所有数据塞进模型。数据会被切成很多小批次，也就是 batch。

关键是：切分的是数据，不是切分出很多套最终权重矩阵。

训练过程中通常只有一套模型参数，包括：

1. W^Q, W^K, W^V
2. 输出投影矩阵 W^O
3. FFN 里的权重矩阵
4. embedding 参数
5. LayerNorm 参数
6. 其他所有可学习参数

这些参数会被反复用于不同 batch。每个 batch 都会基于当前这套参数做一次前向计算、计算损失、反向传播、更新参数。

可以把单机训练过程写成：

初始化一套参数 W

```
for batch in 所有数据切片:
    用当前 W 处理这个 batch
    得到预测结果
    计算 loss
    反向传播得到梯度 grad
    用 grad 更新 W
```

所以训练不是：

```
100T 数据切成 100 份
每份训练出一套 W
最后合并 100 套 W
```

而是：

所有数据切片不断更新同一套 W

对于 W^Q, W^K, W^V 也是一样。它们始终是全局共享参数。不同 batch 会产生不同梯度，这些梯度不断修改同一套 W^Q, W^K, W^V ，让它们逐渐适应更大规模、更丰富的数据分布。

多 GPU 或多机器训练时，容易误以为每张 GPU 各自训练一套模型，最后再把模型权重合并。常见的数据并行并不是这样做的。

数据并行通常是：

1. 每张 GPU 上都有一份相同的当前模型参数
2. 不同 GPU 读取不同的数据 batch
3. 每张 GPU 分别做前向计算和反向传播，得到自己的梯度
4. 所有 GPU 把梯度合并，通常是求和或求平均
5. 每张 GPU 用合并后的梯度更新参数
6. 更新后，各 GPU 上的参数再次保持一致

例如有 4 张 GPU：

```
GPU 1: 用 batch A 算出 grad_1
GPU 2: 用 batch B 算出 grad_2
GPU 3: 用 batch C 算出 grad_3
GPU 4: 用 batch D 算出 grad_4
```

然后合并梯度：

$$\text{grad} = \frac{\text{grad}_1 + \text{grad}_2 + \text{grad}_3 + \text{grad}_4}{4} \quad (290)$$

再统一更新参数：

$$W = W - \eta \text{ grad} \quad (291)$$

其中 η 是学习率。

因此，多卡训练中合并的通常是梯度，不是最终的权重矩阵。权重矩阵会在每一步更新后同步保持一致。

可以把它理解成很多人同时批改不同作业。每个人看到的是同一个学生当前的答案版本，然后分别指出应该怎么改。最后把这些修改意见平均一下，再统一修改同一份答案。不是每个人各自培养一个学生，最后再把这些学生合成一个。

所以大规模训练的核心逻辑是：

1. 数据被切成大量 batch
2. 所有 batch 都服务于同一套模型参数
3. 每个 batch 产生一份梯度信号
4. 梯度不断更新同一套权重矩阵
5. 多 GPU 训练时通常合并梯度，而不是合并最终模型

大数据的作用不是生成很多套 W^Q, W^K, W^V ，而是让同一套矩阵经历足够多、足够多样的梯度更新，逐渐学到稳定且可泛化的表示方式。

14.3. 为什么训练可以并行？

RNN 的主要问题是时间步之间有强依赖。要计算第 t 个隐藏状态：

$$h_t = f(h_{t-1}, x_t) \quad (292)$$

必须先算出 h_{t-1} 。因此同一个序列内部很难把所有位置同时算完。

Self-attention 不一样。给定整句输入矩阵 X 后，可以一次性算出所有位置的 Query、Key、Value：

$$Q = XW^Q \quad (293)$$

$$K = XW^K \quad (294)$$

$$V = XW^V \quad (295)$$

这里的矩阵乘法同时处理所有 token。不是先算第 1 个 token，再算第 2 个 token，再算第 3 个 token；而是把所有 token 堆成矩阵，一次矩阵乘法算完。

接着 attention 分数也是一次矩阵乘法：

$$S = QK^T \quad (296)$$

如果序列长度是 n ，那么 $S \in \mathbb{R}^{n \times n}$ 。这个矩阵一次性包含所有 token 两两之间的匹配分数。第 1 行是第 1 个 token 对所有 token 的关注分数，第 2 行是第 2 个 token 对所有 token 的关注分数，以此类推。

然后对 S 的每一行做 softmax，也可以并行完成：

$$A = \text{softmax}\left(\frac{S}{\sqrt{d_k}}\right) \quad (297)$$

最后再一次矩阵乘法：

$$O = AV \quad (298)$$

得到所有位置更新后的表示 O 。所以一个 self-attention 层的核心计算可以看成：

$$O = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (299)$$

整个式子都是矩阵运算，非常适合 GPU 并行。

更具体地说，并行性体现在几个层面：

1. 同一个序列内，所有 token 的 Q, K, V 可以同时计算
2. 所有 token 两两之间的 attention 分数可以同时计算
3. 所有 attention head 可以同时计算
4. 一个 batch 里的多条句子可以同时计算
5. 矩阵乘法内部的大量乘加操作可以在 GPU 上并行计算

Decoder 训练时也可以并行，这一点容易困惑。虽然生成时必须从左到右一个 token 一个 token 地生成，但训练时目标句子已经完整给出。模型会把目标序列整体右移一位作为输入，让每个位置预测下一个 token。

例如目标序列是：

我 喜欢 机器 学习

训练时可以同时构造多个预测任务：

1. 看到开头符号，预测“我”
2. 看到“我”，预测“喜欢”
3. 看到“我 喜欢”，预测“机器”
4. 看到“我 喜欢 机器”，预测“学习”

这些位置会被放进同一个矩阵里一起计算。为了防止模型偷看未来 token，Decoder 使用 causal mask，把未来位置的 attention 分数设为负无穷。这样 softmax 之后，未来位置权重为 0。

所以训练时虽然所有位置一起算，但第 t 个位置的信息流仍然只能来自 t 及之前的位置。这就是为什么 Transformer decoder 可以并行训练，同时又不违反自回归生成规则。

需要区分训练和推理：

1. 训练时，完整答案已知，可以用 mask 一次性并行计算所有位置的损失
2. 推理时，未来 token 还没有生成，必须从左到右逐个生成

因此 Transformer 的训练并行性很强，但自回归语言模型的推理仍然有顺序依赖。

15. 高效 Attention 机制

标准 scaled dot-product attention 的核心公式是：

$$O = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (300)$$

如果序列长度是 n ，那么 QK^T 会产生一个 $n \times n$ 的分数矩阵。这个矩阵表达所有 token 两两之间的关系，因此计算量和显存开销都大致是 $O(n^2)$ 。

当上下文长度从几千变成几万、几十万时， n^2 会很快变成瓶颈。所以后来的高效 attention 大致分成两类：

1. **Linear Attention**：改变 attention 的数学形式，避免显式构造 $n \times n$ attention 矩阵，把复杂度尽量降到线性级别
2. **FlashAttention**：不改变标准 attention 的数学结果，而是改变 GPU 上的计算顺序，减少显存读写和中间矩阵存储

这两者名字都和高效 attention 有关，但本质很不一样。

15.1. Linear Attention

标准 attention 可以按第 i 个 query 写成：

$$o_i = \frac{\sum_j \exp\left(\frac{q_i \cdot k_j}{\sqrt{d_k}}\right) v_j}{\sum_j \exp\left(\frac{q_i \cdot k_j}{\sqrt{d_k}}\right)} \quad (301)$$

这里的麻烦在于 $\exp(q_i \cdot k_j)$ 同时依赖 q_i 和 k_j 。要得到所有 i, j 的权重，通常就要计算所有 query-key 配对，也就是 $n \times n$ 个分数。

Linear attention 的核心想法是：把 softmax kernel 换成或近似成一种可以拆开的形式。用一个特征映射 φ 表示：

$$\exp(q \cdot k) \approx \varphi(q)^T \varphi(k) \quad (302)$$

于是 attention 可以写成：

$$o_i = \frac{\sum_j \varphi(q_i)^T \varphi(k_j) v_j}{\sum_j \varphi(q_i)^T \varphi(k_j)} \quad (303)$$

因为 $\varphi(q_i)$ 只和当前 query 有关， $\varphi(k_j)$ 只和第 j 个 key 有关，可以把和 j 有关的部分先聚合起来：

$$S = \sum_j \varphi(k_j) v_j^T \quad (304)$$

$$z = \sum_j \varphi(k_j) \quad (305)$$

那么：

$$o_i = \frac{\varphi(q_i)^T S}{\varphi(q_i)^T z} \quad (306)$$

关键变化是：不再先构造完整的 $n \times n$ attention 权重矩阵，而是先把所有 key/value 汇总成 S 和 z ，再让每个 query 去读这个汇总状态。

如果 $\varphi(k)$ 的维度记作 r ， v 的维度记作 d_v ，那么：

$$S \in \mathbb{R}^{r \times d_v}, \quad z \in \mathbb{R}^r \quad (307)$$

这两个量的大小不随序列长度 n 变成 $n \times n$ 。因此 linear attention 的计算和显存通常可以做到近似 $O(n)$ ，更适合长序列。

对于 causal decoder，还可以写成前缀累积形式。第 i 个 token 只能看 $1 \dots i$ ：

$$S_i = \sum_{j \leq i} \varphi(k_j) v_j^T \quad (308)$$

$$z_i = \sum_{j \leq i} \varphi(k_j) \quad (309)$$

$$o_i = \frac{\varphi(q_i)^T S_i}{\varphi(q_i)^T z_i} \quad (310)$$

这样从左到右扫描时，只需要不断更新 S_i 和 z_i ：

$$S_i = S_{i-1} + \varphi(k_i) v_i^T \quad (311)$$

$$z_i = z_{i-1} + \varphi(k_i) \quad (312)$$

这也是 linear attention 和 RNN 有点相似的地方：历史信息被压缩进一个固定大小的状态里，当前 token 读取这个状态。

但 linear attention 也有代价。标准 softmax attention 会为每个 query 生成一行独立的、精确归一化的 n 维权重分布；linear attention 通常使用 kernel 近似或替代函数，把历史信息压缩进固定大小的统计量。这会带来几个影响：

1. 它通常不是标准 softmax attention 的完全等价计算
2. 长距离信息会被压缩，表达能力取决于 φ 的设计和维度
3. 在一些任务上速度和上下文长度更有优势，但效果未必总能直接替代标准 attention

一句话总结：linear attention 是用新的数学结构换取更低的序列长度复杂度。

15.2. FlashAttention

FlashAttention 解决的是另一个问题：标准 attention 在 GPU 上很慢，不只是因为乘法多，还因为中间矩阵太大、显存读写太多。

标准实现通常会显式产生：

$$S = \frac{QK^T}{\sqrt{d_k}} \quad (313)$$

$$A = \text{softmax}(S) \quad (314)$$

$$O = AV \quad (315)$$

其中 S 和 A 都是 $n \times n$ 矩阵。如果序列很长，这两个矩阵会占很多显存。更重要的是，GPU 需要把这些大矩阵反复写入和读出高带宽显存。很多时候瓶颈不是算术单元不会乘加，而是数据搬运太贵。

FlashAttention 的核心思想是：标准 attention 的结果不变，但不要把完整的 S 和 A 存下来。它把 Q, K, V 切成 block，在 GPU 的片上 SRAM 中分块计算：

1. 读入一小块 Q 和一小块 K, V

2. 计算这一小块的 attention 分数
3. 在 block 内做数值稳定的 softmax 累积
4. 立刻乘以对应的 V ，把结果累加到输出
5. 处理下一个 block

这样做的结果仍然等价于：

$$O = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (316)$$

但是实现过程中不会把完整的 $n \times n$ attention 矩阵写到显存里。

一个直观类比是：标准实现先把所有两两分数写成一张巨大表格，再根据表格算输出；FlashAttention 则是一边看表格的一小块，一边更新最终答案，看完就丢掉这块中间结果。

FlashAttention 最关键的技术点是 **online softmax**。softmax 需要知道整行的归一化分母：

$$\sum_j \exp(s_j) \quad (317)$$

看起来必须先拿到整行所有分数。但实际上可以分块维护每一行当前见过的最大值 m 和归一化和 l 。当新 block 到来时，用新的最大值重新缩放旧的累积量，再把新 block 的贡献加进去。这样就能在不保存整行分数的情况下，得到和完整 softmax 一致的结果。

更具体地说，设当前已经处理过一部分 key，维护：

$$m_{\text{old}} = \max \text{已处理分数} \quad (318)$$

$$l_{\text{old}} = \sum \text{已处理} \exp(s_j - m_{\text{old}}) \quad (319)$$

新 block 的分数记为 s_j ，先算：

$$m_{\text{blk}} = \max_j s_j \quad (320)$$

$$l_{\text{blk}} = \sum_j \exp(s_j - m_{\text{blk}}) \quad (321)$$

合并时新的最大值是：

$$m_{\text{new}} = \max(m_{\text{old}}, m_{\text{blk}}) \quad (322)$$

旧的归一化和与新 block 的归一化和都要换到同一个最大值基准下：

$$l_{\text{new}} = \exp(m_{\text{old}} - m_{\text{new}})l_{\text{old}} + \exp(m_{\text{blk}} - m_{\text{new}})l_{\text{blk}} \quad (323)$$

输出累积量也做同样的缩放修正。这样每个 block 都可以边算边丢，不需要把完整的 S 或 A 写回显存。

FlashAttention 前向常会保存每个 query row 的 LSE，也就是 log-sum-exp：

$$\text{LSE} = m + \log l \quad (324)$$

(FIXME 什么是 LSE，什么是 Split-K)

因为：

$$\text{softmax}(s_i) = \exp(s_i - \text{LSE}) \quad (325)$$

保存 LSE 的好处是，它把一行 softmax 的归一化信息压成一个 FP32 标量。反向传播需要 softmax 概率 P 时，可以重算局部 QK^T ，再用 $P_i = \exp(s_i - \text{LSE})$ 还原，而不是保存完整的 $P \in \mathbb{R}^{n \times n}$ 。这就是 FlashAttention 用重计算换显存、用数学恒等变换减少 HBM 压力的典型例子。

所以 FlashAttention 的特点是：

1. 数学上仍然是标准 softmax attention，不是 linear attention
2. 主要降低显存占用和显存访问量
3. 训练和 prefill 阶段收益很明显，因为这些阶段会处理较长序列的整块 attention
4. 对 decode 阶段也有优化版本，但单步 decode 的瓶颈经常还包括 KV cache 读取和 batch 调度

一句话总结：FlashAttention 不是换公式，而是把同一个公式以更适合 GPU 的方式算出来。

15.3. FlashAttention 和 CUTLASS 矩阵分块是什么关系？

FlashAttention 和 CUTLASS 里的矩阵分块确实有相似之处。它们都在做 tiling/blocking：把大矩阵切成适合 GPU thread block、warp、shared memory 和 register 的小块，让数据尽量在片上存储中复用，减少访问 HBM 的次数。

但它们不是同一层东西。

CUTLASS 更像是高性能 GEMM 的 building block。它关心的是一个矩阵乘法怎么快，例如：

$$C = AB \quad (326)$$

为了算这个矩阵乘法，CUTLASS 会把 A, B, C 切成 tile，让每个 thread block 或 warp 负责一小块，并用 Tensor Core/MMA 指令高效完成乘加。

FlashAttention 关心的是整个 attention 算子怎么快：

$$S = QK^T \quad (327)$$

$$A = \text{softmax}(S) \quad (328)$$

$$O = AV \quad (329)$$

如果把它直接拆成两个 GEMM 和一个 softmax，中间会产生两个很大的矩阵：

1. $S \in \mathbb{R}^{n \times n}$ ：attention score
2. $A \in \mathbb{R}^{n \times n}$ ：softmax 后的 attention 权重

普通实现会把 S 写回 HBM，再读出来做 softmax；把 A 写回 HBM，再读出来乘 V 。当序列长度 n 很大时，问题不只是 GEMM 慢，而是这些 $n \times n$ 中间矩阵的读写太贵。

FlashAttention 的关键不是“也分块”，而是 **跨算子融合和 IO-aware tiling**。它把：

matmul + scale + mask + softmax + matmul

放在一个 attention-specific kernel 里重新组织计算顺序。它按 block 计算 QK^T 的一部分，在片上存储中维护 online softmax 的最大值和归一化和，然后马上把这一块 softmax 的贡献累积到 V 上。最终只需要把输出 O 写回 HBM，不需要物化完整的 S 和 A 。

所以二者的关系可以理解为：

CUDA / Tensor Core / MMA

↓
CUTLASS：高性能 GEMM building block

|
FlashAttention: 针对 attention 的融合 kernel

|
vLLM: 请求调度、KV cache 管理、attention backend 选择

因此，vLLM 不能只“直接调用 CUTLASS”就解决 attention 性能问题。CUTLASS 可以把单个 GEMM 做得很快，但标准 attention 的瓶颈经常在中间矩阵的显存读写、softmax、mask、KV cache 访问和动态 batch 调度上。

从 vLLM 的角度看，attention 还有更具体的推理系统问题：

1. Prefill 阶段会处理较长 prompt，FlashAttention 可以减少 QK^T 和 softmax 中间矩阵的 HBM 读写。
2. Decode 阶段每次只生成一个 token，但要读取历史 KV cache，瓶颈经常是 memory bandwidth，而不只是矩阵乘法吞吐。
3. vLLM 的 PagedAttention 还要处理分页 KV cache、block table、prefix cache、continuous batching 等问题，这些不是一个普通 GEMM kernel 能表达的。
4. 实际 attention 里还包含 causal mask、scale、GQA/MQA、不同 layout、RoPE 等逻辑，需要 attention-specific 的 kernel 或 backend 配合。

一句话总结：CUTLASS 解决的是“一个矩阵乘法怎么快”；FlashAttention 解决的是“整个标准 attention 怎么避免写出巨大的中间矩阵”；vLLM 关注的是“真实 LLM 推理中 attention、KV cache 和 batching 怎么整体快”。

15.4. Linear Attention 和 FlashAttention 的区别

标准 Attention:

$QK^T \rightarrow n \times n$ 分数矩阵 $\rightarrow$ softmax $\rightarrow$ 乘 V
数学精确，复杂度 $O(n^2)$

Linear Attention:

用 $\phi(q)^T \phi(k)$ 近似或替代 softmax kernel
先聚合 K/V ，再让 Q 读取聚合状态
改变数学形式，复杂度可接近 $O(n)$

FlashAttention:

仍然计算标准 $\text{softmax}(QK^T)V$
但分块计算，不保存完整 $n \times n$ 中间矩阵
数学结果等价，主要优化显存和访存

Listing 9: Linear Attention 和 FlashAttention 的核心区别

可以把三者放在一起看：

1. **标准 attention**：最直接，表达能力强，但长序列下 $O(n^2)$ 昂贵
2. **linear attention**：牺牲或改变一部分 attention 形式，换取更长上下文的线性复杂度
3. **flash attention**：保留标准 attention 结果，通过更好的 GPU kernel 减少显存和访存开销

因此，如果一个模型论文说自己用了 linear attention，通常意味着模型结构或 attention 公式发生了变化；如果一个推理/训练框架说支持 FlashAttention，通常意味着它在更高效地实现标准 attention。

16. Multi-Head Latent Attention (MLA)

16.1. 动机：KV Cache 为什么是瓶颈？

在自回归推理中，每生成一个新 token，模型需要回顾所有历史 token 的 Key 和 Value。如果每一层、每个头都把完整的 K 和 V 重新计算一遍，推理成本会随序列长度平方级增长。工程上的标准做法是：把已经算好的 K 和 V 缓存起来。这就是 KV Cache。

对标准 MHA (Multi-Head Attention)，假设模型有 L 层、 h 个头，每个头的 Key/Value 维度为 d ，使用 FP16 精度，序列长度为 n ，batch 大小为 b 。那么 KV Cache 的显存占用为：

$$\text{KV Cache size} = 2 \times L \times h \times d \times n \times b \times 2 \text{ bytes} \quad (330)$$

其中系数 2 来自同时存储 Key 和 Value，最后的 $\times 2$ bytes 来自 FP16 每元素占 2 字节。以 DeepSeek-V2 为例， $L = 60$ ， $h = 128$ ， $d = 128$ （每个头的维度）。当上下文长度达到 128K token 时，仅 batch size 为 1 的 KV Cache 就已经是天文数字：

$$\text{KV Cache} \approx 2 \times 60 \times 128 \times 128 \times 131072 \times 2 \approx 515.4 \text{ GB} \quad (331)$$

这相当于约 480 GiB，远远超过一个 80GB A100 GPU 的容量。换句话说，标准 MHA 在长上下文推理时，KV Cache 本身就会成为显存瓶颈。

这个问题的根源在于：MHA 为每个头都存储了完整的 K 和 V 向量。 $L \times h \times d$ 就是总参数量级的“缓存维度”。一种直觉是：这些不同头的 Key 和 Value 之间很可能存在大量信息冗余。如果能找到一种压缩表示，在不丢失太多信息的前提下大幅降低 KV Cache 的大小，长上下文推理的显存压力就能得到根本缓解。

16.2. MQA/GQA 的局限：共享不等于压缩

在 MLA 之前，业界已有两种降低 KV Cache 的主流方案：MQA (Multi-Query Attention) 和 GQA (Grouped-Query Attention)。

MQA 的做法最激进：所有 Query 头共享同一组 Key 和 Value。假设原来有 h 个头，MQA 只有 1 组 KV 头。KV Cache 缩减为原来的 $\frac{1}{h}$ 。代价是模型表达能力明显下降——不同 Query 头被迫使用同样的 Key/Value 进行检索，无法从不同角度关注上下文。

GQA 是 MQA 和 MHA 的折中：将 h 个 Query 头分成 g 组，每组共享一组 Key 和 Value。当 $g = h$ 时退化为 MHA，当 $g = 1$ 时退化为 MQA。GQA 在 $g = 4$ 或 $g = 8$ 时通常能取得较好的性价比，但本质仍然是在“表达能力”和“缓存大小”之间做粗粒度的取舍。

MQA 和 GQA 的共同问题是：它们通过**显式共享** KV 头来减少缓存。共享意味着多个 Query 头被强制使用同样的 Key/Value 检索空间，每个头失去了独立的“观察角度”。随着模型规模和任务复杂度增长，这种共享带来的信息损失会越来越明显。

MLA 走了一条不同的路：不共享头，而是压缩每个头的 KV 表示。

16.3. MLA 的核心思路：压缩而非共享

MLA 的核心想法可以用一句话概括：**不再为每个头单独存储完整的 Key 和 Value，而是把所有头的 Key 和 Value 的信息压缩进一个低维的“潜在向量” (latent vector)，推理时只缓存这个潜在向量；计算注意力时，再从潜在向量中动态解压出各个头的真实 Key 和 Value。**

这与 MQA/GQA 有本质区别：

- MQA/GQA 问的是：“哪些头可以共用一组 KV？”
- MLA 问的是：“能不能用一个更小的向量，同时编码所有头 KV 的信息？”

前者是在“配置”层面减少 KV 头的数量，后者是在“表示”层面压缩 KV 的内容。MLA 保留了每个头独立的解压参数，因此每个头仍然可以从同一个压缩表示中“读出”不同的 K 和 V——表达能力比 GQA 更强，但缓存大小比 MHA 小得多。

具体来说，在暂时忽略位置编码时，MLA 不再缓存每个 head 展开后的 K/V，而是为每个历史 token 缓存一个更小的 KV 联合压缩向量 c_t^{KV} 。加入实际使用的 Decoupled RoPE 后，还需要额外缓存一个很小的位置 Key k_t^R 。

因此，DeepSeek-V2 风格 MLA 每层、每个历史 token 实际缓存的是：

$$(c_t^{KV}, k_t^R) \quad (332)$$

不是只为整段序列缓存一个 c ，也不是缓存展开后的完整 K/V。序列中的每个 token 在每一层都有自己的一对 (c_t^{KV}, k_t^R) 。

DeepSeek-V2 中 $d_c = 512$ 、 $d_r = 64$ ，所以每层、每 token 缓存 $512 + 64 = 576$ 个元素。作为对比，具有 $h = 128$ 个 head、每个 head 维度 $d = 128$ 的标准 MHA，需要缓存：

$$2hd = 2 \times 128 \times 128 = 32768 \quad (333)$$

个元素。这里的系数 2 表示 K 和 V。MLA 把每 token、每层的缓存宽度从 32768 降到 576，约缩小 56.9 倍。

16.4. 公式推导

16.4.1. 输入和符号约定

设当前层的输入为 $X \in \mathbb{R}^{n \times d_{\text{model}}}$ ，其中 n 是序列长度， d_{model} 是模型隐藏维度。在标准 MHA 中，每个头 i 独立计算：

$$Q_i = XW_i^Q, \quad K_i = XW_i^K, \quad V_i = XW_i^V \quad (334)$$

其中 $W_i^Q, W_i^K \in \mathbb{R}^{d_{\text{model}} \times d}$ ， $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d}$ ， d 是每个头的维度。

MLA 的做法分三步：压缩、缓存、解压。

16.4.2. 第一步：压缩 (Down-Projection)

MLA 首先将输入 X 投影到一个低维潜在空间：

$$C^{KV} = XW^{\text{DKV}} \quad (335)$$

其中 $W^{\text{DKV}} \in \mathbb{R}^{d_{\text{model}} \times d_c}$ ， d_c 是 KV 压缩维度（例如 512）。 $C^{KV} \in \mathbb{R}^{n \times d_c}$ 的第 t 行就是第 t 个 token 的压缩向量 c_t^{KV} 。

这个压缩向量同时承载所有 head 的内容 Key 和 Value 信息：不是每个 head 各自缓存一个压缩表示，而是所有 head 从同一个 c_t^{KV} 中读取不同信息。需要注意， C^{KV} 还不是实际 MLA 的全部缓存；Decoupled RoPE 使用的位置 Key K^R 也要缓存，后面会详细说明。

16.4.3. 第二步：解压 (Up-Projection)

计算注意力时，概念上可以从 C^{KV} 中分别解压出每个 head 的内容 Key 和 Value：

$$K_i^C = C^{KV} W_i^{\text{UK}} \quad (336)$$

$$V_i^C = C^{KV} W_i^{\text{UV}} \quad (337)$$

其中 $W_i^{\text{UK}} \in \mathbb{R}^{d_c \times d}$ ， $W_i^{\text{UV}} \in \mathbb{R}^{d_c \times d}$ 是每个 head 独立的上投影矩阵。

也就是说，虽然所有 head 共享同一个压缩表示 C^{KV} ，但每个 head 有自己的投影参数 W_i^{UK} 和 W_i^{UV} 。不同 head 可以从同一个潜在向量中“读出”不同的 Key 和 Value。这比 GQA 的“硬共享”更灵活。

16.4.4. 第三步：Query 投影和注意力计算

DeepSeek-V2 还会压缩 Query，以减少训练时的 activation memory：

$$C^Q = XW^{DQ} \quad (338)$$

$$Q_i^C = C^Q W_i^{UQ} \quad (339)$$

Query 压缩与 KV Cache 大小无关。Decode 时只使用当前新 token 的 Query，所以 c_t^Q 不会像历史 c_t^{KV} 那样进入长期缓存。

暂时忽略 RoPE 时，可以写成：

$$\text{head}_i = \text{softmax}\left(\frac{Q_i^C (K_i^C)^T}{\sqrt{d}}\right) V_i^C \quad (340)$$

$$\text{MLA}(X) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) W^O \quad (341)$$

16.4.5. 推理时的关键优化：矩阵吸收

上面描述的“先从 C^{KV} 上投影得到完整内容 K/V，再计算 attention”是概念上的做法。在实际推理中，MLA 利用矩阵乘法结合律，从而避免显式展开所有历史 token 的完整 K/V。

使用行向量记法，观察单个 Query 和历史 token j 的内容 attention 分数：

$$q_i^C (k_{j,i}^C)^T = q_i^C (c_j^{KV} W_i^{UK})^T \quad (342)$$

根据矩阵乘法结合律：

$$q_i^C (k_{j,i}^C)^T = (q_i^C (W_i^{UK})^T) (c_j^{KV})^T \quad (343)$$

也就是说，可以把 Key 上投影矩阵吸收到 Query 一侧，让变换后的 Query 直接和缓存的 c_j^{KV} 做点积，不必为所有历史 token 显式恢复完整的内容 Key。

在进入 Value 侧之前，还需要先把 Query-Key 匹配分数变成 attention 权重。这里关注 Decode 阶段的一个当前 Query，并约定：

1. i 表示第 i 个 attention head
2. j 表示第 j 个历史 token
3. 当前 cache 中共有 t 个可见 token

暂时忽略 RoPE 时，第 i 个 head 对第 j 个历史 token 的分数是：

$$s_{ij} = \frac{q_i^C (k_{j,i}^C)^T}{\sqrt{d}} \quad (344)$$

对同一个 head 的全部历史分数做 softmax：

$$\alpha_{ij} = \frac{\exp(s_{ij})}{\sum_{m=1}^t \exp(s_{im})} \quad (345)$$

因此：

$$\alpha_{ij} \geq 0, \quad \sum_{j=1}^t \alpha_{ij} = 1 \quad (346)$$

α_{ij} 的含义是：当前 Query 在第 i 个 head 中，应该从第 j 个历史 token 读取多少比例的信息。它由当前 Query 和历史 Key 临时计算出来，不是模型参数，也不会写入 KV Cache；当前 attention 计算结束后即可释放。

得到 α_{ij} 后，Value 侧再使用这些权重做加权求和：

$$\sum_{j=1}^t \alpha_{ij} v_{j,i}^C = \sum_{j=1}^t \alpha_{ij} (c_j^{\text{KV}} W_i^{\text{UV}}) \quad (347)$$

$$= \left(\sum_{j=1}^t \alpha_{ij} c_j^{\text{KV}} \right) W_i^{\text{UV}} \quad (348)$$

所以整个过程的顺序是：先由 Query 和 Key 产生 α_{ij} ，再用 α_{ij} 聚合 Value。由于 Value 上投影 W_i^{UV} 对所有历史 token 相同，可以先直接对缓存的 latent vector 做加权求和，再通过 Value 上投影与输出投影的组合得到最终输出。实际实现不必把全部历史 K/V 长期展开在显存中。

定义 16.4.5.1: 矩阵吸收 (Matrix Absorption)：利用矩阵乘法结合律，把 Key 上投影吸收到 Query 侧，把 Value 上投影吸收到输出侧。推理时可以围绕缓存的 C^{KV} 直接完成 attention，避免长期存储或完整物化所有 head 的 K/V。

16.4.6. 位置编码的难题：Decoupled RoPE

MLA 引入了一个新的技术挑战：旋转位置编码 (RoPE) 与低秩压缩的兼容性。

RoPE 的核心操作是在计算注意力分数之前，对 Query 和 Key 向量施加旋转变换：

$$\text{score}_{i,j} = (R_{\Theta,i} q_i)^T (R_{\Theta,j} k_j) \quad (349)$$

其中 $R_{\Theta,i}$ 是位置 i 对应的旋转矩阵。RoPE 之所以有效，是因为它直接在 Key 向量的**每个维度**对上编码了位置差值信息。这要求 Key 向量是“位置可分的”——向量的第 $2m$ 和第 $2m+1$ 维构成一个二维旋转平面。

问题在于：MLA 的内容 Key K_i^C 是从 $C^{\text{KV}} W_i^{\text{UK}}$ 上投影得到的。如果直接在 K_i^C 上施加 RoPE，矩阵吸收就会失效——因为 $R_{\Theta,j}$ 依赖位置 j ，会夹在 Query 与 Key 上投影矩阵之间，无法作为固定模型参数提前合并。于是推理时将不得不为所有历史 token 重新展开内容 Key。

DeepSeek 的解决方案叫 **Decoupled RoPE (解耦 RoPE)**：

1. 从 C^{KV} 中上投影得到的部分记为 K_i^c (内容 Key)，**不加 RoPE**
2. 额外保留一小段位置 Key K^r ，直接从输入 X 投影得到并施加 RoPE；这部分 Key 在所有 head 之间共享
3. 最终的 Key 是两个部分的拼接：

$$K_i = \text{Concat}(K_i^c, K^r) \quad (350)$$

其中 $K_i^c \in \mathbb{R}^{n \times d}$ (从 C^{KV} 上投影得到, 无位置编码), $K^r \in \mathbb{R}^{n \times d_r}$ (从 X 直接投影得到, 施加 RoPE)。 d_r 通常很小, 例如 DeepSeek-V2 使用 $d_r = 64$, 而内容 head 维度 $d = 128$ 。

Query 也做类似的拆分:

$$Q_i = \text{Concat}(Q_i^c, Q_i^r) \quad (351)$$

其中 Q_i^c 不施加 RoPE, Q_i^r 施加 RoPE。

Q_i^c 对应的内容 Key K_i^c 可以享受矩阵吸收优化, 因为它没有 RoPE。 Q_i^r 和共享的 K^r 负责位置匹配; K^r 需要被缓存, 但 d_r 很小, 所以额外成本可控。

定义 16.4.6.1: Decoupled RoPE: 将 Key (和 Query) 拆分为两部分——内容部分 (无 RoPE, 可被矩阵吸收, 从潜在向量解压) 和位置部分 (施加 RoPE, 直接从输入投影, 维度很小)。这是 MLA 在维持低秩压缩效率的同时保留旋转位置编码能力的关键设计。

因此, 第 l 层的 KV Cache 中实际存储:

1. 每个历史 token 的 KV 联合压缩向量 $c_t^{\text{KV}, l}$, 维度为 d_c
2. 每个历史 token 的共享位置 Key $k_t^{R, l}$, 维度为 d_r

如果当前已经有 n 个 token, 那么这一层缓存的两个矩阵可以写成:

$$C_{1:n}^{\text{KV}, l} \in \mathbb{R}^{n \times d_c} \quad (352)$$

$$K_{1:n}^{R, l} \in \mathbb{R}^{n \times d_r} \quad (353)$$

所以每层缓存 $n(d_c + d_r)$ 个元素; L 层总共是:

$$Ln(d_c + d_r) \quad (354)$$

相比之下, 标准 MHA 需要缓存 $Ln(2hd)$ 个元素。

定义 16.4.6.2: DeepSeek-V2 风格 MLA 缓存的是每层、每个历史 token 的 (c_t^{KV}, k_t^R) 。 c_t^{KV} 联合编码所有 head 的内容 K/V, k_t^R 单独保存位置; 完整的内容 K/V 和 Query 压缩向量 c_t^Q 都不长期缓存。

Decode 第 $t + 1$ 个 token 时, 模型只新计算当前 token 的 Query。它对历史 token j 的匹配分数可以理解为内容分数与位置分数之和:

$$\text{score}_{t+1, j, i} = \frac{q_{t+1, i}^C (k_{j, i}^C)^T + q_{t+1, i}^R (k_j^R)^T}{\sqrt{d + d_r}} \quad (355)$$

其中内容部分可通过矩阵吸收直接围绕缓存的 c_j^{KV} 计算, 位置部分直接读取缓存的 k_j^R 。这就是为什么实际 cache 中既要有 c^{KV} , 又要有一小段 k^R 。

16.5. MLA 的 KV Cache 如何工作?

“KV Cache” 是一个沿用下来的名字。标准 MHA 在里面放展开后的 K 和 V; MLA 仍然把这块推理状态称为 KV Cache, 但其中实际放的是可以参与后续 attention 的压缩状态。

对第 l 层、长度为 n 的序列，逻辑上缓存两个矩阵：

$$C_{\text{cache}}^{\text{KV},l} \in \mathbb{R}^{n \times d_c} \quad (356)$$

$$K_{\text{cache}}^{R,l} \in \mathbb{R}^{n \times d_r} \quad (357)$$

工程实现通常把二者拼在同一条 cache record 中：

$$Z_{\text{cache}}^l = \text{Concat}(C_{\text{cache}}^{\text{KV},l}, K_{\text{cache}}^{R,l}) \in \mathbb{R}^{n \times (d_c + d_r)} \quad (358)$$

所以在 DeepSeek-V2 的典型配置中，每个 token、每层只写入一条 576 元素的记录：前 512 个元素是 c_t^{KV} ，后 64 个元素是 k_t^R 。这里的“拼接”是存储布局，不表示内容和位置在数学上承担相同角色。

16.5.1. Prefill：一次生成并写入整段 Prompt 的 Cache

假设 prompt 长度为 T 。在第 l 层，模型先得到这一层所有 token 的输入：

$$H^l \in \mathbb{R}^{T \times d_{\text{model}}} \quad (359)$$

然后一次性计算需要长期保留的两部分：

$$C^{\text{KV},l} = H^l W_l^{\text{DKV}} \in \mathbb{R}^{T \times d_c} \quad (360)$$

$$K^{R,l} = \text{RoPE}(H^l W_l^{\text{KR}}) \in \mathbb{R}^{T \times d_r} \quad (361)$$

它们被写入第 l 层的 KV Cache：

$$Z_{\text{cache}}^l = \text{Concat}(C^{\text{KV},l}, K^{R,l}) \quad (362)$$

Prefill 还会计算所有 prompt token 的 Query、attention 分数和当前层输出，但这些都是当前计算的临时数据。完成这一层后，长期留下来的只有 $C^{\text{KV},l}$ 和 $K^{R,l}$ 。下一层会根据自己的输入和参数，建立另一份独立的 MLA cache。

因此，一个 L 层模型不是只有一份 C^{KV} ，而是每层都有自己的 cache：

$$\{Z_{\text{cache}}^1, Z_{\text{cache}}^2, \dots, Z_{\text{cache}}^L\} \quad (363)$$

16.5.2. Decode：每步只追加一条 Cache Record

假设当前已经缓存 t 个 token。生成第 $t+1$ 个 token 时，第 l 层拿到当前 token 的隐藏状态 h_{t+1}^l ，先计算：

$$c_{t+1}^{\text{KV},l} = h_{t+1}^l W_l^{\text{DKV}} \quad (364)$$

$$k_{t+1}^{R,l} = \text{RoPE}(h_{t+1}^l W_l^{\text{KR}}) \quad (365)$$

然后把新记录追加到这一层的 cache：

$$Z_{\text{cache}}^l \leftarrow [Z_{\text{cache}}^l; \text{Concat}(c_{t+1}^{\text{KV},l}, k_{t+1}^{R,l})] \quad (366)$$

加入当前记录后，这一层的 cache 长度从 t 变成 $t+1$ 。当前 token 因而可以在 causal self-attention 中读取历史 token，也可以读取自己。

与此同时，只为当前 token 计算 Query。对第 i 个 Query head，把内容 Query 变换到 latent 维度：

$$\tilde{q}_{t+1,i}^C = q_{t+1,i}^C (W_i^{\text{UK}})^T \in \mathbb{R}^{d_c} \quad (367)$$

然后直接扫描 cache 中第 j 条记录，计算：

$$s_{ij} = \frac{\tilde{q}_{t+1,i}^C (c_j^{\text{KV}})^T + q_{t+1,i}^R (k_j^R)^T}{\sqrt{d + d_r}} \quad (368)$$

对所有 $j \leq t + 1$ 的分数做 softmax：

$$\alpha_{ij} = \text{softmax}_j(s_{ij}) \quad (369)$$

Value 侧不必先展开所有历史 V_i^C 。可以直接对 latent cache 做加权求和：

$$z_i = \sum_{j=1}^{t+1} \alpha_{ij} c_j^{\text{KV}} \in \mathbb{R}^{d_c} \quad (370)$$

再通过 Value 上投影与输出投影得到当前 head 对最终输出的贡献：

$$o_i = z_i W_i^{\text{UV}} \quad (371)$$

$$y = \text{Concat}(o_1, \dots, o_h) W^O \quad (372)$$

实际 kernel 可以把 W_i^{UV} 与 W^O 进一步组合。当前 Query、attention 分数和中间输出在本步结束后即可释放；新写入的 $(c_{t+1}^{\text{KV}}, k_{t+1}^R)$ 会留下，供第 $t + 2$ 个 token 使用。

第 l 层，Decode 第 $t+1$ 个 token

当前隐藏状态 $h_{(t+1)}^l$

```

|
+---> c_{(t+1)}^{KV} -----+
|                               | 追加
+---> k_{(t+1)}^R -----+-----> [c_1^{KV}, k_1^R ... c_{(t+1)}^{KV}, k_{(t+1)}^R]
|                               |
+---> 当前 Query -----+-----> attention --> 当前层输出

```

长期保留： $c_{(t+1)}^{KV}$ 、 $k_{(t+1)}^R$

本步释放：当前 Query、attention 分数、softmax 权重和临时展开结果

Listing 10: MLA 在单层 Decode 中追加并读取 KV Cache 的过程

16.5.3. 推理 Backend 不一定采用同一种计算路径

缓存格式和计算路径是两件不同的事。长期 cache 可以始终保存压缩的 (C^{KV}, K^R) ，attention kernel 则可以根据硬件和阶段选择不同方法：

1. **Latent-native 路径**：通过矩阵吸收，让变换后的 Query 直接读取 C^{KV} ，并直接对 latent vector 做加权求和。Decode 阶段特别适合这种方式，因为它能显著减少每步读取的数据量。
2. **Compute-friendly 路径**：临时从 C^{KV} 上投影出内容 K/V，再调用类似普通 MHA/FlashAttention 的 kernel。Prefill 有大量 Query，某些实现会选择这种更适合大矩阵计算的路径，并通过分块限制临时 workspace。

无论 backend 选择哪条计算路径，临时展开的 K/V 都不会变成长期 cache。下一轮仍然读取压缩的 C^{KV} 和 K^R 。

16.5.4. Cache 的分配、分页和释放

在 vLLM 一类推理框架中，MLA cache 仍然可以使用分页管理：

1. 调度器为请求分配 cache block
2. Prefill 把 prompt 对应的 (c_t^{KV}, k_t^R) 写入这些 block
3. 每次 Decode 在逻辑序列末尾追加一条记录
4. attention backend 通过 block table 找到当前请求的历史记录
5. 请求结束后释放 block；如果启用 prefix cache，可让其他具有相同前缀的请求复用已有 block

PagedAttention 改变的是 cache record 如何分块、寻址和复用，并不改变 MLA record 中保存的数学内容。

16.5.5. 每一步读写多少数据？

对单个请求、单层而言，第 $t + 1$ 步 Decode：

1. 新增写入 $d_c + d_r$ 个元素
2. 为 attention 读取大约 $(t + 1)(d_c + d_r)$ 个历史元素
3. 不写入 $2hd$ 维的完整多头 K/V

因此 MLA KV Cache 的容量仍然随 token 数量线性增长，Decode 单步读取量也仍然是 $O(t)$ 。MLA 降低的是每个历史 token 的 cache 宽度和内存带宽压力，并没有把标准全注意力的 Decode 复杂度变成 $O(1)$ 。

16.6. KV Cache 大小对比

假设模型参数与 DeepSeek-V2 类似： $L = 60$ 层， $h = 128$ 头， $d = 128$ （每头维度）， $d_c = 512$ （潜在向量维度）， $d_r = 64$ （RoPE 专用维度），FP16 精度。

MHA: 每层 KV Cache = $2 \times h \times d = 2 \times 128 \times 128 = 32768$ 维
128K 上下文 $\approx 60 \times 32768 \times 131072 \times 2B \approx 515.4$ GB

GQA(g=8): 每层 KV Cache = $2 \times g \times d = 2 \times 8 \times 128 = 2048$ 维
128K 上下文 $\approx 60 \times 2048 \times 131072 \times 2B \approx 32.2$ GB

GQA(g=4): 每层 KV Cache = $2 \times 4 \times 128 = 1024$ 维
128K 上下文 ≈ 16.1 GB

MQA: 每层 KV Cache = $2 \times 1 \times 128 = 256$ 维
128K 上下文 ≈ 4.0 GB

MLA: 每层 KV Cache = $d_c + d_r = 512 + 64 = 576$ 维
128K 上下文 $\approx 60 \times 576 \times 131072 \times 2B \approx 9.1$ GB

Listing 11: 不同注意力机制的每层 KV Cache 维度和 128K 上下文下的估算显存占用（FP16）

以上按十进制 GB 计算；对应的二进制 GiB 约为 MHA 480 GiB、GQA (g=8) 30 GiB、GQA (g=4) 15 GiB、MQA 3.75 GiB、MLA 8.44 GiB。

MLA 的显存占用约 9.1 GB：比 GQA (g=4) 的 16.1 GB 和 GQA (g=8) 的 32.2 GB 更小，但比只有一个 KV head 的 MQA 更大。因为每个 GQA KV group 每 token 需要 $2d = 256$ 个元素，所以 MLA 的 576 个元素相当于：

$$\frac{d_c + d_r}{2d} = \frac{576}{256} = 2.25 \quad (373)$$

个 GQA KV group 的缓存量。与直接把 KV head 减到约 2.25 组不同，MLA 仍可让所有 Query head 通过各自的上投影从同一个潜在表示中读取不同的内容 K/V。

因此，MLA 实际上是在 GQA 的缓存效率和 MHA 的表达能力之间找到了一个新的平衡点：通过低秩压缩而非头共享，在比 MHA 小得多的缓存中保留接近 MHA 的多头多样性。

例 16.6.1: 对比 GQA 和 MLA 的表示方式：

- GQA (g=8): 8 组独立 KV，每组被 16 个 Query head 共享；同组 Query head 读取的是同一组 K/V。
- MLA: 所有 head 共享缓存的 c^{KV} ，但具有各自的内容 K/V 上投影；它们可以从同一个压缩表示中读取不同表示。

所以 GQA 主要通过减少 KV 组数节省缓存，MLA 则通过联合压缩表示节省缓存。

16.7. 训练时如何运作？

常规全序列训练不会像自回归 Decode 那样长期维护 KV Cache。训练时通常按概念公式显式计算各项，并通过反向传播学习压缩与上投影参数：

1. $C^{KV} = XW^{DKV}$
2. 对每个 head: $K_i^C = C^{KV}W_i^{UK}$, $V_i^C = C^{KV}W_i^{UV}$
3. $C^Q = XW^{DQ}$ ，再上投影得到各 head 的内容 Query 和 RoPE Query
4. 标准 attention 计算

训练时需要反向传播更新 W^{DKV} 、 W_i^{UK} 、 W_i^{UV} 、 W^{DQ} 等参数。KV 联合压缩让模型学会如何在有限维度中编码足够的 K/V 信息；Query 压缩还可以减少训练时的 activation memory。

一个直观理解是：MLA 训练就像教会模型“用速记符号做笔记”。 c^{KV} 是速记符号， W_i^{UK} 和 W_i^{UV} 是不同读者解释速记的方式。推理时缓存速记和一小段位置索引，而不是完整笔记——每个读者仍能读取自己关心的部分。

16.8. 总结

MLA 的核心贡献可以归结为三点：

1. **低秩压缩替代头共享**：不同于 MQA/GQA 通过减少 KV 头数量来降低缓存，MLA 用低维潜在向量压缩所有头的 KV 信息，保留了每个头的独立解压参数，表达能力更强。
2. **矩阵吸收优化推理**：Key 的解压矩阵可以与 Query 投影矩阵提前合并，推理时不显式展开完整 Key。这是 MLA 推理效率远超“先解压再计算”朴素方案的原因。
3. **Decoupled RoPE**：将位置编码从压缩表示中解耦出去，用“内容 Key（无 RoPE）+ 位置 Key（有 RoPE）”的结构同时解决低秩压缩和旋转位置编码的兼容性问题。

MLA 本质上是一种**表示层面的压缩**——它不改变 attention 的数学形式（仍然是 softmax attention），不改变 head 的数量，也不改变训练目标。它主要改变 KV Cache 中存储的内容：从展开后的完整 K/V，变成每层、每 token 的 (c_t^{KV}, k_t^R) 。这使它成为 DeepSeek-V2/V3 支持长上下文和高吞吐推理的关键技术，也代表了高效 attention 设计的一种重要方向：从“减少 KV head 数量”走向“压缩 KV 表示”。

17. 从 vLLM 的角度分析

17.1. 为什么 KV 可以 Cache?

KV cache 能成立，关键原因是：在 decoder-only 自回归模型中，历史 token 的表示不会依赖未来 token。

以第 i 个历史 token 为例。因为 causal mask 存在，它在 self-attention 中只能看：

$$x_1, x_2, \dots, x_i \quad (374)$$

不能看：

$$x_{i+1}, x_{i+2}, \dots \quad (375)$$

因此，当未来新生成了 x_{n+1} 时，历史位置 x_i 在每一层里能够看到的内容没有变化。它以前不能看 x_{n+1} ，现在作为历史 token 重新计算时仍然不能看 x_{n+1} 。所以它对应的中间表示、Key、Value 都不需要改变。

这就是 KV cache 的数学依据。

换句话说，对于历史 token：

$$K_i = h_i W^K \quad (376)$$

$$V_i = h_i W^V \quad (377)$$

其中 h_i 是该层输入到 attention 前的历史 token 表示。由于 h_i 不会因为未来 token 出现而改变，所以 K_i 和 V_i 也不会改变。既然不会改变，就可以存下来，下次直接读。

需要注意：cache 的是每一层的 K 和 V，不只是第一层的 K 和 V。因为 Transformer 有多层，第 1 层的新 token 算完以后会进入第 2 层；第 2 层也需要用它自己的历史 K/V；一直到最后一层都是如此。

如果模型有 L 层，那么 KV cache 大致是：

$$\{(K_1, V_1), (K_2, V_2), \dots, (K_L, V_L)\} \quad (378)$$

每一层都保存当前请求所有历史 token 的 K/V。

17.2. 为什么不 Cache Q?

Attention 的计算可以写成：

$$\text{output}_i = \text{softmax}\left(\frac{q_i K^T}{\sqrt{d_k}}\right) V \quad (379)$$

生成下一个 token 时，我们只关心最后一个位置的输出，因为只有最后一个位置会被拿去预测下一个 token。也就是说，Decode 阶段只需要当前新 token 的 q_{new} 。

历史 token 的 Query 用来计算历史 token 自己的输出。但历史 token 的输出在以前已经算过了，而且它们不会因为未来 token 到来而改变，所以历史 Query 没有必要保存。

Key 和 Value 不一样。新 token 要关注历史 token，必须拿自己的 Query 去匹配历史 Key，并从历史 Value 里取信息。因此历史 K/V 是下一轮还会被用到的东西，Q 通常不是。

可以把它理解成一个检索系统：

1. 新 token 的 Query 是这次新提出的问题

2. 历史 token 的 Key 是可以被检索的索引
3. 历史 token 的 Value 是检索后真正取回的内容

下一轮来了一个新问题，就还需要旧索引和旧内容，所以保存 K/V；旧问题本身不再需要，所以不保存 Q。

2026-09-01: 这里非常感性的分析，我无法理解，我认为本质是矩阵运算的过程中，Q K V 中，由于 Q 是第一个矩阵，而且计算仅仅仅仅考虑前面的 token 对于后面的 token 的影响。

17.2.1. QK^T 会不会占用 $O(T^2)$ 的 KV Cache?

Attention 的确需要计算 QK^T ，但它通常只是当前 attention 算子内部的临时结果，不会跨生成步骤保存在 KV Cache 中。需要区分两件事：

1. 计算 attention 时暂时产生多少数据
2. 完成当前步骤后，需要为后续 token 长期保留多少数据

前者可能具有二次规模，后者仍然随上下文长度线性增长。

17.2.2. Prefill 阶段的临时 Attention 矩阵

设输入长度为 T ，单个 head 的维度为 d_{head} ：

$$Q \in \mathbb{R}^{T \times d_{\text{head}}}, \quad K \in \mathbb{R}^{T \times d_{\text{head}}} \quad (380)$$

因此 attention 分数矩阵的形状为：

$$QK^T \in \mathbb{R}^{T \times T} \quad (381)$$

如果普通 attention 实现完整物化这个矩阵，设每个元素占 s_{elem} 字节，那么单层、单个 head 仅分数矩阵就需要：

$$T^2 s_{\text{elem}} \text{ bytes} \quad (382)$$

对于 batch size 为 B 、Query head 数为 H_q 的当前层，对应的量级是：

$$BH_q T^2 s_{\text{elem}} \quad (383)$$

如果把 L 层的分数矩阵全部同时保留，理论上还会多一个 L ，但推理时各层依次执行，而且下一层不需要上一层的 attention 分数，所以没有理由长期保存这些矩阵。完成：

$$A = \text{softmax} \left(\frac{QK^T}{\sqrt{d_{\text{head}}}} \right) \quad (384)$$

$$O = AV \quad (385)$$

之后， QK^T 和 A 都可以释放。

FlashAttention 更进一步：它把 QK^T 分块计算，在片上存储中维护 online softmax 的统计量，并立即把当前分块对 AV 的贡献累加到输出。这样就不需要在 HBM 中物化完整的 $T \times T$ 分数矩阵和权重矩阵。对于固定的 batch、head 数和 head 维度，其额外存储随序列长度通常近似线性增长，而不是二次增长。

但这只是改变计算顺序和内存访问方式。标准全注意力仍然要处理所有允许的 Query-Key 配对，所以 Prefill 的理论计算复杂度仍然是 $O(T^2)$ ；FlashAttention 没有把它变成 $O(T)$ 。

17.2.3. Decode 阶段只计算新的一行

生成第 $t + 1$ 个 token 时，当前层只需要计算新 token 的 Query：

$$q_{\text{new}} \in \mathbb{R}^{1 \times d_{\text{head}}} \quad (386)$$

历史 Key 已经存放在 KV Cache 中：

$$K_{\text{cache}} \in \mathbb{R}^{t \times d_{\text{head}}} \quad (387)$$

二者相乘得到：

$$q_{\text{new}} K_{\text{cache}}^T \in \mathbb{R}^{1 \times t} \quad (388)$$

也就是说，每个 Query head 只产生长度为 t 的一行新分数，不会重新生成完整的 $t \times t$ 矩阵。当前 token 的输出为：

$$a_{\text{new}} = \text{softmax} \left(\frac{q_{\text{new}} K_{\text{cache}}^T}{\sqrt{d_{\text{head}}}} \right) \quad (389)$$

$$o_{\text{new}} = a_{\text{new}} V_{\text{cache}} \quad (390)$$

计算结束后，这一行分数和 softmax 权重也可以立即释放。某些融合 kernel 甚至不会把完整的长度 t 向量写回 HBM，而是在分块读取 KV Cache 时直接完成 softmax 和 Value 聚合。

17.2.4. 为什么 QK^T 也不值得缓存？

因为每轮生成时，新的 Query 都不同。第 $t+1$ 个 token 需要的是：

$$q_{t+1} K_{1:t}^T \quad (391)$$

下一轮需要的是：

$$q_{t+2} K_{1:t+1}^T \quad (392)$$

历史 Key 可以复用，但新的 Query 改变了，所以必须为新 token 计算一行新的匹配分数。以前生成的分数行只服务于以前的 Query；对应位置的输出早已计算完成，后续生成也不会再次读取它们。缓存这些旧分数既没有复用价值，还会把长期存储从线性规模推高到二次规模。

可以把三者的生命周期总结为：

1. K, V ：历史内容的“资料库”，后续每一步都会读取，所以缓存
2. Q ：当前 token 提出的问题，每一步都会变化，所以不缓存
3. QK^T ：当前问题与历史资料的匹配分数，只在当前 attention 计算中使用，所以不长期缓存

17.2.5. 存储复杂度和计算复杂度

数据	Prefill	Decode 单步	长期缓存
KV Cache	$O(T)$	随 token 线性增长	是
完整 attention 矩阵	$O(T^2)$	不需要	否
Decode attention 分数	—	$O(T)$	否
FlashAttention 临时空间	通常近似 $O(T)$	通常为 $O(T)$	否

这里的复杂度省略了层数、batch、head 数和 head 维度等固定因子。更完整地说，标准 KV Cache 的大小与下面的量同阶：

$$LBH_kTd_{\text{head}} \quad (393)$$

其中 H_k 是 KV head 数量；MHA 中通常 $H_k = H_q$ ，GQA/MQA 则通过减少 H_k 降低缓存大小。无论使用哪种方式，它对序列长度 T 都是线性的。

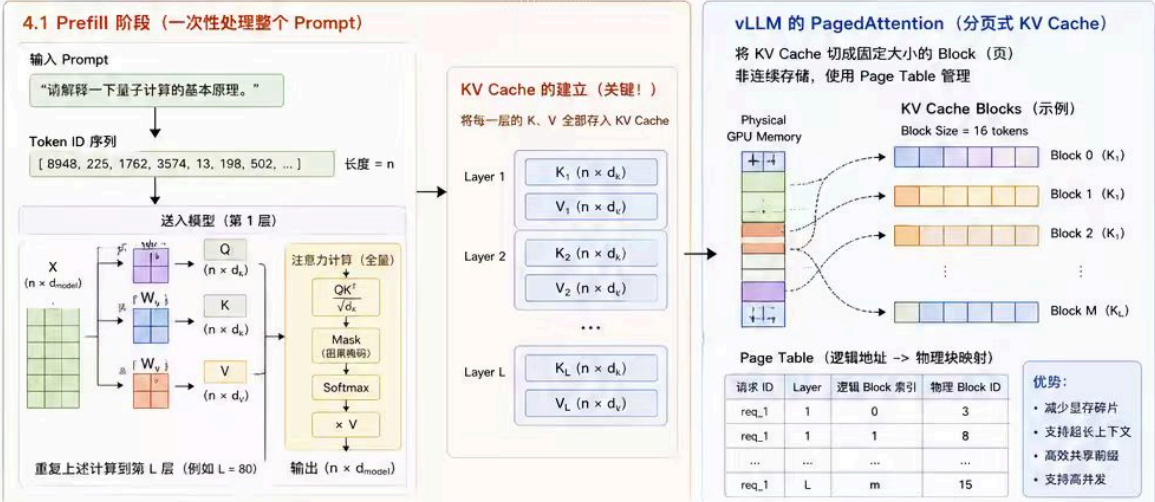
结论 17.2.5.1: 长期 KV Cache 的空间复杂度对上下文长度是 $O(T)$ ，不是 $O(T^2)$ ；但标准全注意力在 Prefill 阶段的计算复杂度仍然是 $O(T^2)$ 。

17.3. 推理为什么要分成 Prefill 和 Decode?

大模型推理全流程 (vLLM 实现) —— 从输入到输出的每一个技术细节



4. 模型推理核心流程 (vLLM 执行时序)



4.2 Decode 阶段 (自回归生成, 每次只算一个新 Token)

Decode Step t (生成第 t 个 Token)

当前已生成的序列 (长度 = n + t - 1)

输入当前最后一个 token: x_{t-1}

送入模型 (每层只处理 1 个新 token)

计算过程:

$$x_{t-1} (1 \times d_{\text{model}}) \xrightarrow{W_Q} Q_{\text{new}} (1 \times d_k) \xrightarrow{\text{注意力计算 (增量)}} Q_{\text{new}} \times K_{\text{cache}}^T \xrightarrow{\text{Softmax}} \times V_{\text{cache}} \xrightarrow{\text{输出}} (1 \times d_{\text{model}})$$

注意力计算 (增量): $Q_{\text{new}} \times K_{\text{cache}}^T$ (与所有历史 token 做注意力)

从 KV Cache 读取 (PagedAttention)

对于每一层:

- 根据 Page Table 找到该请求的所有历史 K、V 块
- 将这些块的 K、V 读出 (无需连续)

$K_{\text{cache}} (n+t-1 \times d_k)$

更新 KV Cache

将 K_{new} 、 V_{new} 写入 Cache (可能分配新 Block)

生成下一个 Token

- 经过最后一层输出
- LM Head 投影到词表
- 得到 logits (vocab size)
- 得到下一个 token: t_t

logits 分布 (示例)

采样策略 (可选):

- Greedy / Top-k / Top-p
- Temperature
- Repetition Penalty 等

生成下一个 token: t_t

循环 (直到遇到 <eos> 或达到最大长度)

t = 1 -> Step 1 -> 输出 t_1

t = 2 -> Step 2 -> 输出 t_2

...

t = T -> Step T -> 输出 t_T -> 结束

Streaming 输出 (实时返回)

通过 WebSocket / SSE 流式返回给用户

"量子计算利用量子比特..."

4.3 Prefill 与 Decode 的区别

对比项	Prefill	Decode
输入长度	n (全部)	1 (增量)
计算量	$O(n^2)$	$O(n)$
是否使用 KV Cache	否 (生成)	是 (读取+更新)
主要瓶颈	计算 (算力)	内存带宽 (显存)
并行度	高	较低 (但可批)
延迟影响	大 (长 prompt 慢)	小 (生成阶段)

KV Cache 内存占用估算

每层每 token 的 KV 大小:

$$2 \times d_{\text{model}} \times \text{bytes_per_elem}$$

示例 (FP16):

$d_{\text{model}} = 8192$

$\text{bytes_per_elem} = 2$

每层每 token $\approx 32 \text{ KB}$

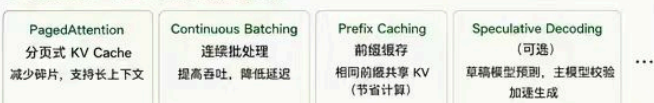
总占用 $\approx 32 \text{ KB} \times n_{\text{tokens}} \times n_{\text{layers}}$

例如: $32 \text{ KB} \times 100k \times 80 \approx 256 \text{ GB}$

5. vLLM 调度与并发 (让 GPU 吃满的关键)



6. 关键技术一览 (vLLM 核心能力)



7. 整体性能影响因素

- Prompt 长度 (Prefill 时间)
- 生成长度 (Decode 时间)
- Batch Size (并发请求数)
- KV Cache 命中率 (前缀共享)
- GPU 显存带宽 (Decode 瓶颈)
- 采样策略 (复杂度)
- 模型大小 (层数、隐藏维度)

8. 总结 (一句话)

Prefill 负责 "理解全文", Decode 借助 KV Cache "增量生成", vLLM 通过 PagedAttention 和智能调度让这一切又快又省又稳!

Figure 3: vLLM 推理流程中的 Prefill、Decode 和 KV Cache

在 vLLM、TensorRT-LLM、llama.cpp 这类推理系统里，经常会看到两个阶段：

1. **Prefill**：一次性处理用户输入的 prompt
2. **Decode**：之后每次生成一个新 token

这不是模型结构里多出来的两个层，而是推理执行方式的划分。它来自自回归语言模型的任务形式：

$$P(x_{n+1} \mid x_1, x_2, \dots, x_n) \quad (394)$$

也就是说，模型先看到 prompt 中已有的 n 个 token，然后预测第 $n+1$ 个 token。生成出第 $n+1$ 个 token 后，再把它接到序列末尾，继续预测第 $n+2$ 个 token：

$$P(x_{n+2} \mid x_1, x_2, \dots, x_n, x_{n+1}) \quad (395)$$

所以推理天然分成两类工作：

1. 第一次请求进来时，prompt 已经完整存在，可以把 $x_1 \dots x_n$ 一次性送入模型
2. 后续生成时，未来 token 还不存在，只能每生成一个 token，再做下一步

第一类工作就是 Prefill，第二类工作就是 Decode。

17.4. Prefill 阶段到底在做什么？

假设 prompt 长度是 n ，输入矩阵为：

$$X_{1:n} \in \mathbb{R}^{n \times d_{\text{model}}} \quad (396)$$

在每一层 self-attention 中，模型会一次性算出 prompt 中所有 token 的：

$$Q_{1:n} = X_{1:n} W^Q \quad (397)$$

$$K_{1:n} = X_{1:n} W^K \quad (398)$$

$$V_{1:n} = X_{1:n} W^V \quad (399)$$

然后做 masked self-attention：

$$O_{1:n} = \text{softmax} \left(\frac{Q_{1:n} K_{1:n}^T + M}{\sqrt{d_k}} \right) V_{1:n} \quad (400)$$

其中 M 是 causal mask，用来阻止第 i 个位置看见第 $i+1$ 之后的未来 token。

Prefill 的特点是：输入是一整段 prompt，所以这一段里的所有 token 可以用矩阵乘法一起算。虽然有 causal mask，但它只是把 attention 矩阵上三角的未来位置屏蔽掉，不妨碍 GPU 并行计算整块矩阵。

因此 Prefill 通常并行度高，但如果 prompt 很长，attention 矩阵大小是 $n \times n$ ，计算量会比较大。

17.5. Decode 阶段到底在做什么？

Prefill 结束后，模型已经根据 prompt 预测出了下一个 token。假设新生成的 token 是 x_{n+1} 。下一步要预测 x_{n+2} 时，输入上下文变成：

$$x_1, x_2, \dots, x_n, x_{n+1} \quad (401)$$

这时没有必要把前面 n 个 prompt token 全部重新算一遍。因为对已经存在的历史 token 来说，它们在当前层的 Key 和 Value 已经算过了。

Decode 时，每一层只需要对新 token 计算：

$$q_{\text{new}} = x_{\text{new}} W^Q \quad (402)$$

$$k_{\text{new}} = x_{\text{new}} W^K \quad (403)$$

$$v_{\text{new}} = x_{\text{new}} W^V \quad (404)$$

然后让新 token 的 Query 去和所有历史 Key 做 attention：

$$o_{\text{new}} = \text{softmax} \left(\frac{q_{\text{new}} K_{\text{cache}}^T}{\sqrt{d_k}} \right) V_{\text{cache}} \quad (405)$$

这里的 K_{cache} 和 V_{cache} 包含了 prompt token 以及已经生成 token 的 Key、Value。算完以后，再把这次新 token 的 k_{new} 和 v_{new} 追加进 cache，供下一轮使用。

所以 Decode 的输入长度看起来是 1，因为这一轮真正新进模型的只有一个 token；但它 attention 的对象不是只有自己，而是整个历史上下文。

17.6. Prefill 和 Decode 的计算量差异

如果 prompt 长度是 n ，Prefill 要处理 n 个 token 之间的 attention，核心矩阵是：

$$Q_{1:n} K_{1:n}^T \in \mathbb{R}^{n \times n} \quad (406)$$

所以 attention 部分大致是 $O(n^2)$ 。

Decode 每一步只处理一个新 token。新 token 的 Query 需要和长度为 $n + t$ 的历史 Key 做匹配：

$$q_{\text{new}} K_{\text{cache}}^T \in \mathbb{R}^{1 \times (n+t)} \quad (407)$$

所以单步 attention 大致是 $O(n + t)$ 。随着生成变长，每一步会越来越贵，但比每一步都重新算整个 $O((n + t)^2)$ 便宜很多。

如果没有 KV cache，生成第 $n + 1$ 、 $n + 2$ 、 $n + 3$ 个 token 时，每一步都要把整个上下文从头跑一遍。大量历史 K/V 会被重复计算，浪费非常大。

有 KV cache 后：

1. Prefill 负责把 prompt 的所有层 K/V 建好
2. Decode 每轮只算新 token 的 K/V
3. 新 token 通过自己的 Query 读取所有历史 K/V
4. 新 token 的 K/V 追加到 cache，供下一轮继续用

这就是 vLLM 文档里说的“Prefill 负责建立 KV Cache，Decode 读取并更新 KV Cache”。

17.7. vLLM 的 PagedAttention 解决什么？

KV cache 的数学逻辑来自 causal self-attention；PagedAttention 解决的是工程上的显存管理问题。

先看 KV cache 为什么会变成推理系统的核心瓶颈。一个请求不是只存一份 token 序列，而是要为每一层、每个 KV head、每个历史 token 保存 Key 和 Value。粗略写成：

$$\text{KV cache size} \approx 2 \times L \times n \times h_{\text{kv}} \times d_{\text{head}} \times \text{bytes} \quad (408)$$

其中：

1. 2 表示 Key 和 Value 两份
2. L 是层数
3. n 是当前上下文长度
4. h_{kv} 是 KV head 数量
5. d_{head} 是每个 head 的维度

所以 KV cache 会随着上下文长度线性增长，而且每个并发请求都要占一份。模型权重通常是固定大小，但 KV cache 会随请求数量、prompt 长度和生成长度动态变化。在线服务里，显存经常不是先被权重耗尽，而是被大量请求的 KV cache 挤满。

问题在于：请求长度不是固定的。不同用户的 prompt 长度不同，生成长度也无法提前准确知道。有的请求只生成几十个 token，有的请求可能生成几千个 token。如果系统要求每个请求的 KV cache 在显存中连续存放，就会遇到类似传统内存分配的问题。

假设一个请求当前有 1000 个 token，但系统为了它未来可能生成到 2048 个 token，提前预留一整段连续空间。若这个请求最后只生成到 1200 个 token，那么后面 848 个 token 的空间就浪费了。反过来，如果一开始只分配 1000 个 token 的连续空间，后面继续生成时可能发现相邻显存已经被别的请求占了，无法原地扩容，只能搬迁或重新分配。

因此，连续 KV cache 管理至少有三个麻烦：

1. **内部浪费**：为了容纳最大生成长度而预留空间，但很多请求实际用不到
2. **外部碎片**：显存里有很多空洞，总空闲显存够，但找不到足够大的连续区域
3. **扩容困难**：decode 每生成一个 token 都要追加 K/V，连续区域不够时很难低成本增长

vLLM 的做法类似操作系统分页：把 KV cache 切成固定大小的 block。每个请求维护一张类似 page table 的映射表，记录“这个请求的第几个 token block 存在显存的哪个物理 block 里”。

也就是说，在逻辑上，一个请求的历史 token 仍然是连续序列：

token block 0, token block 1, token block 2, ...

但在物理显存里，这些 block 可以分散存放：

```
request A logical block 0 -> physical block 17
request A logical block 1 -> physical block 3
request A logical block 2 -> physical block 42
```

PagedAttention 的 attention kernel 在计算时会根据这张 block table 找到对应的 K/V block。对模型数学来说，它仍然是在对完整历史上下文做：

$$o_{\text{new}} = \text{softmax} \left(\frac{q_{\text{new}} K_{\text{cache}}^T}{\sqrt{d_k}} \right) V_{\text{cache}} \quad (409)$$

区别只是 K_{cache} 和 V_{cache} 不再要求物理连续。逻辑连续性由 block table 维护，物理存储由显存 block 池管理。

这样做的好处是：

1. KV cache 不需要连续存储，减少显存碎片
2. 请求增长时可以按 block 追加分配，不需要提前为最大长度预留整段空间
3. 请求结束后释放的是若干独立 block，能立刻回到 block 池给其他请求使用

4. 多个请求可以更容易共享相同前缀的 KV block
5. 调度器可以在 Prefill 和 Decode 请求之间更灵活地安排显存

这里有一个关键点：PagedAttention 不是让 attention 的数学复杂度从 $O(n)$ 变成 $O(1)$ 。Decode 阶段的新 token 仍然要看历史 K/V，单步 attention 仍然大致随历史长度增长。PagedAttention 主要降低的是 KV cache 的显存浪费和动态分配压力，从而让同一张 GPU 能容纳更多并发序列。

可以把它和 FlashAttention 对比：

1. **FlashAttention** 主要优化一次 attention 计算内部的中间矩阵读写，尤其是 prefill 或训练中的大块 attention
2. **PagedAttention** 主要优化推理服务中长期存在的 KV cache 存储、追加、释放和共享

因此二者解决的问题不同，但可以同时存在。Prefill 时可能用 FlashAttention 高效计算 prompt attention；Decode 时则需要不断读取和追加 KV cache，PagedAttention 负责让这些 K/V 在显存中更好管理。

所以要区分两层概念：

1. **KV cache**：来自自回归 Transformer 的数学性质，避免重复计算历史 K/V
2. **PagedAttention**：vLLM 管理 KV cache 的工程方案，让显存分配和并发调度更高效

一句话总结：因为 causal mask 保证历史 token 不依赖未来 token，历史 K/V 算完后不会变，所以可以 cache；因为 prompt 已经完整而未来生成 token 不完整，所以推理自然分成一次性处理 prompt 的 Prefill 和逐 token 生成的 Decode。

17.8. KV cache 会不会不命中？

需要先区分两种 cache。

第一种是同一个请求内部的 KV cache。用户请求进来后，Prefill 会为 prompt 建好每一层的历史 K/V。之后 Decode 第 $n+1$ 、 $n+2$ 、 $n+3$ 个 token 时，新 token 的 Query 会读取这些历史 K/V：

$$o_{\text{new}} = \text{softmax} \left(\frac{q_{\text{new}} K_{\text{cache}}^T}{\sqrt{d_k}} \right) V_{\text{cache}} \quad (410)$$

这种情况下通常不把它理解成“命中/不命中”。因为对同一个请求来说，历史 K/V 本来就是当前上下文的一部分。Prefill 建完后，Decode 必须读它；如果没有这些 K/V，就只能重新从头计算历史上下文。

第二种是跨请求复用的 prefix cache，也可以叫 prompt cache。它才更像传统意义上的 cache hit / miss。

例如两个请求有完全相同的前缀：

请求 A：请总结这篇文章：...
请求 B：请总结这篇文章：...

如果推理框架开启了 prefix caching，那么请求 B 可以复用请求 A 已经算好的前缀 KV blocks。此时就可以说：

1. **prefix cache hit**：前缀 token 匹配，可以复用已有 K/V
2. **prefix cache miss**：没有可复用的前缀 K/V，需要重新 Prefill

常见的不命中原因包括：

1. 这是第一次出现这个 prompt 前缀，还没有可复用的 KV
2. 文本看起来相似，但 tokenizer 之后的 token 序列不完全相同
3. 只有部分前缀相同，只能复用相同 block，后面不同部分仍要重新算
4. cache 容量有限，旧的 KV block 已经被驱逐
5. 模型、LoRA、adapter 或推理配置不同，不能共享同一份 KV
6. 上下文太长，框架因为滑动窗口、截断或显存压力丢弃了部分历史 KV
7. 调度器因为显存压力发生 preemption，某些 KV 可能需要 swap 回来或重新计算

所以更准确的说法是：

1. 同一个请求内部，KV cache 是自回归解码的必要历史状态，Prefill 后 Decode 会持续读取和追加它
2. 不同请求之间，只有前缀 token 完全匹配且对应 KV blocks 仍然可用时，prefix cache 才会命中
3. PagedAttention 负责把 KV cache 切成 block 管理，方便分配、复用和调度；它本身不是判断语义相似 prompt 是否可复用的机制

一句话总结：普通 Decode 读当前请求的 KV cache，不太叫命中；跨请求复用相同前缀时，才会出现 prefix cache hit / miss。

18. Softmax

Softmax 的作用是把一组任意实数分数变成一个概率分布。给定：

$$z = [z_1, z_2, \dots, z_n] \quad (411)$$

softmax 的第 i 项是：

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \quad (412)$$

它满足

1. $\text{softmax}(z)_i > 0$
2. $\sum_i \text{softmax}(z)_i = 1$
3. z_i 越大，对应的 softmax 权重越大

所以 softmax 常用来把 logits 变成概率，或者在 attention 中把匹配分数变成注意力权重。

例如：

$$z = [2, 1, -1] \quad (413)$$

先计算：

$$\exp(2) \approx 7.389, \quad \exp(1) \approx 2.718, \quad \exp(-1) \approx 0.368 \quad (414)$$

归一化后：

$$\text{softmax}(z) \approx [0.705, 0.259, 0.035] \quad (415)$$

注意 softmax 不是只选最大值。它会给所有候选项分配权重，只是分数越大的项权重越大。

为什么要用指数？因为 $\exp(x)$ 永远大于 0、单调递增，并且会放大相对差距。例如两个 logit 相差 1，指数后的比例就是：

$$\frac{\exp(2)}{\exp(1)} = \exp(1) \approx 2.718 \quad (416)$$

也就是说，softmax 不是简单地把原始分数除以总和，而是先用指数把高分项凸显出来，再归一化。

Softmax 还有一个重要性质：它只关心相对差距，不关心整体平移。给所有 logits 同时加上常数 c ：

$$\text{softmax}(z + c)_i = \frac{\exp(z_i + c)}{\sum_j \exp(z_j + c)} \quad (417)$$

$$= \frac{\exp(z_i) \exp(c)}{\sum_j (\exp(z_j) \exp(c))} \quad (418)$$

$$= \frac{\exp(z_i)}{\sum_j \exp(z_j)} \quad (419)$$

$$= \text{softmax}(z)_i \quad (420)$$

所以实际实现中常常先减去最大值：

$$\text{softmax}(z)_i = \frac{\exp(z_i - m)}{\sum_j \exp(z_j - m)}, \quad m = \max_j z_j \quad (421)$$

这样数学结果不变，但可以避免 exp 溢出。

在 attention 中，先得到匹配分数：

$$S = \frac{QK^T}{\sqrt{d_k}} \quad (422)$$

其中 $S_{i,j}$ 表示第 i 个 token 的 Query 和第 j 个 token 的 Key 有多匹配。然后对每一行做 softmax：

$$A_{i,j} = \frac{\exp(S_{i,j})}{\sum_l \exp(S_{i,l})} \quad (423)$$

这样第 i 行就是一个概率分布：

$$\sum_j A_{i,j} = 1 \quad (424)$$

最后：

$$O = AV \quad (425)$$

所以在 attention 公式里：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (426)$$

softmax 负责把“相关性分数”变成“从每个 Value 取多少信息”的权重。

18.1. Softmax 为什么需要 safe 写法?

Softmax 的数学形式很简单，但直接实现：

$$\frac{\exp(z_i)}{\sum_j \exp(z_j)} \quad (427)$$

容易溢出。比如 FP16 的最大有限值大约是 65504，而 $\exp(20)$ 已经远远超过这个范围。attention 里的分数、输出层 logits 都可能出现较大的正数，所以工程实现通常使用 safe softmax：

$$m = \max_j z_j \quad (428)$$

$$\text{softmax}(z)_i = \frac{\exp(z_i - m)}{\sum_j \exp(z_j - m)} \quad (429)$$

这不会改变结果，因为 softmax 对整体平移不敏感。减去最大值后，所有 $z_i - m \leq 0$ ，指数函数不会上溢，最大项的指数也正好是 1。

这个技巧在 attention 里尤其重要。未缩放的点积分数方差会随 d_k 增大而扩大，所以 Transformer 公式里要除以 $\sqrt{d_k}$ 。safe softmax 解决的是指数上溢问题， $\sqrt{d_k}$ 缩放解决的是分数尺度过大导致分布过尖、梯度变小的问题。二者作用不同，但都是为了稳定 softmax。

18.2. Online Softmax

普通 safe softmax 仍然像是要先拿到整行 logits，算出最大值 m ，再算分母：

$$l = \sum_j \exp(z_j - m) \quad (430)$$

长序列 attention 中，一行 logits 可能很长。如果把每一行完整写到 HBM，再读回来做 softmax，访存会非常重。

Online softmax 的关键是：可以分块更新 m 和 l 。假设已经处理了前一部分，状态是：

$$m_a, \quad l_a = \sum_{j \in a} \exp(z_j - m_a) \quad (431)$$

新来一块，状态是：

$$m_b, \quad l_b = \sum_{j \in b} \exp(z_j - m_b) \quad (432)$$

合并后：

$$m = \max(m_a, m_b) \quad (433)$$

$$l = \exp(m_a - m)l_a + \exp(m_b - m)l_b \quad (434)$$

这和把两块拼起来一次性做 safe softmax 得到的分母完全一致。区别只是执行顺序变成了流式的。FlashAttention 正是利用这个性质，在处理 K, V block 时边算边更新输出，避免物化完整 attention 矩阵。

看一个具体的一行 logits：

$$z = [1, 2, 0, -1, 3, 1, 0, 2] \quad (435)$$

最终我们要得到的是这一行的 softmax:

$$\text{softmax}(z)_i = \frac{\exp(z_i - m)}{l} \quad (436)$$

其中:

$$m = \max_j z_j \quad (437)$$

$$l = \sum_j \exp(z_j - m) \quad (438)$$

对这个例子来说, 最后要得到的具体数值是:

$\exp(z - m)$ 为:

$$[0.135335, 0.367879, 0.049787, 0.018316, 1.000000, 0.135335, 0.049787, 0.367879] \quad (439)$$

softmax(z) 为:

$$[0.063708, 0.173175, 0.023437, 0.008622, 0.470739, 0.063708, 0.023437, 0.173175] \quad (440)$$

$$l \approx 2.124319 \quad (441)$$

这里需要注意一个容易误解的点: 只拿到整行最大值 m 和归一化分母 l , 还没有真正写出每个 softmax 元素。要显式得到:

$$\text{softmax}(z)_i = \frac{\exp(z_i - m)}{l} \quad (442)$$

仍然需要每个 z_i 。如果前面没有保存 z_i 或者 $\exp(z_i - m)$, 那么单独计算完整 softmax 向量时, 还需要再读一遍这一行。

所以, online softmax 不是说“读一遍输入, 同时立刻写完所有 softmax 输出”。更准确地说, 它是在线维护 softmax 需要的聚合状态, 例如 m 和 l 。如果任务只是对向量 z 输出完整的 $\text{softmax}(z)$, 通常仍然是:

1. 第一遍在线统计 m 和 l
2. 第二遍用最终的 m 和 l 计算每个 $\text{softmax}(z)_i$

它真正有价值的地方在 attention 里。Attention 最终通常不需要把完整概率矩阵 $P = \text{softmax}(S)$ 写出来, 而是需要:

$$o = \sum_j \text{softmax}(z)_j v_j \quad (443)$$

也可以等价地维护一个未归一化的加权和:

$$u = \sum_j \exp(z_j - m) v_j \quad (444)$$

最后再做:

$$o = \frac{u}{l} \quad (445)$$

这时 online softmax 可以和乘 V 融合: 扫描当前 block 时, 不仅更新 m 和 l , 还同步更新 u 。这样最终可以直接得到 attention 输出 o , 而不需要显式存储整行 $\text{softmax}(z)$ 。

所以 online softmax 的目标不是提前保存整行 $\text{softmax}(z)$ ，而是流式维护足够的信息： m 、 l ，以及 attention 输出需要的 u 。

如果一次性处理整行，先取最大值：

$$m = 3 \quad (446)$$

然后分母是：

$$l = \sum_j \exp(z_j - 3) \quad (447)$$

$$= \exp(-2) + \exp(-1) + \exp(-3) + \exp(-4) + 1 + \exp(-2) + \exp(-3) + \exp(-1) \quad (448)$$

$$\approx 2.124319 \quad (449)$$

Online softmax 的做法是：不要求一次性拿完整一行，而是把这一行切成两块：

$$z_a = [1, 2, 0, -1], \quad z_b = [3, 1, 0, 2] \quad (450)$$

先处理第一块：

$$m_a = 2 \quad (451)$$

$$l_a = \exp(1 - 2) + \exp(2 - 2) + \exp(0 - 2) + \exp(-1 - 2) \quad (452)$$

$$= \exp(-1) + 1 + \exp(-2) + \exp(-3) \quad (453)$$

$$\approx 0.367879 + 1.000000 + 0.135335 + 0.049787 \quad (454)$$

$$\approx 1.553002 \quad (455)$$

此时我们暂时只知道第一块的信息，所以保存状态：

$$(m_a, l_a) \approx (2, 1.553002) \quad (456)$$

接着处理第二块：

$$m_b = 3 \quad (457)$$

$$l_b = \exp(3 - 3) + \exp(1 - 3) + \exp(0 - 3) + \exp(2 - 3) \quad (458)$$

$$= 1 + \exp(-2) + \exp(-3) + \exp(-1) \quad (459)$$

$$\approx 1.000000 + 0.135335 + 0.049787 + 0.367879 \quad (460)$$

$$\approx 1.553002 \quad (461)$$

现在要把两块合并。合并后的最大值是：

$$m = \max(m_a, m_b) = 3 \quad (462)$$

关键点在这里：第一块的 l_a 是按旧最大值 $m_a = 2$ 算的：

$$l_a = \sum_{j \in a} \exp(z_j - m_a) \quad (463)$$

但合并后整行最大值变成了 $m = 3$ ，第一块对整行分母的贡献应该变成：

$$\sum_{j \in a} \exp(z_j - m) \quad (464)$$

这看起来像是要重新计算第一块，其实不用。因为：

$$\exp(z_j - m) = \exp(z_j - m_a) \exp(m_a - m) \quad (465)$$

其中 $\exp(m_a - m)$ 对第一块里的所有 j 都是同一个常数，所以：

$$\sum_{j \in a} \exp(z_j - m) = \exp(m_a - m) \sum_{j \in a} \exp(z_j - m_a) \quad (466)$$

$$= \exp(m_a - m) l_a \quad (467)$$

因此旧状态只需要乘一个缩放因子，不需要重新扫描第一块的每个元素。

完整合并公式是：

$$l = \exp(m_a - m) l_a + \exp(m_b - m) l_b \quad (468)$$

$$= \exp(2 - 3) \cdot 1.553002 + \exp(3 - 3) \cdot 1.553002 \quad (469)$$

$$\approx 0.367879 \cdot 1.553002 + 1.000000 \cdot 1.553002 \quad (470)$$

$$\approx 0.571317 + 1.553002 \quad (471)$$

$$\approx 2.124319 \quad (472)$$

这个结果和一次性处理整行得到的分母完全一样。所谓“重新缩放旧状态”，指的就是把旧的汇总值 l_a 乘上 $\exp(m_a - m)$ 。它不是重新计算第一块的 4 个指数，而是用一个标量修正第一块已经汇总好的结果。

如果同时维护 attention 输出的未归一化累积量 u ，它也用同样的方式合并：

$$u = \exp(m_a - m) u_a + \exp(m_b - m) u_b \quad (473)$$

最后：

$$o = \frac{u}{l} \quad (474)$$

这就是 FlashAttention 可以一边扫描 K, V block，一边更新 softmax 和输出的原因：旧 block 不需要长期保存，也不需要因为后面出现了更大的分数而重新计算，只要把旧的汇总状态按新的最大值缩放一下。

18.3. LSE 为什么有用？

LSE 是 log-sum-exp：

$$\text{LSE}(z) = \log \sum_j \exp(z_j) \quad (475)$$

结合 safe softmax，可以稳定地写成：

$$\text{LSE}(z) = m + \log \sum_j \exp(z_j - m), \quad m = \max_j z_j \quad (476)$$

于是：

$$\text{softmax}(z)_i = \exp(z_i - \text{LSE}(z)) \quad (477)$$

它的价值在于把一整行 softmax 的归一化信息压缩成一个标量。对于 FlashAttention：

1. 前向不需要保存完整 $P = \text{softmax}(S)$
2. 反向可以重算局部分数 S ，再用 $\exp(S - \text{LSE})$ 还原概率
3. split-k / flash-decoding 中，不同 KV 分片的局部结果可以通过 LSE 做稳定合并
4. 跨设备的 sequence/context parallel attention 也可以用同样的在线归一化逻辑合并局部输出

所以 LSE 不是单纯的数学记号，而是 attention kernel 减少显存占用和通信状态的关键中间量。

19. AWQ：激活感知的权重量化

AWQ 全称 **Activation-aware Weight Quantization**，即激活感知的权重量化。

19.1. 动机：为什么需要量化？

LLM 推理有两个基本瓶颈：

1. **显存**：70B 参数的模型，FP16 精度下权重就占 $70 \times 10^9 \times 2 \text{ bytes} = 140 \text{ GB}$ ，单卡根本放不下。
2. **显存带宽**：自回归解码每次只生成一个 token，算力需求不高，但要把所有权重从显存读一遍。以 70B 模型在 A100（带宽约 2 TB/s）上为例，纯权重读取就需要约 $140 \frac{\text{GB}}{2 \text{ TB/s}} = 70 \text{ ms}$ ，这是 decode 阶段延迟的硬地板。

量化的思路很直接：把权重从 16 位浮点数压缩到 4 位整数（INT4），显存和带宽需求同时降为原来的四分之一。

19.2. 量化基础

19.2.1. 最朴素的量化（RTN）

把浮点权重 w 映射到整数值 w_q 的公式是：

$$w_q = \text{round}\left(\frac{w}{\Delta}\right) \quad (478)$$

$$\Delta = \frac{\max(|w|)}{2^{b-1} - 1} \quad (479)$$

其中 b 是目标位数（比如 4 位）， Δ 是量化步长。推理时把 INT4 权重反量化回 FP16，再做矩阵乘法：

$$y = \text{dequant}(W_q)x = (\Delta \cdot W_q)x \quad (480)$$

这样计算结果是近似的：量化引进了误差 $Wx - \Delta W_q x$ 。

问题来了：直接对 LLM 做 RTN 量化到 INT4 会导致明显精度损失。有没有更好的办法？

19.2.2. GPTQ 的思路

GPTQ 的解法是对权重矩阵逐列做量化，量化完一列后，用剩余列的权重补偿这一列的量化误差。它的核心公式基于 OBS（Optimal Brain Surgeon）的逐列误差补偿：

量化第 q 列时，对剩余权重做更新：

$$\delta_F = -\frac{w_q - \text{quant}(w_q)}{[H^{-1}]_{q,q}} H_{:,q}^{-1} \quad (481)$$

其中 H 是 Hessian 矩阵（近似为 $2X^T X$ ， X 是校准数据的激活值）， H^{-1} 是它的逆。更新后的剩余权重能吸收这一列的量化误差，使得整体输出误差最小化。

GPTQ 的问题：它的校准数据用了随机采样，没有考虑不同输入下激活值分布的巨大差异。有些权重通道对应的激活值远大于其他通道，量化这些通道的误差对最终输出的影响也远大于普通通道。GPTQ 不加区分地逐列优化，对 outlier 通道的保护不够。

19.3. AWQ 的核心观察

先看一个问题：对所有权重通道一视同仁地量化是否合理？

取一个 LLM 的线性层权重矩阵 $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ ，用校准数据跑一遍前向，记录每一列（通道）激活值的平均幅度。按激活幅度从大到小排列权重通道，然后只量化 1% 最重要的通道，看看输出误差如何变化：

量化 1% 的 outlier 通道（激活幅度最大的 1%）：
→ 模型输出剧烈退化

量化随机 1% 的通道：
→ 模型输出几乎不变

量化 99% 的普通通道、保留 1% outlier 通道不量化：
→ 模型输出仍然很好

Listing 12: AWQ 的关键实验结论：outlier 通道的量化对模型质量影响远大于普通通道

这个现象的原因在于矩阵乘法的结构。对线性层 $y = Wx$ ：

$$y_i = \sum_j W_{i,j} x_j \quad (482)$$

$W_{i,j}$ 的量化误差 $\varepsilon_{i,j}$ 对 y_i 的贡献是 $\varepsilon_{i,j} x_j$ 。如果 $|x_j|$ 很大，同一个 $\varepsilon_{i,j}$ 对输出造成的绝对误差就等比放大。换句话说：**激活值大的输入通道，对应的权重列对量化误差更敏感。**

AWQ 把这种激活值大的通道称为 **salient channels**（显著通道）。

19.4. AWQ 的方法：通道级缩放

AWQ 的想法非常简洁：既然 outlier 通道的权重量化误差伤害大，就在量化前放大这些通道的权重（缩小对应的激活值），让量化误差的绝对值变小。

具体做法：对权重 W 的每一行引入一个缩放因子向量 $s \in \mathbb{R}^{d_{\text{in}}}$ ：

$$W' = W \text{diag}(s) \quad (483)$$

等价地，把激活值对应缩小：

$$x' = \text{diag}(s)^{-1} x \quad (484)$$

矩阵乘法的结果保持不变：

$$W' x' = W \text{diag}(s) \cdot \text{diag}(s)^{-1} x = W x \quad (485)$$

但此时 W' 中 outlier 通道的权值被放大了，相对量化误差变小； x' 中 outlier 通道的激活值被缩小了，量化误差对输出的影响也变小了。

关键在于 s 的选择。AWQ 用一个简单的搜索策略：

$$s = s_X^\alpha, \quad \alpha \in [0, 1] \quad (486)$$

其中 s_X 是校准数据中激活值的平均幅度（按通道）， α 是超参数。搜索过程是：

1. 取一小批校准数据
2. 对候选的 α 值（在 $[0, 1]$ 中等距采样），分别做通道缩放和 RTN 量化
3. 选出让输出误差最小的 α

定义 19.4.1: AWQ 的通道缩放量化

{ 给定权重矩阵 W 和校准激活值 X ：

1. 计算每通道激活平均幅度： $s_X = \text{mean}(|X|)$ ，形状 d_{in}
2. 搜索最优 α ，令 $s = s_X^\alpha$
3. 缩放权重： $W' = W \text{diag}(s)$
4. 对 W' 做标准 INT4/INT3 RTN 量化： $W'_q = \text{quant}(W')$
5. 推理时激活先缩放： $x' = \text{diag}(s)^{-1}x$ ，再计算 $y = \text{dequant}(W'_q)x'$

最终结果 y 近似等于 Wx ，但量化误差主要分布在普通通道上，outlier 通道的误差被显著抑制。}

把缩放因子吸收到 LayerNorm 或前一层的权重中，可以完全消除额外的运行时开销。推理时的计算量不变，只在数据预处理阶段多了一步等价变换。

19.5. AWQ 和 GPTQ 的对比

GPTQ：

基于 Hessian 的逐列误差补偿
量化每一列后，用剩余列吸收误差
校准数据随机采样，不分通道重要性
需要 $O(d_{\text{in}} * d_{\text{in}})$ 的内存和较多计算

AWQ：

基于激活幅度的通道重要性判断
量化前缩放重要通道，量化后反缩放激活
校准数据用于确定每通道的缩放因子
只需 $O(d_{\text{in}})$ 的额外内存和极少计算

Listing 13: AWQ 和 GPTQ 的方法对比

AWQ 的一个实际优势是实现极其简单。整个方法可以概括为三步：

1. 跑一小批校准数据，统计每通道激活幅度
2. 网格搜索 α ，找到量化误差最小的值
3. 缩放权重 + RTN 量化

不要求 Hessian 逆，不需要逐列补偿，不需要昂贵的矩阵运算。这意味着 AWQ 可以在几分钟内完成一个 70B 模型的量化，而 GPTQ 可能需要数小时。

19.6. 为什么 AWQ 有效？

从数值角度看，RTN 量化的误差大致正比于：

$$\text{error} \propto \Delta \cdot |x_j| \quad (487)$$

其中 Δ 是量化步长（由该通道权重的最大值决定）， $|x_j|$ 是该通道的平均激活幅度。

如果用缩放因子 s_j 处理第 j 个通道：

- 权重变为 $s_j W_{:,j}$ ，量化步长也大致乘以 s_j ： $\Delta' \approx s_j \Delta$
- 激活变为 $\frac{x_j}{s_j}$

量化误差变为：

$$\text{error}' \propto s_j \Delta \cdot \frac{|x_j|}{s_j} = \Delta \cdot |x_j| = \text{error} \quad (488)$$

看起来误差不变？关键在于 RTN 的舍入误差不严格正比于步长——步长内部的舍入误差有上限，不是随步长线性变化的。放大权重后，相同的绝对舍入误差占权重的比例更小，相对精度更高。激活缩小后又补偿了步长增大带来的误差放大效应。最终 net 效果是：**重要通道的权重量化更精确，普通通道的精度略降，总体的输出误差更小。**

从更工程的角度：AWQ 做的是把误差从敏感通道“搬”到不敏感通道。总误差没减少（甚至可能略微增加），但误差分布被重新调整，让模型质量损失降到最低。

19.7. 小结

AWQ 是 LLM 权重量化领域的一个重要方法，它带来了一个看似简单但很有效的视角：**量化之前先想清楚哪些权重要紧，哪些无所谓。**

1. 不是所有权重通道都同等重要。激活值大的输入通道对应的权重列对量化误差更敏感
2. 通道级缩放是一种数学等价的预处理： $Wx = (W \text{diag}(s))(\text{diag}(s)^{-1}x)$
3. 通过简单网格搜索找到合适的缩放强度，就能获得显著优于朴素 RTN 的量化质量
4. 相比 GPTQ，AWQ 不需要 Hessian 逆、不需要逐列误差补偿，实现简单、量化速度快
5. 缩放因子可以融合到相邻层中，推理时零额外开销

AWQ 和本文前面讲过的许多 Infra 优化共享同一种设计哲学：**用数学等价变换，把误差或开销从关键路径上搬走。**FlashAttention 是把中间矩阵的读写搬出 HBM，MLA 是把 KV cache 从显存搬进低秩潜在空间，AWQ 是把量化误差从重要通道搬到不重要通道。

20. 常见误区

- **SVD/PCA 那类线性分解和 Transformer attention 不是同一件事。**Attention 是可学习的动态加权机制，权重依赖当前输入。

21. 总结

Transformer 可以从一个核心公式理解：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (489)$$

这个公式做了三件事：

1. 用 QK^T 计算 token 之间的相关性
2. 用 softmax 把相关性变成权重
3. 用权重对 V 加权求和，得到上下文表示

在此基础上，Multi-Head Attention 提供多个观察视角，位置编码补充顺序信息，Encoder 和 Decoder 组织成完整的序列到序列模型。

Transformer 的真正突破在于：它把序列建模中的信息传递，从“按时间一步步传递”改成了“任意位置直接交互”。这带来了更好的并行性和更强的长距离依赖建模能力。

21.1. Infra 优化的共同模式

RMSNorm、safe softmax、online softmax、Gumbel-Max、FlashAttention、AWQ 看起来是不同主题，但底层逻辑很像：

1. 用数学等价变换保持结果不变，例如 softmax 减最大值、online softmax、LSE 重算、AWQ 的通道缩放
2. 用架构简化换取更低实现成本，例如 RMSNorm 去掉均值，很多 Linear 去掉 bias
3. 用重计算减少显存读写，例如 FlashAttention 反向重算 softmax 概率
4. 把串行或不规则流程改写成逐元素计算加规约，例如 Gumbel-Max 把采样改成加噪后 argmax
5. 把误差或开销从关键路径上搬走，例如 AWQ 把量化误差从敏感通道搬到不重要通道

大模型 Infra 优化经常不是发明一个全新的模型公式，而是在不改变或尽量少改变模型行为的前提下，把计算改写成更适合 GPU 的形状：更少 HBM 往返、更高片上复用、更少同步点、更容易融合成单个 kernel。