Large Language Models: The Complete Guide

卢奇

luqi.code@gmail.com

摘要

本文系统性地梳理大语言模型 (LLM) 领域的核心技术，涵盖从基础理论到前沿应用的完整技术栈。

全文分为八个部分：**基础理论**介绍 Transformer 计算分析、Scaling Law、分词器技术；**核心组件**深入探讨 RoPE 位置编码与 SwiGLU 门控机制；**注意力机制**系统对比 MLA、线性注意力、FlashAttention 及稀疏注意力方案；**模型架构**聚焦 Mixture of Experts 的路由策略与负载均衡；**训练技术**涵盖数据工程、分布式训练 (TP/PP/ZeRO) 与 Muon 优化器；**评测体系**梳理 MMLU、GPQA、HumanEval 等主流 Benchmark；**部署优化**介绍量化技术与推理加速方法；**前沿应用**探讨多模态大模型与推理模型 (o1/R1) 的最新进展。

本文提供详细的数学推导、计算分析和代码示例，旨在为研究者和工程师提供一份完整的 LLM 技术参考。

关键词: 大语言模型, Transformer, 注意力机制, FlashAttention, Mixture of Experts, 分布式训练, 推理优化

目录

Part I

基础理论

1 引言

1.1 背景与动机

深度学习在序列建模任务中的成功离不开模型架构的演进。从早期的循环神经网络 (RNN) 到长短期记忆网络 (LSTM) [**?**]，再到门控循环单元 (GRU)，研究者们不断尝试解决长距离依赖问题。然而，这些基于循环结构的模型存在固有的局限性：

- **顺序计算**：RNN 必须按时间步依次处理序列，难以并行化
- **长距离依赖**：尽管 LSTM 引入了门控机制，信息在长序列中仍会衰减
- **计算效率**：训练和推理速度受限于序列长度

2017 年，Vaswani 等人提出了 Transformer 架构 [**?**]，完全摒弃循环结构，仅使用注意力机制建模序列。这一革命性设计带来了显著优势：

- 完全并行化的计算，大幅提升训练效率
- 通过自注意力机制直接建模任意位置之间的依赖关系
- 灵活的架构设计，易于扩展和迁移

基于 Transformer 的大语言模型 (LLM) 在随后几年取得了惊人的进展。从 GPT 系列 [**?** **?**] 到 LLaMA [**?**]、DeepSeek [**?**] 等开源模型，参数规模从亿级跃升至万亿级，能力边界不断拓展。这些模型不仅在自然语言处理任务上表现卓越，还展现出强大的涌现能力 (emergent abilities)，包括上下文学习、链式推理、代码生成等。

然而，随着模型规模和应用场景的扩展，新的挑战不断涌现：

- **计算效率**：标准注意力的 $O(n^2)$ 复杂度限制了长上下文建模
- **训练成本**：千亿参数模型需要数千 GPU 训练数月
- **部署挑战**：KV Cache 的内存占用成为推理瓶颈
- **能力边界**：复杂推理、多模态理解等任务仍具挑战性

本文旨在系统性地梳理大语言模型领域的核心技术，从基础理论到前沿应用，帮助读者建立完整的技术图谱。

1.2 文档结构

本文共分为八个部分，涵盖大语言模型的完整技术栈：

第一部分：基础理论 介绍 Transformer 的计算基础，包括硬件性能模型（Roofline Model）、内存层次结构、Transformer 各组件的 FLOPs 和参数量分析、Scaling Law（Kaplan 与 Chinchilla 定律）。这些基础知识对于理解后续的优化技术至关重要。

第二部分：Transformer 核心组件 深入探讨 Transformer 的三个关键组件：分词器（BPE、WordPiece、SentencePiece、Tiktoken）、位置编码（RoPE 及其长度外推方法）和门控机制（SwiGLU、Gated Attention 等）。这些组件的设计选择直接影响模型的性能和能力。

第三部分：注意力机制 这是本文的重点部分，系统介绍各类注意力机制的演进：

- **MLA**：DeepSeek 提出的低秩 KV 压缩方案，大幅降低 KV Cache
- **线性注意力**：将 $O(n^2)$ 降至 $O(n)$ 的理论方法，及其与状态空间模型的联系
- **FlashAttention**：IO 感知的高效注意力实现，从 v1 到 v4 的演进
- **稀疏注意力**：NSA、MoBA、DSA 等预训练级稀疏方案的深度对比

第四部分：模型架构 聚焦 Mixture of Experts (MoE) 架构，从基础的 Top-K 路由到 DeepSeek 的细粒度专家分割、无辅助损失负载均衡等工业级实践。

第五部分：训练技术　涵盖 LLM 训练的完整技术栈：预训练数据工程（数据来源、质量过滤、配比策略）、后训练数据（SFT 数据、偏好数据、合成数据）、分布式训练（数据并行 DDP/FSDP、模型并行 TP/PP、ZeRO 优化器），以及 Muon 等新型优化器的原理与实践。

第六部分：评测与 Benchmark　系统介绍 LLM 评测体系，包括知识类评测（MMLU、MMLU-Pro、GPQA）、推理能力（BBH、ARC）、数学能力（GSM8K、MATH-500、AIME）、代码能力（HumanEval、LiveCodeBench、SWE-bench）、指令遵循（IFEval、MT-Bench、Arena-Hard）、长上下文能力（RULER、LongBench）等主流 Benchmark，以及各前沿模型的评测对比。

第七部分：部署优化　介绍模型量化（从 INT8 到 FP8 再到极低位宽）和推理优化（PagedAttention、投机解码、KV Cache 压缩）的核心技术，以及 vLLM、SGLang 等主流推理框架。

第八部分：前沿应用　探讨两个快速发展的前沿方向：

- **多模态大模型**：视觉编码器、模态融合、原生多模态架构（GPT-4o、Gemini）、统一理解与生成（Janus、Show-o）
- **推理大模型**：测试时计算扩展、OpenAI o1、DeepSeek-R1、GRPO 算法、知识蒸馏

本文力求在深度与广度之间取得平衡，既提供数学公式和代码实现等技术细节，也注重阐述设计直觉和工程权衡。希望本文能够成为研究者和工程师理解大语言模型技术的有价值参考。

2 硬件与性能基础

在深入理解 Transformer 架构之前，我们需要掌握现代硬件的性能模型。本节介绍 Roofline 模型及相关概念，这些知识对于理解后续章节中的效率优化技术（如 FlashAttention、MoE）至关重要。

2.1 Roofline Model

Roofline 模型是分析程序性能的经典框架，它通过三个基本约束来界定算法的性能上限：

- 计算速度（FLOPs/秒）
- 数据传输带宽（字节/秒）
- 总内存容量

Definition 2.1 (Arithmetic Intensity)**.** *算术强度 (Arithmetic Intensity) 定义为单位数据传输所执行的计算量：*

$$Arithmetic\ Intensity = \frac{Total\ FLOPs}{Total\ Bytes\ Transferred} \tag{1}$$

单位通常为 FLOPs/Byte。

算术强度是判断程序瓶颈的关键指标。当算术强度高时，计算时间主导整体性能；当算术强度低时，内存带宽成为瓶颈。

2.2 Memory Hierarchy

现代加速器（GPU/TPU）存在明显的内存层次结构，不同层级的带宽和容量差异巨大：

表 1: GPU 内存层次结构对比（以 NVIDIA H100 为例）

Memory Type	Capacity	Bandwidth	Latency
Registers	∼256KB/SM	∼20 TB/s	1 cycle
Shared Memory (SRAM)	228KB/SM	∼19 TB/s	∼20 cycles
L2 Cache	50MB	∼12 TB/s	∼200 cycles
HBM (Global Memory)	80GB	3.35 TB/s	∼400 cycles

这种层次结构意味着：

- SRAM 访问比 HBM 快约 10 倍
- 算法设计应尽量减少 HBM 访问，最大化数据复用
- FlashAttention 等技术正是利用了这一特性

2.3 Compute-bound vs Memory-bound

程序的性能瓶颈可以分为两类：

计算时间与访存时间

$$T_{\text{compute}} = \frac{\text{FLOPs}}{\text{Peak FLOPs/s}} \tag{2}$$

$$T_{\text{memory}} = \frac{\text{Bytes}}{\text{Bandwidth (Bytes/s)}} \tag{3}$$

总执行时间的界限为：

$$\max(T_{\text{compute}}, T_{\text{memory}}) \leq T_{\text{total}} \leq T_{\text{compute}} + T_{\text{memory}} \tag{4}$$

Definition 2.2 (Critical Arithmetic Intensity)**.** 硬件的临界算术强度定义为：

$$I_{critical} = \frac{Peak\ FLOPs/s}{Memory\ Bandwidth\ (Bytes/s)} \tag{5}$$

- 当 $I < I_{\text{critical}}$ 时，程序是 **Memory-bound**，内存带宽是瓶颈
- 当 $I > I_{\text{critical}}$ 时，程序是 **Compute-bound**，计算能力是瓶颈

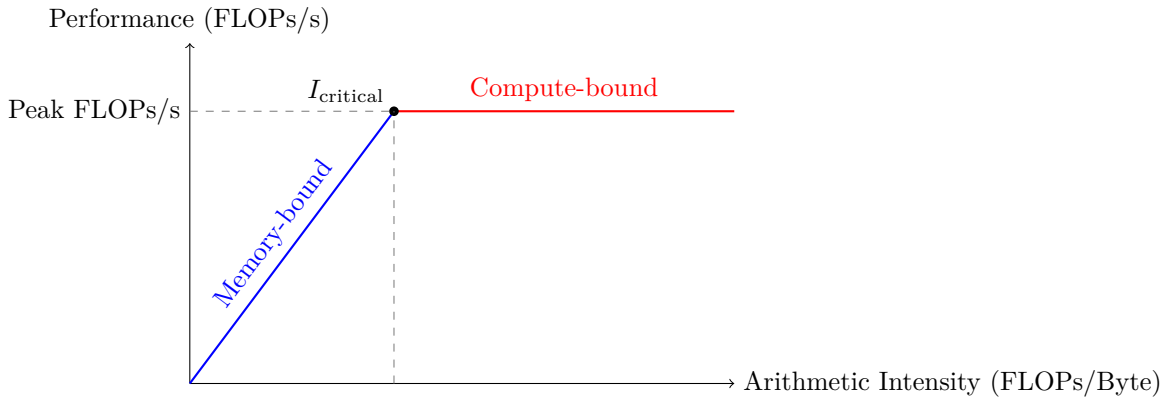

图 1: Roofline 模型示意图：性能受内存带宽和计算能力的双重约束

2.4 Hardware Specifications

表 2列出了主流 AI 加速器的关键规格和临界算术强度。

表 2: 主流 AI 加速器规格对比

Hardware	Peak FLOPs/s (BF16)	HBM Bandwidth	I_{critical}
NVIDIA A100	312 TFLOPs	2.0 TB/s	∼156
NVIDIA H100	990 TFLOPs	3.35 TB/s	∼296
Google TPU v5e	197 TFLOPs	820 GB/s	∼240
AMD MI300X	1,307 TFLOPs	5.3 TB/s	∼247

2.5 Matrix Multiplication Analysis

矩阵乘法是 Transformer 的核心计算。对于 $C = AB$，其中 $A \in \mathbb{R}^{B\times D}$，$B \in \mathbb{R}^{D\times F}$：

$$\text{FLOPs} = 2BDF \tag{6}$$

$$\text{Bytes} = 2(BD + DF + BF) \quad \text{(以 fp16/bf16 计)} \tag{7}$$

因此算术强度为：

$$I = \frac{2BDF}{2(BD + DF + BF)} = \frac{BDF}{BD + DF + BF} \tag{8}$$

当 $B \ll D, F$ 时（小 batch 场景），近似为：

$$I \approx B \tag{9}$$

这意味着：

- 小 batch 推理通常是 memory-bound
- 增大 batch size 可以提高计算效率
- 对于 H100，batch size 需要超过 ∼300 才能充分利用计算能力

Example 2.1 (Transformer 推理分析)**.** *考虑一个 LLaMA-7B 模型在 H100 上的推理:*

- *模型维度 $d = 4096$，FFN 维度 $d_{ff} = 11008$*
- *单 token 推理：$B = 1$，算术强度 $I \approx 1 \ll 296$，严重 memory-bound*
- *Batch size = 512：$I \approx 512 > 296$，可以达到 compute-bound*

这解释了为什么大 batch 推理吞吐量更高。

2.6 Implications for Transformer Design

Roofline 分析对 Transformer 的设计和优化有重要指导意义:

注意力机制 在常见实现中，标准注意力会显式构建大小为 $n \times n$ 的注意力矩阵，并多次读写到 HBM，导致 HBM 访存量随序列长度按 $O(n^2)$ 增长。因此在 Roofline 模型下，注意力计算是典型的 memory-bound kernel。FlashAttention 等优化方法通过将计算分块到片上 SRAM，避免显式构建 n^2 注意力矩阵并减少 HBM 读写，从而显著提升性能（详见 Section 8）。

KV Cache 自回归生成时，KV cache 的加载是主要瓶颈。MQA 和 GQA 通过减少 KV 头数来降低内存访问量。

模型并行 当模型过大无法放入单卡时，需要考虑:

- **Tensor Parallelism**: 切分矩阵维度，增加通信但保持计算
- **Pipeline Parallelism**: 切分层，引入 bubble 但减少通信
- **Expert Parallelism (MoE)**: 专家分布在不同设备，需要 all-to-all 通信

混合精度 使用 INT8 权重 + BF16 激活时:

- 权重加载字节数减半
- 算术强度翻倍
- 临界 batch size 降至原来的一半

理解这些硬件特性是设计高效 Transformer 系统的基础。

2.7 Distributed Training and Sharding

当模型规模超过单个加速器的内存容量时，需要将参数和计算分布到多个设备上。**Sharding**（分片）是将张量切分到设备网格上的核心技术。

Definition 2.3 (Sharding)**.** *给定一个全局形状为 $[I, J]$ 的张量 A 和一个设备网格 $\mathcal{M}$ 包含轴 X, Y，分片记号 $A[I_X, J_Y]$ 表示:*

- *第一维 I 沿网格轴 X 切分*
- *第二维 J 沿网格轴 Y 切分*
- *每个设备持有形状为 $[I/|X|, J/|Y|]$ 的本地分片*

分片约束 一旦某个网格维度被用于切分张量的某个维度，该网格维度就被" 消耗"，不能再用于同一张量的其他维度。

2.8 Communication Primitives

分布式计算中有四个核心通信原语。为便于分析，我们约定:

- V: 每设备本地参与通信的数据量（Bytes）

- N：参与通信的设备数
- W：每设备的有效互连带宽（Bytes/s）
- $\{U_X\}$：表示沿网格轴 X 的未归约（unreduced）状态

AllGather：收集分片 将分布在各设备的分片收集起来，使每个设备都获得完整数据。

$$A[I_X, J] \xrightarrow{\text{AllGather}} A[I, J] \tag{10}$$

之前：`D0:[A]  D1:[B]  D2:[C]  D3:[D]  每设备 V`
之后：`D0:[ABCD] D1:[ABCD] D2:[ABCD] D3:[ABCD]  每设备 4V`

- 输入：每设备持有 V（全局数据的 $1/N$）
- 输出：每设备持有 $N \cdot V$（完整副本）
- 通信量：每设备接收 $(N-1) \cdot V$
- 成本：$T \approx V/W$（ring 算法下，带宽被充分利用）

ReduceScatter：归约后分发 先对各设备数据求和（Reduce），再将结果切分到各设备（Scatter）。

$$A[I, J]\{U_X\} \xrightarrow{\text{ReduceScatter}} A[I, J_X] \tag{11}$$

之前：`D0:[A0,B0,C0,D0] D1:[A1,B1,C1,D1] D2:[A2,B2,C2,D2] D3:[A3,B3,C3,D3]`
`(每设备持有完整但未归约的梯度)`
之后：`D0:[ΣA] D1:[ΣB] D2:[ΣC] D3:[ΣD]`
`(每设备持有1/4的归约结果，其中 ΣA = A0+A1+A2+A3)`

- 输入：每设备持有 V（未归约，需要与其他设备的对应数据相加）
- 输出：每设备持有 V/N（归约结果的一个分片）
- 成本：$T \approx V/W$
- 典型用途：梯度聚合后分片存储（FSDP/ZeRO）

AllReduce：归约后广播 对所有设备的数据求和，并将完整结果发送给每个设备。等价于 ReduceScatter + AllGather。

$$A[I, J]\{U_X\} \xrightarrow{\text{AllReduce}} A[I, J] \tag{12}$$

之前：`D0:[A0,B0,C0,D0] D1:[A1,B1,C1,D1] D2:[...] D3:[...]  每设备 V`
之后：`D0:[ΣA,ΣB,ΣC,ΣD] D1:[ΣA,ΣB,ΣC,ΣD] D2:[...] D3:[...]  每设备 V`
`(每设备都持有完整的归约结果)`

- 输入：每设备持有 V（未归约）
- 输出：每设备持有 V（完整的归约结果）
- 成本：$T \approx 2V/W$（两步操作）
- 典型用途：DDP 梯度同步

AllToAll：重新分片 将数据从一种分片方式转换为另一种，相当于”转置”切分轴。

$$A[I, J_X] \xrightarrow{\text{AllToAll}} A[I_X, J] \tag{13}$$

之前：`D0:[A0,A1,A2,A3] D1:[B0,B1,B2,B3] D2:[C0,C1,C2,C3] D3:[D0,D1,D2,D3]`
`(按"行"切分：每设备持有一整行)`

之后：D0:[A0,B0,C0,D0] D1:[A1,B1,C1,D1] D2:[A2,B2,C2,D2] D3:[A3,B3,C3,D3]
（按"列"切分：每设备持有一整列）

- 输入：每设备持有 V（按 J 维切分）
- 输出：每设备持有 V（按 I 维切分，内容不同但大小相同）
- 通信量：每设备发送 $(N-1)/N \cdot V \approx V$，接收同量
- 成本：$T \approx V/W$

表 3: 通信原语完整对比（$N = 4$ 设备，每设备本地数据量 V）

	AllGather	ReduceScatter	AllReduce	AllToAll
数据变化	分片 → 完整	完整 → 分片	完整 → 完整	分片 → 分片
输入大小	V	V	V	V
输出大小	$N \cdot V$	V/N	V	V
是否归约	否	是	是	否
通信成本	V/W	V/W	$2V/W$	V/W
典型用途	TP 激活收集	ZeRO 梯度分片	DDP 梯度同步	MoE 路由

四种通信原语对比 关键洞察：

- **AllGather**：收集分片，每设备获得相同的完整副本
- **ReduceScatter**：归约后分片，每设备获得不同的归约结果分片
- **AllReduce**：归约后广播，每设备获得相同的完整归约结果（= ReduceScatter + AllGather）
- **AllToAll**：重新分片，每设备获得不同的数据（不归约，仅重分布）

AllToAll 在 MoE 中的应用 MoE（Mixture of Experts）是 AllToAll 的典型应用场景。假设 N 个设备各持有一个 Expert：

1. **Router** 决定每个 token 应由哪个 Expert 处理
2. **Dispatch**（AllToAll）：将 token 发送到目标 Expert 所在设备
3. **Expert 计算**：每设备用本地 Expert 处理收到的 token
4. **Combine**（AllToAll）：将结果发回原设备

AllToAll 使得 token 能够高效地" 找到" 分布在不同设备上的 Expert，而无需在每个设备上复制所有 token。

2.9 Distributed Matrix Multiplication

分布式矩阵乘法 $C = AB$ 的通信策略取决于输入的分片模式：

Case 1: 非收缩维分片（无通信）

$$A[I_X, J] \cdot B[J, K_Y] \to C[I_X, K_Y] \tag{14}$$

输出自动继承输入的分片模式，无需通信。

Case 2: 单输入收缩维分片（AllGather）

$$A[I, J_X] \cdot B[J, K] \xrightarrow{\text{AllGather}} A[I, J] \cdot B[J, K] \to C[I, K] \tag{15}$$

需要先 AllGather 移除 A 的分片。

Case 3: 双输入同轴收缩维分片（AllReduce）

$$A[I, J_X] \cdot B[J_X, K] \to C[I, K]\{U_X\} \xrightarrow{\text{AllReduce}} C[I, K] \tag{16}$$

本地矩阵乘法产生部分和，AllReduce 完成归约。

Case 4: 双输入同轴非收缩维分片（非法）

$$A[I_X, J] \cdot B[J, K_X] \to \text{Invalid (diagonal result)} \tag{17}$$

必须先 AllGather 其中一个输入。

Example 2.2 (策略选择). 对于 $C = AB$，其中 $A \in \mathbb{R}^{B\times D}$，$B \in \mathbb{R}^{D\times F}$，比较两种策略：

策略 *1*（*AllGather* 优先）：

$$T_1 = \underbrace{\frac{2DF}{W}}_{AllGather\ B} + \underbrace{\frac{2BDF}{C}}_{Compute} \tag{18}$$

策略 *2*（*AllReduce* 优先）：

$$T_2 = \underbrace{\frac{4BF}{W}}_{AllReduce\ C} + \underbrace{\frac{2BDF/X}{C}}_{Local\ Compute} \tag{19}$$

当 $D > 2X$（模型维度远大于并行度）时，策略 *2* 更优。

2.10 Parallelism Strategies

大规模 Transformer 训练通常结合多种并行策略：

Data Parallelism (DP) 将 batch 维度切分到多个设备，每个设备持有完整模型副本：

- 前向传播：各设备独立计算
- 反向传播：AllReduce 同步梯度
- 优点：实现简单，通信量与模型大小成正比
- 缺点：内存冗余（每设备存完整模型）

Fully Sharded Data Parallelism (FSDP/ZeRO) 将参数、梯度、优化器状态都沿数据维度切分：

$$\text{Memory/device} = \frac{\text{Model Size}}{N_{\text{devices}}} + \text{Activations} \tag{20}$$

计算前 AllGather 参数，计算后 ReduceScatter 梯度。

Tensor Parallelism (TP) 将矩阵维度切分，每层内部并行计算：

- Column Parallel：$W[D, F_X]$，前向 AllGather，反向 ReduceScatter
- Row Parallel：$W[D_X, F]$，前向 ReduceScatter，反向 AllGather

通常在 Transformer 的 Attention 和 FFN 中交替使用，每层需要 2 次 AllReduce。

Pipeline Parallelism (PP) 将模型层切分到不同设备，形成流水线：

- 优点：通信量少（仅传递激活值）
- 缺点：存在流水线气泡（bubble），降低硬件利用率
- 优化：1F1B 调度、交错调度等减少 bubble

Expert Parallelism (EP) MoE 模型中，不同专家分布在不同设备：

$$\text{Input}[B, D] \xrightarrow{\text{AllToAll}} \text{Expert}_i[B_i, D] \xrightarrow{\text{Compute}} \xrightarrow{\text{AllToAll}} \text{Output}[B, D] \tag{21}$$

需要 AllToAll 进行 token 路由和结果收集。

Sequence Parallelism (SP) 将序列维度切分，与 TP 配合使用：

- LayerNorm 和 Dropout 沿序列维度切分
- 减少激活值内存
- 与 TP 共享通信，无额外开销

2.11 Communication-Computation Overlap

高效的分布式系统通过重叠通信和计算来隐藏通信延迟：

$$T_{\text{total}} = \max(T_{\text{compute}}, T_{\text{comm}}) \quad \text{(理想情况)} \tag{22}$$

实现技术：

- **异步通信**：在计算进行时启动下一次通信
- **分块流水线**：将大张量切分为小块，交替计算和通信
- **梯度累积**：多个 micro-batch 累积后再同步

理解这些分布式并行技术是扩展 Transformer 到千亿参数规模的关键。

3 Transformer 计算分析

本节从计算角度系统分析 Transformer 架构，涵盖 FLOPs 计算、参数量统计、内存占用，以及推理优化。理解这些数学关系对于模型设计、训练规划和部署优化至关重要。

3.1 符号定义

表 4列出了本节使用的符号。图 2展示了 Transformer 单层的完整计算流程。

表 4: Transformer 计算分析符号表

符号	含义
B	Batch size
T	序列长度（Sequence length）
D	模型维度（Hidden dimension）
F	FFN 中间维度（通常 $F = 4D$ 或 $\frac{8}{3}D$）
L	Transformer 层数
N	Query head 数量
K	KV head 数量（MHA: $K = N$，GQA: $K < N$，MQA: $K = 1$）
H	每个 head 的维度（通常 $H = D/N$）
V	词表大小（Vocabulary size）
P	模型总参数量

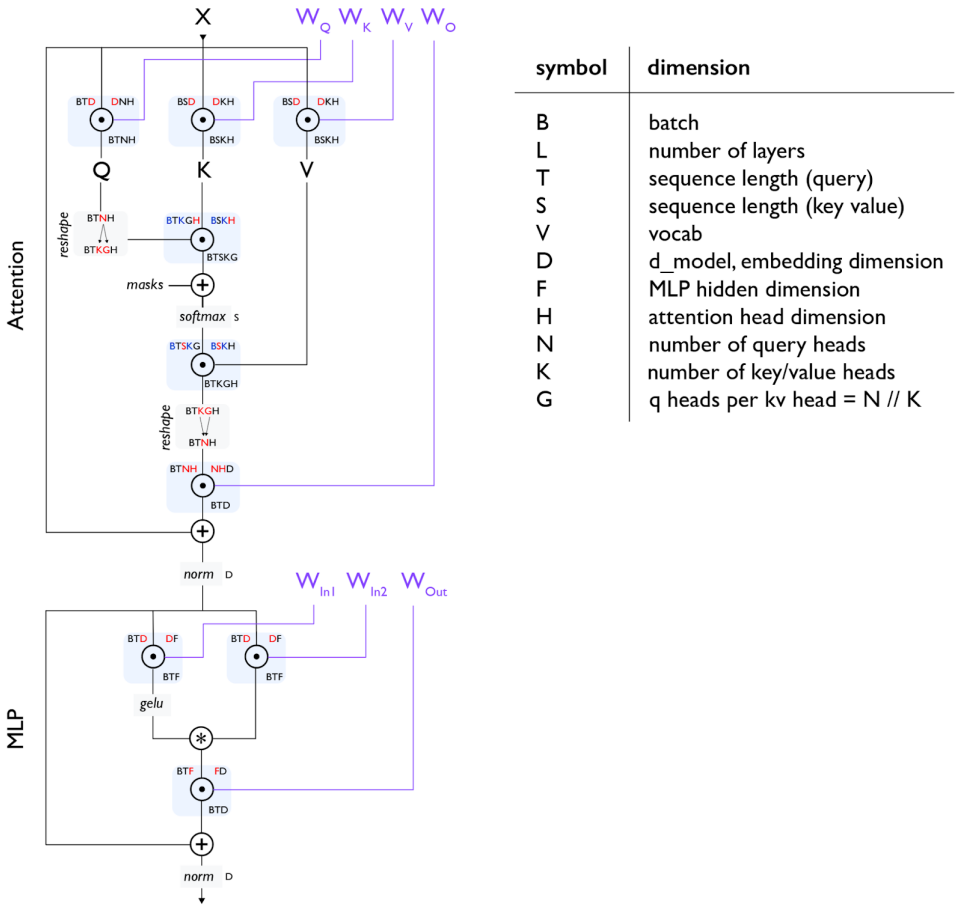

图 2: Transformer 层的计算流程与张量维度。上：Attention 模块；下：MLP 模块（SwiGLU）。

3.2 基础运算

矩阵运算是 Transformer 的计算基础。设矩阵维度为 m, n, k：

表 5: 基本矩阵运算的 FLOPs 与数据传输

运算	表达式	FLOPs	数据量
向量点积	$\boldsymbol{x}\cdot\boldsymbol{y},\ \boldsymbol{x},\boldsymbol{y}\in\mathbb{R}^k$	$2k$	$2k$
矩阵-向量乘	$A\boldsymbol{x},\ A\in\mathbb{R}^{m\times k}$	$2mk$	$mk+k$
矩阵-矩阵乘	$AB,\ A\in\mathbb{R}^{m\times k}, B\in\mathbb{R}^{k\times n}$	$2mkn$	$mk+kn$

前向与反向传播 对于线性层 $Y=XW$（$X\in\mathbb{R}^{m\times k}$，$W\in\mathbb{R}^{k\times n}$）：

- 前向：$Y=XW$，FLOPs $=2mkn$
- 反向：$\frac{\partial L}{\partial X}=\frac{\partial L}{\partial Y}W^\top$（$2mkn$）+ $\frac{\partial L}{\partial W}=X^\top\frac{\partial L}{\partial Y}$（$2mkn$）
- 总计：$6mkn$ FLOPs

这导出了训练 FLOPs 的核心公式：

$$\boxed{\text{Training FLOPs}\approx 6\times P\times T_{\text{tokens}}} \tag{23}$$

其中 P 是参数量，T_{tokens} 是训练 token 总数。

3.3 单层分析

Transformer 每层包含两个核心模块：Attention 和 MLP。

MLP 层

MLP 层（也称 FFN）有两种常见形式：

Standard FFN

$$\text{FFN}(x)=W_2\cdot\text{GELU}(W_1x),\quad W_1\in\mathbb{R}^{D\times F},W_2\in\mathbb{R}^{F\times D} \tag{24}$$

SwiGLU FFN

$$\text{SwiGLU}(x)=W_2\cdot(\text{SiLU}(W_1x)\odot W_3x),\quad W_1,W_3\in\mathbb{R}^{D\times F},W_2\in\mathbb{R}^{F\times D} \tag{25}$$

其中 $\odot$ 是逐元素乘法，起**门控**作用：W_3x 控制 $\text{SiLU}(W_1x)$ 的信息流通。

表 6: MLP 层参数与 FLOPs（per layer, 输入 shape $[B,T,D]$）

类型	参数量	前向 FLOPs	训练 FLOPs
Standard FFN	$2DF$	$4BTDF$	$12BTDF$
SwiGLU	$3DF$	$6BTDF$	$18BTDF$

Remark 3.1 (参数量一致性). 为保持总参数量一致，不同结构调整 F 的取值：

- *Standard FFN*：$F=4D\Rightarrow$ 参数量 $=8D^2$
- *SwiGLU*：$F=\frac{8}{3}D\Rightarrow$ 参数量 $=8D^2$

现代模型（*LLaMA*、*DeepSeek* 等）普遍采用 *SwiGLU*。

Attention 层

Multi-Head Attention 包含四个投影和注意力计算：

$$Q=XW_Q,\quad K=XW_K,\quad V=XW_V,\quad O=\text{Attn}(Q,K,V)W_O \tag{26}$$

其中 $W_Q, W_O \in \mathbb{R}^{D\times D}$，$W_K, W_V \in \mathbb{R}^{D\times KH}$。

$$\text{Attn}(Q,K,V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{H}}\right)V \tag{27}$$

表 7: Attention 层参数与 FLOPs（per layer, GQA with K KV heads）

组件	参数量	训练 FLOPs
Q projection	D^2	$6BTD^2$
K projection	DKH	$6BTDKH$
V projection	DKH	$6BTDKH$
O projection	D^2	$6BTD^2$
$QK^\top$	—	$6BT^2NH$
softmax $\cdot\, V$	—	$6BT^2NH$
Total	$2D^2 + 2DKH$	$12BTD^2 + 12BTDKH + 12BT^2NH$

MHA / GQA / MQA 对比

- **MHA**（Multi-Head Attention）：$K = N$，每个 head 独立 KV
- **GQA**（Grouped-Query Attention）：$1 < K < N$，多个 Q head 共享 KV
- **MQA**（Multi-Query Attention）：$K = 1$，所有 head 共享一个 KV

Attention vs MLP 计算量对比

简化假设（$F = 4D$，$K \ll N$，$NH = D$）下：

$$\frac{\text{Attention FLOPs}}{\text{MLP FLOPs}} \approx \frac{T}{8D} \tag{28}$$

Remark 3.2. 当 $T < 8D$ 时，***MLP* 计算量主导**。对于 $D = 8192$ 的模型，序列长度需超过 65536 才能使注意力成为主要计算瓶颈。这解释了为什么长上下文场景才需要特别关注注意力效率。

3.4 完整模型分析

总参数量

完整 Transformer 模型的参数组成：

$$P_{\text{embed}} = VD \quad \text{（词嵌入）} \tag{29}$$

$$P_{\text{attn}} = L\cdot(2D^2 + 2DKH) \quad \text{（注意力层）} \tag{30}$$

$$P_{\text{mlp}} = L\cdot 3DF \quad \text{（MLP 层，SwiGLU）} \tag{31}$$

$$P_{\text{norm}} = L\cdot 4D \quad \text{（LayerNorm）} \tag{32}$$

$$P_{\text{head}} = DV \quad \text{（输出投影）} \tag{33}$$

总参数量（不共享 embedding 时）：

$$\boxed{P_{\text{total}} = 2VD + L\cdot(2D^2 + 2DKH + 3DF + 4D)} \tag{34}$$

Example 3.1 (LLaMA-7B 参数计算)**.** $D = 4096$, $F = 11008$, $L = 32$, $V = 32000$, $N = K = 32$（MHA），

$H = 128$:

$$P_{embed} = 32000 \times 4096 = 131M \tag{35}$$

$$P_{attn/layer} = 2 \times 4096^2 + 2 \times 4096 \times 32 \times 128 = 67M \tag{36}$$

$$P_{mlp/layer} = 3 \times 4096 \times 11008 = 135M \tag{37}$$

$$P_{total} \approx 2 \times 131M + 32 \times (67M + 135M) \approx \mathbf{6.7B} \tag{38}$$

训练计算量

每 token 的训练 FLOPs（忽略注意力中的 T^2 项）：

$$\text{FLOPs/token} \approx 6P + 12LT \cdot NH \tag{39}$$

第一项 $6P$ 是参数相关计算（占主导），第二项是注意力的序列长度相关计算。

表 8: 常见模型的计算量

模型	参数量	前向 FLOPs/token	训练 FLOPs/token
GPT-2	1.5B	3B	9B
LLaMA-7B	6.7B	13B	40B
LLaMA-70B	70B	140B	420B
GPT-4 (est.)	1.8T	3.6T	11T

训练内存占用

训练时的内存组成：

表 9: 训练内存组成（P 为参数量）

组件	内存	说明
参数（bf16）	$2P$	模型权重
梯度（bf16）	$2P$	反向传播梯度
优化器状态（Adam, fp32）	$8P$	momentum + variance
激活值	$O(BTD \cdot L)$	中间结果，用于反向传播
总计（无优化）	$\approx 12P + \text{activations}$	

Activation Checkpointing 通过只保存每层输入、重新计算中间激活来节省内存：

- 内存：从 $O(L \cdot BTD)$ 降至 $O(BTD)$
- 代价：额外约 33% 重计算（$6P \to 8P$）

3.5 推理分析

推理与训练有本质区别：无需梯度和优化器状态，但自回归生成引入新的挑战。

KV Cache

自回归生成时，每生成一个 token 需要 attend 到所有历史 token。为避免重复计算，需缓存历史的 K 和 V：

$$\boxed{\text{KV Cache Size} = 2 \times B \times S \times L \times K \times H \times \text{bytes}} \tag{40}$$

其中 S 是当前序列长度，K 是 KV head 数，H 是 head 维度。

Example 3.2 (KV Cache 计算)**.** *70B 模型：$D = 8192$, $L = 80$, $K = 8$（GQA）, $H = 128$, $S = 8192$, bf16 精度：*

$$KV\ Cache = 2 \times 1 \times 8192 \times 80 \times 8 \times 128 \times 2 = \mathbf{2.1}\ GB/request \tag{41}$$

KV Cache 的影响

- 内存瓶颈：限制最大 batch size 和上下文长度
- GQA/MQA：通过减少 K 降低 cache 大小（如从 32 降到 8，减少 4 倍）
- 量化：int8/int4 进一步压缩（2-4 倍）

Prefill vs Decode

推理分为两个阶段，具有不同的计算特性：

表 10: Prefill vs Decode 对比

	Prefill	Decode
输入	整个 prompt（T tokens）	单个 token
计算模式	并行处理所有 token	逐 token 生成
瓶颈	Compute-bound	Memory-bound
主要开销	矩阵乘法计算	权重加载 + KV Cache 读写
优化方向	提高计算利用率	增大 batch、减少访存

推理优化策略

Batching 策略

- **Continuous Batching**：动态插入/移除请求，提高 GPU 利用率
- **Chunked Prefill**：将长 prompt 分块处理，与 decode 交错执行

投机解码（Speculative Decoding） 用小模型快速生成候选 token 序列，大模型并行验证：

- 小模型（draft）：快速生成 k 个候选 token
- 大模型（verify）：一次前向验证所有候选，接受正确的前缀
- 加速比：$1.5\times \sim 3\times$（取决于 draft 模型准确率）

量化

- **Weight-only**（W4A16）：权重 int4，激活 fp16，减少权重加载
- **Full quantization**（W8A8）：权重和激活都 int8，利用 INT8 Tensor Core

4 Scaling Law

Scaling Law（缩放定律）揭示了大语言模型性能与计算量、数据量、模型规模之间的幂律关系，是指导 LLM 训练资源分配的核心理论。

4.1 基本概念

什么是 Scaling Law

Scaling Law 描述了模型性能（通常用损失函数 L 衡量）如何随资源增加而改善：

$$L = \left(\frac{N_c}{N}\right)^{\alpha_N} + \left(\frac{D_c}{D}\right)^{\alpha_D} + L_\infty \tag{42}$$

其中：

- N：模型参数量
- D：训练数据量（tokens）
- L_∞：不可约损失（数据本身的熵）
- $N_c, D_c, \alpha_N, \alpha_D$：拟合常数

为什么 Scaling Law 重要

- **资源分配**：给定计算预算，如何分配模型大小和数据量
- **性能预测**：在小规模实验预测大规模模型的性能
- **投资决策**：估算达到目标性能所需的资源

4.2 Kaplan Scaling Law (2020)

OpenAI 的 Kaplan 等人 [**?**] 首次系统研究了 LLM 的 Scaling Law。

核心发现

性能与规模的幂律关系

$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N}, \quad \alpha_N \approx 0.076 \tag{43}$$

$$L(D) = \left(\frac{D_c}{D}\right)^{\alpha_D}, \quad \alpha_D \approx 0.095 \tag{44}$$

$$L(C) = \left(\frac{C_c}{C}\right)^{\alpha_C}, \quad \alpha_C \approx 0.050 \tag{45}$$

其中 C 是计算量（FLOPs）。

关键结论

1. **模型规模主导**：在固定计算预算下，更大的模型（训练更少步数）比小模型（训练更多步数）更优
2. **最优分配**：计算量增加 10 倍时，模型参数应增加约 5.5 倍，数据量增加约 1.8 倍
3. **架构不敏感**：Scaling Law 对 Transformer 的具体超参数（层数、宽度）不敏感

计算最优模型

Kaplan 建议的计算最优分配：

$$N_{opt} \propto C^{0.73}, \quad D_{opt} \propto C^{0.27} \tag{46}$$

这意味着随着计算预算增加，应该更多地投资于模型规模而非数据量。

4.3 Chinchilla Scaling Law (2022)

DeepMind 的 Hoffmann 等人 [**?**] 挑战了 Kaplan 的结论，提出了更均衡的分配策略。

核心发现

数据的重要性被低估 Chinchilla 研究表明，之前的模型普遍欠拟合（undertrained）：

- GPT-3 (175B)：训练了 300B tokens，但最优应该是约 3.5T tokens
- Gopher (280B)：训练了 300B tokens，同样严重不足

新的最优分配

$$N_{opt} \propto C^{0.50}, \quad D_{opt} \propto C^{0.50} \tag{47}$$

更简洁的表述：

$$D_{opt} \approx 20 \times N \tag{48}$$

即**最优训练数据量约为模型参数的 20 倍**。

为什么是 20 倍？推导直觉

这个看似任意的常数背后有清晰的数学逻辑。考虑计算预算约束 $C = 6ND$（FLOPs $\approx 6 \times$ 参数 $\times$ tokens），目标是最小化损失：

$$L(N, D) = \frac{A}{N^\alpha} + \frac{B}{D^\beta} + L_\infty \tag{49}$$

在约束 $C = 6ND$ 下求极值，使用拉格朗日乘数法：

$$\frac{\partial L}{\partial N} \Big/ \frac{\partial L}{\partial D} = \frac{D}{N} \tag{50}$$

代入幂律形式：

$$\frac{\alpha A / N^{\alpha+1}}{\beta B / D^{\beta+1}} = \frac{D}{N} \implies \frac{D}{N} = \left(\frac{\alpha A}{\beta B}\right)^{\frac{1}{\alpha+\beta+2}} \tag{51}$$

Chinchilla 的实验测得 $\alpha \approx 0.34$，$\beta \approx 0.28$，代入后得到 $D/N \approx 20$。

物理直觉 这个结果可以这样理解：模型参数需要足够的” 训练信号” 才能收敛到最优值。每个参数平均需要看到约 20 个 tokens 的信息才能学到有意义的表示。参数太多（相对于数据）会欠拟合——模型有容量但缺乏信号；数据太多（相对于参数）会浪费——模型容量已饱和，额外数据无法被利用。

Kaplan vs Chinchilla：争议的本质 两个团队得出不同结论的根本原因在于**实验设计**：

- Kaplan 固定训练步数，变化模型大小——这倾向于发现” 大模型更好”
- Chinchilla 固定计算量，同时变化模型和数据——这才能找到真正的帕累托最优

Chinchilla 的设计更接近实际资源分配问题：给定固定预算，如何分配？

Chinchilla 模型验证

DeepMind 训练了 70B 参数的 Chinchilla 模型，使用 1.4T tokens：

- 计算量与 Gopher (280B) 相同
- 性能全面超越 Gopher
- 推理成本降低 4 倍（参数量是 1/4）

表 11: Chinchilla vs Gopher 性能对比

模型	参数量	训练 Tokens	MMLU
Gopher	280B	300B	60.0%
Chinchilla	70B	1.4T	67.6%

对工业界的影响

Chinchilla Law 深刻影响了后续模型的设计：

表 12: 主流模型的参数-数据配比

模型	参数量	训练 Tokens	Tokens/参数
GPT-3	175B	300B	1.7×
Chinchilla	70B	1.4T	20×
LLaMA	65B	1.4T	21.5×
LLaMA 2	70B	2T	28.6×
LLaMA 3	70B	15T	214×
Qwen 2	72B	7T+	97×

4.4 超越 Chinchilla：推理最优的视角

Chinchilla 法则解决的是" 训练最优" 问题：给定计算预算，如何最小化训练后的损失? 但在工业部署中，问题的形式不同：给定**总预算**（训练 + 推理），如何最大化用户价值?

训练-推理权衡的数学框架

设总成本为训练成本加上生命周期内的推理成本：

$$\text{Total Cost} = C_{train} + n_{infer} \times C_{infer}(N) \tag{52}$$

其中 $C_{train} = 6ND$，$C_{infer}(N) \propto N$（推理成本与参数量成正比）。

关键洞察 Chinchilla 最优假设 $n_{infer} = 0$——只考虑训练。但当 n_{infer} 很大时，推理成本主导总成本。此时最优策略发生根本转变：

- **小模型**：C_{infer} 低，即使每个 query 成本微小，累积量也可观
- **Over-training**：用额外训练成本换取更小模型的性能，摊薄到海量推理中

定量分析 假设目标性能固定，两种策略的成本对比：

1. Chinchilla 最优：70B 模型 + 1.4T tokens，$C_{train} \propto 70 \times 1.4 = 98$
2. Over-training：8B 模型 + 15T tokens，$C_{train} \propto 8 \times 15 = 120$

训练成本增加 22%，但推理成本降低 88%（8/70）。当 $n_{infer} > 10^9$ 时（对于 ChatGPT 级产品很常见），Over-training 策略的总成本显著更低。

LLaMA 的策略

LLaMA 系列采用激进的 Over-training：

- LLaMA-7B：1T tokens（143× 参数）
- LLaMA 2-7B：2T tokens（286× 参数）
- LLaMA 3-8B：15T tokens（1875× 参数）

Remark 4.1 (Over-training 的收益递减)**.** 虽然 *Over-training* 有益，但收益递减：

- 早期：每增加 *1* 倍数据，性能显著提升
- 后期：需要更多数据才能获得同样提升
- 存在实际上限：数据重复使用最终会导致过拟合

4.5 多维度 Scaling Law

下游任务 Scaling

预训练 Loss 的 Scaling Law 不完全转化为下游任务性能：

- **涌现能力**：某些能力在特定规模突然出现
- **任务依赖**：不同任务的 Scaling 曲线不同
- **评估敏感**：度量方式影响观察到的 Scaling 行为

RLHF/DPO 的 Scaling

后训练也遵循 Scaling Law，但形式不同：

- **基座模型规模**：更大的基座模型对齐效果更好
- **偏好数据量**：收益递减明显，质量比数量重要
- **Reward Model**：RM 规模通常应与 Policy 规模匹配

推理时 Scaling（Test-time Compute）

OpenAI o1 等推理模型展示了新的 Scaling 维度：

$$\text{Performance} = f(\text{Pretraining Compute}, \text{Inference Compute}) \tag{53}$$

- 通过更多推理时计算（更长的思考链）提升性能
- 与预训练计算形成互补
- 开辟了 “用推理换性能” 的新范式

4.6 Scaling Law 的局限与争议

局限性

- **外推风险**：从小规模实验外推大规模可能失准
- **数据质量**：Scaling Law 假设数据质量恒定
- **架构依赖**：不同架构可能有不同的 Scaling 行为
- **任务特定**：通用 Scaling Law 可能不适用于特定任务

数据墙（Data Wall）

高质量数据枯竭

- 互联网高质量文本有限（估计约 10-15T tokens）
- 继续 Scaling 需要合成数据或其他来源
- 数据重复使用有上限

应对策略

- **合成数据**：用模型生成训练数据
- **数据增强**：改写、翻译、多样化
- **多模态数据**：图像、视频、音频

- **代码数据**：GitHub 等代码仓库

涌现能力的争议

涌现是真实的吗? 有研究认为 "涌现能力" 可能是评估度量的假象：

- 使用连续度量（如 token-level 准确率）替代离散度量
- 重新分析后，很多 "涌现" 变成平滑的 Scaling

实践意义 无论涌现是否真实存在，大模型确实展现出小模型没有的能力，这对应用仍然重要。

4.7 Scaling Law 的实践应用

预测模型性能

利用 Scaling Law 进行性能预测：

1. 训练多个小规模模型（不同大小、不同数据量）
2. 拟合 Scaling Law 参数
3. 外推预测大规模模型性能
4. 决定是否值得投资大规模训练

资源分配决策

给定计算预算 C，如何分配：

训练最优（Chinchilla）

$$N = \sqrt{C/6D} \tag{54}$$

$$D = 20N \tag{55}$$

推理最优 根据预期推理量 n_{infer} 调整：

- n_{infer} 小：接近 Chinchilla 最优
- n_{infer} 大：Over-train 更小的模型

Scaling Law 拟合示例

```python
import numpy as np
from scipy.optimize import curve_fit

def scaling_law(x, a, b, c):
    """L = a / x^b + c"""
    return a / np.power(x, b) + c

# 实验数据：(参数量, 损失)
params = [125e6, 350e6, 760e6, 1.3e9, 2.7e9, 6.7e9]
losses = [3.50, 3.25, 3.10, 2.95, 2.80, 2.65]

# 拟合
popt, _ = curve_fit(scaling_law, params, losses)
a, b, c = popt

# 预测70B模型的损失
pred_loss = scaling_law(70e9, a, b, c)
print(f"Predicted loss for 70B: {pred_loss:.2f}")
```

Remark 4.2 (Scaling Law 使用建议).

- **多点拟合**：至少 *3-5* 个不同规模的数据点
- **保守外推**：外推倍数不宜过大（通常 *<10* 倍）
- **任务相关**：关注目标任务的 *Scaling*，而非只看 *Loss*
- **持续监控**：大规模训练时监控是否符合预期
- **小规模验证**：在投入大规模训练前，用小模型验证假设

Part II

Transformer 核心组件

5 分词器（Tokenizer）

分词器是大语言模型的入口，负责将原始文本转换为模型可处理的 token 序列。分词策略的选择直接影响模型的词表大小、训练效率和多语言能力。

5.1 为什么需要分词

神经网络无法直接处理文本，需要将文本转换为数值表示：

$$\text{“Hello world”} \xrightarrow{\text{Tokenizer}} [15496, 995] \xrightarrow{\text{Embedding}} \mathbb{R}^{2\times d} \tag{56}$$

分词粒度的选择面临**词表大小**与**序列长度**的权衡：

表 13: 不同分词粒度的对比

粒度	词表大小	序列长度	问题
字符级	∼256	很长	序列过长，难以建模长距离依赖
词级	∼100K+	短	OOV 问题，词表过大
子词级	∼32K-128K	适中	平衡，主流选择

5.2 子词分词算法

Byte Pair Encoding (BPE)

BPE [**?**] 是最广泛使用的子词分词算法，源自数据压缩领域。

训练算法

1. 初始化词表为所有字符（或字节）
2. 统计相邻 token 对的频率
3. 将最高频的 token 对合并为新 token，加入词表
4. 重复步骤 2-3，直到达到目标词表大小

Example 5.1 (BPE 训练过程)**.** *假设语料为：“low lower lowest”*

1. *初始：* `l, o, w, e, r, s, t, _`（`_` *表示词边界*）
2. *最高频对* `(l, o)` → *合并为* `lo`
3. *最高频对* `(lo, w)` → *合并为* `low`
4. *最高频对* `(low, e)` → *合并为* `lowe`
5. *...*

最终词表包含常见子词如 `low`, `er`, `est` *等。*

分词算法 给定训练好的合并规则，按优先级依次应用：

```python
def bpe_tokenize(text, merges):
    tokens = list(text)  # 初始为字符
    for (a, b) in merges:  # 按训练顺序
        i = 0
        while i < len(tokens) - 1:
            if tokens[i] == a and tokens[i+1] == b:
                tokens = tokens[:i] + [a+b] + tokens[i+2:]
            else:
                i += 1
    return tokens
```

Byte-level BPE

GPT-2 [**?**] 引入的改进，直接在字节级别操作：

- 基础词表为 256 个字节，无需预分词
- 可以表示任何 UTF-8 文本，无 OOV 问题
- 避免了不同语言的特殊处理

```python
# GPT-2的字节到Unicode映射
def bytes_to_unicode():
    # 将256个字节映射到可打印Unicode字符
    # 避免空白字符等控制字符的问题
    bs = list(range(33, 127)) + list(range(161, 173)) + list(range(174, 256))
    cs = bs[:]
    n = 0
    for b in range(256):
        if b not in bs:
            bs.append(b)
            cs.append(256 + n)
            n += 1
    return dict(zip(bs, [chr(c) for c in cs]))
```

WordPiece

WordPiece 由 Google 提出，用于 BERT [**?**]。与 BPE 的主要区别在于合并策略：

- **BPE**：选择频率最高的 token 对
- **WordPiece**：选择使语言模型似然提升最大的 token 对

WordPiece 的合并分数：

$$\text{score}(a, b) = \frac{\text{freq}(ab)}{\text{freq}(a) \times \text{freq}(b)} \tag{57}$$

WordPiece 使用 `##` 标记非词首子词：

```
"tokenization" -> ["token", "##ization"]
```

Unigram Language Model

Unigram [**?**] 采用相反的策略——从大词表开始，逐步删减：

1. 初始化一个较大的候选词表
2. 用 EM 算法估计每个 token 的概率
3. 计算移除每个 token 对似然的影响
4. 移除影响最小的 token
5. 重复直到达到目标词表大小

Unigram 的优势：

- 同一文本可能有多种分词方式，支持采样
- 可用于数据增强（Subword Regularization）

5.3 SentencePiece

SentencePiece [**?**] 是 Google 开源的分词工具，特点：

- **语言无关**：将空格视为普通字符（用 _ 表示），无需预分词
- **支持多种算法**：BPE 和 Unigram
- **可逆**：分词结果可以无损还原原文

- **高效**：C++ 实现，训练和推理都很快

```python
import sentencepiece as spm

# 训练
spm.SentencePieceTrainer.train(
    input='corpus.txt',
    model_prefix='tokenizer',
    vocab_size=32000,
    model_type='bpe',  # 或 'unigram'
    character_coverage=0.9995,
)

# 使用
sp = spm.SentencePieceProcessor(model_file='tokenizer.model')
tokens = sp.encode('Hello world', out_type=str)
# [' Hello', ' world']  ( 表示空格)
```

5.4 Tiktoken

Tiktoken 是 OpenAI 开源的快速 BPE 实现，用于 GPT-3.5/4：

- **Rust 实现**：比 Python 实现快 3-6 倍
- **正则预分词**：用正则表达式预先切分，提高效率
- **特殊 token 处理**：支持 `<|endoftext|>` 等特殊标记

```python
import tiktoken

# GPT-4使用cl100k_base编码
enc = tiktoken.get_encoding("cl100k_base")
tokens = enc.encode("Hello world")  # [9906, 1917]
text = enc.decode(tokens)  # "Hello world"

# 查看词表大小
print(enc.n_vocab)  # 100277
```

预分词正则 Tiktoken 使用复杂的正则表达式预切分文本：

```python
# cl100k_base的预分词正则（简化）
pat = r"""'s|'t|'re|'ve|'m|'ll|'d|
    [\p{L}]+|[\p{N}]{1,3}|
    [^\s\p{L}\p{N}]+|\s+(?!\S)|\s+"""
```

这确保了常见模式（如缩写、数字）被合理切分。

5.5 词表大小的选择

词表大小是重要的设计决策：

词表大小的影响

- **更大词表**：
  - 更短的序列长度（同样文本用更少 token）
  - 更大的 Embedding 层参数（$V \times d$）
  - 更好的多语言支持
- **更小词表**：

表 14: 主流模型的词表配置

模型	词表大小	分词器
GPT-2	50,257	Byte-level BPE
GPT-3/3.5	50,257	Byte-level BPE
GPT-4	100,277	Byte-level BPE (cl100k)
BERT	30,522	WordPiece
LLaMA	32,000	SentencePiece BPE
LLaMA 2	32,000	SentencePiece BPE
LLaMA 3	128,256	Tiktoken BPE
Qwen	151,936	Byte-level BPE
DeepSeek	102,400	Byte-level BPE

- 更长的序列（需要更多 token 表示）
- 更小的模型参数
- 稀有词可能被过度切分

经验法则

- 英文为主：32K-50K 通常足够
- 多语言：100K+ 可以更好覆盖非英语语言
- LLaMA 3 将词表从 32K 扩展到 128K，中文 token 效率提升约 3 倍

5.6 特殊 Token

特殊 token 用于标记序列结构和控制生成：

表 15: 常见特殊 Token

Token	含义	用途
`<|endoftext|>`	文本结束	分隔文档，标记生成结束
`<|im_start|>`	消息开始	ChatML 格式的消息边界
`<|im_end|>`	消息结束	ChatML 格式的消息边界
`[PAD]`	填充	批处理时对齐序列长度
`[CLS]`	分类	BERT 的序列表示
`[SEP]`	分隔	BERT 的句子分隔
`[MASK]`	掩码	BERT 的 MLM 任务

ChatML 格式 现代 Chat 模型使用结构化格式区分角色：

```
<|im_start|>system
You are a helpful assistant.
<|im_end|>
<|im_start|>user
Hello!
<|im_end|>
<|im_start|>assistant
Hi there!
<|im_end|>
```

5.7 多语言分词：一个被低估的公平性问题

分词器的设计深刻影响着不同语言用户的使用体验。这不仅是技术问题，更是公平性问题。

Token 效率差异的根源

不同语言的 token 效率差异显著：

表 16: 不同语言的 Token 效率（GPT-4，相同语义内容）

语言	Token 数	相对英语
英语	100	1.0×
西班牙语	120	1.2×
中文	150	1.5×
日语	180	1.8×
缅甸语	400	4.0×

语言学解释 效率差异源于**书写系统**与**训练数据**的双重因素：

1. **字母文字 vs 表意文字**：英语 26 个字母组合成词，BPE 容易学到常见子词（如 `-tion`, `-ing`）。中文每个字是独立语素，约需 3500 个常用字覆盖 99.9% 文本，但 BPE 难以学到跨字的有意义组合。
2. **训练数据偏斜**：BPE 按频率合并 token 对。当训练语料中英文占 90% 时，英文子词被充分合并（`the`, `and` 成为单 token），而中文词汇因频率低而保持拆分状态。
3. **UTF-8 编码开销**：英文字符占 1 字节，中文字符占 3 字节。在 Byte-level BPE 中，一个中文字至少需要 3 个基础 token 才能表示，除非被合并。

定量分析 设语言 L 在训练语料中的占比为 p_L，其 token 效率大致满足：

$$\text{效率}_L \propto p_L^{\alpha}, \quad \alpha \approx 0.3\text{–}0.5 \tag{58}$$

这解释了为什么缅甸语（训练数据中几乎没有）效率是英语的 1/4——缅甸文字几乎没有被合并，每个字符都被拆成多个字节 token。

实际影响

对非英语用户而言：

- **成本**：相同语义内容消耗 1.5-4 倍 token，API 费用相应增加
- **上下文**：有效上下文窗口缩短（128K tokens 对中文用户相当于英文用户的 85K）
- **延迟**：生成相同内容需要更多解码步骤
- **质量**：过度切分破坏词汇边界，可能影响语义理解

改进策略

扩大词表 LLaMA 3 将词表从 32K 扩展到 128K，新增大量非英语 token。效果：中文 token 效率提升约 3 倍。代价：Embedding 层参数增加 4 倍（从 32K×d 到 128K×d）。

平衡训练语料 在分词器训练时对语言进行上采样，使低资源语言获得更多合并机会。但这会牺牲高资源语言的效率——存在帕累托权衡。

语言感知分词 对特定语言使用专门的预分词规则。例如，中文按字切分后再 BPE，避免跨字合并产生无意义 token。日语可以利用形态分析工具（如 MeCab）进行预切分。

Remark 5.1 (公平性视角)**.** 分词效率差异本质上是训练资源分配的结果。当商业模型按 *token* 计费时，这种差异直接转化为经济不平等——使用小语种的用户为相同服务付出更高代价。这是 *AI* 公平性讨论中容易被忽视的技术细节。

5.8 分词器的实现细节

Tokenizer 的组成

一个完整的分词器包含：

1. **Normalizer**：文本标准化（Unicode 归一化、大小写等）
2. **Pre-tokenizer**：预分词（按空格、标点切分）
3. **Model**：核心分词算法（BPE/WordPiece/Unigram）
4. **Post-processor**：后处理（添加特殊 token）
5. **Decoder**：将 token ID 还原为文本

HuggingFace Tokenizers

HuggingFace 的 `tokenizers` 库提供高效实现：

```python
from tokenizers import Tokenizer, models, trainers, pre_tokenizers

# 创建BPE分词器
tokenizer = Tokenizer(models.BPE())
tokenizer.pre_tokenizer = pre_tokenizers.ByteLevel()

# 训练
trainer = trainers.BpeTrainer(vocab_size=32000, special_tokens=["<|endoftext|>"])
tokenizer.train(files=["corpus.txt"], trainer=trainer)

# 使用
output = tokenizer.encode("Hello world")
print(output.tokens)  # ['Hello', 'Ġworld']
```

5.9 分词的计算开销

分词通常不是瓶颈，但在某些场景需要注意：

- **在线推理**：分词延迟约 0.1-1ms，通常可忽略
- **大规模预处理**：TB 级语料分词可能需要数小时
- **优化方法**：
  - 使用 Rust/C++ 实现（如 Tiktoken）
  - 并行处理
  - 预分词缓存

Remark 5.2 (分词器选择建议).

- **从头训练 *LLM***：使用 *SentencePiece* 或 *HuggingFace Tokenizers* 训练专用分词器
- **微调现有模型**：使用原模型的分词器，保持一致性
- **多语言场景**：选择词表较大（*100K+*）的分词器
- **特殊领域**（代码、数学）：考虑在分词器训练时包含领域数据

6 位置编码

不同于 RNN、CNN 等模型，Transformer 的自注意力机制本身是**置换不变的**（permutation invariant）——纯粹的 Attention 模块无法捕捉输入顺序，即无法区分不同位置的 Token。为此，位置编码的引入是必不可少的。位置编码大体有两种思路：

1. **绝对位置编码**：将位置信息融入到输入中，即 $\boldsymbol{x}'_k = \boldsymbol{x}_k + \boldsymbol{p}_k$ 或 $\boldsymbol{x}'_k = \boldsymbol{x}_k \odot \boldsymbol{p}_k$
2. **相对位置编码**：微调 Attention 结构，使其能分辨不同位置的 Token

表 17: 位置编码方法分类与对比

类型	方法	作用位置	外推性	代表模型
绝对	Sinusoidal	Embedding	差	Transformer
	Learned	Embedding	差	BERT, GPT
	FLOATER	Embedding	中	—
相对	经典式	Attention	中	NEZHA
	T5 Bias	Attention score	好	T5
	ALiBi	Attention score	好	BLOOM, MPT
	RoPE	Q/K 向量	好	LLaMA, Qwen

6.1 绝对位置编码

绝对位置编码的一般形式是在输入的第 k 个向量 $\boldsymbol{x}_k$ 中加入位置向量 $\boldsymbol{p}_k$，其中 $\boldsymbol{p}_k$ 只依赖于位置编号 k。

训练式（Learned）

最朴素的方案是将位置编码作为可训练参数。例如最大长度 512、编码维度 768，则初始化一个 512×768 的矩阵，随训练更新。BERT、GPT 等模型采用此方案。

优点 简单直接，让模型自己学习位置表示。

缺点 缺乏外推性——如果预训练最大长度为 512，则无法处理更长的序列。虽然可以将超长位置随机初始化后继续微调，但效果有限。

三角函数式（Sinusoidal）

原始 Transformer [**?**] 采用正弦余弦函数生成位置编码：

$$p_{k,2i} = \sin\left(k/10000^{2i/d}\right) \tag{59}$$

$$p_{k,2i+1} = \cos\left(k/10000^{2i/d}\right) \tag{60}$$

其中 $p_{k,2i}, p_{k,2i+1}$ 分别是位置 k 的编码向量的第 $2i, 2i+1$ 个分量。

设计直觉 不同维度对应不同频率的周期函数：低维变化快（捕捉局部位置），高维变化慢（捕捉全局位置）。由三角恒等式 $\sin(\alpha+\beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta$，位置 $\alpha+\beta$ 的向量可表示为位置 α 和 β 的向量组合，这提供了表达相对位置信息的可能性。

局限性 尽管有显式的生成规律，但实践中 Sinusoidal 编码的外推能力仍然有限，现代 LLM 已很少直接使用。

递归式（FLOATER）

ICML 2020 的论文提出用微分方程建模位置编码：

$$\frac{d\boldsymbol{p}_t}{dt} = h(\boldsymbol{p}_t, t) \tag{61}$$

其中 h 可以用神经网络建模。这种方法称为 FLOATER，理论上具有较好的外推性（可以证明 Sinusoidal 是其特解），但牺牲了并行性。

相乘式

除了 $\boldsymbol{x}_k + \boldsymbol{p}_k$ 的加法融合，也可以考虑 $\boldsymbol{x}_k \odot \boldsymbol{p}_k$ 的逐位相乘。有实验表明相乘方式可能取得更好的效果，但目前主流模型仍以相加为主。

6.2 相对位置编码

相对位置编码不完整建模每个输入的位置信息，而是在计算 Attention 时考虑当前位置与被 attend 位置的**相对距离**。由于自然语言更依赖相对位置，这类方法通常表现优秀。

经典式

Google 的论文《Self-Attention with Relative Position Representations》首次提出相对位置编码。考虑带绝对位置编码的 Attention：

$$q_i k_j^\top = (x_i + p_i) W_Q W_K^\top (x_j + p_j)^\top \tag{62}$$

展开后将 $p_j W_K$ 替换为二元位置向量 R^K_{i-j}（只依赖相对距离），并进行截断：

$$R^K_{i-j} = p^K[\text{clip}(i-j, p_{\min}, p_{\max})] \tag{63}$$

这样只需有限个位置编码就可以表达任意长度的相对位置。华为的 NEZHA 模型采用了这种编码。

Transformer-XL / XLNet 式

Transformer-XL 对 $q_i k_j^\top$ 完全展开：

$$q_i k_j^\top = x_i W_Q W_K^\top x_j^\top + x_i W_Q W_K^\top p_j^\top + p_i W_Q W_K^\top x_j^\top + p_i W_Q W_K^\top p_j^\top \tag{64}$$

将 p_j 替换为相对位置向量 R_{i-j}（使用 Sinusoidal 生成），p_i 替换为可训练向量 u, v：

$$x_i W_Q W_K^\top x_j^\top + x_i W_Q W_{K,R}^\top R_{i-j}^\top + u W_K^\top x_j^\top + v W_{K,R}^\top R_{i-j}^\top \tag{65}$$

T5 式

T5 进一步简化，认为” 输入-位置” 交互项应该解耦，直接删除中间两项：

$$x_i W_Q W_K^\top x_j^\top + \beta_{i,j} \tag{66}$$

其中 $\beta_{i,j}$ 是只依赖于 (i,j) 的可训练偏置。T5 的特色是对相对位置进行**分桶**处理：邻近位置（0∼7）精细区分，远距离位置共用编码，距离越远共用范围越大。

DeBERTa 式

DeBERTa 保留” 输入-位置” 和” 位置-输入” 项，去掉” 位置-位置” 项：

$$q_i k_j^\top = x_i W_Q W_K^\top x_j^\top + x_i W_Q W_K^\top R_{i,j}^\top + R_{j,i} W_Q W_K^\top x_j^\top \tag{67}$$

DeBERTa 还提出了混合使用相对位置（前 11 层）和绝对位置（后 2 层）的策略，在 SuperGLUE 上取得了优异成绩。

6.3 旋转位置编码（RoPE）

?]提出的 RoPE 是目前最主流的位置编码方法，被 LLaMA、Mistral、Qwen 等模型广泛采用。RoPE 的核心思想是**融合绝对位置与相对位置**：通过在 q、k 上施加绝对位置的旋转操作，使得内积自然地只依赖于相对位置。

Remark 6.1 (RoPE 的理论起源)**.** *RoPE 的设计灵感来自复数的性质：两个复数的内积* $\langle qe^{im\theta}, ke^{in\theta}\rangle = Re[q\bar{k}e^{i(m-n)\theta}]$ *只依赖相对位置* $m-n$*。这一洞察使得我们可以用绝对位置的操作（乘以* $e^{im\theta}$*）达到相对位置的效果。*

问题设定

我们希望通过下述运算给 $\boldsymbol{q}, \boldsymbol{k}$ 添加绝对位置信息：

$$\tilde{\boldsymbol{q}}_m = f(\boldsymbol{q}, m), \quad \tilde{\boldsymbol{k}}_n = f(\boldsymbol{k}, n) \tag{68}$$

由于 Attention 的核心是内积，我们希望内积结果带有**相对位置信息**，即存在恒等关系：

$$\langle f(\boldsymbol{q}, m), f(\boldsymbol{k}, n)\rangle = g(\boldsymbol{q}, \boldsymbol{k}, m-n) \tag{69}$$

初始条件设为 $f(\boldsymbol{q}, 0) = \boldsymbol{q}$，$f(\boldsymbol{k}, 0) = \boldsymbol{k}$。

二维情况的求解

借助复数来求解。在复数中有 $\langle \boldsymbol{q}, \boldsymbol{k}\rangle = \mathrm{Re}[\boldsymbol{q}\bar{\boldsymbol{k}}]$，所以：

$$\mathrm{Re}[f(\boldsymbol{q}, m)\overline{f(\boldsymbol{k}, n)}] = g(\boldsymbol{q}, \boldsymbol{k}, m-n) \tag{70}$$

假设存在复数 $g(\boldsymbol{q}, \boldsymbol{k}, m-n)$ 使得 $f(\boldsymbol{q}, m)\overline{f(\boldsymbol{k}, n)} = g(\boldsymbol{q}, \boldsymbol{k}, m-n)$。用复数的指数形式：

$$f(\boldsymbol{q}, m) = R_f(\boldsymbol{q}, m)e^{i\Theta_f(\boldsymbol{q}, m)} \tag{71}$$

$$f(\boldsymbol{k}, n) = R_f(\boldsymbol{k}, n)e^{i\Theta_f(\boldsymbol{k}, n)} \tag{72}$$

代入后得到方程组：

$$R_f(\boldsymbol{q}, m)R_f(\boldsymbol{k}, n) = R_g(\boldsymbol{q}, \boldsymbol{k}, m-n) \tag{73}$$

$$\Theta_f(\boldsymbol{q}, m) - \Theta_f(\boldsymbol{k}, n) = \Theta_g(\boldsymbol{q}, \boldsymbol{k}, m-n) \tag{74}$$

求解模长 代入 $m = n$：

$$R_f(\boldsymbol{q}, m)R_f(\boldsymbol{k}, m) = R_g(\boldsymbol{q}, \boldsymbol{k}, 0) = R_f(\boldsymbol{q}, 0)R_f(\boldsymbol{k}, 0) = \|\boldsymbol{q}\|\|\boldsymbol{k}\| \tag{75}$$

最简单的解是 $R_f(\boldsymbol{q}, m) = \|\boldsymbol{q}\|$，$R_f(\boldsymbol{k}, m) = \|\boldsymbol{k}\|$，即**模长不依赖于位置**。

求解幅角 同样代入 $m = n$：

$$\Theta_f(\boldsymbol{q}, m) - \Theta_f(\boldsymbol{k}, m) = \Theta_g(\boldsymbol{q}, \boldsymbol{k}, 0) = \Theta(\boldsymbol{q}) - \Theta(\boldsymbol{k}) \tag{76}$$

这说明 $\Theta_f(\boldsymbol{q}, m) - \Theta(\boldsymbol{q})$ 只与 m 相关，记为 $\varphi(m)$。再代入 $n = m-1$，可得 $\{\varphi(m)\}$ 是等差数列，即 $\varphi(m) = m\theta$。

最终解 综上，二维情况下的 RoPE 为：

$$\boxed{f(\boldsymbol{q}, m) = \|\boldsymbol{q}\| e^{i(\Theta(\boldsymbol{q})+m\theta)} = \boldsymbol{q} e^{im\theta}} \tag{77}$$

这正是向量乘以旋转因子 $e^{im\theta}$，对应**旋转角度** $m\theta$。

矩阵形式与高维扩展

二维旋转的矩阵形式：

$$f(\boldsymbol{q}, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} q_0 \\ q_1 \end{pmatrix} \tag{78}$$

由于内积满足线性叠加性，d 维向量可分成 $d/2$ 对，每对独立旋转：

$$f(\boldsymbol{q}, m) = \begin{pmatrix} R_{\theta_0}(m) & & \\ & R_{\theta_1}(m) & \\ & & \ddots \end{pmatrix} \boldsymbol{q} \tag{79}$$

关键性质：给位置 m 的 $\boldsymbol{q}$ 乘 R_m，位置 n 的 $\boldsymbol{k}$ 乘 R_n，则：

$$(R_m \boldsymbol{q})^\top (R_n \boldsymbol{k}) = \boldsymbol{q}^\top R_m^\top R_n \boldsymbol{k} = \boldsymbol{q}^\top R_{n-m} \boldsymbol{k} \tag{80}$$

内积自动包含相对位置信息。此外，R_m 是正交矩阵，**不改变向量模长**，保持模型稳定性。

旋转矩阵形式

完整的 RoPE 公式为：

$$\mathrm{RoPE}(\boldsymbol{x}, m) = \underbrace{\begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & & & \\ \sin m\theta_1 & \cos m\theta_1 & & & \\ & & \cos m\theta_2 & -\sin m\theta_2 & \\ & & \sin m\theta_2 & \cos m\theta_2 & \\ & & & & \ddots \end{pmatrix}}_{R_\Theta(m)} \boldsymbol{x} \tag{81}$$

频率参数 频率 θ_i 采用指数递减的形式：

$$\theta_i = \mathrm{base}^{-2(i-1)/d}, \quad i = 1, 2, \ldots, d/2 \tag{82}$$

其中 base $= 10000$ 是原始设置。不同 θ_i 对应不同的" 波长"，小 i 对应高频（短波长，捕捉局部位置），大 i 对应低频（长波长，捕捉全局位置）。

Base 选择原则

?]从理论上分析了 RoPE 的 base 值与上下文长度的关系，揭示了 **base 存在下界**这一重要性质（参见苏剑林的解读 [**?**]）。

语义聚合性质 设 $\boldsymbol{q}$ 是 query 向量，$\boldsymbol{k}^* = \boldsymbol{q} + \boldsymbol{\epsilon}$ 是语义相似的 key（$\boldsymbol{\epsilon}$ 为小扰动），$\boldsymbol{k}$ 是随机 key。定义**语义区分度**为模型区分相似 token 和随机 token 的能力：

$$B_{m,\theta} = \frac{1}{2\sigma^2} \left(\mathbb{E}[\boldsymbol{q}^\top R_{m,\theta} \boldsymbol{k}^*] - \mathbb{E}[\boldsymbol{q}^\top R_{m,\theta} \boldsymbol{k}] \right) = \sum_{i=1}^{d/2} \cos(m\theta_i) \tag{83}$$

Long-term Decay 随着相对距离 m 增大，$B_{m,\theta}$ 逐渐减小（注意力衰减），且当 base 过小时会变为负值——此时模型反而给随机 token 更高的注意力，丧失语义理解能力。

Base 下界 为保证所有上下文位置的语义区分度非负，base 必须满足：

$$b^* = \inf\{b \mid B_{m,\theta} \geq 0, \forall m \in \{0, 1, \ldots, L-1\}\} \tag{84}$$

数值求解得到不同上下文长度对应的 base 下界：

表 18: RoPE Base 下界与上下文长度的关系

Context Length L	4K	8K	32K	128K	1M
Base 下界 b^*	4.5×10^4	8.4×10^4	6.4×10^5	3.4×10^6	6.5×10^7

渐近分析表明 $b^* \approx O(L)$，即 base 应随上下文长度**线性增长**。

实际模型的 Base 选择

- **LLaMA 3**：训练长度 8192，但 base 选择了 500000，远超下界（8.4×10^4），可能是为更长上下文预留
- **Mixtral**：base = 1000000，支持 128K 上下文

与 Position Interpolation 的联系 语义聚合视角也解释了 PI 的原理：同一个 base 下，$B_{m,\theta}$ 的非负区间是固定的。PI 通过缩小位置间隔（$0, 1/s, 2/s, \ldots$）使更多位置落入非负区间，而非增大 base。两种方法殊途同归。

Remark 6.2 (OOD 理论的不足)**.** 早期基于 *Out-of-Distribution*（*OOD*）的分析认为 *base* 越小越好（使位置遍历整个圆）。但语义聚合视角揭示了 *base* 存在下界——过小的 *base* 会导致模型虽然保持低困惑度，却丧失长距离信息检索能力，形成”表面长上下文能力”。

远程衰减性质

RoPE 具有一个重要的理论性质：**随着相对距离增大，内积会逐渐衰减**。这与自然语言的局部性假设相吻合——距离越近的 token 通常关联越强。

设 $\boldsymbol{q}, \boldsymbol{k}$ 是随机向量，RoPE 编码后的内积期望为：

$$\mathbb{E}[\langle f(\boldsymbol{q}, m), f(\boldsymbol{k}, n)\rangle] \propto \sum_{i=1}^{d/2} \cos((m-n)\theta_i) \tag{85}$$

当 $m = n$ 时（同一位置），内积最大；随着 $|m-n|$ 增大，各维度的余弦项相位错开，求和结果减小。这种衰减是**非单调**的（因为余弦函数），但长程平均效果是衰减的。

与 Sinusoidal 编码的对比 Sinusoidal 编码虽然在设计时也考虑了相对位置表达，但它只是将位置信息**加入**到向量中，相对位置需要模型自己学习。而 RoPE 通过旋转操作，**在数学上保证**内积只依赖于相对位置，无需学习。

几何直觉

RoPE 的核心直觉可以总结为：

> ”根据位置旋转 *query* 和 *key* 向量，让点积自然地揭示它们的相对距离。”

- **相同位置偏移保持角度不变**：如果 $\boldsymbol{q}$ 和 $\boldsymbol{k}$ 同时偏移相同的位置（绝对位置改变，相对位置不变），两者会被旋转相同的角度，它们之间的夹角不变，点积也不变

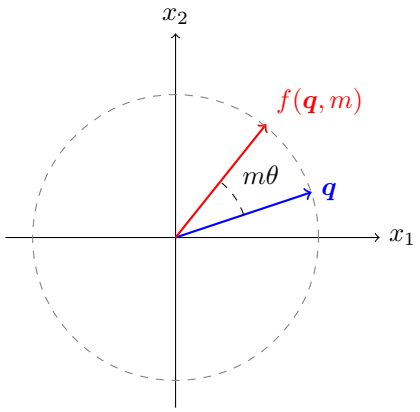

图 3: RoPE 的几何直觉：位置编码 = 向量旋转。位置 m 处的向量被旋转 $m\theta$ 角度。

- **距离衰减**：相对位置越远，累积旋转角度越大，高频维度的点积贡献会因相位错开而减小，形成自然的距离衰减

高效实现

直接使用旋转矩阵需要 $O(d^2)$ 计算。但注意到旋转矩阵是**分块对角**的，每个 2×2 块独立作用，可以展开为逐元素运算。

对于二维情况：

$$\begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} q_0 \\ q_1 \end{pmatrix} = \begin{pmatrix} q_0 \cos m\theta - q_1 \sin m\theta \\ q_0 \sin m\theta + q_1 \cos m\theta \end{pmatrix} \tag{86}$$

推广到 d 维，可以写成向量形式：

$$\text{RoPE}(\boldsymbol{x}, m) = \boldsymbol{x} \odot \cos(m\boldsymbol{\theta}) + \text{rotate\_half}(\boldsymbol{x}) \odot \sin(m\boldsymbol{\theta}) \tag{87}$$

其中 $\boldsymbol{\theta} = (\theta_1, \theta_1, \theta_2, \theta_2, \ldots)$ 是频率向量（每个频率重复两次），$\odot$ 表示逐元素乘法。这种实现的复杂度为 $O(d)$。

Remark 6.3 (实现风格)**.** *存在两种实现风格，区别在于如何定义 rotate_half 操作：*

- ***GPT-NeoX* 风格**：*将向量分成前后两半*

$$\textit{rotate\_half}([x_0, \ldots, x_{d/2-1}, x_{d/2}, \ldots, x_{d-1}]) = [-x_{d/2}, \ldots, -x_{d-1}, x_0, \ldots, x_{d/2-1}]$$

- ***GPT-J* 风格**：*相邻两个维度配对（interleaved）*

$$\textit{rotate\_half}([x_0, x_1, x_2, x_3, \ldots]) = [-x_1, x_0, -x_3, x_2, \ldots]$$

LLaMA 官方实现采用 GPT-J 风格，但 Hugging Face Transformers 库默认采用 GPT-NeoX 风格。两者数学上等价（只是维度排列不同），但加载预训练权重时需要注意兼容性。

6.4 RoPE 的应用

Dense 模型

RoPE 已成为主流 Dense 模型的标准配置：

MoE 模型

MoE 模型同样采用 RoPE，但通常配置更大的 base 值以支持更长上下文：

表 19: RoPE 在 Dense 模型中的应用

Model	Base	Max Position	Context Length
LLaMA	10000	2048	2K
LLaMA 2	10000	4096	4K
Mistral-7B	10000	32768	32K
Qwen-7B	10000	8192	8K

表 20: RoPE 在 MoE 模型中的应用

Model	Base	Max Position	Experts	Notes
Mixtral-8x7B	1000000	131072	8	Sliding window attention
Mixtral-8x22B	1000000	65536	8	—
DeepSeek-MoE	10000	4096	64	Fine-grained experts

MoE 的特殊考虑 MoE 模型中不同 expert 可能专门处理不同位置模式或模态。LLaMA 4 引入了 **iRoPE**（interleaved RoPE）架构：部分层使用 RoPE，部分层不使用位置编码，依靠因果 attention mask 和学习到的模式推断位置。这种混合方法使 LLaMA 4 Scout 能够支持高达 1000 万 token 的上下文窗口（仅在 256K token 上训练）。

6.5 长度外推方法

RoPE 的一个关键问题是**长度外推**（length extrapolation）：如何让模型处理超出训练长度的序列?

Position Interpolation (PI)

最简单的方法是缩放位置索引 [**?**]：

$$m' = \frac{m}{s}, \quad s = \frac{L'}{L} \tag{88}$$

其中 L 是训练长度，L' 是目标长度。

直觉 将长序列" 压缩" 到原始位置范围内。例如，要将 4K 模型扩展到 16K，令 $s = 4$，则位置 16000 被映射到位置 4000。

问题 均匀缩放会破坏高频信息，因为高频维度对位置变化敏感。

NTK-aware Interpolation

基于 Neural Tangent Kernel 理论，调整 base 而非均匀缩放 [**?**]：

$$\text{base}' = \text{base} \cdot s^{d/(d-2)} \tag{89}$$

直觉 将插值压力分散到不同维度：高频维度（编码局部位置）少插值，低频维度（编码全局位置）多插值。

NTK-by-parts

进一步改进，对不同维度采用不同的插值策略：

$$\theta_i' = \begin{cases} \theta_i & \text{if } \lambda_i > \beta \cdot L \text{ (高频，不插值)} \\ \theta_i / s & \text{if } \lambda_i < \alpha \cdot L \text{ (低频，完全插值)} \\ \text{interpolate} & \text{otherwise (中间，渐进插值)} \end{cases} \tag{90}$$

其中 $\lambda_i = 2\pi/\theta_i$ 是波长。对 LLaMA 模型，$\alpha = 1$，$\beta = 32$ 效果最优。

YaRN

YaRN（Yet another RoPE extensioN） [**?**] 结合 NTK-by-parts 和**注意力温度缩放**：

$$\text{Attention}'_{ij} = \frac{\boldsymbol{q}_i^\top \boldsymbol{k}_j}{\sqrt{d} \cdot t} \tag{91}$$

温度系数 t 根据扩展比例 s 设定：

$$\sqrt{1/t} = 0.1 \cdot \ln(s) + 1 \tag{92}$$

效果 YaRN 只需约 400 步微调即可将 LLaMA 2 从 4K 扩展到 64K，相比 PI 减少 10 倍数据和 2.5 倍训练步数。

表 21: 长度外推方法对比

Method	Extrapolation	Finetuning	Notes
PI	2×	Required	均匀缩放
NTK-aware	32×	Optional	无需微调时效果好
NTK-by-parts	16×	Required	分维度处理
YaRN	16×	Minimal	结合温度缩放
Dynamic	64×	None	推理时动态调整

Dynamic Scaling

动态缩放在推理时根据当前序列长度自适应调整插值因子：

$$s = \max\left(1, \frac{\text{current\_length}}{L}\right) \tag{93}$$

优势

- 无需微调
- 短序列保持原始性能
- 长序列自动启用插值

6.6 RoPE vs ALiBi

ALiBi [**?**] 是另一种流行的相对位置编码方法，直接在 attention score 上添加距离惩罚：

$$\text{Attention}_{ij} = \frac{\boldsymbol{q}_i^\top \boldsymbol{k}_j}{\sqrt{d}} - m \cdot |i - j| \tag{94}$$

其中 m 是每个 head 的斜率参数。

表 22: RoPE 与 ALiBi 的详细对比

Feature	RoPE	ALiBi
编码位置	Q/K 向量	Attention score
参数量	0	0
外推能力	中等（需扩展方法）	好
训练速度	基准	更快
KV Cache 友好	是（旋转一次写入）	是
采用模型	LLaMA, Mistral, Qwen	BLOOM, MPT

选择建议

- 如果需要极长上下文且不想微调，ALiBi 可能更合适
- 如果使用主流开源模型生态（LLaMA 系），RoPE 是默认选择
- RoPE 配合 YaRN 等扩展方法可以达到与 ALiBi 相当的外推能力

6.7 最新进展

Resonance RoPE (2024) 针对 OOD（Out-of-Distribution）位置的特征插值问题进行优化，进一步提升长度外推能力。

iRoPE (LLaMA 4) 混合使用 RoPE 层和无位置编码层，配合推理时注意力温度缩放，实现从 256K 训练长度到 10M 上下文窗口的极端外推。

2D/3D RoPE 将 RoPE 扩展到二维（图像）和三维（视频）位置编码，应用于 Vision Transformer 和多模态模型。

Remark 6.4 (为什么 RoPE 成为主流)**.** *RoPE* 成功的关键因素：

1. **零参数**：不增加模型参数
2. **流式友好**：*KV Cache* 只需旋转一次
3. **可扩展**：配合 *PI/YaRN* 等方法可处理超长上下文
4. **生态支持**：*LLaMA* 系模型的标准配置

7 Transformer 中的门控机制

在前几节中，我们分析了 Transformer 的计算复杂度和位置编码设计。本节关注另一个核心问题：如何让模型学会**选择性地传递信息**。标准的线性变换对所有输入一视同仁，而门控机制赋予网络" 开关" 能力——根据上下文动态决定哪些信息应该通过、哪些应该被抑制。

7.1 为什么需要门控?

考虑标准的全连接层 $y = Wx + b$：无论输入 x 的内容如何，权重 W 始终相同。这种**静态映射**存在根本局限——网络无法根据输入内容调整自身行为。

门控机制引入了**数据依赖的动态性**：

$$y = g(x) \odot f(x) \tag{95}$$

其中 $g(x) \in [0, 1]^d$ 是门控信号，$f(x)$ 是候选输出。关键洞察是：g 本身依赖于输入 x，使得变换从静态的 f 变为动态的 $g \odot f$。

信息瓶颈视角 从信息论角度，门控实现了**自适应压缩**。设 $I(X; Y)$ 为输入输出的互信息：

- 无门控：$Y = f(X)$，互信息由 f 的容量决定
- 有门控：$Y = g(X) \odot f(X)$，当 $g \to 0$ 时可主动丢弃信息

这种" 主动遗忘" 能力对于过滤噪声、聚焦关键信息至关重要。在 LSTM 中，遗忘门正是通过这一机制解决了长程依赖问题 [**?**]。

稀疏激活视角 门控还天然诱导稀疏性。当 $g_i \approx 0$ 时，第 i 个维度被" 关闭"。这与 ReLU 的稀疏激活类似，但更灵活：ReLU 的稀疏模式由权重固定，而门控的稀疏模式随输入动态变化。实验表明，门控网络的激活稀疏度可达 60-80%，显著高于非门控网络 [**?**]。

7.2 MLP 层的门控：SwiGLU

如第 3节所述，现代 Transformer 普遍采用 SwiGLU 替代标准 FFN：

标准 FFN

$$\text{FFN}(x) = W_2 \cdot \text{GELU}(W_1 x) \tag{96}$$

SwiGLU FFN

$$\text{SwiGLU}(x) = W_2 \cdot \underbrace{(\text{SiLU}(W_1 x) \odot W_3 x)}_{\text{gating}} \tag{97}$$

这里 $W_3 x$ 作为门控信号，控制 $\text{SiLU}(W_1 x)$ 的信息流通。为保持参数量一致，SwiGLU 将中间维度从 $F = 4D$ 调整为 $F = \frac{8}{3}D$。

7.3 Attention 层的门控：Gated Attention

?]提出在 Scaled Dot-Product Attention（SDPA）输出后添加一个 sigmoid 门控：

标准 SDPA

$$Y = \text{softmax}\left(\frac{QK^\top}{\sqrt{H}}\right) V \tag{98}$$

Gated Attention

$$Y' = Y \odot \sigma(XW_g) \tag{99}$$

其中 σ 是 sigmoid 函数，$W_g \in \mathbb{R}^{D\times D}$ 是可学习的门控投影矩阵。

表 23: 门控位置与激活函数的消融实验

Configuration	Effectiveness	Notes
Gate on Values	✓	有效但非最优
Gate on Keys	✓	有效但非最优
Gate on Dense output	✓	有效但非最优
Gate on SDPA output	✓✓	**最优位置**
Sigmoid activation	✓✓	**最优激活**
SiLU activation	✓	效果略差
Additive gating	×	显著差于乘法门控

为什么在 SDPA 输出后门控最优? 表 23的消融实验揭示了一个有趣现象：门控位置对效果影响显著。理解这一点需要分析 attention 的信息流：

1. **Gate on Values**：$Y = \text{softmax}(QK^\top/\sqrt{H}) \cdot (V \odot g)$
   问题：门控作用于 V 后，softmax 仍强制将权重分配到所有位置。即使某个 value 被门控为零，对应的注意力权重仍会被计算和分配。
2. **Gate on SDPA output**：$Y' = [\text{softmax}(QK^\top/\sqrt{H}) \cdot V] \odot g$
   优势：门控作用于最终输出，可以**整体抑制**整个 attention head 的贡献。这打破了 softmax 的强制分配约束——即使 softmax 必须分配权重，门控可以让整个输出趋近于零。

本质上，SDPA 输出门控实现了 **head 级别的动态剪枝**：网络可以学会在某些上下文中" 关闭" 特定的 attention head，这是 values 门控无法实现的。

7.4 Attention Sink：一个深层的结构性问题

Attention Sink 是理解注意力机制局限性的关键窗口。MIT HAN Lab 在 StreamingLLM [**?**] 中首次系统研究了这一现象，随后 OpenAI、小米、月之暗面等团队提出了各自的解决方案。

现象与发现

现象观察 在长序列任务中，研究者发现一个反直觉的现象：大量注意力权重集中在序列开头的少数 token（通常是第一个 token），即使这些 token 语义上并不重要（如标点符号或填充符）。更重要的是，**越深的层，这种现象越明显**。

StreamingLLM 的关键发现是：当使用滑动窗口注意力时，一旦初始 token 被移出窗口，模型输出会**完全崩溃**——产生无意义的乱码。但只需保留最初的 4 个 token，性能就能大幅恢复。

表面原因：Softmax 的概率约束 问题的直接原因在于 softmax 的归一化特性：

$$\sum_{j=1}^{T} \text{softmax}(q_i^\top k_j/\sqrt{H}) = 1 \tag{100}$$

这一约束意味着每个 query **必须**将全部注意力分配出去，即使理想情况下应该" 不关注任何 token"。网络的应对策略是学习一个" 垃圾桶" 位置来吸收多余的注意力。

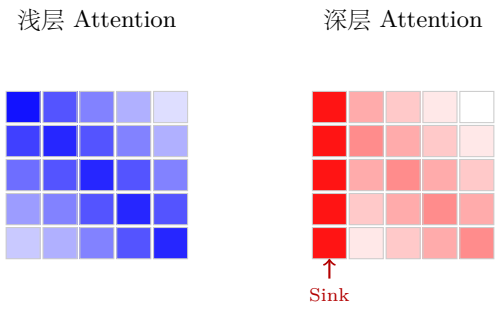

图 4: Attention Sink 现象示意图。浅层注意力主要沿对角线分布（关注局部上下文），深层则在第一列出现显著的注意力聚集（Sink）——所有 query 都倾向于 attend 到第一个 token，即使该 token 语义上并不重要。

深层原因：Context-Aware Identity Layer 假设

仅用 softmax 约束无法解释为什么 sink token 的 value 接近零、为什么越深层 sink 越严重。一个更深刻的假设是：

> *Attention Sink 源于 Transformer 对"上下文感知的恒等层"的内在需求*——模型需要能够根据上下文决定某个 *Attention Block* 不输出任何变化。

证据一：Sink Token 的 Value 接近零 实证研究表明，第一个 token 的 value 向量范数显著小于其他 token。这意味着即使被 attend 到，对输出的贡献也很小。模型**主动学习**将 sink token 的 value 置零。

证据二：Early Decoding 与层深相关性 如果用语言模型头探测中间层的输出，会发现模型存在"early decoding" 现象——从某一层开始，预测结果就与最终层相同。这说明浅层已经完成了主要计算，深层需要做的是**保持恒等变换**。
而实现恒等变换的方式就是让 Attention 输出零。由于 softmax 约束，模型必须通过 attend 到一个 value 为零的特殊 token 来实现这一点。

证据三：Sink Token 的 Key 具有独立子空间 实验表明，当 sink 发生时，sink token 的 key 与其他所有 key 几乎正交或负相关，且模长显著更小。这说明模型为 k_{sink} 分配了一个专用子空间：当 query 想要输出零时，它会指向这个子空间，精确 attend 到 sink token。

解决方案谱系

理解了 Attention Sink 的本质后，各种解决方案可以归为两类：**显式提供 sink** 和**消除对 sink 的需求**。

方案一：保留初始 Token（StreamingLLM） 最简单的方法是在滑动窗口中永久保留前几个 token：

$$\text{Attention Range} = \{1, 2, \ldots, k_{\text{sink}}\} \cup \{t - w + 1, \ldots, t\} \tag{101}$$

StreamingLLM 保留 4 个初始 token，使模型能稳定处理 **400 万 +token** 的流式输入。

方案二：可学习的 Softmax Bias OpenAI 的 GPT-OSS 系列 [**?**] 和小米的 MiMo-V2-Flash 都采用了在 softmax 分母中引入可学习偏置的方案：

$$\text{Attention}_{ij} = \frac{\exp(q_i^\top k_j / \sqrt{d})}{\sum_k \exp(q_i^\top k_k / \sqrt{d}) + \exp(b_h)} \tag{102}$$

其中 b_h 是每个 attention head 的可学习标量。当 b_h 很大时，分母增大，所有注意力权重被" 稀释"，等效于模型可以选择" 不关注任何 token"。这种方法仅增加每个 head 一个参数，但从根本上解决了 softmax 的强制分

配问题。

MiMo-V2-Flash 进一步将此技术与激进的 128-token 滑动窗口结合，实现了 KV Cache 减少约 6 倍，同时通过 sink bias 保持长上下文稳定性。

方案三：Output Gating（Kimi Linear） 月之暗面的 Kimi Linear [**?**] 在线性注意力模块中采用 output gating 来缓解 sink：

$$Y' = Y \odot \text{gate}(X) \tag{103}$$

这使得 attention 可以输出零向量，无需依赖 sink token。其 Kimi Delta Attention (KDA) 模块将此机制与细粒度遗忘门结合，在保持线性复杂度的同时超越了全注意力性能。

方案四：SDPA Output Gating（Qwen） 如前所述，Qwen 的 Gated Attention 在 SDPA 输出后添加 sigmoid 门控，同样消除了对 sink token 的需求。

表 24: Attention Sink 解决方案对比

方案	额外参数	消除 Sink	代表模型
保留初始 Token	0	否（绕过）	StreamingLLM
Softmax Bias	n_h	是	GPT-OSS, MiMo-V2-Flash
Output Gating	D^2	是	Kimi Linear, Qwen

理论统一

这些方案可以从统一的视角理解：它们都在解决 **Attention 如何输出零**的问题。

- **StreamingLLM**：让模型继续使用学到的 sink 机制
- **Softmax Bias**：修改 softmax 使其可以输出接近零的分布
- **Output Gating**：在 softmax 之后乘以可学习门控，强制输出为零

值得注意的是，output gating 不仅消除了 sink，还**释放了被 sink 占用的维度**。模型不再需要为 k_{sink} 分配专用子空间，这些容量可以用于更有意义的表示学习。这可能是 Gated Attention 带来性能提升的深层原因之一。

Remark 7.1 (非归一化注意力)**.** *理论上，如果注意力分数不需要归一化（或可以为负），模型可以直接组合出零输出，不需要 sink。实验表明非归一化注意力确实消除了 sink 现象，但这会带来训练稳定性问题。Softmax Bias 和 Output Gating 是在保持训练稳定性的前提下解决 sink 的折中方案。*

7.5 训练稳定性与长上下文

Gated Attention 还带来额外的训练优势：

训练稳定性

- 损失曲线更平滑，减少 loss spike
- 可以使用更大的学习率（4.0×10^{-3} 到 4.5×10^{-3}）

长上下文外推 结合 YaRN 位置编码插值：

- 训练长度：32k tokens
- 外推测试：128k tokens
- 性能衰减显著小于基线模型

7.6 工业应用

Gated Attention 已被集成到 Qwen3-Next 架构中，并部署于 Qwen3-Next-80B-A3B-Instruct 模型，验证了其在大规模工业应用中的有效性。该工作入选 NeurIPS 2025 Oral（前 1.5%）。

Remark 7.2 (MLP 门控 vs Attention 门控)**.** 两种门控的作用互补：

- ***MLP* 门控（*SwiGLU*）**：在特征变换阶段选择性激活神经元
- ***Attention* 门控**：在信息聚合阶段选择性传递 *attention* 输出

现代模型（如 *Qwen3-Next*）同时采用两者。

Part III

注意力机制

8 FlashAttention: IO 感知的高效注意力

第 2节介绍的 Roofline 模型揭示了一个关键洞察：现代 GPU 的算力远超内存带宽，决定性能的往往不是” 能算多快”，而是” 数据能多快送达”。标准注意力机制正是这一瓶颈的典型案例——它需要将 $O(N^2)$ 规模的中间结果反复写入显存，而这些 IO 操作成为真正的性能杀手。

FlashAttention 的核心思想是重新组织计算顺序，使注意力矩阵**永远不离开 GPU 的高速缓存**。这不是数学上的近似，而是精确等价的重新排列——我们用更多的计算换取更少的内存访问，而现代 GPU 恰好计算过剩、带宽稀缺。

8.1 动机：GPU 内存层次结构

内存瓶颈分析

现代 GPU 的计算能力远超内存带宽。以 NVIDIA A100 为例：

- **计算能力**：312 TFLOPS（FP16 Tensor Core）
- **HBM 带宽**：2 TB/s
- **SRAM 容量**：20 MB（共享内存 + L1 缓存）
- **SRAM 带宽**：约 19 TB/s

算术强度（Arithmetic Intensity）定义为每字节内存访问的 FLOPs：

$$\text{算术强度} = \frac{\text{FLOPs}}{\text{内存访问字节数}} \tag{104}$$

对于 A100，达到峰值计算需要算术强度 ≥ 156 FLOPs/Byte。标准注意力的算术强度远低于此，因此是**内存受限**（Memory-Bound）的。

标准注意力的 IO 问题

标准注意力实现需要多次 HBM 读写：

1. 从 HBM 读取 Q, K，计算 $S = QK^\top$，写回 HBM
2. 从 HBM 读取 S，计算 $P = \text{softmax}(S)$，写回 HBM
3. 从 HBM 读取 P, V，计算 $O = PV$，写回 HBM

总 HBM 访问量：$O(N^2 + Nd)$，其中 N 是序列长度，d 是头维度。对于长序列，N^2 项主导，造成严重的 IO 瓶颈。

8.2 FlashAttention v1

FlashAttention [**?**] 的核心思想是：**永远不将完整的 $N \times N$ 注意力矩阵写入 HBM**。

Online Softmax 算法

Softmax 的标准计算需要两次遍历：

$$m = \max_j(x_j) \tag{105}$$

$$\ell = \sum_j e^{x_j - m} \tag{106}$$

$$\text{softmax}(x)_i = \frac{e^{x_i - m}}{\ell} \tag{107}$$

Online Softmax 允许单次遍历、增量计算。对于两个块 $x^{(1)}, x^{(2)}$ 的拼接：

$$m^{(1)} = \max(x^{(1)}), \quad \ell^{(1)} = \sum_j e^{x_j^{(1)} - m^{(1)}} \tag{108}$$

$$m^{(2)} = \max(x^{(2)}), \quad \ell^{(2)} = \sum_j e^{x_j^{(2)} - m^{(2)}} \tag{109}$$

$$m^{new} = \max(m^{(1)}, m^{(2)}) \tag{110}$$

$$\ell^{new} = e^{m^{(1)} - m^{new}} \ell^{(1)} + e^{m^{(2)} - m^{new}} \ell^{(2)} \tag{111}$$

输出也可以增量更新：

$$O^{new} = \frac{1}{\ell^{new}} \left[e^{m^{old} - m^{new}} \ell^{old} \cdot O^{old} + e^{m^{(j)} - m^{new}} \ell^{(j)} \cdot O^{(j)} \right] \tag{112}$$

Tiling 算法

FlashAttention 将 Q, K, V 分成大小为 $B_r \times d$ 和 $B_c \times d$ 的块：

- $B_c = \lceil M/(4d) \rceil$（SRAM 大小 M 约束）
- $B_r = \min(B_c, d)$

Algorithm 1 FlashAttention 前向传播（简化版）

Require: $Q, K, V \in \mathbb{R}^{N \times d}$，块大小 B_r, B_c
初始化 $O = 0, \ell = 0, m = -\infty$（均为 N 维向量）
for $j = 1$ to $\lceil N/B_c \rceil$ **do**
  从 HBM 加载 $K_j, V_j \in \mathbb{R}^{B_c \times d}$ 到 SRAM
  for $i = 1$ to $\lceil N/B_r \rceil$ **do**
    从 HBM 加载 Q_i, O_i, ℓ_i, m_i 到 SRAM
    在 SRAM 中计算 $S_{ij} = Q_i K_j^\top \in \mathbb{R}^{B_r \times B_c}$
    计算 $m_{ij} = \text{rowmax}(S_{ij})$，$\tilde{P}_{ij} = \exp(S_{ij} - m_{ij})$
    计算 $\ell_{ij} = \text{rowsum}(\tilde{P}_{ij})$
    更新 m_i^{new}, ℓ_i^{new}（Online Softmax 更新）
    更新 $O_i = \text{diag}(\ell_i^{new})^{-1}(\text{diag}(\ell_i) e^{m_i - m_i^{new}} O_i + e^{m_{ij} - m_i^{new}} \tilde{P}_{ij} V_j)$
    将 $O_i, \ell_i^{new}, m_i^{new}$ 写回 HBM
  end for
end for
return O

反向传播与重计算

标准反向传播需要存储 $S, P \in \mathbb{R}^{N \times N}$。FlashAttention 采用**重计算**（Recomputation）策略：

- 前向传播只存储 O 和统计量 (m, ℓ)
- 反向传播时从 Q, K, V 块重新计算 S, P

虽然增加了 FLOPs（约多 30%），但大幅减少 HBM 访问，总体速度更快。

IO 复杂度分析

Theorem 8.1 (FlashAttention IO 复杂度)**.** *设 SRAM 大小为 M，序列长度为 N，头维度为 d。FlashAttention 的 HBM 访问量为：*

$$O\left(\frac{N^2 d^2}{M}\right) \tag{113}$$

而标准注意力的 HBM 访问量为 $O(N^2 + Nd)$。当 $M = \Theta(Nd)$ 时，FlashAttention 减少 $O(N)$ 倍 HBM 访问。

表 25: FlashAttention v1 性能提升

场景	加速比	内存节省
BERT-large (seq=512)	1.15×	5×
GPT-2 (seq=1K)	3×	10×
Long-range (seq=4K)	2.4×	20×

8.3 FlashAttention-2

FlashAttention-2 [**?**] 通过优化并行策略和工作分配，在 A100 上达到 230 TFLOPS（约 73% 峰值利用率），比 v1 快约 2 倍。

并行策略改进

FlashAttention v1：在 batch 和 head 维度并行，每个 thread block 处理一个 attention head。当 batch×heads < 108（A100 SM 数量）时，GPU 利用率低。

FlashAttention-2：额外在**序列长度维度**并行。对于长序列（通常意味着小 batch），这显著提高 GPU 利用率。

Warp 工作分配

GPU 的线程层次：Thread → Warp（32 线程）→ Thread Block → Grid。

v1 的 Sliced-K 方案：将 K, V 在 4 个 warp 间分割，Q 对所有 warp 可见。问题：需要 warp 间同步和共享内存中间结果。

v2 的 Sliced-Q 方案：将 Q 在 4 个 warp 间分割，K, V 对所有 warp 可见。优势：消除 warp 间通信，减少共享内存读写。

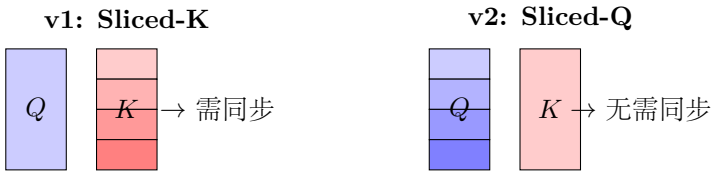

图 5: FlashAttention v1 vs v2 的 Warp 工作分配

减少非矩阵乘法 FLOPs

A100 的矩阵乘法吞吐量是非矩阵乘法的 **16 倍**（312 vs 19.5 TFLOPS）。v2 通过以下方式减少非 matmul 操作：

- 优化 Online Softmax 的 rescaling 操作
- 改进边界检查和因果 mask 的实现

表 26: FlashAttention v1 vs v2 性能对比

指标	v1	v2	提升
A100 峰值 (TFLOPS)	124	230	1.85×
GPU 利用率	25-40%	50-73%	约 2×
GPT-3 训练 (TFLOPS)	173	225	1.3×

8.4 FlashAttention-3

FlashAttention-3 [**?**] 针对 NVIDIA Hopper 架构（H100）设计，充分利用新硬件特性，达到 740 TFLOPS（75% 峰值利用率）。

Hopper 新硬件特性

WGMMA（Warpgroup Matrix Multiply-Accumulate） Hopper 引入的新 Tensor Core 指令，吞吐量显著高于 Ampere 的 `mma.sync`。一个 warpgroup（4 个 warp，128 线程）可以执行大规模矩阵乘法。

TMA（Tensor Memory Accelerator） 专用硬件单元，负责 Global Memory 和 Shared Memory 之间的数据传输：

- 自动处理索引计算和边界检查
- 释放寄存器资源，允许更大的 tile size
- 支持异步传输，与计算重叠

三大优化技术

1. Warp Specialization（Warp 专门化） 将 warp 分为 **Producer** 和 **Consumer**：

- Producer warp：负责 TMA 数据传输
- Consumer warp：负责 WGMMA 计算

数据传输和计算完全异步重叠。

2. Ping-Pong Scheduling 在两个 warpgroup 之间交替执行：

- Warpgroup 1 执行 GEMM 时，Warpgroup 2 执行 Softmax
- 然后角色互换

这种调度将 FP16 前向传播从约 570 TFLOPS 提升到 620 TFLOPS。

3. Intra-warpgroup Overlapping 在单个 warpgroup 内，Softmax 计算与 GEMM 流水线化：

- 当 GEMM 计算当前块时，同时对上一块执行 Softmax
- 进一步提升到 640-660 TFLOPS

FP8 支持

FlashAttention-3 支持 FP8 低精度，通过 **Block Quantization** 和 **Incoherent Processing**（基于 Hadamard 变换）减少量化误差：

- FP8 吞吐量：接近 1.2 PFLOPS
- 量化误差比基线 FP8 注意力低 2.6 倍

表 27: FlashAttention 版本性能演进

版本	GPU	峰值 TFLOPS	利用率	相对 v1 加速
v1	A100	124	40%	1×
v2	A100	230	73%	1.85×
v2	H100	335	35%	2.7×
v3	H100	740	75%	6×
v3 (FP8)	H100	1200	-	9.7×

8.5 FlashAttention-4

FlashAttention-4 [**?**] 针对 NVIDIA Blackwell 架构（B200）设计，是首个突破 PFLOPS 屏障的注意力内核。

五阶段 Warp 流水线

从 v3 的 2 阶段扩展到 **5 阶段流水线**，每种 warp 高度专门化：

1. **Load Warp**：通过 TMA 从 Global Memory 加载 Q, K, V 到 Shared Memory
2. **MMA Warp**：执行矩阵乘法，计算注意力分数和输出累加
3. **Softmax Warps**（8 个）：计算归一化注意力分数，维护 running statistics
4. **Correction Warps**（4 个）：当 scaling factor 变化时重新缩放输出
5. **Epilogue Warps**：将完成的输出块写回 Global Memory

软件 exp2 模拟

传统实现依赖 Special Function Units（SFU）计算指数函数，但 SFU 是稀缺资源。FlashAttention-4 使用**三次多项式近似**：

$$2^x \approx a_0 + a_1 x + a_2 x^2 + a_3 x^3, \quad x \in [0, 1) \tag{114}$$

通过 Horner 方法高效计算，在 CUDA Core 上使用向量化 FMA 指令，避免 SFU 瓶颈。

选择性重缩放

传统 Online Softmax 在每次遇到新最大值时都重新缩放。FlashAttention-4 引入**阈值判断**：

> 只有当最大值变化足以影响数值稳定性时才触发重缩放。

据报告，这将重缩放次数减少约 10 倍，同时保持数值精度。

性能

- 比 cuDNN 注意力快约 20%
- 比 FlashAttention-3 快约 2 倍
- 比原始 FlashAttention 快约 15 倍
- 目前仅支持 FP16/BF16 前向传播，反向传播开发中

8.6 Flash Decoding

FlashAttention 针对训练优化，但在**推理**时存在问题。Flash Decoding [**?**] 专门解决推理瓶颈。

推理时的问题

自回归生成时，每步只生成 1 个 token，即 Q 的序列长度为 1：

- FlashAttention 在 batch 和 head 维度并行
- 当 batch $\times$ heads < 108 时，GPU 严重 underutilized
- 长上下文场景下（batch size 小），**FlashAttention 可能只用到 GPU 的 1%**

KV 序列长度并行

Flash Decoding 的核心思想：在 **KV 序列长度**维度并行。

1. 将 KV Cache 分成 S 个块：$K = [K_1, ..., K_S]$, $V = [V_1, ..., V_S]$
2. 对每个块独立计算部分注意力：

$$O_s = \text{softmax}(QK_s^\top)V_s, \quad (m_s, \ell_s) = \text{统计量} \tag{115}$$

3. 使用 Log-Sum-Exp 合并结果：

$$O = \frac{\sum_s e^{m_s - m_{global}} \ell_s \cdot O_s}{\sum_s e^{m_s - m_{global}} \ell_s} \tag{116}$$

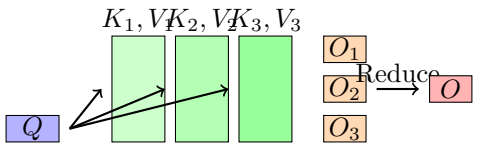

图 6: Flash Decoding：KV 序列并行 + Reduction

性能提升

- 长序列解码加速高达 **8 倍**
- 在 CodeLLaMa-34B 上，注意力操作比 FlashAttention 快 **50 倍**
- 序列长度从 512 增加到 64K，生成速度几乎不变

8.7 FlashDecoding++

FlashDecoding++ [**?**] 进一步优化，在 MLSys 2024 发表。

异步 Softmax

Flash Decoding 的 Reduction 步骤需要同步等待所有部分结果。FlashDecoding++ 引入 **Unified Max Value**：

- 预估一个全局最大值 $m_{unified}$（基于统计或启发式）
- 所有块使用相同的 $m_{unified}$，无需同步
- 细粒度流水线，Prefill 加速 1.05×，Decoding 加速 1.14×

Flat GEMM 优化

推理时的 GEMM 形状是“扁平”的（$1 \times N$），标准实现效率低：

- cuBLAS/CUTLASS 对这种形状有高达 50% 的性能损失
- FlashDecoding++ 使用 Double Buffering 和针对性优化
- Flat GEMM 加速高达 52%

启发式数据流

根据输入形状动态选择最优数据流：

- 不同序列长度、batch size 有不同瓶颈
- 启发式选择避免静态数据流的 50% 性能损失

表 28: Flash Decoding 系列性能对比

方法	相对 HuggingFace	相对 SOTA 引擎
FlashAttention（推理）	基线	-
Flash Decoding	8×（长序列）	-
FlashDecoding++	4.86×	1.37×

8.8 FlashAttention 的工程影响

FlashAttention 已成为现代 LLM 训练和推理的**标配**：

- **PyTorch 2.0+**：内置 `scaled_dot_product_attention` 使用 FlashAttention
- **vLLM、TensorRT-LLM**：推理引擎默认使用
- **所有主流 LLM**：GPT-4、Claude、LLaMA、DeepSeek 等都使用 FlashAttention

上下文长度革命 FlashAttention 将实用上下文长度从 2-4K 提升到 128K+：

- 内存从 $O(N^2)$ 降到 $O(N)$
- 64K 序列在标准注意力下需要 16GB 显存，FlashAttention 只需约 1GB

Remark 8.1 (何时使用 FlashAttention)**.** *FlashAttention 在以下场景收益最大：*

- **长序列**：*序列长度* > 512
- **大 *batch***：*充分利用 GPU 并行*
- **训练**：*内存节省允许更大 batch*

短序列、小模型场景下，标准注意力可能更快（减少 kernel launch 开销）。

8.9 FlexAttention：可编程的 FlashAttention

FlexAttention 是 PyTorch 2.5 引入的新 API，提供 FlashAttention 的灵活编程接口。

动机 FlashAttention 虽然高效，但每种注意力变体（Causal、ALiBi、Sliding Window 等）都需要专门实现。研究者想试验新变体时，往往需要手写 Triton kernel。FlexAttention 通过 `torch.compile` 自动生成高效 kernel，将开发时间从数周缩短到数分钟。

核心 API FlexAttention 提供两个函数式接口：

- **score_mod**：修改 $QK^\top$ 后的分数矩阵（如添加位置偏置）
- **mask_mod**：定义 mask 模式（返回 True 的位置参与计算）

```
from torch.nn.attention.flex_attention import flex_attention, create_block_mask

# Causal mask
def causal(b, h, q_idx, kv_idx):
    return q_idx >= kv_idx

# ALiBi位置编码
def alibi(score, b, h, q_idx, kv_idx):
    return score + (q_idx - kv_idx) * slope[h]

# 使用
block_mask = create_block_mask(causal, B, H, Q_LEN, KV_LEN)
out = flex_attention(q, k, v, score_mod=alibi, block_mask=block_mask)
```

性能 FlexAttention 达到 FlashAttention-2 约 **85-90%** 的性能，但开发效率提升 100 倍。对于 FlashAttention 不原生支持的变体（如 Document Masking），FlexAttention 比标准 SDPA 快 5-8 倍。

适用场景

- 快速原型验证新的注意力变体
- 复杂 mask 模式（document masking、jagged tensors）
- 生产环境如果用标准变体，仍推荐直接使用 FlashAttention

9 Multi-head Latent Attention (MLA)

FlashAttention 解决了训练时注意力矩阵的 IO 瓶颈，但推理时还存在另一个内存挑战：**KV Cache**。自回归生成需要缓存所有历史 token 的 Key 和 Value 向量，随着上下文长度增加，这部分内存占用可能超过模型参数本身。

Multi-head Latent Attention（MLA）是 DeepSeek-V2 [**?**] 针对这一问题提出的解决方案。其核心洞察是：虽然每个注意力头需要独立的 K 和 V，但它们可能存在**低秩结构**——即可以从一个共享的低维" 潜在向量" 中恢复。这种压缩不同于 GQA/MQA 的强制共享，而是让网络学习最优的压缩方式。

9.1 背景与目标

KV Cache 的挑战

如第 3节所述，自回归生成时需要缓存历史的 K 和 V：

$$\text{KV Cache Size} = 2 \times B \times S \times L \times n_h \times d_h \times \text{bytes} \tag{117}$$

对于大模型（如 $n_h = 128$，$d_h = 128$），KV Cache 成为长上下文推理的主要内存瓶颈。

现有方案的局限

表 29: KV Cache 优化方案对比

Method	KV Cache Size	Performance	原理
MHA	$2n_h d_h$	最优	每头独立 KV
GQA	$2\frac{n_h}{g} d_h$	轻微下降	g 个 Q 头共享 KV
MQA	$2d_h$	明显下降	所有头共享 KV
MLA	$d_c + d_h^R$	**接近 MHA**	低秩压缩

GQA/MQA 通过共享 KV 头来减少缓存，但这种强制共享往往损害模型性能。MLA 的核心洞察是：**KV 可以从一个低维潜在向量中恢复**，而不必显式共享。

9.2 MLA 核心原理

低秩压缩思想

MLA 的核心思想是将高维的 Key 和 Value 压缩到一个共享的低维潜在向量（latent vector），推理时从该向量恢复 K 和 V。

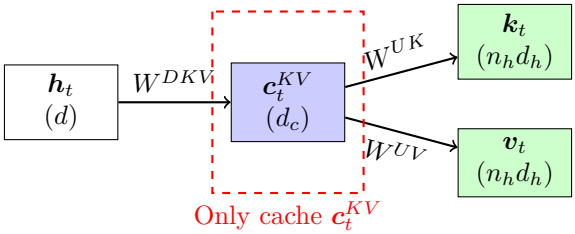

图 7: MLA 的 KV 压缩与恢复。只需缓存低维的 $\boldsymbol{c}_t^{KV}$，K 和 V 在计算时动态恢复。

9.3 数学公式

KV 的低秩压缩

对于输入 $\boldsymbol{h}_t \in \mathbb{R}^d$，首先压缩到潜在向量：

$$\boldsymbol{c}_t^{KV} = W^{DKV}\boldsymbol{h}_t \tag{118}$$

其中 $W^{DKV} \in \mathbb{R}^{d_c \times d}$ 是下投影矩阵，$d_c \ll n_h d_h$ 是压缩维度。
从潜在向量恢复 K 和 V：

$$\boldsymbol{k}_t^C = W^{UK}\boldsymbol{c}_t^{KV} \tag{119}$$
$$\boldsymbol{v}_t^C = W^{UV}\boldsymbol{c}_t^{KV} \tag{120}$$

其中 $W^{UK}, W^{UV} \in \mathbb{R}^{n_h d_h \times d_c}$ 是上投影矩阵。

Query 的低秩压缩

类似地，Query 也可以进行低秩压缩（主要用于减少训练时的激活内存）：

$$\boldsymbol{c}_t^Q = W^{DQ}\boldsymbol{h}_t \tag{121}$$
$$\boldsymbol{q}_t^C = W^{UQ}\boldsymbol{c}_t^Q \tag{122}$$

其中 $W^{DQ} \in \mathbb{R}^{d_c' \times d}$，$W^{UQ} \in \mathbb{R}^{n_h d_h \times d_c'}$。

Decoupled RoPE

RoPE 需要在每个位置应用旋转，但如果直接对压缩后的 $\boldsymbol{c}_t^{KV}$ 应用 RoPE，会破坏后续的权重吸收优化。MLA 采用**解耦 RoPE**（Decoupled RoPE）策略：

1. 将每个注意力头分为两部分：
   - **内容部分**（d_h^C 维）：从压缩向量恢复，**不**应用 RoPE
   - **位置部分**（d_h^R 维）：额外投影，应用 RoPE
2. 最终的 Q 和 K 为两部分的拼接：

$$\boldsymbol{q}_t = [\boldsymbol{q}_t^C; \mathrm{RoPE}(\boldsymbol{q}_t^R, t)] \tag{123}$$
$$\boldsymbol{k}_t = [\boldsymbol{k}_t^C; \mathrm{RoPE}(\boldsymbol{k}_t^R, t)] \tag{124}$$

其中位置部分的计算为：

$$\boldsymbol{q}_t^R = W^{QR}\boldsymbol{c}_t^Q \tag{125}$$
$$\boldsymbol{k}_t^R = W^{KR}\boldsymbol{h}_t \tag{126}$$

Remark 9.1 (为什么需要 Decoupled RoPE)**.** *RoPE 是位置相关的：RoPE*$(\boldsymbol{x}, t)$ *依赖于位置* t*。如果对* $\boldsymbol{c}_t^{KV}$ *应用 RoPE 再恢复* K*，则：*

$$\boldsymbol{k}_t = W^{UK} \cdot \mathit{RoPE}(\boldsymbol{c}_t^{KV}, t)$$

此时 W^{UK} *无法被吸收到* W^Q *中（因为 RoPE 在中间）。Decoupled 策略将 RoPE 隔离到单独的维度，保留了权重吸收的可能性。*

权重吸收

MLA 的一个关键优化是**权重吸收**（Weight Absorption）。由于压缩和恢复之间没有非线性激活，矩阵可以合并：

Query-Key 吸收 注意力分数的计算：

$$\boldsymbol{q}_t^{C\top}\boldsymbol{k}_s^C = (\boldsymbol{c}_t^Q)^\top (W^{UQ})^\top W^{UK}\boldsymbol{c}_s^{KV} \tag{127}$$

$$= (\boldsymbol{c}_t^Q)^\top \underbrace{W^{QK}}_{\text{absorbed}} \boldsymbol{c}_s^{KV} \tag{128}$$

其中 $W^{QK} = (W^{UQ})^\top W^{UK} \in \mathbb{R}^{d_c' \times d_c}$。

Output-Value 吸收 输出投影的计算：

$$W^O\boldsymbol{v}_t^C = W^O W^{UV}\boldsymbol{c}_t^{KV} \tag{129}$$

$$= \underbrace{W^{OV}}_{\text{absorbed}} \boldsymbol{c}_t^{KV} \tag{130}$$

其中 $W^{OV} = W^O W^{UV} \in \mathbb{R}^{d \times d_c}$。

推理流程 权重吸收后，推理时：

1. 缓存 $\boldsymbol{c}_t^{KV}$ 和 $\boldsymbol{k}_t^R$（位置部分）
2. 用 W^{QK} 直接计算内容部分的注意力分数
3. 用吸收后的 W^{OV} 计算输出

9.4 KV Cache 分析

表 30: MLA 的 KV Cache 对比（per token per layer）

Method	Cache Elements	DeepSeek-V2 (具体值)
MHA	$2n_h d_h$	$2 \times 128 \times 128 = 32768$
GQA (8 组)	$2 \times 8 \times d_h$	$2 \times 8 \times 128 = 2048$
MLA	$d_c + d_h^R$	$512 + 64 = 576$

DeepSeek-V2 的配置：$d_c = 512$，$d_h^R = 64$，压缩比达到 $\frac{32768}{576} \approx \mathbf{56.9}\times$。

9.5 PyTorch 实现

以下是 MLA 的简化 PyTorch 实现：

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class MultiHeadLatentAttention(nn.Module):
    def __init__(
        self,
        d_model: int,      # 模型维度
        n_heads: int,      # 注意力头数
        d_c: int,          # KV压缩维度
        d_c_q: int,        # Q压缩维度
        d_head_r: int,     # RoPE维度/头
    ):
        super().__init__()
        self.n_heads = n_heads
        self.d_head = d_model // n_heads
        self.d_c = d_c
        self.d_head_r = d_head_r
```

```
        # KV compression
        self.W_dkv = nn.Linear(d_model, d_c, bias=False)
        self.W_uk = nn.Linear(d_c, d_model, bias=False)
        self.W_uv = nn.Linear(d_c, d_model, bias=False)

        # Q compression
        self.W_dq = nn.Linear(d_model, d_c_q, bias=False)
        self.W_uq = nn.Linear(d_c_q, d_model, bias=False)

        # Decoupled RoPE projections
        self.W_qr = nn.Linear(d_c_q, n_heads * d_head_r, bias=False)
        self.W_kr = nn.Linear(d_model, n_heads * d_head_r, bias=False)

        # Output projection
        self.W_o = nn.Linear(d_model, d_model, bias=False)

    def forward(self, x, rope_fn, kv_cache=None):
        B, T, D = x.shape

        # KV compression: only c_kv needs caching
        c_kv = self.W_dkv(x)  # (B, T, d_c)

        # Decoupled RoPE keys
        k_r = self.W_kr(x)    # (B, T, n_heads * d_head_r)
        k_r = rope_fn(k_r)    # Apply RoPE

        # Handle KV cache
        if kv_cache is not None:
            c_kv_cached, k_r_cached = kv_cache
            c_kv = torch.cat([c_kv_cached, c_kv], dim=1)
            k_r = torch.cat([k_r_cached, k_r], dim=1)
        new_cache = (c_kv, k_r)

        # Q compression
        c_q = self.W_dq(x)    # (B, T, d_c_q)
        q_c = self.W_uq(c_q)  # (B, T, D) - content part
        q_r = self.W_qr(c_q)  # (B, T, n_heads * d_head_r) - RoPE part
        q_r = rope_fn(q_r)

        # Reconstruct K, V from compressed cache
        k_c = self.W_uk(c_kv)  # (B, S, D) - content part
        v = self.W_uv(c_kv)    # (B, S, D)

        # Reshape for multi-head attention
        q_c = q_c.view(B, T, self.n_heads, self.d_head)
        k_c = k_c.view(B, -1, self.n_heads, self.d_head)
        v = v.view(B, -1, self.n_heads, self.d_head)
        q_r = q_r.view(B, T, self.n_heads, self.d_head_r)
        k_r = k_r.view(B, -1, self.n_heads, self.d_head_r)

        # Concatenate content and RoPE parts
        q = torch.cat([q_c, q_r], dim=-1)  # (B, T, H, d_head + d_head_r)
        k = torch.cat([k_c, k_r], dim=-1)  # (B, S, H, d_head + d_head_r)

        # Scaled dot-product attention
        q, k, v = q.transpose(1,2), k.transpose(1,2), v.transpose(1,2)
        scale = (self.d_head + self.d_head_r) ** -0.5
        attn = F.softmax(q @ k.transpose(-2,-1) * scale, dim=-1)
```

```
        out = attn @ v  # (B, H, T, d_head)

        # Output projection
        out = out.transpose(1,2).reshape(B, T, D)
        out = self.W_o(out)

        return out, new_cache
```

Listing 1: MLA 核心实现

Remark 9.2 (实现优化). 上述代码为教学版本。实际部署时:

- 权重吸收: 预计算 $W^{QK} = (W^{UQ})^\top W^{UK}$ 和 $W^{OV} = W^O W^{UV}$
- *Flash Attention*: 使用 *FlashAttention* 加速注意力计算
- 融合算子: 将多个小矩阵乘法融合

9.6 MLA vs 其他方法

表 31: 注意力机制全面对比

Feature	MHA	GQA	MQA	MLA
KV Cache	$2n_h d_h$	$2\frac{n_h}{g} d_h$	$2d_h$	$d_c + d_h^R$
参数量	基准	减少	最少	略增
表达能力	最强	较强	较弱	接近 MHA
推理延迟	高	中	低	中
长上下文	受限	较好	好	**最好**

MLA 的优势

- **极致压缩**: KV Cache 减少 93% 以上
- **性能保持**: 不像 GQA/MQA 强制共享，而是学习最优压缩
- **长上下文友好**: 128K 上下文成为可能

MLA 的代价

- **计算开销**: 需要额外的压缩/恢复计算
- **实现复杂度**: Decoupled RoPE 和权重吸收增加实现难度
- **训练成本**: 低秩约束可能需要更多训练

9.7 质量-效率权衡: 何时使用 MLA?

MLA 并非银弹。其 56.9 倍的压缩比来自于一个强假设: **K 和 V 可以从 512 维潜在空间无损恢复**。这个假设在什么条件下成立?

低秩假设的有效性

考虑注意力模式的多样性需求。设任务需要 r 种本质不同的注意力模式（如: 局部依赖、长程依赖、语法结构、语义关联等），则 KV 表示至少需要 r 个自由度。

压缩维度的下界 若 $d_c < r$，则压缩会造成信息丢失。DeepSeek 选择 $d_c = 512$，隐含假设是: 常见 NLP 任务的注意力模式可以用不超过 512 维的空间表达。

任务依赖性 不同任务对注意力多样性的需求不同：

- **文本生成**：模式相对固定，低秩假设成立，MLA 表现良好
- **代码理解**：需要追踪复杂的变量依赖和作用域，可能需要更高秩的表示
- **数学推理**：多步推理需要维持多条推理链，对注意力多样性要求高

选择指南

表 32: 注意力机制选择指南

场景	推荐方法	理由
长上下文推理 (>32K)	MLA	KV Cache 是主要瓶颈
短上下文、高吞吐	GQA	实现简单，开销低
质量优先（小 batch）	MHA	无压缩损失
代码/数学任务	GQA 或 MHA	注意力多样性需求高
边缘设备部署	MLA	极致内存压缩

经验法则 当 上下文长度 $\times$ batch size $> 10^6$ 时，KV Cache 成为主要瓶颈，MLA 的收益开始显现。对于短上下文或小 batch 推理，GQA 的简单性可能更具优势。

压缩比与质量的帕累托边界

DeepSeek 的消融实验表明，d_c 的选择存在" 甜点区"：

- $d_c < 256$：质量显著下降，压缩过度
- $d_c \in [256, 768]$：质量接近 MHA，压缩有效
- $d_c > 768$：质量无进一步提升，压缩收益递减

$d_c = 512$ 处于帕累托最优附近——进一步压缩会损害质量，进一步扩大则浪费缓存空间。

9.8 应用与扩展

MLA 已在以下模型中应用：

- **DeepSeek-V2**：首次提出，236B 参数 MoE 模型
- **DeepSeek-V3**：进一步优化，671B 参数
- **DeepSeek-R1**：推理模型，继承 MLA 架构

Remark 9.3 (MLA 与 MoE 的协同)**.** *MLA 特别适合 MoE 架构：MoE 的稀疏激活已经减少了计算量，而 MLA 进一步解决了内存瓶颈。两者结合使得 DeepSeek-V2 在保持高性能的同时，训练成本降低 42.5%。*

10 稀疏注意力

前两节介绍的 FlashAttention 和 MLA 分别从计算效率和内存占用角度优化了注意力机制，但它们都保留了完整的 $O(N^2)$ 注意力计算——只是让这个计算更快、更省内存。本节探索另一条路径：如果大多数注意力权重本就接近零，我们能否**跳过这些无意义的计算**?

稀疏注意力的核心思想是：只计算" 重要" 的 token 对，将 $O(N^2)$ 降至 $O(Nk)$，其中 $k \ll N$。与后面将介绍的线性注意力不同，稀疏注意力保留了精确的 softmax 计算，只是在更小的范围内进行。

10.1 稀疏注意力概述

为什么需要稀疏注意力

标准 Softmax Attention 的复杂度为 $O(N^2)$，但实际上并非所有 token 对都同等重要:

- 注意力分布通常是稀疏的（少数 token 获得大部分权重）
- 远距离 token 的注意力通常较弱
- 语义相关的 token 往往聚集在特定位置

稀疏注意力的核心思想：**只计算重要的注意力对**，将复杂度从 $O(N^2)$ 降至 $O(Nk)$，其中 $k \ll N$。

稀疏注意力 vs 线性注意力

表 33: 稀疏注意力与线性注意力对比

特性	稀疏注意力	线性注意力
复杂度	$O(Nk)$	$O(Nd^2)$
注意力类型	精确 Softmax	近似/替代 Softmax
长程精确检索	强	弱
KV Cache	需要完整	可压缩
与原始 Transformer 兼容	高	中

10.2 滑动窗口注意力

滑动窗口注意力（Sliding Window Attention, SWA）是最直观的稀疏注意力形式：每个 token 只关注其周围固定窗口内的 token。

Mistral 的滑动窗口 [?]

Mistral 7B 是首个将滑动窗口注意力规模化部署的开源模型，窗口大小为 4096。

核心机制 每个位置 t 的 token 只关注 $[t-w, t]$ 范围内的 token:

$$\text{Attention}_t = \text{softmax}\left(\frac{q_t K_{[t-w:t]}^\top}{\sqrt{d}}\right) V_{[t-w:t]} \tag{131}$$

层间信息传递 滑动窗口的关键洞察是：通过 Transformer 的堆叠层，信息可以 "跨窗口" 传播。在第 k 层，位置 t 的 token 实际上可以访问到 $[t - k \cdot w, t]$ 范围的信息。对于 32 层模型、窗口大小 4096，理论感受野可达 128K。

推理优化

- **滚动缓存**（Rolling Buffer）：KV Cache 只需保留最近 w 个 token
- **内存节省**：8K 序列长度下节省 50% 缓存
- **速度提升**：配合 FlashAttention 和 xFormers，16K 序列上获得 2 倍加速

Longformer 与 BigBird

Longformer [**?**] 和 BigBird [**?**] 是早期将稀疏注意力系统化的工作。

Longformer 结合三种注意力模式：

1. **滑动窗口**：局部上下文建模
2. **膨胀滑动窗口**（Dilated）：扩大感受野
3. **全局注意力**：特定 token（如 [CLS]）关注全局

BigBird 在 Longformer 基础上增加**随机注意力**：

$$A = A_{\text{sliding}} + A_{\text{global}} + A_{\text{random}} \tag{132}$$

理论证明 BigBird 是完整注意力的通用近似器，支持最长 8 倍于 BERT 的序列。

StreamingLLM [?]

StreamingLLM 解决了一个重要问题：如何让 LLM 处理 “无限长” 的流式输入。

Attention Sink 现象 研究发现，无论输入多长，模型总是对**最开头的几个 token** 分配异常高的注意力权重——即使这些 token 在语义上并不重要。这被称为 “注意力汇聚”（Attention Sink）。

SinkAttention StreamingLLM 提出保留两部分 KV Cache：

- **Sink Tokens**：序列开头的 4 个 token（固定）
- **滑动窗口**：最近的 w 个 token

$$\text{KV Cache} = \text{Sink}_{[1:4]} \cup \text{Window}_{[t-w:t]} \tag{133}$$

效果：在 4M+ token 的流式场景下保持稳定性能，而普通滑动窗口在超过预训练长度后崩溃。

10.3 KV Cache 稀疏化

KV Cache 稀疏化是推理时的稀疏注意力：动态丢弃 “不重要” 的 KV 条目。

H2O [?]

Heavy-Hitter Oracle（H2O）基于一个观察：少数 “重击手”token 累积了大部分注意力权重。

算法

1. 维护每个 token 的累积注意力分数
2. 保留分数最高的 Top-k token（Heavy Hitters）
3. 结合最近的滑动窗口 token

层级策略 不同层使用不同的保留策略，高层（更稀疏的注意力分布）可以更激进地压缩。

SnapKV [?]

SnapKV 的核心洞察是：注意力模式在 prefill 阶段基本确定，可以 “一次性” 剪枝。

方法

1. 在 prefill 阶段末尾，分析最后一个窗口的注意力分布

2. 识别整个上下文中的重要位置
3. 永久保留这些位置的 KV，丢弃其余

优势：只需一次剪枝决策，无需每步动态更新。
局限：无法适应 decoding 阶段的动态变化。

PyramidKV

PyramidKV 发现不同层的注意力稀疏度不同：

- 低层：注意力较分散，需要更多 KV
- 高层：注意力集中，可大幅压缩

因此采用**金字塔形**的 KV 分配：底层多、高层少。

表 34: KV Cache 稀疏化方法对比

方法	剪枝时机	动态性	集成框架
H2O	每步	动态	vLLM
SnapKV	Prefill 后	静态	vLLM
StreamingLLM	持续	静态	–
PyramidKV	层级	静态	–

10.4 MoBA：块稀疏注意力

MoBA（Mixture of Block Attention）[**?**] 是块级稀疏注意力的代表性工作，已部署于 Kimi 的长上下文服务。

核心思想：将 MoE 应用于 Attention

MoBA 的核心洞察是：**并非所有上下文对当前 token 都同等重要**。与其对整个序列计算注意力，不如让模型自主学习 “关注哪些块”。

标准注意力

$$\mathrm{Attn}(q, K, V) = \mathrm{softmax}(qK^\top)V \tag{134}$$

MoBA

$$\mathrm{MoBA}(q, K, V) = \mathrm{softmax}(qK_{[\mathcal{I}]}^\top)V_{[\mathcal{I}]} \tag{135}$$

其中 $\mathcal{I} \subseteq [N]$ 是被选中的 KV 子集，由路由机制决定。

块划分与路由机制

块划分 将长度为 N 的上下文均匀划分为 n 个块，每块大小 $B = N/n$：

$$\mathcal{I}_i = [(i-1) \cdot B + 1, i \cdot B], \quad i = 1, \ldots, n \tag{136}$$

路由分数计算 对每个 query q，计算其与各块的亲和度分数：

$$s_i = \langle q, \mathrm{mean\_pool}(K_{[\mathcal{I}_i]}) \rangle \tag{137}$$

即 query 与块内所有 key 的平均向量的内积。这是一个**无参数**的路由机制。

Top-k 选择 选择分数最高的 k 个块进行注意力计算（典型设置 $k = 3$）：

$$g_i = \begin{cases} 1 & \text{if } s_i \in \text{Top-}k(\{s_j\}_{j=1}^{n}) \\ 0 & \text{otherwise} \end{cases} \tag{138}$$

因果性保证

在自回归场景下：

1. **未来块屏蔽**：对于位置 t 的 query，所有 $i > \lceil t/B \rceil$ 的块设 $s_i = -\infty$
2. **当前块强制选中**：query 所在的块始终被路由

性能表现

表 35: MoBA 性能

指标	MoBA	Full Attention
LM Loss 差异	$< 10^{-3}$	
稀疏度 @32K	95.31%	
加速比 @1M	6.5×	
加速比 @10M	16×	

10.5 NSA：原生稀疏注意力

Native Sparse Attention（NSA） [**?**] 是 DeepSeek 提出的层级稀疏注意力机制，已部署于 DeepSeek-V3.2。

三条注意力路径

NSA 将注意力计算分解为三条并行路径：

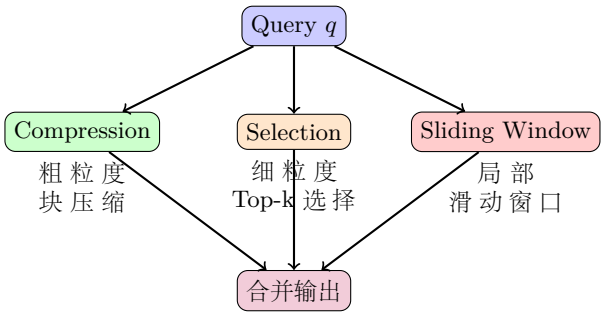

图 8: NSA 的三条注意力路径

1. Compression Attention（压缩注意力） 使用可学习的 MLP 将连续 token 压缩为块级表示：

$$\tilde{K}_i = \text{MLP}(K_{[(i-1)l+1:il]}), \quad \tilde{V}_i = \text{MLP}(V_{[(i-1)l+1:il]}) \tag{139}$$

其中 l 是压缩块大小（NSA 中 $l = 32$）。这捕获**全局粗粒度**信息。

2. Selection Attention（选择注意力） 通过 Lightning Indexer 选择最相关的块保持原始精度：

1. 计算 query 与所有块的相关性分数
2. 选择 Top-n 个块（NSA 中 $n = 16$ 个块，块大小 $l' = 64$）
3. 对选中块进行精确 Softmax 注意力

这保留**细粒度精确**信息。

3. Sliding Window Attention（滑动窗口注意力） 对最近的 w 个 token 进行完整注意力（NSA 中 $w = 512$）：

$$O_{sw} = \text{Attention}(q, K_{[t-w:t]}, V_{[t-w:t]}) \tag{140}$$

这保证**局部上下文**的精确建模。

Lightning Indexer

Lightning Indexer 是 NSA 的核心创新，用于高效选择相关块：

- 维护独立的 **FP8 量化** Key 缓存（非 MLA 的 KV Cache）
- 每个 query 计算与所有块的相关性分数
- 选择 Top-k 块（默认 2048 个 token）
- 硬件优化：DeepGEMM 实现的高效 CUDA kernel

关键设计：索引计算与注意力计算分离，索引使用低精度快速完成。

端到端可训练

与 ClusterKV、MagicPIG 等依赖不可微操作的方法不同，NSA 是**原生可训练**的——从预训练阶段就使用稀疏注意力。

可微的稀疏机制 NSA 的三个分支都支持梯度回传：

1. **Token Compression**：可学习的 MLP ϕ 将块级 KV 压缩为单个表示，带块内位置编码
2. **Token Selection**：块重要性分数通过压缩 token 的注意力 softmax 计算，梯度可流过选择过程
3. **门控机制**：通过 MLP+Sigmoid 学习三个分支的权重

预训练配置 DeepSeek 使用 27B 参数模型（MoE 架构，3B 激活参数）验证 NSA：

- **模型规格**：30 层，hidden_dim=2560，GQA（4 组，64 头）
- **MoE 结构**：72 个路由专家 + 2 个共享专家，top-k=6
- **训练数据**：270B tokens，8K 序列长度
- **长上下文适配**：继续训练 + SFT 在 32K 序列上，使用 YaRN 位置编码

训练加速 NSA 在训练阶段也实现了显著加速（64K 序列长度，A100 GPU）：

- **前向传播**：9.0× 加速
- **反向传播**：6.0× 加速
- 加速比随序列长度增加：8K 时 4×，16K 时 6.4×，32K 时 9.1×，64K 时 11.6×

后训练：推理能力 通过从 DeepSeek-R1 蒸馏，使用 10B tokens 的 32K 长度数学推理轨迹进行 SFT。在 AIME 数学 benchmark 上，NSA-R 比 Full Attention-R 提升 +0.075，证明稀疏注意力不损害复杂推理能力。

硬件对齐设计

NSA 针对现代 GPU 进行了深度优化，使用自定义 Triton kernel：

Group-Centric Loading 利用 GQA 的特性，将同一组内所有 query head 及其共享的稀疏 KV 块索引一次性加载到 SRAM，避免重复内存访问。

Shared KV Fetching 顺序加载被选中的连续 KV 块，最小化内存传输次数。由于不同 query 块的内循环长度几乎相同，可使用 Triton 的 grid scheduler 简化优化。

算术强度平衡 通过组内共享消除冗余 KV 传输，在 GPU 流处理器间均衡计算负载，实现接近最优的算术强度。

NSA 参数配置

表 36: NSA 默认参数

参数	符号	值
压缩块大小	l	32
选择块大小	l'	64
选择块数量	n	16
滑动窗口大小	w	512
滑动步长	d	16

10.6 DSA：DeepSeek 稀疏注意力

DSA（DeepSeek Sparse Attention，2025 年 9 月）是 DeepSeek 在 V3.2 中部署的新一代稀疏注意力，与 NSA 有本质区别。DSA 摒弃了 NSA 复杂的三分支设计，采用更简洁的**细粒度 token 级检索**。

与 NSA 的核心区别

表 37: NSA vs DSA 设计对比

特性	NSA	DSA
选择粒度	块级（block）	Token 级
分支数量	3（压缩 + 选择 + 窗口）	1（直接选择）
重要度计算	可学习 MLP	可学习 w 权重
Attention 变种	GQA	MLA
验证模型	27B	671B

算法设计

DSA 的核心思想：每个 query 只需关注固定数量 k 个最相关的 token（$k = 2048$）。

重要度分数计算 DSA 引入可学习权重 w 计算 token 重要度：

$$\text{score}_i = w \cdot f(q, k_i) \tag{141}$$

这是一个折中方案——比 NSA 的 MLP 简单，但比 MoBA 的无参数 mean-pooling 更有表达力。

Top-k 检索 根据重要度分数选择 Top-k 个 token 进行精确注意力计算：

$$\mathcal{I} = \text{Top-}k(\{\text{score}_i\}_{i=1}^{N}), \quad |\mathcal{I}| = 2048 \tag{142}$$

复杂度分析 单 query 需访问固定 k 个 token，因此整体复杂度为 $O(Nk)$，是真正的**线性复杂度**。

训练挑战与解决

细粒度 token 级检索比块级更难训练。DSA 采用以下技巧：

- **Warm-up**：逐步增加稀疏度
- **知识蒸馏**：从 Full Attention 模型蒸馏
- **MLA 适配**：专门针对 Multi-Latent Attention 优化

工程实现

- **TileLang Kernel**：细粒度稀疏 +MLA 需要定制 kernel，TileLang 比 Triton 性能更优
- **vLLM/SGLang 集成**：Day-0 支持，使用 DeepGEMM 和 FlashMLA
- **Blackwell 优化**：与 NVIDIA 合作优化 B200

性能收益

- 长上下文 API 成本降低约 **50%**
- 64K 序列上实现显著加速
- 671B 模型上验证，质量几乎无损

10.7 NSA vs MoBA vs DSA：三种方法深度对比

2025 年出现了三种重要的 "学习式" 稀疏注意力方法。本节从多个维度进行详细对比。

算法设计对比

表 38: 三种方法算法设计对比

设计维度	NSA	MoBA	DSA
基本信息			
发布时间	2025.02	2025.02	2025.09
提出者	DeepSeek	Moonshot (Kimi)	DeepSeek
Attention 基础	GQA	GQA	MLA
稀疏策略			
选择粒度	块级	块级	Token 级
分支数量	3（压缩 + 选择 + 窗口）	1	1
路由机制	可学习 MLP	无参数 mean-pool	可学习 w
局部窗口	有（w=512）	当前块强制选中	无

超参数与复杂度对比

表 39: 超参数与复杂度对比

参数	NSA	MoBA	DSA
关键超参			
块大小	l=32, l'=64	L=4096	–
选择数量	n=16 块	k=12 块	k=2048 tokens
滑动窗口	w=512	–	–
单 query 访问 token 数			
@32K 上下文	∼2560	49152	2048
@128K 上下文	∼5120	49152	2048
是否随 N 增长	是（$O(N/L)$）	否（固定 kL）	否（固定 k）
复杂度（vs 原版 $O(N^2)$）			
理论复杂度	$O(N^2/L)$	$O(N \cdot kL)$	$O(Nk)$
实际系数	$O(N^2/32)$	$O(N \cdot 49152)$	$O(N \cdot 2048)$
@64K 稀疏度	∼97%	∼23%	∼97%

重要度计算机制

三种方法采用不同的 "检索" 策略来确定哪些 token 值得关注：

NSA：可学习 MLP 压缩

$$\tilde{k}_i = \text{MLP}(K_{[(i-1)l+1:il]}), \quad \text{score}_i = q \cdot \tilde{k}_i \tag{143}$$

使用一个额外的 MLP 将块内所有 key 压缩为单个表示，再计算与 query 的相似度。**优点**：表达力强。**缺点**：引入额外参数和计算。

MoBA：无参数 Mean-Pooling

$$\text{score}_i = q \cdot \text{mean}(K_{[(i-1)L+1:iL]}) \tag{144}$$

直接对块内 key 做平均，再与 query 内积。**优点**：无额外参数，简洁优雅。**缺点**：表达力受限，块内信息被“抹平”。

DSA：可学习权重 w

$$\text{score}_i = w \cdot f(q, k_i) \tag{145}$$

介于两者之间——有可学习参数，但比 MLP 简单。**优点**：平衡表达力与复杂度。**缺点**：具体形式未完全公开。

训练与工程对比

表 40: 训练与工程实现对比

维度	NSA	MoBA	DSA
训练			
训练方式	原生预训练	继续训练	原生预训练
验证模型规模	27B	8B	671B
训练数据量	270B tokens	–	–
训练序列长度	8K→32K	–	–
需要蒸馏	可选（R1 蒸馏）	否	是（关键）
需要 Warm-up	否	否	是（关键）
工程实现			
Kernel 实现	Triton	Triton	TileLang
与 MLA 兼容	否	否	是
推理框架支持	–	–	vLLM/SGLang
加速效果			
前向加速 @64K	9×	6.5×@1M	–
后向加速 @64K	6×	–	–
解码加速 @64K	11.6×	16×@10M	–

设计哲学差异

NSA：全面覆盖，层级融合 NSA 的设计理念是**不遗漏任何重要信息**：

- **压缩分支**：捕获全局粗粒度信息
- **选择分支**：保留细粒度精确信息
- **窗口分支**：确保局部上下文不丢失
- 三分支通过可学习门控融合

代价：超参较多（l, l', n, w, d, 门控参数），调优和迁移复杂。

MoBA：简洁优雅，MoE 思想 MoBA 的设计理念是**将 MoE 的路由思想应用于 Attention**：

- 把 KV Cache 视为“专家池”，每个块是一个“专家”
- 无参数路由，让注意力分数自然决定选择
- 当前块强制选中，保证局部信息

优势：设计最简洁，与现有架构兼容性好。很多人认为 MoBA 更有潜力成为下一代 Transformer 的基础组件。
疑问：块大小 4096 较大（约 1 页 PDF），是否会退化为 sliding window 等 trivial 模式?

DSA：激进稀疏，端到端优化 DSA 的设计理念是**极限稀疏 + 精心训练**：

- Token 级选择，每个 query 只看 2048 个 token（约 3%@64K）
- 与 MLA 深度集成，需要定制 TileLang kernel
- 训练难度大，依赖 warm-up 和蒸馏

优势：稀疏度最高，推理效率最好，在 671B 超大模型上验证。
挑战：Token 级检索是否会遗漏关键信息？小模型能否复现?

稀疏模式可视化理解

假设上下文长度 N=32K，三种方法的稀疏模式差异：

表 41: @32K 上下文的稀疏模式

特性	NSA	MoBA	DSA
块数量	1024 块（每块 32 token）	8 块（每块 4096 token）	32K 个 token
选中数量	16 块 + 窗口	12 块	2048 token
选中 token 数	∼2560	∼49152	2048
稀疏度	92%	0%（看全部）	94%

关键观察：在 32K 这个长度上，MoBA 实际上几乎没有稀疏（选中 12 块 ×4096=49152 > 32K）！MoBA 的稀疏优势在更长序列（如 128K+）才能体现。

适用场景建议

表 42: 适用场景建议

方法	适用场景
NSA	需要精确保留多尺度信息；可接受复杂超参调优；使用 GQA 架构
MoBA	追求简洁设计；希望无缝替换现有 Attention；序列长度 128K+
DSA	使用 MLA 架构；追求极致稀疏度；有足够资源做蒸馏训练；超大模型（100B+）

开放问题与未来方向

固定 k 是否合理? DSA 和 MoBA 都假设每个 query 只需看固定数量的 token，不随 N 增长。这个假设值得深思：

- **反对观点**：上下文很长时，固定 k 可能遗漏关键信息。Needle-in-haystack 任务可能受影响。
- **支持观点**：每个 token 的 “信息容量” 有限。人类阅读长文档时也不会逐字关注。
- **折中思路**：可能需要自适应 k，或结合全局压缩分支（如 NSA）。

稀疏度应该自适应吗? 目前三种方法的稀疏度都是人为设定的超参。未来方向可能包括：

- 不同层使用不同稀疏度（类似 PyramidKV 的思想）
- 不同 query 根据任务复杂度动态调整 k
- 通过强化学习或可微搜索自动确定稀疏策略

稀疏 vs 压缩：两条技术路线

- **稀疏路线**（NSA/MoBA/DSA）：动态 “检索” 重要 token，不改变 KV Cache 大小

- **压缩路线**（Qwen3-Next 线性注意力）：压缩 KV Cache 本身，信息更紧凑
- 两者可能结合：先压缩再稀疏，或用稀疏注意力指导压缩

Post-training 方法是否被“降维打击”? 原生稀疏注意力（NSA/DSA）的出现，对 H2O、SnapKV、Quest 等 post-training 稀疏方法形成挑战：

- Post-training 方法：在已训练好的模型上“事后”稀疏化
- 原生方法：从预训练开始就使用稀疏注意力
- 原生方法的优势：模型从一开始就学会“如何稀疏”，效果更好
- Post-training 方法的优势：可应用于任意已有模型，无需重新训练

10.8 Ring Attention 与上下文并行

当序列长度超过单 GPU 显存容量时，需要将注意力计算分布到多个设备上。

Ring Attention [?]

Ring Attention 将长序列分割到多个 GPU，通过环形通信实现分布式注意力计算。

算法流程

1. 将 Query、Key、Value 按序列维度分割到 P 个 GPU
2. 每个 GPU 持有本地 Query 块，计算与本地 KV 的注意力
3. KV 块在 GPU 之间环形传递，累积计算全局注意力
4. 使用 online softmax 避免数值溢出

通信隐藏 关键优化是**计算-通信重叠**：在计算当前 KV 块注意力的同时，异步传递下一个 KV 块。

LLaMA 3 的上下文并行

LLaMA 3 训练采用 All-Gather 方式的 Context Parallelism [?]：

- 先 All-Gather 收集所有 KV，再计算本地 Query 的注意力
- 为负载均衡，将序列分为 $2 \times \text{CP}$ 个块并 shuffle
- 支持 128K 上下文的高效训练

与 Ring Attention 对比

- Ring Attention：点对点通信，延迟可隐藏
- All-Gather：集合通信，实现更简单但延迟在关键路径

10.9 稀疏注意力方法全景

Remark 10.1 (稀疏注意力的演进)**.** *从 Longformer/BigBird 的“手工设计模式”到 MoBA/NSA/DSA 的“学习式稀疏”，稀疏注意力正经历范式转变。2025 年，稀疏注意力首次在 600B+ 规模模型上得到验证（DeepSeek-V3.2），标志着该技术从学术研究走向工业主流。*

10.10 与其他方法的关系

与 FlashAttention 的关系

稀疏注意力与 FlashAttention 是**正交**的优化：

- FlashAttention 优化 **IO 效率**（不改变计算量）

表 43: 稀疏注意力方法全景对比

方法	稀疏策略	复杂度	全局信息	部署
Sliding Window	固定窗口	$O(Nw)$	无	Mistral
Longformer	窗口 + 全局	$O(N(w+g))$	全局 token	Longformer
BigBird	窗口 + 全局 + 随机	$O(N(w+g+r))$	全局 + 随机	BigBird
StreamingLLM	Sink+ 窗口	$O(N(s+w))$	Sink tokens	–
MoBA	块路由	$O(N \cdot kL)$	Top-k 块	Kimi
NSA	压缩 + 选择 + 窗口	$O(N^2/L)$	压缩 + 选择	–
DSA	Token 级检索	$O(Nk)$	Top-k token	DeepSeek-V3.2
Ring Attention	分布式	$O(N^2/P)$	完整	LLaMA 3

- 稀疏注意力减少**计算量**（只计算部分注意力）
- 两者可以结合：对选中的稀疏 token 使用 FlashAttention 计算

与 KV Cache 压缩的关系

- **KV Cache 压缩**（如 SnapKV、H2O）：推理时丢弃不重要的 KV
- **稀疏注意力**：训练和推理时都只计算部分注意力
- 稀疏注意力是更根本的解决方案，从架构层面减少计算

10.11 实践建议

何时使用稀疏注意力

- 长上下文场景（32K+）
- 需要精确检索能力（passkey retrieval 等）
- 希望保持 Softmax 注意力的特性

何时使用线性注意力

- 超长上下文（1M+）
- KV Cache 显存受限
- 可以接受一定的精度损失

Remark 10.2 (稀疏注意力的发展趋势)**.** 稀疏注意力正从“手工设计模式”（如固定滑动窗口）向“学习式稀疏”演进：

- *MoBA*：无参数路由，让模型学习关注哪些块
- *NSA*：可学习的压缩和选择机制
- 未来：更细粒度、更自适应的稀疏策略

11 线性注意力

稀疏注意力通过" 只计算重要的 token 对" 将 $O(N^2)$ 降至 $O(Nk)$，但它仍然保留了 softmax 的计算形式。本节介绍一条更激进的路径：**彻底改变注意力的数学形式**，使复杂度降至真正的 $O(N)$。

线性注意力的核心洞察是：softmax 注意力之所以需要 $O(N^2)$，是因为必须先算出完整的 $N \times N$ 注意力矩阵再做归一化。如果我们用其他函数替代 softmax，使得计算可以" 重新排列"，就能避免显式构造这个矩阵。这种重排不是免费的——我们用数学等价性换取计算效率，代价是改变了注意力的语义。

11.1 核心思想

从 Softmax Attention 到线性化

标准自注意力（单头）的计算为：

$$\text{Attention}(Q,K,V) = \underbrace{\text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)}_{n\times n} V \tag{146}$$

其中需要显式构造 $n \times n$ 的注意力矩阵，时间和空间复杂度均为 $O(n^2)$。

线性注意力的核心思想是：将 softmax 或 $QK^\top$ 改写为可分解的形式，利用乘法结合律改变计算顺序：

$$\text{Attention}(Q,K,V) \approx \phi(Q) \cdot \underbrace{(\phi(K)^\top V)}_{d\times d} \tag{147}$$

其中 $\phi(\cdot)$ 是某种特征映射函数。关键在于：先计算 $\phi(K)^\top V$（与长度 n 线性相关的 $d \times d$ 矩阵），再左乘 $\phi(Q)$，总复杂度降为 $O(nd^2)$，当 $d \ll n$ 时近似 $O(n)$。

递推形式：Transformer 即 RNN

在自回归（causal）场景下，线性注意力可以写成类似 RNN 的递推式 [**?**]。设 q_t, k_t, v_t 分别为第 t 步的 query、key、value 向量，定义状态矩阵 $S_t \in \mathbb{R}^{d\times d}$：

$$S_t = S_{t-1} + v_t k_t^\top \tag{148}$$

$$o_t = S_t \cdot q_t \tag{149}$$

这揭示了一个深刻联系：**线性 Attention 本质上是一种 RNN**，其隐状态 S_t 累积了历史信息。

Remark 11.1 (线性注意力的双模式)**.** *线性注意力支持两种等价的计算模式：*

- ***并行模式**：训练时使用矩阵乘法，充分利用 GPU 并行性*
- ***递推模式**：推理时使用 RNN 形式，实现 $O(1)$ 的增量更新*

这种双模式特性使线性注意力在训练和推理阶段都能获得最优效率。

11.2 经典线性注意力方法

Linear Transformer [**?**]

最早系统化提出 "Transformer 即 RNN" 的工作。核心思想是将 softmax attention 重写为核函数形式：

$$\text{Attention}(Q,K,V) = \frac{\phi(Q)(\phi(K)^\top V)}{\phi(Q)(\phi(K)^\top \mathbf{1})} \tag{150}$$

其中 $\phi(x) = \text{elu}(x) + 1$ 保证非负性。实验表明在自回归任务上可获得高达 4000 倍的加速。

局限：简单的特征映射难以精确近似 softmax 的行为，在复杂语言任务上存在性能差距。

Performer [?]

提出 FAVOR+（Fast Attention Via positive Orthogonal Random features）方法，用随机特征近似 softmax 核：

$$\text{softmax}(q^\top k) \approx \phi(q)^\top \phi(k) \tag{151}$$

其中 ϕ 通过正交随机特征构造，具有无偏或近似无偏的理论保证。

优势：与原始 Transformer 完全兼容，可作为 drop-in replacement。

局限：随机近似在实际任务中仍有精度损失。

cosFormer [?]

不再硬近似 softmax，而是基于 softmax 的两个关键性质设计线性替代：

1. **非负性**：注意力权重应为非负
2. **分布集中性**：注意力应集中在相关位置

cosFormer 使用 ReLU 保证非负性，并引入基于余弦的位置再加权机制：

$$\text{Attention}_{ij} = \text{ReLU}(q_i)^\top \text{ReLU}(k_j) \cdot \cos\left(\frac{\pi}{2} \cdot \frac{i-j}{n}\right) \tag{152}$$

在 Long-Range Arena 等长序列 benchmark 上取得当时最优性能，是 “好用的线性 Attention” 的代表。

11.3 带遗忘门的线性注意力

原始线性注意力的一个根本问题是：状态矩阵 S_t 只能累加，无法遗忘。随着序列增长，历史信息 “挤在一起”，导致检索能力下降。

RetNet [?]

微软提出的 Retentive Network 引入指数衰减因子 $\gamma \in (0, 1)$：

$$S_t = \gamma S_{t-1} + v_t k_t^\top \tag{153}$$

$$o_t = S_t \cdot q_t \tag{154}$$

这相当于对历史信息施加指数衰减，强调近期 token 的重要性。

Multi-Scale Retention：不同 attention head 使用不同的 γ 值，实现多尺度的记忆保持：

- 小 γ：关注近期信息（短程依赖）
- 大 γ：保留更长历史（长程依赖）

三种计算模式：

1. **并行模式**：训练时的矩阵计算
2. **递推模式**：$O(1)$ 推理
3. **分块递推模式**：长序列的高效处理

性能：7B 模型在 8k 序列长度下，推理速度比 Transformer 快 8.4 倍，内存减少 70%。

Lightning Attention [?]

MiniMax 提出的 Lightning Attention 是目前**首个规模化到商业级的线性注意力架构**。核心创新：

分块计算策略：将注意力计算分为 intra-block 和 inter-block 两部分：

- **Intra-block**：块内使用 “左乘” 形式，可并行计算
- **Inter-block**：块间使用 “右乘” 形式，累积状态

这种分解避免了传统线性注意力中缓慢的 cumsum 操作。

混合架构：每 8 层中，7 层使用 Lightning Attention，1 层使用标准 Softmax Attention，平衡效率与精度。

表 44: MiniMax-01 模型规格

参数	值
总参数量	456B
激活参数量（MoE）	45.9B
专家数量	32
训练上下文长度	1M tokens
推理外推长度	4M tokens

11.4 DeltaNet：基于 Delta Rule 的线性注意力

动机：解决记忆过载问题

原始线性注意力的核心缺陷是**记忆过载**（memory overload）：只能添加新的 key-value 关联，无法擦除已有信息。这导致随着序列增长，检索错误累积。

Delta Rule 更新

DeltaNet [**?**] 引入 "除旧迎新" 的 Delta Rule：

$$S_t = S_{t-1} - \underbrace{(S_{t-1} \cdot k_t - v_t)}_{\text{delta}} \cdot k_t^\top \tag{155}$$

直观理解：

1. $S_{t-1} \cdot k_t$：用当前 key 检索记忆中的 value
2. $S_{t-1} \cdot k_t - v_t$：计算检索值与真实值的差异（delta）
3. 根据 delta 修正记忆，实现 "精准更新"

Gated DeltaNet [?]

ICLR 2025 工作进一步引入门控机制：

$$S_t = \alpha_t \odot S_{t-1} + \beta_t \odot (v_t - S_{t-1} \cdot k_t) \cdot k_t^\top \tag{156}$$

其中 α_t 控制遗忘，β_t 控制更新强度。

互补性：门控实现快速记忆擦除，Delta Rule 实现精准记忆更新，两者结合在多个 benchmark 上超越 Mamba2 和原始 DeltaNet。

工业采用：Gated DeltaNet 已被 Qwen3-Next 采用作为线性注意力层。

11.5 与状态空间模型的联系

Mamba 与结构化状态空间对偶

Mamba [**?**] 是另一条重要的高效序列建模路线，基于选择性状态空间模型（Selective SSM）。Mamba-2 论文 [**?**] 揭示了**结构化状态空间对偶**（Structured State Space Duality, SSD）：

> "与标准自注意力相比，SSD 只有两个区别：去掉 softmax 归一化，并应用一个独立的逐元素掩码矩阵。"

这表明线性注意力和 SSM 可以视为同一框架的不同实例：

- 线性注意力：通过特征映射分解 attention 矩阵
- SSM：通过状态空间方程建模序列
- 两者都有线性复杂度和递推形式

混合架构

实践中，纯线性模型在某些任务上仍有差距，因此出现了混合架构：

- **Jamba**（AI21）：Mamba + Attention
- **MiniMax-01**：Lightning Attention + 稀疏 Softmax Attention
- **Qwen3-Next**：Gated DeltaNet + SwiGLU

11.6 Test-Time Training 视角

苏剑林在“线性注意力简史”[**?**] 中指出，现代线性注意力的核心思想可以统一到 **Test-Time Training**（TTT）框架：

> “将序列建模视为在线学习问题，用优化器构建 RNN。不同的损失函数对应不同的 RNN 模型。”

表 45: 线性注意力方法的 TTT 视角

方法	更新规则	对应优化器
Linear Attention	$S_t = S_{t-1} + v_t k_t^\top$	累积梯度
RetNet	$S_t = \gamma S_{t-1} + v_t k_t^\top$	带衰减的累积
DeltaNet	$S_t = S_{t-1} - (S_{t-1}k_t - v_t)k_t^\top$	Delta Rule
Gated DeltaNet	门控 Delta Rule	自适应学习率

这个视角为设计新型线性注意力提供了原则性指导：选择合适的“优化器”来更新记忆状态。

11.7 工业部署现状

表 46: 线性注意力的工业应用

公司/模型	架构类型	上下文长度	特点
MiniMax-01	Lightning Attention + MoE	1M-4M	首个商业级线性 Attention LLM
MiniMax-M1	Lightning Attention	1M+80k 生成	开源 reasoning 模型
Qwen3-Next	Gated DeltaNet	–	线性层 + 门控 Attention

关键观察：MiniMax 是目前唯一将线性注意力规模化到商业级的厂商。其他厂商（如 Kimi、DeepSeek）更倾向于稀疏注意力路线（见第10节）。

11.8 线性注意力的局限与展望

当前局限

1. **精度差距**：在复杂推理任务上，纯线性注意力仍落后于 Softmax Attention
2. **in-context learning 能力**：线性模型的 few-shot 能力通常弱于 Transformer
3. **长程精确检索**：passkey retrieval 等任务上表现不稳定

发展趋势

1. **混合架构**：结合线性层和稀疏 Softmax 层
2. **门控机制**：更精细的记忆管理（如 Gated DeltaNet）

3. **知识蒸馏**：从 Softmax 模型蒸馏到线性模型（如 LAWCAT）
4. **TTT 原则**：基于优化器视角设计新架构

Remark 11.2 (历史评价)**.** 苏剑林的评价 *[?]*: "线性注意力已从单纯模仿 *Softmax Attention* 发展到'反哺'它——通过核技巧将 *DeltaNet* 改进应用于 *Softmax Attention*。这表明该领域方兴未艾，仍有广阔探索空间。"

Part IV

模型架构

12 Mixture of Experts (MoE)

Mixture of Experts（MoE）是一种稀疏激活架构，通过让每个 token 只激活部分参数，实现了 “大模型容量、小模型计算量” 的目标。本节重点介绍 DeepSeek MoE 架构及其在工业界的广泛应用。

12.1 MoE 基础

核心思想

传统 Dense 模型中，每个 token 都要经过所有参数。MoE 的核心思想是：用**路由器**（Router）为每个 token 选择最相关的**专家**（Expert），只激活部分参数：

$$y = \sum_{i=1}^{N} g_i(x) \cdot E_i(x) \tag{157}$$

其中 E_i 是第 i 个专家（通常是 FFN），$g_i(x)$ 是路由器为 token x 分配给专家 i 的权重。

Top-K 路由

标准的 Top-K 路由机制：

$$s_i = x \cdot W_r^{(i)} \quad \text{(路由分数)} \tag{158}$$

$$g_i = \begin{cases} \text{softmax}(s)_i & \text{if } i \in \text{Top-}K(s) \\ 0 & \text{otherwise} \end{cases} \tag{159}$$

其中 W_r 是路由器的可学习参数。每个 token 只被发送到 K 个得分最高的专家。

表 47: MoE 术语对照

术语	含义
总参数量	模型所有参数（包括所有专家）
激活参数量	单个 token 前向传播使用的参数
专家数 N	可选专家的总数
激活专家数 K	每个 token 选择的专家数
稀疏度	N/K，越大表示越稀疏

12.2 DeepSeek MoE 架构

DeepSeek [**?**] 提出了目前最具影响力的 MoE 设计，被 DeepSeek-V2 [**?**]、DeepSeek-V3 [**?**] 等模型采用。

Fine-grained Expert Segmentation

传统 MoE 使用少量大专家（如 8 个），DeepSeek 提出**细粒度专家分割**：将专家数量增加 m 倍，同时将每个专家的参数减少 m 倍：

$$N \to mN, \quad K \to mK, \quad \text{Expert Size} \to \frac{1}{m} \tag{160}$$

优势：更多专家组合提供更灵活的知识表示。例如，从 8 个专家选 2 个有 $\binom{8}{2} = 28$ 种组合，而从 64 个专家选 16 个有 $\binom{64}{16} \approx 4.9 \times 10^{14}$ 种组合。

Shared Expert Isolation

除了路由专家外，DeepSeek 引入**共享专家**（Shared Experts）：

$$y = \underbrace{\sum_{i=1}^{K_s} E_i^{shared}(x)}_{\text{共享专家}} + \underbrace{\sum_{j=1}^{K_r} g_j(x) \cdot E_j^{routed}(x)}_{\text{路由专家}} \tag{161}$$

设计哲学：

- **共享专家**：捕获所有 token 都需要的通用知识（如语法、常识）
- **路由专家**：捕获领域特定知识（如数学、代码、医学）

这种分离减少了路由专家之间的知识冗余，提高了专业化程度。

DeepSeek-V2/V3 配置

表 48: DeepSeek MoE 模型配置

模型	总参数	激活参数	专家配置
DeepSeek-V2	236B	21B	160 路由 + 2 共享
DeepSeek-V3	671B	37B	256 路由 + 1 共享
DeepSeek-R1	671B	37B	同 V3

DeepSeek-V3 每个 token 激活 8 个路由专家 + 1 个共享专家，稀疏度高达 $256/8 = 32$。

12.3 负载均衡策略

MoE 训练的核心挑战是**负载均衡**：如果某些专家被过度选择，会导致：

1. **路由崩塌**（Routing Collapse）：所有 token 都选同几个专家
2. **计算效率下降**：专家并行时负载不均
3. **知识浪费**：部分专家从未被训练

传统方法：辅助损失

早期方法（如 Switch Transformer [**?**]）使用辅助损失强制负载均衡：

$$\mathcal{L}_{aux} = \alpha \cdot N \cdot \sum_{i=1}^{N} f_i \cdot P_i \tag{162}$$

其中 f_i 是专家 i 实际处理的 token 比例，P_i 是路由分数的平均值。

问题：辅助损失与主任务损失相互竞争，α 过大会损害模型性能。

DeepSeek-V2：多级辅助损失

DeepSeek-V2 引入三级辅助损失 [**?**]：

1. **Expert-level**：平衡单个专家的负载
2. **Device-level**：平衡不同设备上的专家负载
3. **Communication-level**：减少跨设备通信

DeepSeek-V3：无辅助损失负载均衡

DeepSeek-V3 提出革命性的 **Auxiliary-Loss-Free 负载均衡** [**?**]：

核心思想：为每个专家引入可调偏置项 b_i，仅用于路由决策，不参与 loss 计算：

$$s'_i = s_i + b_i \tag{163}$$

动态调整：

- 专家过载时，减小 b_i 降低被选概率
- 专家空闲时，增大 b_i 提高被选概率

关键优势：负载均衡目标与质量优化目标完全解耦，不再相互竞争。实验表明 V3 在整个训练过程中保持良好的负载均衡，无需丢弃任何 token。

12.4 路由约束与通信优化

Node-Limited Routing

在分布式训练中，专家分布在不同节点上。跨节点通信成本高昂。DeepSeek 引入**节点限制路由**：

> 每个 token 最多被发送到 M 个节点。

这限制了 All-to-All 通信的范围，大幅减少通信开销。

Expert Tensor Parallelism

MiniMax 提出 **Expert Tensor Parallel**（ETP）：将单个专家的参数切分到多个设备，而非将不同专家放在不同设备。这种方式更适合细粒度专家架构。

12.5 工业应用对比

表 49: 主流 MoE 模型对比

模型	总参数	激活参数	专家配置	特点
DeepSeek-V3	671B	37B	256+1	无辅助损失均衡
MiniMax-01	456B	45.9B	32 专家	Lightning Attention
Kimi K2	1T	32B	384 路由	MuonClip 优化器
Qwen2-57B-A14B	57B	14B	60+4 共享	Upcycling

DeepSeek-V3

关键创新：

- Auxiliary-Loss-Free 负载均衡
- Multi-Token Prediction (MTP)
- FP8 混合精度训练
- 仅需 2.788M H800 GPU 小时完成训练

性能：671B 参数，每 token 激活 37B，在多项 benchmark 上达到 GPT-4 级别。

MiniMax-01

架构特点：

- 32 个专家，每 token 激活约 45.9B 参数
- 结合 Lightning Attention 的混合架构
- 每 7 层线性注意力后接 1 层 Softmax 注意力

长上下文：训练 1M tokens，推理可扩展至 4M tokens。

Kimi K2

规模：1T 总参数，32B 激活参数——目前最大的开源 MoE 模型之一。

架构：

- 类似 DeepSeek-V3 的 MLA + MoE 架构
- 384 个路由专家，每 token 激活 8 个
- 稀疏度：$384/8 = 48$（高于 DeepSeek-V3 的 32）

训练：使用 Muon 优化器（MuonClip 变体），15.5T tokens 训练，零训练不稳定性。

Qwen MoE

Qwen 采用 **Upcycling** 策略：从 Dense 模型初始化 MoE 专家。

Qwen2-57B-A14B：

- 从 Qwen2-7B upcycle 而来
- 60 个路由专家 + 4 个共享专家
- 激活 14B 参数，性能接近 34B Dense 模型

12.6 MoE 的理论理解

稀疏性与容量

MoE 的核心 trade-off 是**稀疏性**与**模型容量**：

- 更多专家 → 更大容量，但通信开销增加
- 更少激活专家 → 更高效率，但可能欠拟合

DeepSeek-V3 的经验：256 个专家 + 8 个激活是一个 good balance。

专家专业化

理想情况下，不同专家应学会处理不同类型的知识：

- 某些专家处理数学推理
- 某些专家处理代码生成
- 某些专家处理多语言

共享专家的引入帮助路由专家更好地专业化，避免 "每个专家都学一点通用知识"。

Remark 12.1 (MoE vs Dense 的选择)**.** *MoE 并非总是优于 Dense*：

- ***MoE* 优势**：相同计算预算下更大容量；推理时更高效
- ***Dense* 优势**：更简单的训练和部署；在某些任务上更稳定

当前趋势是在超大规模模型（100B+）中使用 MoE，中小规模仍以 Dense 为主。

12.7 MoE 训练技巧

负载均衡

- 优先使用 DeepSeek-V3 的无辅助损失方法
- 如使用辅助损失，系数 α 需仔细调优（通常 $0.01 \sim 0.1$）

专家并行

- 小规模：所有专家放单卡
- 中规模：Expert Parallelism，不同专家在不同卡
- 大规模：结合 TP、EP、PP 的混合并行

Upcycling 从 Dense 模型初始化 MoE 可以加速收敛：

1. 复制 Dense 模型的 FFN 作为各专家的初始化
2. 随机初始化路由器
3. 继续预训练，专家逐渐分化

容量因子 传统 MoE 设置容量因子（Capacity Factor）限制每个专家处理的最大 token 数，超出的 token 被丢弃。DeepSeek-V3 证明：良好的负载均衡策略可以**完全避免 token 丢弃**。

Part V

训练技术

13 大模型数据工程

数据是大语言模型的基石。本节系统介绍预训练数据、后训练数据和评估数据的构建方法，以及数据配比的经验法则。

13.1 预训练数据

数据来源

现代 LLM 的预训练数据通常来自以下来源：

表 50: 主要预训练数据来源

数据源	描述	规模	质量
网页数据			
Common Crawl	最大的网页爬虫数据	PB 级	低
RefinedWeb	过滤后的高质量网页	5T tokens	中
C4	Colossal Clean Crawled Corpus	800B tokens	中
高质量文本			
Wikipedia	百科全书	数十 B tokens	高
Books	书籍（Books3, Pile-Books）	数十 B tokens	高
arXiv	学术论文	数十 B tokens	高
代码			
GitHub	开源代码仓库	数百 B tokens	中-高
The Stack	去重的开源代码	3T tokens	高
StarCoder Data	精选编程数据	1T tokens	高
对话/社交			
Reddit	社交媒体讨论	数百 B tokens	中
StackExchange	问答社区	数十 B tokens	高

数据处理流程

从原始数据到训练数据需要经过多个处理阶段：

1. 文本提取

- HTML 解析：使用 trafilatura、jusText 等工具提取正文
- PDF 解析：使用 PyMuPDF、pdfplumber 处理学术文档
- 代码处理：保留注释和文档字符串

2. 质量过滤

- **启发式规则**：
  - 文档长度过滤（过短/过长）
  - 特殊字符比例
  - 重复行/段落比例
  - 语言检测（fastText 语言分类器）
- **模型打分**：
  - 困惑度过滤（用小模型计算 PPL）
  - 质量分类器（如 GPT-3 用的质量分类器）
  - 教育价值评分（FineWeb-Edu 使用的方法）

3. 去重 去重是预训练数据处理的关键步骤，重复数据会导致：

- 训练效率下降
- 模型记忆而非泛化
- 隐私泄露风险

主要去重方法：

- **精确去重**：基于哈希的完全匹配
- **模糊去重**：MinHash + LSH（Locality-Sensitive Hashing）
- **文档级 vs 段落级**：不同粒度的去重策略

```python
# MinHash去重示例
from datasketch import MinHash, MinHashLSH

def get_minhash(text, num_perm=128):
    m = MinHash(num_perm=num_perm)
    for word in text.split():
        m.update(word.encode('utf8'))
    return m

# 创建LSH索引
lsh = MinHashLSH(threshold=0.8, num_perm=128)
for doc_id, text in documents:
    minhash = get_minhash(text)
    lsh.insert(doc_id, minhash)
```

4. 敏感内容过滤

- PII（个人身份信息）移除：邮箱、电话、身份证号等
- 有害内容过滤：使用分类器检测
- 版权内容处理：根据许可证过滤

数据配比

数据配比（Data Mix）对模型能力有重要影响：

表 51: 主流模型的数据配比（估计值）

模型	网页	代码	书籍	学术	其他
GPT-3	60%	3%	16%	-	21%
LLaMA	67%	4.5%	4.5%	2.5%	21.5%
LLaMA 2	89%	-	-	-	11%
DeepSeek	56%	18%	-	5%	21%
Qwen	类似 LLaMA	较高代码占比	-	-	-

配比原则

- **代码数据**：提升推理能力，通常占 5-20%
- **数学数据**：提升数学能力，但占比过高可能损害通用能力
- **多语言数据**：根据目标市场调整，中文模型通常 30-50% 中文
- **高质量数据**：虽然量少，但应该多次采样（upsampling）

数据规模与 Scaling Law

Chinchilla Scaling Law [**?**] 指出，计算最优的数据量与模型参数成正比：

$$D_{opt} \approx 20 \times N \tag{164}$$

其中 D 是 token 数，N 是参数量。即 7B 模型需要约 140B tokens。
然而，实践中常常使用更多数据（over-training）：

- LLaMA-7B：1T tokens（约 143 倍参数量）
- LLaMA 2-7B：2T tokens（约 286 倍）
- LLaMA 3-8B：15T tokens（约 1875 倍）

为什么 Over-training?

- 推理成本固定，训练成本可摊销
- 更小的模型更容易部署
- 数据重复使用在一定程度内是有益的

多轮训练与课程学习

数据课程（Data Curriculum） 按照特定顺序呈现数据可以提升模型性能：

1. **阶段 1**：通用网页数据，建立基础语言能力
2. **阶段 2**：增加高质量数据比例（书籍、维基百科）
3. **阶段 3**：增加代码和数学数据
4. **阶段 4**：Annealing 阶段，使用最高质量数据，降低学习率

退火阶段（Annealing） 训练末期使用高质量数据的策略：

- 学习率从正常值退火到接近 0
- 数据切换为最高质量子集
- 通常占总训练的 1-5%
- LLaMA 3 报告此阶段显著提升 benchmark 性能

13.2 后训练数据

后训练（Post-training）包括监督微调（SFT）和人类偏好对齐（RLHF/DPO），数据需求与预训练截然不同。

SFT 数据

数据格式 SFT 数据通常是指令-响应对：

```
{
  "instruction": "将以下句子翻译成英文",
  "input": "今天天气很好",
  "output": "The weather is nice today."
}
```

或多轮对话格式：

```
{
  "conversations": [
    {"role": "user", "content": "什么是机器学习？"},
```

```
    {"role": "assistant", "content": "机器学习是..."},
    {"role": "user", "content": "它有哪些应用？"},
    {"role": "assistant", "content": "机器学习的应用包括..."}
  ]
}
```

数据来源

- **人工标注**：质量最高，成本最高
- **众包平台**：如 Scale AI、Surge AI
- **开源数据集**：
  - FLAN Collection：多任务指令数据
  - OpenAssistant：社区标注对话
  - ShareGPT：用户分享的 ChatGPT 对话
  - Alpaca：GPT-4 生成的指令数据
- **合成数据**：用强模型生成（Self-Instruct、Evol-Instruct）

数据规模 SFT 数据量远小于预训练：

- 早期：几千到几万条（Alpaca 52K）
- 现代：几十万到几百万条
- 质量比数量更重要（LIMA 论文：1000 条精选数据即可）

数据质量控制

- 响应长度：避免过短/过长的回复
- 格式一致性：统一的回复风格
- 多样性：覆盖不同任务类型
- 准确性：人工审核或模型验证

偏好数据

RLHF 和 DPO 需要偏好对数据：

数据格式

```
{
  "prompt": "解释量子计算",
  "chosen": "量子计算是一种利用量子力学原理...",
  "rejected": "量子计算就是很快的计算机..."
}
```

收集方法

1. **人工标注**：
   - 给定 prompt，生成多个响应
   - 人类标注者选择更好的响应
   - 成本高，但质量可控
2. **AI 反馈**：
   - 用 GPT-4 等强模型做评判

- Constitutional AI：用规则约束 AI 反馈
- RLAIF（RL from AI Feedback）

3. **隐式反馈**：
   - 用户点赞/点踩
   - 用户选择哪个回复继续对话
   - 用户编辑/重新生成

数据规模

- InstructGPT：约 33K 比较对
- LLaMA 2 Chat：约 100 万偏好数据
- 现代实践：几十万到几百万对

合成数据

合成数据在后训练中越来越重要：

Self-Instruct 用模型自己生成指令和响应：

1. 从种子任务开始
2. 让模型生成新的指令
3. 过滤低质量/重复的指令
4. 让模型生成响应

Evol-Instruct WizardLM 提出的指令进化方法：

- **深度进化**：增加复杂度、约束、推理步骤
- **广度进化**：变换话题、领域、风格

```
# 深度进化prompt示例
"""
请将以下指令改写得更复杂，加入更多约束条件：
原始指令：写一首关于春天的诗
进化指令：写一首关于春天的七言绝句，要求包含至少三种春天的景物，
        使用对仗工整的句式，并在最后一句表达对时光流逝的感慨。
"""
```

蒸馏数据 从强模型蒸馏知识到弱模型：

- 用 GPT-4/Claude 生成高质量响应
- 收集推理过程（Chain-of-Thought）
- DeepSeek-R1 蒸馏：用 R1 的推理轨迹训练小模型

13.3 评估数据

通用能力评估

对话能力评估

- **MT-Bench**：多轮对话评估，GPT-4 作为评判
- **AlpacaEval**：单轮指令跟随，与 GPT-4 比较
- **Arena Hard**：基于 Chatbot Arena 的困难子集
- **Chatbot Arena**：用户盲评，ELO 排名

表 52: 主要评估基准

基准	类型	规模	评估内容
知识与推理			
MMLU	多选题	57 科目	世界知识
HellaSwag	选择题	10K	常识推理
ARC	选择题	7.8K	科学推理
WinoGrande	选择题	44K	常识推理
数学			
GSM8K	应用题	8.5K	小学数学
MATH	竞赛题	12.5K	高中竞赛数学
代码			
HumanEval	代码生成	164	Python 编程
MBPP	代码生成	974	Python 编程
长文本			
RULER	多任务	-	长上下文理解
LongBench	多任务	4.75K	长文本能力

安全性评估

- **TruthfulQA**：测试模型是否会生成虚假信息
- **ToxiGen**：测试有害内容生成
- **BBQ**：测试社会偏见
- **Red-teaming**：对抗性攻击测试

13.4 数据工程实践

数据飞轮

现代 LLM 公司通常建立数据飞轮：

1. 部署模型服务用户
2. 收集用户交互数据（带隐私保护）
3. 标注高质量样本
4. 训练更好的模型
5. 回到步骤 1

数据质量 vs 数量

表 53: 数据质量与数量的权衡

阶段	优先级	理由
预训练早期	数量	需要大量数据建立基础能力
预训练后期	质量	Annealing 阶段，精选数据提升性能
SFT	质量 > 数量	几千条高质量数据可能胜过百万低质量
RLHF	质量	偏好标注错误会导致 reward hacking

LIMA 的启示 LIMA 论文 [**?**] 表明：

- 1000 条精心策划的 SFT 数据可以产生强大的对话模型
- 数据多样性比数量更重要
- 响应风格的一致性很关键

数据污染

评估数据泄露到训练数据中是严重问题：

检测方法

- N-gram 重叠检测
- 困惑度异常检测
- 模型记忆测试（completion）

预防措施

- 训练数据与评估数据去重
- 使用时间切分（训练数据早于评估数据）
- 定期更新评估基准

13.5 开源数据集

表 54: 重要开源数据集

数据集	类型	规模	特点
预训练			
RedPajama	通用	1.2T tokens	LLaMA 数据复现
The Pile	通用	825GB	多源混合
FineWeb	网页	15T tokens	高质量过滤
FineWeb-Edu	网页	1.3T tokens	教育内容
StarCoder Data	代码	1T tokens	许可证友好
SFT			
FLAN Collection	指令	数百万	多任务
OpenAssistant	对话	161K	多语言
UltraChat	对话	1.5M	合成对话
偏好			
HH-RLHF	偏好对	170K	Anthropic 发布
UltraFeedback	偏好对	64K	GPT-4 标注

Remark 13.1 (数据工程建议)**.**

- **预训练**：投资于数据处理流水线，去重和过滤至关重要
- ***SFT***：质量优先，人工审核每一条数据
- **偏好对齐**：确保标注一致性，避免噪声标签
- **持续改进**：建立数据飞轮，不断收集和迭代

14 分布式训练框架

训练大规模语言模型需要跨多 GPU、多节点的分布式系统。本章介绍主流的分布式训练技术，包括数据并行、模型并行、ZeRO 优化器、FSDP 等。

14.1 分布式训练概述

为什么需要分布式训练

现代 LLM 的参数量已达千亿级别：

- GPT-3：175B 参数，需要约 700GB 显存（FP32）
- LLaMA-70B：70B 参数，需要约 280GB 显存（FP32）
- DeepSeek-V3：671B 参数

单 GPU 显存有限（A100: 80GB, H100: 80GB），必须将模型分布到多个设备上。

训练时的显存占用

训练一个参数量为 Φ 的模型，显存占用包括：

表 55: 训练显存占用分解（以 Adam + FP16 混合精度为例）

组件	精度	显存（字节）
模型参数	FP16	2Φ
梯度	FP16	2Φ
优化器状态（Adam）		
- FP32 参数副本	FP32	4Φ
- 一阶矩 m	FP32	4Φ
- 二阶矩 v	FP32	4Φ
激活值	变化	与 batch、seq_len 相关
总计（不含激活）		16Φ

对于 7B 模型：$16 \times 7 \times 10^9 = 112$GB，超过单卡容量。

14.2 数据并行

原理

数据并行（Data Parallelism, DP）是最简单的分布式策略：

1. 每个 GPU 持有**完整的模型副本**
2. 数据集被分割，每个 GPU 处理不同的 mini-batch
3. 前向传播独立进行
4. 反向传播后，**All-Reduce** 同步梯度
5. 每个 GPU 独立更新参数（结果相同）

PyTorch DDP

PyTorch 的 `DistributedDataParallel`（DDP）是标准的数据并行实现：

- 使用 NCCL 后端进行高效 GPU 通信
- 梯度计算与通信重叠（Overlap）
- 支持梯度桶（Gradient Bucketing）优化

局限：每个 GPU 必须能容纳完整模型，无法训练超大模型。

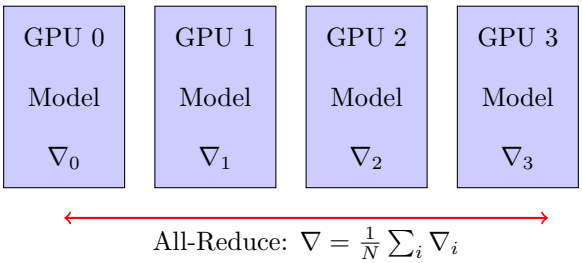

图 9: 数据并行：每个 GPU 持有完整模型，梯度通过 All-Reduce 同步

14.3 ZeRO：零冗余优化器

ZeRO（Zero Redundancy Optimizer）[**?**] 是 DeepSpeed 的核心技术，通过分片消除数据并行中的内存冗余。

三个阶段

ZeRO-1：优化器状态分片 将 Adam 的 m, v 和 FP32 参数副本分片到 N 个 GPU：

$$\text{优化器显存}: 12\Phi \to \frac{12\Phi}{N} \tag{165}$$

每个 GPU 只更新 $1/N$ 的参数，通过 All-Gather 获取完整参数。

显存节省：约 4×（相比普通 DP）

ZeRO-2：+ 梯度分片 梯度也按 $1/N$ 分片，每个 GPU 只保留与其优化器分片对应的梯度：

$$\text{梯度显存}: 2\Phi \to \frac{2\Phi}{N} \tag{166}$$

使用 Reduce-Scatter 替代 All-Reduce。

显存节省：约 8×

ZeRO-3：+ 参数分片 模型参数也分片，前向/反向传播时按需 All-Gather：

$$\text{参数显存}: 2\Phi \to \frac{2\Phi}{N} \tag{167}$$

显存节省：约 N×（线性扩展）

表 56: ZeRO 各阶段显存占用对比（N 个 GPU）

阶段	分片内容	单 GPU 显存	通信量
DDP	无	16Φ	2Φ
ZeRO-1	优化器状态	$4\Phi + 12\Phi/N$	2Φ
ZeRO-2	+ 梯度	$2\Phi + 14\Phi/N$	2Φ
ZeRO-3	+ 参数	$16\Phi/N$	3Φ

ZeRO-Offload 与 ZeRO-Infinity

ZeRO-Offload 将优化器状态和计算卸载到 CPU：

- GPU 只保留 FP16 参数和梯度
- CPU 负责 FP32 参数更新
- 单 GPU 可训练 10B+ 模型

ZeRO-Infinity 进一步卸载到 NVMe SSD：

- 利用 NVMe 的大容量（TB 级）
- 512 GPU 可训练万亿参数模型
- 带宽优化：预取、分层缓存

14.4 模型并行

当单个模型层无法放入单 GPU 时，需要模型并行。

张量并行（Tensor Parallelism）

张量并行 [**?**] 将单个层的参数矩阵切分到多个 GPU。

MLP 层的张量并行 对于 FFN 层 $Y = \text{GeLU}(XW_1)W_2$：

列切分 W_1（沿输出维度）：

$$W_1 = [W_1^{(1)}, W_1^{(2)}], \quad Y_1 = \text{GeLU}(XW_1^{(i)}) \tag{168}$$

每个 GPU 独立计算部分输出，无需通信。

行切分 W_2（沿输入维度）：

$$W_2 = \begin{bmatrix} W_2^{(1)} \\ W_2^{(2)} \end{bmatrix}, \quad Y = \sum_i Y_1^{(i)} W_2^{(i)} \tag{169}$$

需要 All-Reduce 求和。

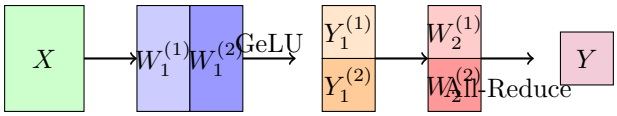

图 10: MLP 层的张量并行：W_1 列切分，W_2 行切分

注意力层的张量并行 多头注意力天然适合张量并行：

- 将 h 个头分配到 N 个 GPU，每个 GPU 处理 h/N 个头
- Q, K, V 投影矩阵按头切分（列切分）
- 输出投影 W_O 按行切分
- 前向需要 1 次 All-Reduce，反向需要 1 次 All-Reduce

通信开销 每个 Transformer 层需要：

- 前向：2 次 All-Reduce（注意力 + MLP）
- 反向：2 次 All-Reduce

张量并行适合**节点内**高带宽互连（NVLink: 600GB/s）。

序列并行（Sequence Parallelism）

序列并行 [**?**] 在序列维度切分，减少激活显存：

- LayerNorm 和 Dropout 在序列维度独立，可以分片
- 与张量并行配合：TP 区域外使用 SP
- 将 All-Reduce 拆分为 Reduce-Scatter + All-Gather

优势：激活显存降低 N 倍（N 为 TP 度），无额外通信。

流水线并行（Pipeline Parallelism）

流水线并行 [?] 将模型按层切分到不同 GPU：

- GPU 0：Layer 0-7
- GPU 1：Layer 8-15
- ...

朴素流水线的问题　顺序执行导致严重的**流水线气泡**（Pipeline Bubble）：

$$\text{气泡比例} = \frac{p-1}{m+p-1} \tag{170}$$

其中 p 是流水线阶段数，m 是 micro-batch 数。

GPipe 调度　将 mini-batch 切分为多个 micro-batch，增大 m 减少气泡：

- $m \gg p$ 时气泡可忽略
- 但需要存储所有 micro-batch 的激活，显存压力大

1F1B 调度　一个前向、一个反向交替执行：

- 稳态时每个 GPU 同时有 1 个 micro-batch 在前向、1 个在反向
- 激活显存只需存储 p 个 micro-batch
- 气泡比例与 GPipe 相同，但显存更低

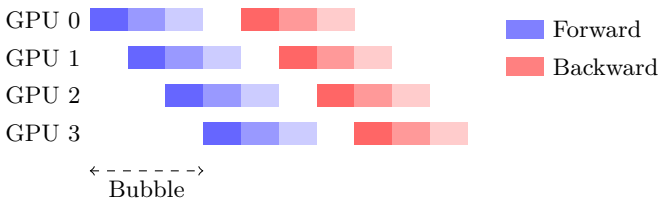

图 11: 1F1B 流水线调度示意图

14.5　3D 并行

Megatron-LM [?] 将 DP、TP、PP 组合成 **3D 并行**：

$$\text{总 GPU 数} = N_{DP} \times N_{TP} \times N_{PP} \tag{171}$$

并行度选择原则

1. **TP 优先用于节点内**：NVLink 带宽高，TP 通信频繁
2. **PP 用于节点间**：通信量小（只传激活），可跨节点
3. **DP 扩展吞吐**：通信可与计算重叠

表 57: 3D 并行配置示例

模型	GPU 数	TP	PP	DP	配置说明
GPT-3 175B	1024	8	8	16	8 GPU/节点
LLaMA-70B	64	8	2	4	单节点 8 卡
DeepSeek-V3	2048	8	-	256	MoE+EP

14.6 PyTorch FSDP

Fully Sharded Data Parallel（FSDP） [**?**] 是 PyTorch 原生的 ZeRO-3 实现。

核心特性

- **参数分片**：模型参数、梯度、优化器状态全部分片
- **按需 All-Gather**：前向/反向时临时恢复完整参数
- **Reduce-Scatter 梯度**：反向后立即分片梯度
- **与 torch.compile 兼容**：可获得额外加速

分片策略

FSDP 支持多种分片粒度：

- `FULL_SHARD`：完全分片（类似 ZeRO-3）
- `SHARD_GRAD_OP`：只分片梯度和优化器（类似 ZeRO-2）
- `NO_SHARD`：不分片（类似 DDP）
- `HYBRID_SHARD`：节点内分片，节点间复制

FSDP2

PyTorch 2.4 引入的 FSDP2 改进：

- **Per-parameter 分片**：更细粒度的控制
- **Selective Activation Checkpointing**：选择性重计算
- **SimpleFSDP**：编译器优化版本，吞吐提升可达 68%

FSDP vs DeepSpeed ZeRO

表 58: FSDP 与 DeepSpeed ZeRO 对比

特性	PyTorch FSDP	DeepSpeed ZeRO
原生支持	PyTorch 内置	独立库
CPU Offload	支持	支持
NVMe Offload	有限	ZeRO-Infinity
torch.compile	兼容	有限
配置复杂度	较低	较高
生态集成	HuggingFace 等	广泛

14.7 混合精度训练

混合精度训练 [**?**] 使用低精度（FP16/BF16）加速计算，同时保持训练稳定性。

数值格式对比

表 59: 浮点数格式对比

格式	位数	指数位	尾数位	动态范围	精度
FP32	32	8	23	$\pm 3.4 \times 10^{38}$	高
FP16	16	5	10	$\pm 6.5 \times 10^{4}$	中
BF16	16	8	7	$\pm 3.4 \times 10^{38}$	低
FP8 (E4M3)	8	4	3	± 448	很低
FP8 (E5M2)	8	5	2	$\pm 5.7 \times 10^{4}$	极低

Loss Scaling

FP16 的动态范围小，梯度可能下溢。**Loss Scaling** 将损失放大：

$$\tilde{L} = s \cdot L \quad \text{(放大损失)} \tag{172}$$

$$\tilde{g} = \nabla \tilde{L} = s \cdot \nabla L \quad \text{(放大梯度)} \tag{173}$$

$$g = \tilde{g}/s \quad \text{(更新前还原)} \tag{174}$$

动态 Loss Scaling：

- 若无溢出，增大 s（如 $s \leftarrow 2s$）
- 若发生溢出（NaN/Inf），减小 s 并跳过此步更新

BF16 的优势

BF16 与 FP32 有相同的指数位，因此：

- **无需 Loss Scaling**：动态范围与 FP32 相同
- **训练更稳定**：不会因范围问题溢出
- **精度略低**：尾数只有 7 位（FP16 有 10 位）

现代框架（PyTorch AMP、DeepSpeed）在支持的硬件上默认使用 BF16。

FP8 训练

FP8 需要**每张量缩放**（Per-tensor Scaling）：

- 每个 FP8 张量有独立的缩放因子
- NVIDIA Transformer Engine 自动管理缩放
- 需要 torch.compile 才能获得加速
- H100 上可比 BF16 快约 10%

14.8 激活检查点

激活检查点（Activation Checkpointing） [**?**] 通过重计算节省激活显存。

原理

前向传播时不保存中间激活，反向传播时重新计算：

- **无检查点**：保存所有激活，显存 $O(N)$
- **全检查点**：只保存输入，显存 $O(1)$，计算 $2\times$
- $\sqrt{N}$ **检查点**：每 $\sqrt{N}$ 层设检查点，显存 $O(\sqrt{N})$

选择性检查点

并非所有操作都值得重计算：

- **重计算**：便宜的操作（LayerNorm、激活函数、Dropout）
- **保存**：昂贵的操作（矩阵乘法、注意力）

PyTorch 的 Selective Activation Checkpointing（SAC）支持细粒度控制。

注意力重计算

注意力层的激活显存是 $O(N^2)$（N 为序列长度），占比最大。FlashAttention 通过：

- 前向时只保存 (m, ℓ) 统计量
- 反向时重计算注意力矩阵

实现了激活显存 $O(N)$，是目前最高效的注意力重计算方案。

14.9 集合通信

分布式训练依赖高效的集合通信原语。

常用通信原语

表 60: 集合通信原语

原语	功能	应用场景
Broadcast	一对多广播	参数初始化
Reduce	多对一归约	梯度聚合到一个节点
All-Reduce	归约后广播	DDP 梯度同步
All-Gather	收集所有分片	ZeRO 参数恢复
Reduce-Scatter	归约后分片	ZeRO 梯度分片
All-to-All	全交换	MoE 专家通信

Ring All-Reduce

Ring All-Reduce 是带宽最优的 All-Reduce 算法：

1. **Reduce-Scatter 阶段**：每个 GPU 向下一个发送 $1/N$ 数据，接收并累加
2. **All-Gather 阶段**：每个 GPU 广播自己的归约结果

通信量：$2(N-1)/N \cdot D \approx 2D$（$D$ 为数据量），与 GPU 数 N 无关。

延迟：$O(N)$ 步，随 GPU 数线性增长（延迟非最优）。

NCCL

NVIDIA Collective Communication Library（NCCL）是 GPU 集群通信的标准库：

- 自动选择最优拓扑（Ring、Tree、NVLink 等）
- 支持多节点、多 GPU
- 与 CUDA 深度集成

14.10 训练框架对比

表 61: 主流训练框架对比

框架	维护方	TP	PP	ZeRO/FSDP	特点
Megatron-LM	NVIDIA	✓	✓	ZeRO	3D 并行先驱
DeepSpeed	Microsoft	✓	✓	ZeRO-3	Offload、Infinity
PyTorch FSDP	Meta	-	-	FSDP	原生集成
Alpa	UC Berkeley	✓	✓	-	自动并行
ColossalAI	HPC-AI	✓	✓	ZeRO	易用性

14.11 实践建议

小模型（< 10B）

- 单卡能放下：使用 DDP
- 单卡放不下：使用 FSDP 或 ZeRO-2

中等模型（10B - 100B）

- 单节点：FSDP/ZeRO-3 + 激活检查点
- 多节点：3D 并行（TP=8 节点内，PP 跨节点）

超大模型（100B+）

- 必须使用 3D 并行
- 结合专家并行（MoE）
- 考虑 FP8 混合精度

Remark 14.1 (通信与计算的权衡)**.** 增加并行度会增加通信开销：

- *TP* 通信最频繁，应限制在高带宽节点内
- *PP* 有气泡开销，需要足够的 *micro-batch*
- *ZeRO-3* 的通信量是 *DDP* 的 *1.5* 倍

最优配置需要根据具体硬件和模型进行调优。

15 Muon 优化器

Adam 优化器自 2014 年提出以来，一直是深度学习的标准选择。然而，Adam 本质上是**逐元素**（element-wise）的优化器，没有利用神经网络参数的**矩阵结构**。Muon（MomentUm Orthogonalized by Newton-schulz）优化器 [**?**] 通过对动量进行矩阵正交化，实现了更高效的参数更新，已被 Kimi 等大规模模型采用 [**?**]。

15.1 从 Adam 到矩阵优化

Adam 的局限

Adam 的更新规则为：

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{175}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{176}$$

$$\theta_t = \theta_{t-1} - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{177}$$

其中所有操作都是**逐元素**进行的。对于矩阵参数 $W \in \mathbb{R}^{d_{out} \times d_{in}}$，Adam 将其展平为向量处理，完全忽略了矩阵的行列结构。

矩阵参数的特殊性

神经网络的核心参数——线性层权重矩阵——具有天然的矩阵结构：

- 行向量对应输出特征的组合权重
- 列向量对应输入特征的映射方向
- 奇异值分解揭示了参数的 “主方向”

一个关键观察是：SGD 和 Adam 产生的梯度更新通常具有**极高的条件数**，即接近低秩矩阵。这意味着更新主要沿少数 “主方向” 进行，而 “稀有方向” 被严重抑制。

15.2 Muon 的核心思想：正交化动量

基本算法

Muon 的核心思想是：将动量矩阵 M 替换为其最近的半正交矩阵，即进行**极分解**（Polar Decomposition）。设 $M = U\Sigma V^\top$ 是 M 的奇异值分解，则：

$$\text{msign}(M) = UV^\top \tag{178}$$

这称为矩阵的**符号函数**（matrix sign function），类似于标量的 sign 函数将所有奇异值映射为 1。

Algorithm 2 Muon 优化器（朴素版）

Require: 学习率 η，动量系数 β，初始参数 W_0
初始化动量 $M_0 = 0$
for $t = 1, 2, \ldots$ **do**
  计算梯度 $G_t = \nabla_W \mathcal{L}(W_{t-1})$
  更新动量 $M_t = \beta M_{t-1} + G_t$
  计算正交化更新 $\Delta_t = \text{msign}(M_t)$
  更新参数 $W_t = W_{t-1} - \eta \cdot \Delta_t$
end for

为什么正交化有效?

要理解正交化的作用，先看 Adam 的问题。设梯度矩阵 $G \in \mathbb{R}^{d_{out} \times d_{in}}$ 的奇异值分解为 $G = U\Sigma V^\top$，其中 $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_r)$，$\sigma_1 \gg \sigma_r$（高条件数）。

Adam 的逐元素缩放 Adam 对每个元素 g_{ij} 独立归一化，这相当于在" 元素坐标系" 下操作。问题在于：梯度的主方向（σ_1 对应）和稀有方向（σ_r 对应）在元素坐标系中是耦合的——无法通过逐元素缩放来均衡它们。

Muon 的谱归一化 $\text{msign}(G) = UV^\top$ 将所有奇异值映射为 1：

$$G = U \cdot \text{diag}(\sigma_1, \ldots, \sigma_r) \cdot V^\top \xrightarrow{\text{msign}} U \cdot \text{diag}(1, \ldots, 1) \cdot V^\top \tag{179}$$

这意味着：

1. **主方向被抑制**：$\sigma_1 \to 1$，更新幅度降低
2. **稀有方向被放大**：$\sigma_r \to 1$，更新幅度提升
3. **方向信息保留**：U, V 不变，只改变" 步长"

几何直觉 想象损失函数的等高线是一个狭长的椭圆（高条件数）。Adam 沿梯度方向走，容易在窄谷中震荡。Muon 将椭圆" 压成圆"——沿各方向走相同步长，这正是牛顿法的效果，但无需计算 Hessian。

Remark 15.1 (与 Shampoo 的关系)**.** *Muon 可以理解为" 无累积的 Shampoo"。Shampoo 通过累积梯度的二阶统计量 $L = \sum GG^\top$, $R = \sum G^\top G$, 然后用 $L^{-1/4}GR^{-1/4}$ 预条件化更新。Muon 直接用当前梯度的 SVD 达到类似效果，避免了:*

- *累积带来的延迟（需要多步才能准确估计统计量）*
- *逆矩阵平方根的高昂计算成本*
- *大规模分布式训练中统计量同步的通信开销*

代价是 Muon 对单步梯度的噪声更敏感，因此使用动量 $M = \beta M + G$ 来平滑。

15.3 Newton-Schulz 迭代

直接计算 SVD 的复杂度为 $O(\min(d_{out}, d_{in})^3)$，对于大矩阵不可接受。Muon 使用 **Newton-Schulz 迭代**高效近似 $\text{msign}(M)$。

迭代公式

Newton-Schulz 迭代通过多项式逼近矩阵符号函数：

$$X_{k+1} = aX_k + b(X_k X_k^\top)X_k + c(X_k X_k^\top)^2 X_k \tag{180}$$

其中 $a = 3.4445$，$b = -4.7750$，$c = 2.0315$ 是优化过的系数。

为什么 Newton-Schulz 迭代有效?

Newton-Schulz 迭代的本质是用多项式逼近函数 $f(x) = \text{sign}(x) = x/|x|$。对于矩阵，这变成逐奇异值操作：$f(\sigma_i) = 1$（将所有正奇异值映射到 1）。

设 $X_0 = M/\|M\|_F$（归一化使奇异值落入 $(0, 1]$），迭代后：

$$X_k = U \cdot \text{diag}(\varphi^{(k)}(\sigma_1/\|M\|_F), \ldots) \cdot V^\top \tag{181}$$

其中 $\varphi(x) = ax + bx^3 + cx^5$ 是设计好的五次多项式。

多项式设计的直觉　目标是让 $\varphi^{(k)}(x) \to 1$ 对所有 $x \in (0, 1]$ 快速收敛。系数 a, b, c 的选择使得：

- $\varphi(1) = 1$（不动点）
- $\varphi'(1) = 0$（在不动点处导数为 0，加速收敛）
- 对于较小的 x，$\varphi(x) > x$（将小值推向 1）

实践中 **5 次迭代**即可达到足够精度。更少迭代会欠近似，更多迭代无显著收益但增加计算成本。

实现代码

```python
def newton_schulz5(G, steps=5, eps=1e-7):
    """近似计算 msign(G)"""
    a, b, c = (3.4445, -4.7750, 2.0315)
    X = G.bfloat16()
    X = X / (X.norm() + eps)  # 归一化

    # 确保 d_out <= d_in（转置处理）
    transpose = G.size(0) > G.size(1)
    if transpose:
        X = X.T

    for _ in range(steps):
        A = X @ X.T
        B = b * A + c * A @ A
        X = a * X + B @ X

    if transpose:
        X = X.T
    return X
```

Listing 2: Newton-Schulz 迭代实现

计算复杂度：每次迭代需要 $O(d_{out} \cdot d_{in} \cdot \min(d_{out}, d_{in}))$ 的矩阵乘法，5 次迭代的总开销约为 $5m/B$（m 为模型维度，B 为 batch token 数），通常 $< 1\%$。

15.4　Muon 的四个版本

苏剑林在 [**?**] 中总结了 Muon 的四个主要变体，它们的唯一区别是 msign 前的**缩放因子**：

表 62: Muon 四个版本的缩放因子

版本	缩放因子	特点
朴素版	1	最简单，但学习率不可迁移
KellerJordan 版	$\sqrt{\max(1, d_{out}/d_{in})}$	默认版本，Keras 实现
MuP 版	$\sqrt{d_{out}/d_{in}}$	学习率可迁移
Moonlight 版	$0.2 \times \sqrt{\max(d_{out}, d_{in})}$	可直接沿用 Adam 学习率

为什么需要缩放?　不同形状的矩阵，$\text{msign}(M)$ 的 Frobenius 范数不同（等于 $\min(d_{out}, d_{in})$）。缩放因子用于：

1. 使不同层的更新幅度匹配
2. 与 AdamW 的更新尺度对齐
3. 实现学习率的跨模型迁移

实践建议

- **Moonlight 版**：可直接沿用 Adam 的学习率
- **其他版本**：需将 Adam 学习率乘以约 $0.2 \times \sqrt{d_{hidden}}$

15.5 d_{in} 和 d_{out} 的识别

不同深度学习框架对线性层的实现方式不同，需要正确识别 d_{in} 和 d_{out}：

PyTorch 使用 $y = xW^\top + b$，即 $W \in \mathbb{R}^{d_{out} \times d_{in}}$：

```
d_out, d_in = W.shape[0], W.shape[1]
```

Keras/TensorFlow 使用 $y = xW + b$，即 $W \in \mathbb{R}^{d_{in} \times d_{out}}$：

```
d_in, d_out = W.shape[0], W.shape[1]
```

注意：搞错 d_{in} 和 d_{out} 会导致缩放因子计算错误，影响训练效果。

15.6 哪些参数使用 Muon?

Muon 专为 **2D 矩阵参数**设计，其他参数仍需使用 AdamW：

表 63: 参数类型与优化器选择

参数类型	优化器	原因
隐藏层 Linear 权重	Muon	核心矩阵参数
Attention 的 W_Q, W_K, W_V, W_O	Muon	矩阵参数
MLP 的 $W_{gate}, W_{up}, W_{down}$	Muon	矩阵参数
Embedding 层	AdamW	本质是查找表
最终分类头	AdamW	与 embedding 对称
LayerNorm 参数	AdamW	1D 向量
Bias	AdamW	1D 向量

15.7 大规模训练：Moonlight

月之暗面（Moonshot AI）在 Moonlight 模型 [**?**] 中验证了 Muon 的大规模可扩展性。

关键技术改进

1. **添加权重衰减**：原始 Muon 没有 weight decay，大规模训练需要加入
2. **参数级缩放**：精确调整每个参数的更新比例（即 Moonlight 版的缩放因子）

分布式训练挑战

Muon 需要完整的梯度矩阵来计算正交化更新，这与现有分布式策略（ZeRO、Megatron 等按元素切分优化器状态）冲突。Moonlight 论文提出了内存最优、通信高效的分布式 Muon 实现。

性能对比

表 64: Muon vs AdamW（Moonlight 实验）

指标	Muon	AdamW
计算效率	$\sim 2\times$	基准
达到相同性能所需 FLOPs	52%	100%
样本效率	1.92×	基准

Moonlight 模型规格：

- 总参数：15.29B（MoE 架构）
- 激活参数：2.24B
- 训练数据：5.7T tokens

15.8 实践指南

何时使用 Muon?

- 大规模预训练：Muon 的效率优势随规模增大
- 对训练效率敏感的场景
- 已有 AdamW 超参数，想进一步优化

超参数设置

- **动量系数**：$\beta = 0.95$（比 Adam 的 0.9 略大）
- **Newton-Schulz 步数**：5 步
- **学习率**：
  - Moonlight 版：直接用 Adam 学习率
  - 其他版本：Adam 学习率 $\times\ 0.2\sqrt{d_{hidden}}$
- **权重衰减**：与 AdamW 相同

注意事项

1. Muon 只用于 2D 矩阵参数，其他参数用 AdamW
2. 正确识别框架中的 d_{in} 和 d_{out}
3. 对于 Attention，建议对 W_Q, W_K, W_V 分别做 Newton-Schulz，而非合并
4. bfloat16 精度足够，无需 float32

Remark 15.2 (从向量到矩阵的本质飞跃)**.** 苏剑林评价 *[?]*: *"Muon* 相比 *Adam* 具有更优雅的设计，体现了从向量到矩阵的本质飞跃。*"Adam* 将矩阵展平为向量处理，而 *Muon* 真正利用了矩阵的几何结构。这种 *"*非 *Element-wise"* 的优化范式可能代表了未来优化器的发展方向。

Part VI

评测与 Benchmark

16 评测与 Benchmark

大语言模型的评测是一个复杂且快速演进的领域。本节系统介绍主流评测基准，重点关注 2024 年以来顶级模型（DeepSeek-V3、LLaMA 3、Qwen 2.5、GPT-4o、Claude 3.5 等）普遍采用的 benchmark。

16.1 评测体系概述

为什么需要多维度评测

单一 benchmark 无法全面反映模型能力：

- **能力多样性**：知识、推理、代码、指令遵循等维度独立
- **数据污染**：训练数据可能包含测试集，导致虚高
- **评测饱和**：旧 benchmark 被刷榜，区分度下降

现代评测框架

主流模型发布时通常报告以下类别的 benchmark：

表 65: 评测维度与代表性 Benchmark

维度	核心 Benchmark	备注
知识与理解	MMLU, MMLU-Pro, C-Eval	多学科知识
推理能力	GPQA, ARC-C, BBH	复杂推理
数学能力	GSM8K, MATH-500, AIME	从小学到竞赛
代码能力	HumanEval, LiveCodeBench	代码生成与执行
指令遵循	IFEval, MT-Bench	指令理解与执行
长上下文	RULER, LongBench	长文本处理
多语言	MGSM, C-Eval	非英语能力
安全对齐	TruthfulQA, BBQ	真实性与偏见

16.2 知识与理解

MMLU (Massive Multitask Language Understanding)

MMLU 是最广泛使用的知识评测基准，覆盖 57 个学科：

- **规模**：约 14,000 道四选一题目
- **学科**：STEM、人文、社科、其他
- **难度**：从高中到研究生水平

评测方式

- Zero-shot 或 Few-shot（通常 5-shot）
- 计算模型对 A/B/C/D 选项的概率
- 报告整体准确率和各学科准确率

当前水平

MMLU-Pro

MMLU 的升级版，解决原版的问题：

- **更多选项**：从 4 选 1 变为 10 选 1，降低猜测收益
- **更难题目**：过滤简单题，保留需要推理的题目
- **减少噪声**：修正原 MMLU 中的错误标注

表 66: MMLU 性能对比（5-shot）

模型	MMLU	发布时间
GPT-4o	88.7%	2024.05
Claude 3.5 Sonnet	88.7%	2024.06
DeepSeek-V3	88.5%	2024.12
Qwen2.5-72B	86.1%	2024.09
LLaMA 3.1-405B	88.6%	2024.07
LLaMA 3.1-70B	86.0%	2024.07

区分度更强

- MMLU 上 GPT-4 与 Claude 差距约 1%
- MMLU-Pro 上差距扩大到 5-10%
- 更能反映真实能力差异

GPQA (Graduate-Level Google-Proof QA)

针对研究生水平的专业问题：

- **来源**：物理、化学、生物领域的博士生出题
- **特点**：问题设计为 "Google-proof"，搜索引擎难以直接找到答案
- **难度**：领域专家准确率约 65%，非专家约 30%

GPQA-Diamond 是其中最难的子集，是区分顶级模型的关键 benchmark。

表 67: GPQA-Diamond 性能对比

模型	GPQA-Diamond
DeepSeek-R1	71.5%
o1-preview	73.3%
DeepSeek-V3	59.1%
Claude 3.5 Sonnet	59.4%
GPT-4o	53.6%
LLaMA 3.1-405B	51.1%
Qwen2.5-72B	49.0%

16.3 推理能力

BBH (BIG-Bench Hard)

BIG-Bench 中最具挑战性的 23 个任务：

- **任务类型**：逻辑推理、因果判断、算法执行等
- **特点**：之前模型表现接近随机
- **评测**：通常使用 Chain-of-Thought prompting

ARC (AI2 Reasoning Challenge)

科学推理问题：

- **ARC-Easy**：简单科学问题
- **ARC-Challenge**：需要多步推理的难题
- **来源**：美国 3-9 年级科学考试

HellaSwag

常识推理与句子补全：

- 给定场景描述，选择最合理的后续
- 测试模型的常识理解能力
- 当前顶级模型准确率 >95%，区分度下降

16.4 数学能力

GSM8K

小学数学应用题：

- **规模**：8,500 道题目
- **难度**：2-8 步推理
- **特点**：需要理解题意并进行多步计算

当前顶级模型准确率 >95%，已接近饱和。

MATH

竞赛级数学问题：

- **来源**：AMC、AIME 等数学竞赛
- **难度分级**：Level 1-5，Level 5 最难
- **领域**：代数、几何、数论、概率等

MATH-500 从 MATH 数据集中精选的 500 道高难度题目，是当前主流评测标准。

表 68: 数学 Benchmark 性能对比

模型	GSM8K	MATH-500	AIME 2024
o1	96.4%	96.4%	74% (44.6/60)
DeepSeek-R1	97.3%	97.3%	79.8% (47.9/60)
DeepSeek-V3	91.1%	90.2%	39.2%
Claude 3.5 Sonnet	96.4%	78.3%	-
GPT-4o	95.8%	76.6%	-
Qwen2.5-72B	95.8%	83.1%	-

AIME (American Invitational Mathematics Examination)

美国数学邀请赛：

- 15 道填空题，每题答案为 0-999 的整数
- 代表高中竞赛最高水平
- 是区分推理模型（o1、R1）与普通模型的关键 benchmark

16.5 代码能力

HumanEval

Python 函数生成：

- **规模**：164 道题目
- **形式**：给定函数签名和 docstring，生成实现
- **评测**：Pass@k（k 次采样至少一次通过）

HumanEval+ 增加更多测试用例，减少假阳性。

MBPP (Mostly Basic Python Problems)

更大规模的 Python 编程问题：

- **规模**：974 道题目
- **难度**：相对简单，入门级编程
- **用途**：与 HumanEval 互补

LiveCodeBench

2024 年最重要的代码评测创新，解决数据污染问题：

- **持续更新**：从 LeetCode、AtCoder、CodeForces 持续收集新题
- **时间标记**：每道题有发布日期，可验证是否在训练数据截止日期之后
- **多维度**：代码生成、自我修复、测试输出预测

为什么 LiveCodeBench 重要

- HumanEval 已被“刷榜”，很多模型在训练数据中见过
- LiveCodeBench 的新题确保公平评测
- 是当前评估代码能力的金标准

SWE-bench

软件工程真实任务：

- **任务**：修复 GitHub 上真实的 issue
- **形式**：给定代码仓库和 issue 描述，生成 patch
- **难度**：需要理解大型代码库，非常具有挑战性

表 69: 代码 Benchmark 性能对比

模型	HumanEval	LiveCodeBench	SWE-bench Verified
Claude 3.5 Sonnet	92.0%	41.4%	50.8%
DeepSeek-V3	82.6%	40.5%	42.0%
GPT-4o	90.2%	34.2%	38.4%
o1-preview	92.4%	-	41.3%
Qwen2.5-72B	86.6%	32.8%	-

16.6 指令遵循

IFEval (Instruction Following Evaluation)

测试模型严格遵循指令的能力：

- **规模**：500+ 条带约束的指令
- **约束类型**：
  - 长度约束：“写超过 400 字”
  - 格式约束：“用 JSON 格式输出”
  - 内容约束：“至少提到 3 次 AI”
  - 结构约束：“分成 5 个段落”
- **评测**：约束是否被满足（可程序验证）

两种指标

- **Prompt-level**：整个 prompt 的所有约束都满足
- **Instruction-level**：单个约束的满足率

IFEval 是 Open LLM Leaderboard 的核心 benchmark 之一。

MT-Bench

多轮对话评测：

- **形式**：80 个两轮对话
- **评分**：GPT-4 作为评判，1-10 分
- **类别**：写作、角色扮演、推理、数学等 8 类

AlpacaEval

单轮指令跟随：

- 与 GPT-4 的输出进行比较
- 报告 Win Rate（胜率）
- AlpacaEval 2.0 使用更严格的评判标准

Arena-Hard

基于 Chatbot Arena 的困难子集：

- 从真实用户对话中筛选的 500 道困难问题
- GPT-4-Turbo 作为评判
- 与 Chatbot Arena 排名高度相关

16.7 长上下文评测

RULER

长上下文能力的系统评测：

- **任务类型**：
  - Needle-in-a-Haystack：在长文本中找到特定信息
  - Multi-hop QA：需要整合多处信息
  - Aggregation：统计或汇总信息
- **长度范围**：4K 到 128K+
- **评测**：不同长度下的准确率衰减曲线

LongBench

多任务长文本评测：

- 6 大类 21 个任务
- 平均长度约 15K tokens
- 涵盖单文档/多文档 QA、摘要、代码补全等

Needle-in-a-Haystack

最简单但直观的长上下文测试：

- 在长文本的随机位置插入一个“针”（关键信息）
- 测试模型能否准确检索
- 生成位置-长度的热力图

16.8 多语言评测

C-Eval / CMMLU

中文知识评测：

- **C-Eval**：52 个学科，覆盖中国教育体系
- **CMMLU**：中文版 MMLU
- 是评估中文能力的核心 benchmark

MGSM (Multilingual GSM)

多语言数学推理：

- GSM8K 翻译成 10 种语言
- 测试非英语数学推理能力
- 揭示模型的语言偏差

16.9 安全与对齐

TruthfulQA

测试模型是否会生成虚假但常见的错误信息：

- 817 个问题，涵盖常见误解
- 人类因为偏见经常答错的问题
- 测试模型是否学习了人类的错误信念

SimpleQA

事实准确性评测（OpenAI 2024 发布）：

- 简单的事实性问题
- 测试模型是否会 “幻觉” 错误信息
- 评估拒绝回答（“我不知道”）的能力

16.10 综合评测平台

Open LLM Leaderboard

Hugging Face 维护的开放评测平台：

- **当前版本（v2）包含**：
  - IFEval（指令遵循）
  - BBH（复杂推理）
  - MATH Level 5（高难度数学）
  - GPQA（研究生水平 QA）
  - MuSR（多步推理）
  - MMLU-Pro（知识理解）
- 任何人可以提交模型评测
- 透明、可复现

Chatbot Arena

基于真实用户投票的评测：

- 用户盲评两个模型的回复

- 使用 ELO 排名系统
- 被认为是最能反映真实用户偏好的评测
- 但难以控制变量，不够 “科学”

LiveBench

抗污染的动态评测：

- 每月更新题目
- 严格的时间控制防止数据污染
- 涵盖数学、代码、推理、语言等多个维度

16.11 评测最佳实践

避免数据污染

- **使用新 benchmark**：LiveCodeBench、LiveBench 等持续更新
- **时间切分**：确保评测数据晚于训练数据截止日期
- **多源验证**：同一能力用多个 benchmark 交叉验证

评测配置标准化

- 明确报告 Few-shot 数量
- 统一 prompt 模板
- 使用相同的解码参数（temperature、top_p 等）

选择合适的 Benchmark

表 70: 评测场景与推荐 Benchmark

评测目标	推荐 Benchmark
通用能力快速评估	MMLU-Pro, GPQA-Diamond
数学推理	MATH-500, AIME
代码生成	LiveCodeBench, SWE-bench
指令遵循	IFEval
长上下文	RULER, Needle-in-Haystack
中文能力	C-Eval, CMMLU
真实用户偏好	Chatbot Arena, Arena-Hard
安全性	TruthfulQA, SimpleQA

Remark 16.1 (评测的局限性)**.**

- ***Benchmark* 真实能力**：高分不代表实际应用效果好
- **优化目标错位**：过度优化 *benchmark* 可能损害通用能力
- **评测演进**：*benchmark* 会饱和，需要持续更新
- **人类评估**：某些能力（创意、共情）难以自动评测

Part VII

部署优化

17 模型量化

模型量化是将神经网络中的浮点数表示转换为低精度表示的技术，是大语言模型高效部署的核心技术之一。本节从基础理论到 LLM 专用方法，系统介绍量化技术。

17.1 量化基础

为什么需要量化

现代 LLM 的参数规模带来严峻的部署挑战：

- **内存需求**：70B 参数模型以 FP16 存储需要 140GB 显存
- **带宽瓶颈**：推理时主要受限于内存带宽而非计算
- **能耗成本**：数据移动消耗的能量远超计算本身

量化通过降低数值精度来解决这些问题：

表 71: 不同精度的存储与计算特性

精度	位宽	相对内存	典型用途
FP32	32	1×	训练梯度累积
FP16/BF16	16	0.5×	标准训练与推理
FP8	8	0.25×	高效训练（Hopper+）
INT8	8	0.25×	量化推理
INT4	4	0.125×	激进量化推理

量化的数学定义

均匀量化（Uniform Quantization）是最常用的量化方式。给定浮点数 x，量化过程为：

$$Q(x) = \text{clamp}\left(\left\lfloor \frac{x}{s} \right\rceil + z, 0, 2^b - 1\right) \tag{182}$$

其中 s 是缩放因子（scale），z 是零点（zero-point），b 是目标位宽，$\lfloor \cdot \rceil$ 表示四舍五入。

反量化（Dequantization）恢复近似值：

$$\hat{x} = s \cdot (Q(x) - z) \tag{183}$$

对称量化 vs 非对称量化 **对称量化**：$z = 0$，量化范围关于零点对称：

$$s = \frac{\max(|x|)}{2^{b-1} - 1} \tag{184}$$

非对称量化：允许 $z \neq 0$，更好地适应非对称分布：

$$s = \frac{\max(x) - \min(x)}{2^b - 1} \tag{185}$$

$$z = \left\lfloor -\frac{\min(x)}{s} \right\rceil \tag{186}$$

对称量化实现更简单（无需存储 z），但非对称量化对偏斜分布更有效。

量化粒度

量化参数 s, z 的计算粒度影响精度与开销的权衡：

- **Per-Tensor**：整个张量共享一组 (s, z)，开销最小但精度损失大
- **Per-Channel**：每个输出通道独立量化，常用于权重

- **Per-Token**：每个 token 独立量化，常用于激活值
- **Per-Group**：将通道分组，每组独立量化，精度与开销的折中

Example 17.1 (Group Quantization)**.** *对于权重矩阵 $W \in \mathbb{R}^{m\times n}$，设 group size 为 g，则需要存储 $\lceil n/g \rceil$ 组量化参数。常见设置 $g = 128$，在 INT4 量化时有效位宽约为 $4 + 32/128 = 4.25$ 位。*

17.2 训练后量化（PTQ）

训练后量化（Post-Training Quantization）在模型训练完成后进行，无需重新训练，是 LLM 量化的主流方法。

基本 PTQ 流程

1. **校准**（Calibration）：使用少量代表性数据（通常几百到几千样本）统计激活值分布
2. **确定量化参数**：根据统计信息计算 s, z
3. **量化权重**：将浮点权重转换为低精度表示
4. **（可选）校正**：通过额外优化减少量化误差

校准策略

MinMax 校准　最简单的方法，使用观测到的最大最小值：

$$s = \frac{\max(x) - \min(x)}{2^b - 1} \tag{187}$$

问题：对离群值敏感。

百分位校准　使用第 p 和 $100 - p$ 百分位数代替最大最小值，p 通常取 0.1-1%。

MSE 校准　最小化量化误差：

$$s^* = \arg\min_s \mathbb{E}\left[(x - \hat{x})^2\right] \tag{188}$$

KL 散度校准　最小化原始分布与量化分布的 KL 散度，由 TensorRT 采用。

权重量化 vs 激活量化

权重量化（W-only）：

- 权重在推理前确定，可以离线计算量化参数
- 分布通常接近高斯，易于量化
- 常见配置：W8A16、W4A16

激活量化：

- 激活值依赖输入，需要动态量化或校准
- LLM 中存在大量**离群值**（outliers），极大增加量化难度
- 常见配置：W8A8、W4A8

17.3 量化感知训练（QAT）

量化感知训练（Quantization-Aware Training）在训练过程中模拟量化效果，让模型学会适应低精度表示。

直通估计器（STE）

量化操作 $Q(x)$ 是阶梯函数，梯度几乎处处为零。**直通估计器**（Straight-Through Estimator）绕过这一问题：

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial Q(x)} \tag{189}$$

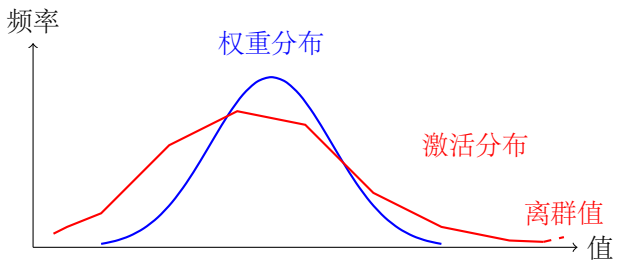

图 12: 权重与激活值的典型分布对比

即前向传播使用量化值，反向传播直接传递梯度。

伪量化

QAT 的核心是在前向传播中插入**伪量化**（Fake Quantization）操作：

$$\hat{x} = s \cdot \left(\text{clamp}\left(\left\lfloor \frac{x}{s} \right\rceil, q_{min}, q_{max} \right) \right) \tag{190}$$

伪量化保持浮点计算，但模拟量化的离散化效果。

QAT vs PTQ

表 72: QAT 与 PTQ 对比

特性	PTQ	QAT
计算成本	低（分钟级）	高（需重新训练）
数据需求	少量校准数据	完整训练数据
精度损失	较大（尤其低位宽）	较小
适用场景	快速部署、8 位量化	极低位宽、精度敏感

对于 LLM，QAT 的计算成本通常不可接受，因此 PTQ 是主流选择。

17.4 LLM 量化的挑战与方法

激活值离群值问题

LLM 的一个关键特性是激活值中存在**离群值**（outliers）：极少数通道包含数值远大于其他通道的激活值。这些离群值：

- 出现在特定通道，跨 token 一致
- 数值可达正常值的 100 倍以上
- 移除这些通道会导致模型性能崩溃

标准量化方法被迫扩大量化范围以覆盖离群值，导致正常值的量化精度严重下降。

SmoothQuant

SmoothQuant [**?**] 是解决激活离群值问题的突破性方法，实现了 LLM 的 W8A8 量化。

核心思想 将量化难度从激活 “迁移” 到权重。观察到离群值出现在固定通道，可以通过 per-channel 缩放来平滑激活：

$$Y = (X \cdot \text{diag}(s)^{-1}) \cdot (\text{diag}(s) \cdot W) = \hat{X}\hat{W} \tag{191}$$

其中 s 是迁移因子。$\hat{X}$ 的分布更均匀，更易量化；$\hat{W}$ 吸收了部分难度，但权重本身分布良好，影响有限。

迁移因子选择

$$s_j = \frac{\max(|X_j|)^\alpha}{\max(|W_j|)^{1-\alpha}} \tag{192}$$

其中 $\alpha \in [0, 1]$ 控制迁移程度。$\alpha = 0.5$ 对大多数模型效果良好。

性能 SmoothQuant 在 OPT-175B 上实现了 INT8 量化，精度损失小于 0.5%，推理速度提升 1.5 倍。

GPTQ

GPTQ [**?**] 是基于二阶信息的权重量化方法，可将 LLM 压缩至 4 位精度。

问题形式化 逐层优化，最小化量化后的输出误差：

$$\arg\min_{\hat{W}} \|WX - \hat{W}X\|_2^2 \tag{193}$$

基于 Hessian 的优化 利用 Optimal Brain Quantization (OBQ) 的思想，但做了关键优化使其适用于 LLM 规模：

1. 以固定顺序（而非贪心顺序）量化权重列
2. 批量处理多列，使用高效的矩阵运算
3. 利用 Cholesky 分解高效更新 Hessian 逆

性能

- 175B 参数模型可在单 GPU 上 4 小时内完成量化
- 3-4 位量化精度损失极小（困惑度增加 < 0.5）
- 广泛应用于开源 LLM 的量化分发

AWQ

AWQ（Activation-aware Weight Quantization） [**?**] 基于一个关键观察：不同权重通道的重要性差异巨大。

核心观察 对于矩阵乘法 $Y = XW$，如果激活值 X 的某些通道数值较大，则对应权重通道的量化误差影响更大。

方法 不均匀地保护重要权重通道：

$$s_j = \sqrt{\frac{\max(|X_j|)}{1 + \epsilon}} \tag{194}$$

通过 per-channel 缩放 $W'_j = W_j \cdot s_j$ 放大重要通道，量化后再除以 s_j 恢复。

性能 在 4 位量化下，AWQ 的困惑度通常优于 GPTQ，且支持更高效的实时推理。

GGUF/GGML

GGUF（GPT-Generated Unified Format）是 llama.cpp 项目定义的模型存储格式，广泛用于本地 LLM 部署。

特点

- 支持多种量化格式：Q4_0, Q4_K_M, Q5_K_M, Q8_0 等
- “K” 表示使用 k-quant 方法，对重要层使用更高精度
- 针对 CPU 推理优化，支持 Apple Metal、CUDA 等后端

常用配置

表 73: GGUF 量化格式对比

格式	有效位宽	精度损失
Q8_0	8.5 bits	极小
Q5_K_M	5.5 bits	小
Q4_K_M	4.8 bits	中等
Q4_0	4.5 bits	较大
Q2_K	3.4 bits	大

17.5 FP8 量化

FP8 是一种 8 位浮点格式，相比 INT8 保留了动态范围，成为现代 GPU 训练和推理的重要选择。

FP8 格式

两种主要格式：

- **E4M3**：4 位指数、3 位尾数，动态范围更大，适合前向传播
- **E5M2**：5 位指数、2 位尾数，精度更高，适合梯度

表 74: FP8 格式对比

格式	动态范围	最大值	最小正数
E4M3	2^{15}	448	2^{-9}
E5M2	2^{31}	57344	2^{-16}
FP16	2^{31}	65504	2^{-24}

FP8 训练

NVIDIA Hopper 和 Blackwell 架构原生支持 FP8 Tensor Core。DeepSeek-V3 [**?**] 展示了 FP8 训练的工业应用：

混合精度策略

- 权重和激活以 FP8 存储和计算
- 累加器使用 FP32 保持精度
- 主权重副本保持 FP32/BF16 用于优化器更新

Block-wise 量化 将张量分块独立量化，每块 128×128 元素共享一个缩放因子，平衡精度与开销。

性能收益

- 内存带宽需求减半
- H100 上 FP8 吞吐量是 FP16 的 2 倍
- DeepSeek-V3 训练仅需 2.788M H800 GPU 小时

FP8 推理

FlashAttention-3 支持 FP8 注意力计算：

- 使用 Block Quantization 减少量化误差
- Incoherent Processing（基于 Hadamard 变换）进一步提升精度
- 接近 1.2 PFLOPS 的吞吐量

17.6 极低位宽量化

INT4/W4A16

4 位权重量化是目前实用的最低位宽：

- 模型大小减少 4 倍
- 推理速度提升 2-4 倍（受限于内存带宽）
- 精度损失可控（困惑度增加通常 < 1）

关键技术

- Group quantization（组大小 128）减少误差
- 对敏感层（如第一层、最后一层）使用更高精度
- GPTQ/AWQ 的二阶优化

BitNet 与 1.58-bit 量化

BitNet 1.58 [**?**] 提出了极端量化：权重仅取 $\{-1, 0, +1\}$ 三个值。

量化函数

$$W_q = \text{RoundClip}\left(\frac{W}{\gamma}, -1, 1\right) \tag{195}$$

其中 $\gamma = \frac{1}{nm}\sum_{ij}|W_{ij}|$ 是平均绝对值。

计算优势 矩阵乘法退化为加减运算，无需乘法器：

$$Y = X \cdot W_q = \sum_j X_j \cdot W_{qj} = \sum_{j:W_{qj}=1} X_j - \sum_{j:W_{qj}=-1} X_j \tag{196}$$

当前状态

- 需要从头训练，不适用于已有模型
- 在同等模型容量下，精度接近 FP16 模型
- 能效提升 71 倍（BitNet b1.58 2B4T 模型）
- 尚未在超大规模模型上验证

17.7 KV Cache 量化

长上下文推理的内存瓶颈主要来自 KV Cache 而非模型权重。KV Cache 量化成为长上下文场景的关键技术。

KV Cache 的内存需求

对于序列长度 L、batch 大小 B、层数 N_L、注意力头数 N_H、头维度 d：

$$\text{KV Cache 大小} = 2 \times B \times L \times N_L \times N_H \times d \times \text{sizeof(dtype)} \tag{197}$$

以 LLaMA-70B（80 层、64 头、128 维）为例：

- 100K 上下文、batch=1、FP16：约 40GB
- 同等设置使用 INT4：约 10GB

KIVI 方法

KIVI [**?**] 是 KV Cache 量化的代表性工作：

非对称量化 Key 和 Value 有不同的分布特性，分别使用 per-channel 和 per-token 量化：

- Key：per-channel 量化（通道间差异大）
- Value：per-token 量化（token 间差异大）

2-bit 量化 KIVI 可将 KV Cache 压缩至 2-bit，内存减少 8 倍，困惑度增加极小（< 0.1）。

实践建议

- 中等上下文（<32K）：4-bit KV Cache 足够
- 长上下文（32K-128K）：考虑 2-bit 量化
- 超长上下文（>128K）：结合 KV Cache 压缩（如 H2O、StreamingLLM）

17.8 量化的系统支持

硬件支持

不同硬件对量化精度的支持差异显著：

表 75: GPU 架构的量化支持

架构	INT8	FP8	INT4
Ampere (A100)	Tensor Core	无	软件模拟
Hopper (H100)	Tensor Core	E4M3/E5M2	软件模拟
Blackwell (B200)	Tensor Core	增强	FP4 原生

推理框架

主流推理框架的量化支持：

- **vLLM**：支持 GPTQ、AWQ、FP8、SqueezeLLM
- **TensorRT-LLM**：INT8/INT4 weight-only、FP8、SmoothQuant
- **llama.cpp**：GGUF 格式，支持 Q2-Q8 多种精度
- **ExLlama**：专注于 4-bit 量化的高效实现

17.9 量化最佳实践

精度选择

- 内存充足：FP16/BF16，无精度损失
- 一般部署：INT8 或 FP8，精度损失极小
- 边缘设备：INT4（GPTQ/AWQ），精度损失可接受
- 极限压缩：2-3 bit，需仔细评估任务影响

量化方法选择

- 快速部署：直接使用预量化模型（HuggingFace 上大量可用）
- 自定义量化：AWQ 通常是最佳起点
- 极致压缩：GPTQ 可达到更低位宽
- 激活量化需求：SmoothQuant 处理离群值

评估指标

- **困惑度**：最基础的质量指标

- **下游任务**：MMLU、GSM8K 等 benchmark
- **推理速度**：tokens/s
- **内存占用**：峰值显存需求

Remark 17.1 (量化的 trade-off)**.** 量化不是免费的午餐。在选择量化策略时需权衡：

- 精度 *vs* 效率：位宽越低，压缩率越高，但精度损失也越大
- 量化时间 *vs* 推理性能：复杂量化方法（如 *GPTQ*）需要更长校准时间
- 通用性 *vs* 专用性：针对特定任务微调的模型可能需要重新量化

当前 *4-bit* 量化（*GPTQ/AWQ*）是精度与效率的良好平衡点，*8-bit* 量化（*INT8/FP8*）则几乎无精度损失。

18 LLM 推理优化

LLM 推理面临独特的挑战：模型参数量巨大、自回归生成逐 token 进行、KV Cache 随序列长度增长。本章介绍主流的推理优化技术和推理引擎。

18.1 推理基础

两阶段推理

LLM 推理分为两个阶段：

Prefill 阶段 处理输入 prompt 的所有 token，生成 KV Cache 和第一个输出 token：

- **计算特性**：并行处理所有输入 token，计算密集型（Compute-bound）
- **瓶颈**：矩阵乘法的计算量
- **指标**：Time To First Token（TTFT）

Decode 阶段 逐个生成后续 token，每步只处理 1 个新 token：

- **计算特性**：自回归生成，内存带宽受限（Memory-bound）
- **瓶颈**：加载模型参数和 KV Cache 的内存带宽
- **指标**：Inter-Token Latency（ITL）或 Tokens Per Second（TPS）

表 76: Prefill vs Decode 对比

特性	Prefill	Decode
Token 数	N（输入长度）	1
计算模式	并行	串行
瓶颈	计算	内存带宽
GPU 利用率	高	低

KV Cache

KV Cache 是推理优化的核心。每个 Transformer 层在处理 token 时会生成 Key 和 Value 向量，这些向量在后续生成中被重复使用。

KV Cache 大小

$$\text{KV Cache} = 2 \times L \times N \times h_{kv} \times d \times \text{bytes} \tag{198}$$

其中 L 是层数，N 是序列长度，h_{kv} 是 KV 头数，d 是头维度。

示例：LLaMA-13B，序列长度 8K，BF16 精度：

$$2 \times 40 \times 8192 \times 40 \times 128 \times 2 = 6.7\text{GB} \tag{199}$$

单个请求的 KV Cache 就需要 6.7GB，成为主要的显存瓶颈。

KV Cache 优化方向

1. **减少 KV 头数**：GQA/MQA（见第9节）
2. **量化**：INT8/FP8 压缩 KV Cache
3. **稀疏化**：只保留重要的 KV 对
4. **共享**：跨层共享 KV Cache

18.2 批处理策略

静态批处理

传统方法：等待凑齐一批请求后统一处理。

问题：

- 请求长度不一，短请求需等待长请求完成
- 填充（Padding）浪费计算资源
- 吞吐量低，延迟高

Continuous Batching

Continuous Batching [**?**] 动态管理请求：

- 请求完成后立即释放资源，新请求立即加入
- 无需等待整批完成
- 迭代级别调度，而非请求级别

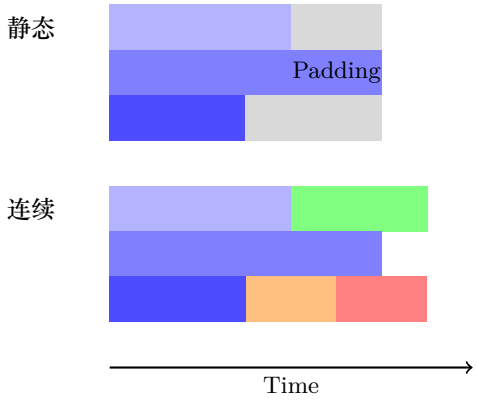

图 13: 静态批处理 vs Continuous Batching

Chunked Prefill

长 prompt 的 Prefill 会阻塞 Decode 请求。Chunked Prefill [**?**] 将 Prefill 分块：

- 长 Prefill 拆分为多个小块
- 每个块与 Decode 请求混合执行
- 减少 Decode 请求的排队延迟

Prefill-Decode 分离

DistServe [**?**] 提出将 Prefill 和 Decode 部署到不同的 GPU：

- **Prefill 服务器**：计算密集，可用更高的张量并行度
- **Decode 服务器**：内存密集，优化批处理大小
- **KV Cache 传输**：通过 NVLink/PCIe 在服务器间传递

优势：

- Prefill 不会阻塞 Decode，降低 ITL
- 可独立扩展两种资源
- 针对不同阶段优化并行策略

18.3 PagedAttention 与 vLLM

vLLM [**?**] 引入 PagedAttention，借鉴操作系统虚拟内存的思想管理 KV Cache。

传统 KV Cache 的问题

传统实现预分配连续内存存储 KV Cache：

- 按最大序列长度预分配，造成**内存浪费**
- 不同请求长度不一，产生**碎片化**
- 无法动态扩展，限制**并发请求数**

PagedAttention 原理

PagedAttention 将 KV Cache 分成固定大小的 **Page**（块）：

- 每个 Page 存储固定数量 token 的 KV
- Page 可以非连续存储（类似虚拟内存）
- 按需分配，用完即释放

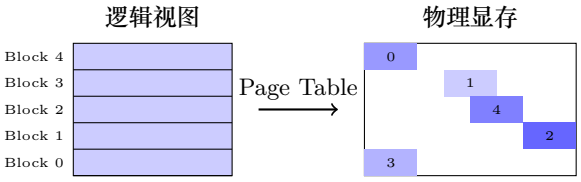

图 14: PagedAttention：非连续 KV Cache 存储

vLLM 性能

vLLM 结合 PagedAttention 和 Continuous Batching：

- 相比 HuggingFace Transformers，吞吐量提升 **24 倍**
- 显存利用率接近 100%（无碎片）
- 支持更大的并发批处理

18.4 Prefix Caching 与 SGLang

Prefix Caching 动机

许多应用场景存在**共享前缀**：

- 多轮对话：System Prompt 相同
- Few-shot 学习：示例相同
- 批量处理：相同的指令模板

重复计算相同前缀的 KV Cache 造成浪费。

RadixAttention

SGLang [**?**] 提出 RadixAttention，使用**基数树**（Radix Tree）管理 KV Cache：

- 树的每条边对应一段 token 序列
- 共享前缀的请求共享 KV Cache
- LRU 策略管理缓存淘汰

优势：

- 自动检测和复用共享前缀

- 相比 vLLM，吞吐量提升可达 **5-6 倍**
- 支持复杂的 LLM 程序（多次调用、分支）

SGLang 特性

SGLang 是目前最快的开源推理引擎之一：

- **RadixAttention**：自动 Prefix Caching
- **结构化输出**：压缩有限状态机加速 JSON 生成
- **投机解码**：支持 EAGLE 等方法
- **多模态**：支持 Vision-Language 模型

18.5 投机解码

投机解码（Speculative Decoding）[**? ?**] 是加速自回归生成的重要技术，其核心思想是 “先用小模型快速猜测，再用大模型批量验收”。

核心思想与工作流程

自回归生成的瓶颈在于每步只生成 1 个 token，且由于大模型是 Memory-bound 的，输入 1 个 token 和输入 K 个 token 的前向时间几乎相同——计算不是瓶颈，读取权重才是。投机解码正是利用了这一特性。

工作流程

1. **Draft 阶段**：用小模型（Draft Model）自回归生成 K 个候选 token
2. **Verify 阶段**：将这 K 个 token **并行**输入大模型（Target Model），一次前向得到每个位置的 next token 预测
3. **Accept/Reject**：从头开始逐个比对，连续一致的 token 直接接受；一旦不一致，在分歧点由大模型给出正确 token，后续 draft token 全部丢弃
4. 重复上述过程直到生成结束

关键保证：即使 draft 全部猜错，也能从大模型获得至少 1 个正确 token，因此投机解码的下限就是普通自回归解码。

采样场景下的验证机制

上述流程在贪婪解码（temperature=0）下很直观：直接比较 token 是否相等。但在采样场景下，大模型本身的输出也是随机的，如何定义 “猜对”?

设 Draft 模型在位置 t 生成 token x 的分布为 $q(x)$，Target 模型的分布为 $p(x)$。投机解码需要保证**最终输出的分布严格等于** $p(x)$（无偏性），而不仅仅是加速。

接受概率 对于 draft 生成的 token x，以如下概率接受：

$$a(x) = \min\left(1, \frac{p(x)}{q(x)}\right) \tag{200}$$

直觉理解：

- 若 $q(x) \le p(x)$：大模型比小模型更认可这个 token，必然接受
- 若 $q(x) > p(x)$：小模型 “过于自信”，以概率 $p(x)/q(x)$ 接受

拒绝后的重采样：为什么需要残差分布? 一个自然的问题：拒绝后为什么不直接从 $p(x)$ 采样？因为接受分支已经 “预支” 了部分概率质量。

概率账本分析：被接受分支输出 token x 的概率为：

$$\Pr[\text{accept and output } x] = q(x) \cdot a(x) = \min(p(x), q(x)) \tag{201}$$

要使最终分布等于 $p(x)$，拒绝分支必须补齐剩余的概率质量：

$$p(x) - \min(p(x), q(x)) = \max(0, p(x) - q(x)) \tag{202}$$

因此，拒绝时从**残差分布**重采样：

$$p'(x) = \frac{\max(0, p(x) - q(x))}{\sum_{x'} \max(0, p(x') - q(x'))} \tag{203}$$

无偏性的严格证明 记总接受概率为 $\beta = \sum_x \min(p(x), q(x))$，则拒绝概率为 $1 - \beta$。对任意 token x，其最终输出概率为：

$$\Pr[\text{output } x] = \underbrace{\min(p(x), q(x))}_{\text{接受分支}} + \underbrace{(1-\beta) \cdot \frac{\max(0, p(x) - q(x))}{1 - \beta}}_{\text{拒绝分支}} \tag{204}$$

$$= \min(p(x), q(x)) + \max(0, p(x) - q(x)) \tag{205}$$

$$= p(x) \quad \checkmark \tag{206}$$

最后一步利用了恒等式 $\min(a, b) + \max(0, a - b) = a$。这证明了投机解码在采样场景下是**无损**的——输出分布严格等于只用 Target 模型逐 token 采样。

替代方案：全局上界 Rejection Sampling 经典 rejection sampling 的另一种做法是：先计算全局上界 $M = \max_x \frac{p(x)}{q(x)}$，然后以概率 $\frac{p(x)}{M \cdot q(x)}$ 接受。在这种方案下：

- 接受分支得到的是 p/M（缩放版的目标分布）
- 拒绝后可以直接从 p 重采样（因为没有 “预支” 问题）
- 但接受率为 $1/M$，通常远低于残差方案的 $\sum_x \min(p(x), q(x))$

残差采样之所以成为标准做法，正是因为它在保证无偏性的同时，最大化了接受率。

Remark 18.1 (两 token 例子)**.** 设词表 $\{A, B\}$，$p(A) = 0.9, p(B) = 0.1$，$q(A) = 0.5, q(B) = 0.5$。

- A 的接受概率：$\min(1, 0.9/0.5) = 1$，接受分支贡献 0.5
- B 的接受概率：$\min(1, 0.1/0.5) = 0.2$，接受分支贡献 $0.5 \times 0.2 = 0.1$
- 拒绝概率：$1 - 0.5 - 0.1 = 0.4$
- 残差分布：A 剩余 $0.9 - 0.5 = 0.4$，B 剩余 $0.1 - 0.1 = 0$，归一化后 $p'(A) = 1$
- 最终：$A = 0.5 + 0.4 \times 1 = 0.9$，$B = 0.1 + 0.4 \times 0 = 0.1$ ✓

若拒绝后直接从 p 采样，B 会变成 $0.1 + 0.4 \times 0.1 = 0.14$，分布已偏离！

贪婪解码作为特例 当 temperature=0 时，p 和 q 退化为在 argmax 上概率为 1 的 delta 分布。此时验证规则简化为：

- 若 draft token 等于 Target 的 argmax ⇒ 接受（$p(x) = q(x) = 1$）
- 否则 ⇒ 拒绝并由 Target 给出 argmax

这解释了为什么在贪婪解码下，“猜对” 直观上就是 “token 相等”——它是采样验证机制的退化情况。

加速比分析

设 Draft 模型的平均接受率为 α，每次投机 K 个 token：

$$\text{平均接受长度} = \frac{1-\alpha^{K+1}}{1-\alpha} \tag{207}$$

加速比取决于：

- 接受率 α：Draft 与 Target 越接近越好
- 投机长度 K：过大会降低接受率
- Draft 模型速度：需要比 Target 快很多（通常 10-50 倍小）
- 场景特点：batch 较小、交互式低延迟场景收益更明显

EAGLE 系列：从 Chain 到 Tree

EAGLE [**?**] 是目前 SOTA 的投机解码方法，其核心创新是利用 Target 模型的**隐状态**来指导 Draft 生成。

EAGLE-1：Chain 模式 与普通 Draft 模型的区别在于输入：

$$\text{Draft Input} = \text{Embedding}(x_t) + \text{Project}(h_t^{\text{target}}) \tag{208}$$

其中 h_t^{target} 是 Target 模型的隐状态。这使得 Draft 模型能够 “看到”Target 模型的中间表示，显著提高猜测准确率。

Stable KV：验证后，清除 Draft 模型中由自身隐状态生成的 KV Cache，只保留由 Target 隐状态生成的部分，确保下一轮的输入分布与训练时一致。

EAGLE-2：Tree 模式 Chain 模式每次只生成一条候选序列。Tree 模式的改进是：每步生成 top-k 个候选 token，形成树状结构，增加命中概率。

- **Expand 阶段**：每层选 top-k，动态维护累计得分最高的路径
- **Select 阶段**：对所有叶节点按累计得分排序，选 top-K 送入 Target 验证
- **Tree Attention**：需要特殊的 attention mask，确保每个 token 只关注其祖先节点

EAGLE-3：统一训练与推理 EAGLE-1/2 的一个问题是：训练时 Draft 模型看到的全是 Target 隐状态（stable KV），但推理时大部分隐状态来自 Draft 自身（unstable KV）。这种 train-test mismatch 导致：

- 第一个 draft token 命中率随数据量提升
- 第二个及之后的 token 命中率急剧下降（累积误差）

EAGLE-3 的解决方案：

1. 训练时加入 unstable KV 情况，统一 train 与 test
2. 去除隐状态监督损失，只保留 next token prediction
3. 使用多层 Target 隐状态（low/mid/high features）

这些改进使 EAGLE-3 获得了 scaling 性质——更多训练数据带来持续的性能提升。

工业应用现状

投机解码已成为主流推理引擎的标配优化：

- **Google**：官方博客明确提到 speculative decoding 用于加速生成，“used across Google products”
- **NVIDIA TensorRT-LLM**：完整支持 speculative sampling，特定配置下可达 3.6× 吞吐提升
- **vLLM/SGLang**：支持 draft model 和 n-gram 等多种 proposal 方式

表 77: 投机解码方法对比

方法	额外模型	训练需求	加速比	特点
独立 Draft	是	无	2-3×	简单通用
Self-Speculative	否	无	1.5-2×	显存友好
Medusa	否	训练 Head	2-3×	并行预测
EAGLE-1	否	训练 Head	2.5-3×	Chain + 隐状态
EAGLE-2	否	训练 Head	3-4×	Tree 扩展
EAGLE-3	否	训练 Head	4-5×	统一 train/test

- **OpenAI**：API 提供 "Predicted Outputs" 功能，用户提供预测文本，模型统计 accepted/rejected tokens——思路与投机解码一致，只是 proposal 来自用户而非 draft model

Remark 18.2 (适用场景)**.** *投机解码在以下场景收益最明显：*

- *交互式低延迟（QPS 中等、batch 小）：大模型逐 token 生成时 GPU 利用率低*
- *Draft 与 Target 分布接近：验收率高才有稳定收益*
- *生成内容可预测性强：代码补全、格式化输出等*

对于大 batch 高吞吐场景，GPU 利用率本已较高，投机解码的收益会减弱。

18.6 KV Cache 压缩

KV Cache 量化

将 KV Cache 从 FP16/BF16 量化到低精度：

INT8/FP8 量化 vLLM、TensorRT-LLM 支持的标准方法：

- 显存减少 50%
- 精度损失很小
- 硬件原生支持

更激进的量化 KVQuant [**?**] 实现 3-4bit 量化：

- Per-channel 量化 + 离群值处理
- 支持百万级上下文（单卡 A100）
- 1% 离群值保留可保持精度

KIVI [**?**] 实现 2bit 量化：

- Key 用 2bit，Value 用 2bit
- 无需训练，即插即用
- 显存减少约 4 倍

KV Cache 稀疏化

并非所有 KV 对都同等重要，可以选择性保留。

H2O (Heavy-Hitter Oracle) 动态识别重要 token [**?**]：

- 基于注意力分数累积判断重要性
- 保留"Heavy Hitter"token
- 吞吐量提升 40%+

SnapKV 基于观察窗口选择重要 KV [**?**]：

- 每个 Head 只关注部分位置
- 聚类保留关键 KV
- 16K 输入：3.6 倍加速，8.2 倍显存节省

18.7 推理引擎对比

表 78: 主流 LLM 推理引擎对比

引擎	核心技术	Prefix Cache	投机解码	特点
vLLM	PagedAttention	支持	支持	最广泛使用
SGLang	RadixAttention	原生	EAGLE 等	结构化输出快
TensorRT-LLM	深度优化	支持	多种	NVIDIA 官方
llama.cpp	CPU 优化	有限	支持	本地部署

vLLM

vLLM 是目前最流行的开源推理引擎：

- **PagedAttention**：高效 KV Cache 管理
- **Continuous Batching**：动态批处理
- **张量并行**：多 GPU 推理
- **量化**：FP8、AWQ、GPTQ
- **生态**：支持 HuggingFace 模型、OpenAI 兼容 API

SGLang

SGLang 专注于复杂 LLM 应用：

- **RadixAttention**：自动 Prefix Caching
- **Frontend 语言**：简化多次调用的编程
- **结构化输出**：JSON Schema 约束生成
- **性能**：在某些场景比 vLLM 快 5-6 倍

TensorRT-LLM

NVIDIA 官方推理库：

- 深度优化的 CUDA Kernel
- 支持 Hopper 特性（FP8、TMA）
- In-flight Batching
- 与 Triton Inference Server 集成

18.8 推理优化最佳实践

延迟优先场景

- 使用 Prefix Caching（SGLang）
- 投机解码（EAGLE）
- 小批处理 + 高并行度

吞吐优先场景

- Continuous Batching（vLLM）

- 大批处理
- KV Cache 量化

长上下文场景

- KV Cache 量化（KVQuant、KIVI）
- KV Cache 稀疏化（SnapKV、H2O）
- Prefill-Decode 分离

Remark 18.3 (推理优化的权衡)**.** 没有万能方案，需要根据场景选择：

- **交互式应用**：优先 $TTFT$ 和 ITL
- **批处理任务**：优先吞吐量
- **边缘部署**：优先显存效率

Part VIII

前沿应用

19 多模态大模型

随着大语言模型（LLM）在文本理解和生成上取得突破性进展，研究者开始探索如何将视觉、音频等多模态信息与语言能力相结合。多模态大模型（Multimodal Large Language Models, MLLMs）已成为人工智能领域最活跃的研究方向之一。本章将系统介绍多模态大模型的架构设计，重点讨论" 原生多模态"（Native Multimodal）这一新兴范式。

19.1 多模态大模型概述

从单模态到多模态

传统的大语言模型只能处理文本输入和输出。为了让模型具备" 看" 和" 听" 的能力，研究者提出了多种将视觉信息融入语言模型的方法。根据模态融合的深度和方式，多模态大模型可分为以下几类：

- **级联式（Cascaded）**：多个独立模型串联，如先用视觉模型提取描述，再输入语言模型
- **适配器式（Adapter-based）**：在预训练 LLM 基础上添加视觉适配器，如 LLaVA、BLIP-2
- **原生式（Native）**：从头开始在多模态数据上联合训练，如 GPT-4o、Gemini

核心挑战

构建多模态大模型面临几个关键挑战：

模态对齐（Modality Alignment）：图像和文本存在于不同的表示空间，需要建立有效的跨模态映射。图像是连续的像素值，而文本是离散的 token 序列，如何让两者在同一语义空间中对齐是核心问题。

信息压缩：一张 224×224 的图像包含 50176 个像素，而典型的视觉编码器会产生 196-576 个视觉 token。如何在保留关键信息的同时压缩视觉表示，避免对 LLM 造成过大的序列长度负担?

理解与生成的统一：视觉理解（如 VQA）需要高层语义抽象，而图像生成需要细粒度的像素级信息。如何在单一模型中同时支持这两种看似矛盾的需求?

19.2 视觉编码器

视觉编码器是多模态大模型的" 眼睛"，负责将图像转换为语言模型可理解的表示。

Vision Transformer (ViT)

Vision Transformer [**?**] 将 Transformer 架构应用于图像处理。其核心思想是将图像切分为固定大小的 patch，然后像处理文本 token 一样处理这些 patch：

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{E}\mathbf{x}_1; \mathbf{E}\mathbf{x}_2; ...; \mathbf{E}\mathbf{x}_N] + \mathbf{E}_{\text{pos}} \tag{209}$$

其中 $\mathbf{x}_i \in \mathbb{R}^{P^2 \cdot C}$ 是第 i 个图像 patch 的展平向量，$\mathbf{E}$ 是 patch embedding 矩阵，$\mathbf{E}_{\text{pos}}$ 是位置编码。

CLIP 与对比学习

CLIP（Contrastive Language-Image Pre-training）[**?**] 通过对比学习在 4 亿图像-文本对上训练视觉编码器，使其输出的图像表示与对应文本描述在语义空间中对齐：

$$\mathcal{L}_{\text{CLIP}} = -\frac{1}{N}\sum_{i=1}^{N}\left[\log \frac{\exp(\text{sim}(\mathbf{v}_i, \mathbf{t}_i)/\tau)}{\sum_{j=1}^{N}\exp(\text{sim}(\mathbf{v}_i, \mathbf{t}_j)/\tau)}\right] \tag{210}$$

CLIP 的视觉编码器（通常是 ViT-L/14）因其强大的跨模态对齐能力，成为早期多模态大模型的标准选择。

SigLIP 与改进

SigLIP [**?**] 对 CLIP 的训练目标进行了改进，使用 sigmoid 损失替代 softmax：

$$\mathcal{L}_{\text{SigLIP}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{N}\log\sigma(y_{ij}\cdot\text{sim}(\mathbf{v}_i,\mathbf{t}_j)\cdot\tau) \tag{211}$$

其中 $y_{ij}=1$ 当 $i=j$，否则 $y_{ij}=-1$。这种设计允许更大的 batch size 训练，且不需要全局负样本同步，使训练更加高效。SigLIP 在 InternVL、Qwen2-VL 等新一代模型中广泛使用。

19.3 模态融合机制

将视觉特征注入语言模型的方式决定了多模态模型的架构设计。目前主流的融合机制包括：

线性投影（Linear Projection）

最简单的方式是使用线性层将视觉特征映射到语言模型的 embedding 空间：

$$\mathbf{H}_v = \mathbf{W}_{\text{proj}}\cdot\mathbf{Z}_{\text{vision}} + \mathbf{b} \tag{212}$$

LLaVA [**?**] 最初采用这种方法，通过一个简单的线性投影矩阵连接 CLIP ViT-L/14 和 Vicuna：

- 保持视觉编码器和 LLM 的参数冻结
- 仅训练投影矩阵（约 2M 参数）
- 两阶段训练：预训练对齐 + 指令微调

LLaVA-1.5 [**?**] 将线性投影升级为两层 MLP，显著提升了多模态能力：

$$\mathbf{H}_v = \mathbf{W}_2\cdot\text{GELU}(\mathbf{W}_1\cdot\mathbf{Z}_{\text{vision}}) \tag{213}$$

Q-Former（Querying Transformer）

BLIP-2 [**?**] 提出了 Q-Former 架构，使用可学习的 query token 通过交叉注意力从视觉特征中提取信息：

$$\mathbf{Q}_{\text{out}} = \text{CrossAttn}(\mathbf{Q}_{\text{learnable}},\mathbf{K}_{\text{vision}},\mathbf{V}_{\text{vision}}) \tag{214}$$

Q-Former 的核心设计：

- 32 个可学习的 query embeddings（维度 768）
- 基于 BERT 初始化的 Transformer 块
- 交叉注意力层与自注意力层交替堆叠
- 输出固定数量的视觉 token（32 个），无论输入图像分辨率

两阶段预训练：

1. 视觉-语言表示学习：使用 ITC、ITM、ITG 三种损失训练 Q-Former 与冻结的视觉编码器对齐
2. 视觉-语言生成学习：Q-Former 输出接入冻结的 LLM，训练生成能力

BLIP-2 在 VQAv2 零样本任务上超越 Flamingo-80B 达 8.7%，而可训练参数仅为后者的 1/54。

交叉注意力适配器（Cross-Attention Adapter）

Flamingo [**?**] 和 LLaMA 3.2 Vision 采用在 LLM 内部插入交叉注意力层的方式：

$$\mathbf{h}'_l = \mathbf{h}_l + \text{CrossAttn}(\mathbf{h}_l,\mathbf{K}_{\text{vision}},\mathbf{V}_{\text{vision}}) \tag{215}$$

Flamingo 的架构特点：

- Perceiver Resampler：将可变长度的视觉特征压缩为固定长度
- 在 LLM 的每几层插入 gated cross-attention 层

- 冻结 LLM 参数，仅训练新增的交叉注意力层

LLaMA 3.2 Vision [**?**] 基于 LLaMA 3.1 构建：

- 在冻结的 LLaMA 3.1 文本模型上添加视觉适配器
- 适配器包含多层交叉注意力，将图像编码器表示注入 LLM
- 训练过程中更新视觉编码器和适配器，但冻结 LLM 参数
- 保持文本能力不变，实现 LLaMA 3.1 的” 即插即用” 替换

融合机制对比

表 79: 模态融合机制对比

方法	代表模型	新增参数	视觉 token 数	特点
线性投影	LLaVA	∼2M	576	简单高效
MLP 投影	LLaVA-1.5	∼20M	576	表达能力更强
Q-Former	BLIP-2	∼107M	32	压缩视觉信息
Cross-Attention	LLaMA 3.2	∼1B	可变	深度融合

19.4 代表性多模态大模型

LLaVA 系列

LLaVA（Large Language and Vision Assistant）是最具影响力的开源多模态大模型之一。

LLaVA-1.0：

- 视觉编码器：CLIP ViT-L/14（冻结）
- 语言模型：Vicuna-7B/13B（冻结）
- 连接方式：线性投影层
- 训练数据：595K 图像-文本对（预训练）+ 158K 视觉指令数据（微调）

LLaVA-1.5 的改进：

- MLP 替代线性投影
- 输入分辨率从 224 提升到 336
- 增加学术 VQA 数据
- 更大的语言模型（Vicuna-13B）

LLaVA-NeXT 进一步支持动态分辨率，将图像切分为多个子图像分别编码。

Qwen-VL 系列

Qwen-VL [**?**] 使用更大的视觉编码器和更高分辨率：

- 视觉编码器：OpenCLIP ViT-bigG（448×448）
- 语言模型：Qwen-7B
- 连接方式：单层交叉注意力

Qwen2-VL [**?**] 的创新：

- **动态分辨率**：移除 ViT 的绝对位置编码，引入 2D-RoPE，支持任意分辨率输入
- **M-RoPE**：Multimodal Rotary Position Embedding，将旋转位置编码分解为时间和空间（高度、宽度）三部分，同时捕获 1D 文本、2D 图像、3D 视频的位置信息
- **Token 压缩**：MLP 层将相邻 2×2 token 压缩为 1 个，224×224 图像仅产生 66 个视觉 token

InternVL 系列

InternVL [**?**] 的独特设计在于视觉编码器的大规模化：

- 视觉编码器扩展到 60 亿参数（InternViT-6B）
- 引入 QLLaMA 作为” 胶水层”（8B 参数）连接视觉和语言
- 三阶段训练：对比学习 → 生成学习 → 指令微调

InternVL 2.5 是首个在 MMMU 基准上突破 70% 的开源模型，达到 GPT-4o 水平。

19.5 原生多模态模型

什么是” 原生多模态”?

” 原生多模态”（Native Multimodal）指的是模型从设计之初就具备多模态处理能力，而非在单模态模型基础上” 嫁接” 其他模态。

非原生多模态（如 ChatGPT with GPT-4V）：

- 文本生成：GPT-4
- 图像理解：GPT-4V（独立的视觉模块）
- 语音识别：Whisper
- 图像生成：DALL-E 3
- 各模块独立，通过 API 或文本中转连接

原生多模态（如 GPT-4o、Gemini）：

- 单一神经网络端到端处理所有模态
- 在多模态数据上从头联合训练
- 模态间共享表示空间，实现深度融合
- 无需模态间的文本中转，减少信息损失

GPT-4o

GPT-4o（”o” 代表”omni”，全能）于 2024 年 5 月发布，是 OpenAI 首个原生多模态旗舰模型。

核心特点：

- 单一模型端到端处理文本、音频、视觉输入
- 可直接生成文本、音频、图像输出
- 实时语音对话延迟降至 232ms（接近人类反应速度）
- 音频输入保留语调、情感等非语义信息

与 GPT-4V 的区别：

- GPT-4V：上传图像 → 视觉模型识别 → 转换为文本描述 → GPT-4 处理 → 生成回复
- GPT-4o：上传图像 → 直接理解并生成回复（无中间转换）

由于 GPT-4o 架构未公开，具体技术细节不得而知，但其性能表明原生多模态训练的巨大优势。

Google Gemini

Gemini [**?**] 是 Google 的原生多模态模型系列。

技术报告声明：

> ”Gemini models are natively multimodal, as they are trained jointly across text, image, audio, and video.”

架构特点：

- 早期融合（Early Fusion）架构
- 从预训练阶段就在多模态数据上联合训练
- 支持 32K（Gemini 1.0）到 1M（Gemini 1.5/2.5）token 上下文

模型系列：

- Gemini Ultra：最大规模，MMLU 首次超越人类专家水平
- Gemini Pro：均衡性能与效率
- Gemini Nano：端侧部署优化
- Gemini 2.5 Pro：2025 年发布，加入” 思考模型” 能力

Meta Chameleon

Chameleon [**?**] 是 Meta 开源的原生多模态模型，采用彻底的早期融合架构。

核心设计：

- 将所有模态（图像、文本、代码）表示为离散 token
- 统一的词表包含文本、代码和图像 token
- 使用标准 Transformer 架构处理混合模态序列
- 端到端从头训练，无需单独的图像编码器/解码器

图像离散化：Chameleon 使用改进的 VQ-VAE 将图像编码为离散 token：

- 图像编码为 1024 个离散 token（32×32 latent grid）
- Codebook 大小 8192
- 与文本 token 共享统一的 embedding 空间

训练规模：

- 7B 和 34B 参数版本
- 约 4.4 万亿 token 训练数据（文本、图像-文本对、交错序列）
- 超过 500 万 A100 GPU 小时

与 Gemini 的区别：Chameleon 是完全端到端的稠密模型，没有路由组件或单独的图像解码器，而 Gemini 使用独立的图像解码器。

19.6 统一理解与生成

传统多模态模型要么专注于理解（如 VQA），要么专注于生成（如文生图）。近期研究开始探索在单一模型中统一这两种能力。

挑战与矛盾

理解和生成对视觉表示有不同要求：

- **理解**：需要高层语义抽象，关注” 是什么”
- **生成**：需要细粒度细节，关注” 怎么画”

使用同一个视觉编码器同时服务两种任务会产生冲突——语义编码器（如 CLIP）擅长理解但生成的图像缺乏细节；像素编码器（如 VQ-GAN）能重建细节但语义理解能力弱。

Show-o

Show-o [**?**] 提出用单一 Transformer 统一理解和生成：

核心设计：

- **Omni-Attention**：对文本 token 使用因果注意力，对图像 token 使用全注意力
- **混合建模**：文本使用自回归生成，图像使用离散扩散模型

- **统一词表**：文本 token 和图像 token（VQ-GAN 编码）共享词表

任务能力：

- 图像描述（Image Captioning）
- 视觉问答（VQA）
- 文本生成图像（Text-to-Image）
- 图像编辑（Inpainting/Outpainting）
- 混合模态生成

Show-o 在 VQAv2 上超越 NExT-GPT 和 Chameleon 等更大模型，同时在图像生成上达到 FID 9.24（MSCOCO 30K）。

Janus

DeepSeek 的 Janus [**?**] 采用" 解耦编码、统一处理" 的策略：

核心洞察：理解和生成需要不同的视觉编码，但可以共享语言模型处理。

双编码器设计：

- **理解编码器**：SigLIP，提取高层语义特征
- **生成编码器**：VQ tokenizer，产生离散视觉表示
- **共享 Transformer**：统一处理两种编码的 token 序列

Janus-Pro（2025 年 1 月）进一步提升：

- 基于 DeepSeek-LLM-7B
- MMBench 达到 79.2（超越 LLaVA-v1.5）
- 图像生成 FID 8.53（MSCOCO 30K）

JanusFlow

JanusFlow 将生成端从离散 token 改为连续流（Rectified Flow）：

- 理解端保持不变（SigLIP 编码器）
- 生成端使用 Rectified Flow 替代 VQ tokenizer
- 图像生成质量进一步提升

19.7 视觉 Tokenizer

视觉 tokenizer 是原生多模态和统一模型的关键组件，负责将连续图像转换为离散 token。

VQ-VAE 与 VQ-GAN

VQ-VAE [**?**] 首次提出将连续表示映射到可学习的离散 codebook：

$$z_q = \arg\min_{e_k \in \mathcal{C}} \|z_e - e_k\|_2 \tag{216}$$

其中 z_e 是编码器输出，$\mathcal{C}$ 是 codebook。

VQ-GAN [**?**] 在 VQ-VAE 基础上引入对抗损失：

$$\mathcal{L}_{\text{VQ-GAN}} = \mathcal{L}_{\text{rec}} + \mathcal{L}_{\text{commit}} + \mathcal{L}_{\text{GAN}} + \mathcal{L}_{\text{perceptual}} \tag{217}$$

VQ-GAN 能将 256×256 图像编码为 16×16=256 个离散 token，每个 token 来自大小为 1024-16384 的 codebook。

表 80: 视觉 Tokenizer 类型对比

类型	代表	Codebook	特点
像素级	VQ-GAN	8K-16K	重建质量高，语义弱
语义级	CLIP-ViT	-	语义强，无法重建
混合	SEED	8K	兼顾语义和重建
统一	TokenFlow	16K	双编码器 + 共享映射

语义 vs 像素 Tokenizer

前沿进展

TokenFlow 首次证明离散视觉输入可以在理解任务上超越连续编码器（如 LLaVA），通过双编码器和共享映射实现语义与细节的联合优化。

UniTok 提出统一框架，将离散视觉和语言序列用统一的 next-token prediction 训练，复用 code embedding 到 MLLM token 空间。

19.8 架构演进与对比

多模态架构发展脉络

1. **2022：适配器时代**
   - Flamingo：Cross-Attention + Perceiver Resampler
   - BLIP-2：Q-Former 桥接冻结编码器和 LLM
2. **2023：简化与规模化**
   - LLaVA：简单 MLP 投影，证明” 简单也有效”
   - Qwen-VL：更大视觉编码器（ViT-bigG）
   - Gemini：原生多模态训练
3. **2024：原生与统一**
   - GPT-4o：原生多模态交互
   - Chameleon：开源早期融合
   - Show-o/Janus：统一理解与生成
   - LLaMA 3.2 Vision：Cross-Attention 适配器
4. **2025：深度融合**
   - InternVL 2.5：6B 视觉编码器
   - Qwen2-VL：动态分辨率 + M-RoPE
   - Janus-Pro：更强的统一模型

架构对比总结

19.9 设计选择与权衡

视觉编码器：冻结 vs 微调

冻结（BLIP-2, LLaMA 3.2）：

- 优点：保持预训练视觉能力，训练效率高
- 缺点：无法针对特定任务优化视觉表示

微调（Qwen2-VL, InternVL）：

- 优点：视觉表示可针对性优化

表 81: 主流多模态架构对比

模型	视觉编码器	融合方式	LLM 训练	原生	统一生成
LLaVA-1.5	CLIP ViT-L	MLP 投影	微调	✗	✗
BLIP-2	EVA-CLIP	Q-Former	冻结	✗	✗
LLaMA 3.2	自定义 ViT	Cross-Attn	冻结	✗	✗
Qwen2-VL	DFN ViT	MLP	全量微调	✗	✗
InternVL 2.5	InternViT-6B	MLP	全量微调	✗	✗
Gemini	内置	早期融合	联合预训练	✓	✓
GPT-4o	内置	端到端	联合预训练	✓	✓
Chameleon	VQ-VAE	Token 统一	从头训练	✓	✓
Show-o	VQ-GAN	Omni-Attn	从 LLM 初始化	部分	✓
Janus	SigLIP+VQ	解耦编码	从 LLM 初始化	部分	✓

- 缺点：可能遗忘预训练知识，需要更多训练资源

视觉 Token 数量

- **固定少量**（32，Q-Former）：计算高效，但可能丢失细节
- **固定中等**（576，LLaVA）：平衡信息保留和效率
- **动态**（Qwen2-VL）：根据图像复杂度调整，最灵活

原生 vs 适配器

适配器方式的优势：

- 可复用强大的预训练 LLM
- 训练成本相对较低
- 保持文本能力不退化

原生多模态的优势：

- 模态间深度融合，减少信息损失
- 端到端优化，无接口瓶颈
- 支持实时多模态交互（如语音对话）

19.10 多模态后训练

与纯文本 LLM 类似，多模态大模型也需要通过后训练（Post-training）来提升指令遵循能力、减少幻觉、与人类偏好对齐。然而，多模态场景引入了新的挑战：视觉-语言对齐、跨模态幻觉、以及图像条件下的偏好学习。

视觉指令微调

视觉指令微调（Visual Instruction Tuning）是多模态后训练的基础阶段，通过高质量的多模态指令数据训练模型遵循视觉相关的指令。

LLaVA 的开创性工作 [**?**]：

- 首次使用纯文本 GPT-4 生成多模态指令数据
- 基于 COCO 图像和边界框/描述，扩展为三类指令：对话式 QA、详细描述、复杂推理
- 训练数据：558K 预训练对齐数据 + 665K 指令微调数据

数据质量的重要性：LLaVA 系列的演进（1.0 → 1.5 → NeXT）充分证明了数据质量对视觉指令微调的关键作用：

- **数据多样性**：覆盖对话、描述、推理、学术 VQA 等多种任务
- **数据复杂度**：从简单问答到多轮对话、长文本描述
- **语言多样性**：ShareGPT4V 等数据集支持多语言指令

两阶段训练范式：

1. **预训练对齐**：在大规模图像-文本对上训练投影层，建立视觉-语言连接
2. **指令微调**：在高质量指令数据上微调，获得指令遵循能力

多模态 RLHF

将 RLHF（Reinforcement Learning from Human Feedback）从文本域扩展到多模态场景，面临独特挑战。

LLaVA-RLHF [**?**] 是首个开源的多模态 RLHF 工作：

核心问题：多模态幻觉——模型生成的文本与图像内容不符。

方法：Factually Augmented RLHF (Fact-RLHF)

- 收集 10K 人类偏好数据，标注者比较两个回复并指出哪个更" 幻觉"
- 使用图像描述或多选题答案作为额外信息校准奖励信号
- 通过 PPO 算法优化策略模型

训练流程：

1. 在 LLaVA-SFT+ 基础上添加 LoRA 模块
2. 用 10K 偏好数据训练奖励模型
3. 通过 PPO 进行强化学习

效果：

- LLaVA-Bench：达到 GPT-4 纯文本版 94% 的水平（此前最佳仅 87%）
- MMHal-Bench：相比基线提升 60%

RLHF-V [**?**] 提出更细粒度的方法：

- 收集片段级（segment-level）纠正标注，而非整体比较
- 使用 Dense DPO 直接在细粒度反馈上优化
- 仅需 1.4K 偏好数据，将幻觉率降低 34.8%

多模态 DPO

DPO（Direct Preference Optimization）因其简单高效，在文本 LLM 对齐中广泛使用。然而，直接将 DPO 应用于多模态场景面临独特问题。

无条件偏好问题（Unconditional Preference Problem）：标准 DPO 在多模态场景中，模型可能忽略图像条件，仅基于文本模式学习偏好。这导致：

- 模型过度依赖语言先验
- 视觉信息未被充分利用
- 对齐效果不稳定

mDPO [**?**]（Multimodal DPO）解决方案：

核心改进：

1. **图像偏好优化**：在优化语言偏好的同时，显式优化图像偏好，防止语言偏好被过度优先
2. **奖励锚点**（Reward Anchor）：强制 chosen 响应的奖励为正，避免其 likelihood 下降——这是相对偏好优化的固有问题

mDPO 损失函数：

$$\mathcal{L}_{\text{mDPO}} = \mathcal{L}_{\text{DPO}} + \lambda_{\text{img}}\mathcal{L}_{\text{img-pref}} + \lambda_{\text{anchor}}\mathcal{L}_{\text{anchor}} \tag{218}$$

实验效果：在多个 benchmark 上显著降低幻觉率，特别是在需要精确视觉理解的任务上。

其他多模态 DPO 变体：

- **V-DPO**：Vision-Guided DPO，通过视觉引导生成负样本
- **SymMPO**：对称多模态偏好优化，在正负样本间建立对称学习
- **负样本增强**：通过随机裁剪或扩散模型编辑图像生成负样本

多模态幻觉与评估

多模态幻觉（Multimodal Hallucination）是指模型生成的内容与输入图像不符，是多模态后训练需要重点解决的问题。

幻觉类型：

- **对象幻觉**（Object Hallucination）：描述图像中不存在的物体
- **属性幻觉**（Attribute Hallucination）：错误描述物体的颜色、大小、位置等
- **关系幻觉**（Relation Hallucination）：错误描述物体间的空间或语义关系
- **遗漏幻觉**（Omission）：未提及图像中的重要内容
- **捏造幻觉**（Fabrication）：添加图像中不存在的信息

POPE 评估基准：Polling-based Object Probing Evaluation 将幻觉检测转化为二分类问题：

- 提问：" 图像中是否有 `<object>`? "
- 三种难度：Random（随机物体）、Popular（常见物体）、Adversarial（相关但不存在的物体）
- 评估指标：Accuracy、Precision、Recall、F1

MMHal-Bench：专门评估真实场景中的幻觉，包含 96 个图像-问题对，覆盖 8 类问题 ×12 个物体主题。

幻觉缓解方法：

- **对比解码**（Contrastive Decoding）：比较原图和扰动图的输出分布
- **视觉对比解码**（VCD）：通过高斯噪声等方式扰动图像
- **鲁棒指令微调**：在训练数据中加入幻觉对比样本
- **自省解码**（Self-Introspective Decoding）：让模型自我检查输出的一致性

LLaVA-Critic：多模态自我评估

LLaVA-Critic [**?**] 是首个开源的多模态通用评估模型，能够评估多模态模型在各种任务上的表现。

核心能力：

1. **LMM-as-a-Judge**：提供可靠的评估分数，性能媲美或超越 GPT 模型
2. **偏好学习**：为偏好学习生成奖励信号，提升模型对齐能力

训练数据：LLaVA-Critic-113k 数据集：

- 46K 图像，113K 评估指令样本
- 两种评估模式：
  - Pointwise Scoring：为单个回复打分
  - Pairwise Ranking：比较两个回复的相对质量
- 30 种成对评估 prompt 模板

自我奖励（Self-Rewarding）效果：使用 LLaVA-Critic 的反馈训练 LLaVA-OV-7B/72B：

- 在 6 个开放式多模态 benchmark 上持续提升
- 效果优于人类偏好训练的奖励模型
- 虽然仅使用图像训练，也能提升视频理解能力

这表明多模态模型可以通过自我评估实现" 超人类反馈" 的自我改进路径。

后训练流程总结

表 82: 多模态后训练方法对比

方法	数据需求	训练复杂度	主要目标	代表工作
视觉指令微调	指令数据	低	指令遵循	LLaVA, LLaVA-1.5
RLHF	偏好数据 + 奖励模型	高	减少幻觉	LLaVA-RLHF
DPO/mDPO	偏好数据	中	偏好对齐	mDPO, V-DPO
细粒度反馈	片段级标注	中	精确纠错	RLHF-V
自我奖励	无需人类标注	中	自我改进	LLaVA-Critic

典型后训练流程：

1. **预训练**：在大规模图像-文本数据上训练模态对齐
2. **指令微调**（SFT）：在高质量指令数据上训练
3. **偏好对齐**（可选）：使用 RLHF、DPO 或 mDPO 进一步对齐
4. **自我改进**（可选）：使用 Critic 模型进行迭代优化

19.11 未来方向

统一所有模态

当前多数模型主要处理图像和文本，未来将扩展到：

- 音频/语音的原生支持
- 视频理解与生成
- 3D 场景理解
- 具身智能（Embodied AI）

更高效的视觉表示

- 自适应 token 数量：根据图像内容复杂度动态调整
- 更强的压缩：在更少 token 中保留更多信息
- 层次化表示：粗粒度语义 + 细粒度细节

训练效率

原生多模态训练需要海量计算资源（Chameleon: 500 万 A100 小时）。未来方向：

- 更高效的预训练方法
- 模块化训练：分阶段训练不同模态
- 合成数据增强

评估与对齐

- 更全面的多模态 benchmark
- 跨模态对齐的理论理解
- 多模态幻觉（hallucination）问题

20 推理大模型

前一章讨论的推理优化技术（KV Cache、投机解码等）都致力于同一个目标：**让模型更快地回答**。但 2024 年的一系列突破揭示了另一个维度：有时候，**让模型更慢地回答**反而能获得更好的结果。

这不是技术的倒退，而是对" 推理" 本身的重新理解。传统 LLM 的自回归生成是一种" 快思考"——每个 token 的生成几乎不假思索。而人类解决复杂问题时，往往需要" 慢思考"：反复尝试、回溯检查、探索多条路径。推理大模型（o1、R1 等）将这种慢思考机制引入 LLM，用推理时的额外计算换取答案质量的提升。

20.1 从快思考到慢思考

传统 LLM 的局限

传统大语言模型采用自回归生成方式，给定输入后直接预测下一个 token，这种"System 1" 式的快速响应在许多任务上表现出色，但在需要复杂推理的任务上存在明显局限：

- **推理深度受限**：每个 token 的生成只依赖前面的上下文，缺乏" 回头检查" 的能力
- **错误累积**：推理链中的早期错误会传播到后续步骤
- **缺乏规划**：无法预先规划解题路径，只能" 边走边看"

测试时计算（Test-Time Compute）

推理大模型的核心思想是**测试时计算扩展**（Test-Time Compute Scaling）：在推理阶段投入更多计算资源，换取更好的输出质量。

Snell 等人（2024） [**?**] 的关键发现：

- 测试时计算的扩展可以比扩展模型参数更有效
- 使用" 计算最优" 策略，测试时计算效率可提升 4 倍以上
- 在 FLOPs 匹配的评估中，小模型 + 测试时计算可超越 14 倍大的模型

测试时计算的主要方式：

1. **搜索**：生成多个候选答案，使用验证器选择最佳
2. **思考**：让模型" 思考" 更长时间，生成详细的推理过程
3. **迭代**：多轮自我修正和优化

20.2 链式思考与自一致性

链式思考（Chain-of-Thought）

链式思考（CoT）提示是推理大模型的基础技术，通过引导模型生成中间推理步骤来提升复杂任务的表现。

基本形式：

```
Q: Roger有5个网球，他又买了2罐网球，每罐3个。
   他现在有多少网球?
A: Roger一开始有5个球。2罐网球共有2*3=6个球。
   5+6=11。答案是11。
```

Zero-shot CoT：仅需添加"Let's think step by step" 即可激发模型的推理能力。

自一致性（Self-Consistency）

自一致性 [**?**] 是对链式思考的重要改进，核心思想是：

- 对同一问题生成多条推理路径
- 通过多数投票选择最一致的答案

- 利用”殊途同归”的直觉——正确答案应该可以通过多种方式得出

效果提升：

- GSM8K：+17.9%
- SVAMP：+11.0%
- AQuA：+12.2%
- StrategyQA：+6.4%

改进方法：

- **CISC**（Confidence-Informed SC）：基于置信度加权投票，减少 40% 以上的采样需求
- **RASC**（Reasoning-Aware SC）：动态调整采样数量
- **LSC**（Latent SC）：基于语义一致性选择，适用于长文本回答

20.3 奖励模型与验证器

结果奖励模型（ORM）

结果奖励模型（Outcome Reward Model）只对最终答案给出奖励信号：

$$r_{\text{ORM}}(x, y) = \begin{cases} 1 & \text{if } y \text{ is correct} \\ 0 & \text{otherwise} \end{cases} \tag{219}$$

优点：标注成本低，只需判断最终答案对错

缺点：

- 信用分配困难：无法区分哪一步出错
- 反馈延迟：只有完成整个推理后才能获得奖励

过程奖励模型（PRM）

过程奖励模型（Process Reward Model）对推理的每一步给出奖励信号 [**?**]：

$$r_{\text{PRM}}(x, y_{1:t}) = \text{score}(y_t | x, y_{1:t-1}) \tag{220}$$

其中 y_t 是第 t 步推理，score 通常为 {-1, 0, +1} 表示 {错误, 中性, 正确}。

OpenAI 的实验结果：使用 pre-RLHF GPT-4 作为基础模型，PRM 在 MATH 测试集上达到 78.2% 准确率，显著优于 ORM。

PRM vs ORM 对比：

表 83: 过程奖励模型与结果奖励模型对比

特性	ORM	PRM
反馈粒度	整体结果	每步过程
标注成本	低	高
信用分配	困难	精确
奖励黑客风险	低	较高
搜索效率	较低	更高

隐式 PRM：最近研究发现，可以通过训练 ORM 然后将其作为 PRM 使用，获得”免费”的过程奖励，无需昂贵的步骤级标注。

过程优势验证器（PAV）

PAV（Process Advantage Verifier）[**?**] 结合了过程监督和优势估计：

- 相比 ORM，搜索准确率提升 8% 以上
- 计算效率提升 1.5-5 倍
- 在线 RL 中样本效率提升 5-6 倍

20.4 搜索与规划

Best-of-N 采样

最简单的搜索策略是生成 N 个候选答案，使用验证器选择最佳：

$$y^* = \arg\max_{y \in \{y_1, \ldots, y_N\}} r(x, y) \tag{221}$$

OpenAI o1 在 AIME 2024 上的表现：

- 单次采样（pass@1）：74%
- 64 次采样 + 共识（consensus@64）：83%

蒙特卡洛树搜索（MCTS）

MCTS 将推理过程建模为树搜索问题，每个节点是一个推理状态，边是推理步骤。

基本流程：

1. **选择**（Selection）：使用 UCB 公式选择有潜力的节点
2. **扩展**（Expansion）：生成新的推理步骤
3. **模拟**（Simulation）：完成推理并获得结果
4. **回传**（Backpropagation）：更新路径上所有节点的价值

UCB 公式：

$$\text{UCB}(s, a) = Q(s, a) + c\sqrt{\frac{\ln N(s)}{N(s, a)}} \tag{222}$$

MCT Self-Refine (MCTSr)：结合 LLM 自我改进与 MCTS，在奥林匹克级数学问题上取得优异表现。

SC-MCTS*：使用对比解码设计可解释的奖励模型，结合推测解码加速，平均每节点速度提升 51.9%。在 Blocksworld 数据集上超越 o1-mini 17.4%。

20.5 OpenAI o1

核心设计

OpenAI o1（2024 年 9 月发布）是首个大规模商用的推理大模型，其核心创新在于将链式思考内化为模型能力。

关键特点：

- **推理 token**（Reasoning Tokens）：模型在回答前生成内部推理过程
- **隐藏思考**：推理 token 对用户不可见（但会计费）
- **强化学习训练**：通过大规模 RL 学习" 如何思考"

OpenAI 官方描述：

> "Similar to how a human may think for a long time before responding to a difficult question, o1 uses a chain of thought when attempting to solve a problem. Through reinforcement learning, o1 learns to hone its chain of thought and refine the strategies it uses."

性能表现

扩展规律

o1 展示了两个维度的扩展规律：

表 84: o1 在主要 benchmark 上的表现

Benchmark	GPT-4o	o1-preview	o1
AIME 2024	12%	44%	74%
Codeforces Rating	808	1673	1891
GPQA Diamond	50.6%	73.3%	78.0%
MATH-500	60.3%	85.5%	94.8%

1. **训练时计算**：更多 RL 训练带来更强的推理能力
2. **测试时计算**：更长的思考时间带来更好的答案质量

这打开了一条新的扩展路径：不仅可以通过增加参数和训练数据来提升性能，还可以通过增加推理时的计算来提升。

20.6 DeepSeek-R1

纯 RL 训练的突破

DeepSeek-R1 [**?**]（2025 年 1 月）是首个证明**纯强化学习可以激发推理能力**的开源模型。

DeepSeek-R1-Zero 的关键发现：

- 无需 SFT，仅通过 RL 即可获得强大推理能力
- 涌现出自我反思、验证、动态策略调整等高级推理模式
- AIME 2024：从 15.6% 提升到 71.0%（pass@1），多数投票达 86.7%

GRPO 算法

DeepSeek 使用 Group Relative Policy Optimization（GRPO）进行强化学习训练：

核心思想：

- 省去传统 RLHF 中与策略模型同等规模的 Critic 模型
- 使用组内相对分数作为基线估计
- 大幅降低训练成本

GRPO 优化目标：

$$\mathcal{L}_{\text{GRPO}} = -\mathbb{E}_{x,\{y_i\}}\left[\sum_i \frac{r(x,y_i) - \bar{r}}{\sigma_r} \log \pi_\theta(y_i|x)\right] \tag{223}$$

其中 $\bar{r}$ 是组内平均奖励，σ_r 是组内奖励标准差。

完整训练流程

DeepSeek-R1 的训练包含四个阶段：

1. **冷启动数据**：少量高质量推理数据，解决 R1-Zero 的可读性问题
2. **推理 RL**：大规模 RL 训练，发现更好的推理模式
3. **拒绝采样 SFT**：收集 RL 模型的优质输出进行 SFT
4. **偏好 RL**：与人类偏好对齐

涌现能力

R1-Zero 在训练过程中涌现出多种高级推理行为：

- **自我反思**：”Wait, let me reconsider...”
- **验证**：检查中间步骤的正确性
- **回溯**：发现错误后退回重试
- **策略切换**：一种方法不行时尝试另一种

20.7 开源推理模型

QwQ（Qwen with Questions）

QwQ [?] 是阿里巴巴 Qwen 团队发布的开源推理模型（2024 年 11 月）。

设计理念：

> "QwQ approaches every problem with genuine wonder and doubt. It knows that it knows nothing, and that's precisely what drives its curiosity."

技术特点：

- 32B 参数，32K 上下文长度
- 使用规则化强化学习嵌入推理能力
- 推理时生成长思考链

性能表现：

- GPQA：65.2%（研究生级科学推理）
- AIME 2024：50.0%
- MATH-500：90.6%
- LiveCodeBench：50.0%

已知局限：

- 可能混合语言或意外切换语言
- 可能陷入循环推理，产生过长输出

Marco-o1

阿里巴巴的另一个推理模型 Marco-o1 使用 MCTS 算法生成合成训练数据，结合 CoT 样本进行训练。

20.8 知识蒸馏

推理能力的迁移

DeepSeek 开创性地证明了推理能力可以通过蒸馏迁移到小模型。

蒸馏方法：

- 使用 DeepSeek-R1 生成 800K 推理样本
- 在小模型上进行 SFT（无需额外 RL）
- 小模型获得类似的推理能力

蒸馏模型系列

表 85: DeepSeek-R1 蒸馏模型性能

模型	基座	AIME 2024	MATH-500
R1-Distill-Qwen-1.5B	Qwen2.5-1.5B	28.9%	83.9%
R1-Distill-Qwen-7B	Qwen2.5-7B	55.5%	92.8%
R1-Distill-Qwen-14B	Qwen2.5-14B	69.7%	93.9%
R1-Distill-Qwen-32B	Qwen2.5-32B	72.6%	94.3%
R1-Distill-Llama-8B	Llama3.1-8B	50.4%	89.1%
R1-Distill-Llama-70B	Llama3.3-70B	70.0%	94.5%

关键发现：

- R1-Distill-Qwen-32B 超越 o1-mini

- 蒸馏效果优于同规模模型直接 RL 训练
- 蒸馏是获得推理能力的高效途径

20.9 技术对比与分析

主要推理模型对比

表 86: 主流推理大模型对比

模型	参数	开源	训练方法	AIME	MATH	发布
GPT-4o	-	✗	SFT	12%	60.3%	2024.05
o1-preview	-	✗	RL	44%	85.5%	2024.09
o1	-	✗	RL	74%	94.8%	2024.12
QwQ-32B	32B	✓	RL	50%	90.6%	2024.11
DeepSeek-R1	671B	✓	RL	79.8%	97.3%	2025.01
R1-Distill-32B	32B	✓	蒸馏	72.6%	94.3%	2025.01

训练范式对比

- **o1**：大规模 RL + 隐藏推理 token + 闭源
- **DeepSeek-R1**：GRPO + 多阶段训练 + 完全开源
- **QwQ**：规则化 RL + 开源权重
- **蒸馏模型**：SFT on 推理数据 + 开源

20.10 应用与局限

适用场景

推理大模型特别适合：

- **数学问题**：竞赛数学、定理证明
- **代码生成**：复杂算法、调试
- **科学推理**：物理、化学问题
- **逻辑推理**：规划、约束满足

当前局限

- **延迟高**：思考时间长，不适合实时交互
- **成本高**：推理 token 消耗大量计算资源
- **过度思考**：简单问题也可能产生冗长推理
- **循环推理**：可能陷入无意义的思考循环
- **语言混杂**：思考过程中可能混合多种语言

开放问题

- **最优思考长度**：如何确定何时停止思考?
- **思考可解释性**：隐藏的推理过程是否可信?
- **通用推理**：当前主要在数学/代码领域，如何扩展到更多领域?
- **效率优化**：如何在保持推理质量的同时降低计算成本?

20.11 未来方向

计算最优策略

根据任务难度动态调整测试时计算：

- 简单问题：快速响应
- 困难问题：深度思考
- 预测难度并自动选择策略

多模态推理

将推理能力扩展到多模态：

- 视觉推理：图像中的逻辑关系
- 视频理解：时序推理
- 具身智能：物理世界的规划

推理与 Agent

推理大模型为 AI Agent 提供了更强的规划能力：

- 任务分解与规划
- 工具使用决策
- 长期目标追踪

效率提升

- **推测解码**：加速推理 token 生成
- **早停策略**：检测到答案收敛时提前停止
- **轻量化**：蒸馏更小的推理模型
- **稀疏激活**：仅激活与推理相关的参数

References