

# DeepSeek 系列完整深度笔记

覆盖范围：DeepSeek-V1 → V2 → V3 → R1 → Prover-V2 → V3.2 → DualPath → Engram

学习方式：从基础概念到工程实践，从原理到前沿

## 目录

- DeepSeek-V1: 开源大模型的起点
- DeepSeek-Math: 数学推理的专项突破
- DeepSeek-V2: MoE架构的革命
- DeepSeek-V3: 从V2到工业级MoE
- DeepSeek-R1: 纯RL解锁推理能力
- DeepSeek-Prover-V2: 形式化定理证明
- DeepSeek-V3.2: 稀疏注意力的探索
- DualPath: 榨干每一块闲置网卡的带宽
- DeepSeek Engram: 让大模型学会查字典

## 1. DeepSeek-V1

### 1.1 论文信息

论文名称：DeepSeek LLM: Scaling Open-Source Language Models with Longtermism

### 1.2 模型结构

DeepSeek-V1 基于 LLaMA 架构，选择"站在巨人肩膀上"——不从零开始，而是在成熟基础上优化。

#### 三大核心组件

##### PreRMSNorm（前置均方根归一化）

在每一层计算之前先对输入做归一化处理，防止某个特征值主导整个计算过程。

传统LayerNorm：计算均值和方差，相对开销较大  
PreRMSNorm：只计算均方根，更轻量，效果相当  
公式： $\text{RMSNorm}(x) = x / \sqrt{\text{mean}(x^2) + \epsilon} * \gamma$

##### SwiGLU（门控线性单元）

FFN层的激活函数，决定哪些信息值得向后传递。相比ReLU，SwiGLU通过门控机制提供了更平滑的梯度流。

$\text{SwiGLU}(x) = \text{Swish}(xW_1) \otimes (xW_2)$   
 $\text{Swish}(x) = x * \text{sigmoid}(\beta x)$

##### RoPE（旋转位置编码）

给每个词打上"位置编号"，让模型感知词与词之间的相对距离。相比绝对位置编码，RoPE对长文本的泛化性更好。

GQA（分组查询注意力）——67B专属优化

核心问题：KV Cache 的显存危机

在推理时，模型需要缓存每个 token 的 K（Key）和 V（Value）向量，称为 KV Cache。上下文越长，KV Cache 越大。

标准MHA的显存占用：

KV Cache 大小 = 层数 × KV头数 × 每头维度 × 序列长度

7B模型：30层 × 32头 × 128维 = 122,880 个数值/token

67B模型（不用GQA）：95层 × 64头 × 128维 = 778,240 个数值/token

67B模型（用GQA）：95层 × 8头 × 128维 = 97,280 个数值/token

GQA的核心思路：多个Q头共享同一组K、V头

MHA：32个Q头 → 32个K头 + 32个V头（每人一本书）

GQA：32个Q头 → 8个K头 + 8个V头（四人共用一本书）

压缩比：4倍

为什么7B不需要GQA，67B必须用？

三个维度同时变大（层数×头数×维度），KV Cache 指数级增长。GQA 让 67B 的 KV Cache 开销控制在和 7B 同一数量级。

具体参数对比：

参数	7B	67B
层数	30	95
模型维度	4096	8192
注意力头数	32	64
KV头数	32	8（GQA）
Context长度	4096	4096
Sequence Batch Size	2304	4608
学习率	4.2e-4	3.2e-4
训练Token数	2.0T	2.0T

1.3 BBPE 分词算法

DeepSeek-V1 使用 **Byte-level BPE (BBPE)** 。

普通BPE的问题:

遇到生僻字 → [UNK] (未知符号) → 信息丢失

BBPE的改进:

在字节 (byte) 级别操作  
任何文字 = 256个基础字节的组合  
→ 永远不会出现UNK ☒  
→ 对中文、代码、数学符号都友好

DeepSeek的BBPE配置:

- 训练语料: 约24GB文本
- 词汇表大小: 102,400个token (比GPT-2的50,257大一倍)
- 原因: 中文字符、数学符号、代码关键字需要更多词表空间

1.4 SFT训练 (监督微调)

目的: 教模型从"续写者"变成"助手"

没有SFT时的问题:

输入: "帮我写一封道歉信"  
Base模型输出: "帮我写一封道歉信的方法有很多种, 古代文人常用..."  
← 在续写, 不是在帮忙!

训练数据:

- 规模: 1.5M 条中英文指令数据
- 中文: 日常对话、知识问答、写作
- 英文: 学术、代码、逻辑推理
- 双语混合: 让模型学会语言间灵活切换

超参数设计:

配置	7B	67B
训练轮数	4 epochs	2 epochs
学习率	1e-5	5e-6

为什么大模型用更小的学习率和更少轮数?

大模型预训练积累了大量知识, 大学习率会导致灾难性遗忘 (Catastrophic Forgetting) 。步子太大会把预训练学到的知识"冲掉"。

## 1.5 DPO训练（直接偏好优化）

**背景：**SFT之后模型能理解指令，但输出质量参差不齐。DPO让模型学会"更好"的回答风格。

**偏好数据构建流程：**

第一步：对同一问题，用Chat Model采样多个候选回答  
第二步：用规则+模型打分区分好坏  
    规则维度（客观）：格式规范、长度合适、回答了问题  
    模型打分维度（主观）：用更强模型当裁判  
第三步：筛选差异明显的对（信号强）

**DPO vs RLHF的优势：**

RLHF流程：人类标注 → 训练RM → PPO训练  
    问题：需要单独训练RM，PPO不稳定，显存压力大  
  
DPO流程：偏好对数据 → 直接优化Policy Model  
    优势：更简单、更稳定、不需要RM、显存更小

**DPO目标函数直觉：**

```
loss = -log(sigmoid(
    β * (log_prob(好回答) - log_prob(差回答)) # 当前模型
    - log_prob(好回答) - log_prob(差回答)) # 参考模型
))
# β控制偏离参考模型的程度
# 让"好回答"概率相对参考模型提高
# 让"差回答"概率相对参考模型降低
```

**DeepSeek-V1 DPO超参数：**

- Batch size: 512（更大batch → 梯度更稳定）
- 学习率: 5e-6（和大模型SFT一样小，防止过度更新）

**为什么用自己的模型生成偏好对？**

- 不是让模型"评判好坏"，而是生成多样性后用规则筛选
- "模型最了解自己的输出分布"
- 成本远低于大规模人工标注

## 2. DeepSeek-Math

### 2.1 主要贡献

1. 可扩展的数学预训练：从Common Crawl中挖掘120B数学tokens
2. 强化学习的探索：GRPO算法在数学推理上的应用

### 2.2 数据处理流程：fastText流水线

核心思路：粗筛 → 精筛的工业流水线

Math Seed (高质量种子)  
↓ 训练fastText分类器  
fastText扫描40B HTML网页 (粗筛, 极快)  
↓ 按分数排名  
去重 + 过滤 (精筛)  
↓ 四轮迭代  
最终产出: 3550万数学网页, 120B tokens

为什么用fastText而不是GPT-4?

维度	fastText	GPT-4
速度	数十万条/秒	几十条/秒
成本	极低	极高
精度	足够 (粗筛不需要完美)	更高
场景	40B网页的初步过滤	精细质量判断

速度差距约10,000倍，面对40B网页，大模型完全不可行。这是工程思维的核心：规模决定工具选择。

fastText训练数据:

- 正样本: 从OpenWebMath随机抽50万条 (已知高质量数学网页)
- 负样本: 从Common Crawl随机抽50万条普通网页
- 关键: 用目标域数据定义"什么是数学", 而不是人工写规则

fastText配置:

- 向量维度: 256
- 学习率: 0.1
- 最大词n-gram长度: 3
- 最小词出现次数: 3
- 训练轮数: 3

2.3 数据清洗四步走

第一步：去重

- URL去重: 相同URL只保留一份
- 近似去重: 相似内容去除
- 将Common Crawl压缩为40B个HTML网页

第二步：召回与排名

- 使用fastText对去重后的网页打分
- 按分数降序排名
- 只保留前40B tokens（通过预训练实验确定阈值）

### 第三步：迭代优化（4轮）

- 识别和注释未收集的数学网页来源
- 丰富种子语料库，重新训练fastText
- 每轮召回更多高质量数据

### 第四步：去污染（Decontamination）

防止训练数据包含测试集内容，避免评测基准被污染。

两段式去污染规则：

```
def is_contaminated(webpage_text, benchmark_texts):
    for bench_text in benchmark_texts:
        length = len(bench_text)

        if length >= 10:
            # 长文本：子串匹配（宽松）
            # 10字符随机碰撞概率极低，误伤少
            if bench_text in webpage_text:
                return True

        elif length >= 3:
            # 短文本：精确匹配（严格）
            # "f dx"这种短公式到处都有，不能用子串匹配
            if bench_text == webpage_text.strip():
                return True

        # < 3字符：直接放弃过滤（无法可靠识别）
    return False
```

设计逻辑：

- 长题目（≥10字符）：出现即删，误伤极少
- 短题目（3-9字符）：必须完全匹配才删，保护正常内容
- 极短（<3字符）：无法区分，跳过

## 2.4 预训练配置

基座模型：DeepSeek-Coder-Base-v1.5 7B（从Code LLM开始是好选择）

为什么从代码LLM开始？

代码和数学有深层共性：

- 都需要严格的符号推理
- 都需要追踪变量状态
- 都有明确的正确/错误标准
- 都需要结构化的步骤分解

代码训练相当于给数学能力"预热"

先做代码预训练 → 再做数学预训练

效果 > 直接做数学预训练

预训练数据分布（500B tokens）：

数据来源	比例	说明
DeepSeekMath语料库	56%	核心数学网页
AlgebraicStack	4%	代数专项
arXiv论文	10%	高质量数学推导
GitHub代码	20%	提升代码-数学协同能力
Common Crawl自然语言	10%	保持通用能力

训练框架：HAI-LLM

优化器：AdamW

预热步数：2000步

学习率：4.2e-4

Batch Size：10M tokens

2.5 指令微调

数据集构成：

- 总量：776K样本
- 语言：中英文混合
- 难度：覆盖不同数学领域和复杂程度

三种解题格式：

格式	全称	适用场景	特点
COT	Chain of Thought	需要语义理解的题	自然语言推理，人类可读
POT	Program of Thought	数值计算密集型	用代码写推理，精度100%
Tool-integrated	工具集成推理	推理+精确计算混合	两全其美，系统较复杂

选择示例: "计算 $123456789 \times 987654321$ "

→ 选POT（代码计算），因为自然语言处理进位信息容易出错

训练配置:

- 数据拼接长度: 4K tokens
- Batch Size: 256
- 学习率:  $5e-5$
- 训练步数: 500步

## 2.6 Pass@K 评估指标

为什么不用普通准确率?

模型对数学/代码题有随机性，一次采样答错不等于模型不会。

**Pass@K的思路:**

对同一道题采样K次  
只要有1次答对 → 算通过  
更公平地反映模型的真实能力上限

计算公式:

```
def pass_at_k(n_samples, n_correct, k):  
    """  
    n_samples: 总采样次数 (如采样16次)  
    n_correct: 其中答对的次数  
    k: Pass@K中的K值  
    """  
    if n_correct == 0:  
        return 0.0  
  
    # 1 - 全部答错的概率  
    prob_all_wrong = (  
        combinations(n_samples - n_correct, k) /  
        combinations(n_samples, k)  
    )  
    return 1.0 - prob_all_wrong  
  
# 例子: 16次采样中答对4次  
# Pass@1 = 4/16 = 25%  
# Pass@64 ≈ 99% (采样越多越容易碰到对的)
```

## 2.7 强化学习: GRPO在数学上的应用

算法: GRPO (Group Relative Policy Optimization)

**Group Size = 64** 的含义:



对每道数学题，采样64个不同的解题过程

可能的结果：

- └─ 64个里有30个答对 → 这道题难度中等
- └─ 64个里有2个答对 → 这道题很难
- └─ 64个里有63个答对 → 这道题很简单

为什么数学任务需要 $G=64$ （比R1-Zero的 $G=16$ 更大）？

数学推导步骤多，出错概率高

竞赛级别的题：16个样本可能全部答错！

→ 全是0的奖励 → 无法估计优势值

$G=64$ ：即使很难的题，64次里大概率有1-2次答对

→ 至少有一个正奖励信号

→ 优势值估计有意义

### 3. DeepSeek-V2

#### 3.1 论文信息

论文名称：DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model

#### 3.2 DeepSeekMoE 架构

MoE（Mixture of Experts，混合专家）的基本思想：不是所有参数都参与每次计算，而是根据输入动态选择激活哪些"专家"。

##### 3.2.1 共享专家：解决知识冗余

没有共享专家时的问题：

刑事律师：需要懂基础法律 + 刑事专业    < 基础法律重复！

民事律师：需要懂基础法律 + 民事专业    < 基础法律重复！

税务律师：需要懂基础法律 + 税务专业    < 基础法律重复！

→ 每个专家都在重复存储"基础知识"

→ 浪费参数，且每人的专业深度被基础知识占用

引入共享专家后：

共享专家（法律顾问）：专门掌握基础法律，人人调用

刑事律师：只专注刑事专业

民事律师：只专注民事专业

税务律师：只专注税务专业

→ 每个路由专家可以专得更深！

论文将此称为：减少路由专家之间的知识冗余。

### 3.2.2 细粒度专家切分

传统MoE（粗粒度）：

4个大专家，每次激活Top-2  
[专家A(大)] [专家B(大)] [专家C(大)] [专家D(大)]  
激活后：[A] + [C]

DeepSeekMoE（细粒度）：

8个小专家，每次激活Top-4（激活参数量相同！）  
[a][b][c][d][e][f][g][h]  
激活后：[a] + [c] + [e] + [g]

为什么切细更好？

专家越细 → 每个专家负责的知识领域越窄 → 专业化程度越高 → 组合更灵活。类比乐高积木：大块只能拼出几种形状，小块能拼出任何形状。

### 3.2.3 FFN输出计算公式

```
h'_t = u_t                                # 原始输入
      +  $\sum \text{FFN}_i^{(s)}(u_t)$           # 共享专家：每次必激活
      +  $\sum g_{\{i,t\}} * \text{FFN}_i^{(r)}(u_t)$  # 路由专家：只激活Top-K个
```

其中门控值：

$g_{\{i,t\}} = s_{\{i,t\}}$ , 如果  $s_{\{i,t\}} \in \text{TopK}(\{s_{\{j,t\}}\}, K_r)$   
 $g_{\{i,t\}} = 0$ , 否则

亲和度分数：

$s_{\{i,t\}} = \text{Softmax}_i(u_t^T * e_i)$

## 3.3 设备受限路由（Device-Limited Routing）

问题背景：

在专家并行时，专家分布在多台机器上。如果没有限制，每个token可能需要联系全集群的专家。

没有设备限制时的通信开销：

V2有160个路由专家，分布在多台机器  
token\_A的Top-6专家 → 可能散落在6台不同机器上  
→ 需要6次跨机器通信  
→ 每次通信 ~100μs  
→ 总等待：600μs

有设备限制(M=3)时：

token\_A的Top-6专家 → 只来自3台机器以内  
→ 最多3次跨机器通信  
→ 总等待：300μs（减半！）

## 关键实验结论：

M=1（只用本机专家）：性能明显下降  
M=2：性能接近无限制  
M=3：性能几乎等于无限制 ← DeepSeek选择这里  
M=∞：性能基准线，但通信爆炸

好专家的分布是局部聚集的，不需要全局搜索。

### 3.4 负载均衡三道防线

#### 第一道：设备受限路由

控制每个token最多联系M个设备，解决"发送侧"流量集中问题。

#### 第二道：Communication Balance Loss

解决"接收侧"流量集中问题。

$$L_{\text{CommBal}} = \alpha_3 \times \sum_i f_{i''} \times P_{i''}$$

$$f_{i''} = (D/MT) \times \sum_t 1(\text{Token } t \text{ is sent to Device } i)$$

$$P_{i''} = \sum_{j \in E_i} P_j$$

含义：

$f_{i''}$ ：设备i实际收到的token比例

$P_{i''}$ ：路由器分配给设备i的概率

两者都大 → 惩罚大 → 路由器被迫分散流量

类比：高速公路收费站引导系统，主动引导车辆去队伍短的窗口。

#### 第三道：Token-Dropping Strategy（Token丢弃）

前两道防线都是"概率性"的，极端情况下某设备仍可能过载。Token Dropping是最后的兜底。

#### 处理流程：

```
def device_forward(tokens_received, avg_budget):
    # 第一步：按路由分数降序排列
    sorted_tokens = sort_by_routing_score(tokens_received,
    order='descending')

    # 第二步：只处理预算以内的token
    tokens_to_process = sorted_tokens[:avg_budget] # ☒ 处理
    tokens_to_drop = sorted_tokens[avg_budget:] # ☒ 丢弃

    # 被丢弃的token不参与该层计算，直接用残差跳过
    return compute(tokens_to_process)
```

#### 为什么按路由分数排序再丢弃是合理的？

- 低分token：与该专家本来匹配度低，即使计算贡献也小，丢掉影响不大
- 高分token：该专家对这些token最重要，必须保留

### 3.5 MLA (Multi-Head Latent Attention)

**核心动机：**GQA是粗暴压缩（减少头数），MLA是精细压缩（压缩每头维度）。

#### 问题规模

V2标准配置下，KV Cache大小：  
128个注意力头  $\times$  128维  $\times$  2(K+V)  $\times$  128K tokens  $\times$  60层  
= 约 240GB (比模型权重还大!)

#### 低秩投影的核心思想

##### 类比：

完整KV (1000页百科全书)

GQA的方案：只带8章，其他章节丢掉  
→ 轻了，但信息有损失

MLA的方案：把1000页压缩成50页精华索引  
需要内容时，从索引重新展开  
→ 同样轻，但信息损失更少！

#### 压缩效果数字对比：

标准MHA每个token缓存：  
 $K + V = 128\text{头} \times 128\text{维} \times 2 = 32,768$  个数值

MLA每个token只缓存：  
一个低秩向量  $c_t^{KV}$ ，维度 = 512

压缩比：32,768 / 512 = 64倍！ (远超GQA的16倍)

#### MLA数据流

第一步：压缩 (每个新token都要做，结果存入缓存)  
 $h_t \xrightarrow{[W^{DKV} \text{降维矩阵}]} c_t^{KV} (512\text{维}) \leftarrow \text{只缓存这个!}$

第二步：展开 (计算注意力时)  
 $c_t^{KV} \xrightarrow{[W^{UK} \text{升维矩阵}]} \text{完整K向量}$   
 $c_t^{KV} \xrightarrow{[W^{UV} \text{升维矩阵}]} \text{完整V向量}$

##### # MLA的KV压缩与展开

###### # 压缩阶段

```
c_t_KV = h_t @ W_DKV      # 降维: [d_model] → [512]
kv_cache.store(c_t_KV)    # 只缓存这512维!
```

###### # 展开阶段 (计算注意力时)

```
c_t_KV = kv_cache.load()  # 取出512维
```

```

k_t = c_t_KV @ W_UK          # 升维回完整K
v_t = c_t_KV @ W_UV          # 升维回完整V

# 注意力计算照常进行
scores = q_t @ k_t.T / sqrt(d_head)
output = softmax(scores) @ v_t

```

## RoPE位置编码的处理

KV压缩状态无法直接加RoPE，MLA将K拆成两部分：

```

K = [K^C ; K^R]
    ↑      ↑
    内容部分 位置部分
    从c_t_KV 从h_t直接
    展开得到 计算RoPE

```

缓存时：

```

c_t_KV (内容) ← 低秩压缩，省空间
k_t_R (位置) ← 单独缓存RoPE结果

```

## 压缩维度的权衡

```

压缩维度 = 16：省了2048倍，但表达能力极有限 → 效果崩塌
压缩维度 = 4096（原始维度）：退化成标准MHA，无压缩
压缩维度 = 512（MLA选择）：
├── 缓存压缩64倍 ☒
├── 512维足够保留主要语义信息 ☒
└── 展开计算开销可接受 ☒

```

## 4. DeepSeek-V3

### 4.1 论文信息

论文名称：DeepSeek-V3 Technical Report

### 4.2 架构改进：从V2到V3

#### 4.2.1 Sigmoid门控替换Softmax

问题背景：

V2到V3，路由专家从160个增加到256个。

Softmax的饱和问题:

所有专家分数加起来 = 1

专家越多  $\rightarrow$  每个专家平均分数  $\approx 1/256 \approx 0.004$

$\rightarrow$  大家的分数都挤在0.004附近

$\rightarrow$  梯度几乎为0

$\rightarrow$  路由器变得"迟钝", 训练极慢

$\rightarrow$  不同专家之间缺乏区分度

Sigmoid的解决:

每个专家独立输出, 范围始终是(0, 1)

256个专家和2个专家的分数分布完全一样

$\rightarrow$  区分度完全保留

**类比:** softmax像全班同学分100分 (人越多每人越少), sigmoid像每人单独打分 (不受人数影响)。

**V3的FFN输出公式 (注意变化):**

$$h'_t = u_t + \sum \text{FFN}_i^{(s)}(u_t) + \sum g_{\{i,t\}} * \text{FFN}_i^{(r)}(u_t)$$

$$g_{\{i,t\}} = g'_{\{i,t\}} / \sum_j g'_{\{j,t\}} \quad \leftarrow \text{归一化}$$

$$g'_{\{i,t\}} = s_{\{i,t\}}, \text{ 如果 } s_{\{i,t\}} \in \text{TopK}(\{s_{\{j,t\}} + b_{\{j\}}\}, K_r)$$

$$g'_{\{i,t\}} = 0, \text{ 否则}$$

$$s_{\{i,t\}} = \text{Sigmoid}(u_t^T * e_i) \quad \leftarrow \text{改为sigmoid!}$$

#### 4.2.2 无辅助Loss负载均衡

**V1/V2的问题:** 用辅助Loss来保证负载均衡, 但这些Loss会与主Loss互相打架, 导致模型效果下降。

**V3的解法:** 直接把辅助Loss扔掉, 换成可学习的bias项:

# V3的路由选择:

`top_k_selection = gate_score + b_i`     # `b_i` 是可学习的偏置项

# 动态调节机制:

# 专家*i*过载  $\rightarrow$  降低`b_i`  $\rightarrow$  该专家被选中概率下降

# 专家*i*空闲  $\rightarrow$  提高`b_i`  $\rightarrow$  该专家被选中概率上升

# 关键优势:

# `b_i`只影响"谁被选中", 不影响"选中后的计算权重"

#  $\rightarrow$  不干扰模型的预测能力!

#### 4.2.3 Complementary Sequence-Wise Auxiliary Loss

在无辅助Loss均衡的基础上, 额外加一个**序列级别**的辅助Loss:

$$L_{\text{Bal}} = \alpha \times \sum_i f_i \times P_i$$

$$f_i = (N_r / K_r \times T) \times \sum_t 1(s_{\{i,t\}} \in \text{TopK}(\{s_{\{j,t\}}\}, K_r))$$

$$P_i = (1/T) \times \sum_t s'_{\{i,t\}} \quad \text{其中 } s'_{\{i,t\}} = s_{\{i,t\}} / \sum_j s_{\{j,t\}}$$

**与V2辅助Loss的区别：**粒度控制在单条数据（序列级别），比batch级别更细，对模型能力影响更小。

#### 4.2.4 No Token-Dropping

有了bias动态调节 + Sequence-Wise Loss，负载均衡更彻底，Token-Dropping这个"应急措施"可以光荣退休。

**V2→V3负载均衡演进：**

机制	V2	V3
专家级辅助Loss	☑	✗（副作用大，删掉）
设备级辅助Loss	☑	✗
通信平衡Loss	☑	✗
bias动态调节	✗	☑（无副作用）
Sequence-Wise Loss	✗	☑（细粒度补充）
Token-Dropping	☑	✗（不再需要）

### 4.3 Multi-Token Prediction（MTP）

#### 4.3.1 传统训练的"浪费"

传统语言模型训练：

输入："今天 天气 很"

预测："好" ← 每次只预测下一个词

处理完这个样本，模型只学到了一件事：

"今天天气很" 后面跟 "好"

其余信息全部丢弃 ← 浪费！

#### 4.3.2 MTP的思路

MTP训练:

输入: "今天 天气 很"

同时预测:

主模型 → "好" (预测第+1个词)

MTP模块1 → ", " (预测第+2个词)

MTP模块2 → "心情" (预测第+3个词)

一个样本, 学到三件事 ← 数据利用率提升3倍!

#### 4.3.3 顺序预测 vs 并行预测

其他工作 (并行预测):

主模型 → [独立头1] → 预测第+1个词

→ [独立头2] → 预测第+2个词

→ [独立头3] → 预测第+3个词

问题: 三个预测互相独立, 没有因果关系

**DeepSeek-V3 (顺序预测):**

主模型 → 预测第+1个词"好"

↓ 把"好"的信息传递下去

MTP模块1 → 知道"好"之后, 预测第+2个词", "

↓ 把", "的信息传递下去

MTP模块2 → 知道"好, "之后, 预测第+3个词"心情"

每一步预测都建立在前一步的基础上 ← 保持完整因果链

#### 4.3.4 MTP具体实现

第k个MTP模块的输入构造:

$$h'_i{}^k = M_k[\text{RMSNorm}(h_i{}^{k-1}); \text{RMSNorm}(\text{Emb}(t_{\{i+k\}}))]$$

解释:

- $\text{RMSNorm}(h_i{}^{k-1})$ : 上一层传下来的隐藏状态
- $\text{RMSNorm}(\text{Emb}(t_{\{i+k\}}))$ : 当前位置+k的词嵌入 (告诉模块"上一步预测了什么")
- 两者拼接后, 经过一个Transformer Block继续处理

每个MTP模块的训练目标:

$$\begin{aligned} L_{\text{MTP}}^k &= \text{CrossEntropy}(P_{\{2+k:T+1\}}^k, t_{\{2+k:T+1\}}) \\ &= -(1/T) \times \sum \log P_i{}^k[t_i] \end{aligned}$$

最终综合MTP目标:

$$L_{\text{MTP}} = (\lambda/D) \times \sum_k L_{\text{MTP}}^k$$

其中 $\lambda$ 是权重系数, D是预测深度



#### 4.3.5 MTP的双重价值

价值1: 训练信号密度提升

传统: 1个样本 → 1个loss

MTP: 1个样本 → D个loss

→ 相同数据量, 模型学到更多

价值2: 迫使模型"提前规划"

要准确预测第+3个词, 模型必须在表示第+1个词时

就已经"想好"后面要说什么

→ 表示质量显著提升

→ 对代码、数学这类需要"想好再说"的任务帮助最大

#### 4.3.6 推理时的投机解码

训练时MTP学会预测多个词, 推理时可以利用MTP做**投机解码 (Speculative Decoding)**, 加速推理。

**投机解码流程:**

步骤1: MTP模块(轻量草稿器)快速猜测接下来4个词

[今天天气很好, 我]

步骤2: 主模型并行验证所有猜测

"好" ☒ ", " ☒ "我" ☒ (应该是"心情")

步骤3: 接受正确前缀"好, ", 只重算"心情"

效率: 原本需要3次完整前向传播, 现在只需1次!

#### MTP模块为什么特别适合做猜测器?

优势1: 训练时顺带训好, 零额外成本 ☒

优势2: 共享主模型的嵌入层和输出头, 和主模型说"同一种语言", 猜对率高 ☒

优势3: 只有一个Transformer Block, 极为轻量, 猜测速度快 ☒

### 5. DeepSeek-R1

#### 5.1 论文信息

**论文名称:** DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning

**核心贡献:** 证明了无监督的RL就能教会LLM推理——不需要任何SFT标注数据, 纯强化学习即可让大模型学会推理。

#### 5.2 整体概览

**两个主要模型:**

1. **DeepSeek-R1-Zero:** 纯RL训练, 无需SFT作为前置步骤

2. **DeepSeek-R1:** RL之前结合多阶段训练和冷启动数据, 解决Zero的可读性差问题

**开源内容:**

- DeepSeek-R1-Zero
- DeepSeek-R1
- 基于Qwen和Llama蒸馏的六个Dense模型（1.5B、7B、8B、14B、32B、70B）

## 5.3 GRPO算法详解

### 5.3.1 PPO的回顾

PPO是process-reward（token级别）的action，目标函数：

$$J_{\text{PPO}}(\theta) = E[q \sim P(Q), o \sim \pi_{\theta_{\text{old}}}(O|q)] \\ (1/|o|) \times \sum_i [\min(\pi_{\theta}(o_i|q, o_{<t})/\pi_{\theta_{\text{old}}}(o_i|q, o_{<t}) \times A_i, \\ \text{clip}(\pi_{\theta}/\pi_{\theta_{\text{old}}}, 1-\epsilon, 1+\epsilon) \times A_i) \\ - \beta \times D_{\text{KL}}(\pi_{\theta} || \pi_{\text{ref}})]$$

**PPO需要的四个模型：**

- Policy Model（策略模型）：要训练的主角
- Reference Model（参考模型）：防止跑偏太远
- Reward Model（奖励模型）：打分
- Value Model（价值模型）：估计"这个状态值多少分"← 和Policy一样大，显存翻倍！

### 5.3.2 GRPO的核心创新

去掉了**Value Model**，用"小组平均分"替代优势值估计：

对于每个问题q，GRPO从旧策略采样G个输出  $\{o_1, o_2, \dots, o_G\}$

优势值计算：

$$A_i = (r_i - \text{mean}(\{r_1, \dots, r_G\})) / \text{std}(\{r_1, \dots, r_G\})$$

解读：

比小组平均好 → 正优势 → 加强这种输出

比小组平均差 → 负优势 → 抑制这种输出

不需要Value Model估计"绝对价值"

只需要组内相对比较！

**GRPO目标函数：**

$$J_{\text{GRPO}}(\theta) = E[q \sim P(Q), \{o_i\}_{i=1}^G \sim \pi_{\theta_{\text{old}}}(O|q)] \\ (1/G) \times \sum_i [\min(\pi_{\theta}(o_i|q)/\pi_{\theta_{\text{old}}}(o_i|q) \times A_i, \\ \text{clip}(\pi_{\theta}/\pi_{\theta_{\text{old}}}, 1-\epsilon, 1+\epsilon) \times A_i) \\ - \beta \times D_{\text{KL}}(\pi_{\theta} || \pi_{\text{ref}})]$$

### 5.3.3 PPO vs GRPO的λ 参数

**λ的含义：**控制未来奖励的折扣程度

$\lambda=0$ : 只看当前步, 完全忽略未来  
 $\lambda=1$ : 把所有步的奖励都加起来, 不折扣

### 为什么语言模型RL中 $\lambda=1$ 比 $\lambda=0.95$ 更好?

语言生成的特殊性: 整个输出完成后才能判断好坏, 奖励无法归因到某个具体token。

$\lambda=0.95$ 时 (大多数开源实现的默认值):

token<sub>10</sub>: 1.0

token<sub>9</sub>: 0.95

...

token<sub>1</sub>: 0.63 ← 第一个token只分到63%!

问题: 前几个token的梯度信号被人为削弱

但"今天天气"这几个token对整个句子质量同样重要!

→ 训练信号不均衡 → 模型学得歪

$\lambda=1.0$ 时:

token<sub>1</sub>到token<sub>10</sub>: 全部 = 1.0

每个token平等地学习 ☒

### 实验结论:

- PPO( $\lambda=0.95$ ): 表现明显差于GRPO
- PPO( $\lambda=1.0$ ): 性能接近GRPO水平
- GRPO: 最稳定, 性能最好

### 5.3.4 KL散度的近似估计

#### 为什么需要近似?

原有KL计算包含期望, 需要实时采样, 计算复杂度高, 且数值不稳定。

近似公式 (来自《Approximating KL Divergence》):

$$D_{KL}[\pi_{\theta} || \pi_{ref}] \approx \pi_{ref}/\pi_{\theta} - \log(\pi_{ref}/\pi_{\theta}) - 1$$

优势:

- 只需计算当前策略和参考策略的概率比值
- 不需要计算KL散度的积分或期望
- 计算复杂度显著降低

直觉:

- 当 $\pi_{\theta} \approx \pi_{ref}$ 时:  $\pi_{ref}/\pi_{\theta} \approx 1$ ,  $\log(\pi_{ref}/\pi_{\theta}) \approx 0$ , 近似值  $\approx 0$  ☒
- 当 $\pi_{\theta} \neq \pi_{ref}$ 时: 近似值给出较大正值, 反映差异 ☒

### 5.4 DeepSeek-R1-Zero

#### 5.4.1 基座模型

DeepSeek-V3-Base

#### 5.4.2 基于规则的奖励系统

##### 为什么不用神经网络奖励模型（Reward Model）？

##### Reward Hacking（奖励过度利用）的危险：


训练后期，模型发现RM的"弱点"：


输出超长的、听起来很有道理的废话

→ RM被"忽悠"，打出0.95的高分

→ 但答案其实是错的！

现实证明（论文图6）：

Reward分数：持续上升 

CodeForces实际表现：持续下降 

← 模型在"刷分"而不是"真的变强"

##### 两种基于规则的奖励：

```
def accuracy_reward(response, ground_truth):
    """
    数学题：提取\boxed{}里的答案，和标准答案比较
    代码题：直接跑编译器，看测试用例通不通

    答案对就是对，错就是错
    模型无法忽悠这个评委
    """
    if extract_answer(response) == ground_truth:
        return 1.0
    return 0.0

def format_reward(response):
    """
    检查思维过程是否在<think></think>标签里
    纯字符串匹配，无法被破解
    """
    if "<think>" in response and "</think>" in response:
        return 1.0
    return 0.0

# 最终奖励：两者直接相加
total_reward = accuracy_reward + format_reward
```

#### 5.4.3 训练模板

A conversation between User and Assistant.  
The assistant first thinks about the reasoning process  
and then provides the user with the answer.  
The reasoning process and answer are enclosed within  
<think></think> and <answer></answer> tags.  
User: {prompt}. Assistant:

模板统一，不针对特定问题调整，避免内容偏差。

#### 5.4.4 训练超参数（升级版报告详细版）

学习率:  $3e-6$   
KL系数: 0.001  
采样温度: 1.0  
Group size: 16  
最大输出长度:  
    前8200步: 32,768 tokens  
    8200步后: 65,536 tokens ← 长度跃升!  
Batch size: 32题 × 16样本 = 512  
参考模型更新: 每400步替换一次  
总训练步数: 10,400步 (约1.6个epoch)

#### 8200步的长度跃升现象:

0~8200步: 性能缓慢提升, 输出长度缓慢增长  
8200步时: 性能突然显著跳升! 输出长度同时跳升!

解释:

模型突破了某个"能力阈值"  
开始真正理解"想得更深=答得更好"  
主动申请更长的思考空间

→ 8200步后把最大长度从32K扩展到64K  
给模型更大的思考空间

#### 每400步替换参考模型的原因:

固定参考模型的问题:

训练后期 $\pi_\theta$ 已经进化很多  
KL越来越大 → KL惩罚越来越重  
→ 模型被死死锁住, 无法继续进步!

动态参考模型:

参考模型跟着策略模型一起进化  
KL始终在可控范围内  
→ 模型可以持续进步

#### 5.4.5 R1-Zero的涌现行为

#### Self-Evolution自我进化:

训练初期 (step 0~2000) :  
<think>1+1=2</think>  
<answer>2</answer>  
(思考过程很短, 直接输出)

训练中期 (step 2000~6000) :

```
<think>
  Let me compute... 1+1=2
  Wait, let me verify this again...  ← 自发出验证行为!
  Yes, 1+1=2, confirmed.
</think>

训练后期 (step 6000~8000) :
<think>
  Let me try approach A... 得出结论X
  Wait, wait. Wait. That's an aha moment.  ← 真实出现在论文里!
  Let me re-evaluate using approach B...
</think>
```

三个未被明确编程的涌现行为:

- 1. 反思行为: 重新审视和重新评估先前步骤
- 2. 验证行为: 主动检查答案正确性
- 3. 替代探索: 自发探索解决问题的替代方法

Aha Moment (顿悟时刻) :

模型学习通过重新评估初始方法来为问题分配更多思考时间  
这体现了强化学习的power and beauty:  
通过简单提供正确的奖励, 大模型会自己找到更优的问题解决策略  
并不需要明确教模型如何解决问题

R1-Zero的缺点:

- 可读性差: 思维过程混乱
- 语言混合: 中文问题可能用英文思考

5.4.6 R1-Zero性能数据

模型	AIME 2024 pass@1	AIME 2024 cons@64	MATH- 500	GPQA Diamond	CodeForces rating
OpenAI-o1-mini	63.6	80.0	90.0	60.0	1820
OpenAI-o1-0912	74.4	83.3	94.8	77.3	1843
<b>DeepSeek-R1-Zero</b>	<b>71.0</b>	<b>86.7</b>	<b>95.9</b>	<b>73.3</b>	1444

5.5 DeepSeek-R1

5.5.1 四阶段训练流水线

基础模型: DeepSeek-V3-Base

↓

阶段1: 冷启动 SFT (几千条长CoT数据)

目标: 学会格式、语言一致、训练稳定

输出: R1-ColdStart 模型

↓

阶段2: 面向推理的 RL

奖励: 准确率奖励 + 格式奖励 + 语言一致奖励

目标: 强化推理能力, 延长思考链

输出: R1-Reasoning 模型

↓

阶段3: 拒绝采样 + SFT

数据: 600K推理数据 + 200K通用数据

目标: 补充通用能力 (写作、问答、翻译)

输出: R1-SFT 模型

↓

阶段4: 全场景 RL

奖励: 规则奖励 (推理) + 奖励模型 (通用)

目标: Helpfulness + Harmlessness

输出: DeepSeek-R1 ☒

### 5.5.2 阶段1: 冷启动SFT

#### 为什么要冷启动?

纯RL从零开始 (没有冷启动) = 冬天冷车直接猛踩油门

症状:

- └─ 早期输出完全混乱, 奖励信号极其稀疏
- └─ 语言混合: 中文问题→英文思考→中文回答
- └─ 格式混乱: 推理过程和答案混在一起

先SFT冷启动再RL = 先热车, 让引擎进入工作温度

解决:

- └─ ☒ 格式稳定: 学会<think>...</think>结构
- └─ ☒ 语言一致: 中文问题→中文思考→中文回答
- └─ ☒ 训练稳定: 有合理初始策略, 梯度更有方向感

#### 冷启动数据收集方法:

- Few-shot prompt (少样本提示)
- 直接prompt with reflection和verification
- 人类后处理DeepSeek-R1-Zero的输出

#### 冷启动数据格式:

```
|special_token|<reasoning_process>|special_token|<summary>
```

每个回答末尾包含总结

过滤掉对读者不友好的回答

### 5.5.3 阶段2: 面向推理的RL

在冷启动模型基础上进行RL训练，RL框架与R1-Zero相同（GRPO）。

引入语言一致性奖励:

```
Reward_language = Num(Words_target) / Num(Words)
```

← 计算思维链中目标语言单词的比例

代价: 准确率略微下降

原因: 语言一致性目标和推理准确性有时冲突

但: 更符合人类偏好, 提高内容可读性

综合奖励 = 准确率奖励 + 语言一致性奖励 (直接相加)

阶段2超参数:

学习率:  $3 \times 10^{-6}$

KL系数: 0.001

GRPO clip ratio  $\epsilon = 10$  (注意: 比通常的0.2大很多!)

采样温度: 1.0

每问题采样: 16个输出, 最大长度32768

批次大小: 32个独立问题, 总batch=512

每400步更新参考模型

### 5.5.4 阶段3: 拒绝采样与SFT

拒绝采样（**Rejection Sampling**）的核心逻辑:

```
def rejection_sampling(prompt, model, n_samples=16):
    """对同一个问题, 采样多个回答, 只保留正确的那些"""
    responses = [model.generate(prompt) for _ in range(n_samples)]

    correct_responses = [
        r for r in responses
        if verify(r) == True          # 答案正确
        and no_language_mixing(r)    # 没有语言混合
        and not too_long(r)          # 没有无意义的重复
    ]
    return correct_responses

# 效果:
# 600K 推理数据: 全部是模型能做对的题
# 训练信号极其干净
```



### 数据构成:

- 推理数据: 600K (设计推理prompt + 拒绝采样 + judge model验证)
- 非推理数据: 200K (写作、事实问答、翻译等, 复用DeepSeek-V3 SFT数据)
- 总量: 800K, 训练2个epoch

### 5.5.5 阶段4: 全场景RL

#### 综合奖励公式:

$$\text{Reward} = \text{Reward\_reasoning} + \text{Reward\_general} + \text{Reward\_language}$$

其中:

$$\text{Reward\_reasoning} = \text{Reward\_rule}$$
 (基于规则, 数学/代码/逻辑推理)

$$\text{Reward\_general} =$$

$$\text{Reward\_helpful}$$
 (若属于有用性数据集)

$$\text{Reward\_safety}$$
 (若属于安全性数据集)

#### 关键工程细节:

Stage4只有1700步训练

最后400步才引入通用指令数据和偏好奖励!

原因:

前1300步: 只用推理数据和规则奖励

- 打稳推理能力基础
- 规则奖励不会导致Reward Hacking

后400步: 引入偏好奖励

- 时间很短, 偏好信号来不及被过度利用
- 避免Reward Hacking!

#### 阶段4超参数:

学习率:  $3 \times 10^{-6}$

KL系数: 0.001

采样温度: 0.7 (比阶段2的1.0更低, 输出更稳定)

批次大小: 512

每400步更新参考模型

### 5.6 奖励模型训练 (升级版报告)

#### 5.6.1 Helpful RM (有用性奖励模型)

##### 数据构建:

- 规模: 66,000对偏好数据
- 方法: Pairwise (成对比较)
- 生成方式: 用Arena-Hard格式, DeepSeek-V3独立查询4次打平均分

- 随机交换A/B位置：防止位置偏见
- 只保留分数差异 $\Delta > 1$ 的样本：确保差异显著
- 控制长度偏差：防止模型学到"长=好"

### 为什么用成对比较？

有用性是主观的，人人打分基准不同

"这个回答有多有帮助？"

标注者A: "3分，有点啰嗦"

标注者B: "5分，很详细"

换成"A和B哪个更好？"

标注者A: "B更好"

标注者B: "B更好"

相对判断消除了主观基准差异 ☒

### 模型架构：

基础模型：DeepSeek-R1

额外组件：顶层加奖励头 (Reward Head)，输出标量偏好分数

公式： $\text{Reward\_helpful} = \text{RM\_helpful}(\text{ResponseA}, \text{ResponseB})$

### 超参数：

- Batch size: 256
- 学习率:  $6e-6$
- 训练轮数: 1轮
- 最大序列长度: 8192 tokens (训练时)，推理时无限制
- 评估重点: 只看最终总结，不干扰推理过程

#### 5.6.2 Safety RM (安全性奖励模型)

##### 数据构建：

- 规模: 106,000条标注数据
- 方法: Point-wise (单独打分)
- 直接标注"安全"或"不安全"，依据预定义安全准则

### 为什么用单独打分？

安全性是客观的，标准一致

"这个回答安全吗？"

标注者A: "不安全，包含有害信息"

标注者B: "不安全，包含有害信息"

标注者C: "不安全，包含有害信息"

大家对"是否安全"容易达成共识

直接点对点打标签更高效 ☒

### 模型架构:

- 基础模型: DeepSeek-R1
- 训练目标: CrossEntropy二分类
- 评估范围: 整个response (包括推理过程)

## 5.7 蒸馏: 赋予小模型推理能力

### 5.7.1 蒸馏方法

直接对Qwen和Llama等开源模型进行SFT

使用的是R1生成的800K条数据

基础模型:

Qwen2.5-Math-1.5B、Qwen2.5-Math-7B、Qwen2.5-14B、  
Qwen2.5-32B、Llama-3.1-8B、Llama-3.3-70B-Instruct

### 5.7.2 为什么蒸馏比小模型直接RL更好?

#### 小模型直接RL的困境:

奖励稀疏问题:

7B模型能力有限

面对难题: 16次采样 → 全部错误 → 奖励全是0

→ 优势值  $A_i = (0-0)/\text{std}(0) = \text{无意义}$

→ 梯度为零, 模型无法更新!

#### 蒸馏的优势:

大模型R1的输出包含完整推理链：

```
<think>
  这道题需要用反证法...
  假设√2是有理数, 设√2 = p/q...
  ... (完整推理过程)
</think>
```

小模型直接学这个完整推理链：

- 不需要自己探索
- 直接学会"反证法"这种推理模式
- 学习的不只是最终答案，而是每一步token的概率分布
- 这就是"隐层含义信息"被传递

数字对比（AIME 2024, Pass@1）：

模型	得分
GPT-4o	9.3
Claude-3.5-Sonnet	16.0
QwQ-32B-Preview	50.0（320亿参数）
<b>DeepSeek-R1-Distill-Qwen-7B</b>	<b>55.5（只有70亿参数!）</b>
DeepSeek-R1-Distill-Qwen-32B	72.6

**蒸馏的上限：**蒸馏只能复制大模型已有的能力，不能超越。突破智能边界仍需更强基础模型和更大规模RL。

5.7.3 蒸馏vs直接RL的总结

- 蒸馏：快速低成本复制已有能力  
小学生照着教授的解题步骤学（方案B）
- 直接RL：探索超越已有能力的边界  
小学生自己摸索（方案A）
- 两者不对立：  
蒸馏 = 让小模型站在大模型肩膀上  
RL = 让大模型突破自身边界

5.8 整体Benchmark测试结果

模型	AIME 2024	Codeforces	GPQA Diamond	MATH- 500	MMLU	SWE- bench
OpenAI-o1-1217	79.2	96.6	75.7	96.4	91.8	48.9
DeepSeek-R1	79.8	96.3	71.5	97.3	90.8	49.2
DeepSeek-V3	39.2	58.7	59.1	90.0	88.5	42.0

DeepSeek-R1在推理任务上实现了与OpenAI-o1-1217相当的性能。

## 6. DeepSeek-Prover-V2

### 6.1 形式化定理证明的挑战

普通数学题 vs 形式化证明的核心区别：

普通数学题（自然语言）：

答案： "大概正确"就能得分

"显然可得..." ← 人类接受这种跳跃

形式化证明（Lean 4语言）：

每一步都必须有严格依据

计算机自动验证： 一个符号错误 = 整个证明无效！

"显然 $a+b=b+a$ "

→ 需要引用交换律定理

→ 需要指定在哪个代数结构上

→ 需要验证类型匹配

搜索空间爆炸问题：

每步有数百种可能的证明策略

一个证明可能需要数百步

总搜索空间  $\approx 100^{100}$  ← 天文数字

### 6.2 核心技术：子目标分解流水线

核心思路： 把一个不可能的大问题，分解成多个可能的小问题。

分工模式：

DeepSeek-V3 (大模型) : 架构师

- 理解定理高层结构
- 分解成合理的子目标 (含sorry占位符)
- 生成证明骨架

DeepSeek-Prover-V2-7B (小专家模型) : 工程师

- 针对简单子目标快速生成策略
- 反复尝试不同战术组合
- 速度快, 可以大量采样

### Lean 4证明骨架示例:

```
theorem induction_ineq (n : ℕ) (h₀ : 4 ≤ n) : n^2 ≤ n! := by

  have base_case : 4^2 ≤ 4! := by
    simp [Nat.factorial] -- 子目标1: 直接计算验证

  have inductive_step : ∀ k ≥ 4, k^2 ≤ k! → (k+1)^2 ≤ (k+1)! := by
    intro k h₁ h₂
    simp_all [Nat.factorial]
    nlinarith -- 子目标2: 数学推导

  -- 子目标3: 组合 (等待小模型填充sorry)
  sorry
```

### 效果:

原始定理搜索空间:  $100^{100}$   
每个子目标搜索空间: 约 $100^{10}$   
难度指数级下降!

## 6.3 冷启动数据生成

### 完整流程:

- 步骤1: V3生成含sorry占位符的Lean代码 (证明骨架)
- 步骤2: 7B专家模型递归解决每个子目标
- 步骤3: 合成完整证明 (子证明组合回骨架)
- 步骤4: Lean编译器验证 (通过=成功, 失败=重试)

**关键:** 一旦子目标被解决, 合成为连贯的思维链数据, 用于后续训练。

## 6.4 课程学习策略

### 朴素方案 (随机顺序) 的问题:

- 模型第一天就遇到费马大定理级别的难题
- 奖励全是0 → 和小模型RL困境一样
- 梯度消失, 学不到东西

## 课程学习（由易到难）：

### 第一阶段：喂简单引理

→ 模型频繁得到正奖励，快速建立基础策略

### 第二阶段：喂中等难度定理

→ 建立在简单引理之上，命中率仍然合理

### 第三阶段：喂复杂定理

→ 此时模型已有足够基础，能把复杂定理分解成已掌握的子目标

## 难度评估的天然来源：

```
def estimate_difficulty(theorem):
    subgoals = decompose(theorem) # V3生成的骨架
    difficulty = {
        'breadth': len(subgoals),          # 子目标数量
        'depth': max_nesting(subgoals),    # 嵌套深度
        'leaf_complexity': avg_leaf_size() # 每个子目标的复杂度
    }
    return weighted_sum(difficulty)
```

## 6.5 一致性奖励

### 强制使用子目标结构的原因：

如果允许跳过子目标直接证明：

模型学到的是"找捷径"，而不是"结构化推理"  
遇到更难定理时，无法分解  
泛化能力崩塌

你说的"逻辑信息被压缩"：

跳过子目标 = 把多步推理压缩成一步

→ 模型内部表示变得不透明  
→ 无法处理更复杂的情况

一致性奖励：

如果最终证明漏掉了V3分解出来的have语句  
→ 给予惩罚  
→ 促使模型保留并使用所有子目标

## 6.6 no-CoT与CoT两阶段训练

### 两种推理模式：

模式	特点	适用场景
no-CoT	直接输出Lean代码，无中间推理	简单定理，速度优先
CoT	自然语言分析 → Lean代码	复杂定理，准确率优先

性能对比（miniF2F, Pass@8192）：

规模	no-CoT	CoT
7B	75.0%	82.0%（+7%）
671B	78.3%	<b>88.9%</b> （+10.6%）

结论：越难的定理，CoT的优势越大；大模型从CoT中获益更多。

6.7 性能对比

miniF2F测试集（Pass@8192）：

方法	通过率
Hypertree Proof Search	41.0%
Goedel-Prover-SFT	64.7%
STP	67.6%
Kimina-Prover	80.7%
<b>DeepSeek-Prover-V2 671B</b>	<b>88.9% SOTA!</b>

PutnamBench（大学竞赛级别）：

- DeepSeek-Prover-V2：解决 49/658 题
- 此前最佳：约10题
- 提升：接近5倍！

7. DeepSeek-V3.2

7.1 论文信息

论文名称：DeepSeek-V3.2-Exp: Boosting Long-Context Efficiency with DeepSeek Sparse Attention

7.2 架构创新



相比V3，V3.2新增两个组件：

V3: MLA (Multi-Head Latent Attention)  
V3.2: MLA + MQA + DSA

MQA = Multi-Query Attention (多查询注意力)  
DSA = DeepSeek Sparse Attention (稀疏注意力)

7.3 MQA (Multi-Query Attention)

注意力家族对比：

类型	Q头	K头	V头	压缩程度
MHA (标准)	128个	128个	128个	无压缩
GQA (V1用)	128个	8个	8个	16倍压缩
MQA (V3.2)	128个	1个！	1个！	极致压缩

```
# MQA的注意力计算：
Q = input @ W_Q # [batch, seq, n_heads, head_dim] (每头独立)

K = input @ W_K # [batch, seq, 1, head_dim] (只有1个头！)
V = input @ W_V # [batch, seq, 1, head_dim]

# 计算时：K、V广播到所有Q头
scores = Q @ K.transpose(-2, -1) # 广播操作
output = softmax(scores) @ V
```

实际工程细节：

论文设计：

- 训练阶段用MHA (保证训练质量)
- 推理阶段用MQA (节省显存)
- 切换方式：把MHA的多个K头通过"均值池化"合并成1个

但实际发布的V3.2-Exp模型：

- inference代码显示 num\_key\_value\_heads = 128
- 实际上还是MHA！
- MQA只是理论设计，工程实现还在探索中

7.4 DSA (DeepSeek Sparse Attention)

核心思路：不对所有历史位置做完整注意力，只挑选最相关的Top-2000个位置。

速度对比：

序列长度: 100K tokens  
完整注意力:  $100K \times 100K = 100$ 亿次运算

DSA:

Lightning Indexer粗筛:  $100K \times \text{低维}$  ← 便宜!  
完整注意力: 只算2000个位置

节省: 省了98%的注意力计算!

#### 7.4.1 Lightning Indexer: 快速找到相关位置

解决"先有鸡还是先有蛋"的悖论:

悖论:

要找出最相关的2000个位置  
需要先和所有位置计算相似度  
但这本身就是完整注意力!

解决方案: 用两个不同精度的操作分工

Lightning Indexer (粗糙但快):  
低维、低精度 (FP8)、ReLU激活  
→ 快速给所有位置打分

完整注意力 (精确但慢):  
只对Top-2000个位置精确计算  
→ 不需要覆盖所有位置

#### Lightning Indexer的核心设计:

```
def lightning_indexer(query_token, all_history):
    # 关键1: 低维向量 (比完整QK便宜100倍)
    q_index = query_token @ W_q_index    # 降维到很小
    k_index = all_history @ W_k_index

    # 关键2: ReLU代替softmax
    # ReLU: 每个位置独立打分, 不需要看其他位置 (快!)
    # softmax: 需要看完所有位置才能归一化 (慢)
    scores = ReLU(q_index @ k_index.T)

    # 关键3: FP8低精度运算 (牺牲精度换速度)
    # (实际实现中使用FP8量化)

    return scores    # 粗糙但够用的相关性估计

# 然后:
top_2000_positions = scores.topk(2000)

# 最后:
final_output = full_attention(query, top_2000_positions)
```

## DSA完整流程:

输入序列: [t1, t2, ... t100K]

当前位置: t\_i

Lightning Indexer (便宜) → 给每个位置打分

↓

Top-K Selection → 选出分数最高的2000个位置

↓

完整注意力 (只算2000个) → 精确的注意力输出

### 7.4.2 精度损失分析

#### 实际损失有多大?

关键观察: 注意力分布本身就是稀疏的!

大多数位置的注意力权重趋近于零

丢掉它们 ~ 丢掉0.049的信息

保留Top-2000 ~ 保留了95%以上的有效信息

RULER基准测试 (专门测长上下文能力):

完整注意力 (基准线): 100%

DSA (Top-2000): 约98%

2%的损失对大多数任务可以接受

#### 最危险的场景:

"大海捞针"任务:

100K token文章里藏着一句关键信息

Lightning Indexer可能打分偏低

被排在Top-2000之外 → 永远看不到!

→ 直接影响答案正确性

## DeepSeek的工程兜底:

```
for layer_id in range(total_layers):
    if layer_id in sparse_layers:
        output = dsa_attention(x, top_k=2000) # 省计算, 轻微损失精度
    else:
        output = full_attention(x)           # 保证精度
# 混合使用, 在速度和精度之间找平衡
```

### 7.5 Cache-Compute Ratio 指标

定义:  $\text{GB/PFLOP}$  = 每做1PFLOP计算, 需要读多少GB的KV-Cache

GB/PFLOP越大 → 越I/O-bound (存储是瓶颈)  
GB/PFLOP越小 → 越compute-bound (计算是瓶颈)

各模型对比（16K-64K上下文）：

模型	GB/PFLOP	说明
Qwen2.5-32B (FP16)	117-267	传统GQA，KV-Cache最大
GPT-OSS-120B	47-95	无特殊KV优化
Qwen3-235B-A22B	39-60	MoE减少计算量
DeepSeek-V3.2 660B	13-36	MLA+DSA，计算减少更多
<b>DeepSeek-V3 660B</b>	<b>4.8-5.8</b>	<b>MLA最优</b>

为什么V3.2比V3的比值更大？

V3.2加入DSA，计算量大幅减少：  
分子（计算量）↓↓  
分母（KV-Cache大小）不变  
→ GB/PFLOP = 分母/分子 ↑↑

这说明：V3.2计算变少，但I/O相对更重要了  
→ 更需要DualPath这样的I/O优化！

7.6 两阶段继续预训练

从DeepSeek-V3.1-Terminus 128K长上下文base ckpt冷启动。

阶段1：密集预热（Dense Warm-up Stage）

目标：初始化Lightning Indexer  
方法：保持密集注意力，冻结除Indexer外的所有参数

为什么要冻结主模型？  
如果全部一起训练：  
主模型每天都在变化  
Indexer追不上 → 目录索引永远无效

冻结主模型：  
书的位置固定 → 专心建立目录 ☑

损失函数：KL散度对齐  
$$L^I = \sum_s p_{t,s} \times \log(p_{t,s}/q_{t,s})$$
  
 $p_{t,s}$ : 主注意力的L1归一化分布（老师）  
 $q_{t,s}$ : Indexer输出的分布（学生）

超参数:

学习率:  $10^{-3}$

训练步数: 1000步

数据量: 2.1B tokens

## 阶段2: 稀疏训练 (Sparse Training Stage)

目标: 让整个模型适应稀疏注意力

方法: 解耦Indexer与主模型的梯度传播

Indexer: 只通过 $L^I$ 优化 (继续对齐注意力分布)

主模型: 通过语言建模Loss优化 (学会稀疏注意力下生成好文本)

为什么要解耦?

两个目标方向可能冲突

解耦后各自优化各自目标, 不互相干扰

超参数:

学习率:  $7.3 \times 10^{-6}$

训练步数: 15,000步 (是阶段1的15倍!)

数据量: 943.7B tokens (是阶段1的450倍)

## 7.7 后训练

### 专家蒸馏 (Specialist Distillation)

构建5个领域专家模型: 数学、编程、逻辑推理等

两种数据生成模式:

thinking mode (长链推理): 复杂任务

non-thinking mode (直接响应): 简单任务

优势:

专家模型数据质量 >> 通用模型数据质量

模型学会根据任务难度自动选择推理深度

### 混合强化学习训练 (Mixed RL Training)

算法: GRPO

创新: 单阶段RL

推理任务 + 代理任务 + 人类对齐 → 同时训练

避免多阶段训练中的灾难性遗忘问题

奖励设计:

代码/数学任务 → 基于规则的奖励

通用任务 → 生成式奖励模型

平衡机制: 长度-准确率、语言一致性-准确率

8. DualPath

8.1 论文信息

论文名称: DualPath: Breaking the Storage Bandwidth Bottleneck in Agentic LLM Inference

核心结论: 离线推理加速1.87×, 在线服务吞吐提升1.96× (实际测试2.25×)

8.2 背景: Prefill和Decode的本质区别

阶段	做什么	硬件特点	瓶颈
Prefill	一次性处理整段prompt	计算密集: 大量矩阵乘法	GPU算力
Decode	逐个生成新token	访存密集: 大量读取KV-Cache	显存带宽

PD分离的必要性:

同一块GPU上的问题:

- Prefill的大计算量抢占Decode的内存带宽
- Decode的频繁小读取打断Prefill的连续计算
- 两个阶段都变慢

PD分离:

- 专门的PE (Prefill Engine) : 多核高算力GPU
- 专门的DE (Decode Engine) : 高带宽显存GPU
- 各司其职, 互不干扰 ☒

8.3 Agent场景的I/O瓶颈

为什么Agent推理这么吃存储带宽?

传统对话 vs Agent:

传统对话: 几轮, 上下文几K tokens

Agent场景 (coding助手) :

- 第1轮: 500 tokens
- 第2轮: 800 tokens
- ...
- 第157轮: 32,700 tokens! (滚雪球式增长)

关键数据 (DeepSeek生产环境coding任务) :

- 平均交互轮数: 157轮
- 平均上下文长度: 32.7K tokens
- 平均每轮新增: 429 tokens
- KV-Cache命中率: 98.7%!

98.7%命中率的含义:

32K token的上下文中:

98.7% (32,271个) : 之前已经算过, 存在外部存储, 只需读出来

1.3% (429个) : 新token, 需要GPU计算

→ 几乎所有时间都花在从存储读旧数据

→ 系统从compute-bound变成I/O-bound!

## 8.4 带宽失衡问题

### PD分离架构下的不均衡:

PE节点的SNIC (存储网卡) :

一直在疯狂读取KV-Cache

利用率: 100% ← 打满, 成为瓶颈!

DE节点的SNIC:

完全空闲

利用率: 0% ← 在睡觉!

原因: 传统设计里只有PE需要从存储读KV-Cache

DE的KV-Cache是PE通过RDMA传过来的

DE根本不需要碰存储

### 硬件趋势三重挤压:

从Ampere到Blackwell:

GPU算力 (FLOPS) : 翻了14倍多

NIC带宽: 只翻了2倍

HBM容量: 缓慢增长

I/O-Compute Ratio下降了14.4倍!

问题只会越来越严重

## 8.5 DualPath的核心思路: 带宽池化

传统只有一条路 (PE Read Path) :

存储 —SNIC—→ PE DRAM —H2D—→ PE HBM → 计算

DualPath新增第二条路 (DE Read Path) :

存储 —SNIC—→ DE DRAM —RDMA/CNIC—→ PE HBM → 计算

DE的SNIC原本在睡觉, 现在被利用起来!

效果: 全集群存储带宽被池化

带宽提升示例 (1台PE配4台DE) :

传统:  $1 \times 400\text{Gbps} = 400\text{Gbps}$

DualPath:  $5 \times 400\text{Gbps} = 2000\text{Gbps} \leftarrow 5\text{倍!}$

## 8.6 两条路径的详细数据流

## PE Read Path (传统路径优化)

步骤1: 存储 → PE DRAM

KV-Cache通过PE的SNIC读到PE的主机内存

步骤2: PE DRAM → PE HBM (逐层, 与计算重叠)

每次只搬一层的KV-Cache

搬完立刻开始该层Prefill计算

GPU在算第L层的同时, CNIC在搬第L+1层

步骤3: PE HBM → DE DRAM

Prefill完成后, 完整KV-Cache通过RDMA传给DE

步骤4: DE DRAM → DE HBM

DE开始自回归Decode

## Layerwise Prefill的收益:

传统方案:

必须把所有60层KV-Cache全部塞进HBM才能开始

= 6GB占用 (假设每层100MB)

→ batch size被迫很小 → GPU利用率低

逐层流水:

每次只持有当前层的KV-Cache (100MB)

算完立刻释放

→ HBM占用从6GB降到100MB

→ batch size可以扩大60倍!

→ GPU利用率大幅提升

## DE Read Path (DualPath新增的路径)

步骤1: 存储 → DE DRAM

KV-Cache通过DE节点的SNIC读到DE的内存

← 这条网卡原本完全空闲!

步骤2: DE DRAM → PE HBM (逐层, 与计算重叠)

通过RDMA (计算网络CNIC) 逐层传输到PE的GPU显存

步骤3: PE传回miss tokens

PE只需计算新增的429个token的KV (miss tokens)

把结果传回DE Buffer, 与已有的hit tokens合并

步骤4: DE DRAM → DE HBM

Decode阶段直接从DE Buffer加载到HBM

DE Path的关键优势:

PE的DRAM完全不参与KV-Cache搬运!

→ PE DRAM压力清零



8.7 流量隔离：Virtual Lane机制

核心问题：

KV-Cache传输需要走计算网络（CNIC）  
但CNIC原本是给推理通信用的（AllToAll、ReduceScatter）  
  
两种流量混在一起，会不会互相干扰？

为什么不用PCIe直连？

PCIe没有QoS（服务质量）机制！  
所有流量无差别竞争带宽，先到先得  
→ KV-Cache传输可能堵住推理通信  
→ 推理延迟爆炸

CNIC-Centric设计：

所有进出GPU的数据都必须经过CNIC！  
即使是本机DRAM → GPU HBM，也走CNIC的RDMA Write绕一圈  
  
为什么绕路更好？  
CNIC是网卡，原生支持QoS  
网卡硬件队列可以区分不同优先级的流量  
PCIe做不到这一点  
  
意外好处：  
RDMA Write提交延迟：~1μs（用户态mmio）  
cudaMemcpyAsync提交延迟：5-7μs（需CUDA驱动栈）  
Layerwise Prefill需要提交数千次 → CNIC方案反而更快！

Virtual Lane流量隔离：

流量类型	VL优先级	带宽分配	类比
推理通信（AllToAll等）	高优先级	~99%	应急车道，优先通行
KV-Cache传输	低优先级	~1%（防饿死）	普通车道，见缝插针

调度算法：加权轮转（Weighted Round Robin）  
每发99个推理数据包，插入1个KV-Cache数据包  
  
推理通信是突发性的（burst），两次burst之间网络大量空闲  
KV-Cache就在这些空隙里“见缝插针”  
← 把原本浪费的带宽利用起来，同时不影响推理延迟  
  
RoCE网络：用DSCP打标 + TC流量分类实现同样效果

## 8.8 自适应调度器

### Inter-Engine调度：PE调度三级优先策略

```
def schedule_pe(waiting_requests, all_pes):
    for request in waiting_requests: # FIFO顺序

        # 第一级：排除过载（算力保底）
        available_pes = [pe for pe in all_pes if pe.token_count <=  $\beta$ ]

        # 第二级：优先选磁盘队列短的PE（存储带宽最大化）
        # 磁盘队列短 = SNIC快要空闲 = 立刻塞请求保持满载
        short_queue_pes = [pe for pe in available_pes if pe.disk_queue <=  $\alpha$ ]

        # 第三级：同级别内选token数最少的（算力均衡）
        if short_queue_pes:
            chosen = min(short_queue_pes, key=lambda pe: pe.token_count)
        elif available_pes:
            chosen = min(available_pes, key=lambda pe: pe.token_count)
        else:
            break # 所有PE都过载，等下一轮

    assign(request, chosen)
```

### 为什么磁盘队列优先于token数？

在Agent场景：

GPU大部分时间在等存储（98.7%的时间是I/O操作）

存储带宽是真正的瓶颈

优化算力 → 收益2%

优化存储带宽 → 收益98%

当然优先保证存储带宽满载！

### 路径选择：

比较PE和DE两侧的磁盘读取队列长度

选更短的一侧读取

→ 哪边空闲就让哪边来读

→ 动态负载均衡

### DE调度两级：

跨组调度：将请求分配到总token数最小的DE group（平衡组间负载）

组内调度：设定高token阈值 $Z = 1.05 \times \text{avg}$ （比组内平均高5%）

优先选低于阈值且HBM剩余充足的DE

## 8.9 Compute Quota：消除GPU气泡

## GPU气泡的成因:

Expert Parallel (专家并行) 模式下:

多块GPU各自负责不同的请求做attention计算  
但必须同步进入FFN层

GPU\_A: 3个短请求, 50ms完成 → 等待250ms ~~zz~~

GPU\_B: 1个超长请求, 300ms完成

GPU\_C: 2个中等请求, 150ms完成 → 等待150ms ~~zz~~

GPU\_A浪费83%, GPU\_C浪费50%!

## Compute Quota解法:

```
def build_batch_with_quota(waiting_requests, quota=300ms):
    batch = []
    total_estimated_time = 0

    for request in waiting_requests:
        estimated_time = predict_attention_time(
            cached_tokens=request.hit_count,
            miss_tokens=request.miss_count
        ) # 基于离线profiling拟合的性能模型

        if total_estimated_time + estimated_time <= quota:
            batch.append(request)
            total_estimated_time += estimated_time
        else:
            # 超限! 用二分搜索找合适的chunk size
            chunk_size = binary_search_chunk(
                request,
                remaining_quota=quota - total_estimated_time
            )
            # Chunked Prefill: 只处理前chunk_size个token
            batch.append(request.chunk(chunk_size))
            break

    return batch
```

**效果:** 同一EP组内各GPU的attention计算时间被控制在差不多的范围内, GPU气泡从83%降到接近0%。

## 8.10 P/D比例安全空间

### 数学推导结论:

设定:

P: PE节点数, D: DE节点数

g=8 (每节点GPU数), s=1 (每节点SNIC数)

B: SNIC带宽, M: DRAM带宽

三个上界约束：

DE CNIC读方向： $P/D \leq (g-2s)/s = 6$   
DE CNIC写方向： $P/D \leq (g-s)/2s = 3.5$   
DE DRAM带宽： $P/D \leq (M/Bs-3)/2 = 3.5$

下界约束（PE SNIC不空转）：

$P/D \geq s/(g-s) = 1/7$

最终安全区间：

$1/7 \leq P/D \leq 3.5$

覆盖几乎所有实际生产部署场景！

DeepSeek实际用的2P4D：P/D=0.5，在安全区间内☑

8.11 实验结果

实验环境：

- 集群：DeepSeek内部InfiniBand集群，最大1152块NVIDIA Hopper GPU
- 每台服务器：8块GPU + 8块400Gbps CNIC + 1块400Gbps SNIC
- 存储：3FS（DeepSeek自研分布式文件系统）
- 基准：真实Agent RL训练trace（生产环境coding Agent任务）

消融实验（离线推理，JCT降低）：

组件	平均JCT降低	做了什么
+Layerwise Prefill	17.21%	GPU每次只装一层KV-Cache，batch size大增
+Dual-Path Loading	38.19%	开辟DE→PE第二条读取路径，池化存储带宽
+Scheduling Algorithm	45.62%	智能调度，平衡NIC和GPU负载

在线服务性能（DS 660B）：

Basic最大吞吐：0.20 Agent/s  
DualPath最大吞吐：0.45 Agent/s  
提升：2.25倍  
  
TTST和TPOT与Basic基本相当（KV-Cache搬运不影响Decode效率）  
TTFT显著降低（存储带宽池化，排队时间大幅减少）

大规模扩展性（24倍扩展）：

2P4D (48 GPU) → 48P96D (1152 GPU) :  
Agent数从2K增加到48K  
JCT几乎不变: 3167s → 3201s (仅增1.1%!)  
→ 近线性扩展 ☒

调度器CPU占用 < 10核, 不是瓶颈

## 9. DeepSeek Engram

### 9.1 论文信息

论文名称: Conditional Memory via Scalable Lookup: A New Axis of Sparsity for Large Language Models

代码: <https://github.com/deepseek-ai/Engram>

### 9.2 核心动机: 两种子任务的浪费

语言建模的两种本质子任务

维度	组合推理	知识检索
典型任务	数学推导、逻辑分析、代码生成	识别命名实体、回忆事实、匹配固定短语
计算特征	动态、深度依赖	局部、静态、高度模式化
理想实现	多层Attention + FFN	O(1)查表
当前实现	MoE条件计算	被迫用计算模拟, 没有专门机制

### 经典案例: 识别实体需要6层网络

识别"Diana, Princess of Wales" (戴安娜王妃) :

Layer 1-2: Wales = 英国的一个地方  
Layer 3: Wales = 欧洲某个国家  
Layer 4: Princess of Wales = 某个王妃头衔 (不确定)  
Layer 5: = 威尔士亲王之妻  
Layer 6: = 戴安娜王妃 (1961-1997) ← 终于认出!

6层Attention+FFN, 只为认出一个众所周知的实体  
这些算力本可以用于真正的推理任务  
却被浪费在"硬背书"上

### Engram不是什么

- **不是RAG**: Engram整个过程发生在模型内部, 不需要外部文档, 不返回原文片段
- **不是KV Cache**: KV Cache是当前上下文的运行时状态; Engram查的是预先训练好的静态记忆表

- **不是取代MoE**：MoE负责组合推理，Engram负责知识检索，两者互补

### 9.3 整体架构：查字典的五步流程

第一步：归一化 (CompressedTokenizer)

Apple/apple/APPLE → 统一成apple

避免查字典时一个词有三个词条

第二步：提取关键词 (N-gram)

把相邻2-3个token组合作为查询词

"Alexander the Great" → 3-gram = (Alexander, the, Great)

第三步：查字典 (多头哈希 + 嵌入表)

用哈希函数定位到嵌入表的位置

O(1)直接取出对应的嵌入向量

第四步：判断是否采纳 (上下文感知门控)

"apple"在科技文 vs 菜谱含义不同

根据当前语境决定是否采纳记忆

第五步：融合 (ShortConv + 残差)

将有用的记忆信息注入Transformer主干

**插入位置**： 只在第2层和第15层插入Engram（不是每层都插入）

为什么是这两层？

第2层（很早）：

在深层计算之前注入静态知识

让后续层不需要浪费算力重建实体

第15层（中间）：

补充第2层可能遗漏的知识

覆盖更复杂的实体识别场景

### 9.4 CompressedTokenizer：词表压缩

**为什么词表压缩对查字典很重要？**

没有压缩时：

"apple" → ID: 12345

"Apple" → ID: 23456 ← 不同ID!

"APPLE" → ID: 34567 ← 又不同!

同一个实体，查出三份不同结果

浪费空间，增加冲突

压缩后：

全部 → 统一ID: 8701

无论大小写，查到同一份嵌入向量 ☒

## NFKC归一化处理内容:

- 全角数字: 1 2 3 → 123
- 罗马数字: Ⅷ → VIII
- 带重音字母: café → cafe
- 特殊符号变体: 统一成基础字符

## 代码实现:

```
class CompressedTokenizer:
    def __init__(self, tokenizer_name_or_path):
        self.normalizer = normalizers.Sequence([
            normalizers.NFKC(),          # Unicode兼容分解
            normalizers.NFD(),           # 规范分解
            normalizers.StripAccents(),   # 去除重音符号
            normalizers.Lowercase(),     # 统一小写
            normalizers.Replace(Regex(r"[ \t\r\n]+"), " "), # 合并空白
            normalizers.Strip(),
        ])
```

效果: 128K词表经压缩后减少约23% (约98K有效词条)

## 9.5 NgramHashMapping: 多头哈希映射

### 为什么不能直接建完整映射表?

3-gram组合数: 词表大小<sup>3</sup> = 100K<sup>3</sup> = 10<sup>15</sup>  
→ 不可能为每种组合都建一个词条  
→ 必须用哈希压缩到有限大小的嵌入表

### 三个关键设计:

#### 关键1: 移位构建N-gram

```
def shift_k(k):
    """将序列向右移动k位, 前面补pad_id"""
    if k == 0: return x
    return np.pad(x, ((0,0),(k,0)), constant_values=pad_id)[: , :T]

# 对于位置t:
# 2-gram = (x[t-1], x[t])
# 3-gram = (x[t-2], x[t-1], x[t])
base_shifts = [shift_k(k) for k in range(max_ngram_size)]
```

#### 关键2: 乘法-XOR混合哈希

```

for n in range(2, max_ngram_size + 1):
    tokens = base_shifts[:n]
    # 核心哈希: 每个token乘以随机奇数系数, 然后XOR混合
    mix = tokens[0] * multipliers[0]
    for k in range(1, n):
        mix = np.bitwise_xor(mix, tokens[k] * multipliers[k])

# 为什么用XOR?
# |—— 计算极快 (位运算)
# |—— 每个位置的token都参与混合
# |—— 顺序敏感: (A,B,C) ≠ (C,B,A) ← 正确!

```

### 关键3: 素数取模 + 多头降冲突

```

# 每个N-gram阶数配K=8个独立哈希头
for j in range(num_heads_for_this_ngram):
    mod = int(head_vocab_sizes[j]) # 质数大小的嵌入表
    head_hash = mix % mod          # 取模得到索引
    all_hashes.append(head_hash)

# 为什么用质数?
# 增大哈希分布均匀性, 减少规则性冲突

# 为什么用多头 (K=8) ?
# 8个独立哈希函数同时查表
# 即使某些头冲突, 其他头不冲突
# 整体信息损失极小
#
# 类比: 8个证人同时作证, 一个记错不要紧

```

## 9.6 Context-aware Gating: 上下文感知门控

### 为什么查到的记忆不能直接用?

哈希冲突可能带来噪声  
 同一个N-gram在不同语境下含义不同  
 ("apple"在科技文 vs 菜谱里)

### 完整门控计算:

```

def context_aware_gating(hidden_state, embeddings):
    """
    hidden_state: 来自Attention之后 (含上下文语义)
    embeddings: Engram查到的静态记忆
    """
    # Key: 记忆向量投影
    key = W_K @ embeddings
    key = RMSNorm(key)

    # Query: 当前上下文 (来自Attention之后, 已整合上下文信息!)

```



```

query = hidden_state
query = RMSNorm(query)

# 相似度计算
similarity = dot(query, key) / sqrt(d)

# sqrt激活 (论文特殊设计, 比sigmoid更敏感)
similarity = sqrt(abs(similarity)) * sign(similarity)

# sigmoid: 把相似度压缩到(0,1)
alpha = sigmoid(similarity)

# 门控调制
value = W_V @ embeddings
output = alpha * value # 相关就用, 不相关就屏蔽

return output

# alpha ≈ 1: 记忆与上下文高度匹配 → 全量注入
# alpha ≈ 0: 记忆与上下文矛盾 → 抑制噪声

```

### 具体场景示例: "我喜欢吃apple派"

"吃"、"派"的上下文 → Query的语义方向: 食物、饮食

知识A (苹果公司: 科技语义):

Query × Key\_A → 方向差异大 →  $\alpha_A \approx 0.08$  → 几乎屏蔽 ✕

知识B (水果苹果: 食物语义):

Query × Key\_B → 方向接近 →  $\alpha_B \approx 0.91$  → 大量注入 ☑

最终: 91%水果苹果 + 8%苹果公司 ≈ 基本只有"水果"语义

### 为什么Query来自Attention之后?

来自Attention之后: 已经看过了"我喜欢吃"和"派", 上下文清晰

来自Attention之前: 只有当前token"apple", 无法区分语义

### 9.7 ShortConv: 深度因果卷积

#### 为什么需要卷积层?

N-gram最大是3-gram: 覆盖当前token + 前2个token = 3个token

但有些实体比3个token更长:

"United States of America" = 4个token ← 超出了!

ShortConv: 把感受野从3个token扩展到10个token

#### 配置:

```
self.conv = nn.Conv1d(
    kernel_size=4,
    groups=total_channels,    # 深度可分离：每个通道独立卷积（计算量极小）
    padding=(4-1)*3,          # 因果：只看过去，不看未来！
    dilation=3,                # 空洞卷积：跳着看，覆盖更大范围
)

# 有效感受野 = 1 + (4-1) × 3 = 10个token
# 覆盖了绝大多数命名实体的长度 ☒
```

### 因果卷积的重要性：

语言模型是自回归的：  
 生成第t个token时，第t+1个不存在  
 如果卷积看了未来 → 训练时能用，推理时崩溃！

因果填充：只在左边填充，右边不填充  
 → 位置t只能看到t之前的token ☒

## 9.8 U型Scaling Law

### 核心实验：给定固定参数预算，MoE和Engram如何分配？

$p=1.0$ ：全部给MoE，没有Engram（纯MoE）  
 $p=0.0$ ：全部给Engram，没有MoE（纯Engram）  
 $p=0.75$ ：75%给MoE，25%给Engram（混合）

### 实验结果：U型曲线，最优点 $p \approx 75-80\%$

纯MoE ( $p=100\%$ ) 的问题：  
 没有专门记忆模块  
 被迫用计算模拟查找  
 → 算力浪费在简单检索上  
 → 复杂推理的算力被稀释

纯Engram ( $p=0\%$ ) 的问题：  
 没有条件计算能力  
 "2+3等于几"→查表→找不到"2+3=5"这条记录  
 → 推理能力丧失

最优混合 ( $p \approx 75\%$ )：  
 MoE (75%)：负责组合推理  
 Engram (25%)：负责知识检索  
 1+1 > 2 ☒

### 无限记忆扩展的Scaling Law：

验证集Loss与嵌入数量呈对数线性关系！  
每增加10倍记忆容量，Loss稳定下降

关键：记忆容量 ≠ 计算量  
更多嵌入 → 存储更多知识模式  
但每个token的计算路径完全不变  
→ 知识容量和计算成本解耦！

9.9 实验结果

实验设置（三者完全公平对比）：

配置	Dense-4B	MoE-27B	Engram-27B	Engram-40B
总参数	4.1B	26.7B	26.7B	39.5B
激活参数	3.8B	3.8B	3.8B	3.8B
训练数据	262B tokens	262B tokens	262B tokens	262B tokens
专家数	-	2+72 (top-6)	2+55 (top-6)	2+55 (top-6)
Engram参数	-	-	5.7B	18.5B

核心结果（Engram-27B vs MoE-27B）：

任务类别	Benchmark	提升
知识检索	MMLU +3.0, CMMLU +4.0, CCPM +7.5	静态事实查询更准
通用推理	BBH +5.0, ARC-Challenge +3.7	复杂逻辑推理增强
代码&数学	HumanEval +3.0, MATH +2.4, GSM8K +2.2	意外提升！

最令人意外的发现： Engram不仅提升了查资料类任务，连代码和数学也提升了！

原因：  
数学题 = 知识检索（识别符号含义）+ 数值推导  
  
没有Engram时：MoE要同时处理符号识别+数值计算  
有Engram时：Engram负责符号识别，MoE专心数值推导  
→ 分工明确，数学能力也提升 ☒

长上下文表现（RULER 32K）：

Multi-Query NIAH (大海捞针多查询) :  
MoE-27B: 84.2  
Engram-27B: 97.0 ← +12.8的巨大提升!

原因: Engram将局部依赖卸载到查表  
释放了Attention带宽用于全局上下文 ☒

9.10 机理分析：三个证据

证据1: LogitLens分析

MoE-27B (没有Engram) :  
Layer 1: Wales = 英国某地方 (模糊)  
...  
Layer 6: = 戴安娜王妃 (终于认出!)  
→ 需要6层才能收敛

Engram-27B:  
Layer 1: = 戴安娜王妃 (直接认出!)  
→ 早期层KL散度显著更低  
→ 第2层的Engram直接注入了知识 ☒

证据2: CKA相似度分析

发现: Engram模型的第5层表示 ≈ MoE模型的第12层表示!

含义:  
Engram在第5层就达到了MoE第12层的表示质量  
相当于"有效深度"比实际层数深得多  
用更少的层数做了更多的事 ☒

证据3: 破坏性实验 (关闭Engram)

任务类型	性能保留率	结论
事实知识 (TriviaQA等)	29-44%	Engram是事实知识的主要存储库
阅读理解 (C3, RACE等)	81-93%	上下文理解主要依赖Attention

功能分离成功! 静态事实存在Engram, 动态推理留在Transformer主干。

门控可视化的发现:

强激活 ( $\alpha \approx 1$ ) 的位置:

"Alexander the Great"、"the Milky Way" 等命名实体  
"By the way"、"Princess of Wales" 等固定短语  
"四大发明"、"张仲景" 等中文成语和历史实体

几乎不激活 ( $\alpha \approx 0$ ) 的位置:

"the"、"is"、"of" 等普通功能词

## 9.11 系统效率：把内存墙变成内存优势

### 确定性预取

#### MoE vs Engram的系统效率对比:

MoE专家调度 (不可预测):

哪个专家被激活? 取决于路由器计算结果  
必须等路由器算完才知道  
→ 无法提前预取  
→ GPU必须等待数据

Engram调度 (完全可预测!):

哪个嵌入被查询? 完全由输入token决定!  
→ CPU可以在GPU计算当前层时, 异步预取下一层的嵌入  
→ 通信与计算完全重叠  
→ 访问延迟几乎为零 ☒

### 推理流水线时序:

GPU计算: [Block 0-14] [Engram Layer2] [Block 2-14] [Engram Layer15]  
CPU预取: [解析hash IDs(L2)] [解析hash IDs(L15)]  
PCIe传输: [Engram嵌入→GPU(L2)] [Engram嵌入→GPU(L15)]  
关键: CPU预取和PCIe传输与GPU计算完全重叠!

### 多级缓存

#### 利用Zipfian分布:

自然语言N-gram服从Zipf分布:

少数高频模式 ("of the", "in the") 占大多数访问  
大量低频模式 (罕见实体) 只占少数访问

缓存层级	存储位置	缓存内容	延迟
L1	GPU HBM	Top高频N-gram	最低 (纳秒)
L2	主机DRAM	中高频N-gram	低 (微秒)
L3	NVMe SSD	长尾低频N-gram	较高但容量极大

极端测试（100B参数卸载到CPU）：

配置	吞吐量	损失
Dense-4B 基线	9,031 tok/s	-
Dense-4B + 100B Engram（CPU卸载）	8,858 tok/s	仅1.9%！
Dense-8B 基线	6,315 tok/s	-
Dense-8B + 100B Engram（CPU卸载）	6,140 tok/s	仅2.8%！

结论：1000亿参数放在便宜的DDR5内存条上，吞吐量损失不到3%！极大降低了扩展记忆的成本。

总结：DeepSeek的核心设计哲学

贯穿始终的主线

从V1到Engram，DeepSeek的核心哲学可以归纳为：

"在给定硬件约束下，通过软件层面的精巧设计，把每一分计算资源、每一分带宽、每一分显存都用在刀刃上。"

具体体现：

1. 计算稀疏化（不是所有参数都参与每次计算）  
V2/V3：MoE细粒度专家切分 + 共享专家  
V3.2：DSA稀疏注意力

2. 存储压缩（不是所有状态都需要完整缓存）  
V2：MLA低秩KV-Cache压缩（64倍压缩）  
V1：GQA多头共享

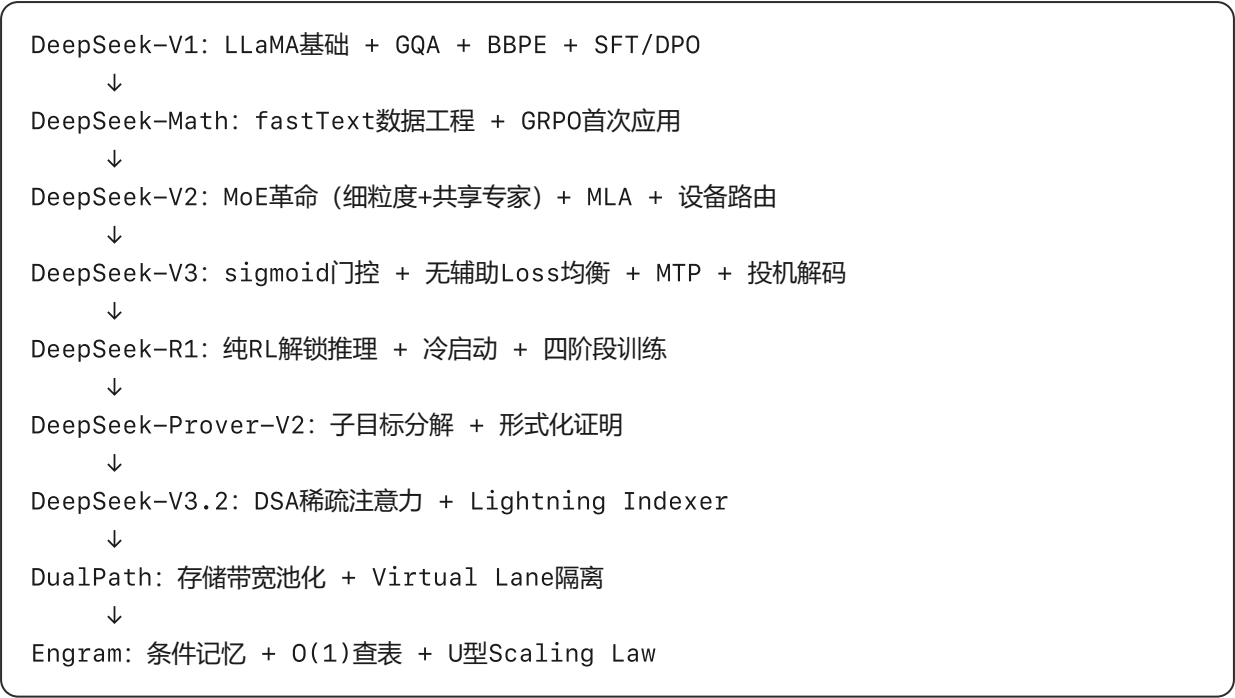
3. 带宽利用（不让任何一块网卡空转）  
DualPath：池化全集群存储带宽

4. 任务分工（不同任务走最适合的路径）  
Engram：知识检索走查表，推理走深层计算

5. 训练效率（不浪费一个训练样本）  
V3 MTP：每个样本同时学多个预测目标  
R1 GRPO：去掉Value Model，降低训练成本

6. 渐进式设计（不一步到位，迭代优化）  
V1→V2：引入MoE和MLA  
V2→V3：消除辅助Loss副作用  
V3→V3.2：探索稀疏注意力  
每一步都解决了上一步的具体痛点

关键技术演进时间线



笔记整理完成。覆盖所有PDF文档的核心知识点，结合完整教学对话的深度解析。