

LLaMA 系列完整深度笔记

覆盖 LLaMA1 → LLaMA2 → CodeLlama → LLaMA3 → LLaMA4

包含模型结构、训练方式、工程实践、前沿探索

目录

1. [LLaMA1](#)
2. [LLaMA2](#)
3. [CodeLlama](#)
4. [LLaMA3](#)
5. [LLaMA4](#)
6. [演进总览](#)

1. LLaMA1

论文：LLaMA: Open and Efficient Foundation Language Models

定位：开源高效基础语言模型，目前主流开源大模型（Dense）基本都是LLaMA架构

1.1 模型结构

相比 GPT，LLaMA1 做出了以下四个核心改动：

Pre-RMSNorm（层归一化）

传统 **Post-LayerNorm** 的问题：

输入 → 注意力/FFN → 输出 → 再做归一化

类比：先跑完100米，再量血压——数值因剧烈运动而波动。

LLaMA1 的 Pre-RMSNorm：

输入 → 先做归一化 → 再做注意力/FFN → 输出

类比：先量好血压，再让你跑步——从一开始就稳定。

RMSNorm vs LayerNorm：

对比项	LayerNorm	RMSNorm
计算内容	均值 + 方差	只有方差（RMS）
计算量	较大	更小
隐含假设	无	激活值天然趋近零均值

RMSNorm 去掉均值计算的合理性：深度网络中，权重初始化（Xavier/He）+ 对称激活函数，使中间层输出天然围绕 0 波动，减去均值几乎没有额外效果，却多花了计算。

公式：

$$y = \frac{x}{\sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2 + \epsilon}}$$

SwiGLU（激活函数）

标准 FFN（ReLU）：

$$\text{输出} = \text{ReLU}(xW_1) \cdot W_2$$

类比：只有开关的水龙头，要么全开，要么关死。

SwiGLU：

$$\text{输出} = (xW_1 \cdot \text{SiLU}(xW_{\text{gate}})) \cdot W_2$$

类比：水龙头 + 调节阀，门控值决定每个维度放多少信息通过。

参数量补偿： SwiGLU 多了一个 W_{gate} 矩阵，LLaMA 通过缩小 FFN 中间维度来补偿：

标准FFN： 2个矩阵，中间维度 $4d$
SwiGLU FFN： 3个矩阵，中间维度 $8d/3 \approx 2.67d$
→ 参数量基本持平

RoPE（旋转位置编码）

绝对位置编码的问题：

给每个位置贴固定标签，训练时见过 1~2K，推理时遇到 2001 就懵了——这个位置标签从未见过。

RoPE 的思路： 不给位置贴标签，而是把位置信息编码进向量的旋转角度里。

位置 m 的词向量：旋转 $m \cdot \theta$ 度
位置 n 的词向量：旋转 $n \cdot \theta$ 度

注意力点积：

$$\begin{aligned} Q \cdot K &= (\text{旋转了 } m\theta \text{ 的向量}) \cdot (\text{旋转了 } n\theta \text{ 的向量}) \\ &= \text{只取决于 } (m-n)\theta \leftarrow \text{相对距离!} \end{aligned}$$

多频率设计（类时钟）：

低维度 → 小 θ → 转得快 → 像秒针 → 感知近距离细粒度差异
高维度 → 大 θ → 转得慢 → 像时针 → 感知远距离粗粒度差异

多个“指针”叠加，才能唯一表示任意位置。若所有维度用同一个 θ ，旋转到 2π 就归零重置，位置 1 和位置 $1+\text{周期}$ 完全一样，模型无法区分。

BPE 分词器

核心思想：合并高频字符对

第一步：统计相邻字符对出现频率
第二步：合并最高频的字符对为一个新token
第三步：重复，直到词表达到目标大小（LLaMA1是32K）

LLaMA1 的特殊处理：

1. 数字强制拆分：所有数字分解为单独数字 token

普通BPE: "123456" → ["123", "456"]
LLaMA1: "123456" → ["1", "2", "3", "4", "5", "6"]

原因：训练时见过 "42"，推理时遇到 "43"，如果整体处理则不认识；拆开后 "4" 和 "3" 都见过。

代价：模型需要隐式维护进位状态，数学计算更难。

2. UTF-8 字节回退：对未知字符回退到字节分解

"🍌" → ["<0xF0>", "<0x9F>", "<0x94>", "<0xA5>"]

意义：没有回退机制时，遇到生僻字只能输出 `<UNK>`，信息丢失；有回退则信息完整保留。

1.2 训练方式

基础的自监督学习模型，没有经过任何形式的特定任务微调。

AdamW 优化器

普通 Adam 的问题： 权重衰减被加入梯度，随后被 Adam 的自适应学习率"稀释"，大权重没有被有效惩罚。

AdamW 的修复：解耦权重衰减

```
# 第一步：正常的Adam梯度更新
weight = weight - lr * adam_update(gradient)

# 第二步：独立的权重衰减（不经过Adam缩放）
weight = weight - lr * λ * weight
```

LLaMA1 具体配置：

```
optimizer = AdamW(
    β1 = 0.9,          # 梯度均值历史权重（90%历史+10%当前）
    β2 = 0.95,         # 梯度方差历史权重（更稳定的步长估计）
    weight_decay = 0.1, # 防止权重过大
    grad_clip = 1.0     # 梯度裁剪，防止梯度爆炸
)
```

梯度裁剪的必要性:

```
if gradient.norm() > 1.0:
    gradient = gradient / gradient.norm()
```

触发场景：训练数据里出现极端样本（格式异常的代码、生僻字文章），loss 暴增，梯度随之暴增。不裁剪则参数直接飞出合理区间，后续 loss 变成 NaN，训练彻底崩溃。

余弦学习率调度

为什么选余弦而不是线性:

线性衰减：匀速下降，前期下降太快，模型还没探索清楚
余弦衰减：前期缓慢→中期快速→后期缓慢，给模型充分探索空间

完整学习率生命周期:

```
def get_lr(step, warmup_steps, total_steps, lr_max, lr_min):
    # 第一阶段: warmup (线性上升)
    if step < warmup_steps:
        return lr_max * (step / warmup_steps)
    # 第二阶段: 余弦衰减
    progress = (step - warmup_steps) / (total_steps - warmup_steps)
    return lr_min + 0.5 * (lr_max - lr_min) * (1 + cos(π * progress))

# 曲线形状:
# lr ↑    /\
# |      /  \_
# |     /    \_
# |    /      \_
# |---warmup---|---余弦衰减---> step
```

Warmup 的作用:

训练最开始参数随机初始化，梯度方向不可信，用极小学习率让各层输出分布趋于正常，再逐步放开学习率大步走。

不同模型大小用不同学习率:

7B/13B 模型: $lr_max = 3 \times 10^{-4}$
33B/65B 模型: $lr_max = 1 \times 10^{-4}$

原因：大模型层数更深，梯度传播时误差累积更多（多层连乘放大误差），且参数耦合更复杂，改动一个引发更长连锁反应，大 lr 容易导致整个网络震荡。

1.3 训练数据

总量 1.4T token，全部公开数据，自监督学习。

数据集	采样比例	Epoch	磁盘大小
CommonCrawl	67.0%	1.10	3.3 TB
C4	15.0%	1.06	783 GB
Github	4.5%	0.64	328 GB
Wikipedia	4.5%	2.45	83 GB
Books	4.5%	2.23	85 GB
ArXiv	2.5%	1.06	92 GB
StackExchange	2.0%	1.03	78 GB

为什么 CommonCrawl 占 67%（而非更多高质量数据）：

1. 数据量天花板：Wikipedia 只有 83GB，重复训练会严重过拟合
2. 多样性比纯净度更重要：CommonCrawl 覆盖人类语言真实场景，让模型"说人话"
3. Epoch 数是真正的质量加权：Wikipedia 训练 2.45 轮，Github 只有 0.64 轮

Github 数据连一轮都没训练完的启示： 代码数据利用率不足，缺乏合成数据构造能力，LLaMA3 通过扩充4倍代码数据+合成代码来解决。

代码数据为何能提升逻辑推理： 代码天然携带严格因果链（if→then）、显式步骤分解（step1→2→3）、零噪声逻辑（对就是对，错就是错），训练后模型学到的是精确推理而非模糊推理。

2. LLaMA2

论文：Llama 2: Open Foundation and Fine-Tuned Chat Models
核心变化：更多训练数据、更长上下文、GQA、完整 RLHF 流程

2.1 模型结构变化

变化点	LLaMA1	LLaMA2
上下文长度	2K	4K
大参数模型注意力	MHA	GQA
FFN 矩阵维度	标准	扩充（增强泛化）
训练数据	1.4T	2T

GQA（分组查询注意力）

KV Cache 的危机：

每生成1个新token → 需要读取所有历史K和V
→ 序列越长，KV Cache占用显存越大
→ $8\text{个头} \times \text{序列长度} \times \text{维度} \times \text{精度} = \text{显存爆炸}$

三种注意力机制对比：

MHA：8个Query头 → 对应8个K头、8个V头（最贵）
GQA：8个Query头 → 共享2个K头、2个V头（分组共享）
MQA：8个Query头 → 共享1个K头、1个V头（最省但效果差）

类比：餐厅每个服务员都有完整菜单（MHA）→ 几个服务员共用一份菜单（GQA）。

GQA 的能力损失： 损失了不同头对 K、V 的差异化关注能力。

- 短上下文、语义简单任务：损失可忽略
- 长上下文、需要同时关注多处细节（如100页法律合同）：损失明显

KV Cache 显存计算：

$\text{显存} = \text{层数} \times \text{组数} \times \text{序列长度} \times \text{维度} \times \text{精度字节数} \times 2(\text{K和V})$

LLaMA4 Scout (1000万token) :
 $= 64 \times 8 \times 10,000,000 \times 128 \times 2 \times 2$
 $= 2,621,440,000,000 \text{ 字节} \approx 2.4 \text{ TB}$
需要约30张H100

2.2 训练数据

预训练使用来自公开可用源的 2T 个数据 token，相当于 LLaMA1 的 1.4 倍。

核心理念："**quality is all you need**"

- 开始时使用公开数据做 SFT，后来改用自有标注数据
- 不同数据源和标注供应商会显著影响下游微调结果
- 数据检查极其重要

2.3 后训练流程

Llama-2-chat 的完整流程：

预训练 → SFT → 奖励模型训练 → RLHF (BoN + PPO) → Llama-2-chat

SFT 阶段

核心洞察： 一万个样本就够了

原因：

预训练阶段：模型已经"见过"几乎所有人类知识
SFT 的真正作用：不是教知识，而是教"对话格式和态度"

SFT 数据堆太多的危害：
→ 模型被过度约束在标注员水平
→ 标注员能力 < 模型潜力上限
→ 潜力被压制

SFT vs RLHF:

SFT = 模仿人类 → 上限是人类
RLHF = 被人类评价 → 上限可超越人类

奖励模型构建

数据必须来自自己的模型：

- ✗ 错误：用 GPT4 的输出
→ 奖励模型只学会判断GPT4风格
→ 遇到LLaMA输出 = 分布外数据 → 打分失效
- ☑ 正确：用自己模型的输出
→ 奖励模型精准感知LLaMA的好坏边界

采样策略： 对不同大小模型做不同 temperature 的采样

7B 模型：temperature=0.7 → 输出偏保守
13B 模型：temperature=0.9 → 输出适中
34B 模型：temperature=1.1 → 输出偏多样
→ 覆盖不同难度样本，奖励模型见识更广

LLaMA2 奖励模型数据统计：

数据源	对比数量
Meta (Safety & Helpfulness)	1,418,091
StackExchange	1,038,480
Anthropic Helpful	122,387
OpenAI WebGPT	13,333

Meta 自收集数据最多的原因：不是资源问题，而是**分布匹配**。OpenAI数据训练的奖励模型只懂GPT风格的好坏，对LLaMA输出是分布外数据。

拒绝采样 (Reject Sampling)

```
for prompt in prompts:
    outputs = model.generate(prompt, n=K)           # 同一prompt生成K个回答
    best = reward_model.pick_best(outputs)         # 奖励模型评分
    sft_data.append(best)                          # 最优答案存入训练集
```

特点：模型权重没有被直接更新，奖励模型只是"筛选器"，无法被针对性作弊。

RLHF 迭代 (共5轮)

RLHF 的哲学：宏观层面的梯度下降

微观梯度下降	宏观 RLHF
loss 函数	人类偏好标注
梯度方向	奖励模型信号
学习率	PPO 更新幅度
一个 batch	一轮迭代数据
时间粒度：毫秒	时间粒度：天/周

迭代策略：

- 第1-4轮：BoN (Best of N)
 - 模型还弱，奖励模型也不够准
 - BoN只是筛选器，无法被作弊
 - 让模型和奖励模型共同成熟
- 第5轮：PPO
 - 模型和奖励模型都已足够稳定
 - 梯度信号可信
 - 做最后一步精准对齐

PPO 不能过早使用的原因：奖励作弊 (Reward Hacking)

- PPO 优化目标：最大化 reward_model(输出) 的分数
- 模型发现捷径：输出超长废话、重复讨好性语句
 - 奖励分数高，但人类觉得很烂

RLHF 迭代收益递减的根本瓶颈：人类标注员辨别能力上限

初级阶段：差距明显 → 人人能判断 → 信号强
高级阶段：差距微小 → 需领域专家 → 普通标注员随机选
→ 奖励模型学到噪声而非信号

破局方向：

1. AI Feedback (RLAIF)：用更强的模型做裁判
2. 任务分解：把"哪个更好"拆成更简单的判断
3. 宪法AI：给模型一套规则让其自我评判

重要结论： LLaMA2 前四次迭代都是简单的 BoN，只有最后一次才是 PPO，说明 RLHF 的核心是 HF（人类反馈），而不是 RL（强化学习算法）。

3. CodeLlama

论文：Code Llama: Open Foundation Models for Code

定位：基于 LLaMA2 训练的代码领域模型

3.1 核心定位：继续预训练

从头训练：随机初始化 → 从零学习所有知识（20年）
继续预训练：LLaMA2基础 → 在代码数据上继续训练（几个月）

类比：培养从小学代码的程序员 vs 让博士转行学编程

混入自然语言数据的必要性： 防止灾难性遗忘（只喂代码数据会让模型忘记怎么说人话）。

3.2 训练流程

起点：LLaMA2（所有尺寸）
↓ 500B代码数据训练（所有尺寸）
↓
├── 7B/13B → Infilling训练
└── 所有尺寸 → 长上下文微调（20B token, 4K→100K）
↓
├── CodeLlama-Python
│ (+100B Python数据)
└── CodeLlama-Instruct
 (+5B指令微调数据)

为什么只做 Python 专项：

语言	在AI场景使用频率	专项训练价值
Python	极高（训练/推理/数据全流程）	最大
C++	中等（CUDA底层/推理引擎）	数据量少，有限
Java	低（与AI场景关系弱）	接近零

专项训练的价值公式：

收益 = 该语言在目标场景的使用频率 × 现有模型在该语言上的能力缺口

3.3 Infilling Task（代码填空）

为什么需要 Infilling：

普通代码生成（自回归）：只能从左到右，给开头续写结尾
 真实编程场景：中间空了需要填入 ← 普通模型完全不会 🤖

数据构造（SPM 格式）：

```
# 从完整代码中随机选一段进行掩码
original = "完整代码..."
prefix = 掩码前的部分
middle = 被掩码的部分（训练目标）
suffix = 掩码后的部分

# 输入格式：
input = f"<PRE> {prefix} <SUF> {suffix} <MID>" # 模型预测 <MID> 部分
```

为什么用 SPM 而不是 BERT 的 [MASK]：

BERT [MASK]：双向注意力，只能预测单个token，不适合生成整段代码
 SPM：
 保持自回归生成，把后缀放到前缀前面输入
 → 模型看到完整上下文后，从左到右生成中间缺失部分
 → "把填空题转化成看着答案范围来续写"

随机掩码策略：

```
def create_infilling_sample(code):
    start = random.randint(1, len(lines)-2) # 随机起始行
    length = random.randint(1, len(lines)//2) # 随机掩码长度
    # 位置和长度完全随机 → 模型必须真正理解代码逻辑
```

为什么只有 7B/13B 做 Infilling：

7B/13B: 部署在IDE插件, 实时代码补全, Infilling是刚需
34B/65B: 用于复杂代码生成和架构设计, 不需要实时补全

Infilling 的局限性: 防御性编程 (如除零判断) 生成不稳定。训练数据从正确代码构造, 掩码位置随机, 有时防御逻辑在前缀里, 模型只需填核心逻辑, 倾向于生成最简单的填充。

工业界解法:

- 1. 数据工程: 提高含防御代码的掩码样本权重
- 2. 指令微调: 显式教会"注意边界情况"
- 3. 后处理: 接静态分析工具兜底 (最实用)

3.4 三个最终产物定位

产品	特点	适用场景
CodeLlama (基础版)	纯代码补全, 无对话能力	嵌入IDE
CodeLlama-Python	Python 专项强化	数据科学/AI开发
CodeLlama-Instruct	自然语言描述需求	对话式编程助手

长上下文的工程价值 (4K→100K) :

跨文件理解整个项目结构 (main.py + utils.py + models/ + tests/), 需要同时看多个文件的完整内容, 4K 完全不够。

4. LLaMA3

- 论文: The Llama 3 Herd of Models
- 包含 LLaMA3 和 LLaMA3.1 两个系列 (差别不大, 多语言/长文本/tool use)

4.1 模型结构变化

变化点	LLaMA2	LLaMA3
Tokenizer	sentencepiece	tiktoken
词表大小	32K	128K
上下文长度	4K	8K (后扩展到128K)
GQA	部分模型	全系列
代码数据	少量	扩充4倍

LLaMA3 主要模型规格:

	8B	70B	405B
层数	32	80	126
模型维度	4096	8192	16384
FFN 维度	14336	28672	53248
注意力头数	32	64	128
KV 头数	8	8	8
词表大小	128000	128000	128000
位置编码	RoPE ($\theta=500,000$)	RoPE ($\theta=500,000$)	RoPE ($\theta=500,000$)

4.2 Tokenizer 升级: sentencepiece \rightarrow tiktoken

词表 32K \rightarrow 128K 的四重影响:

1. **多语言覆盖:** 32K 主要覆盖英语, 其他语言靠字节回退; 128K 可以给中/日/阿拉伯文等分配专属 token
2. **序列压缩效应 (最重要):**

同样文字:
32K词表 \rightarrow 9个token
128K词表 \rightarrow 5个token (压缩44%)

\rightarrow 不改变上下文窗口大小
 \rightarrow 但实际能处理的文本量增加80%
 \rightarrow 相当于免费获得了更长的上下文

3. 推理成本变化:

词表扩大的代价: 输出层对128K个词计算softmax (比32K多4倍)
序列压缩的收益: 序列变短, 省约40%计算
结论: 序列压缩收益 > 词表扩大代价 \rightarrow 整体速度更快

4. **代码处理更好:** tiktoken 保留完整编程关键词 ("for"/"in"/"range"各自是独立token), 而 sentencepiece 会把代码切成很多碎片

4.3 三阶段预训练

阶段一：初始预训练

- └— 数据：15T token 混合数据
- └— 上下文：8K
- └— 目标：建立扎实的通用能力

阶段二：长上下文预训练

- └— 数据：长文本专项数据
- └— 上下文：8K → 128K (逐步扩展)
- └— 目标：让模型适应超长序列

阶段三：退火 (Annealing)

- └— 数据：换成高质量精选数据
- └— 学习率：从当前值降到接近0
- └— 目标：最终精细收敛

为什么要逐步扩展上下文（不能直接从8K跳到128K）：

注意力计算量： $O(\text{序列长度}^2)$

8K: $8000^2 = 6400\text{万}$

128K: $128000^2 = 163\text{亿}$ （贵2500倍）

直接跳的风险：

RoPE位置编码进入从未训练过的角度区间

- 注意力分数分布崩塌
- 模型在8K-128K之间的位置输出乱码

退火的冶金类比：

铸铁急速冷却 → 原子来不及调整 → 结构有内应力 → 刀刃易崩裂
模型直接训 → 参数停在将就位置 → 没充分利用最后优化空间

退火 = 缓慢降温：

- 参数有足够时间在局部精细调整
- 换成高质量数据做最后雕琢（铸剑最后用细磨刀石收尾）

4.4 Scaling Laws

核心发现（OpenAI 2020年）：

$\text{loss} = A / N^\alpha + B / D^\beta + \text{不可消除误差}$

(N=参数量, D=数据量)

- 参数量/数据量翻倍, loss 下降固定比例

LLaMA3 的工程用法：

第一步：在小模型（1B/7B）上训练，记录不同配置的benchmark性能
第二步：拟合每种能力的扩展曲线
第三步：把 405B 参数代入公式，预测最优数据配比
→ 不需要真的训练405B才知道结果

Scaling Laws 的边界：无法预测涌现能力

涌现的本质是多个子能力同时达标，类似木桶装水：

CoT 推理需要同时具备：

子能力A：理解问题语义 → 80% ☒
子能力B：分解成步骤 → 60% ☒
子能力C：每步推理不出错 → 30% ☒ ← 最短板
子能力D：整合多步结果 → 70% ☒

木桶容量 = 最短板 = 30% → CoT 完全失效

当模型增大到子能力C突破60%：

→ 四个子能力同时达标
→ CoT 突然涌现
→ Scaling Laws 看到的是平均loss线性下降
→ 无法预测这种非线性跳变

4.5 数据过滤 Pipeline

四道关卡：

第一关：启发式过滤（快速）

```
# 长度过滤、特殊字符比例、数字比例、重复行比例
if len(doc) < 100 or len(doc) > 100000: return False
if special_char_ratio > 0.2: return False
if digit_ratio > 0.5: return False
if unique_line_ratio < 0.5: return False
```

第二关：NSFW 过滤

用多维分类模型（色情/暴力/仇恨/违法/自伤）打分，针对不同领域（医学/法律/历史）设置不同阈值。

第三关：语义去重

```
# 不能用精确匹配，用向量相似度
vectors = [encoder(doc) for doc in documents]
similar_pairs = find_similar_pairs(vectors, threshold=0.85)
# 保留质量更高的那个，而不是随机删一个
# 用 MinHash+LSH 把复杂度从  $O(n^2)$  降到  $O(n \log n)$ 
```

语义重复数据的三重危害：

- 1. 知识分布失衡，模型对重复内容过度自信
- 2. 死记硬背而非泛化学习，换个问法就不会
- 3. 污染验证集评估（训练/验证集都有相似内容，验证 loss 虚低）

第四关：质量分类器（用上一代 LLaMA 做裁判）

```
quality_model = load_model("llama2") # 上一代模型判断数据质量
```

飞轮效应： LLaMA2 筛选数据 → 训练更好的 LLaMA3 → LLaMA3 筛选数据 → ...

循环偏见风险与防御：

风险	表现	防御方案
偏见放大	每代偏见指数级放大 ($b \rightarrow b^2 \rightarrow b^3$)	多元模型打分 (Ensemble)
错误继承	上代认为低质量的类别被系统过滤	分桶多样性采样（每领域固定配额）
能力天花板	无法超越上代模型的判断能力	人工抽查返工 + 外部基准锚定

最优数据过滤实践：

```
# 不是固定比例，而是质量加权采样
def quality_score(sample):
    return check_logic(sample) + check_consistency(sample) +
    check_format(sample)

sampler = WeightedSampler(samples, weights=[quality_score(s) for s in
samples])

# 分桶保证多样性（每个领域独立筛选，不跨领域竞争）
for domain, docs in domain_buckets.items():
    result.extend(sort_by_quality(docs)[:quota[domain]])
```

4.6 后训练流程

完整迭代循环：

```
for round in range(num_rounds):
    outputs = current_model.generate(prompts, n=K)
    best_outputs = reward_model.select_best(outputs) # 拒绝采样
    preference_pairs = construct_pairs(best, worst)
    sft_model = finetune(base_model, data=best_outputs) # SFT
    dpo_model = dpo_train(sft_model, preference_pairs,
                          mask_shared_prefix=True, # LLaMA3改进
                          nll_regularization=0.2) # LLaMA3改进
    current_model = average_models([dpo_v1, dpo_v2, dpo_v3]) # 模型平均化
```

DPO 改进（针对梯度打架问题）

问题根源：

```
chosen:    "好的，我来帮你...[答案]...<EOS>"
rejected:  "好的，我来帮你...[废话]...<EOS>"
共同前缀: "好的，我来帮你..."
```

DPO loss 同时要求：

- 增加 chosen 中这段话的概率
- 减少 rejected 中这段话的概率
- 同一段话，又增又减 → 梯度互相打架 ✖

两个修复方案：

```
# 修复1: mask 掉共享前缀的 loss
loss = dpo_loss(chosen, rejected, mask_shared_tokens=True)

# 修复2: 加入 NLL 正则项 (系数0.2)
# 确保 chosen 的概率不会因对抗训练而整体下滑
total_loss = dpo_loss + 0.2 * nll_loss(chosen)
```

模型平均化

直接平均 vs Task Arithmetic:

```
# 直接平均 (LLaMA3 用的)
averaged_params = mean([model_A, model_B, model_C])
# 适用条件: 同基础模型、超参数略有不同、参数空间距离近

# Task Arithmetic (更先进)
task_vector_A = model_A.params - base_model.params # 计算相对变化方向
task_vector_B = model_B.params - base_model.params
merged = base_model.params + mean([task_vector_A, task_vector_B])
# 在"方向空间"合并而非"绝对位置空间"
```

每阶段都平均 vs 只在最后平均：

只在最后：误差层层累积放大（RM误差→SFT误差→DPO误差）
每阶段：误差在每层被截断（高质量数据→高质量下游）

学术界更先进的合并方法演进：

- 2022：直接平均
- 2023：Task Arithmetic（方向空间合并）
- 2023：TIES-Merging（解决符号冲突，保留多数同意的方向）
- 2024：DARE（随机丢弃噪声分量）
- 2024：SLERP（球面线性插值，保留特征方向）

为什么 LLaMA3 大量使用 DPO 而不是 PPO

对比项	PPO	DPO
需要模型数量	4个 (Actor/Critic/Reference/Reward)	2个
显存占用	单模型 × 4	单模型 × 2
训练稳定性	极难调参	接近 SFT
工程复杂度	极高	低

5. LLaMA4

官网: <https://ai.meta.com/blog/llama-4-multimodal-intelligence/>
核心变化: MoE 架构、多模态、超长上下文

5.1 三个模型版本

模型	激活参数	总参数	专家数	上下文长度	训练Token
Llama 4 Scout	17B	109B	16	10M	~40T
Llama 4 Maverick	17B	400B	128	1M	~22T
Llama 4 Behemoth	288B	2T	16	-	仍在训练

Scout vs Maverick (推理成本相同, 能力不同) :

两者激活参数都是 17B → 推理FLOPs相同 → 推理速度/成本完全一致

能力差异:

Scout (16专家) : 候选池小, 遇到罕见问题可能无精准专家

Maverick (128专家) : 候选池大, 自发形成更细粒度专业分工

数论专家 / 复分析专家 / 代码专家 / 多语言专家...

MoE 的核心价值: 把"存储成本"和"推理成本"解耦

5.2 文本模型架构

继承 LLaMA2 的特性:

- 前置 RMSNorm
- SwiGLU 激活函数
- GQA (分组查询注意力)

新增特性:

- **QK Norm** (Attention 内部 L2 归一化)
- **iRoPE** (交替 RoPE 编码)
- **MoE 架构** (多个 SwiGLU 专家)

5.3 QK Norm

为什么 MoE 特别需要 QK Norm:

MoE 训练不稳定性:

不同专家更新频率不同 (热门专家频繁, 冷门专家稀少)

→ Q/K 向量的数值范围差异巨大

→ Q@K.T 出现极大值 (如2000) 和极小值 (如0.03)

→ softmax 退化成"只看一个位置"

→ 多头注意力能力丧失 🧠

L2 Norm 的修复:

```
class Llama4TextL2Norm(torch.nn.Module):  
    def _norm(self, x):  
        return x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)
```

归一化后: Q和K向量模长均为1, 注意力分数范围[-1,1] (余弦相似度)

公式: $y = \frac{x}{\sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2 + \epsilon}}$

Scout 用 QK Norm, Maverick 不用的原因:

Scout (16专家): 专家粒度粗, Q/K分布不均匀, 需要归一化兜底

Maverick (128专家): 专家高度专精, 输入分布天然均匀, 无需归一化

且保留向量模长信息 (模长大=该token很确定要找什么)

→ 额外的隐性表达维度

5.4 iRoPE (交替位置编码)

普通 RoPE 的超长上下文危机:

训练: 最长 256K token

推理: 遇到 1000万 token

→ 旋转角度进入从未见过的区域

→ 注意力分数分布崩塌, 输出乱码 🧠

iRoPE 的解法:

```
# 每4层才用一次RoPE
self.use_rope = int((layer_idx + 1) % 4 != 0)

# 不用RoPE的层：用推理时温度缩放代替
if not self.use_rope:
    attn_scales = torch.log(
        torch.floor((cache_position.float() + 1.0) / self.floor_scale) + 1.0
    ) * self.attn_scale + 1.0
    query_states = query_states * attn_scales
```

无 RoPE 层靠什么感知顺序：

第1层 [有RoPE] → hidden_state 携带位置信息
 第2层 [无RoPE] → 继承上层 hidden_state (位置"惯性")
 第3层 [无RoPE] → 继续继承, 做语义加工
 第4层 [有RoPE] → 重新注入/校准位置信息

类比：导航不需要每秒重新定位，定期校准一次，中间靠惯性推算。

iRoPE 的额外好处：

频繁 RoPE 的问题：超长序列时旋转角度累积过大，高频维度转几百圈，数值精度损失
 iRoPE：只有1/4的层做RoPE → 角度累积压力分散 → 长上下文稳定性大幅提升

5.5 MoE 架构

每个 Expert 仍是 SwiGLU 结构（与 LLaMA3 一致）：

```
class Llama4TextMLP(nn.Module):
    def forward(self, x):
        down_proj = self.activation_fn(self.gate_proj(x)) * self.up_proj(x)
        return self.down_proj(down_proj)
```

Shared Expert 设计：

普通MoE：所有expert都参与竞争路由
 LLaMA4： 1个Shared Expert永远激活（处理通用知识）
 + Router选出的Expert（处理专业知识）

Expert Collapse（专家坍塌）问题与防御：

问题：训练初期某几个Expert碰巧被多选

→ 这几个Expert更新更多，变得更强大

→ Router更倾向选它们

→ 其他Expert永远学不到东西 → 退化成2-3个Expert在工作

防御方案一：辅助负载均衡 Loss

```
balance_loss = ((expert_usage - 1/num_experts)2).sum()
```

```
total_loss = main_loss + 0.01 * balance_loss
```

防御方案二：Bias 动态调控 (DeepSeek-V3 方案，更优雅)

```
if expert_i 使用率过高: expert_bias[i] -=  $\delta$ 
```

```
if expert_i 使用率过低: expert_bias[i] +=  $\delta$ 
```

```
# 完全解耦，不干扰主loss的梯度方向
```

5.6 多模态架构

三模块标准架构：

Vision Encoder (MetaCLIP)

↓

Projector (单层MLP) ← 实现简单但有效，本质是LLaVA架构

↓

LLM (LLaMA4文本模型)

```
class Llama4ForConditionalGeneration(Llama4PreTrainedModel):
    def __init__(self, config):
        self.vision_model = Llama4VisionModel(config.vision_config)      #
        self.multi_modal_projector = Llama4MultiModalProjector(config)  #
        self.language_model = Llama4ForCausalLM(config.text_config)      #

class Llama4MultiModalProjector(nn.Module):
    def __init__(self, config):
        self.linear_1 = nn.Linear(vision_output_dim, text_hidden_size,
                                   bias=False)
```

MetaCLIP vs CLIP: 没有模型架构调整，只改进训练数据的质量和分布来提升性能。

5.7 FP8 训练

为什么需要低精度：

Behemoth (2T参数) 用FP32：

参数：8TB + 梯度：8TB + 优化器状态：16TB = 32TB

需要约400张H100 → 工程上不可行

浮点精度对比：

精度	位数	数值范围	显存占用
FP32	32	$\pm 3.4 \times 10^{38}$	基准×1
BF16	16	$\pm 3.4 \times 10^{38}$	基准×0.5
FP8(E4M3)	8	± 448	基准×0.25
FP8(E5M2)	8	± 57344	基准×0.25

FP8 使用策略：

☒

可以用FP8：矩阵乘法输入输出、前向激活值、大部分梯度

☐

必须高精度：优化器状态（Adam动量）、权重更新、Loss计算

核心挑战：动态缩放（防止数值溢出）

```
# 问题：FP8最大值只有448，某层激活值可能出现500 → 溢出 → NaN
# 解决：每个张量有自己的缩放因子
scale = MAX_FP8_VALUE / tensor.abs().max()
fp8_tensor = convert_to_fp8(tensor * scale)
# 使用时还原
original = fp8_tensor / scale
```

分布式训练中的梯度聚合问题：

```
# 问题：不同GPU的梯度在不同scale下压缩，直接相加就像人民币和美元直接加
# 解决：先还原，再聚合
def aggregate_gradients(local_grad, local_scale):
    true_grad = local_grad * local_scale          # 还原到真实值（BF16）
    aggregated = all_reduce(true_grad)             # 跨GPU聚合
    new_scale = compute_scale(aggregated)          # 重新压缩FP8
    return aggregated / new_scale, new_scale
```

FP8 的实际收益（相比BF16）：

- 显存节省：约60%
- 算力提升：H100的FP8算力是BF16的2倍
- 实际加速：约1.5-1.8倍

5.8 MetaP（超参数缩放规律）

传统超参数迁移的问题：

7B模型最优lr: 3×10^{-4}
直接用到70B → 可能完全不对
原因: 参数量增大, 梯度统计特性变化

MetaP 的思路: 找规律而非找最优值

发现: $lr \propto 1/\sqrt{\text{参数量}}$
7B: $lr = C/\sqrt{7B} = 3 \times 10^{-4} \rightarrow$ 求出C
70B: $lr = C/\sqrt{70B} = 1 \times 10^{-4}$ (自动推导)
405B: $lr = C/\sqrt{405B} = ?$ (公式算出)
→ 不需要在大模型上重新调参

MoE 特有超参数不能从 Dense 迁移:

Dense 假设: 所有参数均匀更新 (均匀梯度流)
MoE 实际: 热门Expert频繁更新, 冷门Expert稀疏更新
→ 违反了Dense的均匀假设
→ Dense规律完全失效

MoE 必须重新建立的超参数规律:

- └─ top-k: 激活专家数 (k=1和k=2是质变, 不是量变)
- └─ 负载均衡Loss系数 (太大破坏专家分工, 太小Expert Collapse)
- └─ Shared Expert的独立学习率 (更新频率远高于Routed Expert)
- └─ 路由温度

5.9 后训练流程

总体流程:

轻量SFT → 在线RL → 轻量DPO

SFT 阶段:

用 Llama 模型筛选训练数据
→ 去掉50%被标记为"简单"的问题
→ 只保留较难的问题做 SFT

在线 RL 阶段:

精心选择 medium-to-hard 难度的问题
持续在线RL策略:
→ 交替进行模型训练和提示词筛选
→ 动态保留 medium-to-hard 难度的提示词

DPO 阶段:

针对模型响应质量相关的 corner case 进行轻量 DPO。

5.10 实验结果

预训练模型对比:

任务	LLaMA 3.1 70B	LLaMA 3.1 405B	LLaMA 4 Scout	LLaMA 4 Maverick
MMLU	79.3	85.2	79.6	85.5
MMLU-Pro	53.8	61.6	58.2	62.9
MATH	41.6	53.5	50.3	61.2
MBPP	66.4	74.4	67.8	77.6
ChartQA	-	-	83.4	85.3
DocVQA	-	-	89.4	91.6

指令微调模型对比:

任务	LLaMA 3.3 70B	LLaMA 3.1 405B	LLaMA 4 Scout	LLaMA 4 Maverick
MMMU	-	-	69.4	73.4
GPQA Diamond	50.5	49.0	57.2	69.8
MMLU Pro	68.9	73.4	74.3	80.5
LiveCodeBench	33.3	27.7	32.8	43.4
MGSM	91.1	91.6	90.6	92.3

6. 演进总览

架构演进

特性	LLaMA1	LLaMA2	LLaMA3	LLaMA4
归一化	Pre-RMSNorm	Pre-RMSNorm	Pre-RMSNorm	Pre-RMSNorm + QK Norm
激活函数	SwiGLU	SwiGLU	SwiGLU	SwiGLU (MoE)
位置编码	RoPE	RoPE	RoPE($\theta=500K$)	iRoPE
注意力	MHA	GQA(大模型)	GQA(全系列)	GQA + QK Norm
FFN	Dense	Dense	Dense	MoE
上下文	2K	4K	8K→128K	1M/10M
Tokenizer	sentencepiece(32K)	sentencepiece(32K)	tiktoken(128K)	tiktoken(128K)
训练数据	1.4T	2T	15T	22T~40T
多模态	✗	✗	✗	☑

核心技术演进脉络

稳定性：
Pre-RMSNorm (LLaMA1) → QK Norm (LLaMA4) → FP8动态缩放 (LLaMA4)

效率：
MHA → GQA (LLaMA2) → MoE (LLaMA4)
[显存: 头数×序列长度] → [组数×序列长度] → [激活参数/总参数解耦]

长上下文：
2K → 4K → 8K/128K → 1M/10M
RoPE → iRoPE (每4层用一次)

对齐：
无 (LLaMA1) → BoN+PPO (LLaMA2) → DPO改进 (LLaMA3) → 轻量SFT+在线RL+轻量DPO (LLaMA4)

数据工程：
公开数据 (LLaMA1) → 质量优先 (LLaMA2) → 四道过滤Pipeline (LLaMA3) → AI辅助筛选50% (LLaMA4)

各版本最重要的一个创新

版本	最重要创新	理由
LLaMA1	RoPE	奠定了后续所有版本长上下文的数学基础
LLaMA2	迭代RLHF	证明了"宏观梯度下降"的对齐哲学
CodeLlama	SPM Infilling	把代码填空变成自回归生成，工程上极实用
LLaMA3	数据过滤Pipeline	15T token质量保证，数据工程决定模型上限
LLaMA4	iRoPE + MoE	实现存储/推理解耦，撑起1000万token上下文

笔记整理完成。覆盖文档全部内容，并补充了课程中深度讨论的工程原理、数学推导和工业界实践。