4. 后训练

Table of contents

0.1 本章总览

你可以把后训练理解成“把一个会说话的模型，打磨成一个能稳定干活的系统助手”的过程。本章里的“阶段”指的是典型后训练流水线，不是 LLM 的完整生命周期；在真实工程中，团队会按目标重排甚至跳步，例如先做工具使用 SFT，再做偏好对齐。

先修要求可以简化为三块：第一，你要知道 Transformer 与 next-token 目标在做什么；第二，你要熟悉微调流程，包括数据切分、训练、验证和回归；第三，你要能读懂服务指标，例如延迟、成本、可靠性和失败率。如果这三块都掌握了，本章会更容易读透。

学习目标也很务实：读完后你应当能说清六个阶段分别解决什么问题，面对真实产品约束能快速选出第一优先杠杆，遇到故障时能定位问题来源并提出修复路径，最后还能把训练方案落成可执行的上线评测 gate。

阅读方法建议用一条固定主线：先看当前系统卡在哪里，再看主流方案为什么不够，再看新方法具体改变了哪一层（训练信号、目标函数、推理约束或系统结构），最后评估收益、代价和适用边界。按这个顺序读，你不会停留在“记住名词”，而是能回答“为什么要用、什么时候用、什么时候不用”。

读数学公式时，不必一上来就推导全部细节。你可以先抓四件事：它在优化谁、在哪个范围求平均、有没有惩罚项、哪个超参数在控制强度。本章常见符号可这样读：θ 是模型参数，x 是输入，y 是输出，t 是 token 位置，$\sum$ 表示求和，$\mathbb{E}$ 表示期望，log 常用于把连乘概率改写为连加，KL 用于度量两个分布的差异。只要这四步读清楚，大多数公式都能落到直觉上。

为了在开读前先抓全貌，你可以先记住这条主线：先用 SFT 把行为和格式训稳，再用 PEFT 做低成本专精；随后做偏好对齐与工具训练，把“会说”升级为“会做”；再用推理 RL 追求可验证正确性，最后用蒸馏把能力落到可承受的延迟和成本。

本章使用一个贯穿案例来串联所有阶段：你要做一个财务运营助手，它必须输出严格 JSON 给下游系统，必须会调用工具获取实时或私有数据，必须拒绝高风险操作（如越权转账），还必须支持多业务线的策略差异（多租户）。建议你读每个阶段时都问自己两句：这一步具体解决了这个助手的哪一类问题？它又引入了哪一种新的工程风险？

1 阶段 1：监督微调（SFT）

SFT 是后训练的起点。目标不是“让模型更会说话”，而是让它“按你的产品规则说话”。

1.1 SFT 用于指令遵循与格式控制

预训练模型虽然擅长“续写”，却不擅长稳定执行产品约束，所以在真实系统里常出现 JSON 偶尔失真、拒答风格漂移、工具调用结构不稳等问题。早期团队通常先靠 prompt 工程补救，这在简单场景下能见效，但一旦进入分布变化、长上下文和多轮复杂交互，稳定性就明显下滑。SFT 的关键改动是把“什么叫好回答”直接写进训练数据，让模型不仅学习内容本身，还学习角色、语气、格式与工具 schema；其常见训练目标是最小化助手 token 的 next-token 交叉熵 (Ouyang et al., 2022)：

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = -\sum_{t=1}^{T} \log p_\theta(y_t \mid x, y_{<t}).$$

把这条公式翻成白话是：在每个 token 位置 t，都希望模型给“正确答案 token”更高概率。这里 $p_\theta(\cdot)$ 是当前参数 θ 下的预测概率，y_t 是第 t 个真实 token，$y_{<t}$ 是它之前已经生成的 token，x 是输入上下文；前面的负号表示“概率越大，损失越小”。为什么要用 $\log$？因为整句概率本来是很多小概率连乘，直接算会很小且不稳定；取对数后，连乘变连加，训练更稳定。你可以做个两 token 小例子：若模型给正确 token 的概率分别是 0.8 和 0.4，损失是 $-(\log 0.8 + \log 0.4)$，第二项更差，所以会被更强地推动改进。
这种做法的优势是见效快、可控性强、工程链路成熟，但边界同样清晰：它并不擅长大规模注入新知识（除非数据规模接近持续预训练），而且会放大数据偏差，例如过度拒答、冗长偏置和格式漂移。

在工程里，SFT 数据通常分成四种“训练样本形态”，它们分别对应四类能力训练：单轮指令、多轮对话、工具使用轨迹、安全/风格示范。仍用“财务运营助手”来说明，会更容易看出差异。

第一类是单轮指令数据，结构最简单：一条用户任务配一条助手答案，例如“把差旅报销政策总结成三条可执行规则”。这类数据在序列化时通常是 `instruction -> response`，监督 mask 主要覆盖 `response`，核心目标是把“按要求完成任务 + 按格式输出”训稳定。

第二类是多轮对话数据，重点不在某一句答得多漂亮，而在“连续几轮都不丢上下文”，例如第一轮问“上季度市场部预算是多少”，第二轮追问“那华东区呢”。这类数据会按 `system/user/assistant/...` 串成完整会话，mask 通常覆盖每轮 assistant 回复，主要训练指代消解、上下文记忆和追问衔接。

第三类是工具使用轨迹数据，样本里要显式包含“调用工具前后”的全过程：先输出 `tool_call(如{"tool":"query_erp","args":{"department":"sales","quarter":"Q3"}})`，再接收 `tool_result`，最后给出基于结果的回答。这类数据常见序列是 `assistant(tool_call) -> tool(result) -> assistant(final)`；mask 一般监督工具参数区和最终回答区，而不监督原始工具返回文本，目标是学会四件事：何时调用、调用哪个工具、参数怎么填、如何基于证据作答。

第四类是安全/风格示范数据，用来固定模型边界和表达口径，例如对“帮我绕过审批直接转账”必须拒绝并给出合规替代流程，对“写催款邮件”要保持专业、克制、礼貌。它在序列化上仍然是对话，但标签会明确拒答结构、解释深度和品牌语气；mask 主要覆盖助手输出，目标是把风险边界与语言风格都训成“稳定行为”。

换句话说，这四类数据的区别不在“题目换了”，而在“训练视角换了”。第一，序列化决定你把同一件事按什么顺序写给模型：单轮常是 `instruction -> response`，多轮要写清 `system/user/assistant` 的来回顺序，工具轨迹还要显式写出 `tool_call` 和 `tool_result`。第二，监督 mask 决定哪些 token 会被“计分并反向更新”：通常主要更新 assistant 输出；在工具轨迹里，还会额外更新工具参数区，但不更新原始工具返回文本。第三，行为覆盖范围决定你到底想让模型学会什么能力：单轮偏执行指令，多轮偏上下文连续，工具轨迹偏“调用工具 + 基于证据回答”，安全/风格偏风险边界和表达口径一致性 (Wang et al., 2022)。

理解了 SFT 之后，下一步要解决的是“同一个行为是否能在训练和推理阶段被模型一致读取”。

1.2 聊天模板契约（Chat Template Contract）

聊天模板契约要解决的是“训推格式不一致”这一类隐蔽但高频的问题：同一个模型如果训练时用一种消息序列化格式、推理时用另一种，就会出现角色边界混乱、补全异常、工具调用失败。

过去很多团队把模板当作临时实现细节，靠字符串拼接快速上线，短期可跑但长期难复现。新方法的核心是把模板上升为接口契约，要求数据处理、训练、推理三端使用同一套规则；形式上可写为 (Hugging Face, n.d.)：

$$s = T(\{(r_i, c_i)\}_{i=1}^{n}),$$

把这个公式拆开看会更清楚：$\{(r_i, c_i)\}_{i=1}^{n}$ 表示一段会话里共有 n 条消息，第 i 条消息由两部分组成，r_i 是角色 (如 `system`、`user`、`assistant`、`tool`)，c_i 是该角色说的内容。$T(\cdot)$ 是一个“固定格式化规则”，它的作用不是改写语义，而是把这些结构化消息按约定顺序加上边界标记，拼成唯一的文本序列 s。s 随后进入 tokenizer，被切成 token 并映射为 token id，才真正成为模型看到的输入。

可以看一个完整的小例子。假设输入消息是：(`system, "` 你是财务助手`")`、(`user, "` 查询销售部 Q3 支出`")`、(`assistant, "<tool_call>{...}</tool_call>")`、(`tool, "{\"amount\": 1200000}"`)。模板函数可能把它序列化成：`<|system|>` 你是财务助手 `<|user|>` 查询销售部 Q3 支出 `<|assistant|><tool_call>{...}</tool_call><|tool|>{"amount":1200000}<`
注意，这里关键不是“意思差不多”，而是“符号必须完全一致”。如果训练时用 `<|assistant|>`，推理时换成 `[assistant]`，tokenizer 切出来的 token 序列会变，模型就可能把角色边界读错，出现答非所问或工具调用失败。

所以模板契约的本质是“同样的结构化输入，必须稳定映射到同样的 token 序列”。它的直接收益是减少难复现的线上回归，并让 SFT、DPO、RL 训练结果可比较；代价是工程耦合更强，你必须把模板版本与 tokenizer 版本一起管理，并做快照与回归测试。Hugging Face 文档也明确提示：控制 token 的偏差会显著降低聊天模型表现 (Hugging Face, n.d.)。

当模板契约固定后，下一个关键点就是把训练梯度集中到“模型真正该学的输出 token”上。

1.3 仅补全损失（Completion-only Loss）

仅补全损失要解决的问题是“训练目标错位”：如果对输入和输出所有 token 都计算损失，模型会学会复述提示词，而不是专注回答问题，进而造成容量浪费和 prompt-echo。过去常见做法是全量 token 监督，虽然实现最简单，但在多轮对话中会被用户文本和系统脚手架严重稀释信号。新方法通过监督掩码 (mask) 只训练目标输出 token (通常是 assistant token)，将梯度聚焦到真正要生成的内容上；常见写法是 (Hugging Face TRL, n.d.)：

$$\mathcal{L}(\theta) = -\sum_{t=1}^{T} m_t \log p_\theta(y_t \mid x, y_{<t}),$$

这里新增的关键是 m_t，它像一个“开关”：$m_t = 1$ 表示这个位置要算损失，$m_t = 0$ 表示跳过。你可以把它理解成只批改“答案区”，不批改“题目区”。例如一条样本长度 10，前 6 个 token 是用户问题、后 4 个 token 是助手回答，那么前 6 个位置可设 $m_t = 0$，后 4 个设 $m_t = 1$。这样梯度只来自目标输出，训练信号更干净。
它的优势是同等算力下训练更高效、格式保真更好，但代价是对数据预处理更敏感：一旦 mask 边界出错，训练过程看起来正常，实际几乎学不到目标行为，所以必须用单测核查每条样本的有效监督 token 数量。

```
# Pseudocode: SFT with masking (PyTorch-style)
```

```python
def compute_sft_loss(model, input_ids, labels, vocab_size):
    # labels: non-target tokens are -100
    logits = model(input_ids).logits
    shift_logits = logits[..., :-1, :].contiguous()
    shift_labels = labels[..., 1:].contiguous()
    return F.cross_entropy(
        shift_logits.view(-1, vocab_size),
        shift_labels.view(-1),
        ignore_index=-100,
    )
```

把前面几个概念放在同一条 SFT 流水线里看，会更容易理解它们各自的作用：四类数据形态决定“你喂给模型哪些真实任务场景”，也就是能力覆盖面；聊天模板契约决定“这些场景被翻译成什么 token 序列”，保证训练和推理读到的是同一种输入语言；completion-only mask 决定“模型在这些 token 上到底学什么”，把梯度集中在 assistant 应该输出的部分。三者配合后，SFT 才会从“数据很多但效果不稳”变成“数据、格式、优化目标彼此对齐”的可控过程。换句话说，数据形态负责广度，模板负责一致性，mask 负责学习焦点，缺任何一环都会出现看似训练正常、上线却不稳定的问题。

当基础行为监督完成后，后训练的重点会转向“如何在有限成本下做可规模化的多版本专精”。

这一阶段的核心问题是：模型能否稳定产出产品要求的交互行为。工程上最关键的控制点是模板一致性和监督 mask 的边界正确性；最常见误区则是还没检查序列化和数据边界，就先盲目增加 SFT 数据量。

---

2 阶段 2：参数高效微调（PEFT）

PEFT 的核心目标是“低成本专精”。它让你在不复制整模的前提下维护多个版本。

2.1 LoRA / QLoRA

LoRA/QLoRA 针对的是“全量微调太贵”这个现实问题：每个专用版本都需要完整 checkpoint，训练、存储和部署成本都很高。过去团队通常在“全量微调”和“仅靠 prompt 适配”之间二选一，前者成本高，后者稳定性又不足。LoRA 的核心改动是冻结底模，只学习低秩增量参数 (Hu et al., 2021)：

$$W' = W + \Delta W, \quad \Delta W = AB,$$

可以先把 LoRA 想成“底模不动，只加补丁”。全量微调像是把整本教材重写一遍；LoRA 像是在页边加少量批注，用这些批注去修正模型行为。这样做的关键收益是：你不用改动大部分参数，也能让模型学到新任务。

放到公式里看，$W' = W + \Delta W$ 表示新权重等于“原权重 + 改动量”；而 $\Delta W = AB$ 表示这份改动量不是直接存整块矩阵，而是拆成两个更小的矩阵 A, B 相乘。假设原权重矩阵大小是 $d \times k$，全量微调要训练 dk 个参数；LoRA 只训练 $A(d \times r)$ 和 $B(r \times k)$，参数量变成 $r(d + k)$。只要 r 远小于 d, k，参数和显存都会大幅下降。

看一个具体数字会更直观：若 $d = k = 4096,\ r = 16$，全量需要约 $4096^2 \approx 1678$ 万参数；LoRA 只需 $16 \times (4096 + 4096) = 131072$ 个增量参数，量级接近百倍差距。QLoRA 则在 LoRA 基础上再进一步：把“冻结的底模”压到低比特（常见 4-bit）来省显存，只让 LoRA 增量部分保持较高精度参与训练（Dettmers et al., 2023）。
这类方法的优势是显存占用低、迭代速度快、便于维护多版本；代价是深层能力改造存在上限，若任务需要大范围权重迁移，效果可能不如全量微调。

工程上可以把 LoRA 的调参点理解成三个问题：第一，改哪里；第二，改多少；第三，怎么上线。
“改哪里”对应挂载模块。把 LoRA 挂在 attention 上，通常更直接影响信息选择和指令跟随；原因是 attention 的核心作用就是“当前 token 该看上下文里的哪一部分”，它直接决定模型是否抓住关键约束、关键实体和关键步骤。比如同一段输入里既有背景信息也有“必须输出 JSON”这样的硬约束，attention 的改动会更快体现在“它先看哪里、遵循哪条指令”。挂在 MLP 上，通常更影响表达风格和任务特化；原因是 MLP 更像“把已经取到的信息做非线性变换并组织输出”，对措辞、语气、领域表达模式和任务映射影响更明显。很多团队会先从 attention 起步，再按评测结果决定是否扩到 MLP。经验判定准则可以写成一句话：如果主要问题是“指令跟随不稳、格式约束抓不住、工具参数常错”，attention-only 通常先够用；如果这些问题已经基本稳定，但“术语表达不自然、文风不一致、任务特化不足”仍明显，再加 attention+MLP 往往更有效。
“改多少”对应秩 r。r 可以理解为补丁容量：r 小，参数更省、训练更快，但表达能力可能不够；r 大，能力上限更高，但显存和过拟合风险也会上升。
“怎么上线”对应部署方式。`merge` 是把 adapter 合并进底模，优点是推理链路简单、延迟低；缺点是每次换版本都要重新合并。在线叠加是不合并、按请求加载 adapter，优点是灵活、适合多租户；缺点是会增加加载和调度开销。

有了低成本参数增量后，系统层面下一步就会自然过渡到“如何把这些增量安全、高效地服务给多租户”。

2.2 Adapter 多租户

把“Adapter 多租户”放在 PEFT 这一章，是因为它本质上是 PEFT 的落地形态，而不是一套全新的训练算法。PEFT（比如 LoRA）先把“每个客户/任务的差异”做成很小的参数增量；Adapter 多租户再解决“这些小增量如何在生产环境里按客户正确切换”。如果没有 PEFT 先把改动压小，系统通常只能回到“每客户一整模”，成本会随客户数快速上涨。

因此，LoRA 和 Adapter 多租户是上下游关系。LoRA 负责把改动做成小而独立的 adapter，像在统一底模上制作多个“专属配置包”；Adapter 多租户负责在请求到来时把正确的配置包分发给正确的客户，像一个按租户路由的配置分发系统。一个偏训练参数构造，一个偏在线服务架构，两者合起来才能既省成本又可规模化。

Adapter 多租户要解决的核心矛盾是：客户数量增长时，如何避免模型副本爆炸。最直白但最贵的做法是每个租户部署一整套模型，隔离简单，但存储、显存占用、发布与回滚都会近似线性增长。更实用的做法是“一个共享底模 + 多个小 adapter”：底模共用，差异通过 adapter 表达，请求到来后按租户信息选择对应 adapter。形式化写法为：

$$i = g(z), \quad \pi_{\theta,i}(y \mid x) \text{ uses base } W \text{ plus adapter } a_i.$$

把这条公式拆开读，会更容易理解。第一部分 `i = g(z)` 的意思是“先选包”：z 是请求头里的租户信息（如 `tenant_id`、行业、策略版本），$g(\cdot)$ 是路由规则，它输出一个 adapter 编

号 i。第二部分 $\pi_{\theta,i}(y \mid x)$ 的意思是“再推理”：在输入 x 下，系统使用“同一个底模参数 W + 第 i 个 adapter a_i”来产生输出 y 的概率分布。你可以把它理解成：底模负责通用能力，adapter 负责该租户的策略与风格偏置。

看一个更具体的例子。假设三家客户共用同一个财务底模：`bank_A`（严格合规）、`retail_B`（效率优先）、`saas_C`（解释更详细）。它们分别对应 adapter 17、42、9。用户问题相同：“请生成本月应付账款摘要，并告诉我是否可以跳过二次审批。”请求进入系统后，先读取 z。若 $z = (bank_A, policy_v3)$，则 $g(z) = 17$，加载 adapter 17，模型会强调“不可跳过二次审批”；若 $z = (retail_B, policy_v2)$，则 $g(z) = 42$，模型可能给出“在金额阈值内可走简化流程”；若 $z = (saas_C, policy_v1)$，则 $g(z) = 9$，模型会给出更长的解释和风险提示。注意，这里用户问题完全相同，差异来自“选中的 adapter 不同”。

这也解释了为什么“路由选错”是多租户里最危险的问题：模型可能语法完全正确、逻辑也通顺，但它执行的是另一家客户的策略。比如 `bank_A` 的请求被误路由到 42，系统可能输出本不允许的审批建议。工程上，这类错误往往比普通答错更难被肉眼发现，因为文本本身看起来很像“正常答案”。

这种架构的收益是成本下降、迭代加快、回滚更轻；代价是路由正确率和版本管理会升级为生产级安全边界，所以必须做三件事：adapter 级回归测试、租户切片评测、路由审计日志。常见部署模式包括按请求动态加载、GPU 热驻留、按 adapter ID 分批、单租户 merge 降延迟；本质上是在延迟、吞吐和显存驻留之间做工程权衡。

因此，这一阶段真正要回答的问题不是“能不能训练出 adapter”，而是“能不能在不成倍增加成本的前提下，把大量小版本稳定服务出去”。常见误区是把 PEFT 误解成“可以少做回归测试”，结果在多租户场景出现隐性行为漂移。可以把这一节浓缩为一句话：PEFT 让你“做得出很多小版本”，Adapter 多租户决定你“送不送得对、送不送得稳”。

当多版本专精问题被控制住后，下一阶段的重点会从“能不能专精”转向“回答是否更符合人类偏好”。

---

3 阶段 3：偏好对齐（对话与风格）

偏好对齐关注“用户更喜欢什么”，不是只关注“语言模型更像训练分布”。

先用一个贯穿例子进入本阶段。假设你的财务运营助手面对同一个问题：“请总结本月现金流风险，并告诉我是否建议暂停非核心采购。”模型 A 的回答很“像人写的报告”：先给结论，再列证据（应收账款回款变慢、库存周转下降、下周有大额应付款），最后给分级行动建议和执行优先级；模型 B 也能给结论，但表达泛泛、缺证据、没有明确行动顺序。多数业务用户会更偏好 A，因为它更可执行、更可追责，也更便于管理层直接决策。

阶段 3 做的事，就是把这种“用户更偏好的回答方式”系统地学进去。后面的三节分别对应三步：先在偏好学习里收集和利用这类 A/B 比较信号，再用 KL 约束防止模型为了拿高分而把合规边界学坏，最后用 DPO/ORPO 在离线数据上以更低成本把这种偏好稳定固化。

3.1 偏好学习（Preference Learning）

偏好学习要解决的是 SFT 后常见的“答得对但不好用”：比如语气生硬、风险提示不当、帮助性不足。过去常见补救方式是继续堆 SFT 示范，但“最佳答案”往往不唯一，单一标签很难表达用户偏好差异。新方法把监督形式改成成对比较，让标注者判断哪个回答更好（Ouyang et al., 2022），模型学习的目标可写为偏好概率：

$$\Pr(y^+ \succ y^- \mid x).$$

这个概率可以读成：在同一个问题 x 下，回答 y^+ 被偏好为“好于” y^- 的可能性。若这个值接近 1，说明模型偏向多数人更喜欢的回答；若接近 0.5，说明两者难分。你可以把它想成作文打分中的“二选一胜率”，比“唯一标准答案”更符合对话系统。

举一个更具体的财务助手例子。设问题 x 是：“请评估本月现金流风险，并给出三条可执行建议。”候选回答 A（可视作 y^+）先给风险等级，再列关键依据（应收账款周转、到期应付款、库存变化），最后给三条可执行动作并标出优先级；候选回答 B（可视作 y^-）虽然语法正确，但只有笼统结论“风险可控、建议加强管理”，没有证据链和落地步骤。若 10 位标注者里有 8 位选 A，那么经验上可理解为 $\Pr(y^+ \succ y^- \mid x) \approx 0.8$。偏好学习就是在大量这样的“同题二选一”样本上训练，让模型逐步学会更像 A 这种“有依据、可执行、表达清晰”的回答风格。

这种做法更贴近真实用户体验，但边界也很明确：偏好数据会引入标注偏差，而且“更受喜欢”并不总等于“更正确”。

因此接下来还需要一个机制，既能优化偏好，又能约束模型不要偏离太远。

3.2 带 KL 约束的奖励最大化

带 KL 约束的奖励最大化要解决的是一个很现实的矛盾：模型要“学得更合偏好”，但又不能“学偏到面目全非”。如果只盯奖励分，模型会想办法钻评分器空子（reward hacking）；如果只求稳定不敢更新，又学不到新偏好。这个方法的核心就是把“变好”和“别跑偏”放进同一个目标里（Ouyang et al., 2022）：

$$\max_{\pi_\theta} \mathbb{E}_{y\sim\pi_\theta(\cdot|x)}[r(x,y)] - \beta\,\mathrm{KL}(\pi_\theta(\cdot \mid x)\|\pi_{\mathrm{ref}}(\cdot \mid x))\,.$$

这条式子可以按“油门 + 刹车”来读。前半项 $\mathbb{E}[r(x,y)]$ 是油门：鼓励模型给出更高奖励的回答；后半项 $\beta \cdot \mathrm{KL}(\pi_\theta\|\pi_{\mathrm{ref}})$ 是刹车：不让新策略 π_θ 偏离参考策略 π_{ref} 太远。这里的 KL 可以理解成“两个回答习惯的距离”，KL 越大，说明新模型说话方式、拒答边界、长度分布等行为变化越大。β 就是刹车力度：β 小，允许更激进更新；β 大，更新更保守。

举一个财务助手例子：你希望模型“更主动给可执行建议”（奖励上升），但不希望它因此减少合规提醒（行为漂移）。如果某轮训练里奖励提升了 $+1.2$，同时 KL 为 0.5，那么当 $\beta = 0.1$ 时惩罚是 0.05，净收益仍是正，更新会被接受；当 $\beta = 3$ 时惩罚变成 1.5，就会压过奖励提升，系统会倾向收缩更新。这就是为什么同一批数据，β 不同会得到明显不同的模型性格。

读这类训练曲线时，可以先记住一个通俗区分：你主动优化的指标叫“奖励”，你没打算改变却发生变化的行为叫“漂移”。比如你想提升“建议可执行性”，这是奖励目标；如果同时出现“拒答边界变松、回答异常变长、JSON 有效率下降”，这就是行为漂移。

工程上可以用“双指标法”判断模型到底有没有真的变好：一组看目标收益（偏好胜率、任务成功率、verifier 通过率），一组看护栏稳定（KL、拒答率/安全指标、格式有效率、长度分布）。

四种常见结果可以这样解读：目标指标上升且护栏稳定，通常是真提升；目标指标上升但护栏下滑，常见是奖励投机或漂移；目标指标不变但 KL 持续变大，属于无效漂移；目标指标和护栏同时变差，则是明显退化。这样看，你就不会被“奖励分好看”误导，而能判断模型是在“真实变好”还是“只在评分器上变好”。

有了这个统一目标后，工程上下一问通常是：能不能不用完整 RLHF，也吃到偏好数据的主要收益。

这个问题之所以重要，是因为完整 RLHF 往往“贵在整条链路”而不只是贵在某一项算力。它通常需要四层投入：先做人类偏好标注（持续采集和质检都贵），再训练奖励模型，再进行大规模在线 rollout 生成新样本，最后用 PPO 等 RL 方法反复更新策略并做稳定性调参。与普通离线训练不同，RLHF 常是 on-policy 流程，模型一更新，旧数据价值就会下降，意味着需要持续“边采边训”，推理算力和训练算力同时吃紧。此外，为了防止 reward hacking 和安全回归，还要频繁跑红队评测与回归套件，工程人力成本也显著上升。

所以很多团队不是“反对 RLHF”，而是按阶段采用：先用 DPO/ORPO 在离线偏好数据上拿到大部分可见收益，建立评测和监控闭环；当离线方法收益趋于平台期、且业务确实需要更强在线探索时，再升级到完整 RLHF。这样做的本质是先控制成本和风险，再逐步提高优化强度。

3.3 DPO / ORPO（离线偏好优化）

DPO / ORPO 可以用一句话理解：你已经有“同一题里哪个回答更好”的投票结果，就直接用这些投票训练模型，而不先搭完整 RLHF 流水线。它们最适合的场景是：偏好数据已经有了，但你希望先用更低成本拿到稳定收益 (Rafailov et al., 2023; Hong et al., 2024)。

把它和 RLHF 对比会更清楚。RLHF 常见路径是“先训练奖励模型，再做在线 RL 更新”；DPO/ORPO 的路径是“直接在离线偏好对上优化”。所以 DPO/ORPO 通常更轻量、迭代更快，也更容易先搭出第一版偏好对齐系统。

具体到训练数据，每条样本通常是三元组：问题 x、更好的回答 y^+、较差的回答 y^-. 模型要学到的不是“唯一标准答案”，而是“在同一问题下把好答案概率抬高、把差答案概率压低”。这就是下面公式在做的事：

$$\Delta_\theta(x) = \log \pi_\theta(y^+ \mid x) - \log \pi_\theta(y^- \mid x), \quad \text{train to increase } \Delta_\theta(x).$$

$\Delta_\theta(x)$ 就是“偏好间距”。它越大，说明模型越偏向 y^+；它为负，说明模型还在偏向 y^-. 例如同一题下，模型给 y^+ 的概率是 0.6、给 y^- 的概率是 0.2，那么 $\Delta = \log 0.6 - \log 0.2 = \log 3 > 0$；训练会继续把这个间距拉大。用对数的好处是优化更稳定，便于训练。

DPO 和 ORPO 的差异可以先这样记。DPO 通常有一个“参考模型锚点”，更强调不要偏移过快；ORPO 更强调单阶段目标，把监督学习和偏好优化并在一起，流程更紧凑。实务上，若你更看重训练稳定和漂移可解释性，常先试 DPO；若你更看重流程简化和训练效率，常把 ORPO 作为候选。

这类方法的优点是：实现相对简单、离线数据即可起步、收敛通常更平稳。边界同样明显：它学到的能力上限受偏好数据覆盖限制，而且“更受喜欢”不一定等于“更正确”。因此评估时不能只看偏好胜率，还要同时看任务正确率、安全指标和分布漂移；必要时要配 verifier 或任务评测来补上正确性约束。

当偏好层稳定后，下一阶段就会进入“模型如何在真实系统里可靠调用工具”这个更工程化的问题。

这一阶段的核心是把优化目标从“语言似然更高”转向“用户真实更满意”。落地时最关键的控制点是偏好数据覆盖度、标注质量和漂移约束；最常见误区是把偏好胜率提升直接等同于正确性提升，忽视了“好看答案”和“正确答案”之间的差距。

---

4 阶段 4：工具使用与 RAG

这一步把模型从“会说”推进到“会调用系统完成任务”。

以财务运营助手为例，用户问：“请给出本周现金流预警，并判断是否需要延后市场投放预算。”如果模型只“会说”，它可能给出一段看起来合理但缺乏依据的建议；因为它并没有实时拿到应收账款、应付账款、银行余额和预算执行率。这样的回答语言流畅，但决策风险很高。

进入工具使用与 RAG 阶段后，流程会变成“先查数据，再下结论”：先调用 ERP/资金系统接口拉取最新余额与账期，再调用预算系统读取投放执行进度，必要时通过 RAG 检索内部财务政策，然后基于这些证据生成结论和行动建议。这样得到的回答不只是“像对”，而是“有数据来源、可追溯、可执行”。

4.1 工具使用训练（Tool-use Training）

工具使用训练解决的是“模型会说但不一定会做”这个问题：它会幻觉，难以稳定访问实时/私有数据，也不擅长精确计算。过去常见办法是在提示词里反复强调“不要猜”，但这种软约束无法保证模型稳定触发工具，更不能保证参数结构合法。新方法把工具调用本身纳入训练轨迹，让模型学习“选工具—填参数—读结果—再作答”的完整闭环 (Schick et al., 2023; Patil et al., 2023; Qin et al., 2023)，可抽象为：

$$k \sim p_\theta(k \mid x), \quad a = f_\theta(x), \quad o = \mathrm{Tool}_k(a), \quad y \sim \pi_\theta(\cdot \mid x, o).$$

这串式子可以按流程读：先按概率选工具 k，再生成参数 a，调用工具得到观测 o，最后基于 (x, o) 生成答案 y。其中“$\sim$”表示“按该分布采样”。比如问“今天美元兑人民币是多少”，模型先选汇率 API，再填参数（币种、日期），拿到返回值后再组织语言回答。
RAG 是其中一种特例，即把“工具”具体化为检索器 (Lewis et al., 2020)。这种做法能显著提升 groundedness 和事实性，但代价是系统复杂度明显上升：执行环境、错误恢复、权限边界和工具预算都需要工程化治理。

工具会用了之后，下一步要解决的是“调用结果是否稳定可执行”，也就是解码阶段的结构可靠性问题。

4.2 约束解码（Constrained Decoding）

约束解码针对的是“几乎正确但不可执行”的结构错误：即使工具使用训练做得不错，模型仍可能输出解析失败的 JSON 或不合 schema 的参数。旧做法通常是“先生成、再校验、再修复”，虽然可行，但失败路径长、重试成本高、线上稳定性依赖补丁逻辑。约束解码把校验前移到生成过程，在 token 级别限制输出空间，只允许合法延展：

$$y \in \mathcal{V},$$

这里 y 是模型输出字符串，$\mathcal{V}$ 是“所有合法输出”的集合（比如满足某个 JSON schema 的所有字符串）。$y \in \mathcal{V}$ 的意思是：输出必须属于合法集合，不能越界。直觉上它像一条“轨道”，模型每走一步都不能脱轨。OpenAI Structured Outputs 的 `strict: true` 就体现了这个思想 (OpenAI, 2024)。
它的优势是结构可靠性显著提升，代价是可能增加延迟，而且“格式正确”依然不等于“语义正确”。

当结构可靠性建立后，后训练的重点会转向“如何在可验证任务上系统性提升正确率”。

这一阶段最关键的问题不是“会不会调用工具”，而是“是否在正确时机调用并稳定调用成功”。因此评估重点应放在 schema 有效率、工具成功率和 groundedness，而不是只看工具调用率；后者是最常见误区，因为高调用率并不代表高质量调用。

---

5 阶段 5：推理与智能体 RL（可验证正确性）

这一步的核心是：把“看起来像对”升级成“可验证地对”。

用财务运营助手举例最直观。设任务是“月末关账异常排查”：系统要读取总账、应收、应付和银行流水，定位差异来源，并给出可执行修复步骤。一个“会说但不一定真对”的模型，可能写出很像审计报告的文字，却把借贷方向、科目映射或对账口径搞错；这类错误在表面上不明显，但会直接影响财务决策。

阶段 5 的目标就是把这类任务变成“可机器核验”的训练问题。比如你可以把 verifier 设成：分录平衡是否成立、差异金额是否与系统记录一致、修复步骤是否满足公司关账 SOP。后面三节正好对应这个例子：先决定只看最终是否对账通过 (ORM) 还是连中间步骤都打分 (PRM)；再用自训练闭环扩大“已验证正确”的轨迹数据；最后用 GRPO 类方法在可验证奖励上做更稳定的大规模优化。

5.1 结果监督 vs 过程监督（ORM vs PRM）

ORM vs PRM 讨论的是“监督信号放在哪里”这一核心问题：只看最终答案会让奖励过于稀疏，只模仿推理文本又容易学会“像在推理”而不是真的做对。过去常见路线是结果奖励（实现简单）或纯 SFT（扩展容易），但两者都有明显上限。过程监督的关键改动是把反馈下沉到中间步骤 (Lightman et al., 2023)，对比写法是：

$$r_{\text{out}} = \mathbf{1}[\text{final answer passes}], \qquad R_{\text{proc}} = \sum_{t=1}^{T} r_t.$$

第一个式子里的 $\mathbf{1}[\cdot]$ 是“指示函数”：条件成立取 1，不成立取 0，所以它只看最终对错；第二个式子把每步奖励 r_t 累加，表示“过程对一点就加一点分”。可以类比数学解题：只看最终答案是“对/错”二元评分，过程奖励是“步骤分”。
结果监督便宜稳健但学习慢，过程监督信号密但标注成本高且噪声风险大，两者都高度依赖 verifier 精度。

财务例子（继续沿用“月末对账差异 12 万元排查”）：ORM 的打分规则可以是“最终是否给出正确差异来源 + 调整分录是否借贷平衡 + 是否通过 SOP 校验”，全部满足记 1，否则记

0。PRM 会把同一任务拆成步骤打分，例如“是否先核对银行流水”“是否正确计算应收账龄差”“是否定位到未匹配回款”“是否生成合法调整分录”，每步给局部奖励。这样即使最终答案没全对，模型也能从中间正确步骤中学习，通常比纯结果监督收敛更快。

当监督形式确定后，下一步自然是解决“高质量推理数据从哪里来”的规模化问题。

5.2 自训练闭环（STaR / ReST）

自训练闭环要解决的是推理数据供给瓶颈：高质量轨迹昂贵，靠人工全量标注无法持续扩容。旧路线主要依赖人工扩库，成本高且增长慢。STaR/ReST 的核心改动是“生成候选—自动验证—回灌训练”的循环（Zelikman et al., 2022; Zhang et al., 2024），其 best-of-K 的基本概率收益可写为：

$$\Pr(\text{at least one correct}) = 1 - (1-p)^K.$$

这个公式来自最基础的概率补集：先算“K 次都失败”的概率 $(1-p)^K$，再用 1 减掉它，就得到“至少成功一次”的概率。举例：若单次成功率 $p = 0.2$，采样 $K = 5$ 次，则成功至少一次的概率是 $1 - 0.8^5 = 0.672$，比单次 0.2 高很多。
这种闭环的数据放大效率很高，但也容易进入“自我确认循环”：如果 verifier 偏弱或有偏，系统会稳定学偏。

财务例子（继续同一题）：系统针对“12 万元差异排查”一次采样 8 条候选推理轨迹。每条轨迹都要通过自动校验：差异来源是否与系统账实一致、调整分录是否借贷平衡、模拟过账后差异是否归零、步骤是否符合关账 SOP。若 8 条里只有 2 条通过，就把这 2 条作为“高质量轨迹”回灌训练；下一轮再采样时，模型更容易生成类似轨迹。这个循环的本质是“先多试，再只学被 verifier 证明正确的答案”。

当同一任务的数据闭环跑起来后，训练算法层面就会转向“怎样在大规模下更稳、更省资源地优化策略”。

5.3 GRPO 及其变体

GRPO 及其变体解决的是“PPO 在长推理场景里成本高、稳定性压力大”的问题。过去主流是继续沿用 PPO + critic，路径成熟但资源开销和调参负担都偏重。GRPO 的关键改动是使用同题多样本的组内相对比较构造优势，弱化对显式 critic 的依赖（Shao et al., 2024）：

$$\bar{r} = \frac{1}{K}\sum_{i=1}^{K} r_i, \qquad A_i = r_i - \bar{r}.$$

这里先算组内平均奖励 $\bar{r}$，再把每个样本的优势写成 $A_i = r_i - \bar{r}$，意思是“比组平均好多少/差多少”。例如同题 4 个候选奖励是 $[1, 0, 0, 0]$，则 $\bar{r} = 0.25$，第一个样本优势 $A_1 = 0.75$，其余是 -0.25，更新时自然会推高第一个、压低其余。这个设计把“绝对打分”转成“同题相对比较”，方差通常更小。
后续变体（Dr. GRPO、GSPO、DAPO、LUFFY）本质上都在回答同一个问题：怎样让“同题多采样 + 相对比较”这条路线更稳、更省、更不容易学偏（Liu et al., 2025; Zheng et al., 2025; Yu et al., 2025; Yan et al., 2025）。先把四个关键词讲清楚会更好理解。
基线（`baseline`）：用来衡量“这个样本比平均好多少”。基线估计不稳，优势值就会抖，训练容易乱跳。

clipping：限制每次更新幅度，避免策略一步走太远。它像“安全护栏”，太松会发散，太紧会学不动。
更新粒度：到底按 token 级更新，还是按整条序列更新。粒度越细，信号越密但噪声也可能更大；粒度越粗，稳定性常更好但细节反馈更少。
采样策略：每个问题采几条、怎么采、是否引入更强模型轨迹。采样决定了探索强度，也直接决定训练成本。

对应到各变体，可以这样读：Dr. GRPO 更关注基线/归一化带来的优化偏差与效率问题；GSPO 更强调序列级比率与序列级 clipping，在长输出任务里通常更稳；DAPO 强调解耦 clipping 与动态采样调度，让“稳定性”和“探索强度”更可控；LUFFY 引入 off-policy guidance，让模型不只依赖当前策略采样，还能学习更强轨迹。它们并不是互斥关系，而是沿着这四个旋钮做不同侧重。

这一路线的优势是可扩展性和训练稳定性更好，代价是采样成本线性增加，且 verifier 一旦可被利用，模型会更快学偏。

财务例子（继续同一题）：模型对“12 万元差异排查”生成 4 条完整轨迹，verifier 奖励分别是 [1.0, 0.6, 0.2, 0.0]。可以把它们理解为：第一条差异定位正确且分录可过账；第二条差异定位正确但修复步骤不完整；第三条只给出模糊怀疑点；第四条方向错误。组均值是 $\bar{r} = 0.45$，对应优势是 [0.55, 0.15, -0.25, -0.45]。更新时第一条被明显强化，第二条小幅强化，后两条被抑制。你可以把 GRPO 理解成“同题内部排名学习”：不是问“绝对几分”，而是问“这组里谁更值得学”。

当可验证正确性提升到位后，最后一个工程问题往往是“如何把能力迁移到更便宜的部署形态”。

这一阶段的重点不是“回答写得像不像专家”，而是“答案能不能被机器检查为真”。例如在财务任务里，措辞再专业，只要分录不平衡或金额对不上，就应判定为错。工程上最重要的三件事是：第一，verifier 本身要准（别把错判成对）；第二，要做反作弊评测（防止模型学会钻评分规则空子）；第三，要监控长度漂移（防止模型靠无意义变长刷分）。最常见误区是先把 rollout 数量做大，以为“样本更多就会更好”，却没有先验证奖励和校验器质量；这样只会更快把错误信号放大，模型会稳定学偏。

---

6 阶段 6：蒸馏

蒸馏解决的是“效果可以，但部署太贵”的问题。

6.1 黑盒蒸馏 vs 白盒蒸馏

黑盒/白盒蒸馏解决的是同一个现实矛盾：教师模型效果很好，但部署太贵；学生模型便宜，但直接训练又很难复现教师能力。两者主要区别不在“目标不同”，而在“你能拿到教师模型多少内部信息”。

黑盒蒸馏指你看不到教师内部，只能拿到输入和教师输出文本（常见于调用闭源 API）。在这种设置下，你通常做的是“让学生学会复现教师回答风格与决策模式”，常见做法是采样高质量教师答案后用 SFT 或偏好优化训练学生。它的优点是工程门槛低、适配闭源教师；缺点是信号相对粗糙，学生只能学到“教师最后说了什么”，学不到“教师在每个 token 上如何权衡其他候选词”。

白盒蒸馏指你能访问教师 logits 或概率分布（有时还包括中间层表示）。这时可以直接对齐教师和学生在每一步的分布差异，训练信号更细，迁移通常更充分。典型目标是最小化教师分布与学生分布之间的 KL 距离 (Hinton et al., 2015)：

$$\mathcal{L}_{\mathrm{KD}} = \mathrm{KL}(p_T(\cdot \mid x) \| p_S(\cdot \mid x)) \,.$$

这里 p_T 是教师分布，p_S 是学生分布，KL 越小，表示学生在“每一步 token 判断”上越像教师。一个二分类小例子：若教师给 $[0.9, 0.1]$，学生给 $[0.6, 0.4]$，KL 较大；训练后学生变成 $[0.85, 0.15]$，KL 下降，表示迁移更成功。

什么时候用什么，可以按这条准则判断：如果教师是闭源 API、拿不到 logits、目标是快速降本上线，优先黑盒蒸馏；如果你能访问教师权重或 logits，且任务对保真度要求高（如安全边界、工具格式、专业术语细粒度行为），优先白盒蒸馏。很多团队会采用混合路径：先黑盒快速做第一版学生，再在可控教师上做白盒精修关键能力。

无论黑盒还是白盒，蒸馏都能显著降低延迟和成本，支持高 QPS 和端侧部署；共同风险是学生会继承教师偏差，而且教师版本变化会引发学生行为漂移，因此都需要版本锁定、数据来源追踪和学生回归评测 (Kim & Rush, 2016; Chen et al., 2024)。

理解蒸馏的收益和边界后，本章的六阶段后训练闭环就完整了。

这一阶段的关键问题是：在降本降延迟时，哪些教师能力必须被保住。落地时应把注意力放在蒸馏样本质量、教师版本管理和学生回归评测上；最常见误区是只看压缩比与速度收益，而忽略安全行为和格式一致性回归。

7 章末练习

做题时建议写成一个连贯的小分析，而不是机械列点：先用一到两句话界定当前问题和业务影响；然后简述现有方案及其瓶颈；再说明你选择的新方法如何在数据、目标函数、推理流程或系统约束上改变结果；最后给出它的优势、风险和可量化评测指标。评测指标不要空泛，尽量写成可观测项，例如 JSON 有效率、工具调用成功率、verifier 通过率、延迟与成本变化。

1. 设计题：设计“医疗书记员 (Medical Scribe)”，既认识罕见药名 (CPT)，又严格拒绝开处方（策略 + 评测关卡）。
   参考解析：思路是“两条线并行”。一条线让模型学会医学语言，另一条线把安全边界钉死。前者用领域持续预训练 (CPT) 覆盖罕见药名、缩写和病历写作；后者用策略 SFT/偏好对齐训练“只做记录与总结，不做处方建议”，并在推理侧加硬拒答模板。评测要同时看能力和安全：药名识别召回率、病历摘要准确率、违规处方建议率（目标接近 0）、高风险提示召回率。
2. 系统题：解释 GRPO 相比 PPO 为什么更省内存，以及这对训练 70B+ 模型为何关键。
   参考解析：GRPO 更省内存的核心原因，是它通常不依赖显式价值网络 (critic)，而是用同题组内相对奖励当基线。PPO 常要额外维护价值模型参数与优化器状态，在 70B+ 规模下这部分显存和带宽开销很大。对超大模型训练来说，省下这部分内存往往直接决定你能否放下更长上下文、更大 batch，或者是否还能稳定并行训练。
3. 权衡题：固定算力预算下，预算该投 DPO（训练侧）还是 test-time scaling（推理侧）？

参考解析：先看业务形态。如果是高请求量、长期在线服务，优先投 DPO，因为训练侧一次投入后，每次推理成本更低；如果是低请求量但单次准确性极关键（如高风险决策），优先投 test-time scaling，用多采样与重排换更高单次质量。可以把它理解成：DPO 是“前置投资、长期摊薄”，test-time scaling 是“按次付费、立刻见效”。

4. 排障题： SFT 模型答得对，但工具调用格式总错。该加数据，还是改约束解码？
参考解析：这类问题优先改约束解码，而不是先盲目加数据。原因是“格式错误”本质是结构约束问题，约束解码能直接保证 JSON/schema 合法；加数据只能提高“通常会对”的概率，不能给强保证。落地顺序一般是：先上 schema/grammar 约束保障可执行性，再补充失败样本做工具调用 SFT，提高鲁棒性。

5. 智能体题：多步工具链失败时，如何同时改数据、奖励/verifier 和测试时搜索？
参考解析：要同时改三处，不能只改 prompt。数据侧补“成功轨迹 + 失败恢复轨迹”(重试、改参、回退、终止)；奖励与 verifier 侧加步骤级检查（参数合法、工具返回可用、阶段目标达成、最终目标达成）；测试时搜索侧增加分支探索与重排，同时设置调用预算避免无限试错。评测建议看端到端成功率、步骤成功率、平均步数、无效调用率和超时率。

7.1 本章小结

这一章讲的是同一件事：怎样把“本来会生成文本的模型”，一步步变成“能在真实系统里稳定做事的助手”。后训练不是某一个算法名字，而是一条从行为塑形到上线降本的工程链路。

六个阶段可以按“先能用、再好用、再可靠、再便宜”来理解。阶段 1（SFT）解决“先把话说对格式、说对角色”；重点是模板一致性与监督边界。阶段 2（PEFT）解决“同一个底模如何低成本做很多专精版本”；重点是 adapter 设计、路由正确率和版本管理。阶段 3（偏好对齐）解决“答得对但不好用”的问题；重点是把用户偏好和安全边界一起学进去，并控制行为漂移。阶段 4（工具使用与 RAG）解决“只会说不会做”；重点是把调用工具、读结果、按证据回答这条链路做稳。阶段 5（推理与智能体 RL）解决“看起来像对但不一定真对”；重点是引入可验证奖励，用 verifier 驱动可核验正确性。阶段 6（蒸馏）解决“效果够好但部署太贵”；重点是把能力迁移到更小模型，同时守住安全与格式回归。

如果要抓住后训练最重要的意义，可以记一句话：后训练是在做“能力产品化”。它把模型的原始生成能力，转化成可控行为、可验证质量和可承受成本。没有后训练，模型可能“看起来聪明”；有了后训练，模型才更可能“在生产里可靠”。

落地时建议始终用同一套思维检查每个阶段：你改了哪一个训练杠杆、用什么指标证明它真的变好、出了问题能否快速回滚。真正稳定的方案，不是“单点最优”，而是“训练杠杆 + 评测闭环 + 回滚机制”三者同时成立。

7.2 参考文献

- instructgpt_2022 — Ouyang et al., “Training language models to follow instructions with human feedback,” 2022.
- self_instruct_2022 — Wang et al., “Self-Instruct: Aligning Language Models with Self-Generated Instructions,” 2022.

- hf_chat_templating — Hugging Face, "Chat Templating" documentation.
- trl_sft_trainer — Hugging Face TRL, "SFTTrainer" documentation.
- lora_2021 — Hu et al., "LoRA: Low-Rank Adaptation of Large Language Models," 2021.
- qlora_2023 — Dettmers et al., "QLoRA: Efficient Finetuning of Quantized Language Models," 2023.
- ppo_2017 — Schulman et al., "Proximal Policy Optimization Algorithms," 2017.
- reinforce_1992 — Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," 1992.
- dpo_2023 — Rafailov et al., "Direct Preference Optimization: Your Language Model is Secretly a Reward Model," 2023.
- orpo_2024 — Hong et al., "ORPO: Monolithic Preference Optimization without Reference Model," 2024.
- toolformer_2023 — Schick et al., "Toolformer: Language Models Can Teach Themselves to Use Tools," 2023.
- toollmm_2023 — Qin et al., "ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs," 2023.
- gorilla_2023 — Patil et al., "Gorilla: Large Language Model Connected with Massive APIs," 2023.
- openai_function_calling_docs — OpenAI, "Function calling" documentation.
- openai_structured_outputs_2024 — OpenAI, "Introducing Structured Outputs in the API," 2024.
- rag_2020 — Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," 2020.
- verify_step_by_step_2023 — Lightman et al., "Let's Verify Step by Step," 2023.
- math_shepherd_2023 — Wang et al., "Math-Shepherd: Verify and Reinforce LLMs Step-by-step without Human Annotations," 2023.
- deepseek_math_2024 — Shao et al., "DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models," 2024.
- drgrpo_2025 — Liu et al., "Understanding R1-Zero-Like Training: A Critical Perspective" (Dr. GRPO), 2025.
- gspo_2025 — Zheng et al., "Group Sequence Policy Optimization," 2025.
- dapo_2025 — Yu et al., "DAPO: An Open-Source LLM Reinforcement Learning System at Scale," 2025.
- luffy_2025 — Yan et al., "Learning to Reason under Off-Policy Guidance" (LUFFY), 2025.
- star_2022 — Zelikman et al., "STaR: Bootstrapping Reasoning With Reasoning," 2022.
- rest_mcts_2024 — Zhang et al., "ReST-MCTS*: LLM Self-Training via Process Reward Guided Tree Search," 2024.
- hinton_distill_2015 — Hinton et al., "Distilling the Knowledge in a Neural Network," 2015.

- seq_kd_2016 — Kim & Rush, "Sequence-Level Knowledge Distillation," 2016.
- blackbox_kd_2024 — Chen et al., "Knowledge Distillation of Black-Box Large Language Models," 2024.