课程学习讲义

AI 能自我修正吗？从 Decoding、Workflow 到 Reasoning

整理：Codex；来源：李宏毅 2026《机器学习》第 15 节

2026-04-29

视频作者/频道：啥都会一点的研究生
发布日期：课程合集发布日期：2026-03-13；本节页面创建时间：2026-04-28
本节时长：01:27:42
视频链接：https://b23.tv/3hOmntL
证据说明：B 站接口未提供官方字幕，本讲义基于本地 ASR 转写、视频关键帧和课程画面重构；术语和明显 ASR 错字已按课程上下文规范化。

目录

1 问题框架：什么才算 AI 自我修正?

本节的核心问题不是“语言模型在被人指出错误之后能不能改”，而是更严格的版本：模型输出答案之后，能不能在没有人类额外指出错误的情况下，自己发现问题并修正输出。课程把这个问题放在三个层次里讨论：

1. **Decoding** 层：不改模型参数，只在输出 token 时修改概率分布，让模型远离看起来像错误的状态。
2. **Workflow** 层：让模型先生成答案，再自动插入一个泛用的 verification/reflection 指令，让模型检查并修正。
3. **Reasoning/RL** 层：训练模型，让它在推理过程中自然地产生验证、反思和修正行为。

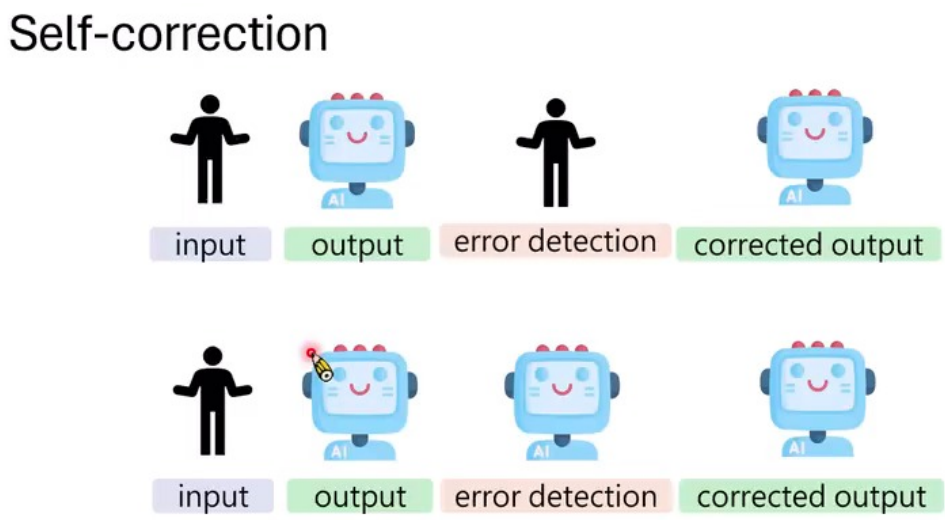

图 1: 自我修正的基本流程：模型先生成输出，再由错误侦测或反思环节推动修正。[1]

本节最重要的问题分解

自我修正至少包含两个不同能力：先判断“我的输出可能错了”，再知道“应该怎样改”。只有知识更多并不自动等于能自我修正；模型可能知道 Hillary Clinton 出生在 Chicago，却仍然在另一个问题里把她作为 New York 出生的政治家输出。

课程开头强调了一个关键差异：人在环路中的纠错和模型自发纠错不是一回事。如果人类告诉模型“你错了，错在这里”，模型能改是比较容易的；真正有研究价值的是，能不能用模型内部信号、自动 workflow 或训练机制，让模型自己触发这个过程。

[1] 视频画面时间区间：00:01:00–00:01:35。

1.1 早期直觉：错误状态能不能从内部表示看出来?

开头部分提到，早期工作尝试观察模型在答对与答错时的内部 representation 是否有差异：如果模型处于“答错状态”，某些中间层表征或 logits 分布可能和答对状态不同。这样就出现了一个很自然的思路：

- 如果我们能识别出“错误状态”的表示，就能做 error detection。
- 如果我们能把输出从错误状态推开，就能做 error correction。
- 这种方法不一定需要重新训练模型，只需要在推理时改变输出概率或中间表示。

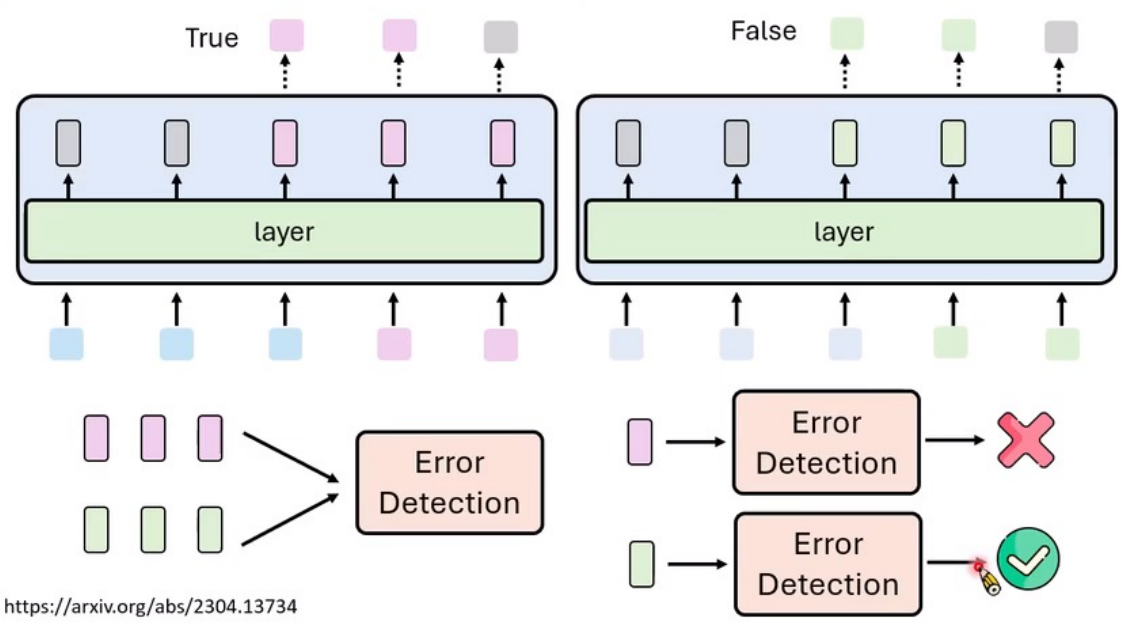

图 2: 从 True/False 状态的内部表示差异出发，尝试做错误侦测与错误修正。[2]

不要把“模型知道事实”误认为“模型会纠错”

课程反复提醒：模型可能拥有正确知识，但在一次生成中仍然走到错误输出。生成是一个连续采样过程，早期 token 一旦错了，后续 token 往往会被迫沿着错误方向解释下去。因此，自我修正需要额外机制打断这个惯性。

1.2 本章小结

本节课的主线是从“输出时干预”走向“流程上检查”，再走向“训练出内生推理”。三条路线都可以被称为自我修正，但它们的强度不同：decoding 是外部算法在帮模型改分布，workflow 是程序自动要求模型再检查，reasoning/RL 则试图让模型自己学会什么时候检查、什么时候停止。

[2]视频画面时间区间：00:04:40–00:05:30。

2 路线一：从 Contrastive Decoding 开始的输出分布修正

Contrastive Decoding 的核心想法是：既然模型在好状态和坏状态下会给出不同的 token 分布，就可以把“坏状态偏好的 token”从“正常状态偏好的 token”里减掉。它不是让模型重新学习知识，而是在解码时重新排序下一步 token 的分数。

直觉

如果一个 token 在正常输入下很可能出现，但在被破坏的输入、较弱模型、较浅层表示或缺失上下文下也很可能出现，那么它未必代表真正可靠的答案。Contrastive Decoding 希望保留“好来源强、坏来源弱”的 token。

可以把这种方法写成一个统一形式。对候选 token v，令正常路径给出的概率为 p_{good}，坏路径或弱路径给出的概率为 p_{bad}，则修正后的分数可以写成：

$$s_t(v) = \log p_{\text{good}}(v \mid x, y_{<t}) - \alpha \log p_{\text{bad}}(v \mid \tilde{x}, y_{<t})$$

- $s_t(v)$：第 t 步候选 token v 的修正后分数。
- x：正常输入或完整上下文。
- $\tilde{x}$：被破坏、弱化或去掉关键信息后的输入。
- $y_{<t}$：第 t 步之前已经生成的 token。
- α：减去坏路径分布的强度；太小修正不明显，太大可能把有用信号也减掉。

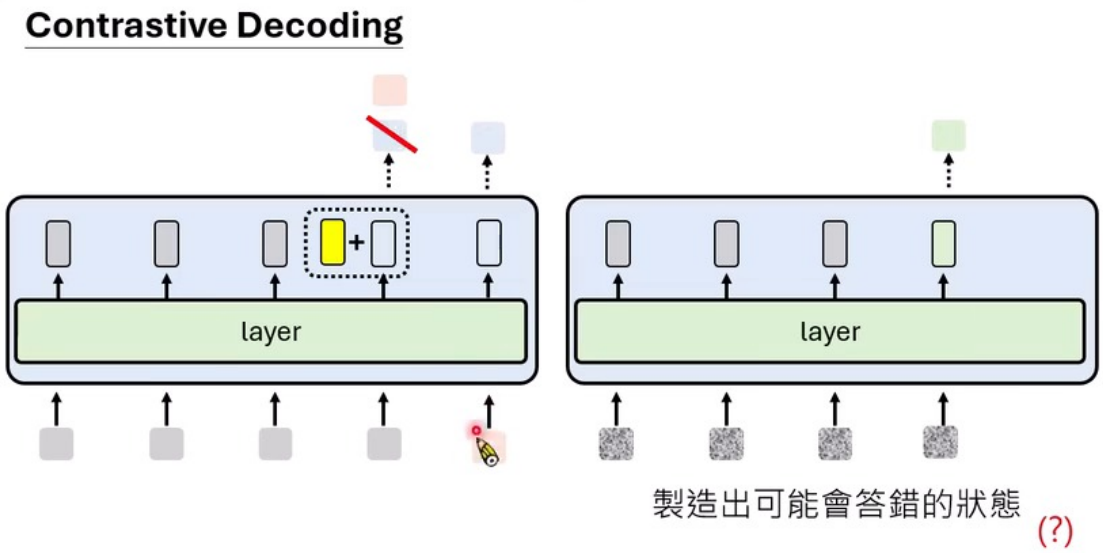

图 3: Contrastive Decoding 在下一 token 分布上做“正常分布减错误分布”的调整。[3]

[3]视频画面时间区间：00:10:00–00:10:40。

2.1 Expert vs Amateur：从大模型和小模型的差异开始

课程提到，最早的 contrastive decoding 直觉可以理解为“expert model 减 amateur model”。大模型和小模型都可能会给某些常见词高概率，但小模型更容易走向低质量或幻觉答案。把小模型分布作为负项减掉，就能凸显大模型特有的、更可靠的偏好。

统一范式

Contrastive Decoding 的本质不是某一种具体 trick，而是一种对比范式：构造一个“希望保留的分布”和一个“希望远离的分布”，再在解码时做差。

2.2 DoLa：用不同层的分布做对比

DoLa（Decoding by Contrasting Layers）的思路是，不一定要用两个模型做对比，也可以在同一个模型的不同层之间做对比。浅层或中间层可能还没有完整整合事实和语义，最终层则包含更成熟的判断。用最终层分布减去较浅层分布，就相当于把“不成熟的早期猜测”压低。

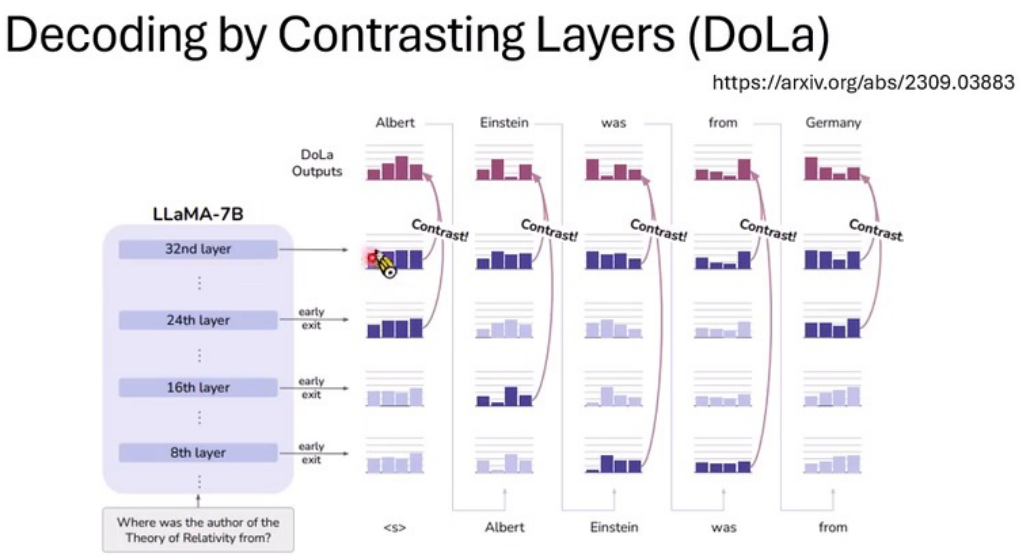

图 4: DoLa 用最终层和较浅层的输出分布差异来调整 token 选择。[4]

这类方法的价值在于它不依赖另一个外部模型，也不需要重新训练；但它也引出一个问题：到底哪一层是“坏来源”，哪一层是“好来源”，并没有对所有任务都稳定成立。

2.3 多模态与上下文版本：LayerCD、ICD、CAD、VCD、Audio-Aware Decoding

课程随后把这个范式扩展到更多场景：

- **Layer Contrastive Decoding**：在视觉语言模型里，对比 vision encoder 不同层的表示，抑制由不充分视觉理解带来的错误 token。

[4]视频画面时间区间：00:18:00–00:19:10。

- **Instruction Contrastive Decoding**：对比不同 instruction 条件下的输出，让模型避开不符合目标指令的模式。
- **Context-Aware Decoding**：在 RAG 场景里，对比“有检索文档”和“没有检索文档”的分布，强调真正由上下文支持的 token。
- **Visual Contrastive Decoding**：在图像问答中，对比干净图像和被遮挡、加噪或弱化后的图像，减少视觉幻觉。
- **Audio-Aware Decoding**：在语音或音频理解里，对比有音频和静音/噪声条件下的分布。

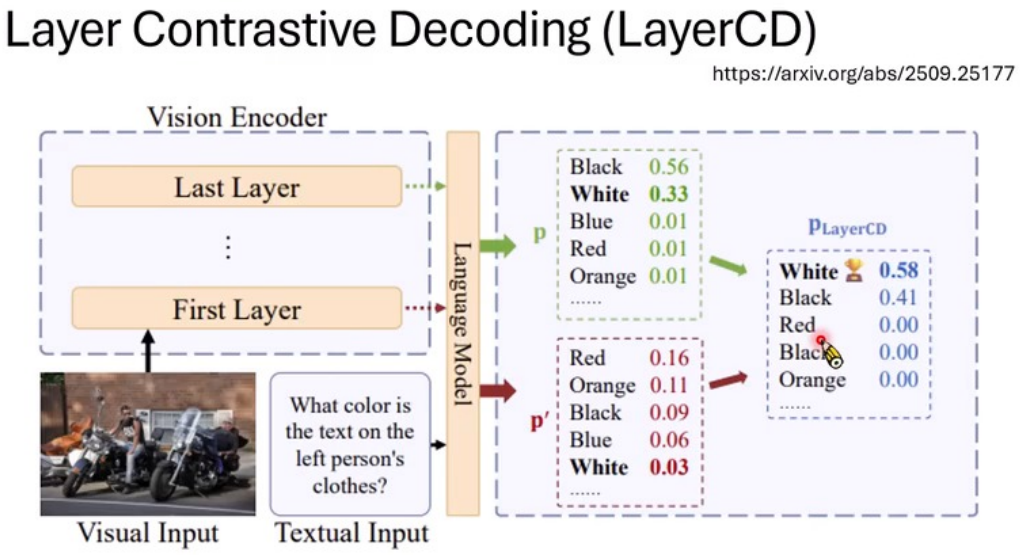

图 5: LayerCD 在视觉语言模型中对比不同视觉层表示，试图降低视觉幻觉。[5]

[5]视频画面时间区间：00:20:00–00:20:50。

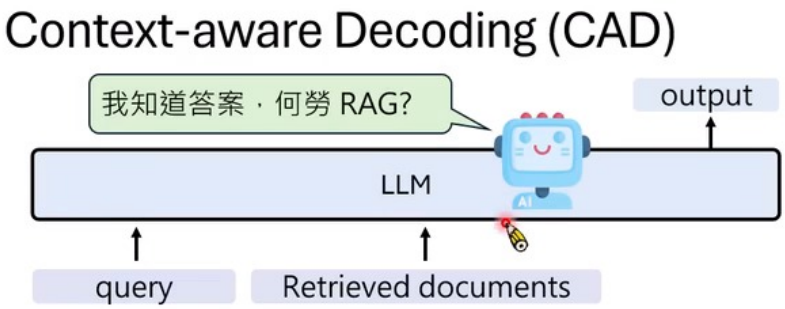

图 6: Context-Aware Decoding 通过“带上下文”与“不带上下文”的分布差异，突出由检索材料支持的回答。[7]

“坏来源”必须真的坏得有意义

如果负项只是随便加噪、随便插入无关词，它不一定能提供有用的反方向信号。课程在 MTI 部分特别提到，像 `Output Error` 或 `Status Error` 这样的负向提示有用，但随便塞无意义词效果很差。

2.4 MTI：只在关键 token 做干预

传统 Contrastive Decoding 需要在每一个生成步骤都跑一次好路径和坏路径，成本很高。Minimal Test-Time Intervention（MTI）提出：能不能只在关键位置启动对比？课程给了一个很直观的标准：当下一 token 分布 entropy 很高、模型很不确定时，这个位置可能就是关键岔路。

[7] 视频画面时间区间：00:23:00–00:23:40。

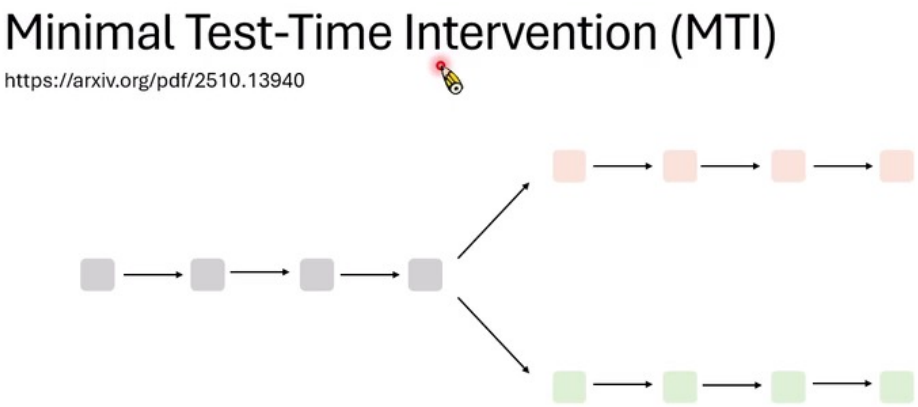

图 7: MTI 的直觉：只有在高不确定性的关键分叉处启动干预。[8]

但课程马上指出一个陷阱：如果为了得到坏路径分布，仍然要从头跑完整个坏输入序列，那么只在少数 token 干预并不一定省算力。MTI 的关键工程点是利用 KV cache。它不在输入前面大幅破坏内容，而是在当前生成位置后面追加很短的负向提示，例如 `Output Error`，这样前缀的 KV cache 可以复用，额外计算只发生在少数新增 token 上。

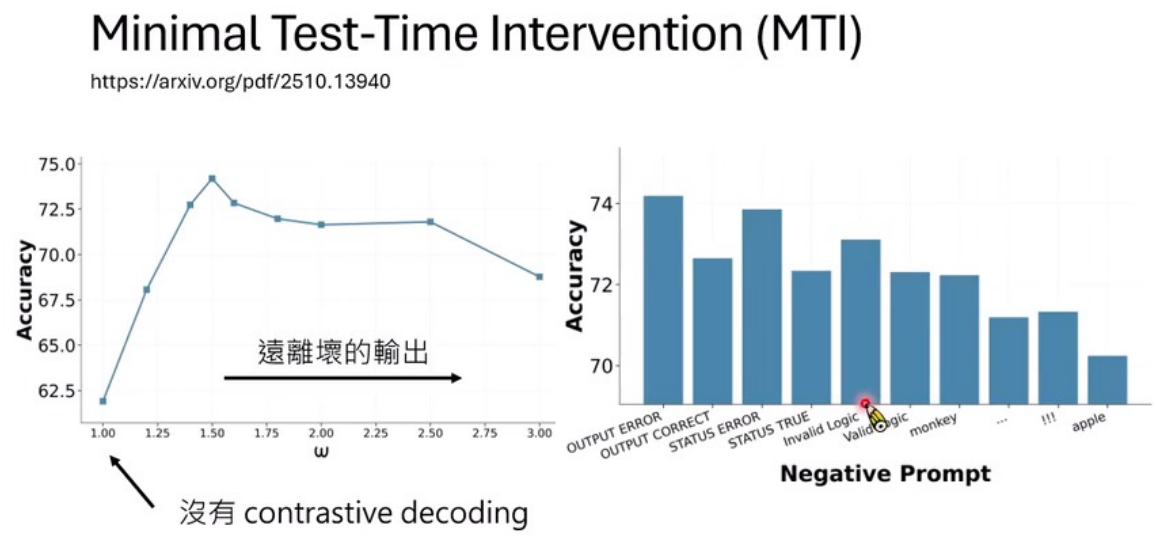

图 8: MTI 通过负向提示和参数强度控制，在性能与过度修正之间做权衡。[9]

[8]视频画面时间区间：00:30:00–00:31:00。

[9]视频画面时间区间：00:36:00–00:37:20。

MTI 的真正贡献

MTI 不只是“少干预几次”。它把“何时干预”和“如何不破坏 KV cache”结合起来，避免为了构造坏分布而重新跑完整上下文。

2.5 Decoding 家族的共同局限

课程在第一部分末尾用表格总结了不同方法：有的改 logits，有的改 representation，有的改 attention；有的负来源来自弱模型，有的来自浅层，有的来自缺失上下文或加噪输入。

Summary

	怎麼拿到錯誤結果	改哪裡
Contrastive Decoding (CD)	小模型	output
Decoding by Contrasting Layers (DoLa)	淺層用 logit lens 生成的結果	output
Layer Contrastive Decoding (LayerCD)	用淺層的 Image encoder layer (影像)	output
Instruction Contrastive Decoding (ICD)	降智咒語	output
Context-aware Decoding (CAD)	拿掉 Context (e.g., RAG 中的 Retrieved documents)	output
Visual Contrastive Decoding (VCD)	影像加雜訊、打亂 Patch、蓋住重要部分	output
Audio-aware Decoding (AAD)	移除聲音	output
Minimal Test-Time Intervention (MTI)	降智咒語，目標: 減少算力消耗	output
Visual Information Steering with Token-logit Augmentation (VISTA)	移除影像	hidden representation
Attention-space Contrastive Guidance (ACG)	計算 Attention 時不考慮影像	attention

图 9: 课程总结的 contrastive decoding 家族：不同方法的差别在于对比来源和干预位置。[10]

这一类方法的优点是无需训练，试错成本低；缺点是它仍然是外部算法在替模型修正，而不是模型真正理解了何时该反思。对任务有效与否，很依赖坏来源构造是否合理、超参数是否合适、以及额外计算是否可接受。

2.6 本章小结

Contrastive Decoding 家族把自我修正问题转化为“从坏分布里减掉错误倾向”。它适合在不训练模型的情况下快速尝试，也适合解释很多多模态幻觉抑制方法。但它没有解决最深层的问题：模型是否真的知道自己错了？它更多是在利用分布差异做 test-time correction。

3 路线二：Workflow 中的 Generation → Verification

第二条路线不直接改 logits，而是在流程上加一个自动检查步骤。模型先生成答案，然后系统自动插入一个泛用的 reflection instruction，例如“再检查一下”“你确定吗”“请修正你的答案”。

[10]视频画面时间区间：00:39:40–00:40:05。

因为这个指令可以由程序自动插入，不需要人类针对具体错误写反馈，所以课程仍然把它归入自动自我修正的范畴。

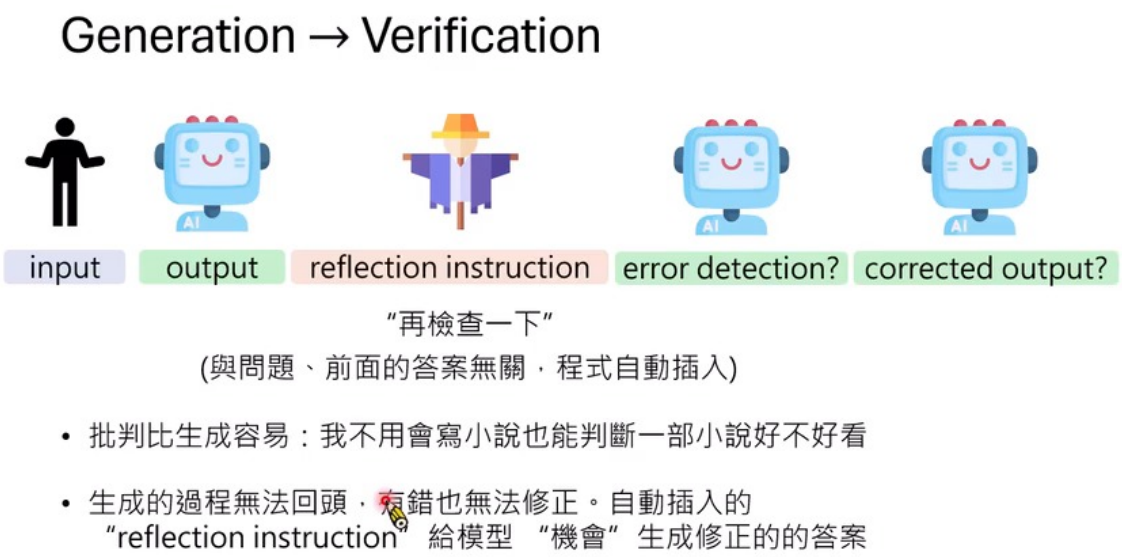

图 10: Generation → Verification：先生成答案，再自动插入反思指令，期待模型侦测并修正错误。[12]

3.1 为什么 verification 可能有用?

课程给了两个直觉解释：

- 批判可能比生成容易：不会写小说的人仍可能判断小说好不好。类似地，模型第一次生成时没发现错误，但回头检查时可能更容易发现。
- 生成过程无法回头：一旦前面的 token 错了，后续生成常常会沿着错答案硬编下去。插入“再检查一下”相当于给模型一个语境切换点，让它有机会从原来的错误轨道跳出来。

但课程也提醒，这些只是人类直觉，不一定适用于语言模型。有些研究发现，模型做“选择哪个答案更好”的能力并不一定比生成能力强；反过来，这类实验本身也可能受选择题格式、模型指令遵循能力等因素影响。

3.2 实证结果：内部反思不稳定，外部反馈更可靠

大规模实验显示，纯 internal self-reflection 有时有用，但并不稳定。有些模型或任务在反思后变好，有些反而变差。External feedback，例如执行代码得到错误信息、联网检索、checklist、人工或外部系统反馈，通常更稳定。

[12] 视频画面时间区间：00:42:40–00:43:30。

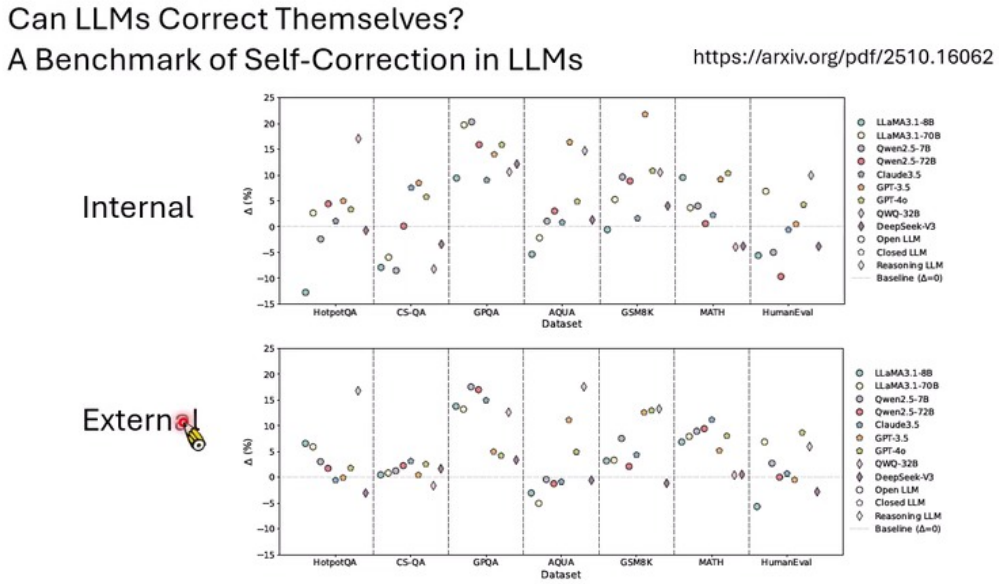

图 11: RefineBench 一类实验显示：纯自我反思的提升较小，外部 checklist 或完整反馈带来的提升更明显。[14]

不要只报告“加 verification 有提升”

如果 verification 需要额外生成很多 token，就必须和同等算力下的 baseline 比较。课程特别强调，很多 workflow 论文如果没有和 majority vote 或多采样比较，结论很容易被质疑。

3.3 诊断自我修正：四种结果、CL 与 CS

课程介绍了一套分析模型修正行为的框架。一次反思前后有四种结果：

- 错 $\rightarrow$ 对：最理想，说明模型接受批判并修正。
- 对 $\rightarrow$ 对：不坏，但消耗算力。
- 错 $\rightarrow$ 错：没有改善，也消耗算力。
- 对 $\rightarrow$ 错：最糟，模型想太多把正确答案改坏。

论文中定义了两个指标：

$$\mathrm{ACC}_2 = \mathrm{ACC}_1 \cdot \mathrm{CL} + (1 - \mathrm{ACC}_1) \cdot \mathrm{CS}$$

- ACC_1：反思前准确率。
- ACC_2：反思后准确率。
- CL：Confidence Level，即本来答对时，反思后仍保持正确的概率。

[14] 视频画面时间区间：00:47:40–00:49:40。

- CS：Critic Score，即本来答错时，反思后改对的概率。

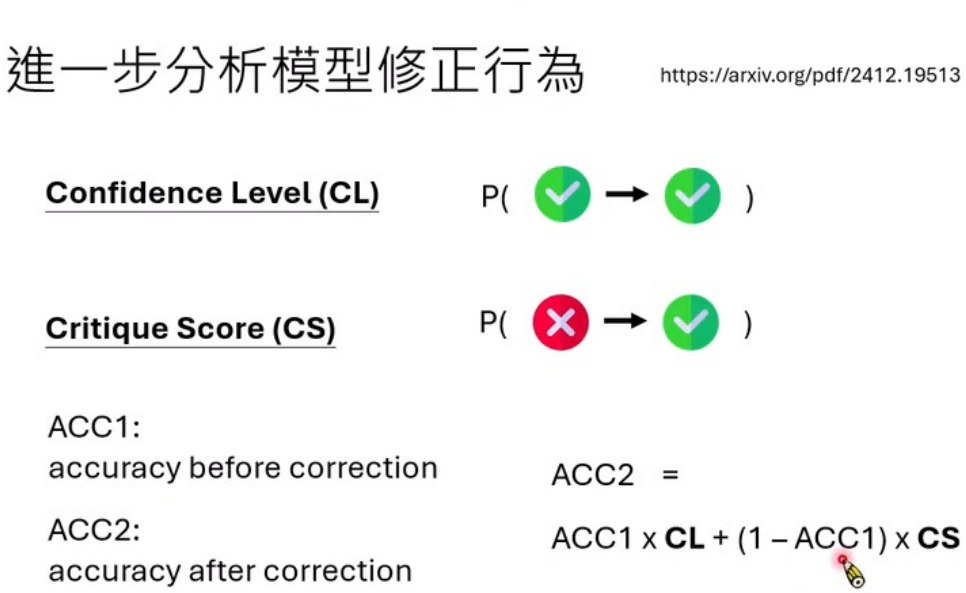

图 12: 用 Confidence Level 和 Critic Score 分解自我修正行为。[15]

CL-CS tradeoff

模型“坚持正确答案”的能力和“接受批评改错”的能力可能存在张力。过度自信的模型不容易改错；过度容易被质疑的模型又可能把正确答案改坏。

3.4 Reflection instruction 本身很关键

同一个模型，在不同反思指令下可能表现出不同性格。中性指令只是让它再做一次；肯定性指令会提高 CL、让模型更固执；批判性指令会提高 CS、让模型更愿意修改。不同论文使用的反思提示不同，可能是文献结论混杂的重要原因。

3.5 算力视角：verification 像奢侈品

课程随后引入一个更尖锐的问题：同样的额外算力，用来让模型反思旧答案，还是用来多 sample 几个新答案再 majority vote，哪个更划算？

实验证据显示，如果横轴只看 sample 数，加 reflection 看起来有提升；但如果横轴换成实际 inference compute budget，在预算有限时，多采样加投票往往更划算。只有当多采样已经接近饱和，继续增加样本不再有效时，额外 verification 才可能发挥作用。

[15]视频画面时间区间：00:51:40–00:52:20。

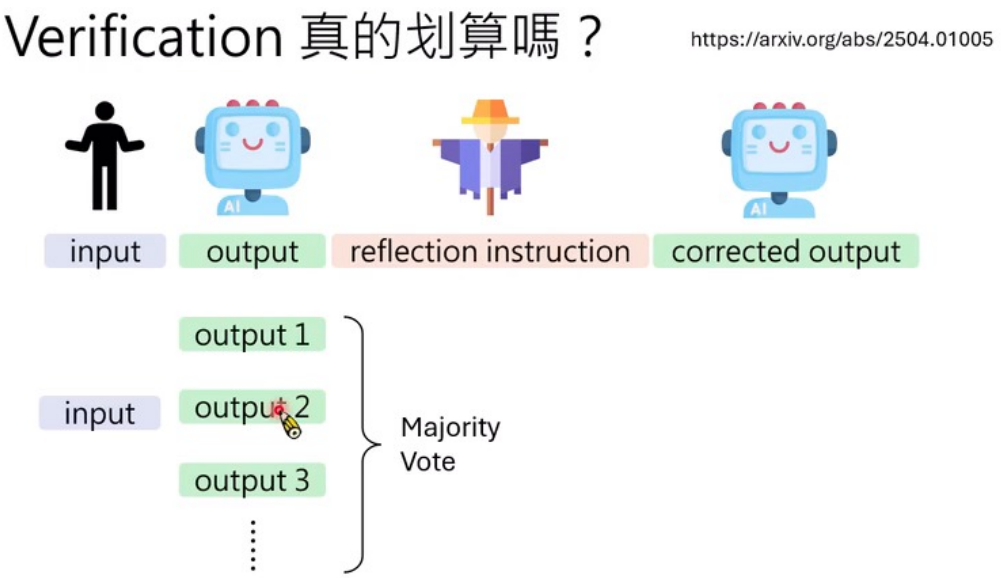

图 13: 把横轴换成算力预算后，verification 的优势显著缩小；在预算有限时 majority vote 常常是更强 baseline。[17]

实验设计底线

如果提出新的 reflection/workflow 方法，至少要比较同等算力下的 majority vote。否则提升可能只是因为多花了 token，而不是因为 verification 本身更有效。

3.6 本章小结

Workflow 让模型看起来会自我反思，但它不是稳定魔法。内部反思有时有效，外部反馈更可靠；反思提示会改变模型行为；最重要的是，verification 必须和同等算力的多采样 baseline 比较。只有在多采样收益饱和后，verification 才更像值得追加的高级策略。

4 路线三：从 Workflow 走向 Reasoning 与训练

Workflow 的缺点是每次都要硬插一个反思指令。即使答案本来对，模型也要多生成一段检查内容，浪费 token。Reasoning 路线希望模型学会：该修时修，不该修时停。

[17] 视频画面时间区间：00:57:40–01:00:55。

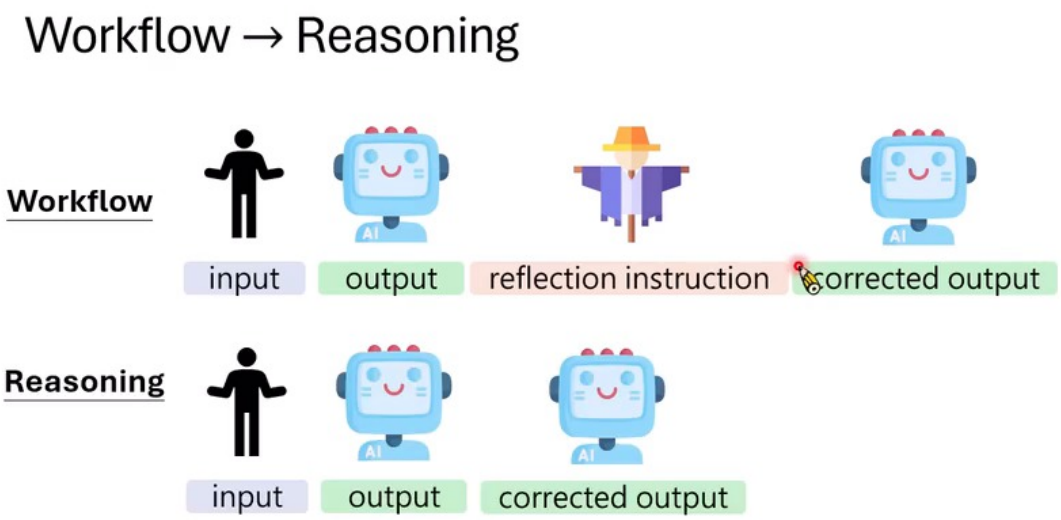

图 14: Workflow 与 Reasoning 的差别：前者显式插入反思指令，后者希望模型在推理过程中自然完成检查与修正。[19]

4.1 知识不等于自我修正

课程用 Hillary Clinton 的例子说明：模型可能知道某个事实，却不能在另一个问题里利用这个事实纠正自己。自我修正不是知识量的直接函数，而可能是一种独立状态或能力。某些工作甚至把“自我修正状态”抽成 steering vector，加到模型中会让模型在不需要修正时也倾向于修正。

能力拆分

一个模型可能知道正确答案、能识别别人答案错误、能在被明确指出后修改，但仍然不能在自己的生成轨迹里自动触发修正。这几个能力不能混为一谈。

4.2 直接训练自我修正：先侦测，再修正

课程介绍的 refine 类方法把自我修正拆成两步训练：

1. 错误侦测：模型看到自己的输出，如果发现错误，就输出一个 `refine` token；如果答案已经正确，就输出结束符号。
2. 错误修正：当输入、原错误输出和 `refine` token 都出现时，模型学习输出正确答案。

这种分解比把“侦测 + 修正”一次性学起来更容易。但课程也指出一个关键问题：fine-tune 之后模型参数变了，它在 inference 时犯的错误也可能变了。训练时见过的“绿色错误”能修，参数改变后出现的“红色错误”未必能修。

[19] 视频画面时间区间：01:02:00–01:02:50。

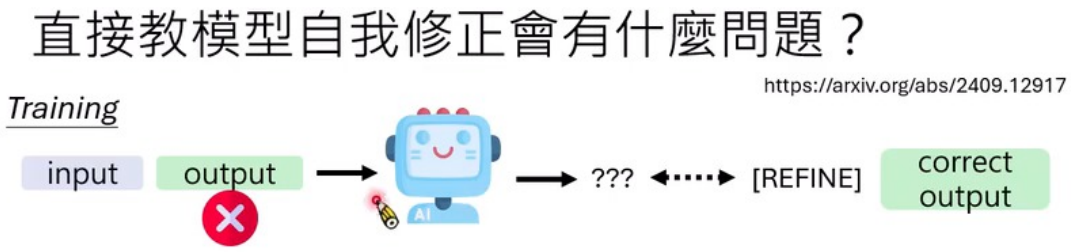

图 15: 直接训练自我修正的困难：训练后模型分布改变，inference 时出现的错误可能不同于训练时的错误。[21]

4.3 RLVR：只奖励最终答案，让修正行为自然出现

因此更常见的路线是 Reinforcement Learning with Verifiable Reward（RLVR）。模型面对输入后可以生成一长串 reasoning tokens；过程不直接监督，只看最终答案是否正确。数学和编程任务特别适合这类训练，因为最终答案或程序运行结果可验证。

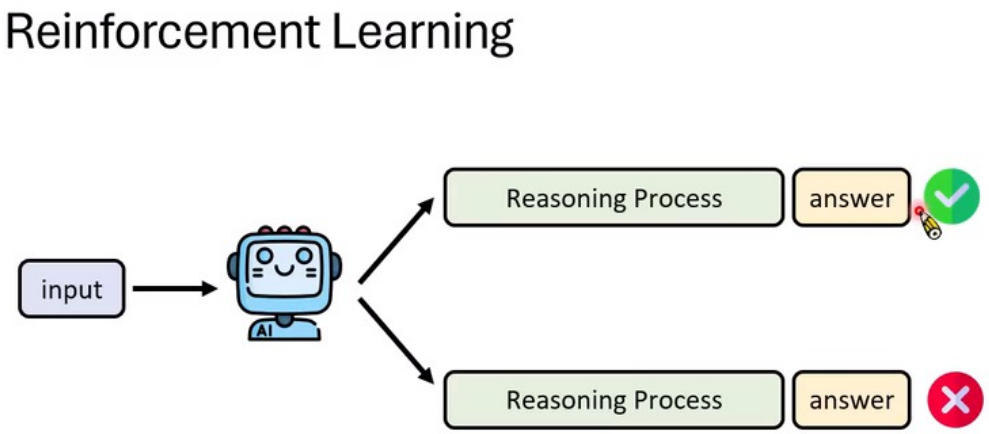

图 16: RLVR：只要最终答案可验证，就可以用对错 reward 训练模型产生更好的 reasoning 轨迹。[23]

[21] 视频画面时间区间：01:06:40–01:07:50。

有趣的是，虽然训练只奖励最终答案，但模型在 reasoning 中会自然出现“先提出一个答案、再检查、发现不对、再修正”的行为。课程把这看成 self-correction 可能从 reasoning 过程中涌现的证据。

为什么模型不一开始就答对?

课程用人的学习经验作类比：有些错误必须自己踩过才知道怎么修。对模型来说，一些复杂问题也可能很难一步到位；先走一条候选路径，再发现并修正，是一种有效的计算过程。

4.4 Reasoning 的样本效率：Parity Check 例子

课程给出一个 sample-complexity 直觉：如果要求模型一步从输入直接映射到答案，输入和输出的组合可能指数增长；如果把问题拆成多个步骤，每个步骤只需要学一个小操作，总数据需求可能大幅下降。

对 parity check，长度为 6 的二进制串有 $2^6 = 64$ 种输入。如果模型只会死记硬背，必须看过 64 种情况。但如果把任务拆成逐步 XOR，每步只需要学会四种局部规则：

$$0 \oplus 0 = 0, \quad 0 \oplus 1 = 1, \quad 1 \oplus 0 = 1, \quad 1 \oplus 1 = 0$$

长度 6 的序列只需 5 步，每步 4 种局部变化，所以大约 $4 \times 5 = 20$ 个局部样本就能教会这种过程。

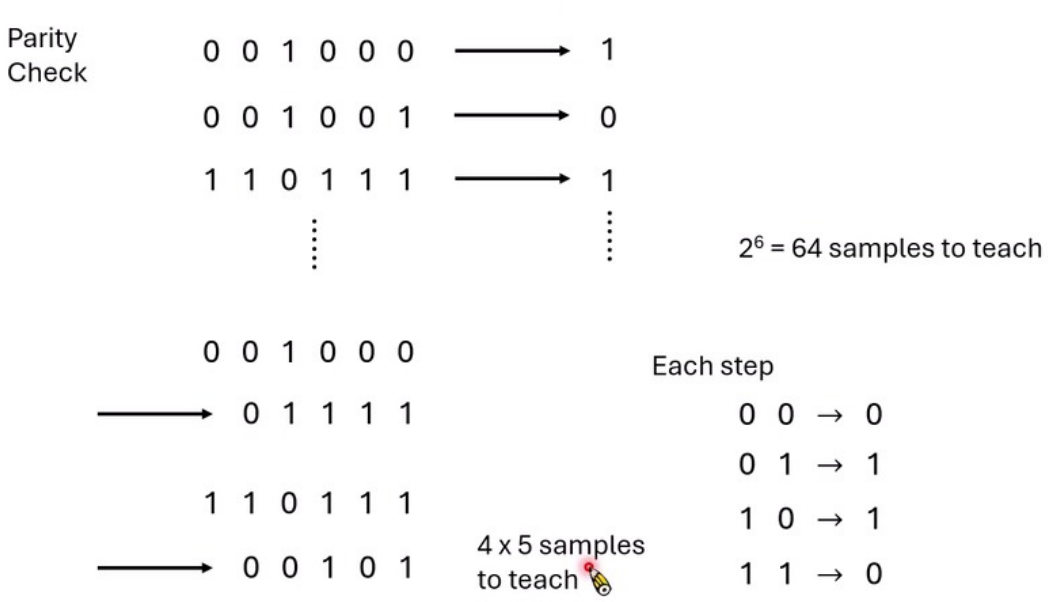

图 17: Parity Check 展示了“直接映射”与“分步推理”的样本复杂度差异。[24]

Reasoning 的价值

Reasoning 不只是“多输出一些 token”。它把一个高组合复杂度的输入输出映射，拆成多个可复用的局部步骤，因此可能提升泛化能力。

[23] 视频画面时间区间：01:09:00–01:10:00。

[24] 视频画面时间区间：01:14:50–01:17:35。

4.5 本章小结

Reasoning 路线比 workflow 更强，因为它试图把“何时检查、何时修正”内化到模型生成过程中。直接训练 refine token 会遇到分布漂移问题，RLVR 则用最终可验证 reward 训练完整生成过程，让 verification/reflection 行为在 reasoning 中自然出现。

5 最终争论：RL 创造了新能力，还是只提高了旧路径概率?

课程最后把问题推向当前研究争论：RL 让模型学到了新的 reasoning 能力吗? 还是 base model 本来就能走出正确路径，只是概率太低，RL 只是把这些路径的概率提高?

5.1 观点一：Base model 本来就可能 sample 到正确路径

一种证据来自 pass@k：如果同一道题让模型 sample 很多次，只要其中一次答对就算通过，那么没有做 RL 的 base model 在 k 很大时也可能接近 RL 模型。这支持一种解释：正确 reasoning path 早就存在，只是概率低；RL 主要是在重排路径概率。

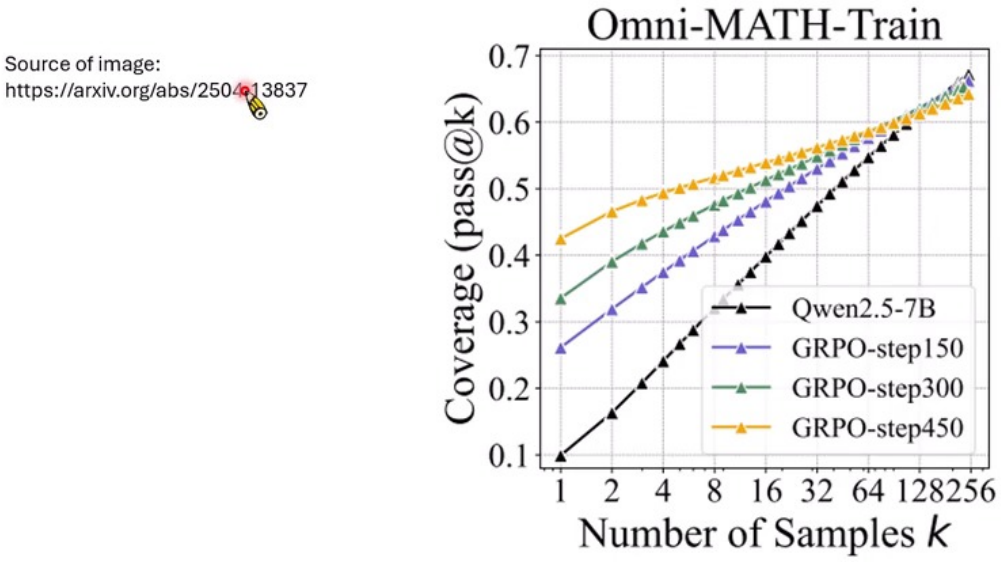

图 18: Pass@k 结果支持一种解释：增加采样次数后，base model 也可能覆盖到正确路径。[25]

这一观点自然引出 training-free sampling：如果正确路径本来就在分布里，那么也许不训练，只改 sampling 策略，就能逼出模型已有的 reasoning 能力。课程提到有工作确实用更复杂的 sampling 方法逼近甚至部分超过 RL 结果。

5.2 观点二：RL 可能确实教会了新的能力

另一派认为，只看最终答案是否正确不够。sample 很多次可能只是猜中答案，尤其当答案空间有限时。于是有研究提出不仅检查最终答案，还检查 chain-of-thought 计算过程，即类似 CoT-pass 的标准。如果计算过程也必须正确，那么没有 RL 的模型就明显落后于有 RL 的模型。

[25] 视频画面时间区间：01:20:00–01:21:30。

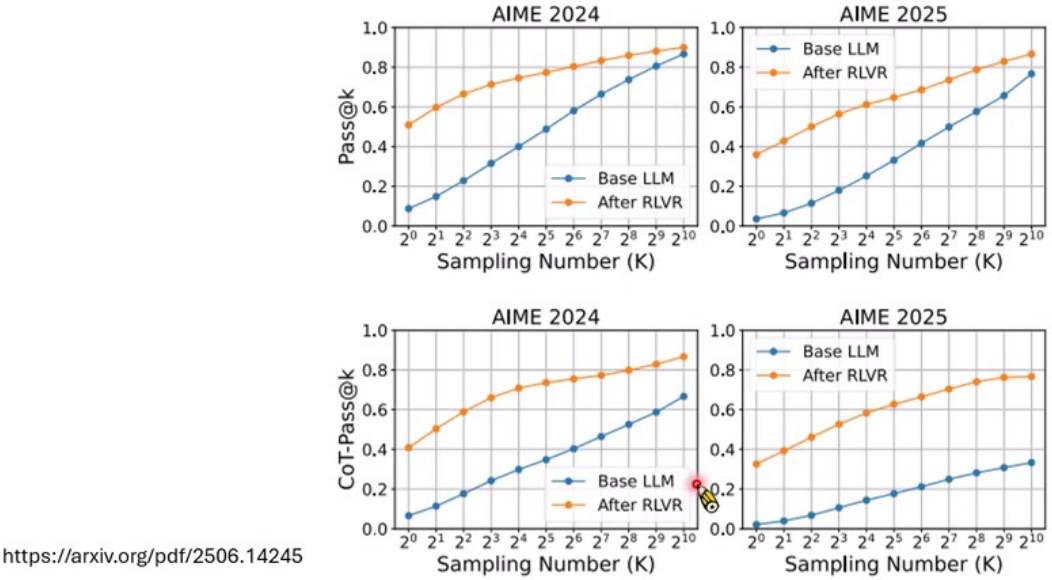

图 19: 课程结尾讨论：仅看 pass@k 可能高估 base model；检查推理过程后，RL 可能仍然带来新能力。[27]

过程验证本身也有争议

如果 CoT 过程由另一个语言模型判断，那么这个 judge 是否可靠仍然是问题。课程对这类证据保持谨慎：它提示 RL 可能学到新能力，但评价方式还需要继续审计。

5.3 课程给出的保守结论

课程最后给出的结论不是简单站队，而是“两者都有可能”：

- 在 RL 训练初期，模型可能主要是在提高已有正确路径的概率。
- 在训练足够久、reward signal 合适时，模型也可能展现出新的能力。
- 真正关键的问题变成：什么样的 RL 算法和 reward signal 更容易激发新能力，而不是只重排旧路径。

最终答案

AI 能不能自我修正？可以，但要分层理解。Decoding 层能做分布干预，workflow 层能做自动检查，reasoning/RL 层可能让修正行为内化。最可靠的自我修正往往不是“模型自己想一想”这么简单，而是分布干预、外部反馈、算力预算和训练信号共同作用的结果。

5.4 本章小结

自我修正研究已经从“让模型再想一次”推进到更细的问题：正确路径是否已在 base model 分布中、RL 是否只改变概率、过程正确性如何评价、reward signal 如何设计。课程的价值在于把看

[27] 视频画面时间区间：01:24:20–01:26:50。

似分散的 decoding、workflow、reasoning 文献，放进同一条技术发展线里。

6 学习路线与复习问题

6.1 一页复习图

层次	核心机制	主要风险
Decoding	构造好分布与坏分布，解码时做对比，代表方法包括 CD、DoLa、CAD、VCD、MTI。	负来源构造不当会无效；额外计算成本可能高；不代表模型真正理解错误。
Workflow	生成后自动插入 verification/reflection 指令，要求模型检查或修正。	内部反思不稳定；提示词会改变行为；同等算力下可能不如 majority vote。
Reasoning/RL	用可验证 reward 训练完整推理过程，让检查和修正在 reasoning 中自然出现。	训练成本高；过程是否真的正确难评价；RL 到底产生新能力还是重排旧能力仍有争议。

6.2 复习问题

1. 为什么“模型知道正确事实”不等于“模型会自我修正”?
2. Contrastive Decoding 中的“坏分布”可以从哪些来源构造? 这些来源各有什么风险?
3. MTI 为什么要考虑 KV cache? 如果只在少数 token 做 contrastive decoding，为什么不一定省算力?
4. CL 和 CS 分别衡量模型的什么行为? 为什么二者可能存在 tradeoff?
5. 为什么 workflow 方法必须和同等算力的 majority vote 比较?
6. RLVR 为什么适合数学和编程? 它为什么可能自然地产生 verification 行为?
7. Pass@k 能说明 base model 有 reasoning 能力吗? 为什么还需要检查过程正确性?

6.3 本章小结

学习这节课时，最重要的是不要把“自我修正”当成单一能力。它是一组机制的组合：错误可检测性、错误可修正性、何时触发修正、修正是否划算、以及训练是否让这些行为内化。把这几层拆开，才能看懂相关论文为什么经常得出看似矛盾的结论。

7 总结与延伸

这节课的主线可以压缩成一句话：AI 自我修正不是一个按钮，而是一条从 test-time decoding 到 agentic workflow，再到 RL-trained reasoning 的技术谱系。

课程的三个面向分别回答了三个问题:

- 如果不训练，能不能让输出更正确？ Contrastive Decoding 家族回答“可以尝试”，前提是能构造有意义的对比分布。
- 如果让模型再检查一次，是否就能自我修正？ Workflow 文献回答“有时可以，但不稳定，而且要和同等算力 baseline 比较”。
- 如果训练模型做 **reasoning**，会不会自然学会反思？ RLVR 文献回答“可能会”，但现在仍在争论这是新能力还是旧路径概率提升。

对实践者来说，最稳妥的策略不是盲目相信“请你反思一下”，而是按成本和可靠性选择层次：

1. 低成本尝试：用 contrastive decoding 或简单多采样检查是否有明显收益。
2. 需要可靠性：优先引入外部反馈，例如工具执行、检索、unit test、checklist 或 verifier。
3. 预算充足且任务可验证：考虑 RLVR 或专门训练，让模型把 verification/refinement 行为内化。

课程最后的开放问题也值得保留：什么样的 reward signal 能真正激发新能力？什么样的过程验证才可信？怎样区分“模型本来会，只是低概率”与“训练后真的学会了”？这些问题仍然是 self-correction 和 reasoning 研究的前沿。