

弹性 MapReduce

TCRL 模型微调服务



腾讯云

【 版权声明 】

©2013–2026 腾讯云版权所有

本文档（含所有文字、数据、图片等内容）完整的著作权归腾讯云计算（北京）有限责任公司单独所有，未经腾讯云事先明确书面许可，任何主体不得以任何形式复制、修改、使用、抄袭、传播本文档全部或部分内容。前述行为构成对腾讯云著作权的侵犯，腾讯云将依法采取措施追究法律责任。

【 商标声明 】



及其它腾讯云服务相关的商标均为腾讯云计算（北京）有限责任公司及其关联公司所有。本文档涉及的第三方主体的商标，依法由权利人所有。未经腾讯云及有关权利人书面许可，任何主体不得以任何方式对前述商标进行使用、复制、修改、传播、抄录等行为，否则将构成对腾讯云及有关权利人商标权的侵犯，腾讯云将依法采取措施追究法律责任。

【 服务声明 】

本文档意在向您介绍腾讯云全部或部分产品、服务的当时的相关概况，部分产品、服务的内容可能不时有所调整。您所购买的腾讯云产品、服务的种类、服务标准等应由您与腾讯云之间的商业合同约定，除非双方另有约定，否则，腾讯云对本文档内容不做任何明示或默示的承诺或保证。

【 联系我们 】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100或 95716。

文档目录

TCRL 模型微调服务

产品简介

TCRL 概述

TCRL 优势

模型与定价

TCRL SDK

快速开始

LoRA 入门

Loss Functions (损失函数)

总览

Cross-Entropy (交叉熵)

Importance Sampling (重要性采样)

PPO (近端策略优化)

DRO (直接奖励优化)

CISPO (剪裁重要性采样策略优化)

Custom Loss Functions (自定义损失函数)

CLI 参考

CLI 参考说明

tinker run

tinker checkpoint

COOKBOOK

快速开始

Supervised Learning (监督学习)

Reinforcement Learning (强化学习)

概述

RL Training Outputs (强化学习训练输出)

Evaluation (评测)

概述

Benchmarks Guide (基准测试指南)

Customizing Benchmarks (自定义基准测试)

Storage (存储)

Preferences (从偏好中学习)

概述

RLHF (基于人类反馈的强化学习)

DPO (直接偏好优化)

TCRL 模型微调服务

产品简介

TCRL 概述

最近更新时间：2026-10-19 21:18:02

腾讯云模型微调服务（Tencent Cloud Reinforcement Learning，以下简称 TCRL）是面向大模型的云原生 Serverless 模型微调服务，提供监督微调、强化学习、任务追踪和用量监控的一站式能力，让用户更聚焦于数据、算法与训练逻辑本身。用户可以从本地或云端等任意开发环境通过 SDK 提交训练任务，由 TCRL 在云端执行，并在控制台查看任务进度、详情与资源用量。

产品定位

TCRL 保留用户对训练循环和所有算法细节的完全控制——它不是让微调变"简单"的黑盒，而是一个在屏蔽分布式训练复杂性的同时保留控制力的清晰抽象。用户只需在 CPU 机器上编写一个简单的训练循环，包括数据/环境定义和损失函数。TCRL 负责将训练高效地分布到多 GPU 上执行，更换模型只需修改代码中的一个字符串。以下是职责划分：

用户负责	用户编写	TCRL 负责
数据与 RL 环境 — 自定义训练数据，在 CPU 上运行	简单 Python 脚本	大模型高效分布式训练 (Qwen3.5-122B-A10B、 Qwen3.8-27B-128K 等)
训练逻辑 — 损失函数、 训练循环、评测	API 调用： forward_backward()、 optim_step()、sample()、 save_state()	可靠性 — 硬件故障透明处理

产品架构

TCRL：面向大模型的云原生 Serverless 后训练服务

技术架构

大模型后训练通常涉及数据准备、资源申请、分布式环境搭建、GPU 调度、训练服务部署、日志观测、实验管理、效果评测与持续迭代等；TCRL 将这些复杂能力平台化封装，帮助团队以标准化流程快速完成从SFT到RL的后训练闭环



功能特性

TCRL 服务当前支持的功能：

- TCRL 支持微调 1B 到 1T+ 参数的开源模型，包括稠密模型和 MoE 架构。完整列表见 [模型与定价](#)。
- TCRL 采用 LoRA 微调而非全量微调。在许多重要场景（尤其是 RL）中，LoRA 与全量微调性能相当（参见 [LoRA Without Regret](#)）。
- 您可以下载训练好的模型权重，在 TCRL 之外使用。

功能速览

TCRL 的核心功能集中在以下几个关键函数：

- forward_backward：输入数据和损失函数，计算并累积梯度。
- optim_step：使用累积的梯度更新模型参数。
- sample：从训练好的模型生成输出。
- 其他用于保存/加载权重和优化器状态的函数。

TCRL 优势

最近更新时间：2026-10-19 21:18:02

过去传统模型训练平台方案

在模型训练平台上完成一次大模型训练，整体要经历五个环节，每个环节都存在明显的工程瓶颈与痛点。

痛点1：算力获取

训练开始前，需要先申请 GPU 卡时配额，或者走采购流程自行购买。这一步的问题在于周期长、供给不确定：采购往往按月计，热门卡型常常一卡难求，申请下来也要等待较长时间；更关键的是算力无法按需弹性伸缩，实验做到一半想加卡时往往加不上，算力供给的节奏跟不上实验迭代的节奏。

痛点2：环境与训练引擎搭建

拿到 GPU 卡后，需要准备基础镜像，安装 CUDA、驱动、NCCL 等依赖，并选定训练引擎，例如 Megatron-LM、FSDP 或 DeepSpeed；若是多机多卡场景，还要额外配置机器间的通信，包括 NVLink、IB/RDMA 以及 NCCL 拓扑等。这一环的痛点是依赖链路长、版本兼容问题多，环境的复现和跨任务复用成本高；通信配置的门槛尤其高，拓扑或带宽没对齐，会直接拖垮整体训练速度。

痛点3：数据与模型准备

需要把语料、词表和基座权重（从几十 GB 到 TB 级）上传并做清洗、打包与预处理。数据搬运耗时长、存储成本高，而且数据格式与训练框架强耦合，换一个模型往往要重新上传一遍，前期投入难以沉淀复用。

痛点4：调试与性能调优

这一环最为耗时。跑通训练、排查 OOM 与 NaN、调整 batch size 与并行策略、做显存和通信优化，基本都要靠 ssh 登录机器看日志、打堆栈，缺少可视化和可观测手段；性能优化高度依赖个人经验，一次实验动辄几小时到几天。同时长周期训练缺乏完善的容错与断点续训机制，中途失败的代价极大。

痛点5：资源调度与多人协作

平台上任务对资源是独占的，一个训练任务占用 N 张卡，其他任务就只能使用剩下的卡，多人使用必须排队。这导致 GPU 空转与任务排队同时存在。此外不同模型的规模与结构差异大，需要单独配置并行策略、单独做训练引擎优化，进一步抬高了重复投入。

TCRL 云原生 Serverless 模型微调服务

定位：TCRL 是面向大模型的云原生 Serverless 模型微调服务，提供监督微调、强化学习、任务追踪和用量监控的一站式能力，让用户更聚焦于数据、算法与训练逻辑本身。在 TCRL 上完成一次大模型训练，对比传统训练方式的5个痛点，TCRL 产品优势如下：

1. 算力获取: 从"申请配额、按月采购"到"即开即练"

- **开箱即用**：训练前只需在控制台获取 API Key 即可发起训练，无需提前申请 GPU 卡时配额，也无需走采购流程持有物理卡。
- **按量付费**：按数据量付费，不为闲置卡时买单。
- **算力供给与实验节奏解耦**：想换规模、起新实验，直接再提交一个任务即可，不必先解决"卡从哪来"的问题，算力不再成为迭代速度的瓶颈。从"搭环境、管集群"转向"调数据、调策略"。沿用本地或云端的既有开发习惯，通过标准 API Keys + SDK 提交任务，训练交由云端完成，10 分钟即可启动首个任务。

2. 环境与训练引擎搭建:从"装依赖、配通信拓扑"到"装个 SDK 就能训练"

- **环境要求**：本地只需安装 SDK 并保持联网，无需 GPU，从本地机器、云端 Notebook 或任意开发环境都能提交训练任务，仅需要一台能联网的终端，就能发起大规模训练。
- **用户无感**：所有底层依赖，以及训练引擎的选型与适配，全部由 TCRL 负责，用户无需感知。

3. 数据与模型准备:从"TB 级搬运、换模型重传"到"model id 直取、边训练边传输"

- **模型侧**：
 - 无需下载和上传基座权重，通过 model id 直接指定，平台侧完成权重的准备与加载；切换模型只改一个参数，不产生额外的搬运成本。
 - 基于 LoRA 覆盖 SFT、RLHF、GRPO 等训练链路，统一训练入口、模型版本、实验记录与 Checkpoint，降低多框架协同成本，并支持自定义 Loss。
- **数据侧**：训练数据通过 API 在训练过程中上传，无需在任务启动前完成全量搬运与预处理，快速开启训练。

4. 调试与性能调优:从"登机翻日志、凭经验调参"到"调优托管、可视化干预"

- **性能调优托管**：并行策略、显存优化、通信优化由 TCRL 在云端自动完成；用户无需手工调整 batch size 与并行策略，也不会因这些配置变化触发 OOM。
- **弹性扩缩容**：训练资源由平台自动调度与弹性伸缩，保障任务稳定跑到结束，用户全程无感。任务提交即调度，多个实验可并行推进，不再受限于本地或集群的空闲卡数，试错和调参周期显著缩短。
- **训练全过程可观测与干预**：
 - TCRL 控制台集中呈现任务进度、详情、日志、Checkpoint 与资源用量，并支持 Checkpoint 下载；
 - 借助 Cookbook，可在本地实时查看训练效果数据，随时判断收敛状况并终止任务，便于效果对比、问题定位、失败恢复与成本治理；不再依赖 ssh 登机翻日志、打堆栈。
- **容错与续训**：支持检查点保存与重加载，长周期训练中断后可从断点恢复，大幅降低中途失败的代价。

5. 资源调度与多人协作:从"独占排队、GPU 空转"到"池化调度、多任务并发"

- **资源调度**：资源由 TCRL 统一池化调度，任务不再独占固定数量的卡，GPU 空转与任务排队并存的情况被消除。
- **任务并发**：支持并发提交多个任务，多人、多实验可同时进行且互不阻塞，团队整体迭代效率显著提升。
- **平台托管**：不同规模、不同结构的模型无需各自配置并行策略、单独做引擎优化，重复投入由平台吸收，用户只关注模型效果本身。资源分配、Ray 分布式环境、训练编排与弹性伸缩均由平台统一处理，用户无需感知。

6. 大数据生态原生集成

- **大数据生态：**与 TCLake/COS、TCCatalog、Xpark、TCRay、EMR、DLC 和 Ray Serve 打通，形成"数据准备—模型后训练—实验追踪—推理部署"的端到端闭环。

使用流程

1. 用户在控制台获取 API Keys。
2. 用户可从本地、云端 Notebook 或其他任意开发环境，通过 SDK 提交训练任务。
3. TCRL 在云端执行训练任务，平台托管 GPU 资源、分布式训练环境与任务调度。
4. 用户在控制台查看任务进度、任务详情、日志、Checkpoint 与资源用量。
5. 训练完成后，模型版本可进入评测、Ray Serve 在线推理和持续迭代流程。

核心能力

- **监督微调：**支持 SFT、LoRA 等模型微调方式。
- **强化学习：**覆盖 RLHF、GRPO 等后训练链路。
- **Serverless 托管：**云端托管 GPU、Ray 分布式环境、训练编排与调度。
- **任务追踪：**统一查看训练进度、任务详情、日志、实验和 Checkpoint。
- **用量监控：**展示资源消耗与积分用量，支持训练成本分析。
- **训推衔接：**训练模型可进一步联动 Ray Serve 完成在线部署。

模型与定价

最近更新时间：2026-10-19 21:18:02

模型与定价

当前 TCRL 以积分收费。1 元 = 1000（1K）积分，Token 单位为百万（M）。Prefill、Sample、Train 定价单位为：K 积分/M Token，即每百万 Token 消费多少 K 积分。以 Qwen3-8B Prefill 为例，每消耗 1M Token，需消费 6.58K 积分，花费6.58元。

Model	Model ID	Type	Architecture	Size	Active	Context	Prefill (K 积分/M Token)	Sample (K 积分/M Token)	Train (K 积分/M Token)
Qwen 3-8B	Qwen/Qwen3-8B	Base	Dense	8 B	8 B	16 K	6.58	20.25	14.85
Qwen 3.5-4B	Qwen/Qwen3.5-4B	Hybrid	Dense	4 B	4 B	16 K	11.14	33.92	24.87
Qwen 3.5-9B	Qwen/Qwen3.5-9B	Hybrid	Dense	9 B	9 B	16 K	22.28	67.33	49.38
Qwen 3.5-27B	Qwen/Qwen3.5-27B	Hybrid	Dense	27 B	27 B	16 K	18.23	45.06	39.72
Qwen 3.5-35B-A3B	Qwen/Qwen3.5-35B-A3B	Hybrid	MoE	35 B -	3 B	16 K	18.23	45.06	39.72

		rid		A3B					
Qwen 3.6-27B	Qwen/Qwen3.6-27B	Hybrid	Dense	27B	27B	16K	62.78	188.83	138.48
Qwen 3.6-35B-A3B	Qwen/Qwen3.6-35B-A3B	Hybrid	MoE	35B-A3B	3B	16K	18.23	45.06	39.72
Qwen 3.5-122B-A10B	Qwen/Qwen3.5-122B-A10B	Hybrid	MoE	122B-A10B	10B	16K	54.68	135.17	119.17
Qwen 3.8-27B	Qwen/Qwen3.8-27B	Hybrid	Dense	27B	27B	16K	62.80	188.85	138.50
Qwen 3.8-27B-128K	Qwen/Qwen3.8-27B-128K	Hybrid	Dense	27B	27B	128K	125.60	377.70	277.00

模型术语

- **Type:** 模型类型
 - **Base (基础模型)**：未经对齐、指令微调或 Chat 调优的原始预训练模型。最适合用于训练后（Post-training）研究，或者由您自己运行完整的训练后流程。
 - **Reasoning (推理模型)**：在给出最终答案之前，始终会先生成思考链（Chain-of-thought）。具有最高水平的智能度，但延迟和 Token 成本也相对更高。

- **Hybrid（混合模型）**：可以在“思考”和“非思考”两种模式下运行。默认开启推理，但可以通过渲染器（Renderer）或参数禁用思考链，从而获得更快、更便宜的直接回答。

- **Architecture：模型架构**

在 AI 和大语言模型（LLM）中，MoE 和 Dense 指的是两种不同的模型架构设计，核心区别在于每次处理数据时，模型中有多少参数在参与计算：

- **Dense（密集架构）**

- 工作原理：处理每一个 Token（字符/词）时，模型中的所有参数都会被全部激活并参与计算。
- 特点：
 - i. 结构简单直接：传统的经典模型（如早期 GPT 系列、LLaMA 等）大多是 Dense 架构。
 - ii. 计算成本高：模型越大，每次推理/训练所需的算力（GPU 显存和计算资源）就成线性增加。
 - iii. 输出稳定：全量参数参与，推理逻辑非常统一。

- **MoE（Mixture-of-Experts，混合专家架构）**

- 工作原理：把一个大模型拆分成许多个小模型（称为“专家 / Experts”），并配有一个“路由网络（Router）”。处理每一个 Token 时，路由网络会自动判断这个词最适合哪个专家，仅激活一小部分专家的参数（例如总参数 100B，但每次只调用 10B）。
- 特点：
 - i. 计算高效：拥有极大的总参数量（知识库广），但每次计算激活的参数量（Active Params）很少，因此计算速度更快、算力开销更低。
 - ii. 计算成本低：在提供与大型 Dense 模型相当的性能/效果的同时，大大降低了推理和训练成本。

说明：

MoE 模型按活跃参数计费，相比同等质量的稠密模型更具性价比。

- **Size**：模型规模的总参数量，由模型发布方声明；
- **Active**：激活参数量，每个 Token 调用的参数量 — 密集模型（Dense）与 Size 相同，混合专家模型（MoE）则为 Size 的一部分。
- **Context**：上下文长度（最大序列长度）；
- **Prefill**：处理输入/prompt token（仅前向传播）
- **Sample**：生成输出 token（前向传播 + 采样）
- **Train**：前向和反向传播，计算梯度

模型选择指南

场景	推荐
研究与后训练	使用 Base 模型

特定任务微调	从 Instruction 或 Hybrid 模型开始
低延迟	使用 Instruction 模型（无思维链）
高智能	使用 Reasoning 或 Hybrid 模型（带思维链）
视觉任务	使用类型中包含 Vision 的模型

TCRL SDK

快速开始

最近更新时间：2026-10-19 21:47:32

安装

```
uv pip install tinker==0.12.0
```

ⓘ 注意：

目前 TCRL 开白体验中，可联系 tcrl@tencent.com 申请开白后，登录 [腾讯云弹性 MapReduce \(EMR\) 控制台](#) 获取 API Key，进行模型训练、实验管理和用量监控。

```
export TINKER_API_KEY="your-api-key-here"
export TINKER_BASE_URL="http://api.tencenttcrl.com/"
```

安装后即可使用 Python SDK (`import tinker`) 和 CLI (`tinker run list` 、 `tinker checkpoint download`) 。

本页介绍两种主要的 LLM 微调 workflow：监督微调（SFT）和强化学习（RL），并展示每一步如何对应 TCRL SDK 调用。

监督微调（SFT）

SFT 训练模型模仿示例：

1. **创建客户端**：连接 Tinker，创建 `TrainingClient`
2. **准备数据**：将样本 tokenize 为带 loss mask 的 `Datum` 对象
3. **训练**：`forward_backward`（梯度）+ `optim_step`（更新权重）
4. **评测** — 保存权重，创建 `SamplingClient`，采样

SFT 监督微调训练流程



强化学习（RL）

RL 训练模型最大化奖励信号：

1. **创建客户端**：连接 Tinker，创建 TrainingClient
2. **获取同策略采样器**： `save_weights_and_get_sampling_client`
3. **采样 rollout**：从同策略模型生成补全
4. **评分**：计算奖励和对数概率
5. **训练**：使用 RL 损失进行 `forward_backward` + `optim_step`
6. **重复**：新权重 → 新 SamplingClient → 再次采样

RL 在线策略训练循环



API 速查

创建客户端

```
import tinker
from tinker import types

# 入口 - 从环境变量读取 TINKER_API_KEY
service_client = tinker.ServiceClient()

# 训练客户端 (LoRA 微调)
training_client = service_client.create_lora_training_client(
    base_model="Qwen/Qwen3-8B", rank=32
)

# 采样客户端 (文本生成)
sampling_client = service_client.create_sampling_client(
    base_model="Qwen/Qwen3-8B"
)

# 分词器
tokenizer = training_client.get_tokenizer()
```

① 注意

子进程采样 (SamplingClient) 可被 pickle——你可以将它传递给其他进程进行并行采样。如果训练循环中有 CPU 密集型工作 (评分、环境逻辑), 设置 `TINKER_SUBPROCESS_SAMPLING=1` 可以在专用子进程中运行 `sample()` 和 `compute_logprobs()`, 避免 GIL 争用。

准备训练数据

`Datum` 是单个训练样本, 包含输入 token、目标 token 和逐 token 的损失权重 (0 = 忽略, 1 = 计算损失)。

```
datum = types.Datum(
    model_input=types.ModelInput.from_ints(tokens=input_tokens),
    loss_fn_inputs=dict(
        weights=weights,          # 0 for prompt, 1 for completion
        target_tokens=target_tokens # shifted by 1 from input
    )
)
```

对于 RL 损失 (importance_sampling、PPO、CISPO), `loss_fn_inputs` 还需包含 `logprobs` 和 `advantages` :

```
rl_datum = types.Datum(
    model_input=types.ModelInput.from_ints(tokens=tokens),
    loss_fn_inputs=dict(
        target_tokens=target_tokens,
        weights=weights,
        logprobs=sampling_logprobs, # from the rollout policy
        advantages=advantages,     # reward - baseline
    )
)
```

文本采样

```
prompt = types.ModelInput.from_ints(tokenizer.encode("The capital of
France is"))
params = types.SamplingParams(max_tokens=50, temperature=0.7, stop=
["\n"])

# 单次采样
result = await sampling_client.sample_async(prompt=prompt,
num_samples=1, sampling_params=params)
```

```
print(tokenizer.decode(result.sequences[0].tokens))

# 一次调用多次采样
result = await sampling_client.sample_async(prompt=prompt,
num_samples=8, sampling_params=params)
for seq in result.sequences:
    print(tokenizer.decode(seq.tokens))
```

计算对数概率

用于 RL 中的评分（比较训练策略与采样策略）。

```
# Prompt logprobs
result = await sampling_client.sample_async(
    prompt=prompt, num_samples=1,
    sampling_params=types.SamplingParams(max_tokens=1),
    include_prompt_logprobs=True,
)
print(result.prompt_logprobs)

# 简写
logprobs = await sampling_client.compute_logprobs_async(prompt)

# Top-k logprobs（用于蒸馏）
result = await sampling_client.sample_async(
    prompt=prompt, num_samples=1,
    sampling_params=types.SamplingParams(max_tokens=1),
    include_prompt_logprobs=True,
    topk_prompt_logprobs=5,
)
```

前向-反向传播

计算给定数据和损失函数的梯度，立即返回一个 future。

```
# SFT: cross-entropy loss
fwdbwd_future = await training_client.forward_backward_async(data=
[datum], loss_fn="cross_entropy")
fwdbwd_result = await fwdbwd_future.result_async()
print(f"Loss: {fwdbwd_result.loss}")
```

```
# RL losses
fwdbwd_future = await training_client.forward_backward_async(data,
"importance_sampling")
fwdbwd_future = await training_client.forward_backward_async(data,
"ppo")
fwdbwd_future = await training_client.forward_backward_async(data,
"cispo")
fwdbwd_future = await training_client.forward_backward_async(data,
"dro")

# 自定义损失
fwdbwd_future = await
training_client.forward_backward_custom_async(data, my_loss_fn)
```

各损失函数的数学原理见 [损失函数](#)。

优化器步进

使用上次 `forward_backward` 的梯度更新模型权重。

```
optim_future = await training_client.optim_step_async(
    types.AdamParams(learning_rate=1e-4)
)
await optim_future.result_async()
```

保存与加载权重

```
# 保存权重 → 获取采样客户端
sampling_client =
training_client.save_weights_and_get_sampling_client(name="checkpoint-
1")

# 保存完整状态（权重 + 优化器）
training_client.save_state(name="step-100")

# 仅从权重恢复
training_client = await
service_client.create_training_client_from_state_async(
    path="tinker://run-id/sampler_weights/checkpoint-1"
```

```
)

# 带优化器状态恢复
training_client = await
service_client.create_training_client_from_state_with_optimizer_async(
    path="tinker://run-id/weights/step-100"
)
```

视觉输入

```
import requests

image_data = requests.get("https://example.com/image.png").content

model_input = tinker.ModelInput(chunks=[
    types.EncodedTextChunk(tokens=tokenizer.encode("<|im_start|>user\n<|vision_start|>")),
    types.ImageChunk(data=image_data, format="png"),
    types.EncodedTextChunk(tokens=tokenizer.encode("<|vision_end|>Describe this image<|im_end|>\n<|im_start|>assistant\n")),
])
```

并发请求

```
import asyncio

# 同一 prompt 多次采样
result = await sampling_client.sample_async(prompt=prompt,
num_samples=16, sampling_params=params)
for seq in result.sequences:
    print(tokenizer.decode(seq.tokens))

# 不同 prompt 并行采样
results = await asyncio.gather(
    sampling_client.sample_async(prompt=prompt1, num_samples=1,
sampling_params=params),
    sampling_client.sample_async(prompt=prompt2, num_samples=1,
sampling_params=params),
```

```
sampling_client.sample_async(prompt=prompt3, num_samples=1,
sampling_params=params),
)

# 流水线训练: overlap forward-backward 与 optimizer step
fwdbwd = await training_client.forward_backward_async(batch1,
"cross_entropy")
optim_future =
training_client.optim_step(types.AdamParams(learning_rate=1e-4))
next_fwdbwd = await training_client.forward_backward_async(batch2,
"cross_entropy")
await optim_future
```

LoRA 入门

最近更新时间：2026-10-19 21:18:01

TCRL 支持 [LoRA 微调](#)——仅调整少量参数，而非全量微调中调整原始模型的所有参数。

LoRA 是什么？

LoRA 全称 Low-Rank Adaptation。原始模型有权重矩阵 W ，我们将其替换为 $W' = W + BA$ ，其中 B 和 A 是低秩矩阵。若 W 是 $n \times n$ 矩阵，则 B 和 A 分别为 $n \times r$ 和 $r \times n$ 矩阵， r 为低秩近似的秩。TCRL 默认 rank 为 32。

LoRA 使用低秩近似这一事实并不关键。我们更倾向于将 LoRA 视为参数空间的一个随机投影，恰好可以高效实现。在 RL 或小规模 SL 训练中，我们只学习少量信息，缩减的参数集绰绰有余。

背景说明

目前的理解是：在 RL 或小规模 SL 数据集上，LoRA 与全量微调性能相当；在更大规模数据集上，LoRA 性能较弱。具体而言：

- 在中小规模的指令微调和推理数据集上，LoRA 与全量微调性能相同。
- 当数据集超过 LoRA 容量时，LoRA 不如全量微调。损失不会达到一个明确的下限，而是训练效率下降，取决于模型容量与数据集大小的关系。
- 在某些场景下，LoRA 对大 batch size 的容忍度低于全量微调——超过某个阈值后损失惩罚更大。增大 LoRA rank 不能缓解此问题，这是矩阵乘积参数化的固有属性。
- 即使在小数据场景下，将 LoRA 应用于所有权重矩阵（尤其是 MLP 和 MoE 层）效果更好。仅对 Attention 做 LoRA 效果较差，即使提高 rank 以匹配参数量也不够。
- RL 中即使使用小 rank，LoRA 与全量微调性能也相当。RL 所需的模型容量很低，这与信息论分析一致。

更多细节和实验结果见 [LoRA Without Regret](#)。

超参数

学习率（LR）通常是 ML 实验中最重要超参数。LoRA 需要比全量微调大得多的 LR——通常大 20–100 倍，取决于模型大小。常见错误是在迁移到 LoRA 时保留了全量微调的 LR，导致误认为 LoRA 效果差。

计算正确的 LoRA 学习率：

```
from tinkercookbook.hyperparam_utils import
get_lora_lr_over_full_finetune_lr

model_name = "meta-llama/Llama-3.1-8B"
print(get_lora_lr_over_full_finetune_lr(model_name))
```

⚠ 注意：

Llama-3.2-1B 的系数为32，而 Llama-3.1-70B 的系数为128。

选择多大的 rank?

Tinker 默认 rank 为32。如果在大型数据集上做 SL，应使用更大的 rank。粗略来说，只要 LoRA 参数量不小于 completion token 数（即 weight=1 的 token 数），LoRA 就能取得良好效果。可以用以下工具计算 LoRA 参数量：

```
from tinker_cookbook.hyperparam_utils import get_lora_param_count

model_name = "meta-llama/Llama-3.1-8B"
print(get_lora_param_count(model_name, lora_rank=32))
```

Loss Functions（损失函数）

总览

最近更新时间：2026-10-19 21:18:03

损失函数（Loss Function）是一个用来衡量模型“当前预测结果”与“真实标准答案”之间差异大小的数学函数。简单来说就是评估 AI 预测得“好不好”的打分标准，或者说是一个“惩罚机制”。在机器学习和深度学习中，它的核心作用是：衡量模型当前的预测输出（Prediction）与真实的标准答案（Ground Truth）之间的差距。输出的数值被称为 Loss（损失值/误差值）。损失值越小，代表模型的预测越准确；损失值越大，代表模型的误差越大。

在不同类型的任务中，使用的损失函数也不同：

1. 分类任务 / 大语言模型（SFT 阶段即监督学习）：

- 常用算法：[Cross-Entropy（交叉熵）](#)
- 作用：让模型在猜下一个词（例如判断“苹果”是水果还是手机）时，给正确选项打出最高的分数/概率。
- 目标：衡量预测的类别概率分布与真实标签的差异，让正确选项的概率接近 1。

2. 强化学习（RL 阶段）：

- 常用算法：[PPO](#)、[Importance Sampling](#)、[CISPO](#)
- 作用：根据人类或环境给出的好/坏反馈（Rewards/Advantages），调整模型说出某句话的概率，同时防止模型“改得太猛而训练崩塌”。
- 目标：根据环境给出的奖励/惩罚（Advantage），调整模型生成某段回答的概率，直到接近最佳答案。

3. 数值预测 / 回归任务：

- 常用算法：[MSE（均方误差）](#)
- 作用：计算预测数值（如预测房价 500 万）与真实数值（如实际售价 520 万）之间的距离平方差。
- 目标：计算预测数值（如预测房价）与真实数值之间的物理距离或平方差。

TCRL 提供用于监督学习和强化学习的内置损失函数。您可以通过向 `forward_backward` 传入一个字符串来选择损失函数：

```
future = await training_client.forward_backward_async(data,
loss_fn="cross_entropy")
result = await future.result_async()
```

总览

损失函数	loss_fn	类型	说明
Cross-Entropy (交叉熵)	<code>cross_entropy</code>	SFT	最大化目标 Token 的对数概率
Importance Sampling (重要性采样)	<code>importance_sampling</code>	RL (policy gradient)	带重要性加权的离策略策略梯度
PPO (近端策略优化)	<code>ppo</code>	RL (clipped)	用于稳定策略更新的裁剪式替代目标函数
DRO (直接奖励优化)	<code>dro</code>	RL (off-policy)	面向离策略 / 离线 RL 的二次惩罚项
CISPO (裁剪重要性采样策略优化)	<code>cispo</code>	RL (clipped grad)	用于策略梯度的分离式裁剪重要性比
Custom Loss Function (自定义损失函数)	—	任意	通过 <code>forward_backward_custom</code> 定义的用户自定义损失

工作原理

每个训练样本都是一个 `Datum` ——一条包含计算损失所需全部信息的序列。一个 `Datum` 包含：

- `model_input`：token 序列（形状：`(N,)`，其中 N 为序列长度）
- `loss_fn_inputs`：损失函数所需的张量字典（`targets`、`weights`、`advantages` 等）

核心设计原则：损失所需的一切都在 `Datum` 中。这意味着每个 `Datum` 都是自包含的，没有批次层面的状态，也不需要外部查找。

SFT 示例

```
import tinker
from tinker import types

datum = types.Datum(
    model_input=types.ModelInput.from_ints(input_tokens), # shape: (N,)
    loss_fn_inputs={
        "target_tokens": target_tokens, # shape: (N,) - what to predict
        "weights": weights, # shape: (N,) - 0 for prompt, 1 for completion
    }
)
```

```
future = await training_client.forward_backward_async([datum],
loss_fn="cross_entropy")
result = await future.result_async()
print(f"Loss: {result.loss}")
```

RL 示例

对于 RL 损失，Datum 还需要包含采样对数概率和优势：

```
rl_datum = types.Datum(
    model_input=types.ModelInput.from_ints(tokens),      # shape: (N,)
    loss_fn_inputs={
        "target_tokens": target_tokens,                 # shape: (N,)
        "weights": weights,                             # shape: (N,)
        "logprobs": sampling_logprobs,                   # shape: (N,) -
from the rollout policy
        "advantages": advantages,                       # shape: (N,) -
reward signal per token
    }
)

future = await training_client.forward_backward_async([rl_datum],
loss_fn="importance_sampling")
```

Cross-Entropy（交叉熵）

最近更新时间：2026-10-19 21:18:03

交叉熵（Cross-Entropy）是信息论和机器学习中用于衡量两个概率分布之间差异程度的数学量。在大语言模型（LLM）和分类任务中，交叉熵常被用作 损失函数（Loss Function），用来量化“模型的预测概率分布”与“真实标签分布”之间的差距。

标准交叉熵损失（负对数似然），它优化策略 p_θ ，使目标 token x 的对数概率最大化：

$$\mathcal{L}(\theta) = -\mathbb{E}_x[\log p_\theta(x)]$$

其中 `weights` 取值非 0 即 1，通常由 `renderer.build_supervised_example()` 生成，该函数返回 `(model_input, weights)`，用于指定需要参与训练的 `assistant` 轮次。

实现

```
# Apply weights and compute elementwise loss
elementwise_loss = -target_logprobs * weights
# Apply sum reduction to get the total loss
loss = elementwise_loss.sum() # scalar
```

输入张量

Key	Shape	说明
<code>target_tokens</code>	<code>(N,)</code> 或 <code>(N, K)</code>	目标 token ID
<code>weights</code>	<code>(N,)</code> 或 <code>(N, K)</code>	逐 token 的损失权重（通常来自 <code>renderer</code> ）

输出张量

Key	Shape	说明
<code>logprobs</code>	<code>(N,)</code> 或 <code>(N, K)</code>	所请求目标 token 的对数概率

诊断信息

- `loss:sum`（标量）— 加权交叉熵损失之和

Top-K 蒸馏

当输入张量的形状为 (N, K) 时，`cross_entropy` 会提取学生模型在每个位置上 K 个目标 token 的 logprobs，并计算：

$$\mathcal{L}_{\text{CE-topK}}(\theta) = - \sum_t \sum_{k=1}^K w_{t,k} \cdot \log p_{\theta}(x_{t,k})$$

其中 $w_{t,k}$ 为用户提供的权重。对于软目标蒸馏，令 $w_{t,k} = \tilde{p}_{\text{teacher}}(x_{t,k})$ ，即教师模型在 top-K token 上重新归一化后的概率。对于硬目标训练，则将单个候选 token 的权重置为 1.0，其余置为 0.0。

示例：将 Qwen3-235B 蒸馏为 Qwen3-30B

下面的示例从教师模型采样一段补全内容，在每个生成位置恢复其 top-K 分布，并训练学生模型去拟合该分布。

```
import tinker
import torch
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen3-235B-A22B-Instruct-2507")
K = 20

service_client = tinker.ServiceClient()
teacher_sampling_client = await
service_client.create_sampling_client_async(
    base_model="Qwen/Qwen3-235B-A22B-Instruct-2507"
)
student_training_client = await
service_client.create_lora_training_client_async(
    base_model="Qwen/Qwen3-30B-A3B-Instruct-2507"
)

messages = [{"role": "user", "content": "Write a Python function to
compute the nth Fibonacci number efficiently."}]
prompt_tokens = tokenizer.apply_chat_template(messages,
add_generation_prompt=True)

# 1. Sample a completion from the teacher.
sample_response = await teacher_sampling_client.sample_async(
    prompt=tinker.ModelInput.from_ints(prompt_tokens),
    num_samples=1,
```

```
sampling_params=tinker.SamplingParams(max_tokens=512,
temperature=0.7),
)
sampled_tokens = list(sample_response.sequences[0].tokens)

# 2. Teacher-force the completion to recover top-K logprobs at each
position.
teacher_forced = tinker.ModelInput.from_ints(prompt_tokens +
sampled_tokens)
topk_response = await teacher_sampling_client.sample_async(
    prompt=teacher_forced,
    num_samples=1,
    sampling_params=tinker.SamplingParams(max_tokens=1),
    include_prompt_logprobs=True,
    topk_prompt_logprobs=K,
)

output_toks_with_logprobs =
topk_response.topk_prompt_logprobs[len(prompt_tokens):]
seq_len = len(output_toks_with_logprobs)

teacher_tokens = torch.tensor(
    [[tok_id for tok_id, _ in row] for row in
output_toks_with_logprobs], dtype=torch.long
) # [seq_len, K]
teacher_logprobs = torch.tensor(
    [[lp for _, lp in row] for row in output_toks_with_logprobs]
) # [seq_len, K]

# Renormalize teacher logprobs over top-K via logsumexp.
teacher_logprobs -= torch.logsumexp(teacher_logprobs, dim=-1,
keepdim=True)

# Build shifted student_input: prompt + completion[:-1], so position t
predicts token t+1.
student_input = prompt_tokens + sampled_tokens[:-1]
gen_start = len(prompt_tokens) - 1

target_tokens = torch.zeros(len(student_input), K, dtype=torch.long)
target_tokens[gen_start : gen_start + seq_len] = teacher_tokens
```

```
weights = torch.zeros(len(student_input), K)
weights[gen_start : gen_start + seq_len] = teacher_logprobs.exp()

datum = tinker.Datum(
    model_input=tinker.ModelInput.from_ints(student_input),
    loss_fn_inputs={
        "target_tokens": target_tokens,
        "weights": weights,
    },
)

fwdbwd_future = await student_training_client.forward_backward_async(
    [datum], loss_fn="cross_entropy"
)

fwdbwd_result = await fwdbwd_future.result_async()
```

Importance Sampling（重要性采样）

最近更新时间：2026-10-19 21:18:03

Importance Sampling（重要性采样）是统计学和强化学习（RL）中的一种采样与估计方法，是一种通过利用“已知分布（旧模型/采样器）”生成的数据，来精确估计和训练“另一个未知或在新变动中的分布（新模型/学习器）”的数学方法。

- 痛点：大模型生成文本（Rollout）极其消耗算力，如果模型参数每更新一次就重新生成一批数据，训练速度会慢到无法接受。
- 解法：允许模型用“旧模型吐出的回答数据（ q ）”来训练“最新的模型（ p_θ ）”。

- 机制：通过乘上比值 $\frac{p_\theta(x)}{q(x)}$ ，如果新模型比旧模型更看好这个回答，就放大该奖励信号；反之则缩小，从而在无需重复生成数据的前提下，实现安全高效的 Off-policy 策略更新。

这是策略梯度目标函数的一个常见变体，当学习策略 p_θ 可能因非确定性、策略更新滞后或其他原因而与采样策略 q 不一致时使用。

朴素目标函数为：

$$\mathcal{L}(\theta) = \mathbb{E}_{x \sim p_\theta} [A(x)]$$

当 $x \sim q$ （采样器）与 $x \sim p_\theta$ （学习器）不匹配时，它存在偏差。重要性采样对此进行修正：

$$\mathcal{L}_{\text{IS}}(\theta) = \mathbb{E}_{x \sim q} \left[\frac{p_\theta(x)}{q(x)} A(x) \right]$$

其中：

- $\log p_\theta(x)$ ：来自学习器的 `target_logprobs`（`forward_backward` 的前向传播）
- $\log q(x)$ ：来自采样器的 `sampling_logprobs`（在采样（rollout）过程中记录）

实现

```
# Compute probability ratio
prob_ratio = torch.exp(target_logprobs - sampling_logprobs)
# Compute importance-weighted loss
loss = -(prob_ratio * advantages).sum()
```

输入张量

Key	Shape	说明
<code>target_tokens</code>	<code>(N,)</code>	目标 token ID（来自采样器 q ）
<code>logprobs</code>	<code>(N,)</code>	这些 token 的 <code>sampling_logprobs</code>
<code>advantages</code>	<code>(N,)</code>	优势值（正值用于强化，负值用于抑制）

输出张量

Key	Shape	说明
<code>logprobs</code>	<code>(N,)</code>	这些 token 的 <code>target_logprobs</code>

诊断信息

`loss:sum`（标量）：重要性加权策略梯度损失之和 $\mathcal{L}_{\text{IS}}$

PPO（近端策略优化）

最近更新时间：2026-10-19 21:18:02

PPO（Proximal Policy Optimization，近端策略优化）是一种在强化学习（RL）中广泛应用的算法（由 Schulman 等人于 2017 年提出），也是大语言模型在 RL/Alignment（对齐）阶段最核心的 Loss Function 之一。是一种通过设置“剪裁区间（Clipping）”来限制新旧策略更新幅度的强化学习算法，确保模型在自我演进和探索过程中不会因单次调整过猛而“崩溃或吐乱码”。

PPO（Schulman 等，2017）引入了一种裁剪目标函数，将策略更新限制在采样分布的邻近范围内。这可以避免破坏性的过大更新，尤其是在同一批采样（rollout）分布上执行多个梯度步时。

裁剪目标函数为：

$$\mathcal{L}_{\text{CLIP}}(\theta) = -\mathbb{E}_{x \sim q} \left[\text{clip} \left(\frac{p_{\theta}(x)}{q(x)}, 1 - \epsilon_{\text{low}}, 1 + \epsilon_{\text{high}} \right) \cdot A(x) \right]$$

最终的 PPO 损失取裁剪后与未裁剪目标函数中的较小值：

$$\mathcal{L}_{\text{PPO}}(\theta) = -\mathbb{E}_{x \sim q} \left[\min \left(\frac{p_{\theta}(x)}{q(x)} \cdot A(x), \text{clip} \left(\frac{p_{\theta}(x)}{q(x)}, 1 - \epsilon_{\text{low}}, 1 + \epsilon_{\text{high}} \right) \cdot A(x) \right) \right]$$

其中 ϵ_{low} 和 ϵ_{high} 为超参数（默认值：0.2）。请注意，裁剪与损失计算都是逐 token 应用的。

实现

```
# Compute probability ratio
prob_ratio = torch.exp(target_logprobs - sampling_logprobs)
# Apply clipping
clipped_ratio = torch.clamp(prob_ratio, clip_low_threshold,
clip_high_threshold)
# Compute both objectives
unclipped_objective = prob_ratio * advantages
clipped_objective = clipped_ratio * advantages
# Take minimum (most conservative)
ppo_objective = torch.min(unclipped_objective, clipped_objective)
# PPO loss is negative of objective
loss = -ppo_objective.sum()
```

输入张量

Key	Shape	说明
<code>target_tokens</code>	<code>(N,)</code>	目标 token ID (来自采样器 q)
<code>logprobs</code>	<code>(N,)</code>	这些 token 的 <code>sampling_logprobs</code>
<code>advantages</code>	<code>(N,)</code>	RL 中的优势值

输出张量

Key	Shape	说明
<code>logprobs</code>	<code>(N,)</code>	这些 token 的 <code>target_logprobs</code>

诊断信息

- `loss:sum` (标量) — PPO 裁剪损失之和

自定义裁剪阈值

```
fwd_bwd_future = await training_client.forward_backward_async(  
    data=data,  
    loss_fn="ppo",  
    loss_fn_config={"clip_low_threshold": 0.9, "clip_high_threshold":  
1.1}  
)  
fwd_bwd_result = await fwd_bwd_future.result_async()
```

DRO（直接奖励优化）

最近更新时间：2026-10-19 21:18:02

DRO（Direct Reward Optimization，直接奖励优化）是一种在大语言模型 RL/Alignment（对齐）阶段用于直接优化人类偏好或隐式奖励的损失函数（Loss Function）。与传统 PPO 需要额外训练独立的奖励模型（Reward Model）不同，DRO 继承了 DPO（Direct Preference Optimization）等算法的思想，通过数学变换将奖励函数直接融合到策略模型的损失计算中，从而无需复杂的两阶段强化学习过程，实现更高效、稳定的偏好对齐与优化。

DRO（[Richemond et al., 2024](#)；[Kimi Team et al., 2025](#)）是一种通用的离策略（甚至离线）RL 方法，它通过二次惩罚项来约束策略更新。请注意，该损失使用了一种不同的（软）优势估计形式，需要在客户端自行实现。

DRO 的目标函数为：

$$\mathcal{L}_{\text{DRO}}(\theta) = \mathbb{E}_{x \sim q} \left[\log p_{\theta}(x) \cdot A(x) - \frac{1}{2} \beta \left(\log \frac{p_{\theta}(x)}{q(x)} \right)^2 \right]$$

二次惩罚项 $\frac{1}{2} \beta (\log p_{\theta}/q)^2$ 可防止学习器偏离采样分布过远。

实现

```
# Compute quadratic penalty term
quadratic_term = (target_logprobs - sampling_logprobs) ** 2
# Compute DRO objective
dro_objective = target_logprobs * advantages - 0.5 * beta *
quadratic_term
# DRO loss is negative of objective
loss = -dro_objective.sum()
```

输入张量

Key	Shape	说明
<code>target_tokens</code>	<code>(N,)</code>	目标 token ID（来自采样器 <code>q</code> ）
<code>logprobs</code>	<code>(N,)</code>	这些 token 的 <code>sampling_logprobs</code>
<code>advantages</code>	<code>(N,)</code>	RL 中的优势值

输出张量

Key	Shape	说明
logprobs	(N,)	这些 token 的 target_logprobs

诊断信息

loss:sum (标量) : DRO 损失之和

自定义 beta

```
fwd_bwd_future = await training_client.forward_backward_async(  
    data=data,  
    loss_fn="dro",  
    loss_fn_config={"beta": 0.05}  
)  
fwd_bwd_result = await fwd_bwd_future.result_async()
```

CISPO（剪裁重要性采样策略优化）

最近更新时间：2026-10-19 21:18:02

CISPO（Clipped Importance Sampling Policy Optimization，剪裁重要性采样策略优化）是一种在大语言模型 RL/Alignment（对齐与强化学习）阶段使用的策略梯度（Policy Gradient）损失函数算法（[Chen et al., 2024](#)；[Khatri et al., 2024](#)）。CISPO 使用裁剪后重要性比率作为系数的策略梯度，与直接裁剪目标函数的 PPO 不同，CISPO 裁剪的是比率本身，并将其从计算图中分离，再用它对对数概率加权。

CISPO 的目标函数为：

$$\mathcal{L}_{\text{CISPO}}(\theta) = \mathbb{E}_{x \sim q} \left[\text{sg} \left(\text{clip} \left(\frac{p_{\theta}(x)}{q(x)}, 1 - \epsilon_{\text{low}}, 1 + \epsilon_{\text{high}} \right) \right) \cdot \log p_{\theta}(x) \cdot A(x) \right]$$

与 PPO 的关键区别：裁剪掩码是分离的（`sg` = 停止梯度），因此它只作为一个纯粹的加权系数，而不会影响梯度流动。

实现

```
# Compute probability ratio
prob_ratio = torch.exp(target_logprobs - sampling_logprobs)
# Apply clipping
clipped_ratio = torch.clamp(prob_ratio, clip_low_threshold,
                             clip_high_threshold)
# Compute CISPO objective (detach the clipped ratio - this is the key
                             difference from PPO)
cispo_objective = clipped_ratio.detach() * target_logprobs * advantages
# CISPO loss is negative of objective
loss = -cispo_objective.sum()
```

输入张量

Key	Shape	说明
<code>target_tokens</code>	<code>(N,)</code>	目标 token ID（来自采样器 <code>q</code> ）
<code>logprobs</code>	<code>(N,)</code>	这些 token 的 <code>sampling_logprobs</code>
<code>advantages</code>	<code>(N,)</code>	RL 中的优势值

输出张量

Key	Shape	说明
<code>logprobs</code>	<code>(N,)</code>	这些 token 的 <code>target_logprobs</code>

诊断信息

`loss:sum` (标量) : CISPO 损失之和

裁剪阈值的选择

由于裁剪后的比率是对 $\log p_\theta$ 的一个分离系数（而不是像 PPO 那样直接裁剪目标函数本身），CISPO 永远不会把某个 token 的梯度置零——它只是约束系数的量级。这正是阈值（尤其是下界）的取值之所以重要的原因。

默认是单侧裁剪。CISPO 默认关闭下界，只对上限作封顶（`clip_low_threshold=0.0`、`clip_high_threshold=4.0`），因此不传 `loss_fn_config` 就能得到这一行为。若要显式指定：

```
fwd_bwd_future = await training_client.forward_backward_async(
    data=data,
    loss_fn="cispo",
    loss_fn_config={"clip_low_threshold": 0.0, "clip_high_threshold":
4.0}
)
fwd_bwd_result = await fwd_bwd_future.result_async()
```

在任何场景下这都是一个安全的默认值。没有下界时，最坏情况不过是系数退回到普通的重要性采样权重，而这本身表现良好。相反，一个正的 `clip_low_threshold`（例如 `0.8`）会为那些比率已远低于 1 的 token 也作托底，这些是策略已经远离的 token，从而抹掉了重要性采样为陈旧 token 提供的衰减作用。

在 on-policy 下这很少产生影响，但在 off-policy（异步场景，采样器 q 落后于训练器 p_θ ）下，它可能形成正反馈循环：采样器与训练器之间的 KL 增大，更多 token 落到区间之外，偏差增大，KL 进一步增大。因此，去掉下界几乎没有代价，在 off-policy 下却有实实在在的收益。

这与引入并扩展 CISPO 的两篇论文都一致。MiniMax-M1（[Chen et al., 2024](#)）报告称：“我们没有对 IS 权重施加下界，而是把 $\epsilon_{\text{low}}^{\text{IS}}$ 设为一个很大的值；我们只调了 $\epsilon_{\text{high}}^{\text{IS}}$ 。”ScaleRL（[Khatri et al., 2024](#)）使用 `clip(r, 0,  $\epsilon_{\text{max}}$ )`，并发现 CISPO 对上限基本不敏感， ϵ_{max} 取 4、5、8 时表现相同。因此在该范围内取一个宽松的上限是安全的默认值。

Custom Loss Functions（自定义损失函数）

最近更新时间：2026-10-19 21:18:02

对于内置损失函数无法覆盖的场景，TCRL 提供了 `forward_backward_custom` 和 `forward_backward_custom_async`，用于计算更通用的一类损失函数（代价是会带来一些额外延迟）。

基本用法

自定义损失函数接收数据和模型的 logprobs，并返回一个标量损失以及可选的指标：

```
def logprob_squared_loss(data: list[Datum], logprobs:
list[torch.Tensor]) -> tuple[torch.Tensor, dict[str, float]]:
    loss = (logprobs ** 2).sum()
    return loss, {"logprob_squared_loss": loss.item()}
```

```
future = training_client.forward_backward_custom(data,
logprob_squared_loss)
result = future.result()
print(f"Loss: {result.loss}, Metrics: {result.metrics}")
```

多序列损失

您也可以定义一次作用于多个序列的损失函数：

```
def variance_loss(data: list[Datum], logprobs: list[torch.Tensor]) ->
tuple[torch.Tensor, dict[str, float]]:
    flat_logprobs = torch.cat(logprobs)
    variance = torch.var(flat_logprobs)
    return variance, {"variance_loss": variance.item()}
```

更实用的场景是在成对比较数据上计算 Bradley-Terry 损失（参见论文 [Learning to Summarize](#)），或 [Direct Preference Optimization](#)（直接偏好优化，DPO）—— 详见 [DPO 与偏好教程](#)。

多目标损失

您可以将 `forward_backward_custom` 与多目标 `target_tokens` 配合使用，在特定序列位置上针对一个小模型候选集定义损失。例如，掩码交叉熵损失会在四个答案 token 上（而不是整个词表上）重新归一化，从而用于训

练多项选择题：

```
import torch
import tink

messages = [{"role": "user", "content": (
    "What is the capital of France?\n"
    "A) London\n"
    "B) Paris\n"
    "C) Berlin\n"
    "D) Madrid\n"
    "Answer Letter:"
)}]

prompt_tokens = tokenizer.apply_chat_template(messages,
add_generation_prompt=True)

# Extract token IDs for the four answer letters.
a_token = tokenizer.encode(" A", add_special_tokens=False)[0]
b_token = tokenizer.encode(" B", add_special_tokens=False)[0]
c_token = tokenizer.encode(" C", add_special_tokens=False)[0]
d_token = tokenizer.encode(" D", add_special_tokens=False)[0]

target_tokens = torch.zeros(len(prompt_tokens), 4, dtype=torch.long)
target_tokens[-1] = torch.tensor([a_token, b_token, c_token, d_token])

datum = tink.Datum(
    model_input=tink.ModelInput.from_ints(prompt_tokens),
    loss_fn_inputs={"target_tokens": target_tokens},
)

CORRECT_ANSWER_IDX = 1 # B) Paris

def masked_ce_loss(data, logprobs_list):
    # logprobs_list[i] has shape [N, K=4] for each datum
    # Renormalize over just the four answer tokens instead of the full
    vocabulary.
    answer_logprobs = logprobs_list[0][-1] # shape [K=4]
    answer_logprobs = answer_logprobs - torch.logsumexp(answer_logprobs,
dim=-1)
    loss = -answer_logprobs[CORRECT_ANSWER_IDX]
```

```
return loss, {"masked_ce": loss.item()}

future = training_client.forward_backward_custom([datum],
masked_ce_loss)
result = future.result()
```

① 说明:

如果您正在使用的自定义损失函数具有通用价值，请告诉我们（联系 tcrl@tencent.com），我们会将其加入内置损失函数列表。这些方法的 [async](#) 版本请参阅 [Async Patterns](#) 教程。

forward_backward_custom 的工作原理

使用 `forward_backward_custom` 并不需要阅读本节，但其中的关键点是：TCRL 不会序列化（pickle）您的函数，也不会将其发送到服务端。相反，它会把梯度计算拆解为一次 `forward` 调用，加上一次针对加权交叉熵损失的 `forward_backward` 调用，从而计算出完全正确的梯度。

从数学上看，从完整的非线性损失出发：

$$\text{loss} = f(\log p_{\theta}(x))$$

TCRL 会构造一个关于 logprobs 呈线性、但梯度完全相同的代理损失：

$$L_{\text{surrogate}} = \sum_i \frac{\partial f}{\partial \log p_{\theta}(x_i)} \cdot \log p_{\theta}(x_i)$$

CLI 参考

CLI 参考说明

最近更新时间：2026-10-19 21:18:03

`tinker` 命令行工具用于管理训练任务、检查点和模型权重。

tinker [*OPTIONS*] *COMMAND*

全局选项（所有命令可用）：

- `-f, --format [table|json]` — 输出格式（默认：table）
- `-h, --help` — 显示帮助信息并退出

命令：

- `tinker run` — 列出和查看训练任务
- `tinker checkpoint` — 列出、下载、发布和管理检查点

tinker version

显示已安装的 Tinker SDK 版本。

```
$ tinker version
tinker 0.16.1
```

tinker run

最近更新时间：2026-10-19 21:18:03

tinker run：管理训练任务

tinker run lis：列出训练任务，显示状态、模型信息和 LoRA 秩

选项：

- `--limit INTEGER` — 最大获取任务数量（默认：20，使用 0 显示全部）
- `-c, --columns TEXT` — 显示的列（逗号分隔）。可用值：`id`、`model`、`owner`、`lora`、`updated`、`status`、`checkpoint`、`checkpoint_time`。默认：`id, model, lora, updated, status`
- `-f, --format [table|json]` — 输出格式（默认：table）

```
$ tinker run list --limit 3
3 training runs (1377 more not shown)
```

Run ID	Base Model	LoRA	Last Upd
Stat			
cda26fb4-...:train:0	Qwen3-8B	Rank 8	3h ago
OK			
ea779368-...:train:0	Llama-3.1...	Rank32	4h ago
OK			
251b8fc3-...:train:0	Nemotron...	Rank32	9h ago
OK			

tinker run info *RUN_ID*：显示指定训练任务的详细信息。

```
$ tinker run info cda26fb4-baf6-5b13-9011-e908ccaf30b9:train:0
```

tinker checkpoint

最近更新时间：2026-10-19 21:18:02

tinker checkpoint: 管理检查点，列出、下载、发布、删除以及上传到 HuggingFace。

所有检查点命令接受格式为 `tinker://run-id/sampler_weights/final` 的 `CHECKPOINT_PATH` 参数。
所有命令支持 `-f, --format [table|json]` 输出格式选项。

tinker checkpoint list: 列出所有训练任务的检查点，或指定训练任务的检查点。

选项：

- `-f, --format [table|json]` 一 输出格式（默认：table）
- `--run-id TEXT` 一 列出指定训练任务的检查点
- `--limit INTEGER` 一 最大显示检查点数量（默认：20，使用 0 显示全部）

```
$ tinker checkpoint list --limit 3
3 checkpoints (1789 more not shown)
```

Checkpoint ID	Type	Size	Pub	Crea	Expir
sampler_weights/lifecycle_test	sampler	88.2 MB	No	3h	Never
sampler_weights/final	sampler	102 MB	No	4h	Never
weights/final	train	306 MB	No	4h	Never

tinker checkpoint info *CHECKPOINT_PATH*: 显示指定检查点的详细信息。

```
$ tinker checkpoint info tinker://run-id/sampler_weights/final
```

选项：

- `-f, --format [table|json]` 一 输出格式（默认：table）

tinker checkpoint download *CHECKPOINT_PATH*: 下载并解压检查点到本地目录。

选项：

- `-f, --format [table|json]` 一 输出格式（默认：table）
- `-o, --output PATH` 一 解压检查点的父目录（默认：当前目录）
- `--force` 一 如果目录已存在则覆盖

下载到当前目录

```
$ tinker checkpoint download tinker://run-id/sampler_weights/final
```

下载到指定目录

```
$ tinker checkpoint download tinker://run-id/sampler_weights/final -o  
./models/
```

强制覆盖

```
$ tinker checkpoint download tinker://run-id/sampler_weights/final --  
force
```

tinker checkpoint publish *CHECKPOINT_PATH*: 将检查点设置为公开访问。其他用户可以通过路径加载。

```
$ tinker checkpoint publish tinker://run-id/sampler_weights/final
```

选项：

- `-f, --format [table|json]` 一 输出格式（默认：table）

tinker checkpoint unpublish *CHECKPOINT_PATH*: 将已发布的检查点恢复为私有。

```
$ tinker checkpoint unpublish tinker://run-id/sampler_weights/final
```

选项：

- `-f, --format [table|json]` 一 输出格式（默认：table）

tinker checkpoint set-ttl *CHECKPOINT_PATH*: 设置或移除检查点的存活时间（TTL）。TTL 到期后，检查点将被自动删除。

选项：

- `-f, --format [table|json]` — 输出格式（默认：table）
- `--ttl SECONDS` — TTL 秒数（必须为正整数，最小值 3600）
- `--remove` — 移除过期设置，使检查点永久保留

```
# 7 天后过期
$ tinker checkpoint set-ttl tinker://run-id/sampler_weights/step-100 --
ttl 604800

# 永久保留（移除过期设置）
$ tinker checkpoint set-ttl tinker://run-id/sampler_weights/step-100 --
remove
```

tinker checkpoint delete *CHECKPOINT_PATH* [...]: 永久删除一个或多个检查点。

```
$ tinker checkpoint delete tinker://run-id/sampler_weights/step-100

# 删除多个
$ tinker checkpoint delete \
    tinker://run-id/sampler_weights/step-100 \
    tinker://run-id/sampler_weights/step-200
```

选项：

- `-f, --format [table|json]` — 输出格式（默认：table）

tinker checkpoint push-hf *CHECKPOINT_PATH*: 将检查点上传到 HuggingFace Hub 作为 PEFT LoRA 适配器。

选项：

- `-f, --format [table|json]` — 输出格式（默认：table）
- `-r, --repo TEXT` — HuggingFace 仓库 ID（例如 `username/my-lora-adapter`）。如果省略，则从训练任务推导。
- `--public` — 创建或上传到公开仓库（默认：私有）
- `--revision TEXT` — 目标分支/版本
- `--commit-message TEXT` — 提交信息
- `--create-pr` — 创建 Pull Request 而非直接推送到 main 分支
- `--allow-pattern TEXT` — 仅上传匹配此模式的文件（可重复使用）

- `--ignore-pattern TEXT` — 跳过匹配此模式的文件（可重复使用）
- `--no-model-card` — 不创建 README.md 模型卡片

```
$ tinker checkpoint push-hf tinker://run-id/sampler_weights/final --repo  
my-org/my-model  
$ tinker checkpoint push-hf tinker://run-id/sampler_weights/final --repo  
my-org/my-model --public --create-pr
```

COOKBOOK

快速开始

最近更新时间：2026-10-19 21:38:00

Cookbook 在 **TCRL SDK** 之上提供了更高层的抽象——SDK 提供原始操作（`forward_backward`、`optim_step`、`sample`），Cookbook 提供可配置的训练管线，自动处理流水线、checkpoint、评测和日志。

- **SFT**: 通过 `train.Config` 一行配置即可运行完整的监督微调循环
- **RL**: 通过 `Env` + `EnvGroupBuilder` + `RLDataset` 组合出完整的 RL 训练流程
- **DPO**: 通过 `Comparison` / `LabeledComparison` 进行偏好学习

Cookbook 安装

```
uv pip install tinkers==0.12.0
```

① 注意：

目前 TCRL 开白体验中，可联系 tcrl@tencent.com 申请开白后，登录 [腾讯云弹性 MapReduce \(EMR\) 控制台](#) 获取 API Key，进行模型训练、实验管理和用量监控。

```
export TINKER_API_KEY="your-api-key-here"
export TINKER_BASE_URL="http://api.tencenttcrl.com/"
```

可选扩展：`[math-rl]`、`[modal]`、`[wandb]`、`[cloud]`、`[inspect]`、`[all]`。

Cookbook 认证

```
tinkers auth login
```

或者在 [Tinker Console](#) 中创建一个 API key，并将其设置为 `TINKER_API_KEY` 环境变量。

如需使用 nightly 构建版本：

```
uv pip install 'tinkers-cookbook @ git+https://github.com/thinking-machines-lab/tinkers-cookbook.git@nightly'
```

如需在本地开发，或按自己的需求定制方案：

```
git clone https://github.com/thinking-machines-lab/tinker-cookbook.git
cd tinker-cookbook
uv pip install -e .
```

Cookbook 在 **Tinker SDK** 之上提供了更高层的抽象。SDK 提供的是底层操作（`forward_backward`、`optim_step`、`sample`），而 Cookbook 提供的是可配置的训练流水线，会自动处理流水线并行、检查点保存、评测和日志记录。

使用 Cookbook 做 SFT

使用 SDK 时，SFT 循环需要在循环中手动依次调用 `forward_backward` → `optim_step` → `save_state`。Cookbook 把这些封装进一个 `train.Config`：

DatasetBuilder 定义数据 → `train.Config` 模型、学习率、rank → `train.main()` 流水线化循环 → 检查点 + 指标

1. 定义数据：实现一个返回按批组织 `Datum` 对象的 `SupervisedDatasetBuilder`
2. 配置训练：设置模型、学习率、LoRA rank、检查点保存计划
3. 运行：`train.main(config)` 负责整个循环

```
from tinker_cookbook.supervised import train
from tinker_cookbook.supervised.types import ChatDatasetBuilder

config = train.Config(
    log_path="~/logs/my-sft-run",
    model_name="Qwen/Qwen3-8B",
    dataset_builder=my_dataset_builder,
    learning_rate=1e-4,
    lora_rank=32,
    save_every=20,
    eval_every=10,
)

asyncio.run(train.main(config))
```

在底层，`train.main` 会创建 `TrainingClient`，将 `forward_backward` + `optim_step` 流水线化以提升吞吐，运行评测器并保存检查点。完整的组件图参见 [SL 架构概览](#)。

使用 Cookbook 做 RL

RL 比 SFT 更复杂，因为它涉及一个 **rollout 循环**：从模型中采样，用奖励函数打分，然后在打分后的轨迹上训练。Cookbook 把这些抽象成可组合的类型：

RLDatasetget_batch() → EnvGroupBuildermake_envs() → Envrollout → 奖励 + 优势 → forward_backward 训练 → 重复

1. **定义环境**：实现 `Env`（问答任务可用 `ProblemEnv`）来产生奖励
2. **对环境分组**：`EnvGroupBuilder` 按批创建环境（GRPO 会在组内对奖励做中心化）
3. **定义数据集**：`RLDataset` 产生按批组织的 `EnvGroupBuilder`
4. **运行**：训练循环负责 rollout、优势计算和权重更新

最简单的起点是 `ProblemEnv`，一个单轮问答环境：

```
from tinker_cookbook.rl.problem_env import ProblemEnv

class MathEnv(ProblemEnv):
    def get_question(self) -> str:
        return "What is 2 + 3?"

    def check_answer(self, response: str) -> float:
        return 1.0 if "5" in response else 0.0
```

完整的组件图参见 [RL 架构概览](#)。

偏好学习（DPO / RLHF）

偏好学习用成对比较而非标量奖励来训练模型：

1. **构造比较样本**：`Comparison` 对象包含同一提示词下的两个补全（A 和 B）
2. **标注偏好**：`LabeledComparison` 增加人类偏好标注（A、B 或平局）
3. **用 DPO 训练**：直接基于偏好数据优化策略

```
from tinker_cookbook.preference.types import Comparison,
LabeledComparison

comparison = Comparison(messages=messages, completion_a=resp_a,
completion_b=resp_b)
labeled = LabeledComparison(**comparison.__dict__, label=1) # B
preferred
```

完整讲解参见 [DPO 指南](#) 和 [RLHF 示例](#)。

后续步骤

- [SL 架构](#)：监督学习的完整组件图
- [RL 架构](#)：强化学习的完整组件图
- [教程](#)：交互式 Notebook

Supervised Learning（监督学习）

最近更新时间：2026-10-19 21:18:02

Cookbook 提供了一个流水线式的 Supervised Learning（SFT，监督学习）训练循环，数据集、评测和检查点保存都可配置。

架构

```
train.Config
├─ model_name           which model to fine-tune
├─ dataset_builder      SupervisedDatasetBuilder → SupervisedDataset
├─ learning_rate        optimizer config (+ lr_schedule, adam_*)
├─ evaluator_builders   what to measure during training
├─ save_every / eval_every  checkpoint and eval cadence
└─ lora_rank            LoRA adapter size

SupervisedDatasetBuilder
└─ __call__() → (train_dataset, eval_dataset)
    |
    ▼
SupervisedDataset
├─ get_batch(index) → list[tinker.Datum]
├─ __len__() → number of batches
└─ set_epoch(seed) → shuffle for next epoch
```

核心组件

tinker_cookbook.supervised.train.Config

核心配置。定义模型、数据、超参数和评测。

```
from tinker_cookbook.supervised import train

config = train.Config(
    log_path="~/logs/my-sft-run",
    model_name="Qwen/Qwen3-8B",
    dataset_builder=my_dataset_builder,
    learning_rate=1e-4,
    lora_rank=32,
```

```
num_epochs=1,
save_every=20,
eval_every=10,
)
asyncio.run(train.main(config))
```

SupervisedDataset / SupervisedDatasetBuilder

`SupervisedDatasetBuilder` 是一个用于构造 `SupervisedDataset` 的配置。数据集返回成批的 `tinker.Datum` 对象。

```
from tinker_cookbook.supervised.types import SupervisedDatasetBuilder,
SupervisedDataset

class MyDatasetBuilder(SupervisedDatasetBuilder):
    def __call__(self) -> tuple[SupervisedDataset, SupervisedDataset |
None]:
        return train_data, eval_data # eval_data is optional
```

内置构建器:

构建器	用途
<code>SupervisedDatasetFromHFDataset</code>	用 <code>map_fn</code> 或 <code>flatmap_fn</code> 包装一个 HuggingFace 数据集
<code>StreamingSupervisedDatasetFromHFDataset</code>	从 HuggingFace 流式加载（内存占用更低）
<code>FromConversationFileBuilder</code>	从聊天对话的 JSONL 文件加载
<code>ChatDatasetBuilder</code>	聊天数据的基类——额外提供 <code>tokenizer</code> + <code>renderer</code>

ChatDatasetBuilder

用于聊天格式的数据。通过 `ChatDatasetBuilderCommonConfig` 提供 `tokenizer` 和 `renderer` 属性:

```
from tinker_cookbook.supervised.types import ChatDatasetBuilder,
ChatDatasetBuilderCommonConfig

class MyChats(ChatDatasetBuilder):
```

```
common_config = ChatDatasetBuilderCommonConfig(  
    model_name_for_tokenizer="Qwen/Qwen3-8B",  
    renderer_name="qwen3",  
    batch_size=8,  
    max_length=4096,  
)
```

训练循环

`train.main(config)` 负责整个流水线：

1. 创建 `ServiceClient` → `TrainingClient`
2. 从 `config.dataset_builder` 构建数据集
3. 对每个 epoch 和每个批次：
 - 用该批次执行 `forward_backward`
 - 执行 `optim_step`
 - 每隔 `eval_every` 运行一次评测器
 - 每隔 `save_every` 保存一次检查点
4. 保存最终检查点

这个循环会对请求做流水线处理以提升吞吐，详情请参见 [时钟周期与流水线](#)。

下一步

- [sl_loop.py](#)：不带抽象层的最小训练循环
- [SL 超参数](#)：如何选择学习率、rank 和批大小
- [教程：你的第一个 SFT](#)：交互式演练

Reinforcement Learning (强化学习)

概述

最近更新时间：2026-10-19 21:18:03

Reinforcement Learning (强化学习) 即从试错中学习。您不再提供输入-输出对，而是提供提示词和奖励函数，由算法自行发现好的输出。

架构

```
RLDataset
├─ get_batch(index) → list[EnvGroupBuilder]
├─ __len__() → number of batches

EnvGroupBuilder
├─ make_envs() → list[Env]           create environments for this
group
├─ compute_group_rewards(trajs, envs) final reward using the whole
group
├─ cleanup()                       release resources (sandboxes,
etc.)
├─ logging_tags()                   tags for metric aggregation

Env (stateful, one per episode)
├─ initial_observation() → (Observation, StopCondition)
├─ step(action, extra) → StepResult
    └─ reward: float
    └─ episode_done: bool
    └─ next_observation
    └─ metrics / logs

Trajectory = list[Transition(ob, ac, reward, episode_done)]
TrajectoryGroup = list[Trajectory] + final_rewards + metrics
```

核心组件

Env

面向单个 episode 的有状态环境。用完即弃。

```
from tinkercookbook.rl.types import Env, StepResult

class MyEnv(Env):
    async def initial_observation(self):
        prompt = renderer.build_generation_prompt([{"role": "user",
"content": self.question}])
        return prompt, renderer.get_stop_sequences()

    async def step(self, action, *, extra=None):
        response = renderer.parse_response(action)
        reward = 1.0 if check_correct(response) else 0.0
        return StepResult(reward=reward, episode_done=True, ...)
```

ProblemEnv

面向单轮问答任务的便捷基类：

```
from tinkercookbook.rl.problem_env import ProblemEnv

class MathEnv(ProblemEnv):
    def get_question(self) -> str:
        return "What is 2 + 3?"

    def check_answer(self, response: str) -> float:
        return 1.0 if "5" in response else 0.0
```

EnvGroupBuilder

构建一组环境。分组可以支持奖励中心化（GRPO）与多智能体配置。

```
from tinkercookbook.rl.types import EnvGroupBuilder

class MyGroupBuilder(EnvGroupBuilder):
    async def make_envs(self) -> list[Env]:
        return [MyEnv(q) for q in self.questions]

    # Optional: compute group-level rewards (default: 0)
    async def compute_group_rewards(self, trajs, envs):
```

```
return [(0.0, {}), for _ in trajts]
```

内置： `ProblemGroupBuilder` 通过工厂函数构建由 `ProblemEnv` 实例组成的组。

RLDataset / RLDatasetBuilder

为训练循环产出成批的 `EnvGroupBuilder` 。

```
from tinker_cookbook.rl.types import RLDataset, RLDatasetBuilder
```

数据处理

rollout 之后，把轨迹转换成训练数据：

```
from tinker_cookbook.rl.data_processing import compute_advantages,
trajectory_to_data, assemble_training_data

# Compute GRPO-style advantages
advantages = compute_advantages(trajectory_groups, loss_fn="cispo")

# Convert to Datum objects for forward_backward
training_data = assemble_training_data(trajectory_groups, advantages,
...)
```

Rollout 策略

控制 rollout 期间的错误处理：

策略	行为
<code>FailFast</code>	遇到第一个错误就停止
<code>RetryOnFailure</code>	重试失败的轨迹，部分成功也继续

训练循环

RL 训练循环：

1. 从 `RLDataset` 采样一批 `EnvGroupBuilder`
2. **Rollout:** 对每个组创建环境、采样补全、收集奖励
3. **计算优势:** 在组内对奖励做归一化 (GRPO)
4. **训练更新:** 用 rollout 数据执行 `forward_backward` + `optim_step`

5. 评测并保存检查点

后续步骤

- [RL 训练循环](#)：最小化的 GRPO 循环实现
- [RL 环境](#)：构建自定义环境
- [RL 超参数](#)：KL 惩罚、优势、奖励 shaping
- [教程：第一个 RL](#)：交互式演练

RL Training Outputs（强化学习训练输出）

最近更新时间：2026-10-19 21:18:03

强化学习训练输出

每次 RL 训练运行都会把文件写入 `log_path`。所有文件 I/O 都经过 [Storage 协议](#)，因此 `log_path` 可以是本地路径，也可以是云端 URI（例如 `gs://bucket/run`）。

本页介绍每个文件以及如何从中提取数据。

写入 `log_path` 的文件

文件	格式	内容
<code>metrics.jsonl</code>	JSONL	每次训练迭代一个 JSON 对象，包含全部标量指标
<code>config.json</code>	JSON	序列化后的训练配置（超参数、模型、数据集等）
<code>checkpoints.jsonl</code>	JSONL	检查点元数据（路径、用于续训的循环状态）
<code>train_iteration_NNNNNN.html</code>	HTML	训练 rollout 的可读 logtree 报告
<code>train_iteration_NNNNNN_logtree.json</code>	JSON	同一份 logtree trace 的机器可读导出
<code>train_iteration_NNNNNN_rollout_summaries.jsonl</code>	JSONL	每条轨迹一个 JSON 对象，含奖励、指标与步级数据
<code>eval_<name>_iteration_NNNNNN.html</code>	HTML	评测 rollout 的 logtree 报告
<code>eval_<name>_iteration_NNNNNN_logtree.json</code>	JSON	评测 logtree trace 的机器可读导出
<code>eval_<name>_iteration_NNNNNN_rollout_summaries.jsonl</code>	JSONL	逐条轨迹的评测数据（用于 <code>RLTestSetEvaluator</code> ）
<code>code.diff</code>	text	训练开始时的 git diff

`<name>` 是评测器名称（已针对文件名做净化处理）；迭代编号补零到 6 位。

`metrics.jsonl`

每一行都是一个以指标名为键的 JSON 对象。常见键（随环境与配置而异）：

- `progress/batch`、`progress/done_frac` — 迭代序号与完成比例
- `env/all/reward/total` — 所有轨迹的总奖励均值
- `env/all/<metric>` — 环境发出的指标（例如 `format_parse`、`correct`）
- `ac_tokens_per_turn` — 每回合生成的 token 数均值
- `entropy` — 逐 token 熵
- `kl_sample_train_v1`、`kl_sample_train_v2` — KL 散度估计量
- `optim/lr` — 学习率
- `time/...` — 各阶段的墙钟耗时

```
df = pd.read_json("path/to/metrics.jsonl", lines=True)
df.plot(x="progress/batch", y="env/all/reward/total")
```

`*_rollout_summaries.jsonl`

每条轨迹一行。最适合做聚合分析（奖励分布、逐步指标）。

```
with open("train_iteration_000010_rollout_summaries.jsonl") as f:
    trajectories = [json.loads(line) for line in f]

# Each trajectory has:
# - metadata: schema_version, split, iteration, group_idx, traj_idx,
#   tags, sampling_client_step
# - episode totals: total_reward, final_reward, trajectory_metrics,
#   final_ob_len
# - steps: list of {step_idx, ob_len, ac_len, reward, episode_done,
#   metrics, logs}
```

`*_logtree.json`

logtree JSON 包含完整的 rollout 记录：提示词、模型回复、评分细节与奖励拆解。当你需要 rollout 的实际文本内容时使用它。

顶层字段：`title`、`started_at`、`path`、`root`。`root` 是一棵节点树，每个节点含 `tag`、`attrs` 和 `children`（可以是文本字符串，也可以是嵌套节点）。

部分节点带有 `data` 字段，内含结构化内容。用 `data` 可以提取诸如对话消息这类带类型的数据：

```
def find_conversations(node):
    """Recursively find all nodes with conversation data."""
    results = []
    if isinstance(node, dict):
        if node.get("data", {}).get("type") == "conversation":
            results.append(node["data"])
        for child in node.get("children", []):
            if isinstance(child, dict):
                results.extend(find_conversations(child))
    return results

with open("eval_test_iteration_000020_logtree.json") as f:
    trace = json.load(f)

for conv in find_conversations(trace["root"]):
    for msg in conv["messages"]:
        print(f"{msg['role']}: {msg['content'][:100] if
isinstance(msg['content'], str) else '...'}")
```

⚠ 注意:

`num_groups_to_log`（默认：4）控制有多少个轨迹组会获得详细的环境级日志。超出该上限的组在 `logtree` 中没有 rollout 内容——只有 `Trajectory Details` 部分（回合级统计）始终存在。

config.json

序列化后的 `chz` 配置，记录了全部训练超参数。便于复现某次运行或跨实验比较配置。

checkpoints.jsonl

每行记录一个已保存的检查点，包含其路径以及保存时的循环状态。续训逻辑用它来从上次中断处继续。

Evaluation (评测)

概述

最近更新时间：2026-10-19 21:18:03

评测：对任意 Tinker 模型或检查点运行标准化基准测试，该框架负责并发控制、评分、轨迹存储和结果聚合。

快速开始

```
from tinkercookbook.eval.benchmarks import run_benchmark,
run_benchmarks, BenchmarkConfig

# Single benchmark
result = await run_benchmark("gsm8k", sampling_client, renderer)
print(f"GSM8K: {result.score:.1%}")

# Multiple benchmarks (parallel by default)
results = await run_benchmarks(
    ["gsm8k", "mmlu_pro", "ifeval"],
    sampling_client, renderer,
)
for name, r in results.items():
    print(f"{name}: {r.score:.1%}")
```

架构

基准测试复用了 RL 的 `Env` 协议——同一个环境既可以驱动评测，也可以驱动训练。

```
BenchmarkBuilder
├─ make_envs(renderer, config) → list[Env]    one Env per example
├─ aggregate(rewards, metrics) → BenchmarkResult
├─ name, multi_turn, requires_sandbox, ...

Runner (run_benchmark / run_benchmarks)
├─ Creates envs from BenchmarkBuilder
├─ Runs rollouts concurrently (semaphore)
├─ Grades responses via Env.step()
├─ Saves trajectories as JSONL
├─ Aggregates into BenchmarkResult
```

```

BenchmarkResult
├── score                raw accuracy
├── score_completed      accuracy excluding truncated/errored examples
├── num_examples, num_correct, num_errors, num_truncated
├── pass_at_k            {1: 0.45, 5: 0.72} (when num_samples > 1)
└── time_seconds
    
```

可用的基准测试

程序化评分（单轮）

基准测试	样本数	说明
<code>gsm8k</code>	1,319	小学数学题
<code>math500</code>	500	竞赛数学（MATH）
<code>aime</code>	30	AIME 数学竞赛
<code>aime_2025</code>	30	AIME 2025
<code>aime_2026</code>	30	AIME 2026
<code>hmmt_feb_2025</code>	30	HMMT 2025 年 2 月
<code>hmmt_nov_2025</code>	30	HMMT 2025 年 11 月
<code>mmlu_pro</code>	12,032	多领域知识
<code>mmlu_redux</code>	~3,000	经过筛选的 MMLU 子集（30 个学科，已过滤错误样本）
<code>gpqa</code>	198	研究生级别科学问答（Diamond 子集）
<code>supergpqa</code>	26,529	扩展版研究生级别问答
<code>ceval</code>	~1,300	中文考试评测（52 个学科，验证集）
<code>ifeval</code>	541	指令跟随
<code>ifbench</code>	300	指令跟随

代码执行

基准测试	类型	说明
<code>mbpp</code>	单轮	Python 代码生成 （已清洗，427）
<code>livecodebench</code>	单轮	竞赛编程
<code>swe_bench</code>	多轮	GitHub issue 修复 （500，agent）
<code>terminal_bench</code>	多轮	终端交互 （agent）
<code>tau2_bench</code>	多轮	客服 agent

LLM 作为裁判

基准测试	说明
<code>arena_hard</code>	ArenaHard （500，需要 <code>judge_sampling_client</code> ）

其他

基准测试	说明
<code>longbench</code>	长上下文评测
<code>bfcl</code>	函数调用 / 工具使用

在训练过程中使用基准测试

`BenchmarkEvaluator` 把任意基准测试包装成训练循环中的评测器。它会在每个评测步运行一小批样本，并将指标记录到 TensorBoard / W&B：

```
from tinker_cookbook.eval.benchmark_evaluator import BenchmarkEvaluator

config = train.Config(
    # ... training config ...
    evaluator_builders=[
        lambda: BenchmarkEvaluator("gsm8k", renderer, max_examples=100),
        lambda: BenchmarkEvaluator("ifeval", renderer, max_examples=50),
    ],
    eval_every=50,
)
```

记录的指标：`eval/gsm8k/score`、`eval/gsm8k/num_correct` 等。

下一步

- [基准测试指南](#)：配置、存储、pass@k、新增基准测试
- [定制基准测试](#)：系统提示词、答案解析、裁判模型、共享 Env 设计
- [评测教程](#)：评测器模式的交互式讲解

Benchmarks Guide（基准测试指南）

最近更新时间：2026-10-19 21:38:00

配置

`BenchmarkConfig` 控制基准测试的运行方式：

```
from tinkering.eval.benchmarks import run_benchmark,
BenchmarkConfig

config = BenchmarkConfig(
    max_examples=200,          # Cap number of examples (None = all)
    concurrency=64,           # Parallel rollouts (single-turn)
    agent_concurrency=8,      # Parallel rollouts (multi-turn)
    timeout_seconds=300,      # Per-example timeout
    max_tokens=32768,         # Max generation tokens
    temperature=0.6,
    save_dir="evals/run_01",  # Save trajectories and results
)

result = await run_benchmark("gsm8k", sampling_client, renderer, config)
```

按模型的默认值

`BenchmarkConfig.for_model()` 会根据内置表设置 `max_tokens`、`timeout_seconds` 和 `context_window`：

```
config = BenchmarkConfig.for_model(
    "Qwen/Qwen3.6-35B-A3B",
    save_dir="evals/qwen3.6",
)

# max_tokens=65536, context_window=65536, timeout_seconds=1800
```

多轮基准测试

Agent 类基准测试（`swe_bench`、`terminal_bench`、`tau2_bench`）需要上下文管理：

```
config = BenchmarkConfig(
    timeout_seconds=1800,
```

```
max_trajectory_tokens=60000,      # Total tokens across all turns
max_generation_tokens=8192,       # Tokens per generation step
agent_concurrency=4,
)
```

沙箱基准测试

需要执行代码的基准测试（`mbpp`、`livecodebench`、`swe_bench`）依赖沙箱：

```
from tinkercookbook.sandbox.modal_sandbox import ModalSandbox

config = BenchmarkConfig(
    sandbox_factory=ModalSandbox.create,
)
```

裁判模型基准测试

`arena_hard` 这类基准测试需要额外的 LLM 裁判。配置细节与示例见 [裁判模型](#)。

保存与查看结果

设置 `save_dir` 以持久化轨迹和结果：

```
evals/run_01/
├── summary.json          # Combined scores (run_benchmarks
only)
├── gsm8k/
│   ├── result.json      # BenchmarkResult
│   └── trajectories.jsonl # One StoredTrajectory per line
└── ifeval/
    ├── result.json
    └── trajectories.jsonl
```

加载结果

```
from tinkercookbook.eval.benchmarks import (
    load_result,
    load_trajectories,
    load_summary,
    print_trajectory,
```

```
)

result = load_result("evals/run_01", "gsm8k")
print(f"Score: {result.score:.1%}, Completed:
{result.score_completed:.1%}")

# Filter trajectories
wrong = load_trajectories("evals/run_01", "gsm8k", incorrect_only=True)
errors = load_trajectories("evals/run_01", "gsm8k", errors_only=True)
print_trajectory(wrong[0])

# Combined summary across benchmarks
summary = load_summary("evals/run_01")
```

可续跑

使用相同的 `save_dir` 重新运行时会自动跳过已完成的样本。去重基于 `example_id`（内容哈希），因此即使数据集被打乱也能稳定工作。

理解分数

- `score` — `num_correct / num_examples`。被截断和出错的样本计为 0。
- `score_completed` — `num_correct / num_completed`。分母中排除被截断和出错的样本。

对于经常触及 `max_tokens` 的思考型模型，用 `score_completed` 与公开分数对比更合适。

```
result = await run_benchmark("gsm8k", client, renderer, config)
print(f"Raw: {result.score:.1%}") # 81.7%
print(f"Completed: {result.score_completed:.1%}") # 95.6%
print(f"{result.num_truncated} truncated, {result.num_errors} errors")
```

对比检查点

```
checkpoints = {
    "base": "tinker://run-abc/sampler_weights/step0",
    "step500": "tinker://run-abc/sampler_weights/step500",
    "final": "tinker://run-abc/sampler_weights/final",
}

for name, path in checkpoints.items():
    client = await sc.create_sampling_client_async(model_path=path)
```

```
results = await run_benchmarks(
    ["gsm8k", "ifeval"], client, renderer,
    BenchmarkConfig(save_dir=f"evals/{name}"),
)
for bench, r in results.items():
    print(f"{name}/{bench}: {r.score:.1%}")
```

用 EvalStore 追踪多次运行

EvalStore 通过云端兼容的存储管理跨检查点的评测运行：

```
from tinkercookbook.stores.eval_store import EvalStore

store = EvalStore("~/experiments/evals")

run_id = store.create_run(
    model_name="Qwen/Qwen3.6-35B-A3B",
    checkpoint_path="tinker://run-123/weights/step500",
    checkpoint_name="step500",
    benchmarks=["gsm8k", "ifeval"],
)

config = BenchmarkConfig(save_dir=store.run_dir(run_id))
await run_benchmarks(["gsm8k", "ifeval"], client, renderer, config)

metadata = store.finalize_run(run_id)
print(metadata.scores)  # {"gsm8k": 0.847, "ifeval": 0.923}
```

Pass@k 评测

设置 `num_samples > 1` 可计算无偏的 pass@k 估计（Codex 公式）：

```
config = BenchmarkConfig(num_samples=4, save_dir="evals/pass_at_k")
result = await run_benchmark("gsm8k", sampling_client, renderer, config)
print(result.pass_at_k)  # {1: 0.45, 2: 0.58, 4: 0.72}
```

自定义

关于系统提示词、自定义评分、答案解析和裁判模型配置，见 [自定义基准测试](#)。

新增基准测试

基准测试复用与 RL 训练相同的 `Env` 协议——单个 `MessageEnv` 即可同时驱动训练和评测。概念性说明见 [共享 Env 抽象](#)。

要新增基准测试，实现一个 `BenchmarkBuilder` 并注册：

```
from tinkercookbook.eval.benchmarks import BenchmarkBuilder,
BenchmarkConfig, register
from tinkercookbook.eval.benchmarks._common import
load_benchmark_dataset, make_example_id
from tinkercookbook.rl.message_env import MessageEnv,
MessageStepResult, EnvFromMessageEnv

class MyMessageEnv(MessageEnv):
    """Single-turn env for one example."""

    def __init__(self, question: str, expected: str, example_id: str):
        self.question = question
        self.expected = expected
        self._example_id = example_id

    @property
    def example_id(self) -> str:
        return self._example_id

    async def initial_observation(self) -> list[Message]:
        return [{"role": "user", "content": self.question}]

    async def step(self, message: Message) -> MessageStepResult:
        response = get_text_content(message).lower()
        correct = self.expected.lower() in response
        return MessageStepResult(
            reward=1.0 if correct else 0.0,
            episode_done=True,
            next_messages=[],
            metrics={"correct": float(correct)},
            logs={"expected": self.expected},
        )

class MyBenchmark(BenchmarkBuilder):
```

```
name = "my_benchmark"

recommended_system_prompt = "Answer concisely."

def make_envs(self, renderer, config):
    ds = load_benchmark_dataset("my/dataset", split="test")
    envs = []
    for row in ds:
        env = EnvFromMessageEnv(
            renderer=renderer,
            message_env=MyMessageEnv(
                question=row["question"],
                expected=row["answer"],
                example_id=make_example_id("my_benchmark",
row["question"]),
            ),
        )
        envs.append(env)
    return envs

register(MyBenchmark())
```

要点：

- 单轮基准测试用 `MessageEnv` ——它负责提示词构建与响应解析
- 通过 `make_example_id()` 设置 `example_id`，保证多次运行间可稳定续跑
- `logs` 存放每条轨迹的诊断数据（期望答案、抽取出的答案）；`metrics` 存放用于聚合的数值
- 多轮基准测试请直接实现 `Env`，并在 `builder` 上设置 `multi_turn = True`

延伸阅读

- [评测概览](#)：可用基准测试与架构
- [自定义基准测试](#)：系统提示词、评分、裁判模型、共享 Env 设计
- [评测教程](#)：交互式演练
- [RL 环境](#)：基准测试使用的 Env 协议

Customizing Benchmarks（自定义基准测试）

最近更新时间：2026-10-19 21:38:00

共享的 Env 抽象

基准测试复用与 RL 训练相同的 `Env` 协议。两者的联系是直接的——`BenchmarkBuilder.make_envs()` 返回的 `Env` 实例与 `RLDataset` 中使用的完全一致：

RL Training	Benchmark Evaluation
<code>RLDataset</code>	<code>BenchmarkBuilder</code>
└─ <code>EnvGroupBuilder</code>	└─ <code>make_envs()</code>
└─ <code>make_envs() → list[Env]</code>	└─ <code>list[Env]</code>
└─ <code>Env.step() → reward</code>	└─ <code>Env.step() → reward</code>

两条路径都使用 `EnvFromMessageEnv` 把消息级环境适配到 token 级的 `Env` 接口。同一个 `MessageEnv` 实现可以同时驱动训练和评测——只需在任一场景下用 `EnvFromMessageEnv` 包装它。rollout 基础设施也是共享的：`run_benchmark()` 内部会调用 RL 模块中的 `do_single_rollout()`。

关于实现并注册基于 `MessageEnv` 的基准测试的完整示例，参见 [添加新的基准测试](#)。

系统提示词

提示词如何解析

每个基准测试都定义了一个能提升得分的 `recommended_system_prompt`（例如，数学基准测试会指示模型使用 `\boxed{}`）。运行器按以下顺序解析提示词：

1. `BenchmarkConfig.system_prompt`：如果设置了，始终优先
2. `BenchmarkBuilder.recommended_system_prompt`：当没有设置覆盖值时自动应用
3. 无系统提示词：如果两者都没有设置

按模型覆盖

不同模型可能需要不同的提示词策略。可以在配置层面覆盖：

```
# Thinking model - needs explicit formatting instruction
thinking_config = BenchmarkConfig.for_model(
    "Qwen/Qwen3.6-35B-A3B",
```

```
system_prompt="Think step by step. Put your final answer in
\\boxed{ }.",
)

result = await run_benchmark("gsm8k", client, renderer, thinking_config)
print(f"thinking: {result.score:.1%}")
```

内置的推荐提示词

基准测试	recommended_system_prompt
gsm8k	"Put your final answer in \boxed{ }."
math500	"Put your final answer in \boxed{ }."
aime* , hfmt*	"Put your final answer in \boxed{ }."
mmlu_pro , gpqa , supergpqa	无（选择题格式说明放在用户提示词中）
ifeval	无
其他大多数	无

答案解析与评分

内置抽取工具

框架在 `tinker_cookbook.eval.benchmarks._common` 中提供了常用的答案抽取函数：

函数	用途
<code>extract_boxed(text)</code>	从 <code>\boxed{...}</code> 中抽取内容（支持嵌套花括号）
<code>extract_gsm8k_answer(text)</code>	抽取数值答案——依次尝试 <code>\boxed{ }</code> 、 <code>####</code> 、 <code>answer is</code> ，最后取最后一个数字
<code>extract_mcq_answer(text)</code>	抽取选择题选项字母（A/B/C/D）
<code>extract_python_code(text)</code>	从围栏代码块中抽取 Python 代码
<code>check_gsm8k(response, expected)</code>	带容差的浮点数比较

每个基准测试的 `MessageEnv.step()` 都使用这些函数来评分。抽取出的答案和预期答案会存入 `logs` 以便查看。

自定义评分函数

`BenchmarkConfig.grade_fn` 可以在不修改 `Env` 的情况下覆盖内置评分逻辑。运行器会在 rollout 完成后调用它，替换奖励值：

```
def my_grader(response: str, logs: Logs) -> float:
    """Custom grading function.

    Args:
        response: Last assistant turn (thinking stripped).
        logs: Benchmark-specific fields - e.g., "expected", "input",
            "extracted".

    Returns:
        Reward value (typically 0.0 or 1.0).
    """
    expected = logs["expected"]
    # Use your own extraction logic
    extracted = my_custom_extract(response)
    return 1.0 if extracted == expected else 0.0

config = BenchmarkConfig(grade_fn=my_grader)
result = await run_benchmark("gsm8k", sampling_client, renderer, config)
```

这在以下情况下很有用：

- 内置抽取逻辑漏掉了你的模型使用的某种合法答案格式
- 你想要更宽松或更严格的评分标准
- 你需要特定领域的解析逻辑

无需重跑的重新评分

`regrade_trajectories()` 会用一个新评分函数处理已保存的轨迹——不需要再做模型推理：

```
from tinkercookbook.eval.benchmarks import regrade_trajectories

# Try a more lenient grader on existing results
def lenient_grader(response: str, logs: Logs) -> float:
    expected = logs["expected"]
```

```
# Accept answers with or without \boxed{}
return 1.0 if expected in response else 0.0

regraded = regrade_trajectories("evals/run_01", "gsm8k",
grade_fn=lenient_grader)
print(f"Original: {original.score:.1%}, Regraded: {regraded.score:.1%}")
```

裁判模型

像 `arena_hard` 这样的基准测试使用 LLM 裁判来评分。裁判与候选模型完全独立——不同的模型、不同的渲染器、不同的规模。

配置裁判

```
# Candidate model: the model being evaluated
candidate_client = await
sc.create_sampling_client_async(model_path="tinker://run-
abc/weights/final")
candidate_renderer = get_renderer("qwen3_5", candidate_tokenizer)

# Judge model: a separate, typically stronger model
judge_client = await
sc.create_sampling_client_async(base_model="Qwen/Qwen3.5-397B-A17B")
judge_renderer = get_renderer("qwen3_5", judge_tokenizer)

config = BenchmarkConfig(
    judge_sampling_client=judge_client,
    judge_renderer=judge_renderer,
)

result = await run_benchmark("arena_hard", candidate_client,
candidate_renderer, config)
```

评分如何工作

对于 `arena_hard`，流程如下：

1. 候选模型针对基准测试问题生成回答
2. 裁判模型接收问题与回答并打分（1-10）
3. 得分 ≥ 7 被判为正确

如果没有显式设置，裁判渲染器默认使用候选渲染器。

更换裁判

比较不同裁判对同一个模型的评分：

```
for judge_name, judge_client in judges.items():
    config = BenchmarkConfig(
        judge_sampling_client=judge_client,
        judge_renderer=judge_renderer,
    )
    result = await run_benchmark("arena_hard", candidate_client,
                                candidate_renderer, config)
    print(f"Judge={judge_name}: {result.score:.1%}")
```

了解更多

- [评测概览](#)：可用的基准测试与整体架构
- [基准测试指南](#)：配置、存储、pass@k
- [RL 环境](#)：与基准测试共享的 Env 协议

Storage（存储）

最近更新时间：2026-10-19 21:18:02

训练与评测数据（指标、检查点、rollout、轨迹）都通过统一的存储层保存。默认情况下，数据写入本地磁盘。要使用云存储（GCS、S3、Azure），只需把路径改成 URI 即可。

如需云存储支持：

```
uv pip install 'tinker-cookbook[cloud]'
```

训练

把云 URI 作为 `log_dir` 传入：

```
# Local (default)
ml_logger = setup_logging(log_dir="/tmp/my_run", config=config)

# GCS
ml_logger = setup_logging(log_dir="gs://bucket/my_run", config=config)

# S3
ml_logger = setup_logging(log_dir="s3://bucket/my_run", config=config)
```

所有训练数据（指标、检查点、rollout）都会自动写入云路径。

评测

把云 URI 作为 `save_dir` 传入：

```
# Local
config = BenchmarkConfig(save_dir="/tmp/evals/run1")

# GCS
config = BenchmarkConfig(save_dir="gs://bucket/evals/run1")

result = await run_benchmark("gsm8k", client, renderer, config)
```

读回数据

```
from tinkercookbook.stores import TrainingRunStore, storage_from_uri

# Works with local paths and cloud URIs
store = TrainingRunStore(storage_from_uri("gs://bucket/my_run"))

config = store.read_config()
metrics = store.read_metrics()
rollouts = store.read_rollouts(0)
checkpoints = store.read_checkpoints()
```

支持的后端

URI	存储后端
<code>/local/path</code>	本地文件系统
<code>gs://bucket/prefix</code>	Google Cloud Storage
<code>s3://bucket/prefix</code>	Amazon S3
<code>az://container/prefix</code>	Azure Blob Storage

任何 [fsspec](#) 支持的文件系统 都可以使用。

云后端的 flush

为了性能，云端写入会先在本地暂存。请在检查点处调用 `flush()`，以确保数据被上传：

```
storage = storage_from_uri("gs://bucket/run")
store = TrainingRunStore(storage)

for step in range(num_steps):
    store.write_metrics({"loss": loss}, step=step)
    if step % save_every == 0:
        storage.flush()
```

或者使用上下文管理器实现自动 flush：

```
with storage_from_uri("gs://bucket/run") as storage:
    store = TrainingRunStore(storage)
    # ... training ...
```

```
# auto-flush on exit
```

Preferences（从偏好中学习）

概述

最近更新时间：2026-10-19 21:18:02

在 Preferences（从偏好中学习）这一部分，我们聚焦于从**成对反馈**中学习，也就是拥有这样一类偏好数据：对于给定提示词，它能指出两个补全中哪一个更好。这类反馈天然适用于那些无法通过程序化方式计算简单正确性标准的任务。这些偏好可能来自人类评估者，也可能由模型生成。

偏好学习的两种路径

当您拥有成对偏好数据时，有两条主要路径：

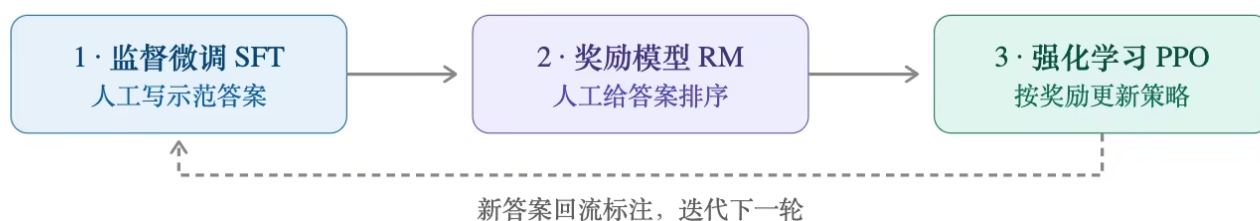
- **DPO（直接偏好优化）**：直接更新策略，使其更偏好 chosen 响应而非 rejected 响应，不需要单独的奖励模型。这种方法更简单，计算成本也更低。详见 [DPO 指南](#)。
- **RLHF（基于人类反馈的强化学习）**：先在偏好数据上训练一个奖励模型，然后用强化学习针对该奖励模型优化策略。这种两阶段方式提供了更大的灵活性。详见 [RLHF 示例](#)。

RLHF（基于人类反馈的强化学习）

最近更新时间：2026-10-19 21:18:03

RLHF（Reinforcement Learning from Human Feedback，基于人类反馈的强化学习）的核心思路是：让模型不只学“像人一样说话”，而是学“说人喜欢的话”。它把难以写进损失函数的“偏好”，例如“有用、无害、礼貌、简洁”学成一个可计算的奖励信号，再用强化学习去优化。

RLHF 通过一个三阶段流水线让语言模型与人类偏好对齐。每个阶段都建立在上一阶段之上：监督微调产出一个初始策略，奖励模型学习为输出打分，然后通过强化学习针对学到的奖励进行策略优化。



阶段 1：监督微调

基座模型在 [no_robots](#) 数据集上微调，该数据集包含人工编写的指令跟随示例，设计上对齐 InstructGPT 的方法论。这让模型具备基本的指令跟随能力，并产出 RL 后续要打磨的初始策略。

阶段 2：奖励模型

另一个单独的模型在 [Anthropic HHH](#) 成对比较数据集上训练。给定同一提示词的两个补全，模型学习预测人类更偏好哪一个。`ComparisonRenderer` 用分节标记格式化每一对，让模型在一次前向传播中同时看到两个补全。

阶段 3：强化学习

SFT 策略通过自博弈针对偏好模型进行优化。对每个提示词采样多个补全，偏好模型以锦标赛方式对所有配对打分。每个补全依据其胜率获得奖励，策略随之更新，从而更多地生成那些获胜的响应。

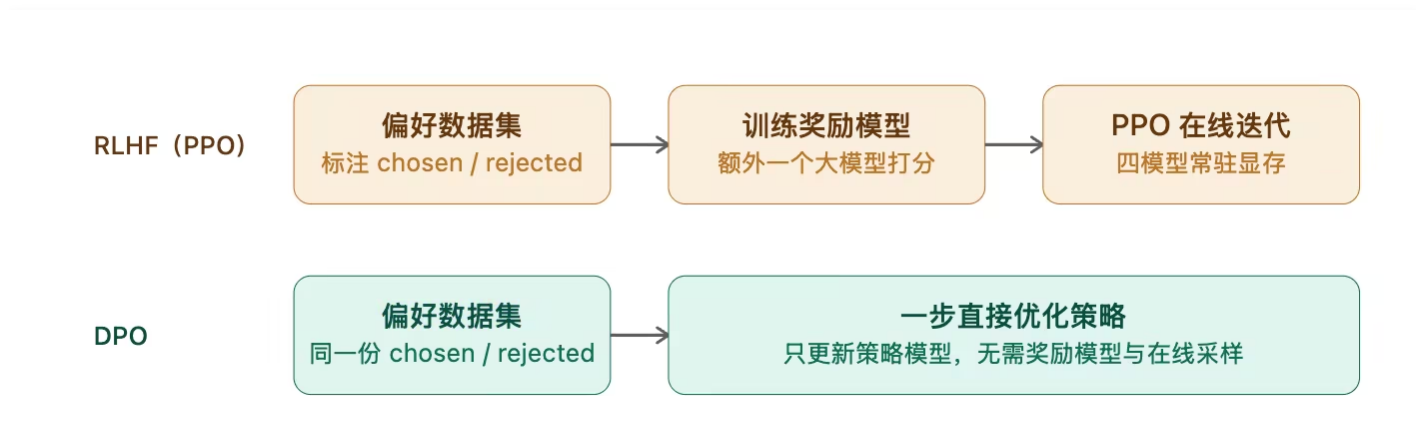
资源

- [RLHF 教程](#)：交互式 marimo notebook，带代码逐步走完三个阶段
- [RLHF 案例](#)：生产级流水线脚本，带 CLI 配置和 wandb 日志记录

DPO（直接偏好优化）

最近更新时间：2026-10-19 21:38:00

DPO（Direct Preference Optimization，直接偏好优化）是 2023 年 NeurIPS 的一篇论文（Rafailov et al.）提出的方法，核心主张是：RLHF 里那个“先训奖励模型、再用 PPO 做强化学习”的两步流程，其实可以压成一步普通的监督学习。它现在是几乎所有开源对齐模型（Zephyr、Llama-3、Qwen 系列）的默认后训练算子。DPO 用分类损失训练模型，使其更偏好 chosen 回复而非 rejected 回复，不需要单独的奖励模型。



DPO 与 RLHF

DPO 直接优化策略，因此不再需要单独的奖励模型。它比经典 RLHF 更简单、更便宜，但要求基座模型已经与偏好数据同分布。

DPO 损失

$$\mathcal{L}_{\theta} = -\mathbb{E}_{x, y_{\text{chosen}}, y_{\text{rejected}} \sim \mathcal{D}} \left[\log \sigma \left(\beta \log \frac{\pi_{\theta}(y_{\text{chosen}}|x)}{\pi_{\text{ref}}(y_{\text{chosen}}|x)} - \beta \log \frac{\pi_{\theta}(y_{\text{rejected}}|x)}{\pi_{\text{ref}}(y_{\text{rejected}}|x)} \right) \right]$$

损失函数公式

- π_{θ} — 正在训练的当前策略
- π_{ref} — 参考模型（冻结，通常是 DPO 之前的检查点）
- β — 控制偏好强度（越大越保守）

DPO：偏好数据 chosen 与 rejected → 参考策略 π_{ref} （冻结）→ DPO 损失 β 加权 → 对齐后的模型 π_{θ}

可用数据集

数据集	来源	说明
hhh	Anthropic	Helpful-Harmless-Honest 成对偏好数据
helpsteer3	NVIDIA	HelpSteer3 偏好数据集
ultrafeedback	UltraFeedback	二值化的偏好比较

关键超参数

- `dpo_beta`：从 `0.1` 开始。取值越大越保守。
- `learning_rate`：比 SFT 更低，通常取 `1e-5` 到 `1e-6`。
- 基座模型：应与偏好数据同分布。可以先做一轮轻量 SFT，或收集 on-policy 的偏好数据。

训练指标

- `dpo_loss`：分类损失（应下降）
- `accuracy`：模型偏好 chosen 回复的频率
- `margin`：chosen 与 rejected 之间的平均奖励差
- `chosen_reward` / `rejected_reward`：平均隐式奖励

延伸阅读

- [DPO 与偏好教程](#)：动手实践、可交互的演练
- [RLHF 流程教程](#)：带奖励模型的完整三阶段 RLHF
- [DPO Recipe](#)：生产环境训练脚本