PR #6673 完整报告

verl-project/verl

[reward] fix: add optional per-sample compute_score timeout to NaiveRewardManager

合并时间：2026-06-12 14:39

原文链接：http://prhub.com.cn/verl-project/verl/pull/6673

执行摘要

- 一句话：为 NaiveRewardManager 添加 per-sample 超时机制
- 推荐动作：建议在以下场景启用：使用自定义 compute_score 可能存在性能波动（如基于规则的数学验证、符号计算）时。PR 设计谨慎，适合合入。后续可关注线程安全的超时机制（如 concurrent.futures 进程池）以替代信号方案。

功能与动机

作者在 PR body 中描述：'NaiveRewardManager scores samples serially in the driver process with no timeout, one stuck call blocks the entire training loop with no recovery.' 具体案例是行内多教师蒸馏训练中，学生模型退化生成长序列导致 compute_score 内部（如 normalize_final_answer）长时间阻塞。本 PR 防止此类 hang 影响训练。

实现拆解

1. 实现超时上下文管理器：在 naive.py 中新增 _score_timeout(seconds) 上下文管理器，基于 signal.SIGALRM 和 signal.setitimer。当 seconds 为 None 或 <=0 时跳过；若当前线程不支持 SIGALRM（非主线程）则降级为 no-op，不安装信号处理器。
2. 修改构造函数：NaiveRewardManager.__init__ 新增 compute_score_timeout=None 参数，保存到实例属性。在 docstring 中说明。
3. 改造打分循环：在 __call__ 方法的 compute_score 调用处包裹 with _score_timeout(self.compute_score_timeout):，并捕获 TimeoutError。若超时，将 reward 设为 0.0，打印警告，继续处理下一样本。
4. 新增单元测试：创建 tests/workers/reward_manager/test_naive_on_cpu.py，测试 _score_timeout 的正常通过、超时抛出、禁用效果；测试 NaiveRewardManager 在超时配置下快速返回且赋 0 分；验证默认无超时行为不变。

关键文件：

- verl/workers/reward_manager/naive.py（模块 奖励管理器；类别 source；类型 core-logic；符号 _score_timeout, _handler, init, call）：核心源码修改：新增超时上下文管理器，修改构造函数和 __call__打分循环，实现超时保护逻辑。
- tests/workers/reward_manager/test_naive_on_cpu.py（模块 测试套件；类别 test；类型 test-coverage；符号 _DummyTokenizer, decode, _make_minimal_data, test_score_timeout_allows_fast_calls）：全新测试文件，覆盖超时机制各场景，保障回归。

关键符号：_score_timeout, NaiveRewardManager.init, NaiveRewardManager.call

关键源码片段

verl/workers/reward_manager/naive.py

核心源码修改：新增超时上下文管理器，修改构造函数和 __call__打分循环，实现超时保护逻辑。

```
import signal
from contextlib import contextmanager
from typing import Optional

@contextmanager
def _score_timeout(seconds: Optional[float]):
    """上下文管理器，在指定秒数后抛出 TimeoutError。
    仅对主线程有效，非主线程降级为 no-op。
    """
    if not seconds or seconds <= 0:
        yield
        return

    def _handler(signum, frame):
        raise TimeoutError(f"compute_score timed out after {seconds}s")

    try:
        old_handler = signal.signal(signal.SIGALRM, _handler)
    except ValueError:
        # 非主线程，无法安装 SIGALRM
        yield
        return

    signal.setitimer(signal.ITIMER_REAL, seconds)
    try:
        yield
    finally:
        signal.setitimer(signal.ITIMER_REAL, 0)
        signal.signal(signal.SIGALRM, old_handler)


class NaiveRewardManager(AbstractRewardManager):
    def __init__(self, tokenizer, num_examine, compute_score=None,
                 reward_fn_key="data_source", compute_score_timeout=None):
        """
        Args:
            ...
            compute_score_timeout: 可选，每样本 compute_score 超时秒数。
                超时则赋 0 分并继续。默认 None，不变。
        """
        self.tokenizer = tokenizer
```

```
        self.num_examine = num_examine
        self.compute_score = compute_score or default_compute_score
        self.reward_fn_key = reward_fn_key
        self.compute_score_timeout = compute_score_timeout # 存储超时参数

    def __call__(self, data, return_dict=False):
        # ... 前置处理 ...
        for i in range(len(data)):
            # ... 解码 prompt/response，提取 ground_truth 等 ...
            try:
                with _score_timeout(self.compute_score_timeout):
                    score = self.compute_score(
                        data_source=data_source,
                        solution_str=response_str,
                        ground_truth=ground_truth,
                        extra_info=extra_info,
                    )
            except TimeoutError:
                # 超时：赋 0 分，记录警告，继续循环
                reward = 0.0
                logger.warning(f"compute_score timed out for sample {i}, reward set to 0.0")
            else:
                if isinstance(score, dict):
                    reward = score["score"]
                    # 处理额外信息
                else:
                    reward = score
            # 赋值 reward_tensor 等后续操作
```

tests/workers/reward_manager/test_naive_on_cpu.py

全新测试文件，覆盖超时机制各场景，保障回归。

```
import time
import numpy as np
import pytest
import torch
from verl import DataProto
from verl.workers.reward_manager.naive import NaiveRewardManager, _score_timeout

class _DummyTokenizer:
    def decode(self, token_ids, skip_special_tokens=True):
        return "dummy" # 简化的解码

def _make_minimal_data(batch_size=1, prompt_len=2, response_len=2):
    total = prompt_len + response_len
    return DataProto.from_single_dict({
        "prompts": torch.zeros((batch_size, prompt_len), dtype=torch.long),
        "responses": torch.zeros((batch_size, response_len), dtype=torch.long),
        "attention_mask": torch.ones((batch_size, total), dtype=torch.long),
```

```
        "data_source": np.array(["dummy"] * batch_size, dtype=object),
        "reward_model": np.array([{"ground_truth": "1"} for _ in range(batch_size)], dtype=object),
    })

def test_naive_reward_manager_times_out_slow_score():
    """超时场景：慢 compute_score 应被中断并返回 0.0。"""
    def slow_compute_score(data_source, solution_str, ground_truth, extra_info=None):
        time.sleep(10) # 人工延时
        return 1.0

    manager = NaiveRewardManager(
        tokenizer=_DummyTokenizer(),
        num_examine=0,
        compute_score=slow_compute_score,
        compute_score_timeout=1.0, # 1s 超时
    )
    data = _make_minimal_data()
    start = time.time()
    out = manager(data, return_dict=True)
    elapsed = time.time() - start
    assert elapsed < 5.0 # 远小于 10s，证明超时生效
    assert out["reward_tensor"].sum().item() == 0.0 # 超时样本得 0 分

def test_naive_reward_manager_no_timeout_by_default():
    """默认无超时：保持原有行为。"""
    def fast_compute_score(data_source, solution_str, ground_truth, extra_info=None):
        return 0.5
    manager = NaiveRewardManager(
        tokenizer=_DummyTokenizer(),
        num_examine=0,
        compute_score=fast_compute_score,
    )
    data = _make_minimal_data()
    out = manager(data, return_dict=True)
    assert out["reward_tensor"].sum().item() == pytest.approx(0.5)
```

评论区精华

Review 中无实质性评论。作者在 PR body 中主动讨论了信号机制的局限性，并欢迎改用进程池方案（如 PrimeRewardManager 的做法）以支持非主线程场景。

- 暂无高价值评论线程

风险与影响

- 风险：
  1. 信号安全：SIGALRM 只在主线程有效。若未来 NaiveRewardManager 运行在 Ray actor 的非主线程，超时将静默失效（降级 no-op），失去保护。需在架构变更时同步检查。

2. 精度损失：超时样本直接赋 0.0 可能掩盖一些实际情况（如该样本实际应得高分被误杀），但日志有警告，可人工审计。
3. 信号干扰：若已有自定义 SIGALRM 处理器，本实现会暂存并恢复旧处理器，降低冲突风险。 - 影响：影响范围：仅 NaiveRewardManager 用户；默认行为不变，无升级风险。启用后可避免单一样本打分 hang 造成训练阻塞，提升大规模训练鲁棒性。测试覆盖主要路径，核心路径改动较小。 - 风险标记：SIGALRM 仅主线程有效，超时样本赋 0 分可能掩盖问题，默认关闭需显式配置

关联脉络

- 暂无明显关联 PR