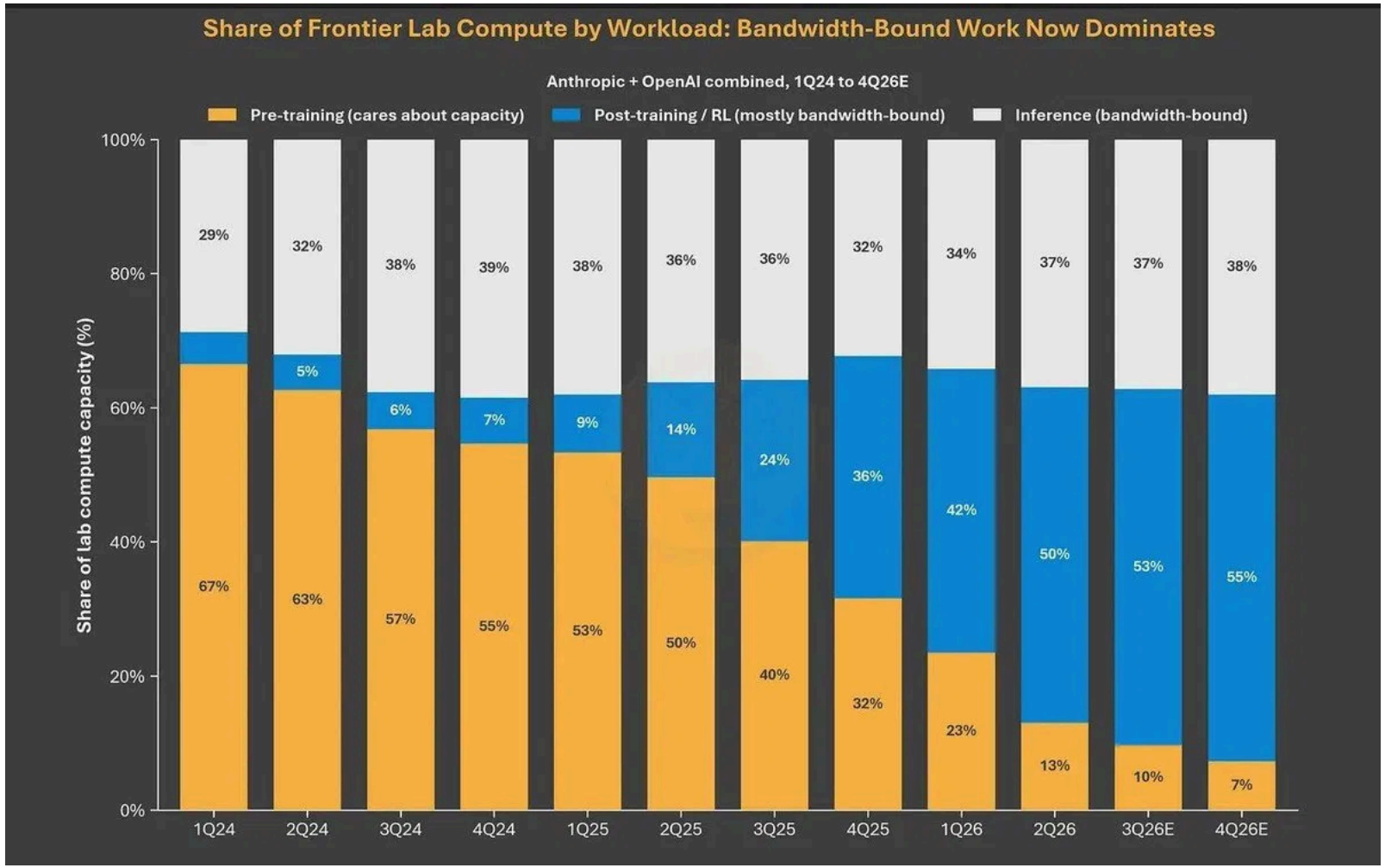
From Classical RL to Frontier LLMs

18 NOVEMBER 2026

Nathan Lambert

Post-training (RL) now takes most of a frontier lab's compute



Share of combined Anthropic and OpenAI compute by workload, 1Q24 to 4Q26 (later quarters are estimates). Pretraining falls from 67% to 7%, while post-training and RL rise from 5% to 55%.

SemiAnalysis, via OfficeChai

How the times have changed!

RL compute relative to pretraining compute for the same model, in GPU-hours:

Model	Pretraining	RL	RL / pretraining
DeepSeek R1 Jan 2025 671B total, 37B active	**2.66M** H800 GPU-hours (DeepSeek-AI, 2025)	**0.14M** H800 GPU-hours (Guo et al., 2025)	**~5%**
Beam Oct 2026 501B total, 23B active	**< 4.1M** 6,144 GB300s for under 4 weeks	**~7.1M** 10.5K GB300s for 4 weeks	**> 1.7×**
Inkling 2 Small Nov 2026 594B total, 17B active	**~0.11M*** ~1.5K GPUs for under 3 days	**~0.44M** 2,048 GPUs for 9 days	**~4×***

*Estimated: Thinking Machines report the pretraining duration but not the GPU count. GPU-hours ignore utilization. RL spends much of its time sampling, so its share of FLOPs is smaller than its share of GPU-hours.

What this lecture covers

How RL is approached with LLMs and what the challenges of RL are for frontier models.

RL is mostly an engineering, stability, and throughput problem at the frontier. The learning signal is important, and mostly comes through data engineering (not the focus of this lecture).

What this lecture covers

How RL is approached with LLMs and what the challenges of RL are for frontier models.

Some examples we include:

1. **Policy gradients in LLM code.** How LLM log-probs, logits, and a surrogate loss turn REINFORCE into a few lines of PyTorch.
2. **Asynchronous RL systems.** How RL training is usually done for large LLMs.
3. **Training-inference mismatch.** A common challenge that many of the latest RL algorithms address.

Notation in this lecture

Symbol	Meaning	Book / lectures
x	Prompt	s
$a = (a_1, \ldots, a_T)$	Completion, one token at a time	a
$v \in \mathcal{V}$	Any token in the vocabulary	v
p_θ	**Learner**: the training framework's (e.g., FSDP) next-token distribution	$\pi_\theta^{\text{learner}}$
q	**Sampler**: the inference engine's (e.g., vLLM) next-token distribution (no gradient updates)	$\pi_\theta^{\text{sampler}}$
f_θ	Logits, so $p_\theta = \mathrm{softmax}(f_\theta)$	
$\boldsymbol{\xi}_v = \nabla_\theta \log p_v$	Gradient "**score**" of token v	$\boldsymbol{\xi}_v$
A	Advantage for the whole completion, e.g., $r - b(x)$	A

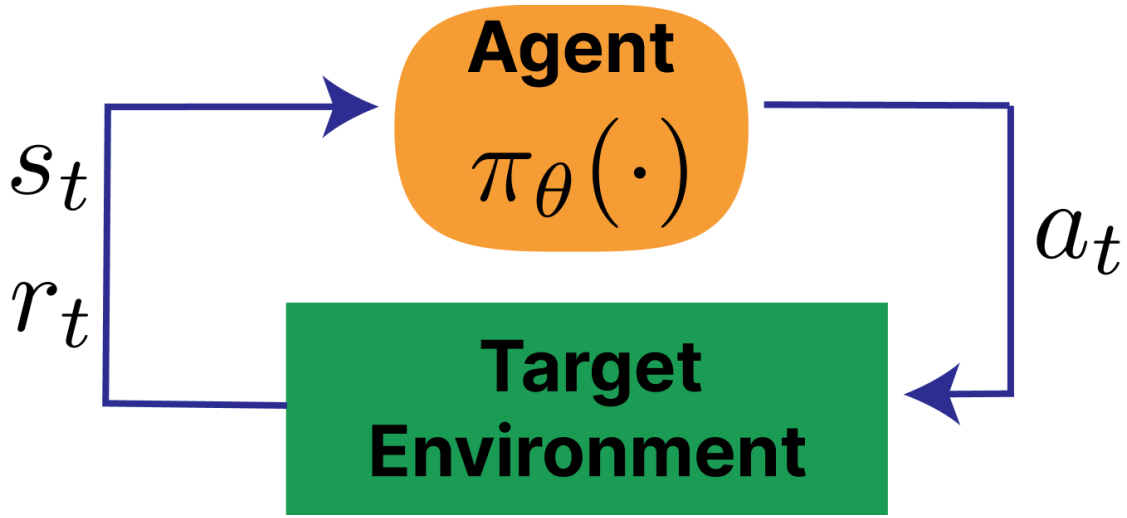
p_v and q_v are shorthand for the probability of token v at one position, given x and the prefix $a_{<t}$. CS 4789 uses the same x and a.

Implementing RL basics with an LLM

RL review: The agent–environment loop

Classical RL: agent in an MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$

- Reward is a known function from the environment per step
- Optimize cumulative return: $J(\pi) = \mathbb{E}_{\tau \sim p_\pi}[\sum_t \gamma^t r(s_t, a_t)]$



Language models as episodic MDPs

MDP concept	Language model
State s_t	Prompt + tokens generated so far: $(x, a_{<t})$
Action a_t	Next token a_t
Transition	Deterministic-ish: append token to sequence (tokens are sampled probabilistically)
Policy $\pi_\theta(a_t \mid s_t)$	LM next-token distribution
Episode	One prompt → completion
Terminal reward	RM score, verifier output, judge-scores
γ	Typically 1.0 (no discounting)

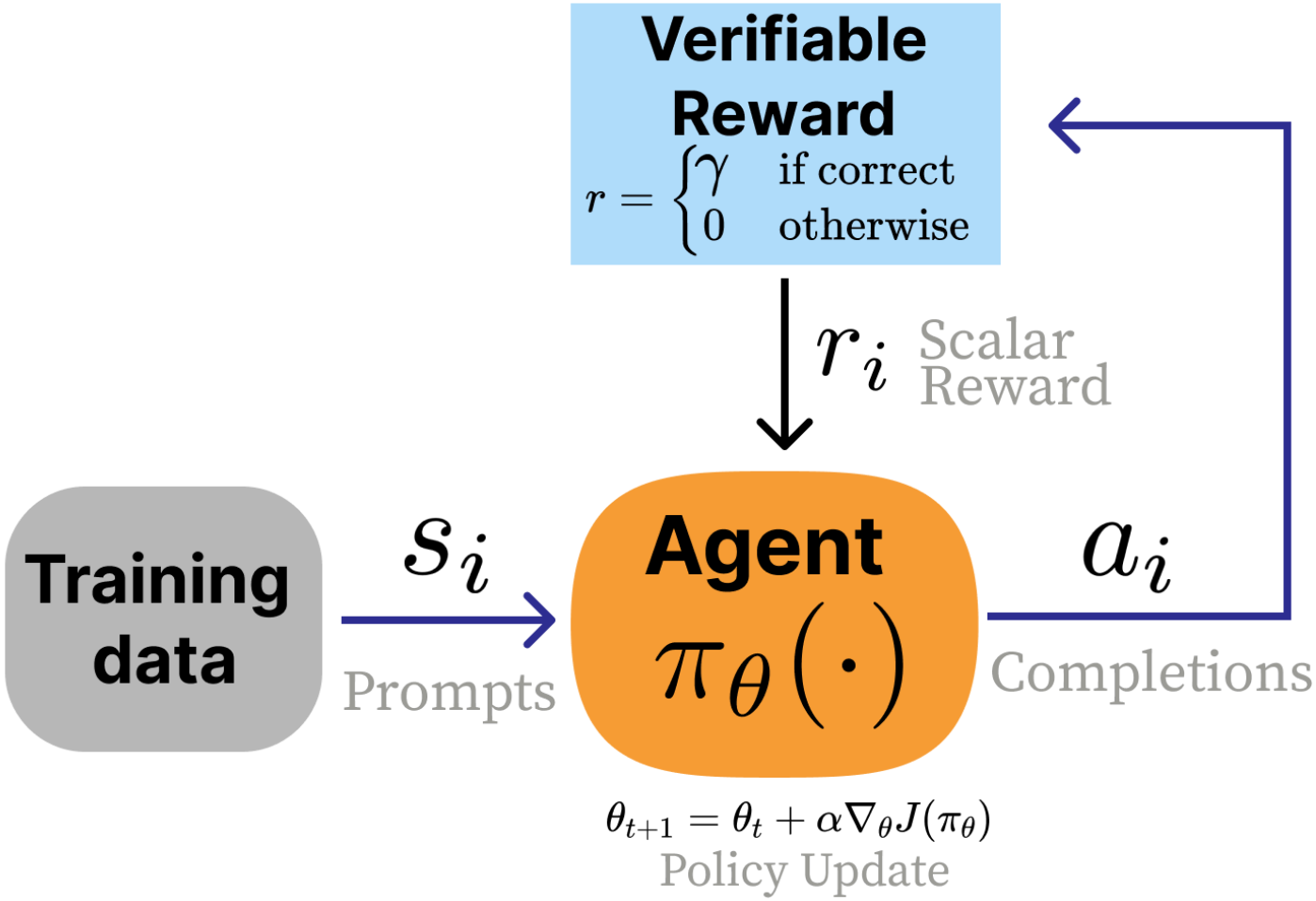
Two implementation-level views of this MDP: **token-level** (each token is a separate action, used in PPO with GAE) vs. **sequence-level** (entire completion is one action, used in REINFORCE/GRPO).

Agentic RL complicates the MDP

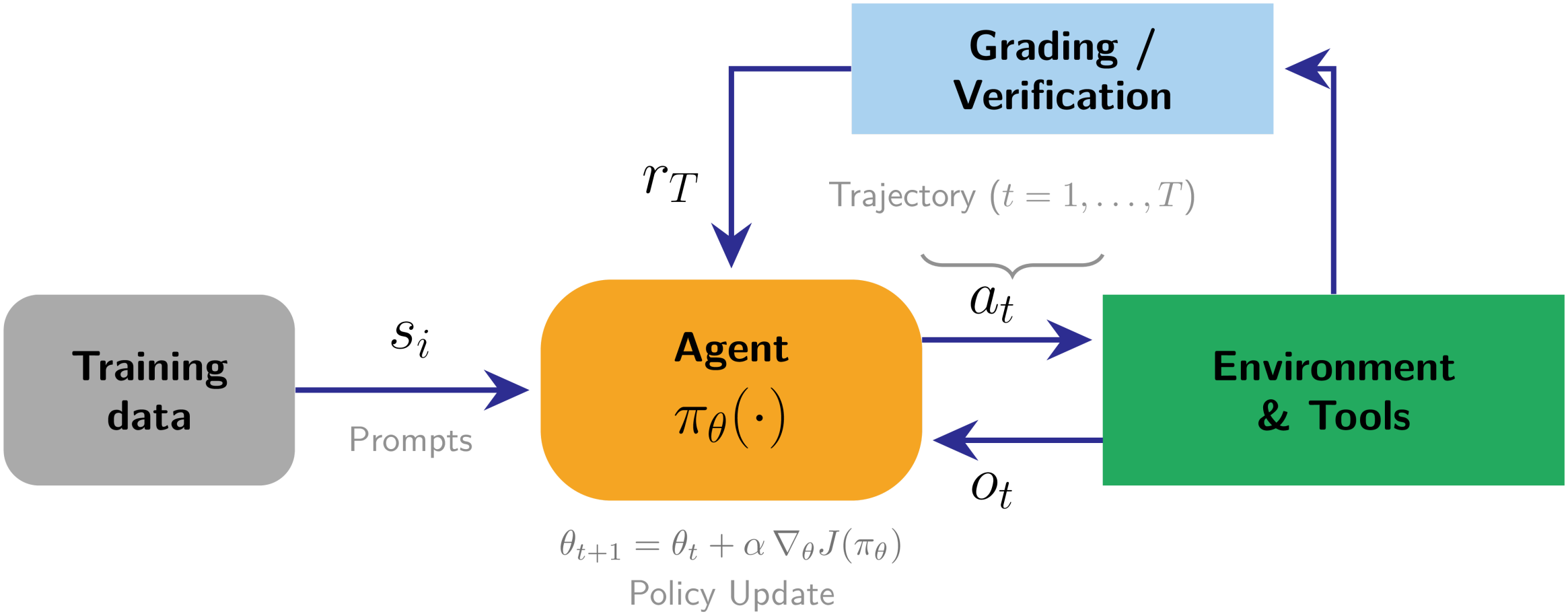
MDP concept	RLVR	Agentic RL
State s_t	Prompt + tokens so far	Context (prompt + tokens + tool outputs), plus hidden environment state (files, services, other users)
Action a_t	Next token	Next token (many are tool calls)
Transition	Deterministic-ish: append token	More stochastic: the environment runs the call and appends its output
Episode	One prompt → completion	Many turns; Inkling 2 Small's p99 rollout takes 4 hours
Terminal reward	Verifier on the final answer	Tests on the final state of the environment
Loss	Every completion token	Tool-output tokens are masked: the model must not learn to predict the environment

Math/code simply verifiable RL needed prompts and answer checkers. Agentic RL needs **environments**: containers, real file systems, services, verification tests (Ivison et al., 2026).

RL for LLMs: RLVR



RL for LLMs: Agentic RL



Recall: The policy gradient

The objective and its gradient:

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta}[R(\tau)]$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta}\left[\sum_{t=0}^{T} \Psi_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right]$$

The gradient says: for each sampled token, compute the direction in parameter space that makes it more likely ($\nabla \log \pi$), then scale by how much better than expected it was (Ψ_t).

Recall: Why losses are sums of log-probs

A language model generates one token at a time, so a trajectory's probability is a product of next-token probabilities. The next state is just the context plus the new token, so the dynamics add no terms that depend on θ:

$$p_\theta(\tau) = \prod_{t=0}^{T} \pi_\theta(a_t \mid s_t)$$

The log turns the product into a sum, so its gradient is a sum of per-token terms:

$$\nabla_\theta \log p_\theta(\tau) = \nabla_\theta \sum_{t=0}^{T} \log \pi_\theta(a_t \mid s_t) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)$$

Recall: A loss whose gradient is the policy gradient

So we write a loss whose negative gradient is the policy gradient, treating each advantage A_i as a constant across the trajectory:

$$L(\theta) = -\frac{1}{N}\sum_{i=1}^{N} A_i \sum_t \log \pi_\theta(a_{i,t} \mid s_{i,t})$$

$$-\nabla_\theta L(\theta) = \frac{1}{N}\sum_{i=1}^{N} A_i \sum_t \nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})$$

Gradient descent on L is gradient ascent on J.

Reminder: when implementing algorithms like this, we write code for the loss, and autodiff yields the gradients.

From the loss to the gradient

In code, we compute the **log-probs** and let autodiff handle the gradient:

```
seq_log_probs = (token_log_probs * completion_mask).sum(dim=-1)
loss = -(seq_log_probs * advantages.detach()).mean()
loss.backward()  # fills each parameter's .grad with ∇L
optimizer.step()  # θ ← θ − η ∇L
```

Line 2 implements the policy gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta}\left[\sum_{t=0}^{T} A \, \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right]$$

`loss.backward()` runs backpropagation: it computes $\partial L / \partial \theta$ for every parameter. `optimizer.step()` applies the update. The loss values themselves are often weird in RL (e.g. zero-centered distributions, rather than nice decreasing accuracy metrics of pretraining) Most RL algorithms used today are variations on this computation.

Computing log-probabilities (expanded)

The fundamental building block: per-token log-probabilities from the policy.

```
# Forward pass through the model
logits = model(input_ids).logits              # (B, L, vocab_size)

# Autoregressive shift: logit at position t predicts token at t+1
logits = logits[:, :-1, :]                    # (B, L-1, vocab_size)
labels = input_ids[:, 1:]                     # (B, L-1)
completion_mask = completion_mask[:, 1:]       # (B, L-1)

# Per-token log-probs
log_probs = logits.log_softmax(dim=-1)        # (B, L-1, vocab_size)
token_log_probs = log_probs.gather(dim=-1, index=labels.unsqueeze(-1)).squeeze(-1) # (B, L-1)

# Per-sequence log-prob: sum over completion tokens
seq_log_probs = (token_log_probs * completion_mask).sum(dim=-1)  # (B,)
```

Everything in log-space to avoid numerical underflow from multiplying many small probabilities.

The REINFORCE loss in code

The simplest policy gradient loss:

```
# rewards: (B,) — one reward per sequence
# seq_log_probs: (B,) — sum of log-probs over completion tokens

# Baseline: average reward in the batch
baseline = rewards.mean()
advantages = rewards - baseline

# REINFORCE loss (negative because we minimize)
loss = -(advantages * seq_log_probs).mean()
```

Positive advantage → increase probability. Negative → decrease.

(Aside: How we reduce per-token losses to a scalar — `.mean()` here — turns out to matter more than you'd expect. More in the book and another lecture.)

Challenges of distributed RL training

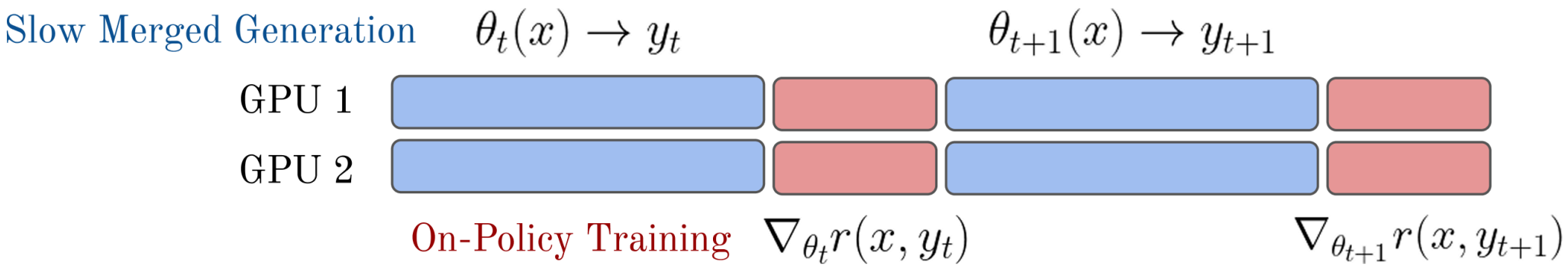
On-policy vs. off-policy

Policy-gradient theory is built on the assumption of on-policy data – from the current model weights in the training framework of the LLM.

As models get bigger, this is hard because: sampling is expensive, sampling is dominated by long-trail generations, most people use separate systems for inference v training, moving large model weights around is slow, inference weights can be further quantized for speed relative to training weights, multi-cluster deployments, etc.

Synchronous: Share generation and training

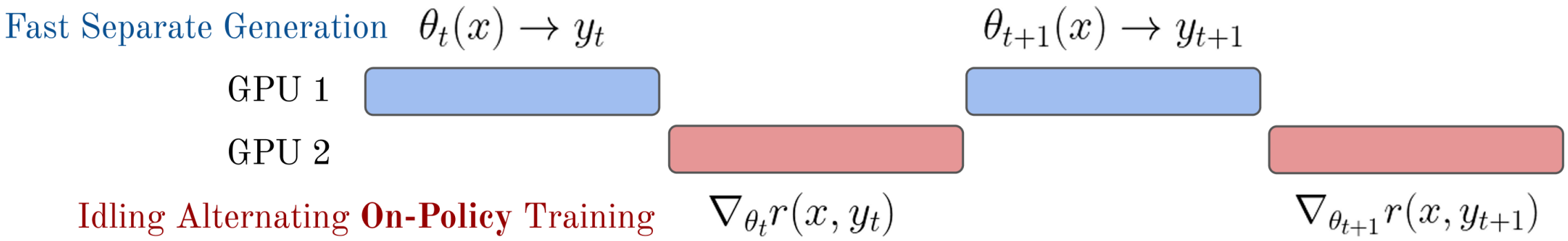
The default implementation for policy-gradient algorithms is **on-policy** execution: the generations from the language model are scored before updating the model.



Noukhovitch et al., 2025

Intermediate step: Separate GPUs, still on-policy

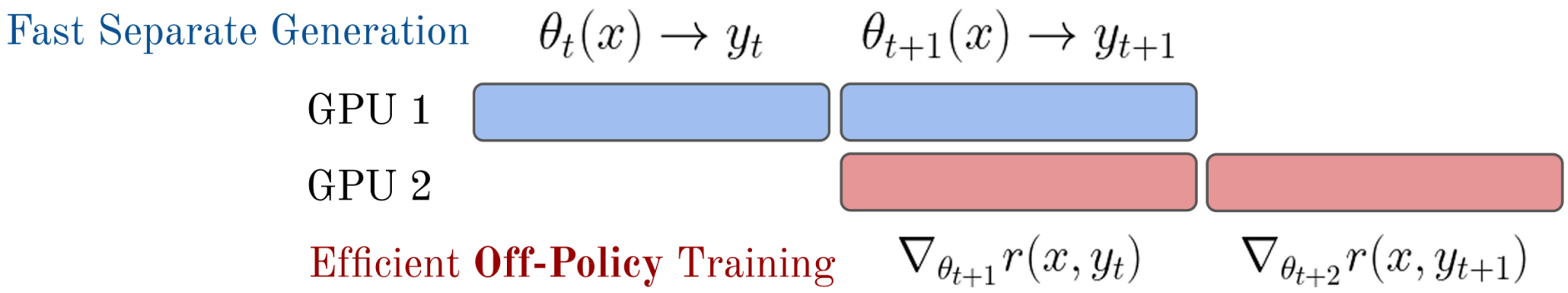
Moving generation to an inference engine on its own GPUs makes it fast. To stay on-policy, each side waits for the other.



Noukhovitch et al., 2025

Asynchronous: Off-policy training

The common solution is to constantly run inference and training on separate GPU nodes. The model used for generation may be 1–N steps behind the training model.



This turns out to be mostly "fine" – but it can be taken too far.

Noukhovitch et al., 2025

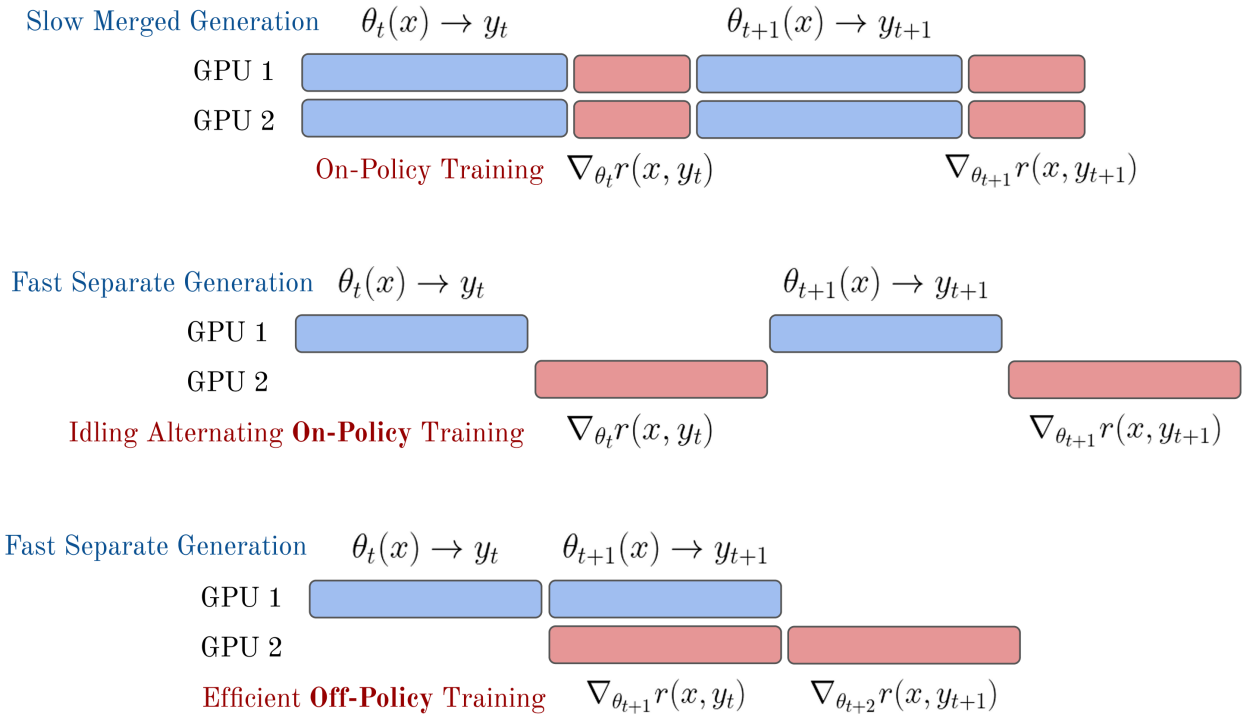
Synchronous vs. asynchronous training

Ideal / Theory (on-policy): generate → update → generate → update

Each batch of completions is scored and used for a short update window (one or a few epochs), then discarded.

Reality (async): generation and training overlap on different GPU groups for better throughput. The model used for generation may be 1–N steps behind the training model.

Tradeoff: perfect on-policy is slow (GPUs idle during generation or training). Slight staleness (how off-policy the data is) is usually fine.



Synchronous training idles GPUs; separating generation and training helps, but on-policy requires waiting. Async (off-policy) overlaps both for full utilization.

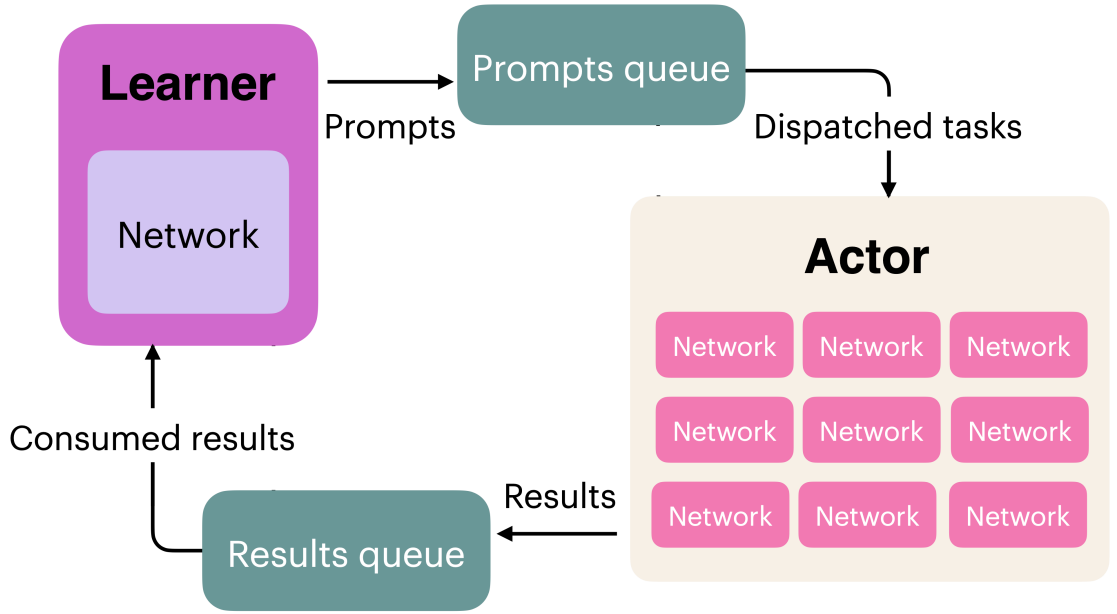
Noukhovitch et al., 2025

Asynchronous RL training

Modern RL for LLMs splits compute into two groups:

- **Actors** (inference GPUs): generate completions using vLLM or similar
- **Learners** (training GPUs): compute policy gradient updates

A process management library (e.g., Ray) coordinates data flow between them. Model weights are synced periodically from learner → actor.

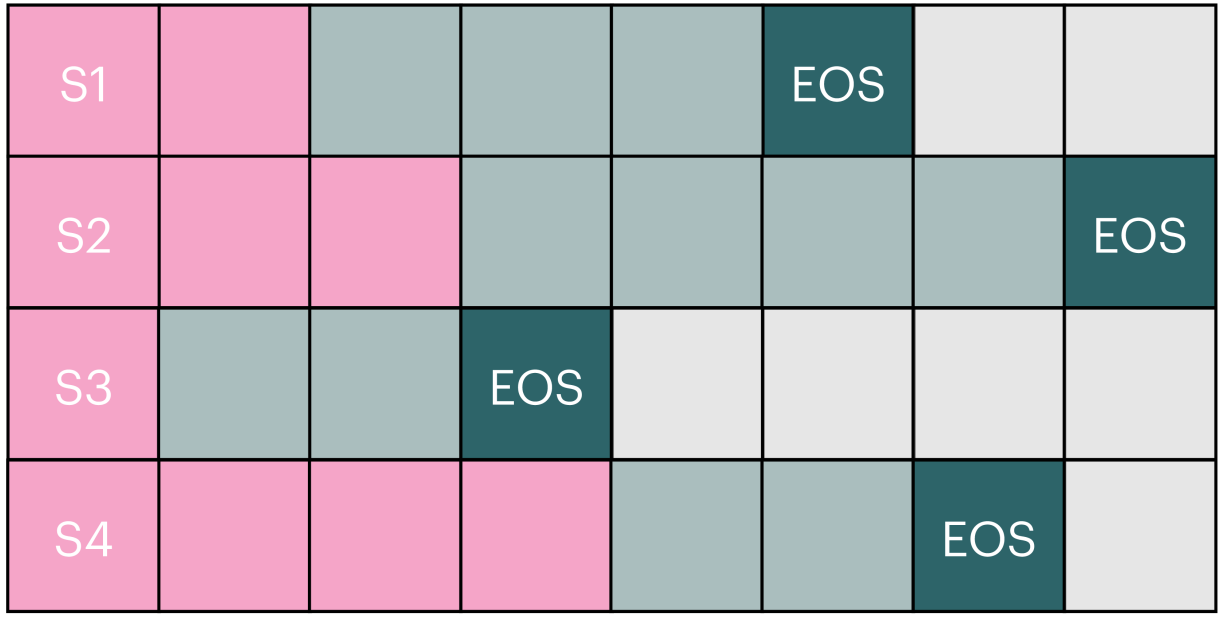


Distributed RL system with prompt and result queues between learner and actor GPUs.

The long-tail generations cause idle GPUs

With **static batching**, each actor generates a whole batch and returns it only when the longest sequence finishes.

- Wasted compute is (max length - mean length) / max length.
- Olmo 3 at a 32K limit: mean generation length 14,628, so up to **54%** of compute would be wasted.
- Agentic RL is worse: one trajectory makes 2 tool calls, another makes 200. Tool calls add additional lag and complexity.



Static batching. Pink: prefill, green: decoding, dark green: EOS, grey: idle until the batch finishes.

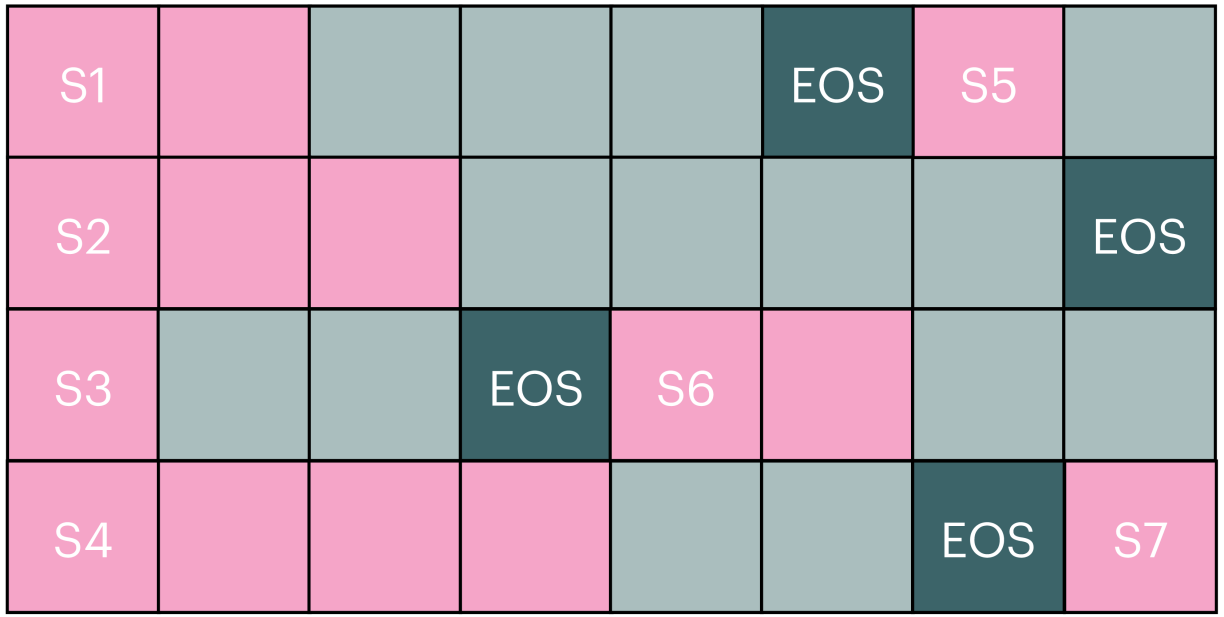
Olmo et al., 2025

Solution: Update inference model weights on the fly

Olmo 3 RL infrastructure	Tokens/s
Olmo 2 baseline	881
+ continuous batching	975
+ better threading	1,358
+ inflight updates	**2,949**

Continuous batching: start a new prompt as soon as a sequence finishes.

Inflight updates: load new weights without pausing generation or dropping the KV cache. The biggest speedup, but tokens in one completion now come from different weights.



Continuous batching backfills finished rows immediately (S5–S7), so no slot sits idle.

Olmo et al., 2025

Inflight updates: PipelineRL

The canonical reference for inflight weight updates is **PipelineRL** (Piché et al., 2025).

arxiv.org/abs/2509.19128

arXiv:2509.19128v2 [cs.LG] 26 Sep 2025

PipelineRL: Faster On-policy Reinforcement Learning for Long Sequence Generation

Alexandre Piché alexandrelpiche@gmail.com
ServiceNow AI Research

Ehsan Kamalloo
ServiceNow AI Research

Rafael Pardinas
ServiceNow AI Research

Xiaoyin Chen
Mila, Université de Montréal

Dzmitry Bahdanau
ServiceNow AI Research *
Mila, McGill University
Canada CIFAR AI Chair

Abstract

Reinforcement Learning (RL) is increasingly utilized to enhance the reasoning capabilities of Large Language Models (LLMs). However, effectively scaling these RL methods presents significant challenges, primarily due to the difficulty in maintaining high AI accelerator utilization without generating stale, off-policy data that harms common RL algorithms. This paper introduces PipelineRL, an approach designed to achieve a superior trade-off between hardware efficiency and data on-policyness for LLM training. PipelineRL employs concurrent asynchronous data generation and model training, distinguished by the novel *in-flight weight updates*. This mechanism allows the LLM generation engine to receive updated model weights with minimal interruption during the generation of token sequences, thereby maximizing both the accelerator utilization and the freshness of training data. Experiments conducted on long-form reasoning tasks using 128 H100 GPUs demonstrate that PipelineRL achieves approximately $\sim 2x$ faster learning compared to conventional RL baselines while maintaining highly on-policy training data. A scalable and modular open-source implementation of PipelineRL is also released as a key contribution.

1 Introduction

Reinforcement Learning (RL) has recently become a popular tool to enhance the reasoning and agentic capabilities of Large Language Models (LLMs) (Guo et al., 2025; Wei et al., 2025). While RL expands the range of training signals one can use to enhance LLMs, this advanced learning paradigm comes with extra challenges, including being particularly hard to effectively scale to more compute. The scaling difficulty arises from the fact that AI accelerators (like GPUs and TPUs) deliver high throughput only when generating sequences at a large batch size. Hence, naively adding more accelerators to an on-policy RL setup brings increasingly diminishing learning speed improvements because the per-accelerator throughput decreases, while the overall generation latency reaches a plateau. The common workaround of generating training data for multiple optimizer steps results in a lag between the currently trained policy and the behavior policy that generates the training data. The lagging off-policy data is known to harm the commonly used effective

*Currently affiliated with another institution.

1

One training step, end to end

```
# 1. Sample completions on the inference engine (pseudocode)
input_ids, completion_mask = sampler.generate(prompts, n=G)

# 2. Score them, then center within each group
rewards = verifier(prompts, input_ids)                  # (B*G,)
advantages = group_center(rewards, G)                   # (B*G,)

# 3. Recompute log-probs on the trainer, this time with gradients
logits = model(input_ids).logits[:, :-1, :]
token_logprobs = logits.log_softmax(-1).gather(
    -1, input_ids[:, 1:].unsqueeze(-1)).squeeze(-1)
seq_logprobs = (token_logprobs * completion_mask[:, 1:]).sum(-1)

# 4. Update, then sync the new weights back to the sampler
(-(advantages * seq_logprobs).mean()).backward()
optimizer.step()
sampler.load_weights(model)
```

Defining the mismatch

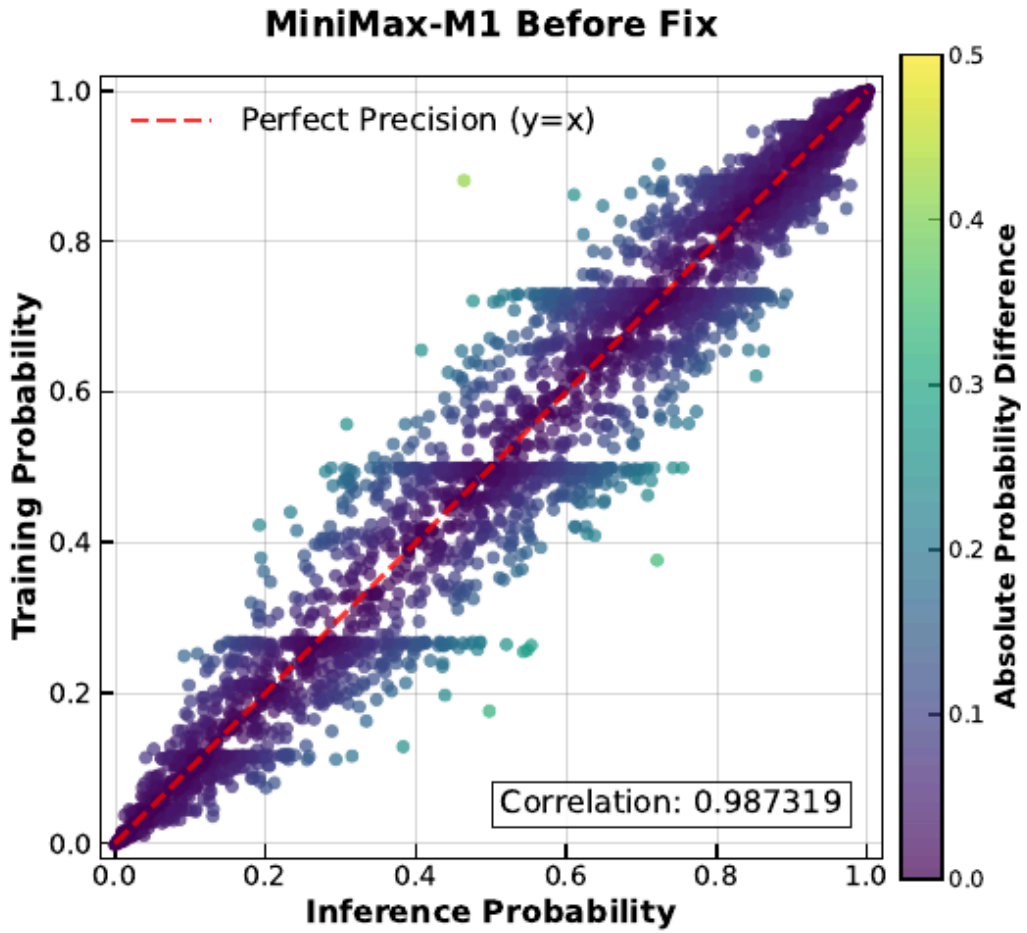
At one position, with prompt x and prefix $a_{<t}$, each system defines a next-token distribution:

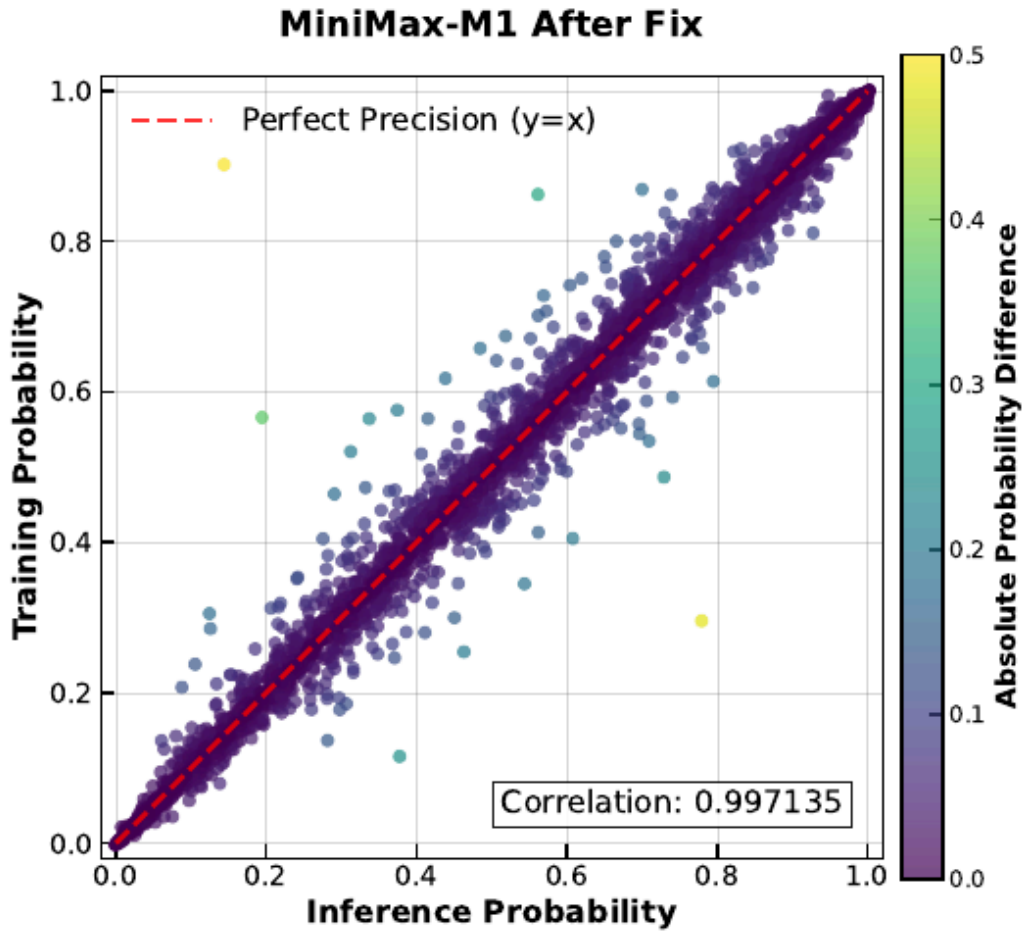
$$p_v = \mathrm{softmax}\big(f_\theta(x, a_{<t})\big)_v \quad \text{learner: current weights, training numerics}$$
$$q_v = \mathrm{softmax}\big(\tilde{f}_{\tilde{\theta}}(x, a_{<t})\big)_v \quad \text{sampler: synced weights, inference numerics}$$

Training-inference mismatch (TIM) is when $q \neq p$. There are two independent sources:

- **Staleness:** $\tilde{\theta} \neq \theta$, because the weights have moved since the last syn (on the trainer side).
- **Numerics:** $\tilde{f} \neq f$, because of different kernels, precision, parallelism, or a quantized KV cache (the cached attention keys and values for the prefix, stored in fewer bits). See Yao et al. and (Marek & Ryabinin, 2026).

Same tokens, different probabilities





Training vs. inference probability for the same tokens in MiniMax-M1, before and after a fix. Agreement improves, but the two systems still differ.

MiniMax, 2025

What training actually estimates

The gradient we want samples from the learner:

$$\nabla_\theta J(\theta) = \mathbb{E}_{a \sim p_\theta}\left[A \sum_{t=1}^{T} \nabla_\theta \log p_\theta(a_t \mid x, a_{<t})\right]$$

The gradient we compute samples from the sampler, but differentiates through the learner:

$$\hat{g} = \mathbb{E}_{a \sim q}\left[A \sum_{t=1}^{T} \nabla_\theta \log p_\theta(a_t \mid x, a_{<t})\right]$$

These differ whenever $q \neq p$, and the gap has two parts: the tokens are drawn from the wrong distribution, and so are the prefixes they follow.

The classical fix: Importance sampling

Reweighting by the sequence ratio recovers the gradient exactly (Marek & Ryabinin, 2026):

$$\nabla_\theta J(\theta) = \mathbb{E}_{a\sim q}\left[w(a)\, A \sum_{t=1}^{T} \boldsymbol{\xi}_{a_t}\right], \qquad w(a) = \frac{p_\theta(a \mid x)}{q(a \mid x)}$$

But w can take very large values on rare tokens, which inflates the variance of the gradient.

Truncated importance sampling (Ionides, 2008) caps the per-token ratio at a constant C, trading a small bias for bounded variance in the policy gradient:

$$w_t = \frac{p(a_t \mid x, a_{<t})}{q(a_t \mid x, a_{<t})}, \qquad \tilde{w}_t = \min(w_t, C)$$

Truncated importance sampling in code

In practice, LLM RL systems apply TIS as a per-token correction weight on the policy-gradient loss:

```
# Shape: (B*G, L)
C = 2.0  # TIS cap

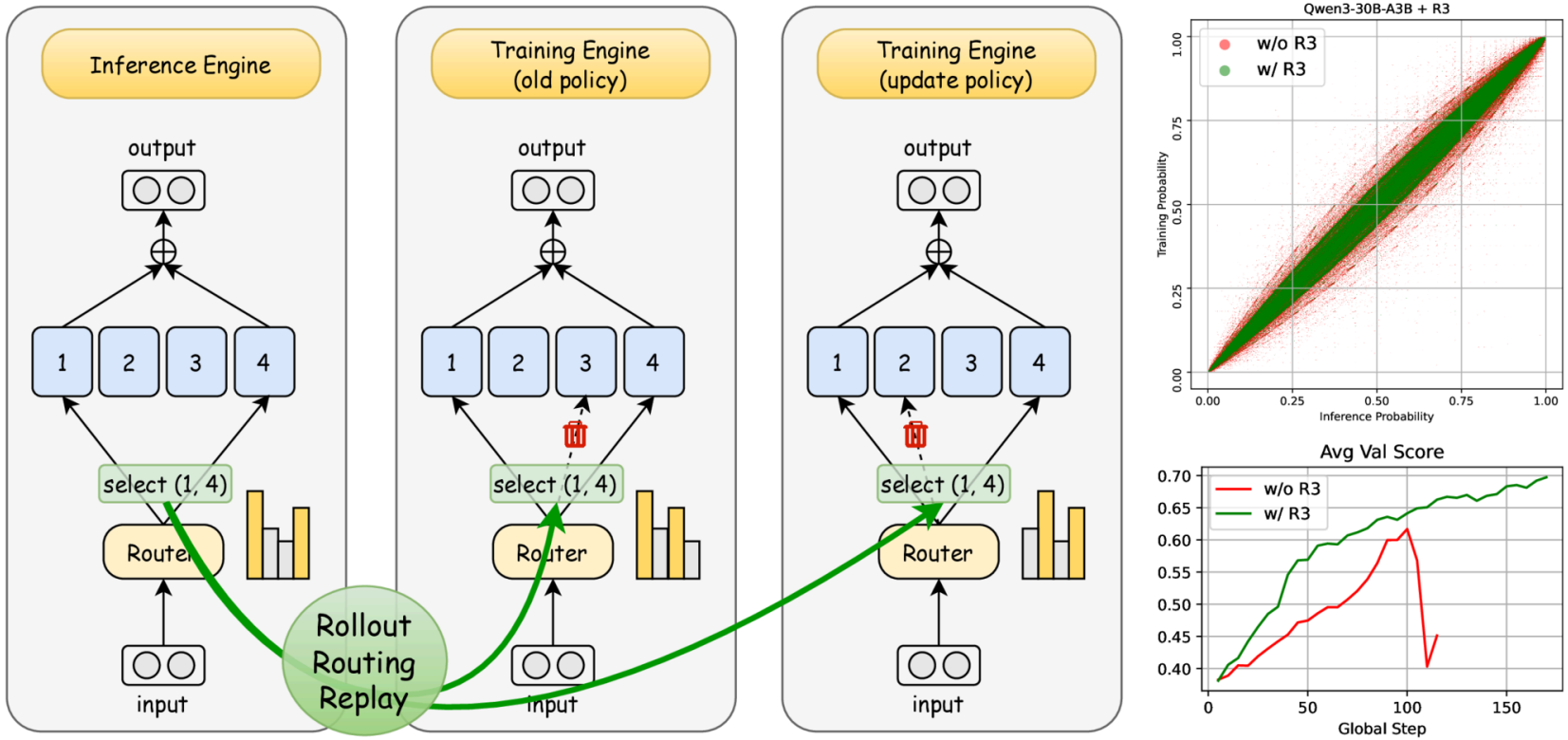
logratio = learner_logprobs - sampler_logprobs
logratio = logratio.clamp(-10.0, 10.0)              # numerical safety
tis_weight = torch.exp(logratio).clamp(max=C)        # one-sided truncation

# Use as a fixed correction weight on the per-token PG loss
per_token_pg_loss = per_token_pg_loss * tis_weight.detach()
```

TIS for the learner–sampler ratio has been adopted across major open-source RL frameworks (VeRL, TRL, OpenRLHF, SkyRL, OAT, and Open Instruct, which uses $C = 2$).

Removing a source of mismatch: Routing replay

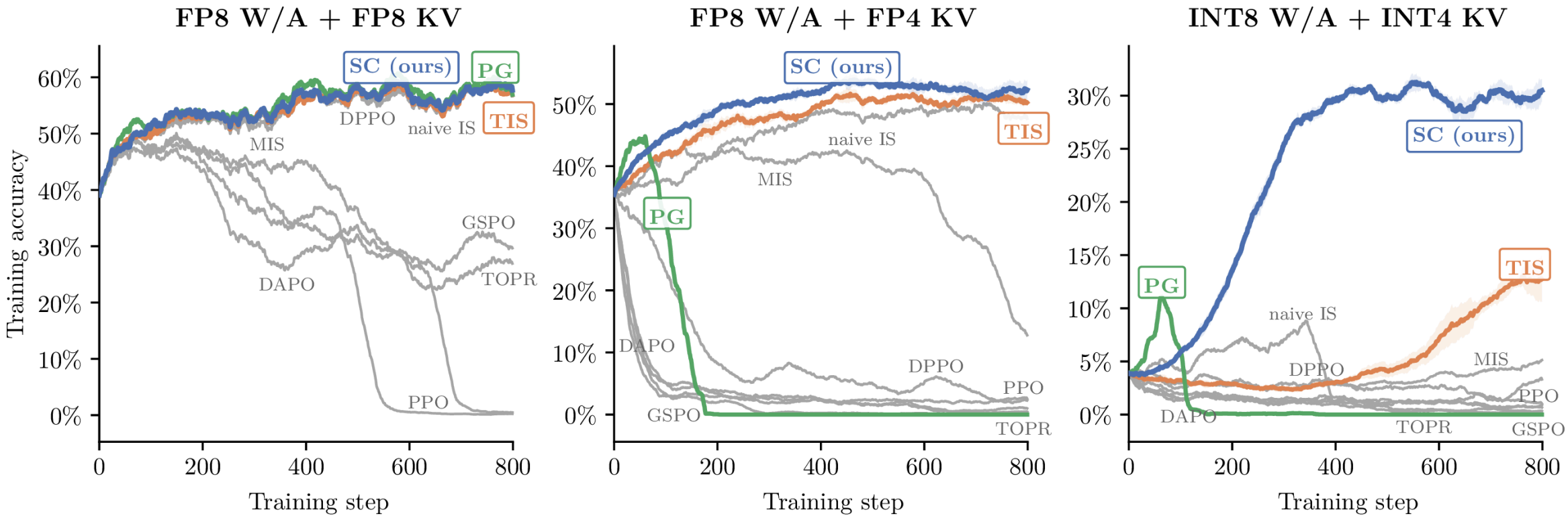
In mixture-of-experts (MoE) models, the trainer's router can pick different experts than the sampler's for the same token. Rollout Routing Replay (R3) records the experts chosen at inference and reuses them in the training forward passes.



Ma et al., 2025

Score centering: managing train-inference mismatch

Train-Inference-Mismatch (TIM) can collapse training



Qwen3-30B-A3B-Base trained with RL on math, with the sampler quantized more and more heavily from left to right. W/A is weights and activations; KV is the KV cache. With an FP8 sampler, plain policy gradient (PG) trains stably. With an FP4 KV cache, PG collapses within 200 steps. With INT8 weights and an INT4 KV cache, score centering (SC) reaches 30% accuracy, TIS 12%, and every other method ends below 5%.

Marek & Ryabinin, 2026

Recall: The policy gradient is a sum of scores

The policy gradient for one completion:

$$\nabla_\theta J(\theta) = \mathbb{E}_{a \sim p_\theta} \left[A \sum_{t=1}^{T} \nabla_\theta \log p_\theta(a_t \mid x, a_{<t}) \right]$$

Call each term the **score** of the token, $\boldsymbol{\xi}_v = \nabla_\theta \log p_\theta(v \mid x, a_{<t})$:

$$\nabla_\theta J(\theta) = \mathbb{E}_{a \sim p_\theta} \left[A \sum_{t=1}^{T} \boldsymbol{\xi}_{a_t} \right]$$

The score has zero mean on-policy

At one position, the expected score under the learner, written as a sum:

$$\mathbb{E}_{v\sim p}[\boldsymbol{\xi}_v] = \sum_{v\in\mathcal{V}} p_v \, \nabla_\theta \log p_v$$

Shorthand: $p_v = p_\theta(v \mid x, a_{<t})$, the learner's probability of token v at this position (θ dropped for brevity); $\mathcal{V}$ is the vocabulary.

Apply the log-derivative identity, $p_v \nabla_\theta \log p_v = \nabla_\theta p_v$:

$$= \sum_{v\in\mathcal{V}} \nabla_\theta p_v$$

Pull the gradient out of the sum. Probabilities sum to one:

$$= \nabla_\theta \sum_{v\in\mathcal{V}} p_v = \nabla_\theta 1 = \mathbf{0}$$

This is why baselines are free: $\mathbb{E}[b(x)\,\boldsymbol{\xi}_{a_t}] = b(x)\cdot\mathbf{0}$. It relies on the expectation and the score using **the same distribution**.

Williams, 1992

Off-policy, the gradient scores are not zero mean

Tokens now come from q, but the score is still computed through p. Define the expected score under the sampler:

$$\bar{\boldsymbol{\xi}} = \mathbb{E}_{v \sim q}[\boldsymbol{\xi}_v] = \sum_{v \in \mathcal{V}} q_v \, \boldsymbol{\xi}_v$$

Shorthand: $q_v = q(v \mid x, a_{<t})$ for the sampler and $p_v = p_\theta(v \mid x, a_{<t})$ for the learner (θ dropped for brevity).

Apply the same log-derivative identity as before:

$$\bar{\boldsymbol{\xi}} = \sum_{v \in \mathcal{V}} \frac{q_v}{p_v} \nabla_\theta p_v$$

The weights are now q_v / p_v instead of 1, so the sum no longer collapses to $\nabla_\theta 1 = \mathbf{0}$. The expected score is zero when $q = p$, but not in general.

Computing the gradient of one token

Start from the estimated gradient, and move the sum outside the expectation:

$$\hat{g} = \mathbb{E}_{a \sim q}\left[\sum_t A \, \boldsymbol{\xi}_{a_t}\right] = \sum_t \mathbb{E}_{a \sim q}\left[A \, \boldsymbol{\xi}_{a_t}\right]$$

Sequences are sampled one token at a time, so split each term into the prefix, then the token and everything after it:

$$\hat{g} = \sum_t \mathbb{E}_{a_{<t} \sim q}\left[\mathbb{E}_{a_{\geq t} \sim q}\left[A \, \boldsymbol{\xi}_{a_t}\right]\right]$$

From here on, fix the prefix (context) and study the inner term. We write it as $\mathbb{E}_q[A \, \boldsymbol{\xi}_{a_t}]$. With the prefix fixed, the token a_t and the advantage A are both still random variables.

Which pieces of the gradient contribute to learning

Use the covariance identity, $\mathbb{E}[XY] = \mathbb{E}[X]\,\mathbb{E}[Y] + \mathrm{Cov}(X, Y)$, starting on-policy, the expectation of the gradient:

$$\begin{aligned}\mathbb{E}_p[A\,\boldsymbol{\xi}_{a_t}] &= \mathbb{E}_p[A]\,\mathbb{E}_p[\boldsymbol{\xi}_{a_t}] + \mathrm{Cov}_p(A, \boldsymbol{\xi}_{a_t}) \\ &= \mathbb{E}_p[A] \cdot \mathbf{0} + \mathrm{Cov}_p(A, \boldsymbol{\xi}_{a_t}) \\ &= \mathrm{Cov}_p(A, \boldsymbol{\xi}_{a_t})\end{aligned}$$

Now the same identity under the sampler, where $\mathbb{E}_q[\boldsymbol{\xi}_{a_t}] = \bar{\boldsymbol{\xi}}$:

$$\mathbb{E}_q[A\,\boldsymbol{\xi}_{a_t}] = \mathbb{E}_q[A]\,\mathbb{E}_q[\boldsymbol{\xi}_{a_t}] + \mathrm{Cov}_q(A, \boldsymbol{\xi}_{a_t}) = \underbrace{\mathbb{E}_q[A]\,\bar{\boldsymbol{\xi}}}_{\text{drift}} + \underbrace{\mathrm{Cov}_q(A, \boldsymbol{\xi}_{a_t})}_{\text{signal}}$$

Only the covariance sees which tokens led to which outcomes. The drift depends on the advantages only through their mean – not per-token attribution.

What the drift does

The drift, $\mathbb{E}_q[A]\,\bar{\boldsymbol{\xi}}$, depends on the advantages only through their mean, so it moves the weights the same way regardless of which token was sampled.

We don’t want that: in practice, it can pull the learner’s weights toward a stale and/or quantized sampler, causing potentially compounding instabilities.

Marek & Ryabinin, 2026

Score centering: A baseline for the score

Baselines center the reward. Score centering centers the per-token log-prob gradients (called "score" in statistics), subtracting its expected value under the sampler:

$$\tilde{\boldsymbol{\xi}}_{a_t} = \boldsymbol{\xi}_{a_t} - \bar{\boldsymbol{\xi}}$$

Recall: the score of token v is $\boldsymbol{\xi}_v = \nabla_\theta \log p_\theta(v \mid x, a_{<t})$, the gradient of its log-probability under the learner.

$\bar{\boldsymbol{\xi}}$ is a sum over the whole vocabulary at a fixed prefix, so it is a constant with respect to the sampled token:

$$\mathbb{E}_q[\tilde{\boldsymbol{\xi}}_{a_t}] = \mathbb{E}_q[\boldsymbol{\xi}_{a_t} - \bar{\boldsymbol{\xi}}] = \mathbb{E}_q[\boldsymbol{\xi}_{a_t}] - \bar{\boldsymbol{\xi}} = \bar{\boldsymbol{\xi}} - \bar{\boldsymbol{\xi}} = \mathbf{0}$$

The centered score has zero mean under the sampler! which restores the property that made on-policy learning attractive. When $q = p$ (on-policy), $\bar{\boldsymbol{\xi}} = \mathbf{0}$ and the correction does nothing.

The corrected, per-token gradient

With the centered score, the policy-gradient estimate becomes:

$$\hat{g}_{\text{centered}} = \mathbb{E}_{a\sim q}\left[A\sum_{t=1}^{T}\tilde{\boldsymbol{\xi}}_{a_t}\right] = \sum_{t}\mathbb{E}_{a_{<t}\sim q}\left[\mathbb{E}_q[A\,\tilde{\boldsymbol{\xi}}_{a_t}]\right]$$

Apply the covariance identity to the inner, per-token term:

$$\begin{aligned}
\mathbb{E}_q[A\,\tilde{\boldsymbol{\xi}}_{a_t}] &= \mathbb{E}_q[A]\,\mathbb{E}_q[\tilde{\boldsymbol{\xi}}_{a_t}] + \mathrm{Cov}_q(A, \tilde{\boldsymbol{\xi}}_{a_t}) \\
&= \mathbb{E}_q[A]\,\mathbb{E}_q[\boldsymbol{\xi}_{a_t} - \bar{\boldsymbol{\xi}}] + \mathrm{Cov}_q(A, \boldsymbol{\xi}_{a_t} - \bar{\boldsymbol{\xi}}) \\
&= \mathbb{E}_q[A]\,\left(\mathbb{E}_q[\boldsymbol{\xi}_{a_t}] - \bar{\boldsymbol{\xi}}\right) + \mathrm{Cov}_q(A, \boldsymbol{\xi}_{a_t} - \bar{\boldsymbol{\xi}}) \\
&= \mathbb{E}_q[A]\,\left(\bar{\boldsymbol{\xi}} - \bar{\boldsymbol{\xi}}\right) + \mathrm{Cov}_q(A, \boldsymbol{\xi}_{a_t} - \bar{\boldsymbol{\xi}}) \\
&= \mathrm{Cov}_q(A, \boldsymbol{\xi}_{a_t})
\end{aligned}$$

The last step uses that a constant does not change a covariance: $\mathrm{Cov}(X, Y - c) = \mathrm{Cov}(X, Y)$.

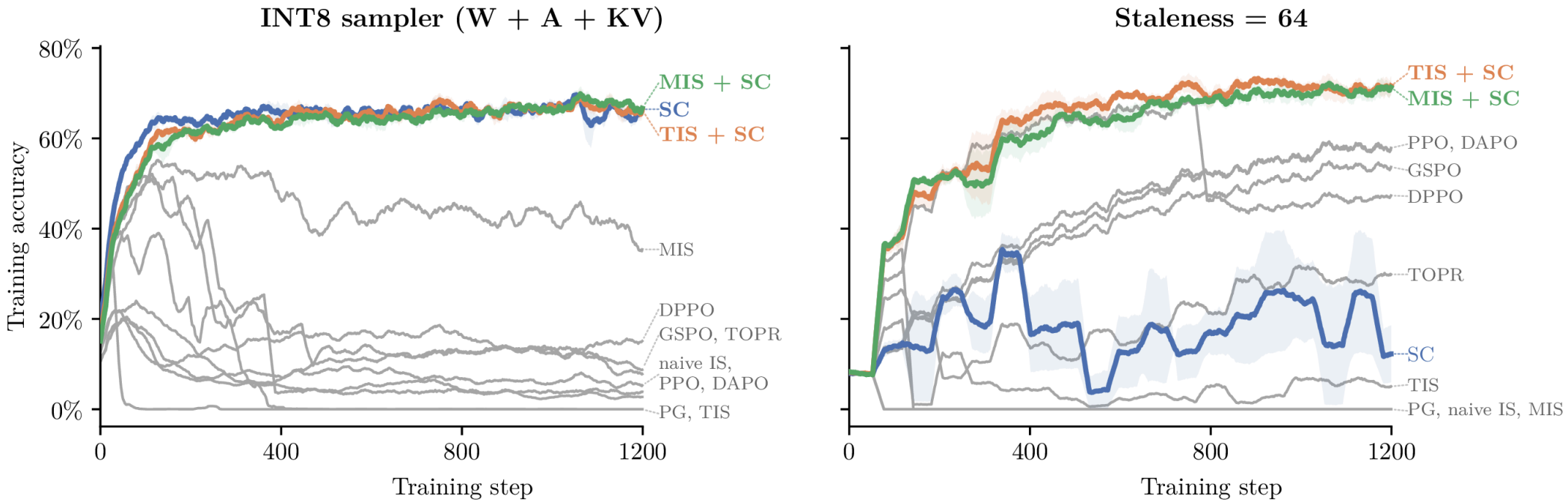
Score centering complements other interventions like importance sampling

- **Score centering** removes the drift, the pull toward the sampler that carries no learning signal.
- It does not make training on-policy: the signal is still $\mathrm{Cov}_q(A, \boldsymbol{\xi}_{a_t})$, measured under the sampler, not the learner.
- **Importance sampling** (e.g., TIS) reweights samples toward the learner, but clipping the ratios lets some drift back in.
- They fix different problems and are composable. The score-centering paper recommends using both!

Marek & Ryabinin, 2026

Quantization vs. staleness

- **Quantization:** The sampler has the learner's weights, just lower precision. Score centering alone is enough.
- **Staleness:** The learner moves away from the sampler, so score centering needs importance sampling too. Taking several gradient steps per batch causes the same kind of staleness.



Qwen3-0.6B on Countdown with an INT8 sampler (left) and a sampler updated only every 64 steps (right).

Marek & Ryabinin, 2026

Recent algorithms differ mostly in how they clip the log-ratio

Method	Level	Stat	Low	High	Outside	Sign	Implementation notes
Naive IS (Williams, 1992)	token	ratio	0	∞	-	both	—
TIS^{V} (Yao et al., 2025) CISPO (MiniMax, 2025)	token	ratio	0	2.0	clip	both	CISPO's dynamic sampling and length penalty omitted.
MIS (IcePop)^{P} (Ling Team, 2025)	token	ratio	0.5	5.0	drop	both	Separate PPO surrogate between policy versions omitted.
PPO^{P,V} (Schulman et al., 2017)	token	ratio	0.8	1.2	drop	ppo	Dual clipping at 3 (`verl` default).
DAPO^{P,V} (Yu et al., 2025)	token	ratio	0.8	1.28	drop	ppo	Dual clipping at 3; dynamic sampling and overlong reward penalty omitted.
GSPO^{P,V} (Zheng et al., 2025)	seq	ratio	0.9997	1.0004	drop	ppo	Token gradients summed rather than response-averaged; added dual clipping at 3.
TOPR^{P} (Le Roux et al., 2025)	seq	ratio	0	1.0	clip	neg	—
DPPO^{P} (Qi et al., 2026)	token	tv	0	0.2	drop	ppo	No importance-ratio clipping.

Importance-sampling corrections as a grid: per-token or per-sequence ratio, the clipping range (Low to High), what happens outside it, and which advantage signs are corrected.

Marek & Ryabinin, 2026

Score-centering in the wild (aka industry)

StableReinforce and GRPO

(Supposedly) Many labs are supposedly using **StableReinforce**, the loss Tinker added, rather than GRPO. This is unconfirmed, but the two are related.

The group advantages are nearly the same. GRPO normalizes each reward by the group mean and standard deviation:

$$A_i^{\text{GRPO}} = \frac{r_i - \text{mean}(r_1, \ldots, r_G)}{\text{std}(r_1, \ldots, r_G)}$$

REINFORCE with a group baseline is usually RLOO, which leaves each sample out of its own baseline:

$$A_i^{\text{RLOO}} = r_i - \frac{1}{G-1}\sum_{j \neq i} r_j = \frac{G}{G-1}\left(r_i - \text{mean}(r_1, \ldots, r_G)\right)$$

Without the std, the two differ only by a constant scale. The bigger difference is the loss: GRPO usually wraps the advantage in PPO's clipped ratio, while REINFORCE uses the plain log-prob loss.

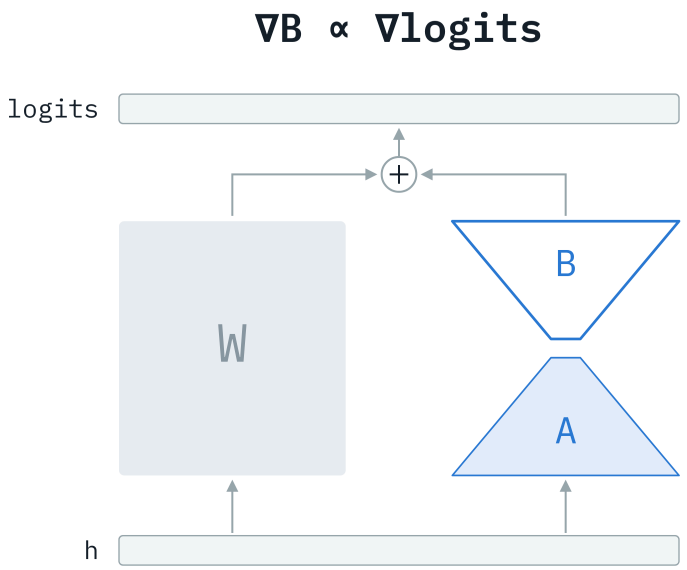
Reverse-engineering a loss from its gradients

Tinker does not expose the loss or the "stability info," but it does let you take a step and download the weights.

Step 1: Recover the raw gradient. Tinker has no SGD, so use Adam with $\epsilon = 10{,}000$. This dominates Adam's second moment:

$$\Delta w \approx -\frac{\mathrm{lr} \cdot g}{\epsilon}$$

Step 2: Read it in logit space. Train a rank-1 LoRA adapter on the unembedding of Qwen3.5-4B. The gradient with respect to B equals the gradient with respect to the logits, up to scale.



A rank-1 LoRA adapter added to the unembedding matrix, so the gradient on B is proportional to the logit gradient. From Marek's thread.

Marek's thread

Why the rank-1 LoRA trick reads out the logit gradient

With hidden state $h \in \mathbb{R}^d$ and unembedding $W \in \mathbb{R}^{|\mathcal{V}| \times d}$, the logits are $z = Wh$. A rank-1 LoRA adds BA, with $A \in \mathbb{R}^{1 \times d}$ and $B \in \mathbb{R}^{|\mathcal{V}| \times 1}$:

$$z = (W + BA)\, h = Wh + B\,(Ah)$$

Ah is a single number, so write $c = Ah$. The adapter adds c times the vector B to the logits:

$$z = Wh + c\,B$$

By the chain rule, with $g = \partial L / \partial z$ the gradient on the logits:

$$\frac{\partial L}{\partial B} = \frac{\partial L}{\partial z}\,\frac{\partial z}{\partial B} = c\,g$$

So the gradient on B is the logit gradient, scaled by c. LoRA starts at $B = 0$, so the forward pass is unchanged. With Adam's $\epsilon = 10{,}000$, $\Delta B \approx -\mathrm{lr} \cdot c\,g/\epsilon$, so the change in B after one step shows g.

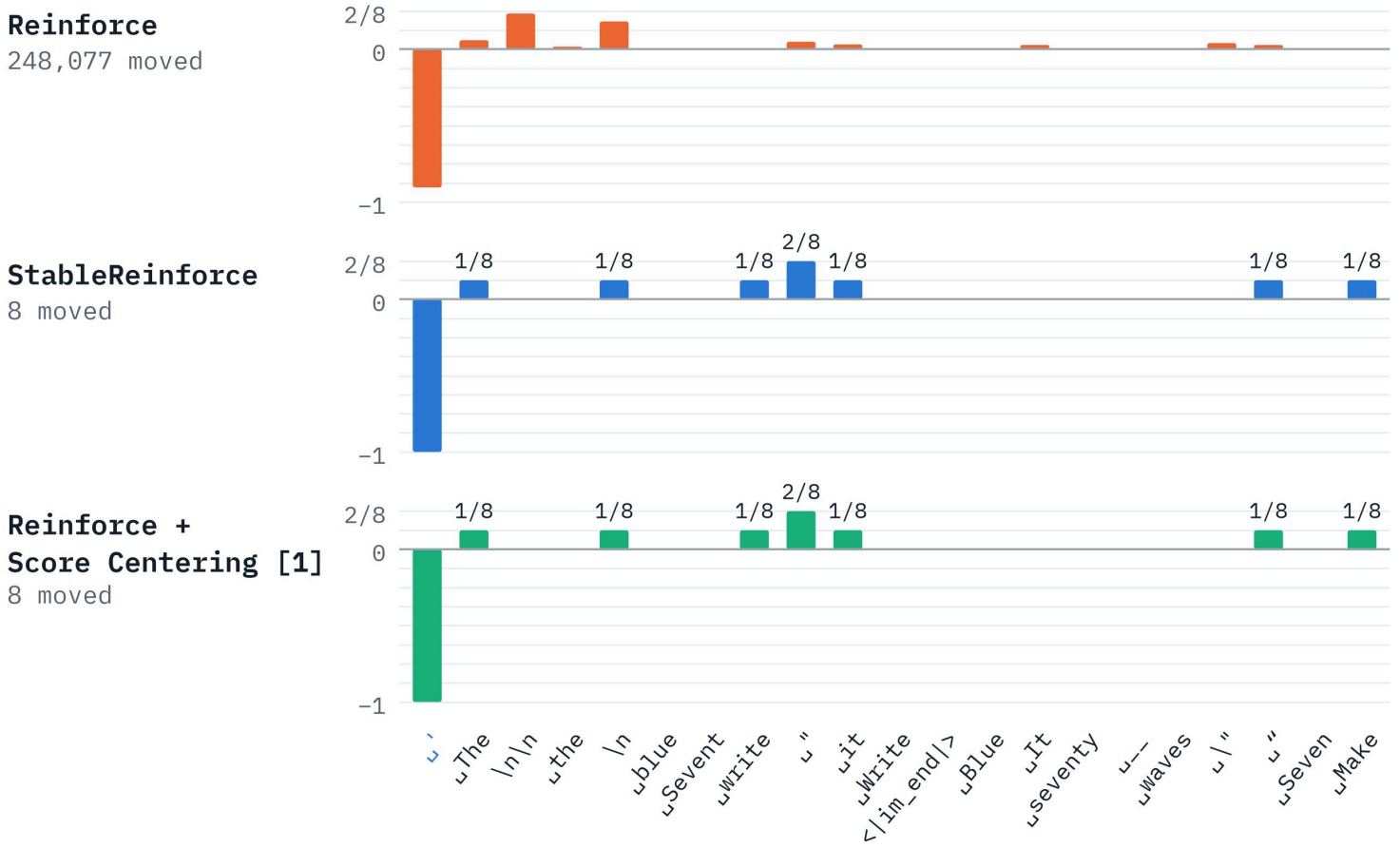
Marek's thread

Comparing logits/gradients in Tinker

For a single-token update:

- **Reinforce** puts a nonzero gradient on all logits: 248,077 of them move.
- **StableReinforce** moves only 8.

The bars come in multiples of $1/8$, and the third row, Reinforce with score centering, is identical.



[1] Martin Marek and Max Ryabinin. *Score Centering Stabilizes Off-policy Reinforcement Learning.* arXiv:2609.20807

Logit gradients for a single-token update under Reinforce, StableReinforce, and Reinforce with score centering. From Marek's thread.

Marek's thread

Tinker uses a different score-centering approximation (token-based sample)

The plot shows gradients with respect to the logits z. For a softmax, the gradient of $\log p_v$ is the one-hot vector e_v minus the probabilities:

$$\boldsymbol{\xi}_v = \nabla_z \log p_v = e_v - p$$

Estimate $\bar{\boldsymbol{\xi}}$ by drawing 8 tokens from the sampler, $v_1, \ldots, v_8 \sim q$:

$$\bar{\boldsymbol{\xi}} \approx \frac{1}{8}\sum_{i=1}^{8} \boldsymbol{\xi}_{v_i} = \frac{1}{8}\sum_{i=1}^{8} (e_{v_i} - p) = \frac{1}{8}\sum_{i=1}^{8} e_{v_i} - p$$

Subtract it from the score of the sampled token. The dense $-p$ cancels exactly:

$$\boldsymbol{\xi}_{a_t} - \bar{\boldsymbol{\xi}} \approx (e_{a_t} - p) - \left(\frac{1}{8}\sum_{i=1}^{8} e_{v_i} - p\right) = e_{a_t} - \frac{1}{8}\sum_{i=1}^{8} e_{v_i}$$

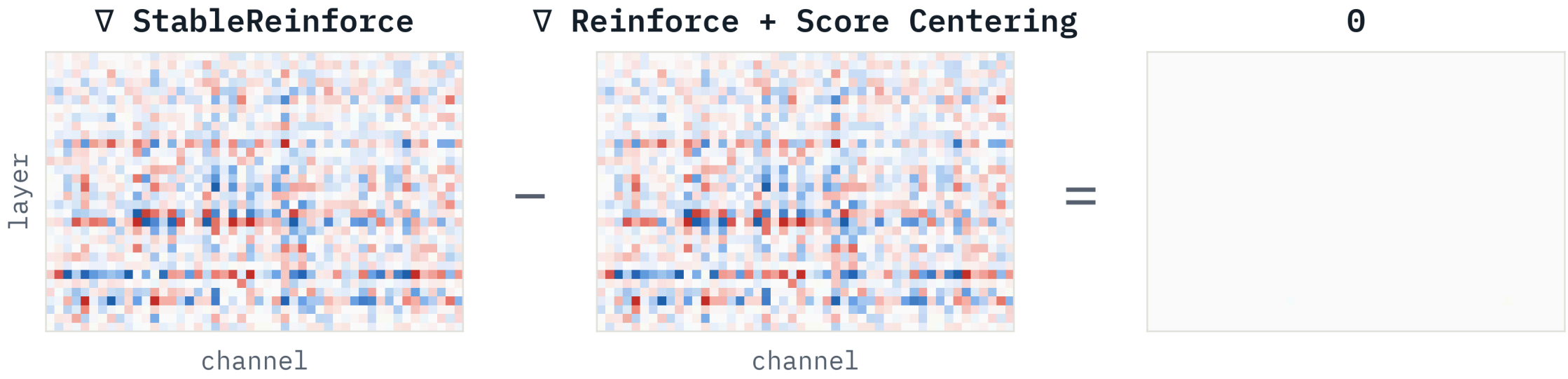
Flip the signs for a loss and this is the figure: -1 on the sampled token, $+1/8$ per draw.

Marek's thread

The gradients match exactly!

Hypothesis: StableReinforce is REINFORCE plus score centering, with the expected score computed over 8 randomly sampled tokens.

Test: Implement that as a custom Tinker loss, take an optimizer step, and compare. The gradients are within numeric precision across all layers.



The StableReinforce gradient minus the Reinforce with score centering gradient is zero across layers. From Marek's thread.

Marek's thread

Score centering in a recent model

Ripple-65B Preview, a 65B open-weight model from Ricursive (built on a lot of Olmo 3 infra!), names score centering in its post-training:

> *"Our mixed RL recipe uses truncated importance sampling, score centering, and linear length penalty with high-quality long-horizon synthetic environments."*

Wrapping up

Takeaways

RL in industry is mostly a systems/engineering problem.

RL on language models has shifted remarkably little from some of the earliest deep RL algorithms.

Even since DeepSeek R1 / GRPO, algorithmic innovation is very minimal.

RL in industry is often more brute force than clever/insightful.

Thank you

I have a lot of content expanding on this stuff!

- **Course:** 14 lectures, Q&As, and guest episodes at rlhfbook.com/course
- **Book:** rlhfbook.com, also in print from Manning
- **Newsletter:** interconnects.ai

nathan@natolambert.com



References (1/2)

DeepSeek-AI. "*DeepSeek-V3 Technical Report*." 2025. [link]

Guo, D., Yang, D., Zhang, H., and others. "*DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning*." *Nature*, 2025.

Ionides, E. "*Truncated importance sampling*." *Journal of Computational and Graphical Statistics*, 2008.

Ivison, H., Yin, J., Shao, R., Xiao, T., Lambert, N., et al. "*TMax: A Simple Recipe for Terminal Agents*." *arXiv preprint arXiv:2606.23321*, 2026. [link]

Ma, W., Zhang, H., Zhao, L., Song, Y., Wang, Y., et al. "*Stabilizing MoE Reinforcement Learning by Aligning Training and Inference Routers*." *arXiv preprint arXiv:2510.11370*, 2025. [link]

Marek, M., and Ryabinin, M. "*Score Centering Stabilizes Off-policy Reinforcement Learning*." 2026. [link]

MiniMax. "*MiniMax-M1: Scaling Test-Time Compute Efficiently with Lightning Attention*." 2025. [link]

Noukhovitch, M., Huang, S., Xhonneux, S., Hosseini, A., Agarwal, R., et al. "*Asynchronous RLHF: Faster and More Efficient Off-Policy RL for Language Models*." *International Conference on Learning Representations (ICLR)*, 2025.

Olmo, T., Ettinger, A., Bertsch, A., Kuehl, B., Graham, D., et al. "*Olmo 3*." 2025. [link]

Piché, A., Kamalloo, E., Pardinas, R., Chen, X., and Bahdanau, D. "*PipelineRL: Faster On-policy Reinforcement Learning for Long Sequence Generation*." *arXiv preprint arXiv:2509.19128*, 2025. [link]

References (2/2)

Williams, R. "*Simple statistical gradient-following algorithms for connectionist reinforcement learning.*" *Machine learning*, 1992.

Extra slides

More clues in Stable Reinforce

Tinker's docs describe the behavior, but not the main loss:

> *"In our internal post-training, this loss function has led to extremely stable training, even under highly off-policy async RL and low precision numerics."*

They do reveal a clue: sampling must record extra information, through `record_stability_info=True`.

Frontier losses are rarely published, so public ideas are rarely confirmed. Here, Martin Marek, the score centering paper's first author, tried to recover the loss from its gradients alone.

Tinker docs

Measuring the mismatch at one token

The ratio between the two systems for a sampled token, and its log:

$$w_t = \frac{p(a_t \mid x, a_{<t})}{q(a_t \mid x, a_{<t})}, \qquad \delta_t = \log w_t = \log p(a_t \mid x, a_{<t}) - \log q(a_t \mid x, a_{<t})$$

Tokens are sampled from q, so take the expectation under q:

$$\mathbb{E}_{a_t \sim q}[\delta_t] = \sum_{v \in \mathcal{V}} q_v \left(\log p_v - \log q_v\right)$$

Flip the sign to recognize the KL divergence:

$$\mathbb{E}_{a_t \sim q}[\delta_t] = -\sum_{v \in \mathcal{V}} q_v \log \frac{q_v}{p_v} = -\mathrm{KL}(q_t \,\|\, p_t)$$

If the systems agree, every δ_t is zero. Otherwise, log-ratios are negative on average.

Mismatch adds up over a completion

The ratio for a whole completion is the product of the token ratios:

$$w(a) = \frac{p_\theta(a \mid x)}{q(a \mid x)} = \prod_{t=1}^{T} w_t, \qquad \log w(a) = \sum_{t=1}^{T} \delta_t$$

Take the expectation under q, and apply the previous slide at each position:

$$\mathbb{E}_{a\sim q}[\log w(a)] = \sum_{t=1}^{T} \mathbb{E}_{a_{<t}\sim q}\Big[\mathbb{E}_{a_t\sim q}[\delta_t]\Big] = -\sum_{t=1}^{T} \mathbb{E}_{a_{<t}\sim q}\Big[\mathrm{KL}(q_t \,\|\, p_t)\Big]$$

This sum is the KL between the two sequence distributions, $\mathrm{KL}\big(q(a \mid x) \,\|\, p_\theta(a \mid x)\big)$, so it grows with length.

For illustration, a small per-token KL of 0.001 over $10{,}000$ tokens gives a sequence KL of 10 nats. A typical weight is then near $e^{-10} \approx 5 \times 10^{-5}$.

What is the score?

In statistics, the **score** is the gradient of the log-likelihood with respect to the parameters. For one token v at one position:

$$\boldsymbol{\xi}_v = \nabla_\theta \log p_\theta(v \mid x, a_{<t})$$

- It is a vector the size of the parameters: the direction that makes token v more likely.
- The policy gradient is an advantage-weighted sum of the scores of the sampled tokens:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[A \sum_{t=1}^{T} \boldsymbol{\xi}_{a_t}\right]$$

Why the drift compounds

Distilling toward a **fixed** teacher is harmless. It converges to the best fit of that teacher.

In online RL, the teacher moves with the student:

- The sampler is a biased copy of the learner, e.g., quantized or stale.
- Each step pulls the learner toward the sampler's errors.
- The new weights are synced back to the sampler, carrying those errors with them.
- The next sampler is biased again, so the error compounds instead of converging.
- The bias does not average out with more samples, since $\bar{\xi}$ is fixed at each prefix. It needs a direct correction.

Marek & Ryabinin, 2026

Do group baselines already fix this?

Not fully. Group centering makes advantages sum to zero **over the rollouts of a prompt**:

$$\frac{1}{G}\sum_{i=1}^{G}(r_i - \bar{r}) = 0$$

But drift arises **at each prefix**, and $\mathbb{E}_q[A \mid x, a_{<t}]$ is generally not zero there:

- A prefix likely to reach a correct answer has positive expected advantage.
- A prefix that already contains a mistake has negative expected advantage.

Group centering shrinks drift relative to all-positive rewards, and it was the most stable reward setting in the paper's online experiments. It does not eliminate drift.

Marek & Ryabinin, 2026

From the update to a loss

We want a scalar whose gradient is $A\,(\boldsymbol{\xi}_{a_t} - \bar{\boldsymbol{\xi}})$.

The first piece is already a gradient, $\boldsymbol{\xi}_{a_t} = \nabla_\theta \log p_{a_t}$. For the second:

$$\bar{\boldsymbol{\xi}} = \sum_{v \in \mathcal{V}} q_v \, \nabla_\theta \log p_v$$

The sampler's probabilities are logged values, so we mark them as constants with a stop-gradient and pull the gradient out:

$$\bar{\boldsymbol{\xi}} = \nabla_\theta \sum_{v \in \mathcal{V}} \mathrm{sg}[q_v] \log p_v$$

Combine the two pieces, and multiply by $-A$:

$$L = -A \left(\log p_{a_t} - \sum_{v \in \mathcal{V}} \mathrm{sg}[q_v] \log p_v \right), \qquad -\nabla_\theta L = A\,(\boldsymbol{\xi}_{a_t} - \bar{\boldsymbol{\xi}}) = A\,\tilde{\boldsymbol{\xi}}_{a_t}$$

Marek & Ryabinin, 2026

Score centering in code

The correction is one extra term in the REINFORCE loss from earlier:

```
# Learner: log-probs over the whole vocabulary, with gradients
learner_vocab_logprobs = logits.log_softmax(-1)              # (B*G, L, V)
learner_logprobs = learner_vocab_logprobs.gather(
    -1, tokens.unsqueeze(-1)).squeeze(-1)                    # (B*G, L)

# Sampler: logged at generation time, so it has no gradient path
sampler_probs = sampler_vocab_logprobs.exp()                 # (B*G, L, V)

# The gradient of `centering` is the expected score, s-bar
centering = (sampler_probs * learner_vocab_logprobs).sum(-1) # (B*G, L)

# advantages: (B*G, 1), broadcast to every token
per_token_loss = -advantages * (learner_logprobs - centering)
loss = ((per_token_loss * mask).sum(1) / mask.sum(1)).mean()
```

No ratios, no masks, and no clipping. On-policy, the gradient of `centering` is zero.

The full distribution is too expensive

Computing $\bar{\xi}$ exactly needs the sampler's next-token distribution at every position:

- Qwen3's vocabulary has 152K tokens, so one position in fp32 takes about 608KB.
- A batch of 1024 rollouts with 32K tokens each would take about 20TB.

Instead, log only the sampler's **head** H, its k most likely tokens at that position, with $k = 128$. Model the **tail** with the learner, rescaled to carry the sampler's missing mass:

$$\hat{q}_v = \begin{cases} q_v & v \in H \\ \rho\, p_v & v \notin H \end{cases} \qquad \rho = \frac{1 - \sum_{v \in H} q_v}{1 - \sum_{v \in H} p_v}$$

Marek & Ryabinin, 2026

The tail cancels

Substitute $\hat{q}$ into the expected score, and split the sum into head and tail:

$$\bar{\boldsymbol{\xi}} \approx \sum_{v} \hat{q}_v \, \boldsymbol{\xi}_v = \sum_{v \in H} q_v \, \boldsymbol{\xi}_v + \rho \sum_{v \notin H} p_v \, \boldsymbol{\xi}_v$$

The zero-mean identity splits the same way, which rewrites the tail in terms of the head:

$$\mathbf{0} = \sum_{v} p_v \, \boldsymbol{\xi}_v = \sum_{v \in H} p_v \, \boldsymbol{\xi}_v + \sum_{v \notin H} p_v \, \boldsymbol{\xi}_v \quad \Rightarrow \quad \sum_{v \notin H} p_v \, \boldsymbol{\xi}_v = - \sum_{v \in H} p_v \, \boldsymbol{\xi}_v$$

Substitute, and collect the terms. Only the k head tokens remain:

$$\bar{\boldsymbol{\xi}} \approx \sum_{v \in H} q_v \, \boldsymbol{\xi}_v - \rho \sum_{v \in H} p_v \, \boldsymbol{\xi}_v = \sum_{v \in H} (q_v - \rho \, p_v) \, \boldsymbol{\xi}_v$$

The paper reports that $k = 128$ and $k = 32$ both perform on par with the full vocabulary, at under 1% wall-clock overhead for $k = 128$.

Marek & Ryabinin, 2026

Top-k score centering in code

```
# Logged during generation: sampler_topk_logprobs, sampler_topk_ids (B*G, L, k)
learner_logprobs = learner_vocab_logprobs.gather(-1, tokens[..., None])[..., 0]
head_logprobs = learner_vocab_logprobs.gather(-1, sampler_topk_ids)  # (B*G, L, k)
learner_head_probs = head_logprobs.exp()
sampler_head_probs = sampler_topk_logprobs.exp()

# rho: the ratio of the sampler's tail mass to the learner's tail mass
sampler_tail = (1 - sampler_head_probs.sum(-1)).clamp_min(1e-6)
learner_tail = (1 - learner_head_probs.sum(-1)).clamp_min(1e-6)
rho = sampler_tail / learner_tail                                    # (B*G, L)

# detach() is required: coef contains the learner's own probabilities
coef = (sampler_head_probs - rho[..., None] * learner_head_probs).detach()
centering = (coef * head_logprobs).sum(-1)                           # (B*G, L)

per_token_loss = -advantages * (learner_logprobs - centering)
```

Marek & Ryabinin, 2026