LLM Post-Training

What it adds to a base model, how it works, and how to do it on Aurora

Filippo Simini fsimini@anl.gov

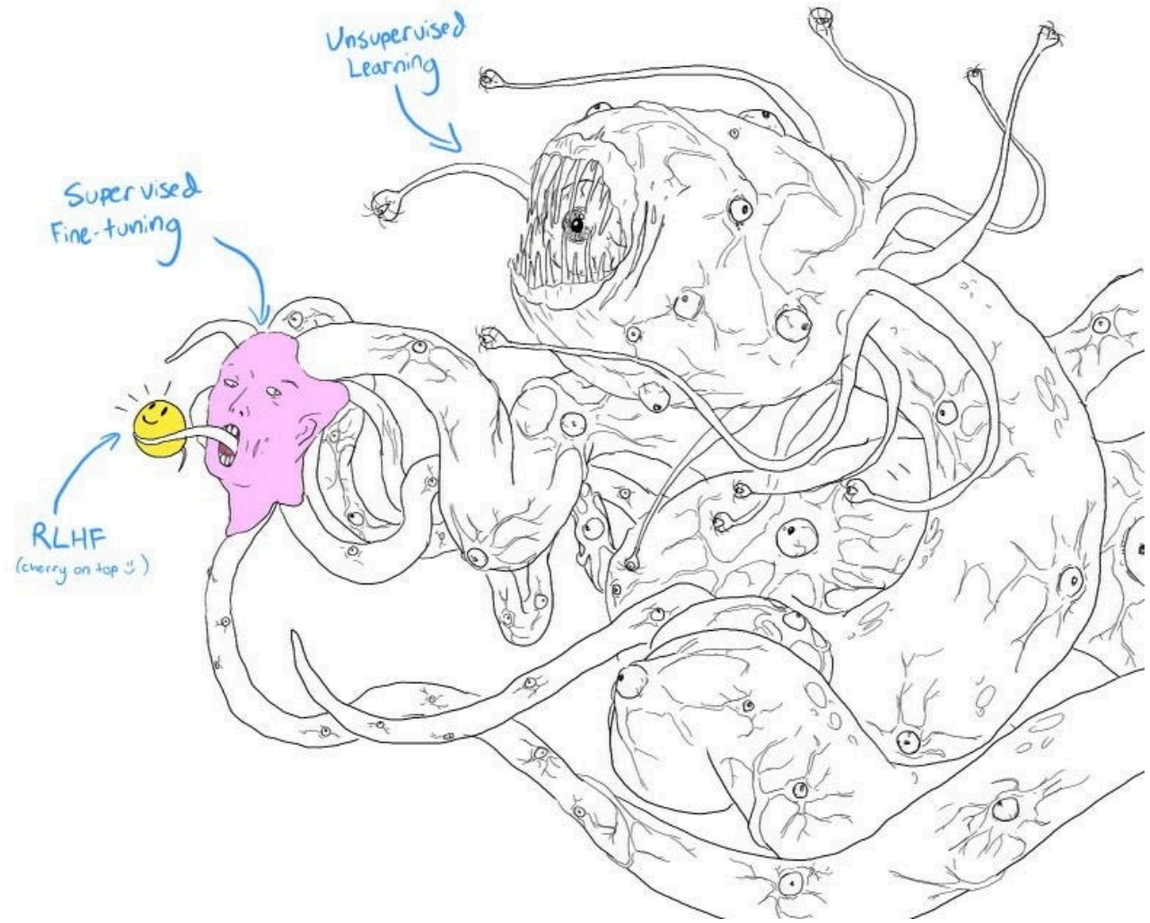
August 3, 2026

Outline

- Part 1: **What is post-training (PT)**
  - Capabilities gained by post-training a LLM
  - Overview of post-training recipes of popular open LLMs, and PT libraries on Aurora
- Part 2: **Supervised Fine-Tuning (SFT)**
  - How SFT differs from pre-training; what SFT accomplishes; SFT on Aurora
  - Hands on: SFT for instruction following
- Part 3: **Reinforcement Learning (RL) for LLMs**
  - How RL differs from pre-training; RL flavors and goals; RL on Aurora
  - Hands on: RL with Verifiable Reward (RLVR)
- Part 4: **Open questions**

LLM training stages

1. **Pre-training**
    - Learns grammar, semantics, knowledge & facts
    - Capabilities: text generation/completion
2. **Mid-training**
    - Long context extension, deeper knowledge in broad domains (e.g. math, coding, logic)
3. **Post-training**
    - Domain knowledge, instruction following, alignment, reasoning, tool use



https://knowyourmeme.com/sensitive/memes/shoggoth-with-smiley-face-artificial-intelligence

Capabilities that Post-training adds to a Base LLM

INSTRUCTION FOLLOWING

- Understands complex, multi-step prompts
- Follows explicit format & system constraints
- Adapts tone, style, and target persona

HUMAN ALIGNMENT

- Reduces bias, toxicity & harm
- Enforces robust safety guardrails
- Refuses harmful or illegal requests

REASONING & LOGIC

- Develops Chain-of-Thought (CoT)
- Solves complex math & coding problems
- Self-corrects and verifies logic steps

TOOL USE & AGENTS

- Executes function & API calls
- Performs real-time web searches
- Runs autonomous agentic workflows

Post-Training Recipes of open LLMs

Model	Openness	Pipeline	Contributions
DeepSeek-R1	weights	Cold-start SFT → RL (GRPO) → SFT → RL	Emergent chain-of-thought from pure RL; GRPO algorithm
OLMo 3	fully	SFT → DPO → RLVR (3 variants: Think, Instruct, RL-Zero)	OlmoRL framework; Dolci PT datasets; Delta Learning DPO
Qwen 3	weights	Cold-start SFT → RL (GRPO) → Thinking Mode Fusion (SFT) → RL	dual thinking mode in one model; minimal cold-start SFT + maximal RL
SmolLM 3	fully	SFT → APO → Model merging	APO (anchored DPO); model merging to recover long context
Nemotron 3	fully	SFT → RLVR+RLHF	Two-Stage SFT Loss; Multi-Environment RLVR ; async GRPO

and many more…

SmolLM3 – HuggingFace

Post-training Recipe

The post-training recipe starts of with the checkpoint after pretraining and long context adaptation. The optimizer state is not re-used from earlier training.

During mid-training the model is trained for **5 epochs** on a mix of OpenThoughts and Nemotron post-training data totalling **175B tokens** (35B unique).

The model is further trained on a mix of **25 high quality datasets** mixing reasoning/non-reasoning data for **4 epochs** and **10B tokens** (2.5B unique).

Finally, we linearly merge the model soup checkpoint with long context checkpoint using a 90/10 ratio to recover the long context capabilities.

We found that model souping, where intermediate checkpoints from the APO stage are merged together further improves the downstream performance.

We apply Anchored Preference Optimization (APO), a variant of DPO, for 1 epoch with a thinking preference pair for every non-thinking preference pair.

Features:

- Dual reasoning mode
- RL with APO (anchored DPO)
- model merging as a practical fix for capability regression

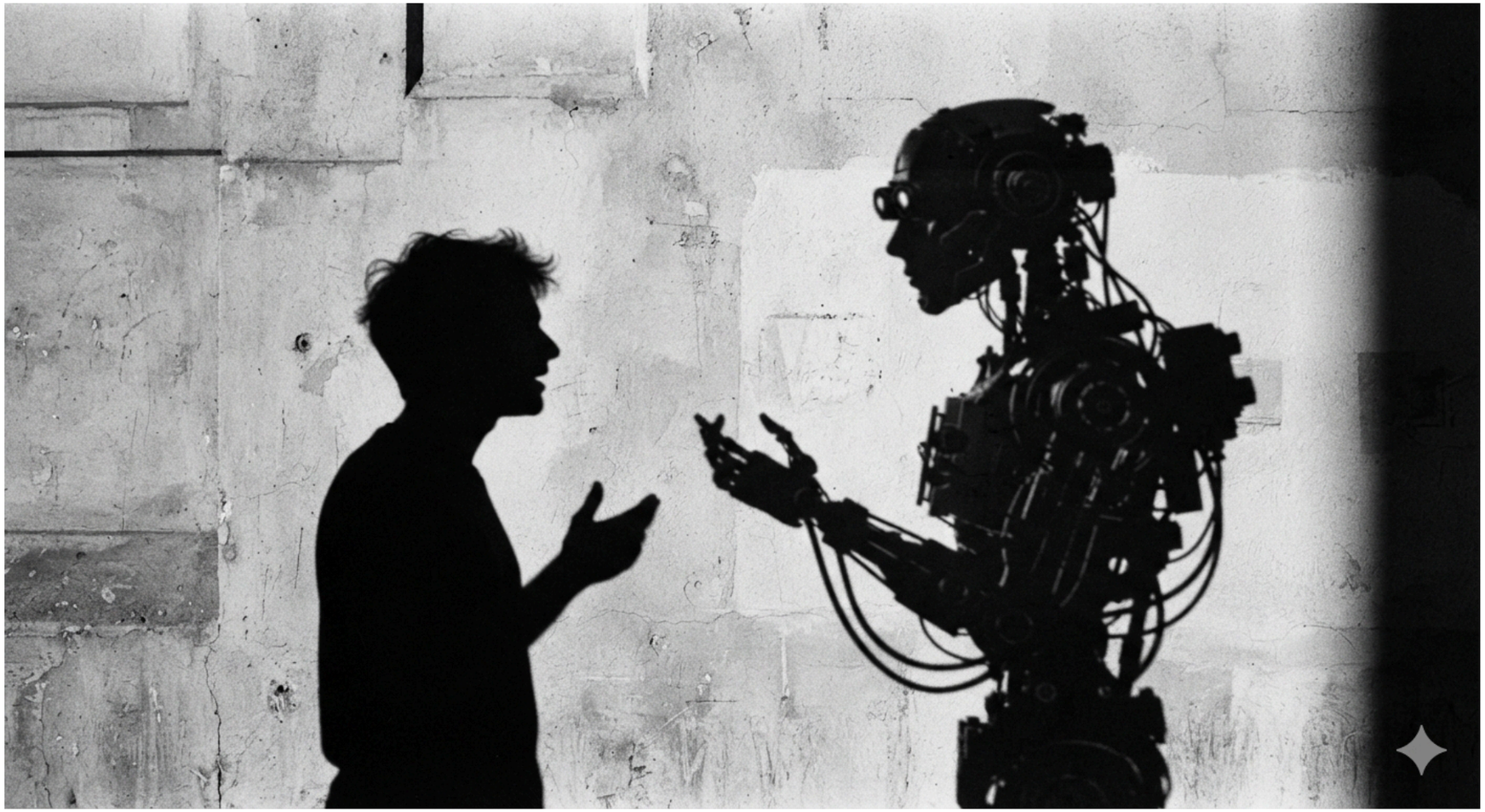
Post-Training Frameworks

- RL library ecosystem is evolving at an *exceptionally high* velocity!
- This July 2025 survey on "Open Source RL Libraries for LLMs" is a good starting point, but things have improved.

RL libraries on Aurora

- **TRL** (HuggingFace)
  - Very popular and tightly bound to the HF ecosystem.
  - Single-GPU to multi-node via FSDP (or DeepSpeed), without an external orchestrator for RL.
- **TorchTitan** (Meta)
  - Pytorch library for pre-training, now includes post-training at scale (replacing TorchTune, also functional and performant on XPUs, and superseding TorchForge).
  - In RL, it decouples generation and training via Monarch actors and TorchStore weight sync. Partially ported and tested on Aurora (🛠️ WIP).
- Ready-to-run examples of both TRL and torchtitan are available on 🍋 ezpz.

Part 2: Supervised Fine-Tuning



What SFT Accomplishes

SFT teaches: instruction following, direct answers, turn-taking, style & tone

Prompt:

> *"What is the capital of France? What is the capital of Spain?"*

Base model (raw text completion):

> *What is the capital of Japan? Answers: London, Madrid, Tokyo. Vocabulary: city, capital. Now that we have some vocabulary, let's try another one […]*

After SFT (question answering):

> *The capital of France is Paris, and the capital of Spain is Madrid.*

How SFT Differs from Pre-Training

	Pre-Training	SFT
Data	Raw web text	Curated conversation pairs
Loss	NLL on all tokens	NLL on **assistant tokens only**
Format	Plain text	Structured chat template (ChatML)
Scale	Trillions of tokens	~1M conversations
Duration	Weeks / months	Minutes / hours

SFT Data Format

- A base model has no notion of roles and turns – it just completes text.
- Training models for question answering rather than raw text generation requires structured data formats.
- Example of the conversational format with multi-turn support (used in the hands-on):

```
{"messages": [
  {"role": "system",    "content": "You are a helpful assistant."},
  {"role": "user",      "content": "What is the capital of France?"},
  {"role": "assistant", "content": "The capital of France is Paris."}
]}
```

- Dataset: **SmolTalk** (~1.04M conversations) — a curated instruction-tuning dataset
- In TRL, `SFTTrainer` supports multiple dataset formats: language modeling (`text`), conversational (`messages`), and prompt-completion (`prompt`/`completion`).
- A **chat template** is used to convert structured `jsonl` data into a flat token sequence the model can train on.

From Data to Tokens: the Chat Template

Step 1 — Raw data: JSONL with roles (e.g. *conversational* format)

```
{"messages": [{"role": "user", "content": "What is the capital of France?"},
              {"role": "assistant", "content": "The capital of France is Paris."}]}
```

Step 2 — Chat template: JSONL data → single string with special tokens:

Template script: (e.g. `chatml.jinja`):

```
{%- for m in messages -%}
{{- '<|im_start|>' + m['role'] + '\n' -}}
{{- m['content'] + '<|im_end|>\n' -}}
{%- endfor -%}
{%- if add_generation_prompt -%}
{{- '<|im_start|>assistant\n' -}}
{%- endif -%}
```

Formatted text string (e.g. ChatML template):

```
<|im_start|>user
What is the capital of France?<|im_end|>
<|im_start|>assistant
The capital of France is Paris.<|im_end|>
```

Step 3 — Tokenizer: converts the string → token IDs:

```
[1, 32001, 882, 10, ..., 32002, 32001, 20227, 10, ..., 32002]
  im_start user \n     im_end im_start assistant \n    im_end
```

- Special tokens (`<|im_start|>`, `<|im_end|>`) must be in the tokenizer vocabulary – if missing, add them and resize model's input embeddings and output layer

Assistant-Only Loss Masking

- In SFT we only want to train on the **assistant's responses**, not the user prompts or system message.

```
<|im_start|>user                          ← masked
What is the capital of France?<|im_end|>   ← masked
<|im_start|>assistant                      ← masked
The capital of France is Paris.<|im_end|>   ← LOSS COMPUTED HERE
```

- Add `{% generation %}` … `{% endgeneration %}` markers in the chat template script:

```
 {%- for m in messages -%}
 {{- '<|im_start|>' + m['role'] + '\n' -}}
-{{- m['content'] + '<|im_end|>\n' -}}
+{%- if m['role'] == 'assistant' %}{% generation %}{{ m['content'] + '<|im_end|>\n' }}{% endgeneration -%}
+{%- else %}{{ m['content'] + '<|im_end|>\n' -}}
+{%- endif -%}
 {%- endfor -%}
 {%- if add_generation_prompt -%}
 {{- '<|im_start|>assistant\n' -}}
 {%- endif -%}
```

- In TRL, set `assistant_only_loss=True` in `SFTConfig` : loss is computed only on tokens within `{% generation %}` … `{% endgeneration %}` markers

Inference: Base vs. Fine-Tuned

- At inference time, the prompting format must match how the model was trained
- Use the same chat template used for training

```
{%- for m in messages -%}
{{- '<|im_start|>' + m['role'] + '\n' -}}
{%- if m['role'] == 'assistant' %}{% generation %}{{ m['content'] + '<|im_end|>\n' }}{% endgeneration -%}
{%- else %}{{ m['content'] + '<|im_end|>\n' -}}
{%- endif -%}
{%- endfor -%}
{%- if add_generation_prompt -%}
{{- '<|im_start|>assistant\n' -}}
{%- endif -%}
```

- Append the assistant start token to any prompt with `tokenizer.apply_chat_template(messages, add_generation_prompt=True)`
- Add `<|im_end|>` to the set of stop tokens (EOS)

Example: SFT on Aurora with TRL

Framework, installed in a `venv` with `--no-deps` inheriting from the `frameworks` module:

- TRL with `SFTTrainer`
- other dependencies: peft, accelerate

Distribution strategy:

- Launch pattern: `mpiexec --ppn 12` (1 MPI rank per XPU tile, not `torchrun`) with `CCL_ATL_TRANSPORT=mpi`
- Map PALS env vars → PyTorch distributed vars (`RANK`, `LOCAL_RANK`, `WORLD_SIZE`)
- Set `export CCL_PROCESS_LAUNCHER=none` and `CCL_LOCAL_RANK` / `CCL_LOCAL_SIZE` for oneCCL
- **DDP** for models that fit in a single tile (64 GB HBM), `--no_fsdp`
- **FSDP** for larger models – `full_shard` only (`hybrid_shard` not supported by oneCCL)
- **Low Rank Adaption:** (LoRA) is available with TRL on Aurora

Example: SFT Training Results

Model: SmolLM2-1.7B | Data: SmolTalk (~1M conversations) | System: 4 Aurora nodes (48 tiles)

Step	Loss	Token Accuracy	Tokens (M)
1	3.368	0.416	0.2
100	0.797	0.765	16.3
500	0.736	0.779	81.4
750	0.698	0.789	122.3
1000	0.699	0.788	163.2

- **Throughput**: ~75.6K tokens/s across 48 tiles (~1,575 tokens/s per tile)
- **Step time**: ~2.1 s/step
- **Total runtime**: ~36 min for 1000 steps (about 18.5% of the dataset)
- **Effective batch size**: 48 tiles × 4 = 192
- The model learns instruction-following early (by step 500)
- *How to improve throughput?*

⌨️ Hands On: SFT

Fine-tune SmolLM2-1.7B on SmolTalk to answer questions.

`Ssh` into Aurora and follow these guidelines: https://github.com/argonne-lcf/ATPESC_MachineLearning/tree/master/06_post_training/sft/README.md

Part 3: Reinforcement Learning for LLMs

What RL accomplishes

RL enhances an LLM ability to

- 🧭 **adhere to target example behaviors** ("do this, don't do that"): align with human preferences, values, goals, and ethical principles
- 🎯 **perform tasks that have a measurable outcome** (success/fail): solve problems that have a verifiable solution (math), use tools

Why is RL needed?

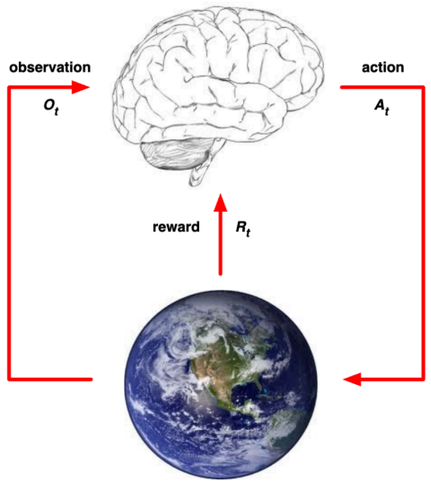
These abilities are hard to learn with pre-training or SFT because

- 🚫 the pre-training cross entropy loss is always positive: it **cannot discourage** or inhibit specific answers/behaviours
- 🔍 the cross entropy loss **cannot distinguish** between correct and incorrect answers/behaviours that sound very similar
- 📏 it is difficult (impossible?) to design a differentiable loss to **fully describe good and bad** answers/behaviours

…so, what *is* RL?

Mapping LLMs to RL Components

Agent and Environment



- At each step t the agent:
  - Executes action A_t
  - Receives observation O_t
  - Receives scalar reward R_t
- The environment:
  - Receives action A_t
  - Emits observation O_{t+1}
  - Emits scalar reward R_{t+1}
- t increments at env. step

RL cycle, from the RL course by David Silver

How to train an LLM within the RL context?

RL concept	Definition	LLM concept
👁️ observation	information received by the environment	context (prompt + generated tokens)
🎬 action	agent's decision to act on the environment	next token (LLM output)
🧠 policy	strategy to decide what action to take	LLM
⭐ reward	feedback by the environment on the agent's action	scalar measuring of quality of LLM response

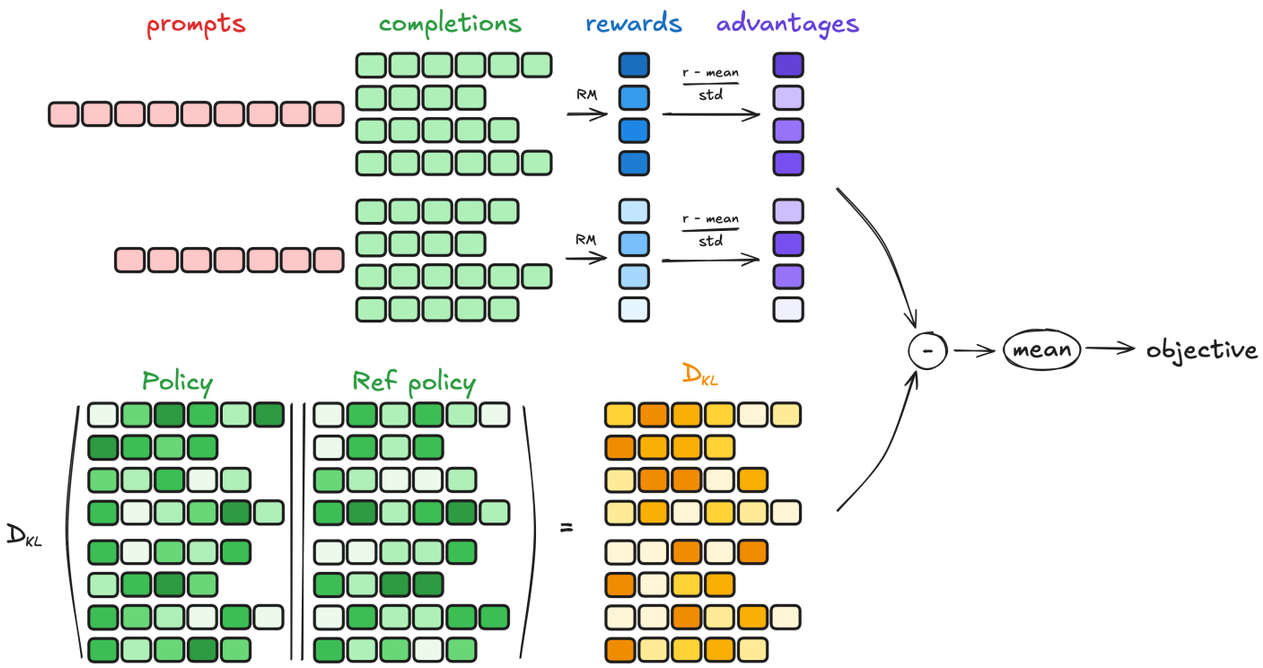
RL Algorithms for LLMs

How RL algorithms update the LLM parameters:

- A **quality "signal"** is needed for every answer: an explicit reward function (+1/-1), or an implicit preference label (good/bad, correct/incorrect).
- The RL loss assigns positive values to correct responses and **negative values to incorrect ones**, generating gradients that increase the likelihood of right answers while suppressing wrong ones.
- RL loss is fundamentally a **contrastive loss**, driving learning primarily through "good" incorrect answers (near misses) rather than correct ones.
  - Correct answers are nothing new – they were already covered in pre-training and SFT – making **high-quality wrong answers the true learning signal in RL**.
- **Where do high-quality wrong answers come from?**
  - Offline RL and RL with human feedback RLHF (like DPO) use curated human-annotated or high-quality synthetic datasets.
  - Online RL with Verifiable Rewards (like GRPO) uses real-time rollouts generated directly by the model being trained.
- Strong **regularizations**, like loss clipping and KL-anchoring to a base reference model, are used to prevent reward hacking and catastrophic forgetting.

GRPO

Group Relative Policy Optimization (DeepSeek, 2025) — RL without a critic model



GRPO traininer – TRL

$$\mathcal{L} = -\frac{1}{G}\sum_{i=1}^{G} \min\left(\frac{\pi_{\theta,i}}{\pi_{\text{old},i}}\hat{A}_i,\ \text{clip}\left(\frac{\pi_{\theta,i}}{\pi_{\text{old},i}}, 1\pm\epsilon\right)\hat{A}_i\right) + \beta\, D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}})$$

For each prompt, sample a **group** of G completions from the current policy π_θ. Score each with a reward function r_i. Compute **group-relative advantages**:

$$\hat{A}_i = \frac{r_i - \text{mean}(\{r_1, \ldots, r_G\})}{\text{std}(\{r_1, \ldots, r_G\})}$$

Update the policy with a clipped surrogate objective (PPO-style) + KL penalty:

- 🚫 **No critic model** — advantages come from within-group comparison, not a learned value function (unlike PPO)
- 🔄 **Self-contrastive** — the model's own completions provide the positive/negative signal
- ✅ **Works with verifiable rewards** — binary (0/1) or rule-based scoring (no reward model needed)

Building a GRPO Pipeline with TRL

1. Dataset — `prompt` column with chat-format messages; extra columns forwarded to reward functions. The prompt must instruct the model on the expected **output format** so the reward function can parse the response:

```
{"prompt": [{"role": "user", "content": "Use 2, 3, 5 to make 11. "
    "Show your reasoning, then give the answer in <answer></answer> tags."}],
 "target": 11, "nums": [2, 3, 5]}
```

2. Reward function — receives decoded completions + extra columns, returns list of floats

```
def countdown_reward(completions, target, nums, **kwargs):
    # parse <answer> tags, check arithmetic, return 0.0 / 0.1 / 1.0 per completion
```

3. Trainer — `GRPOTrainer` handles rollout generation, group normalization, and PPO updates

```
trainer = GRPOTrainer(model=model, reward_funcs=[countdown_reward],
    args=GRPOConfig(num_generations=4, num_iterations=2, beta=0.0, ...),
    train_dataset=dataset, processing_class=tokenizer)
trainer.train()
```

Practical notes: `beta=0.0` skips reference model (memory saved, no KL objective); `num_iterations` >1 needed to activate the clipping objective; LR much lower than SFT (1e-6 to 5e-6)

Example: RL on Aurora with TRL

Practical considerations: frameworks (TRL, torchtitan), distributed RL on Aurora.

Framework, same as SFT: installed in a `venv` with `--no-deps` inheriting from the `frameworks` module:

- TRL with `GRPOTrainer`
- dependencies: peft, accelerate

Distribution strategy:

- Launch pattern, same as SFT: `mpiexec --ppn 12` (1 MPI rank per XPU tile, not `torchrun`)
- Rollouts generation options: colocated (HF, vLLM) or server mode (vLLM)
- **DDP** for models that fit in a single tile (64 GB HBM), `--no_fsdp`
- **FSDP** for larger (3B+ parameters) models – `full_shard`
- **Low Rank Adaption (LoRA)**: available with TRL on Aurora

Example: RL training results

GRPO on Qwen2.5-3B-Instruct, 1 Aurora node (12 XPU tiles), ~26 min

- **Task:** Countdown – given a target integer and 3 numbers, produce an arithmetic expression (+, -, *) that equals the target.
- **Reward**: 0.0 (no answer tags) / 0.1 (valid format, wrong answer) / 1.0 (correct)

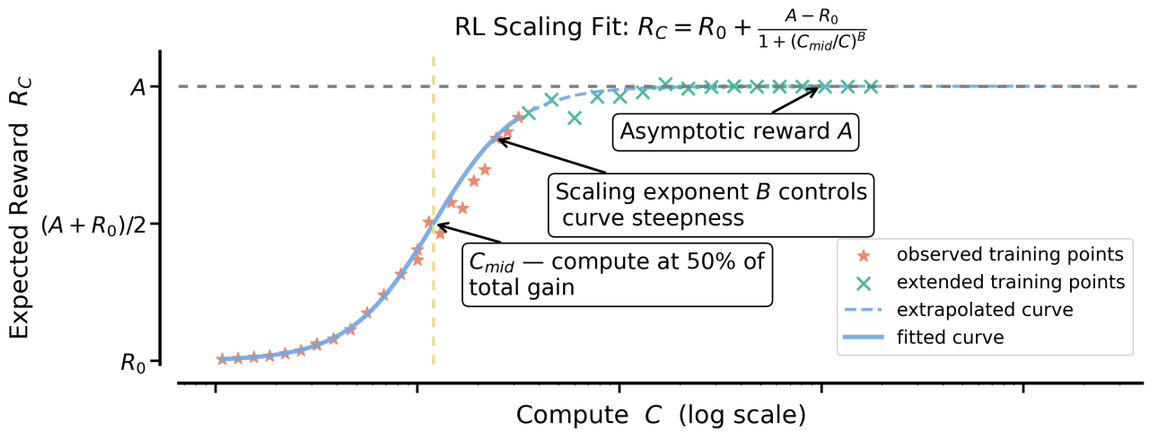
Distribution	FSDP (12 tiles)
Batch size	4·tile (48 effective)
Generations/prompt	4
Updates/batch	2
Max completion	512 tokens
Steps	40
Step time	~37 s
Throughput	~254 tok/s

Key observations:

- 📈 Reward: 0.23 → 0.71 (peak 0.77 at step 32)
- 📉 Completion length: 407 → 214 tokens (model learns conciseness)
- 💉 Entropy: 0.93 → 0.36 (output distribution sharpens)
- ✂️ Clip ratio: 0.3–1.6% of tokens (stable policy updates)

RL Scaling Laws

Systematic study of how RL performance scales with compute budget.



The Art of Scaling RL Compute for LLMs (Meta, 2025)

Key findings (>400K GPU-hours of experiments):

- **The scaling law lets you fit A, B from early runs** and predict large-scale performance (validated: fit on 50K GPU-hrs predicts 100K accurately)
- **Bigger models, larger batches, longer contexts, FP32 logits** increase the asymptotic performance (A).
- **Most design choices affect efficiency (B), not ceiling (A)** – loss aggregation, advantage normalization, off-policy max data reuse primarily modulate how fast you get there
- **Generations per prompt is second-order** – at fixed total batch size, sweeping group size (8→32) leaves scaling curves essentially unchanged

⌨️ Hands On: GRPO

Train SmolLM2-1.7B with GRPO on the Counting Problem, an arithmetic task with verifiable rewards

`Ssh` into Aurora and follow these guidelines: https://github.com/argonne-lcf/ATPESC_MachineLearning/tree/master/06_post_training/grpo/README.md

Part 4: Open Questions

- Catastrophic forgetting
- RLVR with long-running tasks (simulations, experiments)
  - Async RL
- How to get to superintelligence without verifiable rewards?
  - Difference between “current RL for LLMs” vs “RL in games (chess, go, …)”
  - R-Zero: Self-Evolving Reasoning LLM from Zero Data

ARGONNE TRAINING PROGRAM ON EXTREME-SCALE COMPUTING

Produced by Argonne National Laboratory, a U.S. Department of Energy Laboratory managed by UChicagoArgonne, LLC under contract DE-AC02-06CH11357.

Special thanks to the National Energy Research Scientific Computing Center (NERSC) and Oak Ridge Leadership Computing Facility (OLCF) for the use of their resources during the training event.