

Lecture 16

REINFORCEMENT LEARNING FROM VERIFIABLE REWARDS

CS336

(Remaining content from last lecture..)

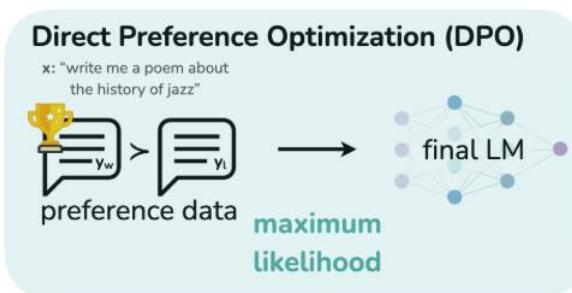
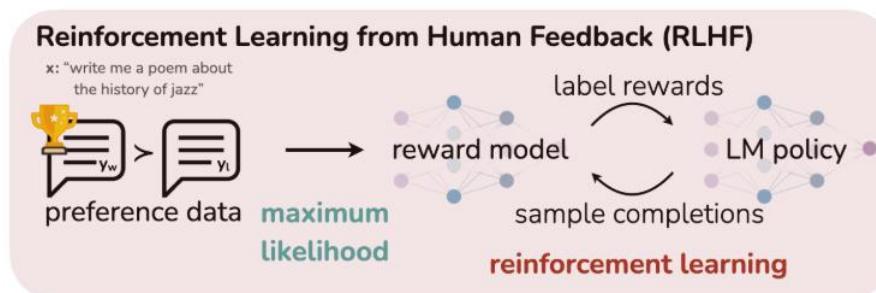
DPO – RLHF without tears?

Try to simplify PPO by..

- Getting rid of the reward model
- Getting rid of any on-policy stuff (rollouts, outer loops etc)

Instead

- Take gradient steps on log-loss of good stuff
- Take negative gradient steps on bad stuff (appropriately weighted).



DPO – derivation from the RLHF formula

Our goal is to optimize

$$\max_{\pi_\theta} \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_\theta(y|x)} [r_\phi(x, y)] - \beta \mathbb{D}_{\text{KL}} [\pi_\theta(y | x) || \pi_{\text{ref}}(y | x)]$$

Assume that the policy π is the set of all policies (nonparametric assumption).

The maximizer is then,

$$\pi_r(y | x) = \frac{1}{Z(x)} \pi_{\text{ref}}(y | x) \exp \left(\frac{1}{\beta} r(x, y) \right)$$

Solve for the ‘implied reward’

$$r(x, y) = \beta \log \frac{\pi_r(y | x)}{\pi_{\text{ref}}(y | x)} + \beta \log Z(x).$$

(This is the equivalence also used in the kimi-think paper)

DPO derivation 2

We can now optimize the implied reward as a reward model via the Stiennon objective

$$\text{loss}(r_\theta) = -E_{(x, y_0, y_1, i) \sim D} [\log(\sigma(r_\theta(x, y_i) - r_\theta(x, y_{1-i})))] \quad \text{and} \quad r(x, y) = \beta \log \frac{\pi_r(y | x)}{\pi_{\text{ref}}(y | x)} + \beta \log Z(x).$$

This gives the DPO objective,

$$\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}} \left[\log \sigma \left(\beta \log \frac{\pi_\theta(y_w | x)}{\pi_{\text{ref}}(y_w | x)} - \beta \log \frac{\pi_\theta(y_l | x)}{\pi_{\text{ref}}(y_l | x)} \right) \right].$$

The key steps?

1. Make a nonparametric assumption (links π_θ and r in closed form)
2. Parametrize reward r via the policy
3. Optimize the reward using supervised losses (which in turn, optimizes the policy)

Conceptually.. This is MLE on the pairwise rewards, under nonparametric assumption + alternative parametrization

DPO updates and components

In some sense, reduces to “pos gradient on good, neg gradient on bad”

What does the DPO update do? For a mechanistic understanding of DPO, it is useful to analyze the gradient of the loss function \mathcal{L}_{DPO} . The gradient with respect to the parameters θ can be written as:

$$\begin{aligned} \nabla_{\theta} \mathcal{L}_{\text{DPO}}(\pi_{\theta}; \pi_{\text{ref}}) = \\ -\beta \mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}} \left[\underbrace{\sigma(\hat{r}_{\theta}(x, y_l) - \hat{r}_{\theta}(x, y_w))}_{\text{higher weight when reward estimate is wrong}} \left[\underbrace{\nabla_{\theta} \log \pi(y_w | x)}_{\text{increase likelihood of } y_w} - \underbrace{\nabla_{\theta} \log \pi(y_l | x)}_{\text{decrease likelihood of } y_l} \right] \right], \end{aligned}$$

(Scaled by ‘prediction error’ of the implied reward model)

Results – controlled comparison in AF

Compared to our previous PPO implementation? Same perf (on sim) with no pain!

Method	Simulated Win-rate (%)	Human Win-rate (%)
GPT-4*†	79.0 ± 1.4	69.8 ± 1.6
ChatGPT*†	61.4 ± 1.7	52.9 ± 1.7
PPO	46.8 ± 1.8	55.1 ± 1.7
DPO	46.8 ± 1.7	-
Best-of-1024	45.0 ± 1.7	50.7 ± 1.8
Expert Iteration	41.9 ± 1.7	45.7 ± 1.7
SFT 52k	39.2 ± 1.7	40.7 ± 1.7
SFT 10k	36.7 ± 1.7	44.3 ± 1.7
Binary FeedME	36.6 ± 1.7	37.9 ± 1.7
Quark	35.6 ± 1.7	-
Binary Reward Conditioning	32.4 ± 1.6	-
Davinci001*	24.4 ± 1.5	32.5 ± 1.6
LLaMA 7B*	11.3 ± 1.1	6.5 ± 0.9

DPO works

T	Model	Average	ARC	HellaSwag	MMLU	TruthfulQA	Winogrande	GSM8K
■	udkai/Turdus	74.66	73.38	88.56	64.52	67.11	86.66	67.7
■	fblgit/UNA-TheBeagle-7B-v1	73.87	73.04	88	63.48	69.85	82.16	66.72
■	argilla/distilabeled-Marcoro14-7B-slelp	73.63	70.73	87.47	65.22	65.1	82.08	71.19
■	mlabonne/NeuralMarcoro14-7B	73.57	71.42	87.59	64.84	65.64	81.22	70.74
◆	abideen/NexoNimbus-7B	73.5	70.82	87.86	64.69	62.43	84.85	70.36
■	Neuronovo/neuronovo-7B-v0.2	73.44	73.04	88.32	65.15	71.02	80.66	62.47
■	argilla/distilabeled-Marcoro14-7B-slelp-full	73.4	70.65	87.55	65.33	64.21	82	70.66
■	Cultrix/MistralTrix-v1	73.39	72.27	88.33	65.24	70.73	80.98	62.77
■	ryandt/MusingCaterpillar	73.33	72.53	88.34	65.26	70.93	80.66	62.24
■	Neuronovo/neuronovo-7B-v0.3	73.29	72.7	88.26	65.1	71.35	80.9	61.41
■	Cultrix/MistralTrixTest	73.17	72.53	88.4	65.22	70.77	81.37	60.73
◆	samir-fama/SamirGPT-v1	73.11	69.54	87.04	65.3	63.37	81.69	71.72
◆	SanjiWatsuki/Lelantos-DPO-7B	73.09	71.08	87.22	64	67.77	80.03	68.46

From Chris Manning – Most ‘top open-source’ RLHF models are DPO’d

Variants

Lots of variants (too many variants), but maybe two of note from the Tulu 3 paper..

SimPO (no ref)

$$\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E} \left[\log \sigma \left(\beta \log \frac{\pi_\theta(y_w | x)}{\pi_{\text{ref}}(y_w | x)} - \beta \log \frac{\pi_\theta(y_l | x)}{\pi_{\text{ref}}(y_l | x)} \right) \right]$$

$$\mathcal{L}_{\text{SimPO}}(\pi_\theta) = -\mathbb{E} \left[\log \sigma \left(\frac{\beta}{|y_w|} \log \pi_\theta(y_w | x) - \frac{\beta}{|y_l|} \log \pi_\theta(y_l | x) - \gamma \right) \right]$$

Length normalized DPO

$$\max_{\pi_\theta} \mathbb{E}_{y_c, y_r \sim \mathcal{D}} \left[\log \sigma \left(\frac{\beta}{|y_c|} \log \frac{\pi_\theta(y_c | x)}{\pi_{\text{ref}}(y_c | x)} - \frac{\beta}{|y_r|} \log \frac{\pi_\theta(y_r | x)}{\pi_{\text{ref}}(y_r | x)} \right) \right].$$

But PPO does too (and sometimes better?)

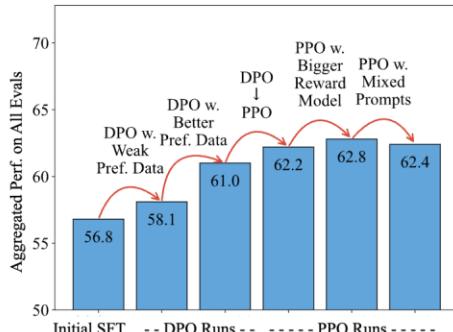
The trickiness of RL-related empirical work..

Lots of results are highly contingent (on the specifics of the experiment setup)

Unpacking DPO and PPO: Disentangling Best Practices for Learning from Preference Feedback

Hamish Ivison^{***} Yizhong Wang^{***} Jiacheng Liu^{***}
Zequi Wu^{*} Valentina Pyatkin^{***} Nathan Lambert^{*}
Noah A. Smith^{***} Yejin Choi^{***} Hannaneh Hajishirzi^{***}

^{*}Allen Institute for AI ^{**}University of Washington
hamishiv@cs.washington.edu



Tülu 3: Pushing Frontiers in Open Language Model Post-Training

Nathan Lambert^{▼1,*} Jacob Morrison^{▼1} Valentina Pyatkin^{▼1,2} Shengyi Huang^{▼1} Hamish Ivison^{▼1,2}
Faeze Brahman^{▼1} Lester James V. Miranda^{▼1}

Alisa Liu² Nouha Dziri¹ Xinxin Lyu¹ Yuling Gu¹ Saumya Malik¹ Victoria Graf² Jena D. Hwang¹
Jiangjiang Yang¹ Ronan Le Bras¹ Oyvind Tafjord¹ Chris Wilhelmi¹

Luca Soldaini¹ Noah A. Smith^{1,2} Yizhong Wang^{1,2} Pradeep Dasigi¹ Hannaneh Hajishirzi^{1,2}

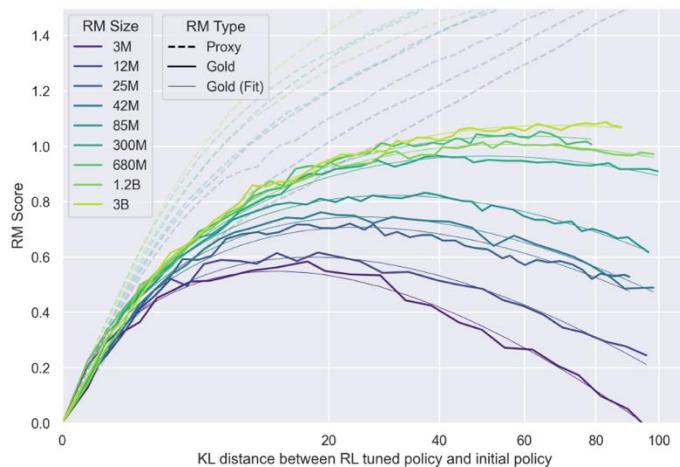
¹Allen Institute for AI, ²University of Washington

*Tülu 3 was a team effort. ▼ marks core contributors. See full author contributions here.
Contact tülu@allenai.org.

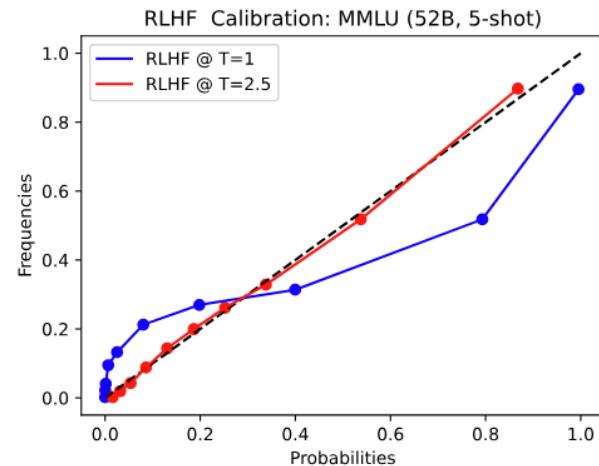
Algorithm	LR	$\gamma - \beta$ ratio	β	Epochs	Batch Size	Average Score
SFT Base	-	-	-	-	-	55.7
SimPO	5.00E-07	0.5	2	1	128	51.8
SimPO	5.00E-07	0.3	10	1	128	52.9
DPO	5.00E-07	-	0.1	3	32	55.2
PPO	1.00E-06	-	0.0325	1	64	54.5
PPO	1.00E-06	-	0.05	1	64	55.5
DPO-norm	1.00E-07	-	5	3	32	56.1
DPO-norm	5.00E-07	-	10	3	32	55.2
DPO-norm	5.00E-07	-	15	3	32	55.7
DPO-norm	5.00E-07	-	2	3	32	46.8
DPO-norm	5.00E-07	-	5	3	32	53.4
DPO-norm	5.00E-07	-	5	1	32	57.3

Things to watch out for in RLHF

Overoptimization / overfitting on the reward

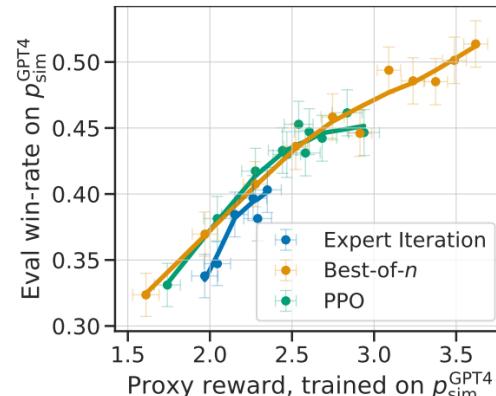
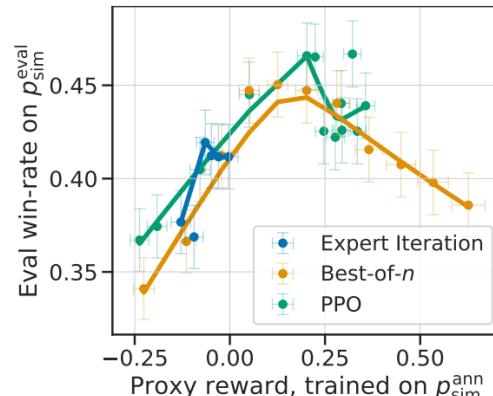
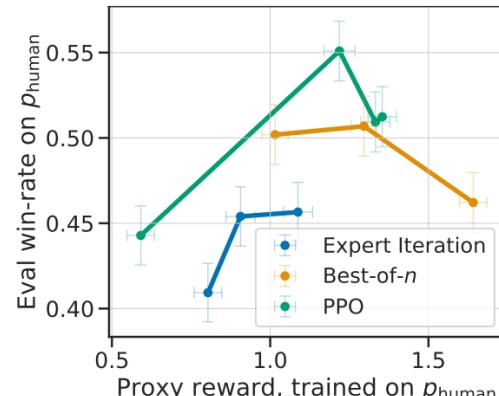


Mode collapse / entropy



Things to watch out for - Overoptimization

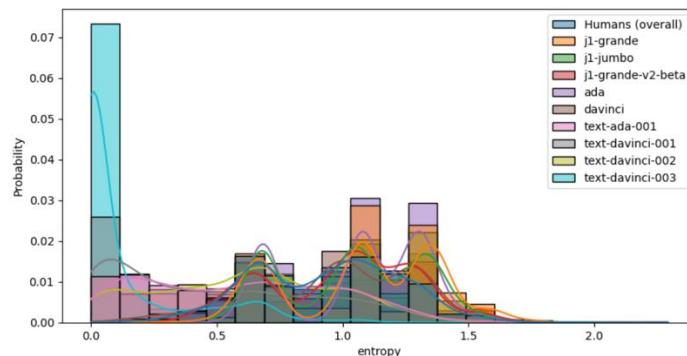
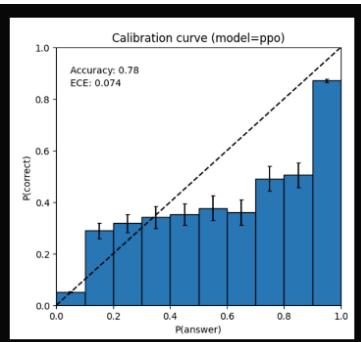
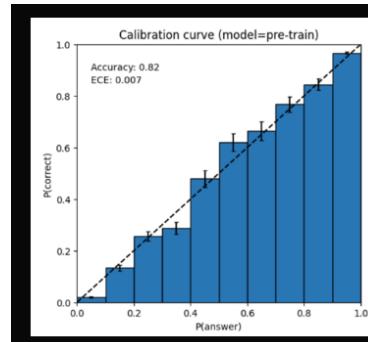
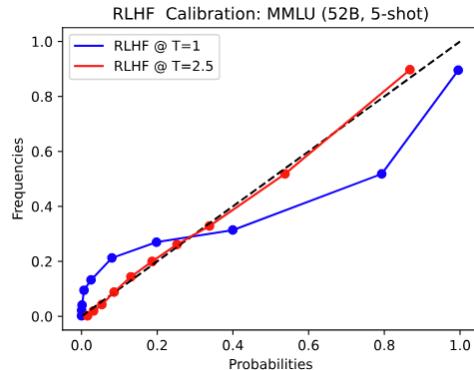
Across many different RLHF-style optimizers. Optimizing for reward overfits past a point



Holds true for human pref (left), noisy LM pref (mid) but not noiseless LM pref (right)

Things to watch out for - mode collapse

RLHF makes models no longer ‘probabilistic models’ – no calibration by default



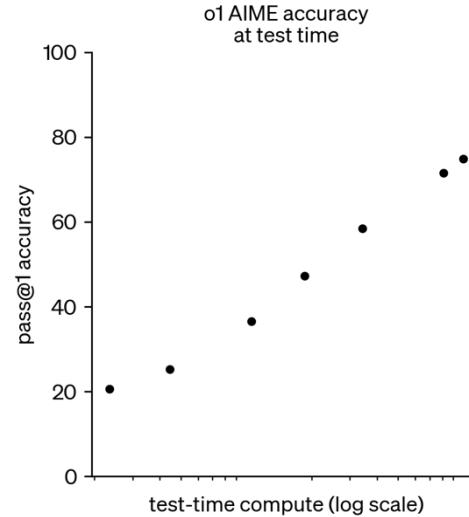
The class thus far

Pre-training + RLHF gets you to ~ GPT3.5... **Today we'll get all the way to o1/r1**

OMG! What Will Happen
When A.I. Makes BuzzFeed
Quizzes?

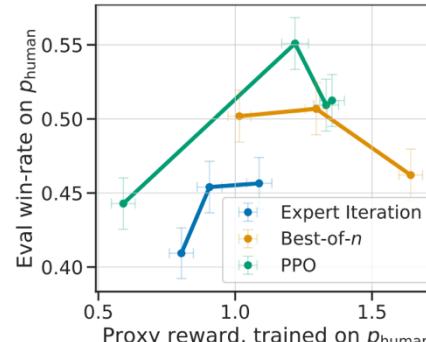
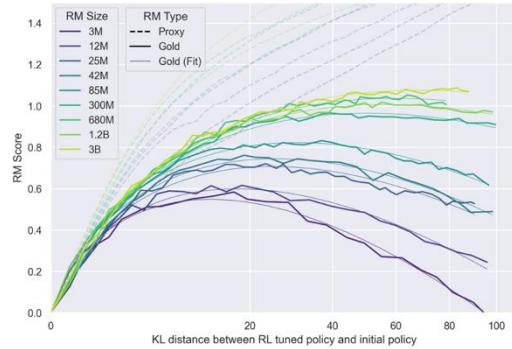
The site announced plans to use tools from the creator of ChatGPT to create content. Former quiz writers aren't surprised.

ChatGPT: Optimizing
Language Models
for Dialogue



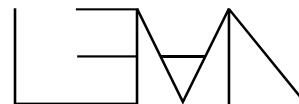
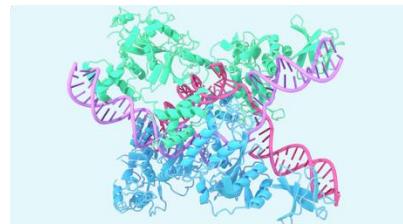
The goal – expand the scope and power of RL

With RLHF: we can't cleanly scale out due to overoptimization



(a) Human preferences ■

Can we work in domains where RL excels? Where we optimize *exactly* what we want



The lecture today

1. Core algorithms..

PPO → GRPO



DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models

Zhibong Shao^{1,2*}, Peiyi Wang^{1,3†}, Qiaohu Zhu^{1,3†}, Runxin Xu¹, Junxiao Song¹,
Xiao Bi¹, Haowei Zhang¹, Mingchuan Zhang¹, Y.K. Li¹, Y. Wu¹, Daya Guo^{1,2}

¹DeepSeek-AI, ²Tsinghua University, ³Peking University

(zhibongshao, wangpeiyi, zhuhq, guoday)@deepseek.com
<https://github.com/DeepSeek-AI/DeepSeek-Math>

GRPO variants

Understanding R1-Zero-Like Training: A Critical Perspective

Zichen Liu^{*1,2} Changyu Chen^{*1,3} Wenjun Li^{*3} Penghui Qi^{*1,2}

Tianyu Pang¹ Chao Du¹ Wee Sun Lee² Min Lin¹

¹Sea AI Lab ²National University of Singapore ³Singapore Management University

2. Case studies



DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning

DeepSeek-AI

research@deepseek.com

KIMI K1.5: SCALING REINFORCEMENT LEARNING WITH LLMs

TECHNICAL REPORT OF KIMI K1.5

Kimi Team



2025-05-14

Qwen3 Technical Report

Qwen Team

- <https://duggingface.co/Qwen>
- <https://modelscope.cn/organization/qwen>
- <https://github.com/QwenLM/Qwen3>

Recap: PPO in theory

A brief (and high level) intro to the various ideas in PPO..

Attempt 1: Policy gradients (variances are too high)

$$\nabla_{\theta} E_{p_{\theta}}[R(z)] = E_{p_{\theta}}[R(z) \nabla_{\theta} \log p_{\theta}(z)]$$

Attempt 2: TRPO (Linearize the problem around the current policy)

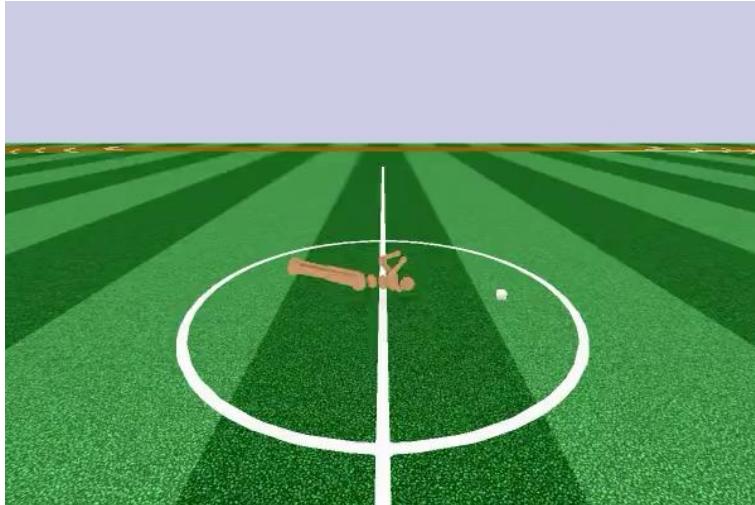
$$\begin{aligned} & \underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)} \hat{A}_t \right] \\ & \text{subject to} \quad \hat{\mathbb{E}}_t [\text{KL}[\pi_{\theta_{\text{old}}}(\cdot | s_t), \pi_{\theta}(\cdot | s_t)]] \leq \delta. \end{aligned}$$

Attempt 3: PPO (Clip the ratios at some eps)

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_{\theta}(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \text{clip} \left(\frac{\pi_{\theta}(a|s)}{\pi_{\theta_k}(a|s)}, 1 - \epsilon, 1 + \epsilon \right) A^{\pi_{\theta_k}}(s, a) \right)$$

PPO

Policy gradient method for optimizing rewards in actual RL tasks..



From the PPO announcement blog (2017)



OpenAI Five (2019)

PPO – at a conceptual level

The objective is basically all there is to it at a conceptual level

Algorithm 1 PPO-Clip

- 1: Input: initial policy parameters θ_0 , initial value function parameters ϕ_0
- 2: **for** $k = 0, 1, 2, \dots$ **do**
- 3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
- 4: Compute rewards-to-go \hat{R}_t .
- 5: Compute advantage estimates, \hat{A}_t (using any method of advantage estimation) based on the current value function V_{ϕ_k} .
- 6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg \max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \min \left(\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_k}(a_t | s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.

- 7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg \min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \left(V_{\phi}(s_t) - \hat{R}_t \right)^2,$$

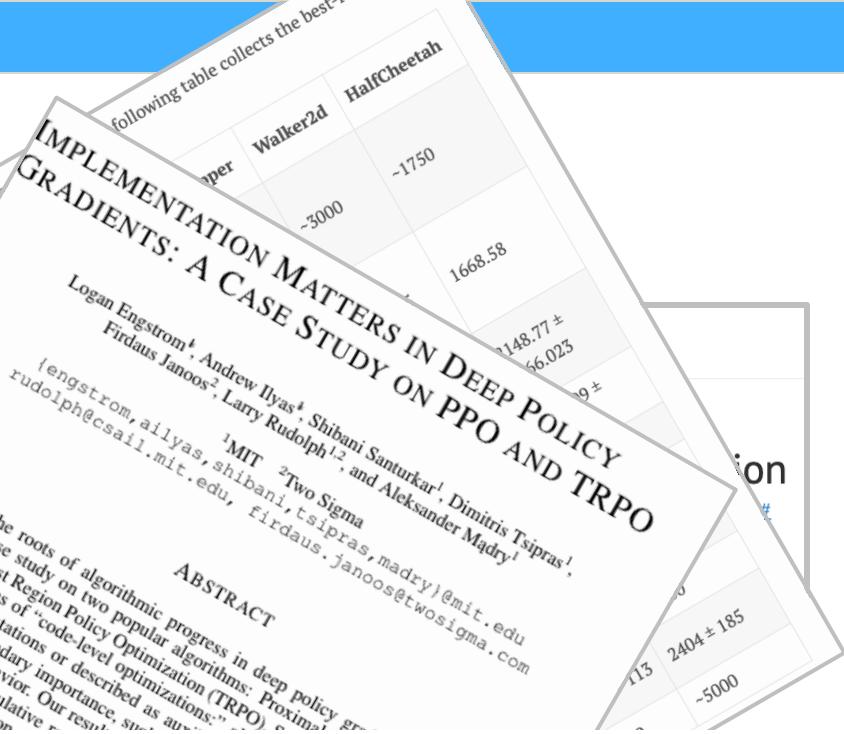
typically via some gradient descent algorithm.

- 8: **end for**
-

PPO in practice

RL Library	GitHub Stars	Benchmark Source
Baselines ppo1 (da99706)	1.1k	GitHub
Baselines ppo2 (7bfbcf1 ea68f3b)	1.1k	GitHub

In recent years, reproducing PPO's results has become a challenge. We have collected the performance of PPO in popular RL libraries in Atari and MuJoCo environments.

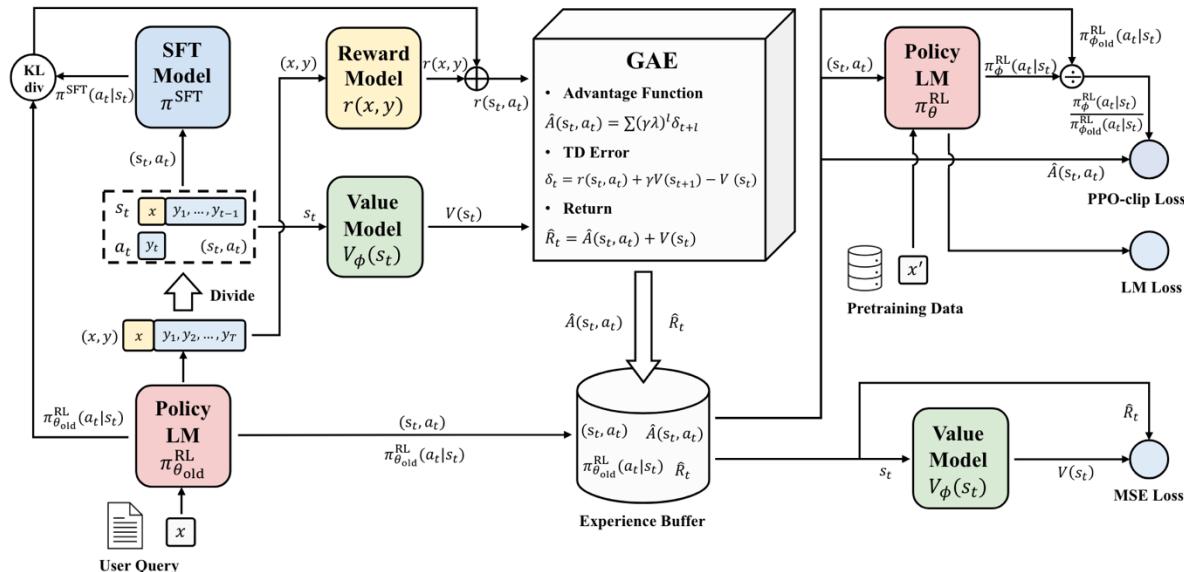


We need to look at a live implementation when talking about PPO

Stable-Baselines3	GitHub Stars (5)	repo	docs (3)	paper (4) (*)	paper (6) (*)
CleanRL	7.5k				
Tianshou	51k				
Ray/RLlib	6.7k				
SpinningUp	1.2k				
ChainerRL	393				

PPO – idealization (?) for language models

In the language model setting..



Pretty similar to the RL formulation. Actions operate over tokens, big dense reward at the very end operating on full sequence

PPO – let's look at an implementation

Implementation – AlpacaFarm's PPO

Method	Simulated Win-rate (%)	Human Win-rate (%)
GPT-4* [†]	79.0 ± 1.4	69.8 ± 1.6
ChatGPT* [†]	61.4 ± 1.7	52.9 ± 1.7
PPO	46.8 ± 1.8	55.1 ± 1.7
DPO	46.8 ± 1.7	-
Best-of-1024	45.0 ± 1.7	50.7 ± 1.8
Expert Iteration	41.9 ± 1.7	45.7 ± 1.7
SFT 52k	39.2 ± 1.7	40.7 ± 1.7

Used by several others to reasonable success

SALMON: SELF-ALIGNMENT WITH INSTRUCTABLE REWARD MODELS

Zhiqing Sun^{1,2*} Yikang Shen¹ Hongxin Zhang³ Qinhong Zhou³ Zhenfang Chen¹

David Cox¹ Yiming Yang² Chuang Gan^{1,3}

¹MIT-IBM Watson AI Lab, IBM Research

²Language Technologies Institute, CMU

³UMass Amherst

ALIGNING LARGE MULTIMODAL MODELS WITH FACTUALLY AUGMENTED RLHF

Zhiqing Sun*[♦], Sheng Shen*[♦], Shengcao Cao*[◊]
Haotian Liu[◊], Chunyuan Li[‡], Yikang Shen[△], Chuang Gan^{†▽△}, Liang-Yan Gui^{†◊}
Yu-Xiong Wang^{†◊}, Yiming Yang^{†♦}, Kurt Keutzer^{†♦}, Trevor Darrell^{†♦}

[♦]UC Berkeley, [♣]CMU, [◊]UIUC, [◊]UW-Madison, [▽]UMass Amherst

[‡]Microsoft Research, [△]MIT-IBM Watson AI Lab

PPO in practice

PPO outer loop. Invoke an inner loop to optimize the loss over some rollouts.

```
def step_with_rollouts(self, rollouts):
    """Based on fixed rollouts, run PPO for multiple epochs."""
    assert isinstance(self.optimizer, AcceleratedOptimizer), (
        "'optimizer' must be pushed through `accelerator.prepare`. "
        "Otherwise the `accelerator.accumulate` context manager won't correctly disable `zero_grad` or `step`."
    )
    rollouts_dataloader = self.get_rollouts_dataloader(rollouts=rollouts)
    stats_list = []
    for epoch_idx in range(self.args.noptepochs):
        for batch_idx, rollouts_batch in tqdm.tqdm(
            enumerate(rollouts_dataloader, 1), disable=not self.accelerator.is_main_process, desc="gradstep"
        ):
            with self.accelerator.accumulate(self.policy):
                ppo_loss, stats_for_this_step = self.compute_loss(rollouts_batch)
                self.accelerator.backward(ppo_loss)
                if self.accelerator.sync_gradients:
                    # Gradient norm almost blows up at some point, but stabilizes eventually, even w/o clipping.
                    if self.args.max_grad_norm is not None:
                        self.accelerator.clip_grad_norm_(self.policy.parameters(), self.args.max_grad_norm)
                    stats_for_this_step["loss/grad_norm"] = self._compute_grad_norm()
                stats_list.append(stats_for_this_step)
            self.optimizer.step()
            self.optimizer.zero_grad(set_to_none=True)
    return common.merge_dict(stats_list, torch.stack) # list of dict -> dict: str -> 1-D tensor
```

PPO in practice – loss computation

AlpacaFarm code – loss computation. Pretty standard stuff.

```
class PPOTrainer(rl_trainer.RLTrainer):

    def compute_loss(self, rollouts: Dict[str, Tensor]) -> Tuple[Tensor, Dict]:
        values, old_logprob, returns, advantages, queries, query_attn_masks, responses = common.prepare_inputs(
            common.unpack_dict(
                rollouts,
                keys=("values", "logprobs", "returns", "advantages", "queries", "query_attn_masks", "responses"),
            ),
            device=self.accelerator.device,
        )
        outputs = self.policy(queries, query_attn_masks, responses, temperature=self.args.temperature)

        vpred = outputs["values"]
        vpredclipped = torch.clamp(
            vpred,
            min=values - self.args.cliprange_value,
            max=values + self.args.cliprange_value,
        )
        vf_losses1 = (vpred - returns) ** 2.0
        vf_losses2 = (vpredclipped - returns) ** 2.0
        vf_loss = 0.5 * torch.maximum(vf_losses1, vf_losses2).mean()
        vf_clipfrac = (vf_losses2 > vf_losses1).to(torch.get_default_dtype()).mean()

        logprob = outputs["logprobs"]
        ratio = torch.exp(logprob - old_logprob)
        # When current policy is close to the old policy, the KL component of this advantage is approximately correct.
        pg_losses = -advantages * ratio
        pg_losses2 = -advantages * torch.clamp(ratio, min=1.0 - self.args.cliprange, max=1.0 + self.args.cliprange)
        pg_loss = torch.maximum(pg_losses, pg_losses2).mean()
        pg_clipfrac = (pg_losses2 > pg_losses).to(torch.get_default_dtype()).mean() # noqa

        loss = pg_loss + self.args.vf_coef * vf_loss
```

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \text{clip} \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1 - \epsilon, 1 + \epsilon \right) A^{\pi_{\theta_k}}(s, a) \right)$$

Cliprange=0.2

PPO in practice – rollouts.

```
102     def rollout(self, queries_data) -> Dict[str, Tensor]:
103         """Rollout trajectories with policy.
104
105         Args:
106             queries_data: Sequence of batches or DataLoader.
107                 Each batch is a dict with keys 'queries' and 'query_attn_masks'.
108
109         Returns:
110             Dictionary with keys
111                 'queries', 'query_attn_masks', 'responses',
112                 'logprobs', 'ref_logprobs', 'values',
113                 'rewards', 'non_score_rewards', 'shaped_rewards'.
114
115         """
116         # Give up dropout throughout.
117         self.policy.eval()
118         self._make_fsdp_happy()
119         # `keep_fp32_wrapper` retains the autocast wrapper of model.forward created by accelerate:
120         # recall one sets mixed precision options with accelerator.
121         # The precise value of this arg doesn't matter here, since we use the unwrapped model only for respond.
122         # Generally, try to use the wrapped model as much as you can, since it's got the autocast/cast-back wrappers.
123         unwrapped_policy = self.accelerator.unwrap_model(self.policy, keep_fp32_wrapper=True)
124
125         self.ref_policy.eval()
126         self.reward_model.eval()
127
128         rollouts = []
129         for batch_idx, batch in tqdm.tqdm(
130             enumerate(queries_data),
131             disable=not self.accelerator.is_main_process,
132             desc="rollout",
133         ):
134             # Sample rollouts.
135             queries, query_attn_masks = common.unpack_dict(
136                 common.prepare_inputs(batch, device=self.accelerator.device),
137                 keys=("queries", "query_attn_masks"),
138             )
139             respond_outputs = unwrapped_policy.respond(queries, query_attn_masks, temperature=self.args.temperature)
140             (responses,) = common.unpack_dict(respond_outputs, ("responses",))
141
142             # Evaluate logprobs of the samples.
143             rollouts_batch = {"queries": queries, "query_attn_masks": query_attn_masks, "responses": responses}
144             policy_outputs = self.policy(**rollouts_batch, temperature=self.args.temperature)
145             ref_policy_outputs = self.ref_policy(**rollouts_batch, temperature=self.args.temperature)
146             policy_outputs = common.unpack_dict(
147                 policy_outputs, keys=("logprobs", "values", "entropies"), return_type=dict
148             )
149             ref_policy_outputs = common.unpack_dict(
150                 ref_policy_outputs, keys=("logprobs", "entropies"), return_type=dict
151             )
152             rollouts_batch.update(policy_outputs)
153             rollouts_batch.update({f"ref_{key}": value for key, value in ref_policy_outputs.items()})
```

```
154         # Evaluate reward of the samples.
155         text_queries, text_responses = tuple(
156             self.tokenizer.batch_decode(tensor, skip_special_tokens=True, clean_up_tokenization_spaces=True)
157             for tensor in (queries, responses)
158         )
159         del queries, responses # Prevent mistakes.
160
161         # We retokenizer, since policy and reward model might not have the same tokenizer.
162         # TODO(lxuchen): Avoid retokenization when policy and reward tokenizer are the same.
163         text_sequences = [q + r for q, r in utils.zip_(text_queries, text_responses)]
164         # TODO(lxuchen): This response retokenization has issues with OPT, since the tokenizer always prepend
165         # <bos_token>. But the issue is local to post_reward, which isn't an issue if we don't penalize.
166         sequences, responses = tuple(
167             self.tokenizer(text, return_tensors="pt", padding=True, truncation=True)
168             for text in (text_sequences, text_responses)
169         )
170         sequences, responses = common.prepare_inputs((sequences, responses), device=self.accelerator.device)
171
172         reward_outputs = self.reward_model(**sequences)
173         reward_outputs = self.post_reward(reward_outputs, responses.input_ids)
174         rollouts_batch.update(reward_outputs)
175
176         # Shape reward with KL penalty.
177         shape_reward_outputs = self._shape_reward(
178             rewards=rollouts_batch["rewards"],
179             responses=rollouts_batch["responses"],
180             logprobs=rollouts_batch["logprobs"],
181             ref_logprobs=rollouts_batch["ref_logprobs"],
182         )
183         rollouts_batch.update(shape_reward_outputs)
184
185         rollouts_batch_cpu = {key: value.cpu() for key, value in rollouts_batch.items()}
186         rollouts.append(rollouts_batch_cpu)
187
188         # Items in dict need to be of same shape.
189         rollouts = common.merge_dict(rollouts, merge_fn=torch.cat)
190         # Estimating advantages outside the loop gives more samples for reward normalization.
191         advantages = self._estimate_advantage(
192             rewards=rollouts["shaped_rewards"].to(self.accelerator.device),
193             values=rollouts["values"].to(self.accelerator.device),
194         )
195         advantages = {key: value.cpu() for key, value in advantages.items()}
196         return {**rollouts, **advantages}
```

PPO in practice – reward shaping

High level – add per-token KL penalty, last-token full reward

In practice? Clip KL for sequences where new policy logp < reference logp

```
67     def _shape_reward(
68         self, rewards: Tensor, responses: Tensor, logprobs: Tensor, ref_logprobs: Tensor
69     ) -> Dict[str, Tensor]:
70         # For some reason, line below doesn't work.
71         # kl = (logits.softmax(dim=-1) * (logits.log_softmax(dim=-1) - ref_logits.log_softmax(dim=-1))).sum(dim=-1)
72         kl = torch.clamp(logprobs - ref_logprobs, min=0.0)
73         non_score_rewards = -self.kl_ctl.value * kl
74         shaped_rewards = non_score_rewards.clone()
75         # This introduces a small index off by one bug if pad_token_id == eos_token_id.
76         terminal_positions = (responses != self.tokenizer.pad_token_id).sum(dim=1) - 1
77         shaped_rewards[list(range(rewards.size(0))), terminal_positions] += rewards
78         return dict(shaped_rewards=shaped_rewards, non_score_rewards=non_score_rewards, kl=kl)
```

Helps with stability? If we blow up our model, this prevents kl from diverging

PPO in practice – generalized advantage estimate

Instead of reward, we use advantages

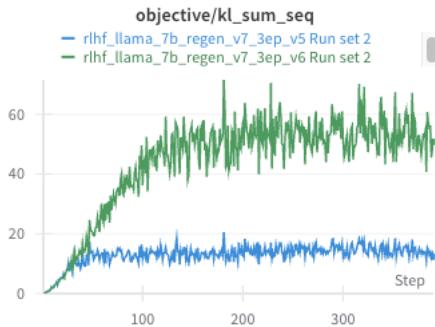
$$\hat{A}_t^{\text{GAE}(\gamma, \lambda)} := \sum_{l=0}^{\infty} (\gamma \lambda)^l \delta_{t+l}^V \quad \text{where} \quad \delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t),$$

```
80     def _estimate_advantage(self, rewards: Tensor, values: Tensor) -> Dict[str, Tensor]:
81         """Generalized advantage estimation.
82
83         Reference:
84             https://arxiv.org/abs/1506.02438
85         """
86
87         if self.args.whiten_rewards:
88             rewards = torch_ops.whiten(rewards, shift_mean=False)
89         lastgaelam = 0
90         advantages_reversed = []
91         gen_length = self.args.response_len
92         for t in reversed(range(gen_length)):
93             nextvalues = values[:, t + 1] if t < gen_length - 1 else 0.0
94             delta = rewards[:, t] + self.args.gamma * nextvalues - values[:, t]
95             lastgaelam = delta + self.args.gamma * self.args.lam * lastgaelam
96             advantages_reversed.append(lastgaelam)
97         advantages = torch.stack(advantages_reversed[::-1], dim=1)
98         returns = advantages + values
99         advantages = torch_ops.whiten(advantages, shift_mean=True)
100        return dict(returns=returns, advantages=advantages)
```

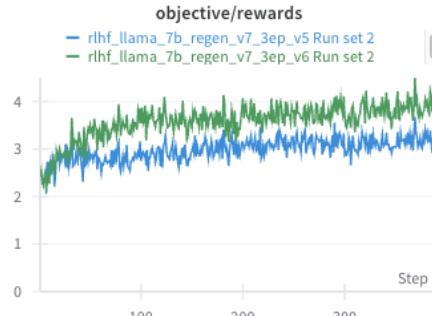
Funny detail – this is a bandit problem and gamma=lambda=1 works – this is the reward-to-go vs the value

What do you expect to see in PPO?

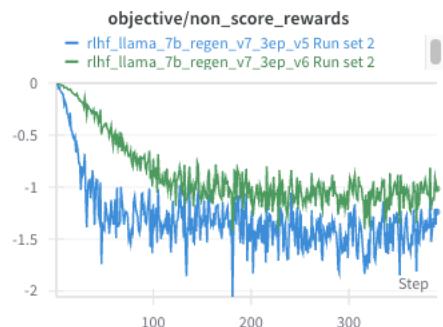
Increasing overall rewards



Incl. reward model



Negative KL rewards



This is a bandit setting, you expect reasonable training curves

Why do we need yet another RL algorithm..?

Why not PPO?

- In practice, complicated implementation
- Value model (memory hungry, involves additional tuning for training)

Why not DPO?

- Data not inherently pairwise (or in the form of Bradley-Terry comparisons)
- Offline (though could be made online by iterating)

New kid on the block: GRPO

What's GRPO?

- Start with PPO (many parts are similar)
- Remove the value function / advantage computation
- Calculate the advantage as “z-score within group”

Group Relative Policy Optimization In order to save the training costs of RL, we adopt Group Relative Policy Optimization (GRPO) ([Shao et al., 2024](#)), which foregoes the critic model that is typically the same size as the policy model, and estimates the baseline from group scores instead. Specifically, for each question q , GRPO samples a group of outputs $\{o_1, o_2, \dots, o_G\}$ from the old policy $\pi_{\theta_{old}}$ and then optimizes the policy model π_θ by maximizing the following objective:

$$\mathcal{J}_{GRPO}(\theta) = \mathbb{E}[q \sim P(Q), \{o_i\}_{i=1}^G \sim \pi_{\theta_{old}}(O|q)] \frac{1}{G} \sum_{i=1}^G \left(\min \left(\frac{\pi_\theta(o_i|q)}{\pi_{\theta_{old}}(o_i|q)} A_i, \text{clip} \left(\frac{\pi_\theta(o_i|q)}{\pi_{\theta_{old}}(o_i|q)}, 1 - \epsilon, 1 + \epsilon \right) A_i \right) - \beta \mathbb{D}_{KL}(\pi_\theta || \pi_{ref}) \right), \quad (1)$$

$$\mathbb{D}_{KL}(\pi_\theta || \pi_{ref}) = \frac{\pi_{ref}(o_i|q)}{\pi_\theta(o_i|q)} - \log \frac{\pi_{ref}(o_i|q)}{\pi_\theta(o_i|q)} - 1, \quad (2)$$

where ϵ and β are hyper-parameters, and A_i is the advantage, computed using a group of rewards $\{r_1, r_2, \dots, r_G\}$ corresponding to the outputs within each group:

$$A_i = \frac{r_i - \text{mean}(\{r_1, r_2, \dots, r_G\})}{\text{std}(\{r_1, r_2, \dots, r_G\})}. \quad (3)$$

(PPO for reference)

$$\min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \text{clip} \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1 - \epsilon, 1 + \epsilon \right) A^{\pi_{\theta_k}}(s, a) \right)$$

In the **online case (rollout+immediate update)**, this is just policy gradient with group normalized rewards

GRPO is very simple (thanks to lack of value function..)

You can (and people do) write tiny GRPO implementations

- Compute reward for each rollout
- Mean/Var normalization per group
- Compute KL term
- Gradient updates on the loss

We can walk through this example from

<https://github.com/McGill-NLP/nano-aha-moment>

```
def compute_pg_loss(
    policy_model: Union[DeepSpeedEngine, PreTrainedModel],
    reference_model: Union[DeepSpeedEngine, PreTrainedModel],
    batch: Dict[str, torch.Tensor],
    total_response_len: int,
) -> Tuple[torch.Tensor, Dict[str, float]]:
    """
    Compute the policy gradient loss with KL penalty between policy and reference models.

    This function:
    1. Computes log probabilities for both policy and reference models
    2. Calculates KL divergence penalty between the models
    3. Computes policy gradient loss using advantages
    4. Combines the losses with KL coefficient

    Args:
        policy_model: The model being trained
        reference_model: The reference model for KL penalty calculation
        batch: Dictionary containing:
            - input_ids: Tensor of shape [batch_size, seq_len]
            - attention_mask: Tensor of shape [batch_size, seq_len]
            - labels: Tensor of shape [batch_size, seq_len] with -100 for ignored positions
            - advantages: Tensor of shape [batch_size, seq_len]

    Returns:
        Tuple containing:
            - loss: Combined policy gradient and KL penalty loss (scalar tensor)
            - metrics: Dictionary with detailed loss components:
                - policy_loss: Pure policy gradient loss
                - kl_penalty: KL divergence penalty
                - entropy: Policy entropy
    .....
    input_ids = batch["input_ids"] # [batch_size, seq_len]
    attention_mask = batch["attention_mask"] # [batch_size, seq_len]
    labels = batch["labels"] # [batch_size, seq_len]
    advantages = batch["advantages"] # [batch_size, seq_len]

    model_inputs = {
        "input_ids": input_ids,
        "attention_mask": attention_mask,
        "labels": labels,
    }

    labels_mask = (labels..., 1:] != -100).float() # [batch_size, seq_len-1]

    with torch.no_grad():
        ref_logs = compute_token_log_probs(
            reference_model, model_inputs, TEMPERATURE
        ) # [batch_size, seq_len-1]

        logs = compute_token_log_probs(policy_model, model_inputs, TEMPERATURE) # [batch_size, seq_len-1]

        kl_penalty = torch.exp(ref_logs - logs) - (ref_logs - logs) - 1 # [batch_size, seq_len-1]
        kl_penalty = kl_penalty * labels_mask # [batch_size, seq_len-1]

        entropy = -logs.sum() / labels_mask.sum() # scalar

        policy_loss = -logs * advantages..., 1:] # [batch_size, seq_len-1]
        policy_loss = policy_loss * labels_mask # [batch_size, seq_len-1]

        loss = (policy_loss + KL_COEFFICIENT * kl_penalty).sum() / total_response_len # scalar
```

Advantage computation is also very simple..

Basically just the ‘vanilla’ GRPO setup.

Main difference here is just the 1e-4 stability factor in the std calculation

```
assert len(all_generations) == len(all_finish_reasons)
assert len(all_generations) == len(samples) * GENERATIONS_PER_SAMPLE

# Process responses and calculate rewards
groups = [
    list(range(i, i + GENERATIONS_PER_SAMPLE))
    for i in range(0, len(all_generations), GENERATIONS_PER_SAMPLE)
] # example: [[0, 1, 2], [3, 4, 5], [6, 7, 8]]

all_query_token_ids, all_responses_token_ids, all_advantages = [], [], []

stats = {
    "response_lengths": [],
    "rewards": [],
    "non_stop_rate": []
}

for sample, group_indices in zip(samples, groups):
    finish_reasons = [all_finish_reasons[i] for i in group_indices]
    response_token_ids = [all_generations[i] for i in group_indices]
    responses = tokenizer.batch_decode(response_token_ids, skip_special_tokens=False)

    rewards_and_metrics = [compute_reward(resp, sample) for resp in responses]
    rewards, reward_metrics = zip(*rewards_and_metrics)

    rewards = np.array(rewards) # [group_size]
    response_advantages = (rewards - rewards.mean()) / (rewards.std() + 1e-4)

    advantages = [
        [resp_adv] * len(resp)
        for resp_adv, resp in zip(response_advantages, response_token_ids)
    ]

    all_query_token_ids.extend([sample["input_ids"]] * GENERATIONS_PER_SAMPLE)
    all_responses_token_ids.extend(response_token_ids)
    all_advantages.extend(advantages)

    stats["rewards"].extend(rewards)
    stats["non_stop_rate"].extend([fr != "stop" for fr in finish_reasons])
    stats["response_lengths"].extend([len(ids) for ids in response_token_ids])
    for rm in reward_metrics:
        for k, v in rm.items():
            stats.setdefault(f'reward_metrics/{k}', []).append(v)

episodes = {
    "all_query_token_ids": all_query_token_ids,
    "all_response_token_ids": all_responses_token_ids,
    "all_advantages": all_advantages,
}

return episodes, stats
```

How well does it work?

GRPO from the original paper

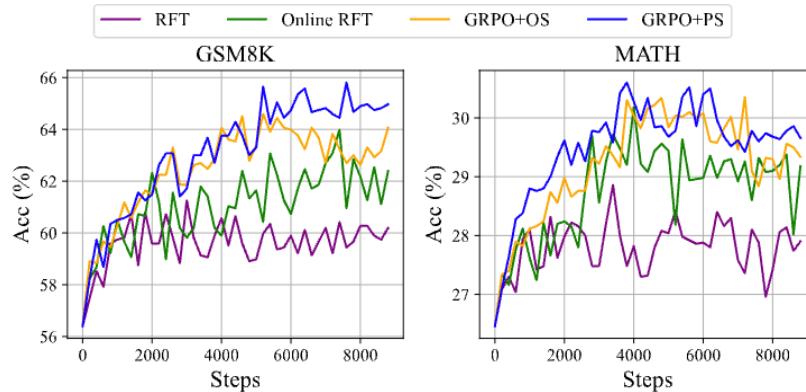


Figure 5 | Performance of the DeepSeekMath-Instruct 1.3B model, which was further trained using various methods, on two benchmarks.

Outperforms RFT (reinforcing correct answers),
with some gains from process supervision

We will get back to this later..

Thinking carefully about the GRPO objective..

The key difference in GRPO vs PPO: the advantage

$$A_i = \frac{r_i - \text{mean}(\{r_1, r_2, \dots, r_G\})}{\text{std}(\{r_1, r_2, \dots, r_G\})}.$$

Is this good? A minor RL detour..

13.4 REINFORCE with Baseline

The policy gradient theorem (13.5) can be generalized to include a comparison of the action value to an arbitrary *baseline* $b(s)$:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a (q_\pi(s, a) - b(s)) \nabla \pi(a|s, \boldsymbol{\theta}). \quad (13.10)$$

The baseline can be any function, even a random variable, as long as it does not vary with a ; the equation remains valid because the subtracted quantity is zero:

$$\sum_a b(s) \nabla \pi(a|s, \boldsymbol{\theta}) = b(s) \nabla \sum_a \pi(a|s, \boldsymbol{\theta}) = b(s) \nabla 1 = 0.$$

Baselining: We can subtract any state-dependent term from our rewards

The policy gradient theorem with baseline (13.10) can be used to derive an update rule using similar steps as in the previous section. The update rule that we end up with is a new version of REINFORCE that includes a general baseline:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \left(G_t - b(S_t) \right) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}. \quad (13.11)$$

GRPO doesn't use a “valid” baseline

The division by the stdev term is not a valid baseline that preserves unbiasedness.

What is an unbiased-gradient version of GRPO?

GRPO

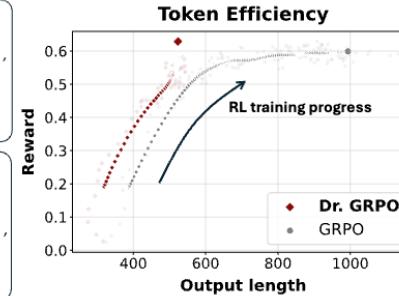
$$\frac{1}{G} \sum_{i=1}^G \frac{1}{|\mathbf{o}_i|} \sum_{t=1}^{|\mathbf{o}_i|} \left\{ \min \left[\frac{\pi_\theta(o_{i,t} | \mathbf{q}, \mathbf{o}_{i,< t})}{\pi_{\theta_{old}}(o_{i,t} | \mathbf{q}, \mathbf{o}_{i,< t})} \hat{A}_{i,t}, \text{clip} \left(\frac{\pi_\theta(o_{i,t} | \mathbf{q}, \mathbf{o}_{i,< t})}{\pi_{\theta_{old}}(o_{i,t} | \mathbf{q}, \mathbf{o}_{i,< t})}, 1 - \epsilon, 1 + \epsilon \right) \hat{A}_{i,t} \right] \right\},$$

where $\hat{A}_{i,t} = \frac{R(\mathbf{q}, \mathbf{o}_i) - \text{mean}(\{R(\mathbf{q}, \mathbf{o}_1), \dots, R(\mathbf{q}, \mathbf{o}_G)\})}{\text{std}(\{R(\mathbf{q}, \mathbf{o}_1), \dots, R(\mathbf{q}, \mathbf{o}_G)\})}$.

Dr. GRPO
GRPO Done Right (without bias)

$$\frac{1}{G} \sum_{i=1}^G \sum_{t=1}^{|\mathbf{o}_i|} \left\{ \min \left[\frac{\pi_\theta(o_{i,t} | \mathbf{q}, \mathbf{o}_{i,< t})}{\pi_{\theta_{old}}(o_{i,t} | \mathbf{q}, \mathbf{o}_{i,< t})} \hat{A}_{i,t}, \text{clip} \left(\frac{\pi_\theta(o_{i,t} | \mathbf{q}, \mathbf{o}_{i,< t})}{\pi_{\theta_{old}}(o_{i,t} | \mathbf{q}, \mathbf{o}_{i,< t})}, 1 - \epsilon, 1 + \epsilon \right) \hat{A}_{i,t} \right] \right\},$$

where $\hat{A}_{i,t} = R(\mathbf{q}, \mathbf{o}_i) - \text{mean}(\{R(\mathbf{q}, \mathbf{o}_1), \dots, R(\mathbf{q}, \mathbf{o}_G)\})$.



Also note the modification on the length-normalizer term on the left..

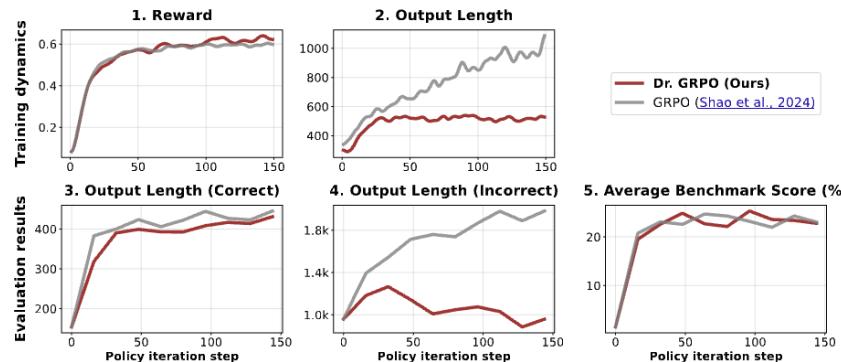
Length biases of GRPO

What do these terms do? **Stdev** – upweights too easy or hard questions.

Length normalization:

Response-level length bias: This arises from dividing by $|\mathbf{o}_i|$. For positive advantages ($\hat{A}_{i,t} > 0$, indicating a correct response), this bias results in greater gradient updates for shorter responses, leading the policy to favor brevity in correct answers. Conversely, for negative advantages ($\hat{A}_{i,t} < 0$, indicating an incorrect response), longer responses are penalized less due to their larger $|\mathbf{o}_i|$, causing the policy to prefer lengthier responses among incorrect ones.

What does the fix do?



Case studies

Let's look at a few representative open models with RLVR.



DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning

DeepSeek-AI
research@deepseek.com

KIMI K1.5:
SCALING REINFORCEMENT LEARNING WITH LLMs

TECHNICAL REPORT OF KIMI K1.5

Kimi Team



2025-05-14

Qwen3 Technical Report

Qwen Team

 <https://huggingface.co/Qwen>
 <https://modelscope.cn/organization/qwen>
 <https://github.com/QwenLM/Qwen3>

Deepseek R1

Central to many recent RLVR efforts
Lots of interesting details

Kimi K1.5

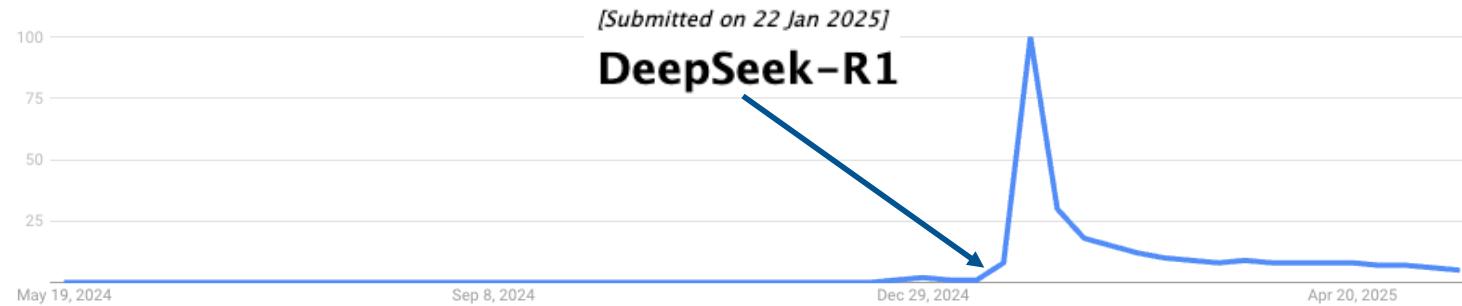
Contemporaneous to R1, RLVR
Complementary details to R1

Qwen 3

Most recent open reasoning model attempt
Low-data RLVR

Deepseek R1

The paper that launched a bit of a social phenomenon..



What's remarkable about R1?

- Performance exceeding OpenAI O1
- Open RL recipe (that is also pretty simple)
 - Ended speculations on the necessity of MCTS/PRMs
- SFT insights (both R1-zero and distil-r1)

Algorithm - GRPO

They build on the results of GRPO from DeepSeekMath..

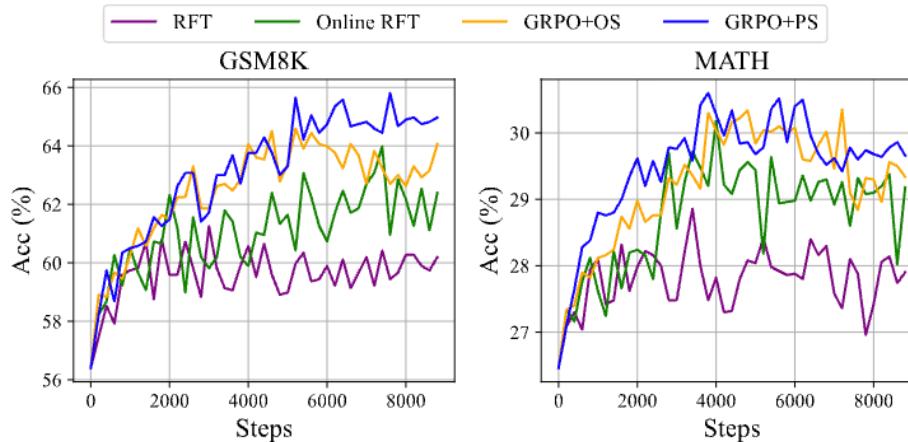


Figure 5 | Performance of the DeepSeekMath-Instruct 1.3B model, which was further trained using various methods, on two benchmarks.

But they do not use process supervision in R1 (more on this later..)

Controlled setting – R1 zero.

Setup:

Rewards

- Accuracy rewards (is it correct?)
- Format rewards (use thinking tags)

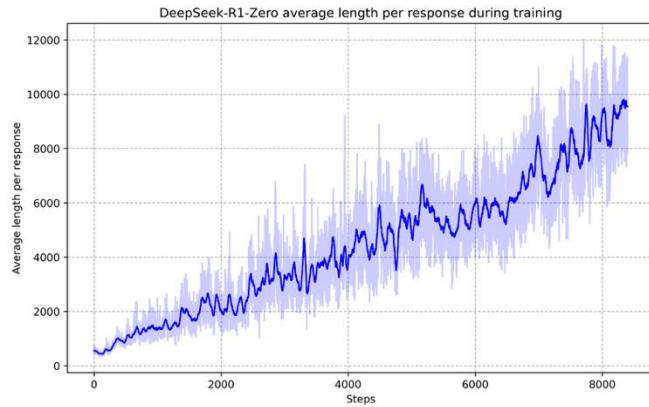
Data (not public)

Base model: Deepseek-V3

Results: a bit worse than Openai O1

Model	AIME 2024		MATH-500	GPQA Diamond	LiveCode Bench	CodeForces
	pass@1	cons@64	pass@1	pass@1	pass@1	rating
OpenAI-o1-mini	63.6	80.0	90.0	60.0	53.8	1820
OpenAI-o1-0912	74.4	83.3	94.8	77.3	63.4	1843
DeepSeek-R1-Zero	71.0	86.7	95.9	73.3	50.0	1444

Interesting phenomena (?)



Longer CoTs during training

‘aha’ moment?

Question: If $a > 1$, then the sum of the real solutions of $\sqrt{a - \sqrt{a+x}} = x$ is equal to

Response: <think>

To solve the equation $\sqrt{a - \sqrt{a+x}} = x$, let's start by squaring both ...

$$(\sqrt{a - \sqrt{a+x}})^2 = x^2 \implies a - \sqrt{a+x} = x^2.$$

Rearrange to isolate the inner square root term:

$$(a - x^2)^2 = a + x \implies a^2 - 2ax^2 + (x^2)^2 = a + x \implies x^4 - 2ax^2 - x + (a^2 - a) = 0$$

...

Wait, wait. Wait. That's an aha moment I can flag here.

Let's reevaluate this step-by-step to identify if the correct sum can be ...

We started with the equation:

$$\sqrt{a - \sqrt{a+x}} = x$$

First, let's square both sides:

$$a - \sqrt{a+x} = x^2 \implies \sqrt{a+x} = a - x^2$$

Next, I could square both sides again, treating the equation: ...

...

But maybe a bit overstated

From some follow-up analyses in Dr. GRPO..

Length due to
biased objective?

Base model
already has ‘aha’

GRPO

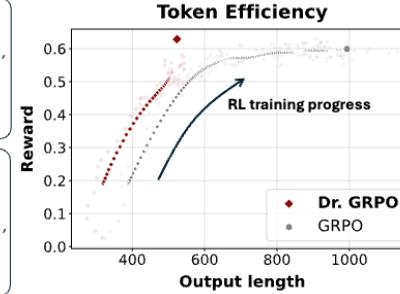
$$\frac{1}{G} \sum_{i=1}^G \frac{1}{|\mathbf{o}_i|} \sum_{t=1}^{|\mathbf{o}_i|} \left\{ \min \left[\frac{\pi_\theta(o_{i,t} | \mathbf{q}, \mathbf{o}_{i,<t})}{\pi_{\theta_{old}}(o_{i,t} | \mathbf{q}, \mathbf{o}_{i,<t})} \right] \hat{A}_{i,t}, \text{clip} \left(\frac{\pi_\theta(o_{i,t} | \mathbf{q}, \mathbf{o}_{i,<t})}{\pi_{\theta_{old}}(o_{i,t} | \mathbf{q}, \mathbf{o}_{i,<t})}, 1 - \epsilon, 1 + \epsilon \right) \hat{A}_{i,t} \right] \right\},$$

where $\hat{A}_{i,t} = \frac{R(\mathbf{q}, \mathbf{o}_i) - \text{mean}(\{R(\mathbf{q}, \mathbf{o}_1), \dots, R(\mathbf{q}, \mathbf{o}_G)\})}{\text{std}(\{R(\mathbf{q}, \mathbf{o}_1), \dots, R(\mathbf{q}, \mathbf{o}_G)\})}$.

Dr. GRPO
GRPO Done Right (without bias)

$$\frac{1}{G} \sum_{i=1}^G \sum_{t=1}^{|\mathbf{o}_i|} \left\{ \min \left[\frac{\pi_\theta(o_{i,t} | \mathbf{q}, \mathbf{o}_{i,<t})}{\pi_{\theta_{old}}(o_{i,t} | \mathbf{q}, \mathbf{o}_{i,<t})} \right] \hat{A}_{i,t}, \text{clip} \left(\frac{\pi_\theta(o_{i,t} | \mathbf{q}, \mathbf{o}_{i,<t})}{\pi_{\theta_{old}}(o_{i,t} | \mathbf{q}, \mathbf{o}_{i,<t})}, 1 - \epsilon, 1 + \epsilon \right) \hat{A}_{i,t} \right] \right\},$$

where $\hat{A}_{i,t} = R(\mathbf{q}, \mathbf{o}_i) - \text{mean}(\{R(\mathbf{q}, \mathbf{o}_1), \dots, R(\mathbf{q}, \mathbf{o}_G)\})$.



Question(a): Find the least positive integer n such that $1/\sin 45^\circ \sin 46^\circ + 1/\sin 47^\circ \sin 48^\circ + \dots + 1/\sin 133^\circ \sin 134^\circ = 1/\sin n^\circ$.

Response

First, I notice that the left-hand side of the equation has the form $\sum_{k=1}^n \frac{1}{\sin a_k \sin a_{k+1}}$, where $a_1 = 45^\circ, a_2 = 47^\circ, \dots, a_{90} = 133^\circ$. I also notice that $\sin a_{k+1} = \sin(a_k + 2^\circ)$.

...

Simplifying, I get $\sin(3a) = 3 \sin a - 4 \sin^3 a$.

Aha! I can use this to get $\sin(a+2b) = \sin((a+b)+b) = 3 \sin(a+b) - 4 \sin^3(a+b)$...

This looks messy, but I notice that it has the form $\sin a \cos b + \cos a \sin b$ again. This gives me an idea. Maybe I can use this identity repeatedly to simplify the left-hand side of the equation.

...

Pushing performance further – R1

R1 vs R1-zero

Key differences:

- SFT initialization
- Language consistency reward for CoT
- Non-verifiable rewards (in stage 2)



SFT initialization..

Unlike DeepSeek-R1-Zero, to prevent the early unstable cold start phase of RL training from the base model, for DeepSeek-R1 we construct and collect a small amount of long CoT data to fine-tune the model as the initial RL actor. To collect such data, we have explored several approaches: using few-shot prompting with a long CoT as an example, directly prompting models to generate detailed answers with reflection and verification, gathering DeepSeek-R1-Zero outputs in a readable format, and refining the results through post-processing by human annotators.

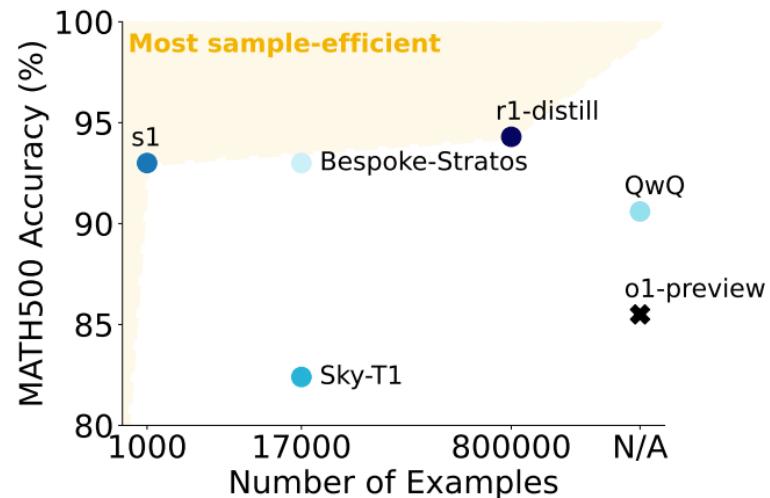
Start with plain ol (long) CoT, then maybe additional verification (vague)

- **Claimed Benefit:** interpretability
- **Note:** Origins on the data not quite clear..

SFT for reasoning / math..

Even a small number of samples is effective for bootstrapping reasoning from LMs

Model	# ex.	AIME 2024	MATH- 500	GPQA Diamond
API only				
o1-preview	N.A.	44.6	85.5	73.3
o1-mini	N.A.	70.0	90.0	60.0
o1	N.A.	74.4	94.8	77.3
Gemini 2.0	N.A.	60.0	N.A.	N.A.
Flash Think.				
Open Weights				
Qwen2.5- 32B-Instruct	N.A.	26.7	84.0	49.0
QwQ-32B	N.A.	50.0	90.6	65.2
r1	>>800K	79.8	97.3	71.5
r1-distill	800K	72.6	94.3	62.1
Open Weights and Open Data				
Sky-T1	17K	43.3	82.4	56.8
Bespoke-32B	17K	63.3	93.0	58.1
s1 w/o BF	1K	50.0	92.6	56.6
s1-32B	1K	56.7	93.0	59.6



1k Math and science questions + Long CoTs from Gemini / r1

RL step

The RL part is basically the same..

Minor difference: additional language consistency loss

clear solutions. During the training process, we observe that CoT often exhibits language mixing, particularly when RL prompts involve multiple languages. To mitigate the issue of language mixing, we introduce a language consistency reward during RL training, which is calculated as the proportion of target language words in the CoT. Although ablation experiments show that such alignment results in a slight degradation in the model's performance, this reward aligns with human preferences, making it more readable. Finally, we combine the accuracy of reasoning tasks and the reward for language consistency by directly summing them to form the final reward. We then apply RL training on the fine-tuned model until it achieves convergence on reasoning tasks.

The note on language switching is interesting.. RL naturally leads to mixed languages?

SFT/RLHF

The usual post-training process happens *after* RLVR / reasoning RL.

SFT step, 2 epochs.

Reasoning data – non-verifiable tasks ('write a proof of X'), use V3 as a judge (600k)
Non-reasoning data – V3 SFT dataset (200k)

RLHF step

Re-use R1-zero style reasoning RLHF in here
Non-verifiable tasks - V3 RLHF pipeline. Still uses GRPO (for RLHF)

How well does R1 work?

It's pretty good (but you knew that)

Benchmark (Metric)		Claude-3.5-Sonnet-1022	GPT-4o 0513	DeepSeek V3	OpenAI o1-mini	OpenAI o1-1217	DeepSeek R1
English	Architecture	-	-	MoE	-	-	MoE
	# Activated Params	-	-	37B	-	-	37B
	# Total Params	-	-	671B	-	-	671B
	MMLU (Pass@1)	88.3	87.2	88.5	85.2	91.8	90.8
	MMLU-Redux (EM)	88.9	88.0	89.1	86.7	-	92.9
	MMLU-Pro (EM)	78.0	72.6	75.9	80.3	-	84.0
	DROP (3-shot F1)	88.3	83.7	91.6	83.9	90.2	92.2
	IF-Eval (Prompt Strict)	86.5	84.3	86.1	84.8	-	83.3
	GPQA Diamond (Pass@1)	65.0	49.9	59.1	60.0	75.7	71.5
	SimpleQA (Correct)	28.4	38.2	24.9	7.0	47.0	30.1
Code	FRAMES (Acc.)	72.5	80.5	73.3	76.9	-	82.5
	AlpacaEval2.0 (LC-winrate)	52.0	51.1	70.0	57.8	-	87.6
	ArenaHard (GPT-4-1106)	85.2	80.4	85.5	92.0	-	92.3
	LiveCodeBench (Pass@1-COT)	38.9	32.9	36.2	53.8	63.4	65.9
	Codeforces (Percentile)	20.3	23.6	58.7	93.4	96.6	96.3
Math	Codeforces (Rating)	717	759	1134	1820	2061	2029
	SWE Verified (Resolved)	50.8	38.8	42.0	41.6	48.9	49.2
	Aider-Polyglot (Acc.)	45.3	16.0	49.6	32.9	61.7	53.3
	AIME 2024 (Pass@1)	16.0	9.3	39.2	63.6	79.2	79.8
Chinese	MATH-500 (Pass@1)	78.3	74.6	90.2	90.0	96.4	97.3
	CNMO 2024 (Pass@1)	13.1	10.8	43.2	67.6	-	78.8
	CLUEWSC (EM)	85.4	87.9	90.9	89.9	-	92.8
	C-Eval (EM)	76.7	76.0	86.5	68.9	-	91.8
	C-SimpleQA (Correct)	55.4	58.7	68.0	40.3	-	63.7

Distillation – can we get non-reasoning models to reason?

Pipeline

Have R1 generate CoT traces (800k!)

Teach Qwen 2.5 via distillation

Model	AIME 2024		MATH-500	GPQA Diamond	LiveCode Bench	CodeForces
	pass@1	cons@64	pass@1	pass@1	pass@1	rating
GPT-4o-0513	9.3	13.4	74.6	49.9	32.9	759
Claude-3.5-Sonnet-1022	16.0	26.7	78.3	65.0	38.9	717
OpenAI-o1-mini	63.6	80.0	90.0	60.0	53.8	1820
QwQ-32B-Preview	50.0	60.0	90.6	54.5	41.9	1316
DeepSeek-R1-Distill-Qwen-1.5B	28.9	52.7	83.9	33.8	16.9	954
DeepSeek-R1-Distill-Qwen-7B	55.5	83.3	92.8	49.1	37.6	1189
DeepSeek-R1-Distill-Qwen-14B	69.7	80.0	93.9	59.1	53.1	1481
DeepSeek-R1-Distill-Qwen-32B	72.6	83.3	94.3	62.1	57.2	1691
DeepSeek-R1-Distill-Llama-8B	50.4	80.0	89.1	49.0	39.6	1205
DeepSeek-R1-Distill-Llama-70B	70.0	86.7	94.5	65.2	57.5	1633

Other, relevant observations

There is a whole unsuccessful attempts section

PRMs
(PRM800k, DeepseekMath)

Process Reward Model (PRM) PRM is a reasonable method to guide the model toward better approaches for solving reasoning tasks ([Lightman et al., 2023](#); [Uesato et al., 2022](#); [Wang et al., 2023](#)). However, in practice, PRM has three main limitations that may hinder its ultimate success. First, it is challenging to explicitly define a fine-grain step in general reasoning. Second, determining whether the current intermediate step is correct is a challenging task. Automated annotation using models may not yield satisfactory results, while manual annotation is not conducive to scaling up. Third, once a model-based PRM is introduced, it inevitably leads to reward hacking ([Gao et al., 2022](#)), and retraining the reward model needs additional training resources and it complicates the whole training pipeline. In conclusion, while PRM demonstrates a good ability to rerank the top-N responses generated by the model or assist in guided search ([Snell et al., 2024](#)), its advantages are limited compared to the additional computational overhead it introduces during the large-scale reinforcement learning process in our experiments.

MCTS

Monte Carlo Tree Search (MCTS) Inspired by AlphaGo ([Silver et al., 2017b](#)) and AlphaZero ([Silver et al., 2017a](#)), we explored using Monte Carlo Tree Search (MCTS) to enhance test-time compute scalability. This approach involves breaking answers into smaller parts to allow the model to explore the solution space systematically. To facilitate this, we prompt the model to generate multiple tags that correspond to specific reasoning steps necessary for the search. For training, we first use collected prompts to find answers via MCTS guided by a pre-trained value model. Subsequently, we use the resulting question-answer pairs to train both the actor model and the value model, iteratively refining the process.

However, this approach encounters several challenges when scaling up the training. First, unlike chess, where the search space is relatively well-defined, token generation presents an

Kimi K1.5

KIMI K1.5: SCALING REINFORCEMENT LEARNING WITH LLMs

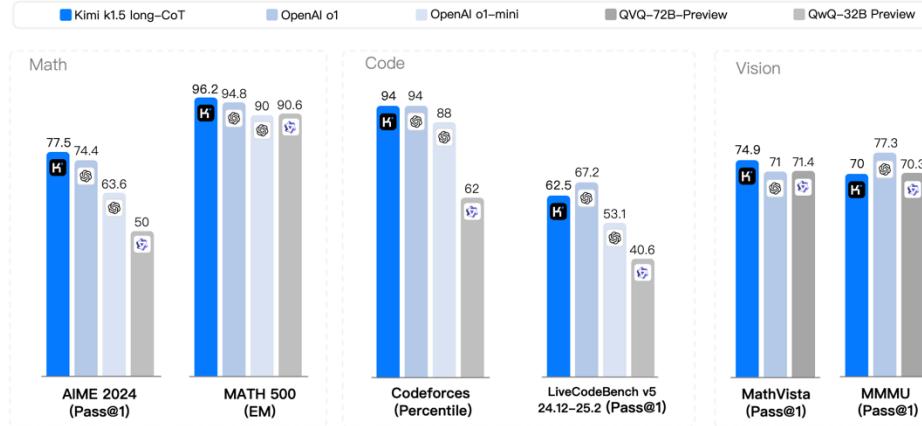
TECHNICAL REPORT OF KIMI K1.5

Kimi Team

Why do we study this one?

- Released at the same time as R1
- Also beats o1 using RL

Long COT reasoning strategy



Key steps

- Dataset construction (difficulty filtering)
- SFT (for long COT)
- RL (with their own policy gradient loss)

Data curation + SFT

Data curation

- Standard curation across math-style settings, balancing topics

To achieve diverse coverage in the prompt set, we employ automatic filters to select questions that require rich reasoning and are straightforward to evaluate. Our dataset includes problems from various domains, such as STEM fields, competitions, and general reasoning tasks, incorporating both text-only and image-text question-answering data. Furthermore, we developed a tagging system to categorize prompts by domain and discipline, ensuring balanced representation across different subject areas ([M. Li et al. 2023](#); [W. Liu et al. 2023](#)).

- Exclude multiple choice / true false (false positives)
- Select only examples that models fail on best-of-8

We adopt a model-based approach that leverages the model's own capacity to adaptively assess the difficulty of each prompt. Specifically, for every prompt, an SFT model generates answers ten times using a relatively high sampling temperature. The pass rate is then calculated and used as a proxy for the prompt's difficulty—the lower the pass rate, the higher the difficulty. This approach allows difficulty evaluation to be aligned with the model's intrinsic capabilities, making it highly effective for RL training. By leveraging this method, we can prefilter most trivial cases and easily explore different sampling strategies during RL training.

SFT – little description, just described as ‘prompt engineering’ (distillation?)

Kimi RL

In kimi – reference based reward model, so the optimization problem is

$$\max_{\theta} \mathbb{E}_{(x,y^*) \sim \mathcal{D}} [\mathbb{E}_{(y,z) \sim \pi_{\theta}} [r(x, y, y^*)] - \tau \text{KL}(\pi_{\theta}(x) || \pi_{\theta_i}(x))] ,$$

RL algorithm inspired by DPO-type derivation.

$$r(x, y, y^*) - \tau \log Z = \tau \log \frac{\pi^*(y, z|x)}{\pi_{\theta_i}(y, z|x)} .$$

(Nonparametric assumption + solve for r)

$$L(\theta) = \mathbb{E}_{(x,y^*) \sim \mathcal{D}} \left[\mathbb{E}_{(y,z) \sim \pi_{\theta_i}} \left[\left(r(x, y, y^*) - \tau \log Z - \tau \log \frac{\pi_{\theta}(y, z|x)}{\pi_{\theta_i}(y, z|x)} \right)^2 \right] \right] .$$

Use squared loss as a surrogate

Baselined policy gradient w/ regularization

$$\frac{1}{k} \sum_{j=1}^k \left(\nabla_{\theta} \log \pi_{\theta}(y_j, z_j|x) (r(x, y_j, y^*) - \bar{r}) - \frac{\tau}{2} \nabla_{\theta} \left(\log \frac{\pi_{\theta}(y_j, z_j|x)}{\pi_{\theta_i}(y_j, z_j|x)} \right)^2 \right) .$$

Length control in Kimi

The kimi objective doesn't have the same GRPO length bias problem.

.. But they want to further compress the CoTs.

For each batch, they have a *length reward*

$$\text{len_reward}(i) = \begin{cases} \lambda & \text{If } r(x, y_i, y^*) = 1 \\ \min(0, \lambda) & \text{If } r(x, y_i, y^*) = 0 \end{cases}, \quad \text{where } \lambda = 0.5 - \frac{\text{len}(i) - \text{min_len}}{\text{max_len} - \text{min_len}}.$$

How should I interpret this?

- The lambda is in [0.5, -0.5], with longer sequences in a group being negative
- Correct answers are incentivized to be short
- Incorrect answers are incentivized to be shorter than the center of the range of rollouts
(They also only enable this later on in training, due to its effects on perf)

Additional details

Curriculum

- Assign difficulty labels to dataset, go from easy to hard
- Sample problems proportional to $(1-\text{success_rate})$ to avoid repeating solved ones

Rewards

- For code – take problems with ground truth solutions, generate new test cases
- For math – 800k samples to train a CoT reward model for answer equivalence checks

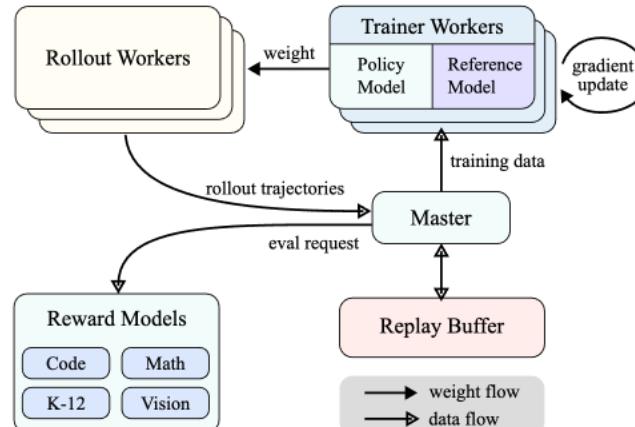
During our manual spot checks, the Classic RM achieved an accuracy of approximately **84.4**, while the Chain-of-Thought RM reached **98.5** accuracy. In the RL training process, we adopted the Chain-of-Thought RM to ensure more correct feedback.

RL Infra

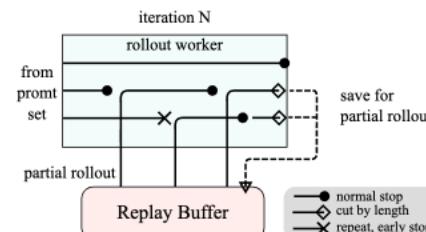
Systems and utilization aspects of RL are very important.

Why is RL hard to make efficient?

- On policy = rollouts, which means (slow) inference
- Switching from training to rollouts often means different frameworks
- Long CoTs can make batches very uneven.



(a) System overview



(b) Partial Rollout

Kimi's RL setup in detail

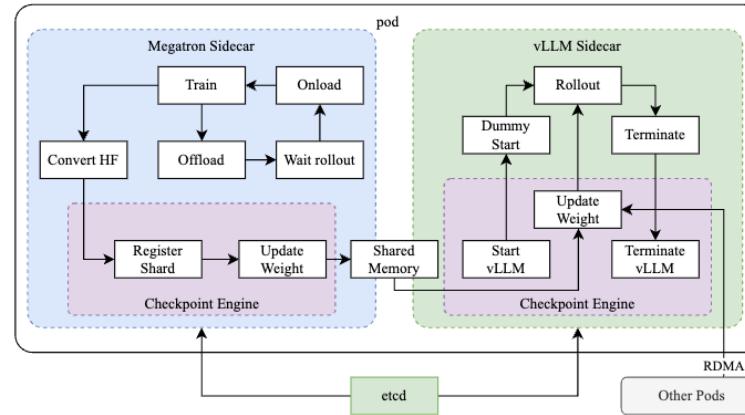
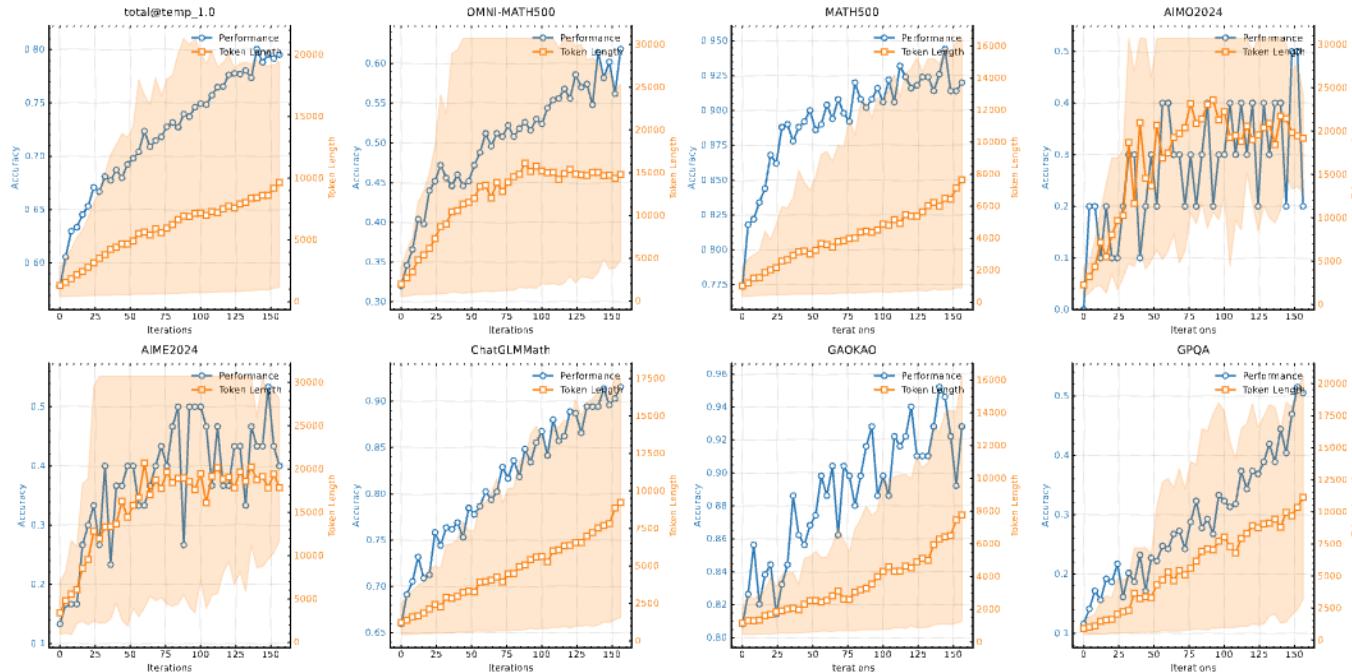


Figure 4: Hybrid Deployment Framework

- **Training Phase:** At the outset, Megatron ([Shoeybi et al. 2020](#)) and vLLM ([Kwon et al. 2023](#)) are executed within separate containers, encapsulated by a shim process known as checkpoint-engine ([Section 2.6.3](#)). Megatron commences the training procedure. After the training is completed, Megatron offloads the GPU memory and prepares to transfer current weights to vLLM.
- **Inference Phase:** Following Megatron’s offloading, vLLM starts with dummy model weights and updates them with the latest ones transferred from Megatron via Mooncake ([Qin et al. 2024](#)). Upon completion of the rollout, the checkpoint-engine halts all vLLM processes.
- **Subsequent Training Phase:** Once the memory allocated to vLLM is released, Megatron onloads the memory and initiates another round of training.

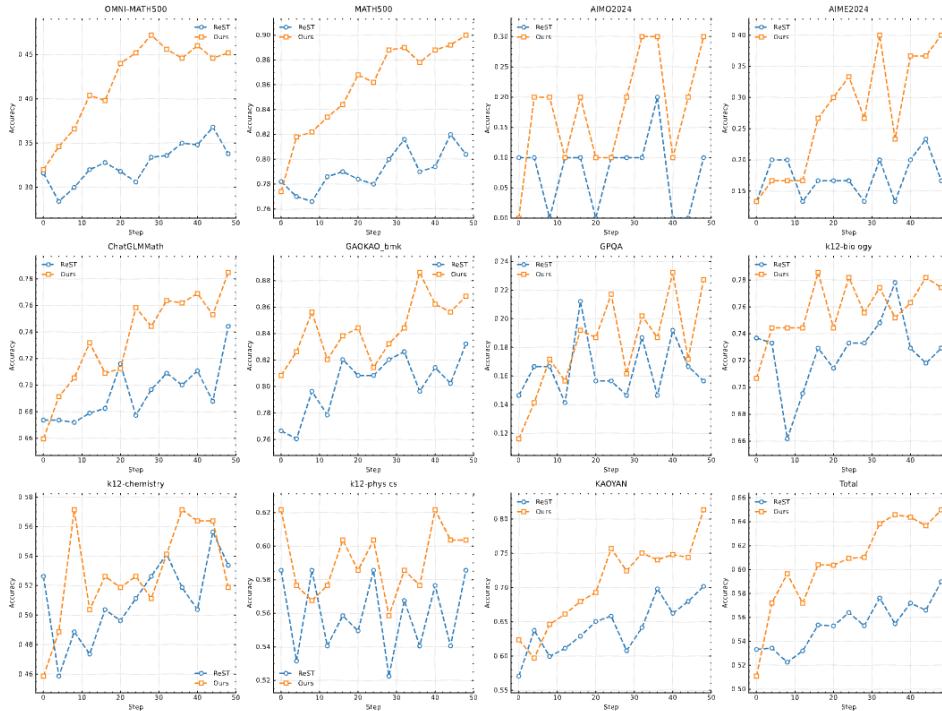
Scaling results

We already know this roughly matches / beats o1. Other interesting results..



For a small model on just the math data

Ablation / comparison to expert iteration



Could we avoid RL-style negative gradients and just learn from positives?

Final case study – Qwen 3



	Qwen3-235B-A22B MoE	Qwen3-32B Dense	OpenAI-o1 2024-12-17	Deepseek-R1	Grok 3 Beta Think	Gemini2.5-Pro	OpenAI-o3-mini Medium
ArenaHard	95.6	93.8	92.1	93.2	-	96.4	89.0
AIME'24	85.7	81.4	74.3	79.8	83.9	92.0	79.6
AIME'25	81.5	72.9	79.2	70.0	77.3	86.7	74.8
LiveCodeBench v5, 2024.10-2025.02	70.7	65.7	63.9	64.3	70.6	70.4	66.3
CodeForces Elo Rating	2056	1977	1891	2029	-	2001	2036
Aider Pass@2	61.8	50.2	61.7	56.9	53.3	72.9	53.8
LiveBench 2024-11-25	77.1	74.9	75.7	71.6	-	82.4	70.0
BFCL v3	70.8	70.3	67.8	56.9	-	62.9	64.6
MultilF 8 Languages	71.9	73.0	48.8	67.7	-	77.8	48.4

Better than o1 and R1 (though comes later)

.. But interesting scaling and data results

Overall picture

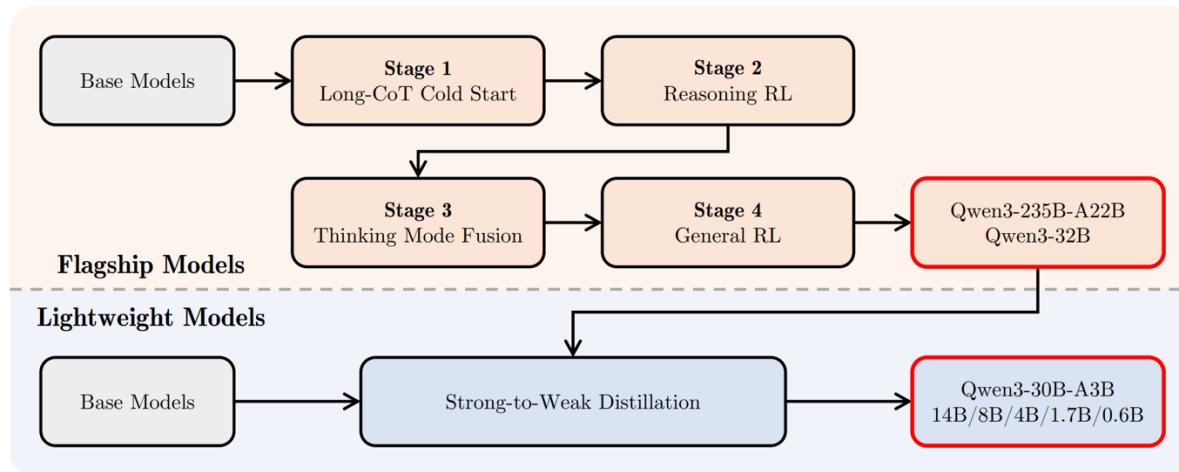


Figure 1: Post-training pipeline of the Qwen3 series models.

RLHF comes *after* reasoning RL (like r1), distillation after that.

SFT + Reasoning RL

We know the playbook by now, and Qwen uses a lot of this..

- Filtering for difficulty (by best-of-n, like kimi)
 - Remove things that the model gets right w/o CoT
 - Remove things too similar to validation data
- Manual filtering for the quality of CoTs (guessing vs getting it right)
- RL with GRPO on **only 3995 examples**

Qwen 3 specific new stuff.

Thinking mode fusion – controlling the length of the CoT.

1. Mix non-thinking and thinking data with tags

Thinking Mode	Non-Thinking Mode
< im_start >user {query} /think< im_end > < im_start >assistant <think> {thinking_content} </think> {response}< im_end >	< im_start >user {query} /no_think< im_end > < im_start >assistant <think> </think> {response}< im_end >

2. Early stopping termination via a special string

implementing budget control over the model's thinking process. Specifically, when the length of the model's thinking reaches a user-defined threshold, we manually halt the thinking process and insert the stop-thinking instruction: "Considering the limited time by the user, I have to give the solution based on the thinking directly now.\n</think>.\n\n". After this instruction is inserted the model proceeds to generate a final response based on its accumulated reasoning up to that point. It is worth noting that this ability is not explicitly trained but emerges naturally as a result of applying Thinking Mode Fusion.

Test time scaling

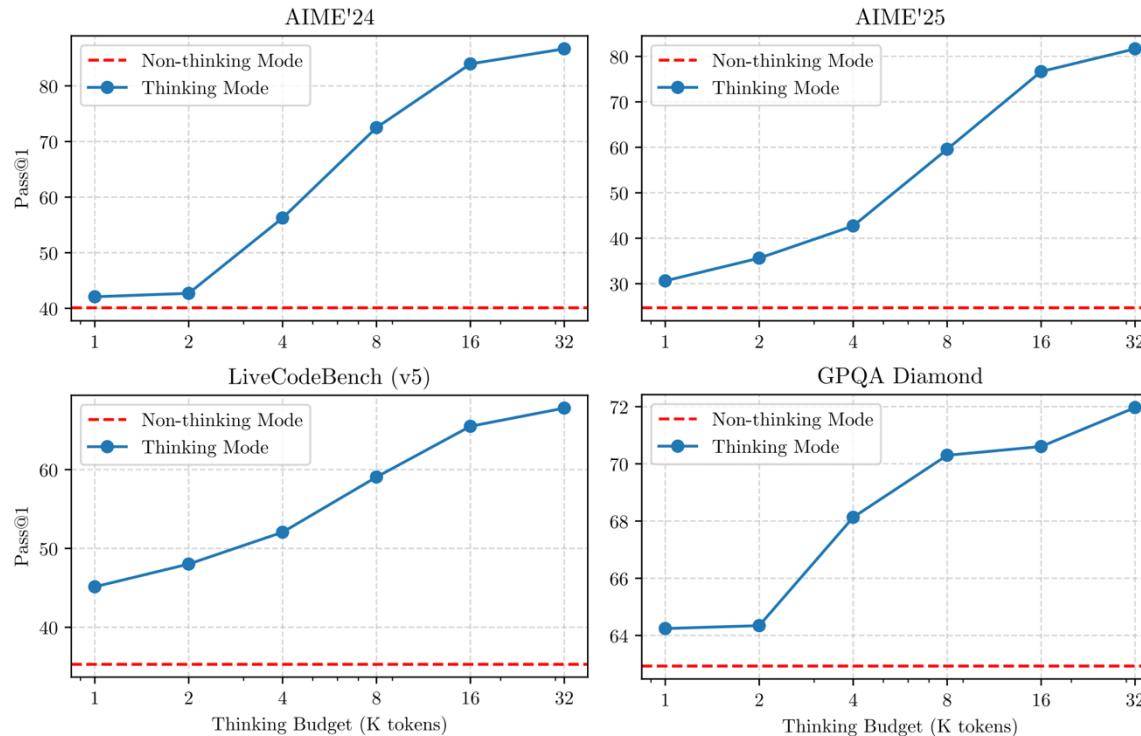


Figure 2: Performance of Qwen3-235B-A22B with respect to the thinking budget.

Composition of the different stages

	Benchmark	Stage 2 Reasoning RL		Stage 3 Thinking Mode Fusion		Stage 4 General RL	
		Thinking		Thinking	Non-Thinking	Thinking	Non-Thinking
<i>General Tasks</i>	LiveBench 2024-11-25	68.6		70.9 _{+2.3}	57.1	74.9 _{+4.0}	59.8 _{+2.8}
	Arena-Hard	86.8		89.4 _{+2.6}	88.5	93.8 _{+4.4}	92.8 _{+4.3}
	CounterFactQA*	50.4		61.3 _{+10.9}	64.3	68.1 _{+6.8}	66.4 _{+2.1}
<i>Instruction & Format Following</i>	IFEval strict prompt	73.0		78.4 _{+5.4}	78.4	85.0 _{+6.6}	83.2 _{+4.8}
	Multi-IF	61.4		64.6 _{+3.2}	65.2	73.0 _{+8.4}	70.7 _{+5.5}
	LengthCtrl*	62.6		70.6 _{+8.0}	84.9	73.5 _{+2.9}	87.3 _{+2.4}
	ThinkFollow*	-			88.7		98.9 _{+10.2}
<i>Agent</i>	BFCL v3	69.0		68.4 _{-0.6}	61.5	70.3 _{+1.9}	63.0 _{+1.5}
	ToolUse*	63.3		70.4 _{+7.1}	73.2	85.5 _{+15.1}	86.5 _{+13.3}
<i>Knowledge & STEM</i>	MMLU-Redux	91.4		91.0 _{-0.4}	86.7	90.9 _{-0.1}	85.7 _{-1.0}
	GPQA-Diamond	68.8		69.0 _{+0.2}	50.4	68.4 _{-0.6}	54.6 _{+4.3}
<i>Math & Coding</i>	AIME'24	83.8		81.9 _{-1.9}	28.5	81.4 _{-0.5}	31.0 _{+2.5}
	LiveCodeBench v5	68.4		67.2 _{-1.2}	31.1	65.7 _{-1.5}	31.3 _{+0.2}

Note that math/stem abilities go down (a bit) with general purpose RLHF

Recap for today

Overoptimization is a problem.. RL in narrow domains is one solution

GRPO is simple (but with some flaws), and enables RLVR

Lots of successful recipes in the wild (R1, Kimi 1.5, Qwen3)