REINFORCEMENT LEARNING VIA SELF-DISTILLATION

Jonas Hübotter[1] **Frederike Lübeck**[*, 1, 2] **Lejs Behric**[*, 1] **Anton Baumann**[*, 1]
Marco Bagatella[1, 2] **Daniel Marta**[1] **Ido Hakimi**[1] **Idan Shenfeld**[3]
Thomas Kleine Buening[1] **Carlos Guestrin**[4] **Andreas Krause**[1]
[1]ETH Zurich [2]Max Planck Institute for Intelligent Systems [3]MIT [4]Stanford

ABSTRACT

Large language models are increasingly post-trained with reinforcement learning in verifiable domains such as code and math. Yet, current methods for reinforcement learning with verifiable rewards (RLVR) learn only from a scalar outcome reward per attempt, creating a severe credit-assignment bottleneck. Many verifiable environments actually provide rich textual feedback, such as runtime errors or judge evaluations, that explain *why* an attempt failed. We formalize this setting as reinforcement learning with rich feedback and introduce **Self-Distillation Policy Optimization** (**SDPO**), which converts tokenized feedback into a dense learning signal without any external teacher or explicit reward model. SDPO treats the current model conditioned on feedback as a self-teacher and distills its feedback-informed next-token predictions back into the policy. In this way, SDPO leverages the model's ability to retrospectively identify its own mistakes in-context. Across scientific reasoning, tool use, and competitive programming on LiveCodeBench v6, SDPO improves sample efficiency and final accuracy over strong RLVR baselines. Notably, SDPO also outperforms baselines in standard RLVR environments that only return scalar feedback by using successful rollouts as implicit feedback for failed attempts.

1 INTRODUCTION

Progress in deep reinforcement learning has shown that iterating on experience—acting, receiving feedback, and updating a policy—can unlock capabilities that are difficult to obtain from static supervision alone (Mnih et al., 2015; Silver et al., 2016; 2017; Berner et al., 2019). The same theme now appears in large language models (LLMs): large-scale post-training with reinforcement learning (RL) has substantially improved performance on reasoning-heavy tasks, especially in settings with programmatic or otherwise verifiable evaluation (Jaech et al., 2024; Guo et al., 2025; Kimi et al., 2025; Olmo et al., 2025).

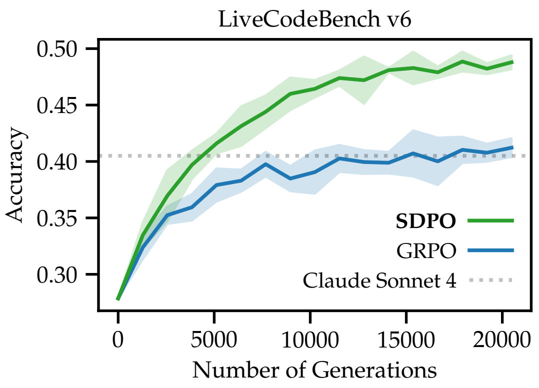


Figure 1: **SDPO substantially outperforms an improved version of Group Relative Policy Optimization (GRPO) on LCB v6 with Qwen3-8B.** Further, SDPO achieves GRPO's final accuracy in $4\times$ fewer generations. Claude Sonnet 4 is the strongest instruct model on the public LCBv6 leaderboard. Shaded regions show the standard deviation across 3 seeds.

Nevertheless, the dominant RL recipe for LLM post-training remains bottlenecked by credit assignment. Most current approaches operate in the setting of reinforcement learning with verifiable rewards (RLVR): given a question x, the model samples an answer $y \sim \pi_\theta(\cdot \mid x)$ and receives a scalar reward $r \in \mathbb{R}$, often binary (e.g., unit-tests pass/fail in code generation). Modern policy gradient RLVR methods such as Group Relative Policy Optimization (GRPO; Shao et al., 2024) estimate advantages from these sparse outcome rewards. Furthermore, when all rollouts in a group receive the same (often zero) reward, GRPO advantages collapse to zero and learning stalls.

[*]Equal second authorship. Correspondence to `jonas.huebotter@inf.ethz.ch`.

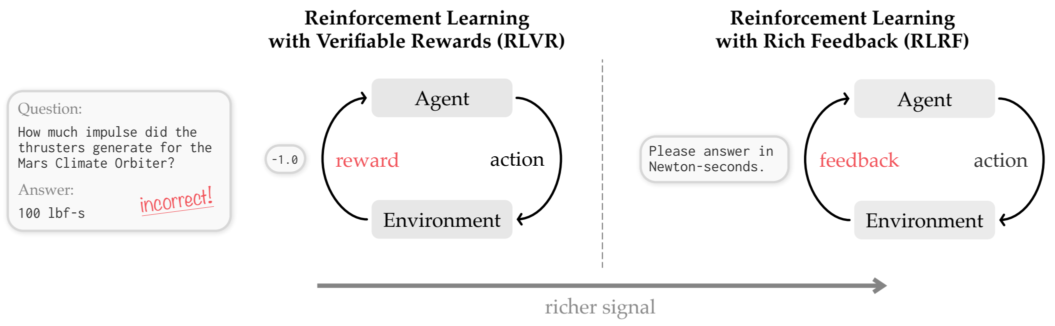

Reinforcement Learning with Verifiable Rewards (RLVR)
Reinforcement Learning with Rich Feedback (RLRF)
Question:
How much impulse did the thrusters generate for the Mars Climate Orbiter?
Answer:
100 lbf-s
incorrect!
-1.0
Agent
reward
action
Environment
Please answer in Newton-seconds.
Agent
feedback
action
Environment
richer signal

Figure 2: **Comparison of RLVR and RLRF settings.** In Reinforcement Learning with Verifiable Rewards (RLVR), the agent learns from a scalar reward, which often acts as an information bottleneck by masking the underlying environment state. In contrast, Reinforcement Learning with Rich Feedback (RLRF) utilizes tokenized feedback. This provides a significantly richer signal than a scalar reward, as the feedback can encapsulate a reward and observations of the state (e.g., runtime errors from a code environment or feedback from a judge).

To overcome this sparsity, one might prefer distillation from a strong teacher (Guo et al., 2025; Yang et al., 2025; Lu & Thinking Machines Lab, 2025; Guha et al., 2026), which provides dense, token-level supervision. However, strong teachers are often unavailable in online learning, where the goal is to raise the capability ceiling beyond existing models.

In this work, we argue that the key limitation is not RL per se, but the information bottleneck imposed by scalar outcome rewards. Many verifiable environments expose *rich tokenized feedback* beyond scalar rewards r, such as runtime errors, failing unit tests, or evaluations from an LLM judge. This feedback not only reveals *whether* a rollout was wrong, but also *what* went wrong. We formalize this more general setting as **Reinforcement Learning with Rich Feedback** (**RLRF**) and illustrate its difference to RLVR in Figure 2. Here, feedback can be any tokenized representation of any state reached by an agentic system. The central question becomes: how can we convert rich feedback into effective credit assignment without requiring external supervision from a strong teacher?

Our starting point is the observation that LLMs already possess a powerful mechanism for using feedback: in-context learning (Brown et al., 2020; Wei et al., 2022). When conditioned on feedback, the same model can often identify plausible mistakes and propose a corrected approach. A common example of such feedback is the summary of failed test cases on coding platforms like LeetCode (Figure 8 in the appendix). Many recent works leverage this capability to iteratively generate corrections (Chen et al., 2021; Madaan et al., 2023; Shinn et al., 2023; Yao et al., 2024; Yuksekgonul et al., 2025; Lee et al., 2025). In contrast, we use the current policy as a “self-teacher” that, rather than sampling a new response, re-evaluates the *existing* rollout after receiving rich feedback. Including the feedback in-context transforms the model’s next-token distribution, allowing the self-teacher to agree or disagree with the student’s original choices at specific tokens. This yields dense, logit-level credit assignment. Crucially, this mechanism incurs no sampling overhead: we simply re-compute the log-probabilities of the original attempt under the self-teacher’s feedback-augmented context.

Building on this idea, we introduce **Self-Distillation Policy Optimization** (**SDPO**), an on-policy algorithm that performs RL via self-distillation. SDPO samples rollouts from the current policy, obtains rich environment feedback, and then minimizes a logit-level distillation loss that matches the current policy’s next-token distribution to that of the self-teacher. Conceptually, SDPO addresses the central limitation of applying distillation to online learning: the absence of a stronger external teacher. Instead of relying on a fixed teacher, SDPO leverages the model’s ability to recognize its own mistakes in hindsight. By conditioning the current policy on the rich feedback it just received, we construct a self-teacher that provides the dense supervision of distillation while retaining the exploration benefits of on-policy RL. Table 2 in the appendix summarizes how this positions SDPO relative to RLVR and distillation baselines. We include a summary of related work in Appendix G.

We show that SDPO is a policy gradient algorithm whose advantages are estimated using the self-teacher. This enables the implementation of SDPO with minor changes to standard RLVR pipelines, simply by swapping out advantages.

Summary of evaluation results:

- **Learning without rich feedback** (§3): We evaluate standard RLVR environments that do not return any feedback beyond scalar rewards. Here, SDPO treats successful attempts sampled in the current batch as "feedback" for failed attempts on the same question. We perform training runs on scientific reasoning and tool use, starting with Qwen3-8B and Olmo3-7B-Instruct. We find that SDPO outperforms a strong GRPO baseline that integrates recent improvements: 68.8% vs. 64.1% final accuracy on aggregate. SDPO achieves higher accuracy with up to $7\times$ shorter generation lengths compared to GRPO, demonstrating that effective reasoning need not be verbose.
- **Learning with rich feedback** (§4): We evaluate competitive programming problems from LiveCodeBench v6 with LeetCode-style feedback. As shown in Figure 1, SDPO substantially improves over GRPO, reaching a higher final accuracy (48.8% vs. 41.2%) and achieving GRPO's final accuracy in $4\times$ fewer generations. SDPO's gains grow with model scale, suggesting that the ability for self-teaching emerges as models become stronger in-context learners.

We include a comprehensive discussion of related work in Appendix G.

2 SDPO: Self-Distillation Policy Optimization

We propose an algorithm that uses the in-context learning ability of the current policy for assigning credit. Our key object is the *self-teacher*, $\pi_\theta(\cdot \mid x, f)$, which refers to the current policy (the "student") prompted with the question x and the rich feedback f. Next to the students' original attempt y, f may incorporate two key kinds of feedback: any environment output (such as runtime errors from a code environment) and a sample solution if x was already solved with another attempt in the rollout group.[1] As discussed before, the self-teacher $\pi_\theta(\cdot \mid x, f)$ should have a higher accuracy than the student $\pi_\theta(\cdot \mid x)$ since it sees additional information in-context. This leads us to observe:

We can use the same policy in two different roles: As the student for the initial attempt and as the teacher to determine the value of actions in hindsight.

We introduce **Self-Distillation Policy Optimization** (**SDPO**) which repeatedly distills the self-teacher into the student. Given a question x, we first sample rollouts from the student π_θ and obtain corresponding environment feedback. We then use the KL-divergence, $\mathrm{KL}(p\|q) = \sum_i p(i) \log {}^{p(i)}/_{q(i)}$, as a distance measure for the next-token distributions of student and teacher, and optimize a standard logit distillation loss:

$$\mathcal{L}_{\mathrm{SDPO}}(\theta) := \sum_t \mathrm{KL}(\pi_\theta(\cdot \mid x, y_{<t}) \| \mathrm{stopgrad}(\pi_\theta(\cdot \mid x, f, y_{<t}))) \tag{1}$$

where the stopgrad operator blocks gradients from flowing through the teacher, and thus prevents it from regressing towards the student and ignoring f. The intuitive role of the teacher is to determine where and how the students' original attempt y was wrong through retrospection based on the feedback f. Figure 3 shows an example of self-teaching with Qwen3-8B as student and self-teacher. We summarize SDPO in Algorithm 1 and display the teachers' reprompt template in Table 3.

We can derive the SDPO gradient as follows (see Appendix E.1 for details):

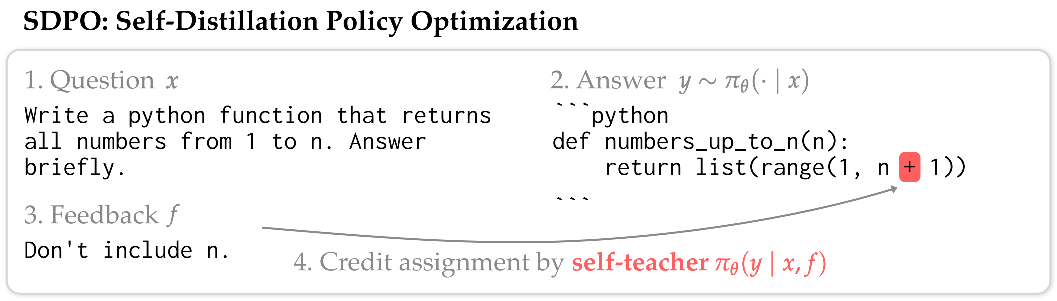
Algorithm 1 SDPO

```
Require: Language model $\pi_\theta$; dataset with questions $x$;
    number of rollouts $G$ per question; environment to
    obtain feedback for attempts.
1: repeat
2:     Sample question $x$ from dataset.
3:     Sample responses: $\{y_i\}_{i=1}^G \sim \pi_\theta(\cdot \mid x)$.
4:     Evaluate responses to obtain feedback $f_i$.  ▷
    Self-distillation:
5:     Compute log-probs of self-teacher
           $\log \pi_\theta(y_{i,t} \mid x, f_i, y_{i,<t})$.
6:     Update $\theta$ with gradient descent on $\mathcal{L}_{\mathrm{SDPO}}(\theta)$.
7: until converged
```

Proposition 2.1. *The gradient of $\mathcal{L}_{\mathrm{SDPO}}$ is*

$$\nabla \mathcal{L}_{\mathrm{SDPO}}(\theta) = \mathbb{E}_{y \sim \pi_\theta(\cdot|x)}\left[\sum_{t=1}^{|y|} \mathbb{E}_{\hat{y}_t \sim \pi_\theta(\cdot|x, y_{<t})}\left[\log \frac{\pi_\theta(\hat{y}_t \mid x, y_{<t})}{\pi_\theta(\hat{y}_t \mid x, f, y_{<t})} \cdot \nabla_\theta \log \pi_\theta(\hat{y}_t \mid x, y_{<t})\right]\right]. \tag{2}$$

[1]In standard RLVR implementations a rollout group contains multiple simultaneous attempts for x.


SDPO: Self-Distillation Policy Optimization
1. Question x
2. Answer $y \sim \pi_\theta(\cdot \mid x)$
3. Feedback f
4. Credit assignment by **self-teacher** $\pi_\theta(y \mid x, f)$

```
Write a python function that returns
all numbers from 1 to n. Answer
briefly.
```

~~~
```python
def numbers_up_to_n(n):
    return list(range(1, n + 1))
```
~~~

```
Don't include n.
```

Figure 3: Example of self-teaching with Qwen3-8B. The answer is generated by the model before seeing the feedback. Then, we re-evaluate the log-probs of the original attempt with the *self-teacher* after seeing the feedback. We show the per-token $\log(\mathbb{P}(\text{self-teacher})/\mathbb{P}(\text{student}))$, with red indicating negative values (self-teacher disagrees) and white indicating values around zero. Notably, in this example, Qwen3-8B identifies the error through retrospection without an explicit solution. Further, the activation is sparse, identifying where mistakes happen.

2.1 Comparison to RLVR

Note that the SDPO gradient is a (negated) logit-level policy gradient where the advantages are estimated using the self-teacher.[2] We can therefore reuse standard RLVR implementations and simply swap out the advantages. Let y_i be the i-th rollout from a rollout group of size G for question x, then comparing GRPO and SDPO we have:

$$A_{i,t}^{\text{GRPO}}(\hat{y}_{i,t}) := \frac{\mathbb{1}\{y_{i,t} = \hat{y}_{i,t}\}}{\pi_\theta(\hat{y}_{i,t} \mid x, y_{i,<t})}\left(r_i - \text{mean}\{r_i\}_{i=1}^G\right), \quad A_{i,t}^{\text{SDPO}}(\hat{y}_{i,t}) = \log \frac{\pi_\theta(\hat{y}_{i,t} \mid x, f_i, y_{i,<t})}{\pi_\theta(\hat{y}_{i,t} \mid x, y_{i,<t})}.$$

The GRPO advantages are zero on any non-generated token and constant within a rollout y_i.[3] In contrast, the SDPO advantages are zero only for tokens where student and teacher perfectly agree. The SDPO advantage is positive for tokens which are more likely under the teacher while being negative for tokens which are less likely under the teacher. Thus, SDPO can be seen as a direct extension of standard RLVR methods in two ways:

1. from 1-bit feedback to *allowing arbitrary sequences of tokens as feedback*, and
2. leveraging this rich feedback to *estimate dense logit-level advantages*.

This tight connection to RLVR methods also enables a straightforward extension of the SDPO gradient from Equation (2) to off-policy data via PPO-style clipped importance sampling (Schulman et al., 2017), see Appendix E.4. We further show in Appendix C.1 that SDPO only incurs a minor compute overhead compared to GRPO and discuss stability improvements in Appendix C.2.

3 Learning without Environment Feedback

We first evaluate SDPO in standard RLVR environments, where feedback is limited to scalar rewards. Instead of using the scalar reward, SDPO treats successful attempts sampled in the current batch as "feedback" for failed attempts on the same question. By comparing the student's attempt with a correct solution, the self-teacher can identify where the student was wrong and provide dense credit assignment.

Experimental setting We evaluate tasks on which the model has not been explicitly fine-tuned:

- **Science Q&A** (Chemistry, Physics, Biology, Materials science): Undergraduate-level scientific reasoning using reasoning subsets from SciKnowEval (Feng et al., 2024a).
- **Tool use**: Mapping an API specification and user request to the correct tool call (ToolAlpaca; Tang et al., 2023).

[2]See Appendix E.4 for a detailed comparison of the SDPO gradient to the standard policy gradient.
[3]We use the GRPO (Shao et al., 2024) advantage without normalization (Liu et al., 2025b).

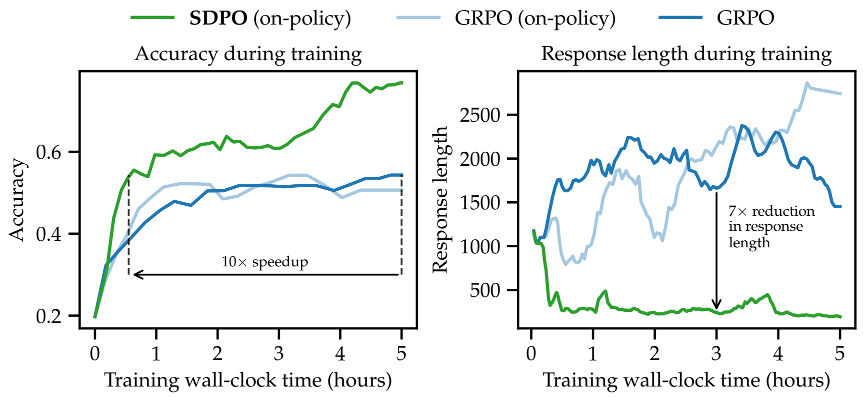


Figure 4: Training progression of Olmo3-7B-Instruct on Chemistry. We report the average accuracy across 16 samples per question and a rolling average of response lengths over 5 steps. We report GRPO with the optimal hyperparameters for this model and task.

We perform a train-test split to test in-domain generalization. We use Qwen3-8B (Yang et al., 2025) and Olmo3-7B-Instruct (Olmo et al., 2025) as initial checkpoints and report avg@16 relative to wall-clock training time, excluding initialization & validation.

Baselines. We compare SDPO to an improved variant of **GRPO** (Shao et al., 2024), which incorporates several recent modifications (cf. Appendix A). We additionally report the special case of **on-policy GRPO** (matching the hyperparameters of vanilla SDPO). For both baselines, we perform a hyperparameter sweep and report results for the models that achieve the highest validation performance across all target tasks. Hyperparameters and training details are provided in Appendix I. We use the `verl` library (Sheng et al., 2025) for fast multi-GPU training.

Results. Table 1 summarizes our results. We find that SDPO outperforms GRPO across almost all runs, often leading to substantial improvements. SDPO learns notably faster than GRPO, performing close to 5 hours of GRPO training after only 1 hour of training with SDPO in several cases. SDPO achieves a particularly substantial improvement over GRPO on the Chemistry task, as is displayed in Figure 4 (left). With Olmo3-7B-Instruct, *SDPO achieves the 5h GRPO accuracy in 30 minutes of wall-clock training time*, a 10× speedup. Moreover, SDPO's 5h accuracy is more than 20%-points higher than that of GRPO.

We remark that our results with SDPO use strictly on-policy training (i.e., one gradient step per generation batch). Given the known efficiency gains of performing multiple gradient steps per generation batch, studying SDPO with off-policy updates is an exciting direction for future research.

Self-distillation learns to reason concisely. We consistently observe that SDPO produces substantially shorter generations than GRPO while achieving higher accuracy. SDPO's responses are more than 3× shorter on average across tasks (cf. Table 8 in Appendix H). On Chemistry with Olmo3-7B-Instruct, SDPO even achieves a 7× reduction in response length relative to GRPO while maintaining higher accuracy (Figure 4 (right)). While recent progress in RLVR has demonstrated that scaling response length is a powerful driver of emergent reasoning capabilities (Jaech et al., 2024; Guo et al., 2025; Muennighoff et al., 2025), our results suggest that effective reasoning need not always be verbose. We find that SDPO improves the *efficiency* of reasoning.

Qualitatively, we observe that the longer responses from GRPO often stem from "superficial" reasoning rather than necessary analytical steps. GRPO frequently generates filler phrases like "Hmm" and "Wait" or enters circular logical loops that repeat previous steps verbatim. Figure 5 displays a representative example of this phenomenon. Remarkably, SDPO's generations remain concise and avoid these superficial patterns. This may be explained by SDPO's dense credit assignment, which assigns a specific advantage to each next-token prediction, leading to sparse advantages (cf. Figure 17 in Appendix J). By improving the efficiency of reasoning, SDPO reduces inference generation time and demonstrates that reasoning performance can be improved by refining *how* the model reasons, not just how *long* it reasons.

	Chemistry		Physics		Biology		Materials		Tool use	
	1h	5h	1h	5h	1h	5h	1h	5h	1h	5h
Qwen3-8B	35.6		59.2		27.9		58.9		57.5	
+ GRPO	54.7	60.0	63.8	72.7	34.3	51.8	**74.3**	77.1	64.9	67.7
+ GRPO (on-policy)	54.2	69.6	63.6	63.6	44.4	44.4	73.9	74.1	60.2	65.7
+ **SDPO** (on-policy)	**60.0**	**70.1**	**66.6**	**75.6**	**51.5**	**52.9**	72.1	**78.4**	**68.0**	**68.5**
Olmo3-7B-Instruct	18.8		37.7		18.1		36.7		39.3	
+ GRPO	32.7	46.8	55.3	63.3	47.8	62.0	70.9	75.0	56.4	**65.0**
+ GRPO (on-policy)	48.8	54.3	**62.7**	62.7	54.2	**63.8**	73.3	73.5	56.8	60.6
+ **SDPO** (on-policy)	**59.2**	**76.8**	59.9	**66.1**	**56.1**	58.3	**73.7**	**79.1**	**60.8**	62.1

Table 1: **Comparison of SDPO and GRPO on reasoning-related benchmarks.** We report the highest achieved avg@16 within 1 hour and 5 hours of wall-clock training time, respectively. Both SDPO and on-policy GRPO perform one gradient step per generation batch, while GRPO performs 4 off-policy mini batch steps. We select optimal hyperparameters for SDPO and baselines based on 5h accuracy. Each run is performed on a node with 4 NVIDIA GH200 GPUs. Together with initialization and validation, each run takes approximately 6 hours.

. . . Alternatively. . . Closer to D? No. . . Wait I'm going in circles. . . Wait, perhaps the correct answer is B. . . $10^{1.85} \approx 69.3$. . . Ah, this works. . . Wait I think I messed up. . . Hmm. . . $10^{1.85} \approx 69.3$. . .
Thus, the correct answer is likely B: 1.85.
<answer>
B
</answer>

(a) GRPO (5,549 tokens)

. . . At pH 7.4, all functional groups are neutral. . . maintaining a balance between hydrophobic and hydrophilic character. . . [The] overall polarity. . . keeps logD from being very high. . . or very low. . . [typically falling] in the 2.0-3.0 range, with 2.61 (C) being a reasonable estimate. . .
<answer>
C
</answer>

(b) SDPO (764 tokens)

Figure 5: Example responses from GRPO and SDPO after 50 training steps to: "What is the correct octanol/water distribution coefficient logD under the circumstance of pH 7.4 for the molecule O=C1O[C@@H](COc2ccon2)CN1c1ccc(C2=CCOCC2)c(F)c1?" The answer options are A: 1.32, B: 1.85, C: 2.61, D: 3.76. The correct answer is **C**. GRPO's answer contains 5× "Hmm.", 9× "No.", and 25× "Wait". Further, GRPO's answer repeats calculations such as "$10^{1.85} \approx 69.3$", which appears four times, and the model even explicitly generates "Wait I'm going in circles". SDPO's answer avoids any circular reasoning and is more than 7× shorter. The base model is Qwen3-8B.

4 Learning with Environment Feedback

We next evaluate SDPO on coding tasks. Coding is a canonical example of an RL environment that provides rich feedback, such as runtime errors and failed unit tests. Learning to solve these coding problems requires strong credit assignment since the student must identify its precise mistakes to avoid repeating them in the future. LiveCodeBench (LCB; Jain et al., 2025) provides a set of contest-style coding problems, ranging from simple to competition-level. We restrict our evaluation to the most recent LCBv6 subset of LCB, which contains 131 questions released between February and May 2025. We consider a setting with public and private unit tests, common for code contests and coding platforms like LeetCode, where the public tests are used for evaluation during training and the private tests are used for validation (Chen et al., 2022; Le et al., 2022; El-Kishky et al., 2025; Samadi et al., 2025).[4]

We use the Qwen3 (Yang et al., 2025) model family for our experiments, with Qwen3-8B as default unless otherwise specified. We report the average accuracy over 4 rollouts and use the same GRPO baseline as in Section 3.

[4]We select public tests as a 50% random subset of private tests.

Results. Figure 1 compares the learning curves of SDPO and GRPO on LCBv6. We find that SDPO achieves a substantially higher final accuracy (48.8%) than GRPO (41.2%) while also outperforming the strongest instruct models on the public LCBv6 leaderboard:[5] Claude Sonnet 4 (40.5%) and Claude Opus 4 (39.7%). Furthermore, SDPO reaches the final accuracy of GRPO in $4\times$ fewer generations. We include an extended comparison to other RLVR baselines that perform similarly to GRPO in Table 9 in the appendix.

4.1 Self-distillation benefits from stronger models

A central question for our work is whether SDPO is sensitive to the in-context learning ability of the base model. Intuitively, we expect that SDPO benefits from a strong in-context learner, since this enables the teacher to perform more accurate retrospection.

To answer this question, we perform a scaling study with different model sizes from the Qwen3 (Yang et al., 2025) family. As shown by extensive prior work, the ability to learn in-context increases with model size (e.g., Brown et al., 2020). As depicted in Figure 6, SDPO significantly outperforms GRPO on larger models while only slightly improving over GRPO on smaller models. In a scaling study with the weaker Qwen2.5 model family, we observe the same trend, with SDPO underperforming GRPO on the weakest model (Qwen2.5-1.5B), as seen in Figure 15 in Appendix H.

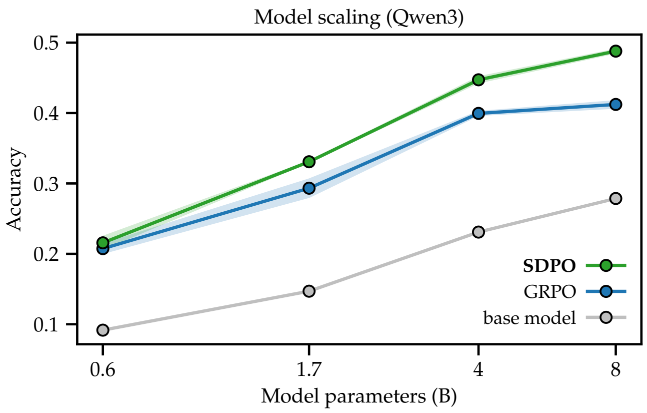


Figure 6: **SDPO improves with model size.** We compare the final LCBv6 validation accuracy of SDPO and GRPO at train step 80, across model sizes from Qwen3. The ability of SDPO's teacher to perform accurate retrospection appears to be an emergent phenomenon with scale. We include an additional scaling study with Qwen2.5-Instruct in the appendix (cf. Figure 15), supporting this finding. Error bars indicate standard error over 3 seeds.

4.2 Self-distillation performs dense credit assignment

Whereas GRPO assigns a constant advantage to each generated token, SDPO assigns an individual advantage to *each possible next token* along the generated sequence based on the agreement of student and teacher. At each position t in the generated sequence y, there are $|\mathcal{V}|$ possible next tokens where $\mathcal{V}$ is the vocabulary. In distillation, this level is typically called the *logit-level* since it corresponds to the logits of the model. In practice, we approximate the full next-token distribution by the top-K tokens, and as such, SDPO assigns $|y| \cdot K$ unique advantages per sequence.

A natural question is whether the performance gains of SDPO are due to leveraging rich feedback in RLRF or due to the dense credit assignment of SDPO. To answer this question, we ablate three configurations:

- **Logit-level SDPO:** credit assignment over the 100 most likely tokens (under the student) at each position.
- **Token-level SDPO:** credit assignment over the most likely token at each position.
- **Sequence-level SDPO:** We compute SDPO advantages for all generated tokens and average them to produce a single scalar advantage per sequence (as in GRPO). This does not perform denser credit assignment than GRPO but still leverages the rich feedback f.

As shown in Figure 7 (left), the dense credit assignment of logit-level SDPO leads to significant performance gains over token-level SDPO and sequence-level SDPO. Nevertheless, even sequence-

[5]On the public leaderboard, the LCBv6 subset can be obtained by selecting February to May 2025.

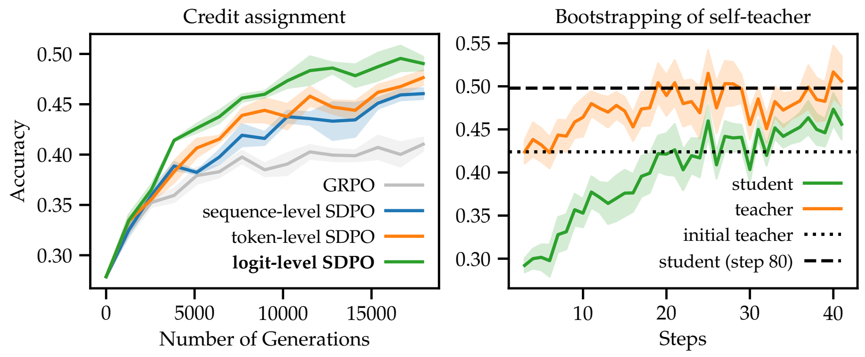


Figure 7: **Left: Rich feedback in RLRF and dense credit assignment of SDPO are complementary.** We compare logit-level, token-level, and sequence-level SDPO advantages to GRPO. While denser credit assignment in SDPO is beneficial (logit-level > token-level > sequence-level), even sequence-level SDPO significantly outperforms GRPO due to leveraging the rich feedback. Error bars indicate the standard error across 3 seeds. **Right: The self-teacher improves during training.** We display the generative accuracy of the self-teacher compared to student on the current training batch (with a rolling average over 5 steps). The final student score is taken at step 80. Notably, the performance of the student significantly surpasses the initial teacher's accuracy. Error bars indicate the standard deviation across 3 seeds.

level SDPO outperforms GRPO, indicating that leveraging rich feedback in RLRF can lead to gains over RLVR methods even without dense credit assignment.

4.3 The self-teacher improves during training

Contrary to standard distillation, the self-teacher in SDPO is not frozen, but updated throughout training. This is a critical component of SDPO, since it enables the teacher to improve over time, which means that the student can learn from a stronger target. To investigate whether the self-teacher improves during training, we plot the average accuracy when *generating* using the self-teacher in Figure 7 (right). We find that the self-teacher improves significantly during training. Most notably, the student's accuracy surpasses the initial teacher's accuracy in later stages of training. This demonstrates that SDPO enables bootstrapping of a weak model to a strong model, without the initial self-teacher's performance limiting the final student.

We further show in Appendix D that SDPO avoids catastrophic forgetting, and we analyze which feedback is most informative for the self-teacher.

5 Conclusion and Limitations

We introduced **Reinforcement Learning with Rich Feedback** (RLRF), a paradigm where environments provide tokenized feedback beyond scalar rewards, and argued that this removes a key information bottleneck of RLVR. We then proposed **Self-Distillation Policy Optimization** (SDPO), which uses the current policy as a feedback-conditioned *self-teacher* and distills its corrected log-probabilities into the student. This leverages the model's ability to learn from context for dense credit assignment. We further demonstrated that SDPO can be implemented as a minimal, drop-in modification to standard RLVR pipelines. Empirically, SDPO demonstrates superior sample efficiency and wall-clock convergence compared to GRPO on reasoning tasks, even when training in standard RLVR environments without rich feedback. SDPO's gains grow with model scale, suggesting that the capacity for self-correction scales with the model's in-context learning capabilities.

SDPO enables learning from rich feedback in a way that that is arguably closer to human cognition: utilizing precise outcomes rather than just binary rewards. By allowing the model to determine retrospectively how it should have acted, we demonstrate that language models can convert diverse tokenized feedback into effective self-supervision.

Limitations. Our findings show that SDPO's performance depends on a model's in-context learning ability, suggesting that SDPO is primarily applicable for RL-training stronger base models, while it can underperform GRPO on weaker models. Moreover, performance depends on the quality of the environment feedback. If the environment provides uninformative or misleading feedback, a model may not be able to learn from it through SDPO. Finally, SDPO adds a small computational overhead compared to GRPO for computing the log-probs of the retrospective model. While often negligible, this may be a larger overhead for smaller models with shorter generation lengths, where generation time is comparatively small.

REFERENCES

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *ICLR*, 2024.

Afra Amini, Tim Vieira, and Ryan Cotterell. Better estimation of the kullback–leibler divergence between language models. In *NeurIPS*, 2025.

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *NeurIPS*, 2017.

Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In *NeurIPS*, 2017.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint ArXiv:2005.14165*, 2020.

Bowen Cao, Deng Cai, and Wai Lam. Infiniteicl: Breaking the limit of context window size via long short-term memory transformation. In *ACL*, 2025.

Aili Chen, Aonian Li, Bangwei Gong, Binyang Jiang, Bo Fei, Bo Yang, Boji Shan, Changqing Yu, Chao Wang, Cheng Zhu, et al. Minimax-m1: Scaling test-time compute efficiently with lightning attention. *arXiv preprint arXiv:2506.13585*, 2025a.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. In *ICLR*, 2022.

Howard Chen, Noam Razin, Karthik Narasimhan, and Danqi Chen. Retaining by doing: The role of on-policy data in mitigating forgetting. *arXiv preprint arXiv:2510.18874*, 2025b.

Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. In *NeurIPS*, 2021.

Wentse Chen, Jiayu Chen, Fahim Tajwar, Hao Zhu, Xintong Duan, Ruslan Salakhutdinov, and Jeff Schneider. Retrospective in-context learning for temporal credit assignment with large language models. In *NeurIPS*, 2025c.

Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Banghua Zhu, Hao Zhang, Michael Jordan, Joseph E Gonzalez, et al. Chatbot arena: An open platform for evaluating llms by human preference. In *ICML*, 2024.

Eunbi Choi, Yongrae Jo, Joel Jang, and Minjoon Seo. Prompt injection: Parameterization of fixed inputs. *arXiv preprint arXiv:2206.11349*, 2022.

Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, Qixin Xu, Weize Chen, et al. Process reinforcement through implicit rewards. *arXiv preprint arXiv:2502.01456*, 2025.

Zi-Yi Dou, Cheng-Fu Yang, Xueqing Wu, Kai-Wei Chang, and Nanyun Peng. Re-rest: Reflection-reinforced self-training for language agents. In *EMNLP*, 2024.

Ahmed El-Kishky, Alexander Wei, Andre Saraiva, Borys Minaiev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub Pachocki, et al. Competitive programming with large reasoning models. *arXiv preprint arXiv:2502.06807*, 2025.

Sabri Eyuboglu, Ryan Ehrlich, Simran Arora, Neel Guha, Dylan Zinsley, Emily Liu, Will Tennien, Atri Rudra, James Zou, Azalia Mirhoseini, et al. Cartridges: Lightweight and general-purpose long context representations via self-study. In *ICLR*, 2026.

Kehua Feng, Keyan Ding, Weijie Wang, Xiang Zhuang, Zeyuan Wang, Ming Qin, Yu Zhao, Jianhua Yao, Qiang Zhang, and Huajun Chen. Sciknoweval: Evaluating multi-level scientific knowledge of large language models. *arXiv preprint arXiv:2406.09098*, 2024a.

Xidong Feng, Bo Liu, Yan Song, Haotian Fu, Ziyu Wan, Girish A Koushik, Zhiyuan Hu, Mengyue Yang, Ying Wen, and Jun Wang. Natural language reinforcement learning. *arXiv preprint arXiv:2411.14251*, 2024b.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning. In *ICML*, 2025.

Prasoon Goyal, Scott Niekum, and Raymond J Mooney. Using natural language for reward shaping in reinforcement learning. In *IJCAI*, 2019.

Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Guo, Mohammad Gheshlaghi Azar, et al. Bootstrap your own latent-a new approach to self-supervised learning. In *NeurIPS*, 2020.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. Minillm: Knowledge distillation of large language models. 2024.

Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, et al. Openthoughts: Data recipes for reasoning models. In *ICLR*, 2026.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Ali Hatamizadeh, Syeda Nahida Akter, Shrimai Prabhumoye, Jan Kautz, Mostofa Patwary, Mohammad Shoeybi, Bryan Catanzaro, and Yejin Choi. Rlp: Reinforcement as a pretraining objective. In *ICLR*, 2026.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *ICLR*, 2025.

Devvrit Khatri, Lovish Madaan, Rishabh Tiwari, Rachit Bansal, Sai Surya Duvvuri, Manzil Zaheer, Inderjit S Dhillon, David Brandfonbrener, and Rishabh Agarwal. The art of scaling reinforcement learning compute for llms. In *ICLR*, 2026.

Yoon Kim and Alexander M Rush. Sequence-level knowledge distillation. In *EMNLP*, 2016.

Kimi, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. Kimi k1.5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*, 2025.

Kalle Kujanpää, Pekka Marttinen, Harri Valpola, and Alexander Ilin. Efficient knowledge injection in LLMs via self-distillation. *TMLR*, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *PSIGOPS*, 2023.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. Tulu 3: Pushing frontiers in open language model post-training. In *COLM*, 2025.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *NeurIPS*, 2022.

Kyungjae Lee, Dasol Hwang, Sunghyun Park, Youngsoo Jang, and Moontae Lee. Reinforcement learning from reflective feedback (rlrf): Aligning and improving llms via fine-grained self-reflection. *arXiv preprint arXiv:2403.14238*, 2024.

Yoonho Lee, Joseph Boen, and Chelsea Finn. Feedback descent: Open-ended text optimization via pairwise comparison. *arXiv preprint arXiv:2511.07919*, 2025.

Tianle Li, Wei-Lin Chiang, Evan Frick, Lisa Dunlap, Tianhao Wu, Banghua Zhu, Joseph E Gonzalez, and Ion Stoica. From crowdsourced data to high-quality benchmarks: Arena-hard and benchbuilder pipeline. In *ICML*, 2025.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *ICLR*, 2023.

Grace Liu, Michael Tang, and Benjamin Eysenbach. A single goal is all you need: Skills and exploration emerge from contrastive rl without rewards, demonstrations, or subgoals. In *ICLR*, 2025a.

Hao Liu, Carmelo Sferrazza, and Pieter Abbeel. Chain of hindsight aligns language models with feedback. *arXiv preprint arXiv:2302.02676*, 2023.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective. In *COLM*, 2025b.

Kevin Lu and Thinking Machines Lab. On-policy distillation. *Thinking Machines Lab: Connectionism*, 2025. URL https://thinkingmachines.ai/blog/on-policy-distillation.

Renjie Luo, Zichen Liu, Xiangyan Liu, Chao Du, Min Lin, Wenhu Chen, Wei Lu, and Tianyu Pang. Language models can learn from verbal feedback without scalar rewards. *arXiv preprint arXiv:2509.22638*, 2025.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. In *NeurIPS*, 2023.

Purbesh Mitra and Sennur Ulukus. Semantic soft bootstrapping: Long context reasoning in llms without reinforcement learning. *arXiv preprint arXiv:2512.05105*, 2025.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540), 2015.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori B Hashimoto. s1: Simple test-time scaling. In *EMNLP*, 2025.

Team Olmo, Allyson Ettinger, Amanda Bertsch, Bailey Kuehl, David Graham, David Heineman, Dirk Groeneveld, Faeze Brahman, Finbarr Timbers, Hamish Ivison, et al. Olmo 3. *arXiv preprint arXiv:2512.13961*, 2025.

Xue Bin Peng, Aviral Kumar, Grace Zhang, and Sergey Levine. Advantage-weighted regression: Simple and scalable off-policy reinforcement learning. *arXiv preprint arXiv:1910.00177*, 2019.

Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. In *ICLR*, 2015.

Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*, 2011.

Mehrzad Samadi, Aleksander Ficek, Sean Narenthiran, Siddhartha Jain, Wasi Uddin Ahmad, Somshubra Majumdar, Vahid Noroozi, and Boris Ginsburg. Scaling test-time compute to achieve ioi gold medal with open-weight models. *arXiv preprint arXiv:2510.14232*, 2025.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *ICML*, 2015.

Jérémy Scheurer, Jon Ander Campos, Tomasz Korbak, Jun Shern Chan, Angelica Chen, Kyunghyun Cho, and Ethan Perez. Training language models with language feedback at scale. *arXiv preprint arXiv:2303.16755*, 2023.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *ICML*, 2015.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *ICLR*, 2016.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Amrith Setlur, Chirag Nagpal, Adam Fisch, Xinyang Geng, Jacob Eisenstein, Rishabh Agarwal, Alekh Agarwal, Jonathan Berant, and Aviral Kumar. Rewarding progress: Scaling automated process verifiers for llm reasoning. In *ICLR*, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Idan Shenfeld, Jyothish Pari, and Pulkit Agrawal. Rl's razor: Why online reinforcement learning forgets less. In *ICLR*, 2026.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *EuroSys*, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587), 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

Charlie Snell, Dan Klein, and Ruiqi Zhong. Learning by distilling context. *arXiv preprint arXiv:2209.15189*, 2022.

Moritz Stephan, Alexander Khazatsky, Eric Mitchell, Annie S Chen, Sheryl Hsu, Archit Sharma, and Chelsea Finn. Rlvf: Learning from verbal feedback without overgeneralization. In *ICML*, 2024.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*, 2023.

Belen Martin Urcelay, Andreas Krause, and Giorgia Ramponi. From words to rewards: Leveraging natural language for reinforcement learning. In *TMLR*, 2026.

Hanyang Wang, Lu Wang, Chaoyun Zhang, Tianjun Mao, Si Qin, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Text2grad: Reinforcement learning from natural language feedback. In *ICLR*, 2026.

Peiyi Wang, Lei Li, Zhihong Shao, RX Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. In *ACL*, 2024a.

Shenzhi Wang, Le Yu, Chang Gao, Chujie Zheng, Shixuan Liu, Rui Lu, Kai Dang, Xionghui Chen, Jianxin Yang, Zhenru Zhang, et al. Beyond the 80/20 rule: High-entropy minority tokens drive effective reinforcement learning for llm reasoning. In *NeurIPS*, 2025.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, et al. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. In *NeurIPS*, 2024b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3), 1992.

Qizhe Xie, Minh-Thang Luong, Eduard Hovy, and Quoc V Le. Self-training with noisy student improves imagenet classification. In *CVPR*, 2020.

Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2reward: Reward shaping with language models for reinforcement learning. In *ICLR*, 2024.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Zhaorui Yang, Tianyu Pang, Haozhe Feng, Han Wang, Wei Chen, Minfeng Zhu, and Qian Liu. Self-distillation bridges distribution gap in language model fine-tuning. In *ACL*, 2024.

Feng Yao, Liyuan Liu, Dinghuai Zhang, Chengyu Dong, Jingbo Shang, and Jianfeng Gao. Your efficient rl framework secretly brings you off-policy rl training, 2025. URL https://fengyao.notion.site/off-policy-rl.

Weiran Yao, Shelby Heinecke, Juan Carlos Niebles, Zhiwei Liu, Yihao Feng, Le Xue, Rithesh Murthy, Zeyuan Chen, Jianguo Zhang, Devansh Arpit, et al. Retroformer: Retrospective large language agents with policy gradient optimization. In *ICLR*, 2024.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. In *NeurIPS*, 2025.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative ai by backpropagating language model feedback. *Nature*, 639:609–616, 2025.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D Goodman. Star: Bootstrapping reasoning with reasoning. In *NeurIPS*, 2022.

Kai Zhang, Xiangchao Chen, Bo Liu, Tianci Xue, Zeyi Liao, Zhihan Liu, Xiyao Wang, Yuting Ning, Zhaorun Chen, Xiaohan Fu, et al. Agent learning via early experience. *arXiv preprint arXiv:2510.08558*, 2025.

Tianjun Zhang, Fangchen Liu, Justin Wong, Pieter Abbeel, and Joseph E Gonzalez. The wisdom of hindsight makes language models better instruction followers. In *ICML*, 2023.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, et al. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*, 2025.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models. *arXiv preprint arXiv:2311.07911*, 2023.

Ruiyang Zhou, Shuozhe Li, Amy Zhang, and Liu Leqi. Expo: Unlocking hard reasoning with self-explanation-guided reinforcement learning. In *NeurIPS*, 2025.

Xiangxin Zhou, Zichen Liu, Anya Sims, Haonan Wang, Tianyu Pang, Chongxuan Li, Liang Wang, Min Lin, and Chao Du. Reinforcing general reasoning without verifiers. In *ICLR*, 2026.

Yuxin Zuo, Kaiyan Zhang, Shang Qu, Li Sheng, Xuekai Zhu, Biqing Qi, Youbang Sun, Ganqu Cui, Ning Ding, and Bowen Zhou. Ttrl: Test-time reinforcement learning. In *NeurIPS*, 2025.

APPENDICES

A BASELINES

We use an improved variant of GRPO which incorporates several recent modifications (Olmo et al., 2025; Khatri et al., 2026) such as asymmetric clipping (Yu et al., 2025), avoiding biased normalization (Liu et al., 2025b), and correcting for off-policy data when using efficient inference frameworks (Yao et al., 2025). We integrate these modifications into a GRPO implementation that represents a strong baseline, as detailed in Equation (11) in Appendix E.4. GRPO enables off-policy training through PPO's clipped importance weighting (Schulman et al., 2017).

B FIGURES

This section contains figures supporting the main text.

- Table 2 summarizes how SDPO is positioned relative to RLVR and distillation baselines.
- Figure 8 shows an example for rich feedback in a code environment.
- Figure 3 illustrates the SDPO algorithm.
- Algorithm 1 shows the SDPO training loop.
- Table 3 shows the reprompt template for the self-teacher.
- Figure 9 illustrates dense credit assignment in SDPO.

Method	**Sampling**	**Signal**	**Feedback**
SFT / Distillation (Hinton et al., 2015)	× off-policy	✓ rich	× strong teacher
On-Policy Distillation (Agarwal et al., 2024)	✓ on-policy	✓ rich	× strong teacher
RLVR (such as GRPO) (Lambert et al., 2025)	✓ on-policy	× weak	✓ environment
RL via Self-Distillation (SDPO) (ours)	✓ on-policy	✓ rich	✓ environment

Table 2: Comparison of self-distillation to alternative methods for post-training LLMs.

```
Runtime Error
ZeroDivisionError: division by zero
Line 73 in separateSquares (Solution.py)

Last Executed Input
[[26,30,2],[11,23,1]]
```

Figure 8: Example of feedback from our code environment, inspired by LeetCode. Listings 5, 6, and 7 in the appendix show examples of feedback in case of a wrong answer, a memory error, and an index error.

User:	prompt
	Correct solution: successful_previous_rollout
	The following is feedback from your unsuccessful earlier attempt: environment_output
	Correctly solve the original question.
Assistant:	original_response

Table 3: Template for self-teacher. prompt is replaced with the question. A sample solution previously generated by the student is substituted for successful_previous_rollout (if available for this question; otherwise the paragraph is skipped). environment_output is replaced with the environment output (see, e.g., Figure 8) from the models' original attempt (if it was not successful and there is no solution; otherwise the paragraph is skipped). If the models' original attempt was successful, this attempt is passed as the correct solution. original_response is replaced with the models' original attempt to re-evaluate its log-probabilities under the self-teacher.

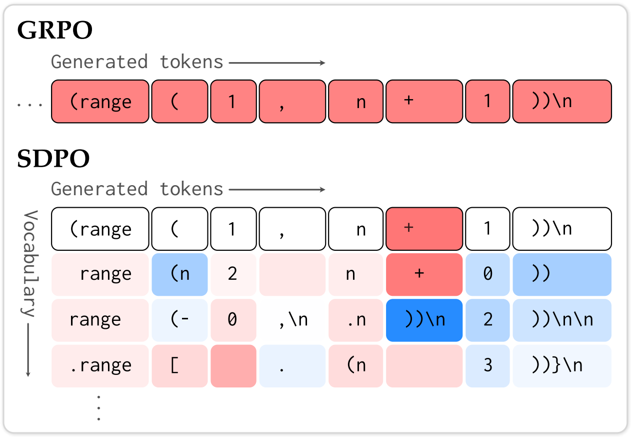


Figure 9: Dense credit assignment in SDPO in the example from Figure 3. Shown in blue are tokens which become more likely under the self-teacher. The self-teacher identifies how the returned range has to be modified so that it does not contain `n`.

C Extended Section 2

C.1 Compute time & memory

The only computational overhead of SDPO compared to GRPO is the additional computation of log-probs from the self-teacher, which can be effectively parallelized and is substantially faster than sequential generation. Figure 10 compares the compute time of SDPO and GRPO. As expected, the compute overhead of SDPO is relatively small. Here, we use a micro batch size of 2;[6] compute time can be further reduced by using larger micro batch sizes.

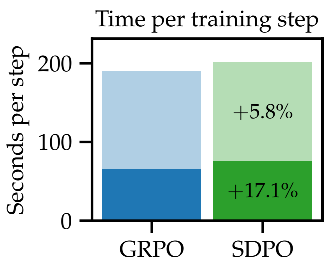


Figure 10: Time per step for SDPO vs GRPO (solid: without code environment, light: with code environment).

Naively computing the KL divergence between student and teacher requires holding full logits of both models in memory. To avoid this, we approximate the KL divergence in the SDPO loss by performing top-K distillation (i.e., only computing the top-K logits of the student and the corresponding logits of the teacher alongside a term capturing the tail probability; cf. Appendix E.3). With a reasonable choice of K (e.g., $K = 100$), this avoids virtually any memory overhead while capturing most of the information.

C.2 Stability improvements

We find that two practical modifications significantly enhance the training stability of SDPO. First, we employ a regularized self-teacher, implemented either via an exponential moving average (EMA) of the student parameters or by interpolating the current teacher with the initial teacher (cf. Appendix E.2). As detailed later, both strategies effectively stabilize learning. Second, we adopt the symmetric Jensen-Shannon divergence for the distillation loss; this formulation has similarly been shown to improve stability in on-policy distillation from external teachers (Agarwal et al., 2024).

[6]The micro batch size corresponds to # rollouts we train on at a time while accumulating gradients.

	Task:	Holdout tasks:				
	LCBv6	IFEval	ArenaHard-v2 (hard prompt)	ArenaHard-v2 (creative writing)	MMLU-Pro	**Avg.** (holdout)
Base	27.9	83.9	14.0	13.7	62.5	43.5
SFT on self-teacher	42.7	83.7	11.2	8.9	61.9	41.4
GRPO	41.2	82.2	12.0	10.8	62.3	41.8
SDPO	48.8	83.2	12.3	11.1	62.9	42.4

Table 4: **On-policy methods do not suffer from catastrophic forgetting.** We compare the accuracy of the final checkpoint on the training task LCBv6 and on holdout tasks IFEval, ArenaHard-v2, and MMLU-Pro. We compare to a baseline that trains directly on responses generated by the initial self-teacher with SFT. Overall, SDPO achieves the best performance–forgetting tradeoff. We include additional baseline results in Table 9 in the appendix.

D Extended Section 4

D.1 On-policy self-distillation avoids catastrophic forgetting

Prior work has shown that a key benefit of on-policy algorithms, such as GRPO, is that models tend not to forget previously obtained capabilities (Shenfeld et al., 2026; Chen et al., 2025b; Lu & Thinking Machines Lab, 2025). This is practically desirable since it enables continual training pipelines where a model is trained sequentially on diverse tasks without the need to retrain from scratch. To evaluate forgetting, we test the final checkpoints of GRPO and SDPO on diverse holdout tasks: IFEval (Zhou et al., 2023), which tests the ability of a model to follow precise format instructions; ArenaHard-v2 (Li et al., 2025), which is an LLM-judged benchmark of real-world instruction-following prompts derived from LMArena (Chiang et al., 2024); and MMLU-Pro (Wang et al., 2024b), which tests broad multi-task knowledge and reasoning. As displayed in Table 4, SDPO learns the new task while mitigating degradation of initial capabilities, overall achieving a better performance–forgetting tradeoff than GRPO.

Off-policy self-distillation baseline. As an additional baseline, we consider training the student via supervised fine-tuning (SFT) on successful generations from the self-teacher (Scheurer et al., 2023; Dou et al., 2024; Zhou et al., 2025).[7] This requires $2\times$ the generations of SDPO for the same number of steps, since we have to generate from both the student and the teacher. We report SFT on the successes of the self-teacher, which achieves a higher accuracy than also including initial successes from the student in the SFT data. As shown in Table 4, SFT on the self-teacher significantly underperforms SDPO on LCBv6, while leading to worse forgetting of prior capabilities. This mirrors prior findings on the instability of off-policy imitation (see, e.g., Agarwal et al., 2024).

D.2 Can GRPO and SDPO be combined?

GRPO utilizes Monte Carlo advantages, which are unbiased with respect to the objective of maximizing expected reward $J(\theta) := \mathbb{E}_{y\sim\pi_\theta(\cdot|x)}[r(y \mid x)]$. In contrast, SDPO advantages are inherently biased with respect to $J(\theta)$ due to being computed from rich feedback and a self-teacher. This dichotomy parallels the fundamental distinction between Monte Carlo and bootstrapped advantages in RL: while the latter are biased, they typically yield lower variance (Sutton & Barto, 1998; Schulman et al., 2016). This motivates a hybrid approach that combines reward-derived GRPO advantages with feedback-derived SDPO advantages:

$$A_{i,t}^{\text{SDPO+GRPO}}(\hat{y}_{i,t}) := \lambda A_{i,t}^{\text{GRPO}}(\hat{y}_{i,t}) + (1-\lambda) A_{i,t}^{\text{SDPO}}(\hat{y}_{i,t}), \quad \lambda \in [0,1]. \tag{3}$$

As shown in Figure 11, SDPO+GRPO appears to be more robust to weaker models than SDPO. Intuitively, in a weaker model such as Qwen3-0.6B, the SDPO advantages are less reliable, and hence including the GRPO advantage helps to stabilize training. In contrast, we find that SDPO+GRPO

[7]SFT on a teacher's predictions is a standard off-policy distillation approach (Kim & Rush, 2016).

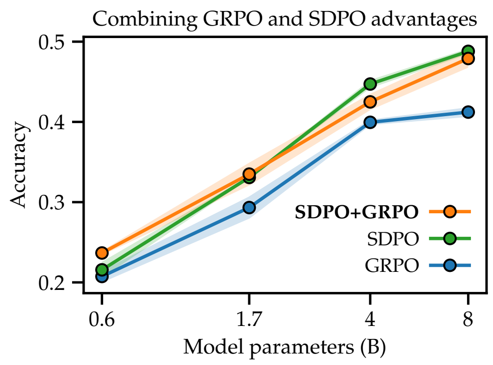


Figure 11: We compare the LCBv6 validation accuracy at step 80, across model sizes from Qwen3. SDPO+GRPO significantly outperforms SDPO on the weaker Qwen3-0.6B, while slightly underperforming SDPO on stronger models. We use $\lambda = 0.9$. Error bars indicate the standard error across 3 seeds.

slightly underperforms SDPO on stronger models such as Qwen3-8B. This suggests that the signal of GRPO, only informed by a scalar reward, can be actively harmful with a strong initial model.

D.3 Which feedback is most informative?

To understand which type of rich feedback is most informative, we ablate the three types of feedback present in a verifiable environment like code generation: the sample solution (if a successful rollout is available in the current rollout group), the environment output (such as runtime errors), and the student's original attempt.

Sample solutions. Including a sample solution from a failed attempt's rollout group (if available) closely mirrors the group-relative advantages of GRPO. We emphasize that these sample solutions are always generated by the student, as in GRPO, and do not require an expert model. They allow for disincentivizing unsuccessful approaches if the model is already able to solve the question. However, unlike GRPO where all tokens receive the same negative advantage, the self-teacher can identify specific mistakes and provide feedback on how to fix them.

Environment output. The environment output describes the state of the environment after the student's attempt. This is complementary to sample solutions since it can provide useful signal even if the student has never solved the question before. Leveraging environment output is a key differentiating factor between RLRF and RLVR settings.

Student's original attempt. The student's original attempt y does not have to be included in the reprompting template of the teacher. Indeed, we find that including it biases the teacher towards the student's attempt (cf. Table 5). This reduces the entropy of the student's distribution (particularly for initially uncertain tokens), thereby reducing exploration.

We summarize results in Table 5 where we evaluate the effect on SDPO training as well as the direct impact on the self-teacher. We find that environment output & sample solutions are complementary, each providing informative feedback. Generally, we observe that performance is not sensitive to syntactic variations of the reprompting template from Table 3.

D.4 Teacher regularization improves training stability

As described in Appendix C.2, SDPO uses a regularized teacher to stabilize training. As can be seen in Table 6, a non-regularized teacher significantly underperforms the regularized teachers. Furthermore, trust-region and EMA teachers outperform the teacher frozen at the initial teacher's parameters, showing that the teacher improves through parameter sharing with the student. Yet, SDPO performs well even with a frozen teacher.

	Teacher before training		Student trained with SDPO	
	$\uparrow$ Acc. (%)	$\downarrow$ Same output (%)	$\uparrow$ Acc. (%)	Avg. entropy
$f =$ output	32.5 ± 0.5	13.7 ± 0.6	39.8 ± 0.2	0.40 ± 0.0
$f =$ solution	$\mathbf{42.4} \pm 1.0$	12.1 ± 0.7	36.8 ± 2.7	*0.07* ± 0.0
$f =$ output + solution	$\mathbf{42.5} \pm 1.2$	$\mathbf{10.1} \pm 0.2$	$\mathbf{48.9} \pm 0.9$	0.37 ± 0.0
$f = y$ + output + solution	39.3 ± 0.8	30.0 ± 0.9	44.5 ± 1.8	*0.23* ± 0.0

Table 5: **Performance of varying kinds of feedback.** We evaluate informativeness of feedback based on SDPO training (until step 70) as well as the direct impact on the self-teacher. "Same output" measures the percentage of cases where the teacher receives the same environment output as the student's initial attempt (i.e., not exploring alternative approaches). We observe that environment output and sample solutions are complementary and each provide informative feedback. Naively including only solutions or initial attempts y significantly reduces diversity in the teacher and student. We remark that the sample solutions are generated by the student, enabling similar group-relative advantage estimation to GRPO. Error bars indicate standard deviation across 3 seeds.

Teacher	Accuracy	Avg accuracy
q_θ	36.1 ± 1.6	29.8 ± 1.3
$q_{\theta_{\mathrm{ref}}}$	48.8 ± 0.7	44.4 ± 0.2
Trust-region	$\mathbf{50.6} \pm 0.9$	$\mathbf{45.6} \pm 0.2$
EMA	49.3 ± 0.3	$\mathbf{45.3} \pm 0.2$

Table 6: Best/average accuracy until step 90 of various methods for teacher regularization. Trust-region and EMA teachers use $\alpha = 0.01$. Training of the q_θ eventually diverges. Error ranges indicate standard errors across 3 seeds.

E Implementation of SDPO

The following pseudocode in Figure 12 outlines the implementation of SDPO:

```
def compute_sdpo_loss(batch, teacher_context, loss_mask):
    """
    Computes probabilities of response y under the self-teacher
        and the per-logit SDPO loss.
    """
    # Compute model probabilities for response y
    logprobs_student = compute_log_prob(batch) # (T,V)
    probs_student = logprobs_student.exp() # (T,V)

    # Compute self-teacher probabilities for response y
    teacher_batch = reprompt(batch, teacher_context)
    logprobs_teacher = compute_log_prob(teacher_batch).detach() # (T,V)

    # Compute SDPO loss: per-token divergence
    per_token_loss = divergence(logprobs_student, logprobs_teacher) # (T,)
    return agg_loss(per_token_loss, loss_mask, loss_agg_mode="token-mean")
```

Figure 12: The pseudo-code of SDPO within a standard RL training pipeline. Omitted here is the filtering to top-K logprobs for student and teacher (including a tail term) as described in Appendix E.3. Further, we omit here any importance sampling weights to correct for off-policy data. `reprompt` modifies the batch to incorporate teacher context (i.e., rich feedback). `divergence` implements any per-token divergence such as reverse-KL, forward-KL, or Jensen-Shannon.

In the following, we provide further details on:

- The gradient estimator used in our implementation (Appendix E.1)
- Teacher regularization (Appendix E.2)
- Approximating logit-distillation with the top-K logits for saving GPU memory (Appendix E.3)
- Generalizing PPO-style policy gradient algorithms to logit-level advantages (Appendix E.4)

To disambiguate the notation of the self-teacher, we use $q_\theta(\cdot \mid x, f) := \pi_\theta(\cdot \mid \mathrm{reprompt}(x, f))$ in the following. Here, `reprompt` denotes the reprompt template of the self-teacher.

E.1 Gradient Estimators

In this seciton, we discuss two possible gradient estimators for the KL divergence between the current policy $\pi_\theta(y \mid x)$ and the teacher policy $q_\theta(y \mid x, f)$.

Per-token estimator. Deriving the gradient of the SDPO loss as defined in Equation (1):

$$\mathcal{L}_{\text{token}}(\theta) := \mathbb{E}_{y\sim \mathrm{stopgrad}(\pi_\theta(\cdot|x))}\left[\sum_{t=1}^{T} \mathrm{KL}(\pi_\theta(\cdot \mid x, y_{<t}) \| \mathrm{stopgrad}(\pi_\theta(\cdot \mid x, f, y_{<t})))\right] \tag{4}$$

leads to the following estimator (see a detailed proof in Appendix F.1), which corresponds to the sum of gradients of the KL divergence at each token:

$$\nabla \mathcal{L}_{\text{token}}(\theta) = \mathbb{E}_{y\sim\pi_\theta(\cdot|x)}\left[\sum_{t=1}^{T} \mathbb{E}_{\hat{y}_t\sim\pi_\theta(\cdot|x,y_{<t})}\left[\nabla_\theta \log \pi_\theta(\hat{y}_t \mid x, y_{<t}) \cdot \log \frac{\pi_\theta(\hat{y}_t \mid x, y_{<t})}{\pi_\theta(\hat{y}_t \mid x, f, y_{<t})}\right]\right]. \tag{5}$$

This corresponds to the estimator presented in Proposition 2.1. This gradient estimator effectively assumes that the sampling distribution generating y is fixed.

Sequence-level estimator. An alternative self-distillation objective minimizes the sequence-level KL divergence between student and self-teacher, i.e.,

$$\begin{aligned}\mathcal{L}_{\text{seq}}(\theta) := \mathrm{KL}(\pi_\theta \| q_\theta) &= \mathbb{E}_{y \sim \pi_\theta(\cdot|x)}\left[\log \frac{\pi_\theta(y \mid x)}{q_\theta(y \mid x, f)}\right] \\ &= \sum_{t=1}^{T} \mathbb{E}_{s_t \sim \Pi_\theta}\left[\mathrm{KL}(\pi_\theta(\cdot \mid s_t) \| q_\theta(\cdot \mid s_t, f))\right],\end{aligned} \tag{6}$$

where $s_t = (x, y_{<t})$ is the prefix ("state") at step t and Π_θ denotes the prefix distribution under policy π_θ. Estimating the gradient of this objective additionally takes into account how the choice of y_t influences future states $y_{>t}$ (due to the additional dependence on Π_θ).

Amini et al. (2025) show that the corresponding gradient estimator is given by

$$\nabla \mathcal{L}_{\text{seq}}(\theta) = \nabla \mathcal{L}_{\text{token}}(\theta) + \mathbb{E}_{y \sim \pi_\theta(\cdot|x)}\left[\sum_{t=1}^{T} \mathrm{KL}(\pi_\theta(\cdot \mid s_t) \| q_\theta(\cdot \mid s_t, f)) \nabla_\theta \log \Pi_\theta(s_t)\right]. \tag{7}$$

The additional term of the sequence-level gradient captures how prefixes influence the self-distillation divergence of future tokens. We also experimented with this sequence-level gradient estimator but did not find measurable gains relative to its additional complexity.

E.2 Regularized Teacher

In contrast to standard distillation, the teacher in SDPO changes throughout training. This bootstrapping enables the teacher to improve, but it may also lead to training instability. To stabilize training, we seek to prevent the teacher q from quickly diverging from the initial teacher $q_{\theta_{\text{ref}}}$. We can achieve this by placing an explicit trust-region constraint on q (Schulman et al., 2015; Peng et al., 2019), that is:

$$\sum_t \mathrm{KL}(q(y_t \mid x, f, y_{<t}) \| q_{\theta_{\text{ref}}}(y_t \mid x, f, y_{<t})) \le \epsilon, \quad \epsilon > 0. \tag{8}$$

This trust-region can be implemented in two ways:

1. **Explicit trust-region:** We can define the teacher as the policy closest to q_θ while satisfying the trust-region constraint. This teacher can be expressed as

$$q(y_t \mid x, f, y_{<t}) \propto \exp\big((1-\alpha)\log q_{\theta_{\text{ref}}}(y_t \mid x, f, y_{<t}) + \alpha \log q_\theta(y_t \mid x, f, y_{<t})\big), \tag{9}$$

   with $\alpha \in (0, 1)$ the inverse Lagrange multiplier for the trust-region constraint. We include a full derivation in Appendix F.2. We can plug this explicitly constrained teacher directly into the SDPO objective.

2. **Exponential moving average (EMA):** Alternatively, we can stabilize the teacher's parameters directly; parameterizing $q_{\theta'}$ by θ' and updating as $\theta' \leftarrow (1-\alpha)\theta' + \alpha\theta$ with $\alpha \in (0, 1)$.

Note that each implementation has a different practical advantage: The EMA teacher requires additional GPU memory for θ' yet does not introduce any runtime overhead. In contrast, the trust-region teacher requires an additional log-prob computation with $q_{\theta_{\text{ref}}}$ yet does not require additional GPU memory if θ_{ref} is used for explicit KL regularization.

E.3 Approximate Logit Distillation

To save GPU memory, we perform distillation only on the top-K tokens predicted by the student:

$$\begin{aligned}\mathcal{L}_{\mathrm{SDPO}}(\theta) &= \sum_{t=1}^{T} \mathrm{KL}(\pi_\theta(\cdot \mid x, y_{<t}) \| \mathrm{stopgrad}(q_\theta(\cdot \mid x, f, y_{<t}))) \\ &\approx \sum_{t=1}^{T} \sum_{\hat{y}_t \in \mathrm{top}_K(\pi_\theta)} \pi_\theta(\hat{y}_t \mid x, y_{<t}) \cdot \log \frac{\pi_\theta(\hat{y}_t \mid x, y_{<t})}{\mathrm{stopgrad}(q_\theta(\hat{y}_t \mid x, f, y_{<t}))} \\ &\quad + \underbrace{\left(1 - \textstyle\sum_{\hat{y}_t \in \mathrm{top}_K(\pi_\theta)} \pi_\theta(\hat{y}_t \mid x, y_{<t})\right) \cdot \log \frac{1 - \sum_{\hat{y}_t \in \mathrm{top}_K(\pi_\theta)} \pi_\theta(\hat{y}_t \mid x, y_{<t})}{\mathrm{stopgrad}\left(1 - \sum_{\hat{y}_t \in \mathrm{top}_K(\pi_\theta)} q_\theta(\hat{y}_t \mid x, f, y_{<t})\right)}}_{\text{tail}}\end{aligned} \tag{10}$$

Here, the top-K is with respect to student. Without top-K distillation, we would have to keep two copies of logits in memory: one for teacher and student each. Top-K distillation avoids virtually any memory overhead without impacting performance significantly, since most tokens of the vocabulary are not informative at a given time.

E.4 Off-Policy Training: Generalization to Logit-Level Losses

PPO-style clipping (Schulman et al., 2017) with truncated importance sampling (Yao et al., 2025), clip-higher (Yu et al., 2025), fixed length normalization (Liu et al., 2025b):

$$\mathcal{L}_{\mathrm{token}}(\theta) := -\frac{1}{\sum_{i=1}^{G} |y_i|} \sum_{i=1}^{G} \sum_{t=1}^{|y_i|} \min\left(w_{i,t}^{\mathrm{TIS}}, \rho\right) \min\left(w_{i,t} A_{i,t}, \mathrm{clip}(w_{i,t}, 1-\varepsilon_{\mathrm{low}}, 1+\varepsilon_{\mathrm{high}}) A_{i,t}\right), \tag{11}$$

with $w_{i,t} := \frac{\pi_\theta(y_{i,t} \mid x, y_{i,<t})}{\pi_{\theta_{\mathrm{old}}}(y_{i,t} \mid x, y_{i,<t})}$, $w_{i,t}^{\mathrm{TIS}} := \frac{\pi_{\theta_{\mathrm{old}}}(y_{i,t} \mid x, y_{i,<t})}{\pi_{\theta_{\mathrm{old}}}^{\mathrm{rollout}}(y_{i,t} \mid x, y_{i,<t})}$, and $A_{i,t}$ denotes the per-token advantage.

We extend this to a logit-level loss:

$$\begin{aligned}\mathcal{L}_{\mathrm{logit}}(\theta) := -\frac{1}{\sum_{i=1}^{G} |y_i|} \sum_{i=1}^{G} \sum_{t=1}^{|y_i|} \sum_{\hat{y}_{i,t}} \min\left(\pi_{\theta_{\mathrm{old}}}(\hat{y}_{i,t} \mid x, y_{i,<t}), \rho \pi_{\theta_{\mathrm{old}}}^{\mathrm{rollout}}(\hat{y}_{i,t} \mid x, y_{i,<t})\right) \\ \min\left(w_{i,t}(\hat{y}_{i,t}) A_{i,t}(\hat{y}_{i,t}), \mathrm{clip}(w_{i,t}(\hat{y}_{i,t}), 1-\varepsilon_{\mathrm{low}}, 1+\varepsilon_{\mathrm{high}}) A_{i,t}(\hat{y}_{i,t})\right),\end{aligned} \tag{12}$$

where $\hat{y}_{i,t}$ sums over all possible tokens at position t for rollout i (or the K most likely under $\pi_{\theta_{\mathrm{old}}}$, cf. Appendix E.3). The TIS changes since we explicitly weight each logit by its probability under $\pi_{\theta_{\mathrm{old}}}$ rather than relying on a Monte Carlo estimate of the expectation over next-token predictions. Here, $A_{i,t}(\hat{y}_{i,t})$ is a per-logit advantage.

In our experiments for SDPO, we apply the TIS term on a token-level rather than logit-level.

F THEORETICAL ANALYSIS

This section is organized as follows:

- Appendix F.1 derives the SDPO gradient from Theorem 2.1.
- Appendix F.2 derives the trust-region regularized teacher discussed in Appendix E.2.

To disambiguate the notation of the self-teacher, we use $q_\theta(\cdot \mid x, f) := \pi_\theta(\cdot \mid \mathrm{reprompt}(x, f))$ in the following. Here, `reprompt` denotes the reprompt template of the self-teacher.

F.1 PROOF OF PROPOSITION 2.1.

Proof. In the following, we derive the gradient of $\mathcal{L}_{\text{SDPO}}$.

$$\begin{aligned}
\nabla_\theta \mathcal{L}_{\text{SDPO}}(\theta) &= \nabla_\theta \sum_{t=1}^{T} \mathrm{KL}(\pi_\theta(\cdot \mid x, y_{<t}) \| \mathrm{stopgrad}(q_\theta(\cdot \mid x, f, y_{<t}))) \\
&= \nabla_\theta \sum_{t=1}^{T} \sum_{\hat{y}_t} \pi_\theta(\hat{y}_t \mid x, y_{<t}) \log \left(\frac{\pi_\theta(\hat{y}_t \mid x, y_{<t})}{\mathrm{stopgrad}(q_\theta(\hat{y}_t \mid x, f, y_{<t}))} \right)
\end{aligned}$$

Let $A_{t,k} := \log \left(\frac{\mathrm{stopgrad}(q_\theta(\hat{y}_t \mid x, f, y_{<t}))}{\pi_\theta(\hat{y}_t \mid x, y_{<t})} \right)$. Then,

$$\begin{aligned}
&= -\nabla_\theta \sum_{t=1}^{T} \sum_{\hat{y}_t} \pi_\theta(\hat{y}_t \mid x, y_{<t}) A_{t,k} \\
&= -\sum_{t=1}^{T} \sum_{\hat{y}_t} \pi_\theta(\hat{y}_t \mid x, y_{<t}) \nabla_\theta A_{t,k} + A_{t,k} \nabla_\theta \pi_\theta(\hat{y}_t \mid x, y_{<t}).
\end{aligned}$$

We have that $\nabla_\theta A_{t,k} = -\nabla_\theta \log \pi_\theta(\hat{y}_t \mid x, y_{<t})$ is the negative score function. Using the score trick, $\pi_\theta(\hat{y}_t \mid x, y_{<t}) \nabla_\theta \log \pi_\theta(\hat{y}_t \mid x, y_{<t}) = \nabla_\theta \pi_\theta(\hat{y}_t \mid x, y_{<t})$. Hence, the first term simplifies to

$$-\sum_{t=1}^{T} \sum_{\hat{y}_t} \pi_\theta(\hat{y}_t \mid x, y_{<t}) \nabla_\theta A_{t,k} = \sum_{t=1}^{T} \sum_{\hat{y}_t} \nabla_\theta \pi_\theta(\hat{y}_t \mid x, y_{<t}) = \sum_{t=1}^{T} \nabla_\theta \underbrace{\sum_{\hat{y}_t} \pi_\theta(\hat{y}_t \mid x, y_{<t})}_{=1} = 0.$$

Thus, the gradient of $\mathcal{L}_{\text{SDPO}}$ is

$$\begin{aligned}
\nabla_\theta \mathcal{L}_{\text{SDPO}} &= -\sum_{t=1}^{T} \sum_{\hat{y}_t} A_{t,k} \nabla_\theta \pi_\theta(\hat{y}_t \mid x, y_{<t}) \\
&= -\sum_{t=1}^{T} \sum_{\hat{y}_t} \pi_\theta(\hat{y}_t \mid x, y_{<t}) \Big(A_{t,k} \nabla_\theta \log \pi_\theta(\hat{y}_t \mid x, y_{<t}) \Big) \\
&= -\sum_{t=1}^{T} \mathbb{E}_{\hat{y}_t \sim \pi_\theta(\cdot \mid x, y_{<t})} \left[A_{t,k} \nabla_\theta \log \pi_\theta(\hat{y}_t \mid x, y_{<t}) \right].
\end{aligned}$$

□

Notably, the above implies that the gradient of $\mathcal{L}_{\text{SDPO}}$ is equivalent to the gradient of the loss if $A_{t,k} = \mathrm{stopgrad}\left(\log \frac{q_\theta(y_t \mid x, f, y_{<t})}{\pi_\theta(y_t \mid x, y_{<t})} \right)$.

F.2 TRUST-REGION TEACHER

To stabilize training, we seek to prevent the teacher q from diverging from the initial teacher $q_{\theta_{\text{ref}}}$. We can achieve this by placing an explicit trust-region constraint on the teacher q (Schulman et al., 2015; Peng et al., 2019), that is:

$$\sum_t \mathrm{KL}(q(y_t \mid x, f, y_{<t}) \| q_{\theta_{\text{ref}}}(y_t \mid x, f, y_{<t})) \leq \epsilon, \quad \epsilon > 0. \tag{13}$$

In the following, we derive a teacher q which satisfies the trust-region constraint while staying close to the target q_θ. The following optimization problem characterizes such a q (Peng et al., 2019):

$$\begin{aligned} \arg\max_{q\in\Delta} \; & \sum_t \sum_{y_t} q(y_t \mid x, f, y_{<t}) \log \frac{q_\theta(y_t \mid x, f, y_{<t})}{q_{\theta_{\text{ref}}}(y_t \mid x, f, y_{<t})} \\ \text{s.t.} \; & \sum_t \mathrm{KL}(q(y_t \mid x, f, y_{<t}) \| q_{\theta_{\text{ref}}}(y_t \mid x, f, y_{<t})) \leq \epsilon, \end{aligned} \tag{14}$$

where Δ denotes the probability simplex. Intuitively, the solution is the q satisfying the trust-region constraint, which is closest to q_θ (i.e., has minimal cross-entropy to q_θ) while being farthest from $q_{\theta_{\text{ref}}}$ (i.e., has maximal cross-entropy to $q_{\theta_{\text{ref}}}$).

Proposition F.1. *The solution to Equation* (14) *can be expressed in closed form as*

$$q^*(y_t \mid x, f, y_{<t}) \propto \exp\big((1-\alpha)\log q_{\theta_{\text{ref}}}(y_t \mid x, f, y_{<t}) + \alpha \log q_\theta(y_t \mid x, f, y_{<t})\big). \tag{15}$$

Proof. To simplify notation, we omit the conditioning in the following. The Lagrangian (with $\lambda \geq 0$ for the KL constraint and ν for normalization) is

$$\mathcal{L}(q, \lambda, \nu) = \sum_t \sum_{y_t} q(y_t) \log \frac{q_\theta(y_t)}{q_{\theta_{\text{ref}}}(y_t)} - \lambda\Big(\sum_{y_t} q(y_t) \log \frac{q(y_t)}{q_{\theta_{\text{ref}}}(y_t)} - \epsilon\Big) + \nu\Big(\sum_{y_t} q(y_t) - 1\Big).$$

Stationarity gives, for all y_t,

$$0 = \frac{\partial \mathcal{L}}{\partial q(y_t)} = \log \frac{q_\theta(y_t)}{q_{\theta_{\text{ref}}}(y_t)} - \lambda\Big(\log \frac{q(y_t)}{q_{\theta_{\text{ref}}}(y_t)} + 1\Big) + \nu.$$

Let $\alpha := 1/\lambda$. Then, the solution to Equation (14) can be characterized in closed form as

$$\begin{aligned} q^*(y_t) &\propto q_{\theta_{\text{ref}}}(y_t) \exp\Big(\alpha \log \tfrac{q_\theta(y_t)}{q_{\theta_{\text{ref}}}(y_t)}\Big) \\ &\propto \exp\big((1-\alpha)\log q_{\theta_{\text{ref}}}(y_t) + \alpha \log q_\theta(y_t)\big). \end{aligned}$$

□

Chen et al. (2025c) perform a similar derivation, but use reference $\pi_{\theta_{\text{ref}}}$, which we observe to underperform compared to the reference $q_{\theta_{\text{ref}}}$.

G Related Work

G.1 Reinforcement Learning with LLMs

Recently, large-scale RL training on diverse tasks has significantly improved the performance of LLMs on general reasoning tasks (Guo et al., 2025; Kimi et al., 2025; Olmo et al., 2025; Jaech et al., 2024; Lambert et al., 2025). This progress is primarily enabled by RLVR methods that use Monte Carlo estimates of rewards, such as STaR or GRPO (Zelikman et al., 2022; Shao et al., 2024), similar to the classical REINFORCE algorithm (Williams, 1992). While several traditional RLVR algorithms rely on learning separate value networks (Schulman et al., 2017), they incur substantial memory costs and retain the information bottleneck of scalar rewards.

In the RLVR setting, it is common for an (outcome) reward to be given only at the end of a sequence. To improve credit assignment, several works learn so-called process reward models (PRMs) that estimate rewards for each step in the sequence (Lightman et al., 2023; Wang et al., 2024a; Setlur et al., 2025). Unlike our RLRF setting, PRMs are typically trained on scalar rewards, either on value estimates for intermediate states or on outcome rewards (Cui et al., 2025). Unlike the self-teacher in SDPO, PRMs are a distinct model from the student, introducing significant memory overhead. Our work shows that *each language model is implicitly a PRM* through retrospection if given rich feedback.

Conceptually, our work is related to “bootstrapping your own latent” (BYOL; Grill et al., 2020) and “expert iteration” (Anthony et al., 2017) where a student is bootstrapped by repeatedly imitating an improved version of itself (called the “expert”). Canonically, the expert combines the student with test-time search, such as tree search (Anthony et al., 2017) or majority voting (Zuo et al., 2025). In contrast, SDPO leverages the student’s ability to learn from rich feedback provided in-context, which is related to “augmented views” in BYOL.

G.2 Learning from Rich Feedback and through Retrospection

Beyond scalar outcome rewards, recent works have leveraged rich execution or verbal feedback to guide generation (Gehring et al., 2025; Feng et al., 2024b; Yuksekgonul et al., 2025). A primary line of research focuses on translating verbal feedback into reward functions for RL. This is often achieved by mapping feedback to discrete token-level rewards using an external frozen model (Wang et al., 2026), or by employing strong external LLMs to explicitly construct state-wise reward functions (Goyal et al., 2019; Xie et al., 2024; Urcelay et al., 2026).

Alternatively, feedback can be utilized without explicit reward modeling. Several approaches focus on in-context improvement without integrating the process into the RL optimization loop (Chen et al., 2021; Madaan et al., 2023; Shinn et al., 2023; Yao et al., 2024; Yuksekgonul et al., 2025; Lee et al., 2025). Others manually curate preference datasets by pairing responses before and after feedback to train with direct preference optimization (Stephan et al., 2024; Lee et al., 2024), though this requires additional generation and lacks the direct credit assignment of SDPO. Various recent works bootstrap thinking traces from known answers, using these answers as rich feedback (Zhou et al., 2026; Hatamizadeh et al., 2026; Zhang et al., 2025).

A central object in several recent works is a feedback-conditioned policy $\pi_\theta(y \mid x, f)$, which learns answers y that lead to feedback f (Liu et al., 2023; Zhang et al., 2023; Luo et al., 2025), typically through supervised objectives. The idea behind these approaches is to deploy a policy conditioned on desirable (i.e., positive) feedback for deployment. This approach is conceptually related to goal-conditioned RL (Schaul et al., 2015; Liu et al., 2025a), where one can learn from negative examples through goal relabeling (Andrychowicz et al., 2017). Feedback-conditioned policies view feedback as a goal, whereas RLRF views feedback as a state that can be used to determine whether the goal x is achieved. Unlike SDPO, these methods do not use feedback for credit assignment in negative trajectories, but rather as a data transformation for goal relabeling.

G.3 Distillation

Distillation is frequently employed as an alternative to supervised fine-tuning (SFT) when a strong teacher model is available. This approach transfers capabilities by training a student to mimic the output distribution or intermediate representations of the teacher (Hinton et al., 2015; Romero et al., 2015; Kim & Rush, 2016; Sanh et al., 2019; Xie et al., 2020). Distillation is typically performed on fixed off-policy datasets. To address the distribution shift between training and inference, recent works explore on-policy distillation, where the student learns from feedback of an external teacher on its own generations (Agarwal et al., 2024; Gu et al., 2024; Yang et al., 2025; Lu & Thinking Machines Lab, 2025). This mitigates the train-test mismatch, which relates closely to earlier work on online imitation learning (Ross et al., 2011).

G.4 Self-Distillation

The concept of self-distillation was first proposed by Snell et al. (2022) in a setting akin to supervised learning, introducing the idea of sampling from a model provided with extra context and training the same model to mimic these predictions without that context. This mechanism has proven effective for compressing behavior (Bai et al., 2022; Choi et al., 2022; Yang et al., 2024) and factual information (Eyuboglu et al., 2026; Kujanpää et al., 2025; Cao et al., 2025) into model weights. Beyond compressing a fixed context into model weights, recent works have used self-distillation to learn from environment feedback (Scheurer et al., 2023; Dou et al., 2024; Zhou et al., 2025; Mitra & Ulukus, 2025). These approaches use an *off-policy* self-distillation objective, which substantially underperforms SDPO's on-policy learning. Off-policy self-distillation trains the student on generations from the teacher, whereas SDPO trains the student to avoid mistakes in its own generations. In concurrent work, Chen et al. (2025c) apply on-policy self-distillation to grid world settings where feedback is a scalar reward, and a reflection stage in the self-teacher diagnoses possible mistakes, showing improved credit assignment compared to learning value networks for advantage estimation.

H Additional Results & Ablations

This section is organized as follows:

- Appendix H.1 contains results and ablations for Section 3.
- Appendix H.2 contains results and ablations for Section 4.

H.1 Learning without rich environment feedback

- Table 7 reports results when optimal hyperparameters are selected for each model/task combination.
- Table 8 compares average response lengths of SDPO and GRPO.

	Chemistry		Physics		Biology		Materials		Tool use	
	1h	5h	1h	5h	1h	5h	1h	5h	1h	5h
Qwen3-8B	35.6		59.2		27.9		58.9		57.5	
+ GRPO	54.2	69.6	62.9	74.5	34.3	51.8	**74.3**	77.1	**61.7**	68.1
+ GRPO (on-policy)	54.2	69.6	62.9	74.8	30.3	49.4	73.3	75.8	**61.7**	68.1
+ **SDPO** (on-policy)	**59.9**	**70.1**	**70.6**	**80.6**	**53.1**	**53.1**	72.1	**78.3**	56.4	**68.5**
Olmo3-7B-Instruct	18.8		37.7		18.1		36.7		39.3	
+ GRPO	42.7	54.3	55.3	63.3	54.2	**63.8**	73.8	78.1	56.4	**65.0**
+ GRPO (on-policy)	48.8	54.3	**62.7**	62.7	54.2	**63.8**	67.9	74.4	56.0	61.3
+ **SDPO** (on-policy)	**59.2**	**76.8**	60.3	**71.4**	**56.1**	58.3	**75.3**	**79.2**	**57.3**	62.5

Table 7: **Comparison of SDPO and GRPO on reasoning-related benchmarks.** We report the highest achieved avg@16 within 1 hour and 5 hours of wall-clock training time, respectively. Both SDPO and on-policy GRPO perform one gradient step per generation batch, while GRPO performs 4 off-policy mini batch steps. We select optimal hyperparameters for SDPO and baselines based on 5h accuracy. We perform this selection independently for each model and dataset. Each run is performed on a node with 4 NVIDIA GH200 GPUs. Together with initialization and validation, each run takes approximately 6 hours. *As opposed to Table 1 which selects globally optimal hyperparameters per method, this table selects optimal hyperparameters individually for each model/task combination based on 5h accuracy.* The hyperparameter grid is described in Section I.2.1.

Model	GRPO	SDPO	Reduction of SDPO
Qwen3-8B	820.8	255.8	3.2×
Olmo3-7B-Instruct	1095.4	343.9	3.2×

Table 8: Average response lengths of SDPO and GRPO (averaged across tasks from Section 3). Both algorithms are evaluated in the on-policy setting.

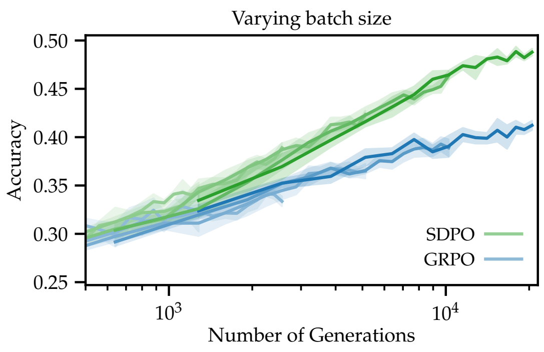
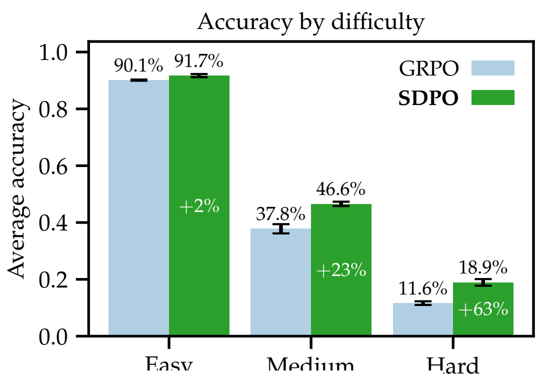


Figure 14: Accuracy (pass@1) for varying train batch sizes (4, 8, 16, 32) and number of rollouts (4, 8) for training SDPO and GRPO with Qwen3-8B (Yang et al., 2025) on LCBv6, ± stderr across 3 seeds. Different shades of the same color correspond to different runs.

H.2 Learning with rich environment feedback

H.2.1 Additional Results

Figure 13 shows the average accuracy of SDPO and GRPO stratified by question difficulty. LCB differentiates between easy, medium, and hard questions.



In Figure 14, we compare different train batch sizes and number of rollouts for training GRPO and SDPO on LCBv6.

Complementing the results shown in Figure 6, we show additional results using Qwen2.5-Instruct (Qwen et al., 2024) in Figure 15.

ing until
ars show

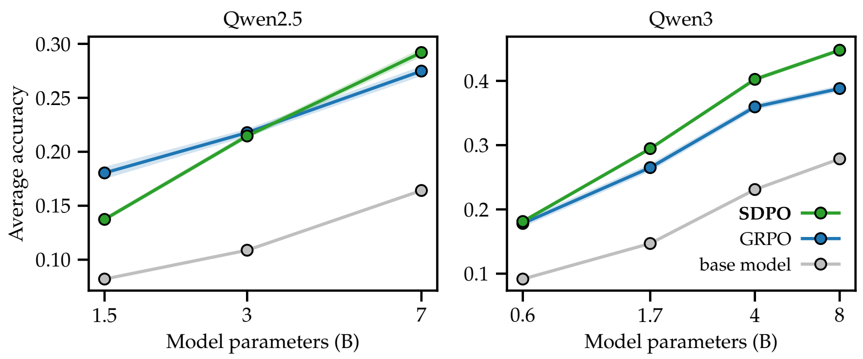


Figure 15: Average validation accuracy by model size, ± std across 3 seeds. With Qwen2.5-Instruct (Qwen et al., 2024) and Qwen3 (Yang et al., 2025) on LCBv6. Until step 65 for Qwen2.5 and until step 80 for Qwen3.

H.2.2 Training Stability

Figure 16 shows diverse metrics logged during training, including the loss, entropy, average gradient norm, and average response length.

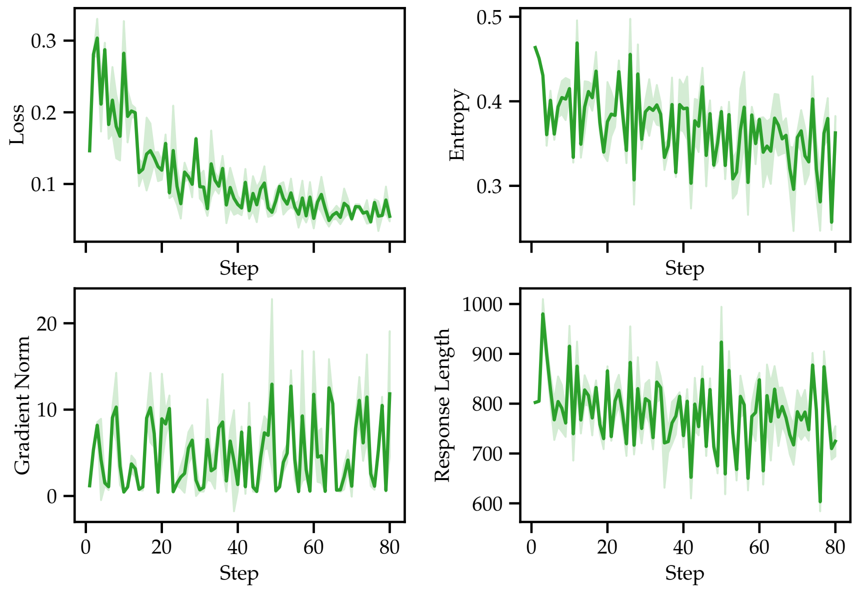


Figure 16: Loss, entropy, avg. gradient norm and avg. response length during training of SDPO on LCBv6 (Section 4

.

H.2.3 Baselines

Table 9 compares the performance on LCBv6 of various baselines, including two variants of GRPO, GSPO, and CISPO to SDPO.

	Accuracy	Avg accuracy
GRPO	41.2 ± 0.8	38.2 ± 0.0
+ only high-entropy tokens (Wang et al., 2025)	37.8 ± 2.2	35.9 ± 0.1
GSPO (Zheng et al., 2025)	40.1 ± 2.3	37.7 ± 0.1
CISPO (Chen et al., 2025a)	41.2 ± 1.8	37.8 ± 0.1
SDPO	$\mathbf{48.8} \pm 0.6$	$\mathbf{43.8} \pm 0.0$

Table 9: Performance on LCBv6 at/until training step 80 with std over 3 seeds. We compare to GSPO (Zheng et al., 2025) and CISPO (Chen et al., 2025a). With Qwen3-8B.

I Experiment Details

I.1 Technical setup

All experiments were conducted on a single node equipped with four NVIDIA GH200 GPUs, for a total of 378GB VRAM. Our environment is built on top of the NVIDIA PyTorch container `nvcr.io/nvidia/pytorch:25.02-py3`, with CUDA 12.8 and PyTorch v2.7.0.

Our implementation is based on the `verl` library (Sheng et al., 2025). We use PyTorch Fully Sharded Data Parallel (FSDP2) for distributed training. For rollout generation, we employ `vLLM` (Kwon et al., 2023), which enables efficient batched inference on the multi-GPU node.

I.2 Hyperparameters

We summarize hyperparameters used for SDPO in Table 10 and those used for GRPO in Table 11.

Parameters	**Without Feedback** Section 3	**With Feedback** Section 4
General		
Model	Qwen/Qwen3-8B allenai/Olmo3-7B-Instruct	Qwen/Qwen3-8B
Thinking	False	False
Data		
Max. prompt length	2048	2048
Max. response length	8192	8192
Batching		
Question batch size	32	32
Mini batch size	32	1
Number of rollouts	8	8
Rollout		
Inference engine	vllm	vllm
Temperature	1.0	1.0
Validation		
Number of rollouts	16	4
Temperature	0.6	0.6
Top-p	0.95	0.95
SDPO loss		
Top-K distillation	100	20
Distillation divergence	Jensen–Shannon	Reverse-KL
Clip advantages	–	–
Teacher-EMA update rate	0.05	0.01
Rollout importance sampling clip	2	2
Training		
Optimizer	AdamW	AdamW
Learning rate	1×10^{-5} (constant)	1×10^{-6} (constant)
Warmup steps	10	0
Weight decay	0.01	0.01
Gradient Clip Norm	1.0	1.0

Table 10: Hyperparameters used for **SDPO** for each experimental setup.

Parameters	**Experiment 1** Section 3
General	
Model	Qwen/Qwen3-8B allenai/Olmo3-7B-Instruct
Thinking	False
Data	
Max. prompt length	2048
Max. response length	8192
Batching	
Question batch size	32
Mini batch size	8 (default) / 32 (on-policy)
Number of rollouts	8
Rollout	
Inference engine	vllm
Temperature	1.0
Validation	
Temperature	0.6
Top-p	0.95
Number of rollouts	16
Loss	
ϵ-high	0.28
Rollout importance sampling clip	2
KL coefficient (λ)	0.0
Training	
Optimizer	AdamW
Learning rate	1×10^{-6} (default) / 1×10^{-5} (on-policy)
Warmup steps	10
Weight decay	0.01
Gradient Clip Norm	1.0

Table 11: Hyperparameters used for **GRPO**.

I.2.1 Details on Hyperparameter Selection (Section 3)

For GRPO in the experiments in Section 3, we perform a grid search over learning rates $\{10^{-5}, 10^{-6}\}$ and minibatch sizes $\{8, 32\}$. For on-policy GRPO, we search over the same learning rates while fixing the minibatch size to 32. For SDPO, we grid-search over KL variants (forward KL, Jensen–Shannon), learning rates $\{10^{-5}, 10^{-6}\}$, and minibatch sizes $\{8, 32\}$. For each method (GRPO, on-policy GRPO, and SDPO), we select a *single* hyperparameter configuration that achieves the highest validation accuracy within the first 5 hours of training, evaluated across all datasets and models used in Section 3. We further report results obtained by selecting the optimal hyperparameter configuration separately for each model and dataset in Table 1.

I.3 User Templates

For multiple-choice questions and tool use, the model must be prompted in a task-specific manner. We therefore provide the prompt templates used for these settings below.

```
Given a question and four options, please select the right answer. Respond in the
    following format:
```

```
<reasoning>
...
</reasoning>
<answer>
...
</answer>

For the answer, only output the letter corresponding to the correct option (A, B, C,
     or D), and nothing else. Do not restate the answer text. For example, if the
    answer is "A", just output:
<answer>
A
</answer>
```

Listing 1: **System prompt: Multiple Choice Questions**

```
{question}
Please reason step by step.
```

Listing 2: **User prompt: Multiple Choice Questions**

```
You are a helpful function-calling AI assistant. You are provided with function
    signatures within <functions></functions> XML tags. You may call one or more
    functions to assist with the user query. Output any function calls within <
    function_calls></function_calls> XML tags. Do not make assumptions about what
    values to plug into functions.
```

Listing 3: **System prompt: Tool use**

```
Your task is to answer the user's question using available tools.
You have access to the following tools:
Name: Axolotl
Description: Collection of axolotl pictures and facts
Documentation:
getRandomAxolotlImage: Retrieve a random axolotl image with information on the
    image source.
Parameters: {}
Output: Successful response.
 - Format: application/json
 - Structure: Object{url, source, description}
searchAxolotlImages: Search for axolotl images based on specific criteria such as
    color, gender, and size.
Parameters: {"color": "string. One of: [wild, leucistic, albino]. The color of the
    axolotl (e.g., 'wild', 'leucistic', 'albino', etc.).", "gender": "string. One
    of: [male, female]. The gender of the axolotl ('male', 'female').", "size": "
    string. One of: [small, medium, large]. The size of the axolotl ('small', '
    medium', 'large').", "page": "integer. The page number for pagination purposes
    ."}
Output: Successful response.
 - Format: application/json
 - Structure: Object{results: Array[Object{url, source, description}], pagination:
    Object{current_page, total_pages, total_results}}
getAxolotlFacts: Retrieve interesting facts about axolotls such as their habits,
    habitats, and physical characteristics.
Parameters: {"category": "string. One of: [habits, habitat, physical
    characteristics]. The category of facts to retrieve (e.g., 'habits', 'habitat',
     'physical characteristics').", "limit": "integer. The maximum number of facts
    to return."}
Output: Successful response.
```

```
  - Format: application/json
  - Structure: Array[Object{fact, source}]

Use the following format:
Thought: you should always think about what to do
Action: the action to take, should be one of the tool names.
Action Input: the input to the action, must be in JSON format. All of the action
     input must be realistic and from the user.

Begin!
Question: Hey, can you show me a random picture of an axolotl?
```

Listing 4: **Example user prompt: Tool use**

J Qualitative Examples

J.1 Visualization of Advantages

Figure 17 compares the advantages of SDPO and GRPO in a representative example.

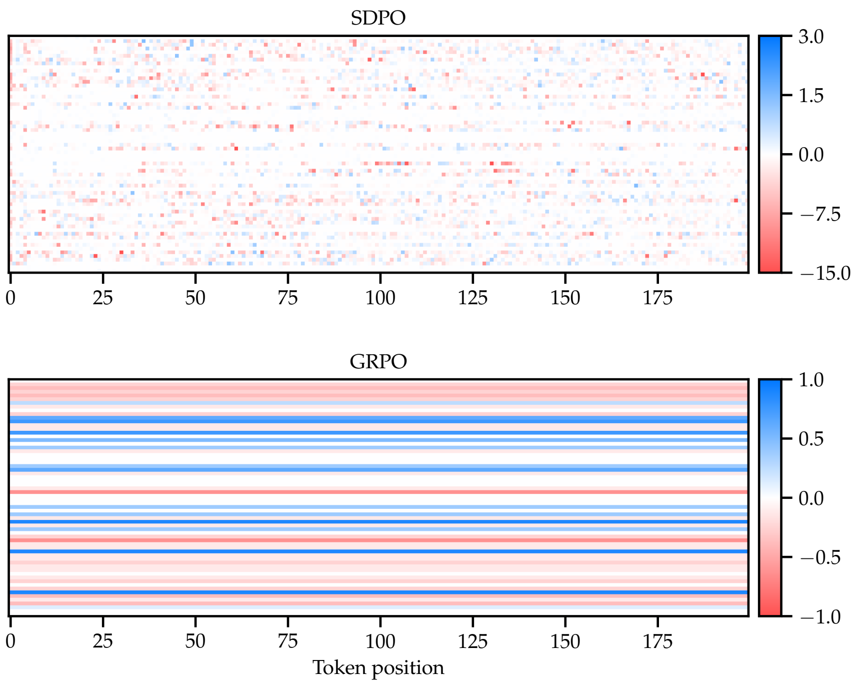


Figure 17: Visualization of advantages in SDPO and GRPO with Olmo3-7B-Instruct in a batch from the Chemistry task of Section 3. Each row corresponds to the beginning of a response. The color indicates the advantage value at that token position, with positive advantages shown in blue and negative advantages shown in red.

J.2 Examples

Below, we show an example from training SDPO on LCBv6 using Qwen3-8B.

```
[Prompt]

You are a coding expert. You will be given a coding problem, and you need to write a
     correct Python program that matches the specification and passes all tests. The time
     limit is 1 second. You may start by outlining your thought process. In the end,
     please provide the complete code in a code block enclosed with ``` ```.

You are given a binary string s of length n, where:

'1' represents an active section.
'0' represents an inactive section.

You can perform at most one trade to maximize the number of active sections in s. In a
     trade, you:

Convert a contiguous block of '1's that is surrounded by '0's to all '0's.
Afterward, convert a contiguous block of '0's that is surrounded by '1's to all '1's.

Return the maximum number of active sections in s after making the optimal trade.
Note: Treat s as if it is augmented with a '1' at both ends, forming t = '1' + s + '1'.
     The augmented '1's do not contribute to the final count.

Example 1:
```

```
Input: s = "01"
Output: 1
Explanation:
Because there is no block of '1's surrounded by '0's, no valid trade is possible. The
     maximum number of active sections is 1.

Example 2:

Input: s = "0100"
Output: 4
Explanation:

String "0100" -> Augmented to "101001".
Choose "0100", convert "101001" -> "100001" -> "111111".
The final string without augmentation is "1111". The maximum number of active sections is
      4.


Example 3:

Input: s = "1000100"
Output: 7
Explanation:

String "1000100" -> Augmented to "110001001".
Choose "000100", convert "110001001" -> "110000001" -> "111111111".
The final string without augmentation is "1111111". The maximum number of active sections
      is 7.


Example 4:

Input: s = "01010"
Output: 4
Explanation:

String "01010" -> Augmented to "1010101".
Choose "010", convert "1010101" -> "1000101" -> "1111101".
The final string without augmentation is "11110". The maximum number of active sections
     is 4.



Constraints:

1 <= n == s.length <= 10^5
s[i] is either '0' or '1'

Your solution should have the following signature: ```python
def maxActiveSectionsAfterTrade(s: str) -> int:
```

[Response]

<think>

</think>

To solve this problem, we need to understand the structure of the string and the trade
     operation. Here's the plan:

### Key Insights:
1. We are given a binary string `s`, and we are allowed **at most one trade**.
2. A **trade** consists of two steps:
   - Convert a contiguous block of `'1'` that is **surrounded by `'0'` into `'0'` (i.e.,
     flip it).
   - Then convert a contiguous block of `'0'` that is **surrounded by `'1'` into `'1'` (i
     .e., flip it).
3. The actual count of active sections is the number of `'1'` characters **after** the
     trade. The augmented `'1'` at the beginning and end do not count in the final result.

### Strategy:
- We simulate the trade by considering each possible way to flip a block of `'1'` (
     bounded by `'0'`) and then a block of `'0'` (bounded by `'1'`).
- For each possible valid block, simulate the trade and compute the resulting number of
     `'1'` characters.
- We only need to consider the original string `s`, not the augmented one (since we can
     simulate the trade directly on `s` by checking boundaries).
```
```

~~~
We'll use a sliding window technique to find contiguous blocks of `'1'` and `'0'` and
     simulate the trade.

---

### Implementation Steps:
1. Simulate the trade by checking for a block of `'1'` surrounded by `'0'` or at the
     edges (we simulate the augmented `1` at both sides).
2. For each such block, simulate converting it to `'0'`, then convert a `'0'` block (
     again, surrounded by `'1'`) to `'1'`.
3. Count the number of `'1'` in the final string and track the maximum.

---

### Python Code:
```python
def maxActiveSectionsAfterTrade(s: str) -> int:
    if not s:
        return 0

    max_ones = 0
    n = len(s)

    # We are simulating the augmented '1's on both ends, so we treat the boundaries as
     '1's
    # So '0' at index 0 or n-1 is considered surrounded by '1's

    # Step 1: Find all candidate positions for the first part of the trade
    # (convert a block of 1s surrounded by 0s into 0s)
    for i in range(n):
        if s[i] == '1':
            start = i
            while i < n and s[i] == '1':
                i += 1
            end = i - 1
            # Check if this block is surrounded by '0's
            left_boundary = (start == 0 or s[start - 1] == '0')
            right_boundary = (end == n - 1 or s[end + 1] == '0')
            if left_boundary and right_boundary:
                # This is a candidate block of '1's to flip
                # Now, look for a block of '0's surrounded by '1's in the remaining
     string
                # Replace the '1' block with '0's
                # Now find a '0' block surrounded by '1's
                # We can simulate the remaining string
                temp = list(s)
                for j in range(start, end + 1):
                    temp[j] = '0'

                # Now find a contiguous block of '0's surrounded by '1's
                max_trade_ones = 0
                for k in range(n):
                    if temp[k] == '0':
                        start0 = k
                        while k < n and temp[k] == '0':
                            k += 1
                        end0 = k - 1
                        # Check if this block is surrounded by '1's
                        left_0 = (start0 == 0 or temp[start0 - 1] == '1')
                        right_0 = (end0 == n - 1 or temp[end0 + 1] == '1')
                        if left_0 and right_0:
                            # Flip this block of '0's to '1's
                            for j in range(start0, end0 + 1):
                                temp[j] = '1'
                            # Count the number of '1's
                            count = sum(temp)
                            max_ones = max(max_ones, count)
                            break  # Only take the first valid block for this trade
                # After trying this '1' block, restore the original string
                # No need, we just used a copy
    return max_ones
```

---

### Optimization Note:
The above approach is **not optimal** for very large strings. A more efficient method
     would involve precomputing the positions of blocks of `'1'` and `'0'`, and checking
     valid trades between these blocks in linear time.
~~~

```
But with the constraints given (up to 10^5), we need to ensure linear or near-linear
     performance. The current code is correct but **not efficient** for large inputs.

Would you like me to provide a more optimized version that runs in **O(n)** time and uses
      only one pass through the string?


[Feedback]

b'Runtime Error\nTypeError: unsupported operand type(s) for +: \'int\' and \'str\'\nLine
     48 in maxActiveSectionsAfterTrade (Solution.py)\n\nLast Executed Input\n"11000"'
```

J.3 Environment Feedback

We show three examples of feedback in our coding environment, inspired by LeetCode.

```
Test Case 3: Wrong Answer

Input
[[3,6,1],[4,21,4]]

Output
6

Expected
22.87500

Test Case 6: Wrong Answer

Input
[[12,25,3],[3,14,2]]

Output
14

Expected
25.83333
```

Listing 5: Example of feedback "Wrong Answer" from our code environment in case of a wrong answer, inspired by LeetCode

```
Runtime Error
MemoryError:
Line 91 in <module> (Solution.py)
Line 25 in solve (Solution.py)

Last Executed Input
10
633 9312
1314 8548
8857 1062
6410 3289
8594 1263
8549 733
3858 5973
... (3 more lines)
```

Listing 6: Example of feedback "Memory Error" from our code environment in case of a wrong answer, inspired by LeetCode

```
Runtime Error
IndexError: list index out of range
Line 28 in sortMatrix (Solution.py)

Last Executed Input
[[-1,-1,-1,-1,-1,-1,-1,-1,...
```

Listing 7: Example of feedback "Index Error" from our code environment in case of a wrong answer, inspired by LeetCode

J.4 Illustrative Example

Figure 18 shows an illustrative example of the dense credit assignment in SDPO.

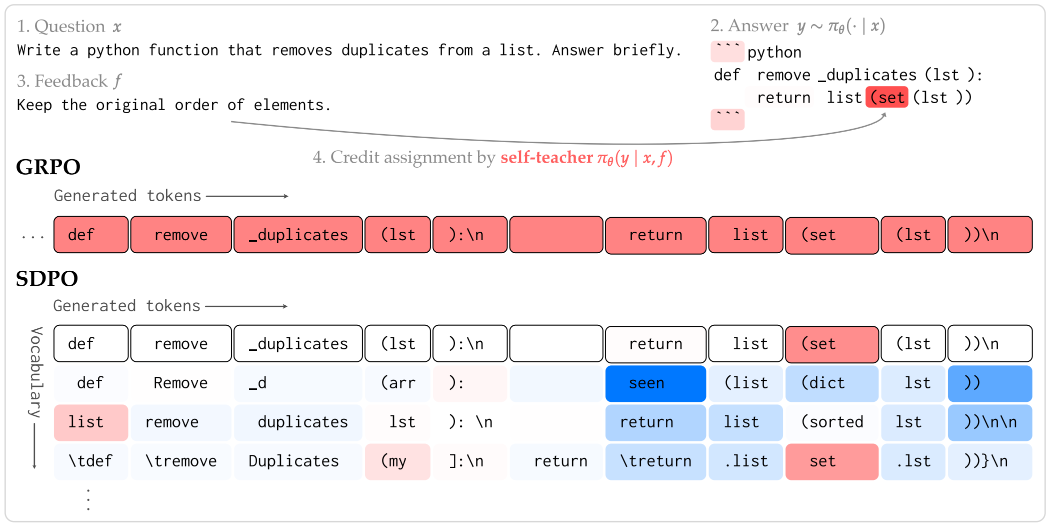

1. Question x
Write a python function that removes duplicates from a list. Answer briefly.
3. Feedback f
Keep the original order of elements.
2. Answer $y \sim \pi_\theta(\cdot \mid x)$
```python
def remove_duplicates (lst ):
return list (set (lst ))
```
4. Credit assignment by self-teacher $\pi_\theta(y \mid x, f)$
GRPO
Generated tokens
... def remove _duplicates (lst):\n return list (set (lst))\n
SDPO
Generated tokens
Vocabulary
def remove _duplicates (lst):\n return list (set (lst))\n
def Remove _d (arr): seen (list (dict lst))
list remove duplicates lst): \n return list (sorted lst))\n\n
\tdef \tremove Duplicates (my]:\n return \treturn .list set .lst))}\n
⋮

Figure 18: **Dense credit assignment through self-teaching in SDPO.** The answer is generated by then model (Qwen3-8B) before seeing the feedback. Then, we re-evaluate the log-probs of the original attempt with the self-teacher after seeing the feedback. We show the per-token $\log(\mathbb{P}(\text{self-teacher})/\mathbb{P}(\text{student}))$, with red indicating negative values (self-teacher disagrees), blue indicating positive values (teacher reinforces), and white indicating values around zero. Using binary rewards, GRPO would assign the same, negative advantage to all tokens in the sequence. In contrast, SDPO turns the feedback into dense credit assignment across the sequence. The first row shows the tokens of the generated response. The 3 other rows show the top-k logits of the self-teacher that are used during self-distillation, suggesting alternative tokens. Notably, in this example, the self-teacher identifies the error through retrospection without an explicit solution. The credit assignment on the generated sequence, and the alternative top-k logits correctly show that replacing `set` with `dict` maintains the order of elements. Further, in the seventh shown position, the model also identifies an alternative solution path which starts with the `seen` token, instead of directly returning the output. The activation is sparse, identifying where mistakes happen and adjusting to the students' response distribution for specifically these few tokens.