TEST-TIME 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

We study the discovery problem in difficult binary-reward tasks, where the goal is to find a solution in as few attempts as possible. Whereas prior approaches rely on repeatedly sampling from the base model reflecting on past failures, we introduce a Test-Time Training (TTT) method that enables the model to continue learning at test-time well beyond its limited context length. Previous applications of TTT to discovery problems have been limited to continuous rewards, since they allow hill-climbing on suboptimal solutions with scalar rewards. This fails in binary-reward tasks where the reward only provides a learning signal once a solution has already been found. We introduce Test-Time Self-Distillation, which converts environment feedback on past failures into dense learning signals through **Self-Distillation Policy Optimization (SDPO)**. 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. On difficult competitive programming problems from LiveCodeBench, test-time self-distillation achieves the same discovery probability as best-of-k sampling or multi-turn conversations with $3\times$ fewer attempts.

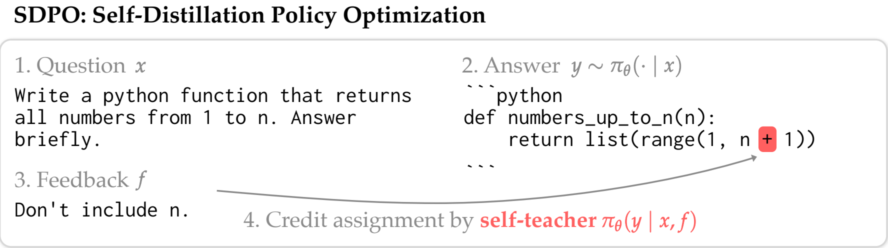
1 INTRODUCTION

Test-time training (TTT) is a powerful paradigm for adapting Large Language Models (LLMs) at inference time by allocating additional compute to specialize the model to a specific task (Sun et al., 2020; 2025; Hardt & Sun, 2024; Hübotter et al., 2025a; Akyürek et al., 2025; Behrouz et al., 2025; Tandon et al., 2025; Hübotter et al., 2026). While TTT has recently led to notable discoveries in continuous-reward problems (e.g., Wang et al., 2025; Yuksekgonul et al., 2025), continually learning from attempts of binary-reward problems is more challenging. In contrast to continuous-reward problems that provide a clear learning signal even for suboptimal attempts, rewards in binary-reward problems yield no learning signal until the first solution has been found. For this reason, methods based on reinforcement learning with verifiable rewards (RLVR) do not improve over simple best-of-k sampling. Thus, while standard RLVR methods such as Group Relative Policy Optimization (GRPO; Shao et al., 2024) have proven effective for improving *train-time* reasoning (Guo et al., 2025; Kimi et al., 2025; Olmo et al., 2025; Jaech et al., 2024; Lambert et al., 2025), they are ill-suited for the test-time regime because they cannot learn before the first solution is found. Without auxiliary rewards (e.g., via process reward models, Lightman et al., 2023; Wang et al., 2024; Setlur et al., 2025) or a curated learning curriculum of increasingly difficult instances (Zhao et al., 2025; Huang et al., 2026; Diaz-Bone et al., 2025; Hübotter et al., 2025b), the model is left without learning signal.

However, many verifiable environments already expose *rich textual feedback* beyond a binary reward. In coding environments such as LeetCode, failed attempts are often accompanied by runtime errors, tracebacks, or failing unit tests. This feedback not only reveals *whether* a rollout was wrong, but also *what* went wrong. Building on the strong in-context learning capabilities of LLMs (Brown et al., 2020), many recent works use this feedback to iteratively generate corrections (Chen et al., 2021a; Madaan et al., 2023; Shinn et al., 2023; Yao et al., 2024; Yuksekgonul et al., 2025; Lee et al., 2025). Yet, multi-turn loops are fundamentally limited by the transformer's fixed context window

---

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



`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 1: 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 and adjusting to the students' response distribution.

and the increasing noise of long-horizon interactions. The central question becomes: how can we convert rich feedback into effective gradient signals that guide the discovery of solutions at test-time?

We introduce Test-Time Self-Distillation, which converts rich environment feedback into persistent model updates without requiring external supervision or auxiliary reward models. At the heart of our approach is Self-Distillation Policy Optimization (SDPO), an on-policy RL algorithm that leverages the model's own in-context learning ability to act as a "self-teacher". 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. For example, when provided with the feedback from Figure 4, the self-teacher can identify how the initial attempt should be modified to avoid the runtime error. 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. SDPO distills these feedback-informed insights back into the model weights, effectively "compressing" long-context reasoning into the policy itself, illustrated in Figure 2. We include a detailed summary of related work in Section 4.

On very hard competitive programming questions from LiveCodeBench (LCB; Jain et al., 2025) where the base model's pass@64 is below 0.03, test-time SDPO significantly accelerates discovery compared to best-of-k and multi-turn sampling, requiring 3× fewer attempts to discover a solution.

2 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 student's original attempt y, f may incorporate any environment feedback such as runtime errors from a coding environment. Intuitively, 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 environment feedback. We use the KL-divergence, $\mathrm{KL}(p \| q) = \sum_i p(i) \log {p(i)}/{q(i)}$, as a distance measure for next-token distributions of student and teacher, and optimize a standard logit distillation loss:

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

where the stopgrad operator, sg, 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 1 in the appendix shows an example of self-teaching with Qwen3-8B as student and self-teacher. We summarize SDPO in Algorithm 1 in the appendix.

We derive the SDPO gradient as follows (cf. Section D.1):

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

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

## 3 SOLVING HARD PROBLEMS VIA TEST-TIME SELF-DISTILLATION

We study a test-time setting where the model is given only a single hard (binary-reward) question $x$ and must discover a solution as quickly as possible:

**Definition 3.1** (Discovery time)**.** The discovery time is the number of trials needed until a solution is found (i.e., the smallest $k$ with the $k$-th attempt $y_k$ receiving reward 1).

Based on this notion, we can define a measure of the efficacy of discovery:

$$\begin{aligned} \text{discovery@}k &:= \mathbb{P}(\text{discovery time} \leq k) \\ &= \mathbb{P}(r(y_1 \mid x) = 1 \text{ or } r(y_2 \mid x) = 1 \text{ or} \ldots \text{or } r(y_k \mid x) = 1), \end{aligned} \tag{3}$$

where the probability is over any randomness in the algorithm producing $y_k$ and the rewards. Thus, the discovery@$k$ metric quantifies the probability of discovering the solution within $k$ steps.[1] While prior work has studied discovery with continuous rewards (Romera-Paredes et al., 2024; Novikov et al., 2025; Georgiev et al., 2025; Wang et al., 2025; Phan et al., 2025; Surina et al., 2025; Yuksekgonul et al., 2026), discovery with language models in sparse or binary-reward settings does not allow "hill-climbing" a continuous reward and has remained less well understood. The canonical pass@$k$ metric for best-of-$k$ sampling is exactly the probability of discovering at least one solution within $k$ independent samples from a fixed model, coinciding with discovery@$k$. The discovery@$k$ metric generalizes pass@$k$ to algorithms that sample attempts sequentially.

In this section, we seek to answer the question: Can repeatedly compressing context into model weights via self-distillation accelerate discovery for hard questions?

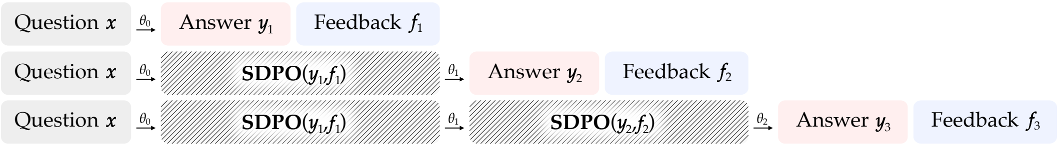


Figure 2: **Compressing context into model weights via self-distillation.** We illustrate the process of distilling the interaction history (context $c$) into the model parameters $\theta$. The model $\pi_\theta$ repeatedly attempts a fixed hard question $x$, generating an answer $y$ and receiving feedback $f$. Rather than appending this history to the context window, the model updates its weights $\theta_t \rightarrow \theta_{t+1}$ with SDPO (batch size 1) based on the feedback, effectively "fixing" mistakes by encoding $\pi_\theta(\cdot \mid x, c)$ directly into the policy $\pi_{\theta'}(\cdot \mid x)$.

### 3.1 EXPERIMENTAL SETTING

We consider a particularly challenging subset of questions from LiveCodeBench[2] that are at Qwen3-8B's performance ceiling and require significant test-time sampling to find any solution. Concretely, we define two groups using Qwen3-8B's pass@$k$: *Hard tasks* with pass@64 $< 0.5$ and *very hard*

---

[1] Our proposed discovery@$k$ metric is a canonical metric in the study of runtime speedup (i.e., time until termination, Dolan & Moré (2002)).

[2] We consider the LCBv6 subset, which contains questions released between February and May 2025.

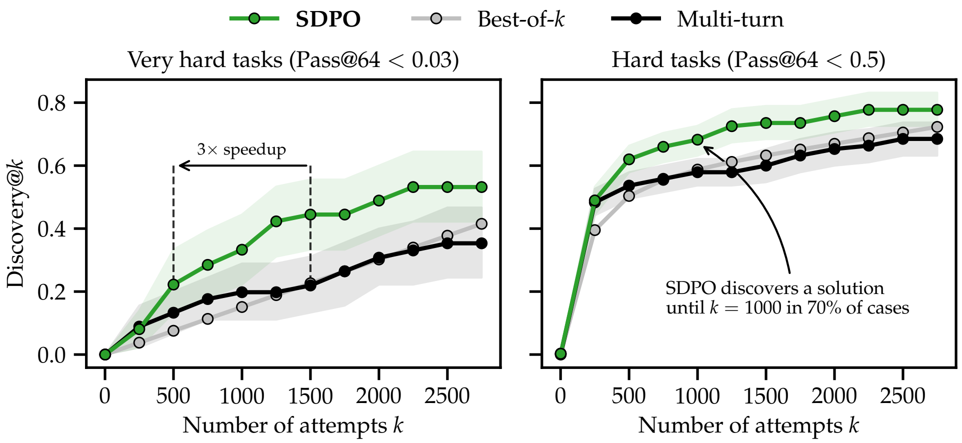


Figure 3: **Self-distillation at test-time significantly accelerates discovery for hard LiveCodeBench questions. Left:** Very hard questions (9 total) from LCBv6 where the base model achieves pass@64 $< 0.03$, i.e., in less than 3% cases, sampling 64 responses yields any success. **Right:** Hard questions (19 total) from LCBv6 where the base model achieves pass@64 $< 0.5$. We report the discovery@$k$ metric, representing the probability of discovering at least one solution within $k$ total generations. Across both difficulty levels, SDPO achieves higher discovery@$k$ rates at almost all generation budgets, compared to the base model and a multi-turn conversation baseline that receives the feedback in-context. We report the mean and bootstrapped 90% confidence intervals of the mean across 5 random seeds per question.

*tasks* with pass@64 $< 0.03$. Among these, we retain questions for which any of best-of-$k$, multi-turn, or SDPO find at least one solution within 512 steps across 5 seeds. This results in 19 hard and 9 very hard questions.

For best-of-$k$ sampling under the base model, we report the standard pass@$k$ estimate (Chen et al., 2021b) from 2944 independent rollouts. As multi-turn sampling, we sequentially reprompt the model in-context using the concatenated feedback from previous attempts. To remain within Qwen3-8B's 40k-token context limit, we employ a first-in, first-out sliding window, discarding the earliest feedback once the maximum prompt length (32k tokens) is reached. We ablate the multi-turn reprompting strategy in Figure 8 in Section F and find that retaining only past feedback while forgetting earlier attempts significantly outperforms the baseline that additionally retains past attempts. We evaluate SDPO with a batch size of 16. We ablate this choice in Figure 8 in Section F and find that overall performance differences are marginal, yet smaller batch sizes are beneficial for improvements at low generation budgets, while larger batch sizes result in more stable updates that still learn to solve questions at later stages into the run.

### 3.2 RESULTS

Figure 3 compares discovery@$k$ for SDPO, multi-turn sampling, and best-of-$k$ sampling on very hard (left) and hard (right) questions from LCBv6. Across both difficulty levels, SDPO achieves substantially higher discovery@$k$ rates at almost all generation budgets.

On very hard tasks, SDPO achieves a discovery@2750 of 53.2%, significantly higher than the $41.5\%$ and $35.6\%$ achieved by best-of-k and multi-turn sampling, respectively. SDPO not only solves more questions overall but also does so with substantially fewer attempts. Notably, to reach a 22% discovery probability on very hard questions, SDPO requires approximately 3× fewer generations than best-of-$k$ and multi-turn sampling. On hard tasks, SDPO reaches a $67\%$ discovery probability with roughly 2.4× fewer generations than best-of-$k$ and multi-turn sampling. The context window length for multi-turn sampling is reached after 837 (±466) steps for hard questions and after 1007 (±349) steps for very hard questions, offering a possible explanation for its diminishing gains at high generation budgets.

**Question 3 is only solved by SDPO.** SDPO solves all questions that are solved by best-of-$k$ and multi-turn sampling. Beyond that, SDPO uniquely discovers a solution for Q3, which is neither solvable with multi-turn sampling nor with best-of-$k$ sampling within 2750 attempts. In contrast, SDPO first discovers a solution for Q3 after 321 attempts, which corresponds to 20 iteration steps of self-distillation based on feedback with a batch size of 16. We include detailed per-question results in Table 3 in Section F.

**The initial self-teacher does not solve hard questions.** Notably, the self-teacher's initial accuracy is $< 1\%$ for almost all questions, and even exactly $0\%$ on $78\%$ of them (Table 4 in Section F). This shows that a single turn of in-context feedback is insufficient to solve the problem. Despite this, the self-teacher's credit assignment is sufficiently effective for SDPO to iteratively refine the policy and eventually solve these questions.

## 4 RELATED WORK

### 4.1 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., 2024; 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.

### 4.2 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., 2024; 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., 2021a; 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.

### 4.3 DISTILLATION

Distillation is frequently employed as an alternative to supervised fine-tuning (SFT) when a strong teacher model is available. Distillation 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). While often 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 on its own generations provided by an external teacher (Agarwal et al., 2024; Gu et al., 2024; Yang et al., 2025a; 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).

### 4.4 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; 2025b) and factual information (Eyuboglu et al., 2026; Kujanpää et al., 2025; Cao et al., 2025a) 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; Song et al., 2026). These approaches use an *off-policy* self-distillation objective, which we find to substantially underperform 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. (2025) 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. Other concurrent work studies SDPO on a fixed dataset of expert demonstrations, without online environment interaction (Shenfeld et al., 2026; Zhao et al., 2026).

## 5 CONCLUSION

We 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. By distilling these retrospective insights back into the model weights, SDPO effectively "compresses" complex interaction histories into the policy parameters.

Our experiments on hard LiveCodeBench questions demonstrate that SDPO significantly accelerates test-time discovery on hard, binary-reward tasks, requiring 3× fewer attempts to reach the same discovery probability as best-of-$k$ and multi-turn sampling. Our work highlights continual improvement at test-time on difficult tasks as an exciting direction for future work, and provides a first demonstration of the potential of test-time training in this setting.

## 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.

Ekin Akyürek, Mehul Damani, Adam Zweiger, Linlu Qiu, Han Guo, Jyothish Pari, Yoon Kim, and Jacob Andreas. The surprising effectiveness of test-time training for few-shot learning. In *ICML*, 2025.

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.

Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to memorize at test time. In *NeurIPS*, 2025.

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*, 2025a.

Meng Cao, Shuyuan Zhang, Xiao-Wen Chang, and Doina Precup. Scar: Shapley credit assignment for more efficient rlhf. *arXiv preprint arXiv:2505.20417*, 2025b.

Alex J Chan, Hao Sun, Samuel Holt, and Mihaela Van Der Schaar. Dense reward for free in reinforcement learning from human feedback. In *ICML*, 2024.

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*, 2021a.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021b.

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*, 2025.

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.

Leander Diaz-Bone, Marco Bagatella, Jonas Hübotter, and Andreas Krause. Discover: Automated curricula for sparse-reward reinforcement learning. In *NeurIPS*, 2025.

Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2), 2002.

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.

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.

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*, 2024.

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.

Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864*, 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.

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.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*, 2018.

Moritz Hardt and Yu Sun. Test-time training on nearest neighbors for large language models. In *ICLR*, 2024.

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.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-zero: Self-evolving reasoning llm from zero data. In *ICLR*, 2026.

Jonas Hübotter, Patrik Wolf, Alexander Shevchenko, Dennis Jüni, Andreas Krause, and Gil Kur. Specialization after generalization: Towards understanding test-time training in foundation models. In *ICLR*, 2026.

Jonas Hübotter, Sascha Bongni, Ido Hakimi, and Andreas Krause. Efficiently learning at test-time: Active fine-tuning of llms. In *ICLR*, 2025a.

Jonas Hübotter, Leander Diaz-Bone, Ido Hakimi, Andreas Krause, and Moritz Hardt. Learning on the job: Test-time curricula for targeted reinforcement learning. *arXiv preprint arXiv:2510.04786*, 2025b.

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.

Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2), 1998.

Amirhossein Kazemnejad, Milad Aghajohari, Eva Portelance, Alessandro Sordoni, Siva Reddy, Aaron Courville, and Nicolas Le Roux. Vineppo: Refining credit assignment in rl training of llms. In *ICML*, 2025.

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.

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.

Sergey Levine. Reinforcement learning and control as probabilistic inference: Tutorial and review. *arXiv preprint arXiv:1805.00909*, 2018.

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.

Andrew Y Ng, Stuart Russell, et al. Algorithms for inverse reinforcement learning. In *ICML*, 2000.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 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.

Peter Phan, Dhruv Agarwal, Kavitha Srinivas, Horst Samulowitz, Pavan Kapanipathi, and Andrew McCallum. Migrate: Mixed-policy grpo for adaptation at test-time. *arXiv preprint arXiv:2508.08641*, 2025.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. In *NeurIPS*, 2023.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995), 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.

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, Mehul Damani, Jonas Hübotter, and Pulkit Agrawal. Self-distillation enables continual learning. *arXiv preprint arXiv:2601.19897*, 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.

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

Yuda Song, Lili Chen, Fahim Tajwar, Remi Munos, Deepak Pathak, J Andrew Bagnell, Aarti Singh, and Andrea Zanette. Expanding the capabilities of reinforcement learning via text feedback. *arXiv preprint arXiv:2602.02482*, 2026.

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.

Yu Sun, Xiaolong Wang, Zhuang Liu, John Miller, Alexei Efros, and Moritz Hardt. Test-time training with self-supervision for generalization under distribution shifts. In *ICML*, 2020.

Yu Sun, Xinhao Li, Karan Dalal, Jiarui Xu, Arjun Vikram, Genghan Zhang, Yann Dubois, Xinlei Chen, Xiaolong Wang, Sanmi Koyejo, et al. Learning to (learn at test time): Rnns with expressive hidden states. In *ICML*, 2025.

Anja Surina, Amin Mansouri, Lars Quaedvlieg, Amal Seddas, Maryna Viazovska, Emmanuel Abbe, and Caglar Gulcehre. Algorithm discovery with llms: Evolutionary search meets reinforcement learning. In *COLM*, 2025.

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

Arnuv Tandon, Karan Dalal, Xinhao Li, Daniel Koceja, Marcel Rød, Sam Buchanan, Xiaolong Wang, Jure Leskovec, Sanmi Koyejo, Tatsunori Hashimoto, et al. End-to-end test-time training for long context. *arXiv preprint arXiv:2512.23675*, 2025.

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*, 2024.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, et al. Thetaevolve: Test-time learning on open problems. *arXiv preprint arXiv:2511.23473*, 2025.

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*, 2025a.

Wenkai Yang, Yankai Lin, Jie Zhou, and Ji-Rong Wen. Distilling rule-based knowledge into large language models. In *COLING*, 2025b.

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.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, et al. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

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.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. In *NeurIPS*, 2025.

Siyan Zhao, Zhihui Xie, Mengchen Liu, Jing Huang, Guan Pang, Feiyu Chen, and Aditya Grover. Self-distilled reasoner: On-policy self-distillation for large language models. *arXiv preprint arXiv:2601.18734*, 2026.

Tianyu Zheng, Tianshun Xing, Qingshui Gu, Taoran Liang, Xingwei Qu, Xin Zhou, Yizhi Li, Zhoufutu Wen, Chenghua Lin, Wenhao Huang, et al. First return, entropy-eliciting explore. *arXiv preprint arXiv:2507.07017*, 2025.

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.

Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, Anind K Dey, et al. Maximum entropy inverse reinforcement learning. In *AAAI*, 2008.

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.

# A FIGURES

This section contains figures supporting the main text.

- Table 1 summarizes how SDPO is positioned relative to RLVR and distillation baselines.
- Figure 4 shows an example for rich feedback in a code environment.
- Figure 1 illustrates the SDPO algorithm.
- Algorithm 1 shows the SDPO training loop.
- Table 2 shows the reprompt template for the self-teacher.
- Figure 5 illustrates dense credit assignment in SDPO.
- Figure 2 illustrates the TTT setting.

| **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 1: 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 4: Example of feedback from our code environment, inspired by LeetCode. Listings 1, 2, and 3 show examples of feedback in case of a wrong answer, a memory error, and an index error.

Algorithm 1 SDPO training loop (self-distillation with rich feedback)

```
Require: Language model $\pi_\theta$; dataset of questions $x$; number of rollouts $G$ per question; environment that returns feedback $f$ for an attempt.
repeat
    Sample question $x$ from the dataset.
    Sample responses $\{y_i\}_{i=1}^{G} \sim \pi_\theta(\cdot \mid x)$.
    Evaluate each response to obtain feedback $\{f_i\}_{i=1}^{G}$.
    Self-distillation (teacher = current policy):
    Compute token log-probabilities $\log \pi_\theta(y_{i,t} \mid x, f_i, y_{i,<t})$
    Update $\theta$ by gradient descent on $\mathcal{L}_{\mathrm{SDPO}}(\theta)$.
until converged
```

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 2: 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 4) 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.

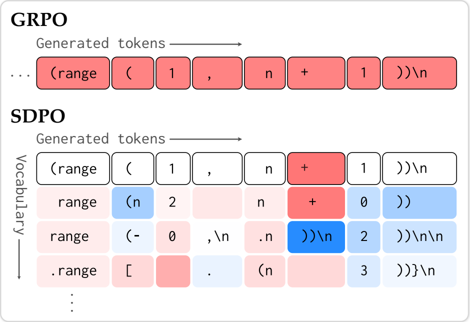


Figure 5: Dense credit assignment in SDPO in the example from Figure 1. Shown in blue are tokens which become more likely under the self-teacher. The self-teacher identifies how the returned range has to be modified so that it does not contain `n`.

B EXTENDED SECTION 2

B.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 6 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;[3] compute time can be further reduced by using larger micro batch sizes.

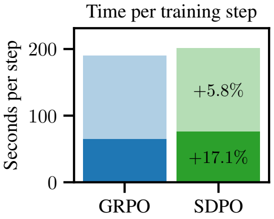


Figure 6: Time per step for SDPO vs GRPO (solid: without code environment, light: with code environment).

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. Section D.3). With a reasonable choice of K (e.g., $K = 100$), this avoids virtually any memory overhead while capturing most of the information.

B.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. Section D.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).

---

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

C RELATED WORK

C.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., 2024; 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 retrospective model 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.

C.2 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., 2025a; 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).

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) and factual information (Eyuboglu et al., 2026; Kujanpää 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; 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 recent work, Chen et al. (2025) 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.

Self-Distillation. Self-distillation has proven effective for diverse tasks such as compressing factual information (Eyuboglu et al., 2026; Kujanpää et al., 2025; Cao et al., 2025a) or prompt-based behaviors (Choi et al., 2022) into model weights. Regarding feedback-based learning, prior off-policy methods (Mitra & Ulukus, 2025) differ fundamentally from SDPO. Specifically, off-policy self-distillation optimizes the student to mimic the teacher’s success, whereas SDPO’s on-policy formulation optimizes the student to correct its own specific failures. We also note concurrent work by

Chen et al. (2025), who apply on-policy self-distillation in grid world settings with scalar rewards. Their method uses a reflection stage to diagnose mistakes, showing that self-distillation can offer improved credit assignment over learned value networks even in simpler, non-linguistic domains.

C.3 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., 2024; 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., 2021a; 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.

C.4 FURTHER RELATED WORK

Value networks and Monte Carlo advantage estimation. Several prior approaches aim to improve credit assignment but face the same information bottleneck as GRPO. Classical RL frequently trains value networks which provide token-level advantages, but themselves are learned from scalar rewards (Schulman et al., 2016; 2017). Furthermore, value networks incur significant computational and memory overhead and are therefore typically not used to train LLMs. Other recent work estimates token-level advantages by performing additional generations starting from various positions in the original attempt (Kazemnejad et al., 2025; Zheng et al., 2025). While this can learn with fewer gradient steps than GRPO it still uses only scalar rewards as signal and requires costly additional generations.

Dense credit assignment with a reward model. Several recent works have explored assigning dense (per-token) rewards given access to an external reward model, leveraging internal structure of the reward model (Chan et al., 2024; Cao et al., 2025b).

Partial observability. From the perspective of classical RL, many verifiable domains for LLMs are naturally *partially observable*: executing a proposed solution induces a latent environment state (e.g., failing tests or states of an agentic system) that is revealed only through rich feedback. This aligns with the formalism of partially observable Markov decision processes (POMDPs), where agents must act under incomplete observations of state (Kaelbling et al., 1998; Sutton & Barto, 1998). By contrast, RLVR and RLHF pipelines typically discard this observation channel and learn only from terminal scalar rewards or pairwise preferences.

C.5 TEST-TIME TRAINING

Our setting from Section 3 can be seen as a special case of test-time training where the model itself is updated at test-time using self-distillation. Updating the model at test-time is known as test-time training (Sun et al., 2020; 2025; Hardt & Sun, 2024; Hübotter et al., 2025a;b; Akyürek et al., 2025; Behrouz et al., 2025; Tandon et al., 2025; Hübotter et al., 2026). Unlike prior work, self-distillation uses the in-context learning ability of the current model to attribute credit after receiving feedback. This can be seen as simulating long-context reasoning with periodic compression of context into the model weights.

C.6 SDPO AS MAXIMUM ENTROPY RL

The SDPO objective resembles the objective in maximum entropy RL (e.g., Levine, 2018; Haarnoja et al., 2018) with a particular choice of reward function.

Maximum Entropy RL Consider optimizing

$$\arg\max_{\theta} \mathbb{E}_{y\sim\pi_\theta(\cdot|x)}\left[\sum_t r(y_t \mid x, y_{<t})\right] + \lambda \mathrm{H}[\pi_\theta(\cdot \mid x)], \quad \lambda > 0 \tag{4}$$

where $\pi_\theta(y \mid x) = \prod_{t=1}^{T} \pi_\theta(y_t \mid x, y_{<t})$ and $\mathrm{H}[\pi_\theta(\cdot \mid x)] = \mathbb{E}_{y\sim\pi_\theta(\cdot|x)}[-\log \pi_\theta(y \mid x)]$ is the entropy of the policy. Here, $r(y_t \mid x, y_{<t})$ is an arbitrary reward function, possibly “dense” (i.e., per-token). Equation (4) is known as maximum entropy RL. It is known that this objective is equivalent to solving a variational inference problem which discuss next.

To this end, we define a Bernoulli random variable $\mathcal{C}$ which is 1 if the attempt y is correct and 0 otherwise. We then define its distribution as $p(\mathcal{C} = 1 \mid x, y) \propto \exp(\frac{1}{\lambda}\sum_t r(y_t \mid x, y_{<t}))$. Further assuming w.l.o.g. that the “prior” over responses is uniform, we can express the posterior conditioned on the event of correctness as

$$\pi^\star(y \mid x) := p(y \mid x, \mathcal{C} = 1) \propto p(\mathcal{C} = 1 \mid x, y) \propto \exp\left(\frac{1}{\lambda}\sum_t r(y_t \mid x, y_{<t})\right). \tag{5}$$

Then, Equation (4) is equivalent to minimizing the KL divergence with respect to $\pi^\star$:

$$\arg\min_{\theta} \sum_t \mathrm{KL}(\pi_\theta(y_t \mid x, y_{<t}) \| \pi^\star(y_t \mid x, y_{<t})). \tag{6}$$

SDPO optimizes an implicit reward defined by the teacher Note that Equation (6) is equivalent to the SDPO objective (Equation (1)) with implicit reward $r(y_t \mid x, y_{<t}) = \log q(y_t \mid x, f, y_{<t})$ and $\lambda = 1$. In this sense, SDPO can be seen as a maximum entropy RL algorithm with dense rewards constructed implicitly through the retrospective model.

This also points to a connection of SDPO to inverse RL (Ng et al., 2000; Ziebart et al., 2008; Rafailov et al., 2023), where the goal is to recover an unknown reward function. In SDPO, the student learns an implicit reward function defined by the retrospective model.

D IMPLEMENTATION OF SDPO

The following pseudocode in Figure 7 outlines the implementation of SDPO:

```python
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 7: 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 Section D.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 (Section D.1)
- Teacher regularization (Section D.2)
- Approximating logit-distillation with the top-K logits for saving GPU memory (Section D.3)
- Generalizing PPO-style policy gradient algorithms to logit-level advantages (Section D.4)

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

D.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):

$$\mathcal{L}_{\text{token}}(\theta) := \mathbb{E}_{y\sim\text{stopgrad}(\pi_\theta(\cdot|x))}\left[\sum_{t=1}^{T} \text{KL}(\pi_\theta(\cdot \mid x, y_{<t}) \| \text{stopgrad}(\pi_\theta(\cdot \mid x, f, y_{<t})))\right] \tag{7}$$

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

$$\boldsymbol{\nabla}\mathcal{L}_{\text{token}}(\theta) = \mathbb{E}_{y\sim\pi_\theta(\cdot|x)}\left[\sum_{t=1}^{T} \mathbb{E}_{\hat{y}_t\sim\pi_\theta(\cdot|x,y_{<t})}\left[\boldsymbol{\nabla}_\theta \log \pi_\theta(\hat{y}_t \mid x, y_{<t}) \cdot \log \frac{\pi_\theta(\hat{y}_t \mid x, y_{<t})}{\pi_\theta(\hat{y}_t \mid x, f, y_{<t})}\right]\right]. \tag{8}$$

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.,

$$\begin{aligned}\mathcal{L}_{\text{seq}}(\theta) := \text{KL}(\pi_\theta \| q_\theta) &= \mathbb{E}_{y \sim \pi_\theta(\cdot \mid x)}\left[\log \frac{\pi_\theta(y \mid x)}{q_\theta(y \mid x, f)}\right] \\ &= \sum_{t=1}^{T} \mathbb{E}_{s_t \sim \Pi_\theta}\left[\text{KL}(\pi_\theta(\cdot \mid s_t) \| q_\theta(\cdot \mid s_t, f))\right],\end{aligned} \tag{9}$$

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

$$\boldsymbol{\nabla}\mathcal{L}_{\text{seq}}(\theta) = \boldsymbol{\nabla}\mathcal{L}_{\text{token}}(\theta) + \mathbb{E}_{y \sim \pi_\theta(\cdot \mid x)}\left[\sum_{t=1}^{T} \text{KL}(\pi_\theta(\cdot \mid s_t) \| q_\theta(\cdot \mid s_t, f)) \boldsymbol{\nabla}_\theta \log \Pi_\theta(s_t)\right]. \tag{10}$$

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.

D.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:

$$\sum_t \text{KL}(q(y_t \mid x, f, y_{<t}) \| q_{\theta_{\text{ref}}}(y_t \mid x, f, y_{<t})) \leq \epsilon, \quad \epsilon > 0. \tag{11}$$

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

   $$q(y_t \mid x, f, y_{<t}) \propto \exp\big((1-\alpha)\log q_{\theta_{\text{ref}}}(y_t \mid x, f, y_{<t}) + \alpha \log q_\theta(y_t \mid x, f, y_{<t})\big), \tag{12}$$

   with $\alpha \in (0, 1)$ the inverse Lagrange multiplier for the trust-region constraint. We include a full derivation in Section E.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.

D.3 APPROXIMATE LOGIT DISTILLATION

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

$$
\begin{aligned}
\mathcal{L}_{\text{SDPO}}(\theta) &= \mathbb{E}_{y\sim\pi_\theta(\cdot|x)} \sum_{t=1}^{T} \mathrm{KL}(\pi_\theta(y_t \mid x, y_{<t}) \| \mathrm{stopgrad}(q_\theta(y_t \mid x, f, y_{<t}))) \\
&\approx E_{y\sim\pi_\theta(\cdot|x)} \sum_{t=1}^{T} \sum_{y_t \in \mathrm{top}_K(\pi_\theta)} \pi_\theta(y_t \mid x, y_{<t}) \cdot \log \frac{\pi_\theta(y_t \mid x, y_{<t})}{\mathrm{stopgrad}(q_\theta(y_t \mid x, f, y_{<t}))} \\
&\quad + \underbrace{\left(1 - \textstyle\sum_{y_t \in \mathrm{top}_K(\pi_\theta)} \pi_\theta(y_t \mid x, y_{<t})\right) \cdot \log \frac{1 - \sum_{y_t \in \mathrm{top}_K(\pi_\theta)} \pi_\theta(y_t \mid x, y_{<t})}{\mathrm{stopgrad}\left(1 - \sum_{y_t \in \mathrm{top}_K(\pi_\theta)} q_\theta(y_t \mid x, f, y_{<t})\right)}}_{\text{tail}}
\end{aligned}
\tag{13}
$$

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.

D.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):

$$
\mathcal{L}_{\text{token}}(\theta) := -\frac{1}{\sum_{i=1}^{G} |y_i|} \sum_{i=1}^{G} \sum_{t=1}^{|y_i|} \min\left(w_{i,t}^{\text{TIS}}, \rho\right) \min\left(w_{i,t} A_{i,t}, \mathrm{clip}(w_{i,t}, 1 - \varepsilon_{\text{low}}, 1 + \varepsilon_{\text{high}}) A_{i,t}\right), \tag{14}
$$

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

We extend this to a logit-level loss:

$$
\begin{aligned}
\mathcal{L}_{\text{logit}}(\theta) := -\frac{1}{\sum_{i=1}^{G} |y_i|} \sum_{i=1}^{G} \sum_{t=1}^{|y_i|} \sum_{\hat{y}_{i,t}} &\min\left(\pi_{\theta_{\text{old}}}(\hat{y}_{i,t} \mid x, y_{i,<t}), \rho \pi_{\theta_{\text{old}}}^{\text{rollout}}(\hat{y}_{i,t} \mid x, y_{i,<t})\right) \\
&\min\left(w_{i,t}(\hat{y}_{i,t}) A_{i,t}(\hat{y}_{i,t}), \mathrm{clip}(w_{i,t}(\hat{y}_{i,t}), 1 - \varepsilon_{\text{low}}, 1 + \varepsilon_{\text{high}}) A_{i,t}(\hat{y}_{i,t})\right),
\end{aligned}
\tag{15}
$$

where $\hat{y}_{i,t}$ sums over all possible tokens at position t for rollout i (or the K most likely under $\pi_{\theta_{\text{old}}}$, cf. Section D.3). The TIS changes since we explicitly weight each logit by its probability under $\pi_{\theta_{\text{old}}}$ 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.

E THEORETICAL ANALYSIS

This section is organized as follows:

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

E.1 PROOF OF PROPOSITION 2.1.

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

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

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 $\mathcal{L}_{\mathrm{SDPO}}$ is

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

□

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

E.2 TRUST-REGION TEACHER

To stabilize training, we seek to prevent the teacher q from diverging from the initial teacher $q_{\theta_{\mathrm{ref}}}$. 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:

$$\sum_t \mathrm{KL}(q(y_t \mid x, f, y_{<t}) \| q_{\theta_{\mathrm{ref}}}(y_t \mid x, f, y_{<t})) \leq \epsilon, \quad \epsilon > 0. \tag{16}$$

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):

$$\begin{aligned} \arg\max_{q \in \Delta} \sum_t \sum_{y_t} q(y_t \mid x, f, y_{<t}) \log \frac{q_\theta(y_t \mid x, f, y_{<t})}{q_{\theta_{\mathrm{ref}}}(y_t \mid x, f, y_{<t})} \\ \text{s.t.} \sum_t \mathrm{KL}(q(y_t \mid x, f, y_{<t}) \| q_{\theta_{\mathrm{ref}}}(y_t \mid x, f, y_{<t})) \le \epsilon, \end{aligned} \tag{17}$$

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 $q_{\theta_{\mathrm{ref}}}$ (i.e., has maximal cross-entropy to $q_{\theta_{\mathrm{ref}}}$).

Proposition E.1. *The solution to Equation (17) can be expressed in closed form as*

$$q^*(y_t \mid x, f, y_{<t}) \propto \exp\big((1-\alpha)\log q_{\theta_{\mathrm{ref}}}(y_t \mid x, f, y_{<t}) + \alpha \log q_\theta(y_t \mid x, f, y_{<t})\big). \tag{18}$$

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

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

Stationarity gives, for all y_t,

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

Let $\alpha := 1/\lambda$. Then, the solution to Equation (17) can be characterized in closed form as

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

$\square$

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

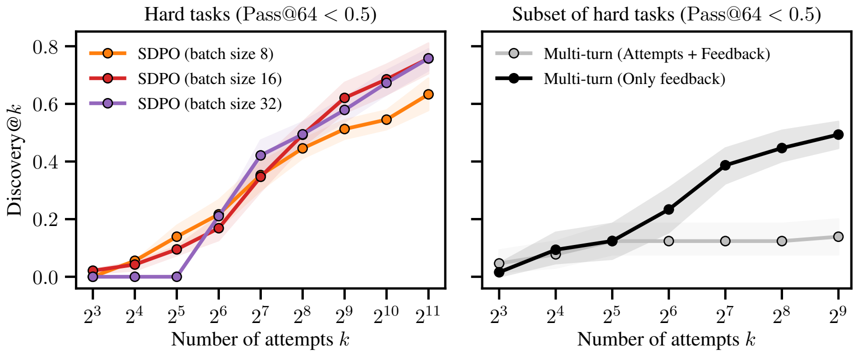


Figure 8: **Ablations self-distillation at test-time on hard tasks. Left:** Impact of SDPO batch size on pass@k curves. While smaller batch sizes (8 and 16) can lead to slightly earlier discoveries at very low generation budgets ($k < 2^6$), larger batch sizes (16, 32) result in more stable updates that significantly improve the discovery rate as the budget scales. **Right:** Comparison of multi-turn reprompting templates on a subset of hard questions. The "Only feedback" template concatenates the feedback from previous attempts using a first-in, first-out sliding window. The "Attempts + Feedback" template concatenates the full turn, also using a sliding window. Including only the feedback substantially outperforms concatenating full conversations.

F ADDITIONAL RESULTS & ABLATIONS

Baselines For best-of-k sampling under the base model, we report the standard pass@k estimate (Chen et al., 2021b) from 2944 independent rollouts. As multi-turn sampling, we sequentially reprompt the model in-context using the concatenated feedback from previous attempts. To remain within Qwen3-8B's 40k-token context limit, we employ a first-in, first-out sliding window, discarding the earliest feedback once the maximum prompt length (32k tokens) is reached. We ablate the multi-turn reprompting strategy in Figure 8, and find that retaining only past feedback while forgetting earlier attempts significantly outperforms the baseline that additionally retains past attempts. We evaluate SDPO with a batch size of 16. We ablate this choice in Figure 8 and find that overall performance differences are marginal.

Results for individual questions We show the discovery@k curves for all hard questions in Figure 9, and report the mean number of generations until the first discovery in Table 3. Further, Table 4 shows the per-question accuracy of the self-teacher at the initial training step of SDPO.

Ablations In Figure 8, we ablate the choice of batch size for SDPO and the in-context reprompting strategy for multi-turn sampling. We find that overall performance differences are marginal, yet smaller batch sizes are beneficial for improvements at low generation budgets, while larger batch sizes result in more stable updates that still learn to solve questions at later stages into the run.

Multi-turn context length The context window length for multi-turn sampling is reached after 837 (±466) steps for hard questions and after 1007 (±349) steps for very hard questions, offering a possible explanation for its diminishing gains at high generation budgets.

Further Details In the selection of hard questions, we have discarded one malformed question (Q9) where the coding environment did not correctly validate the solution due to rounding inaccuracies, which led to failures even with correct logic.

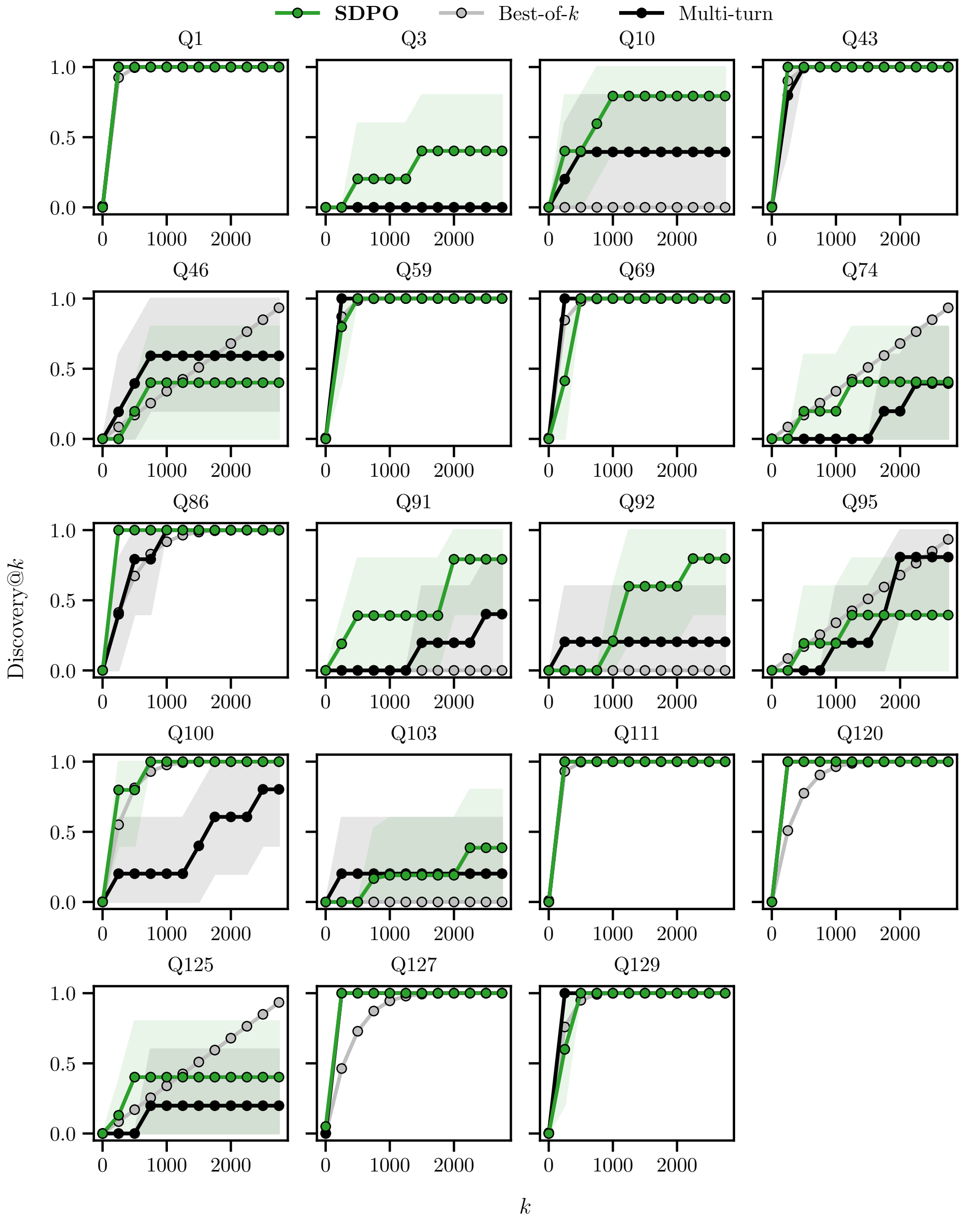


Figure 9: **Individual task results self-distillation at test-time.** Discovery@k for each of the 19 questions evaluated in Section 3. In most cases, SDPO finds a successful solution significantly earlier than both the base model and the multi-turn baseline. Notably, for one question (Q3) where the base model and the multi-turn baseline maintain a discovery@k of zero for the entire budget up to 2750 , SDPO discovers a solution after 321 attempts. Curves represent the mean and 90% confidence intervals across 5 random seeds per question.

Question	**SDPO**	**Best-of-k**	**Multi-turn**	**Speedup**
				Best-of-k $\rightarrow$ SDPO
1	104	98	**59**	0.9×
3*	**1987**	≥ 2750	≥ 2750	1.4×
10*	**938**	≥ 2750	1706	2.9×
43	111	**109**	111	1.0×
46*	1852	1466	**1315**	0.8×
59	172	123	**76**	0.7×
69	280	**134**	**134**	0.5×
74*	1948	**1466**	2405	0.8×
86	**85**	421	335	5.0×
91*	**1360**	≥ 2750	2384	2.0×
92*	**1575**	≥ 2750	2203	1.8×
95*	1948	**1466**	1794	0.8×
100	**277**	294	1596	1.1×
103*	2246	≥ 2750	**2210**	1.2×
111	85	95	**39**	1.1×
120	**24**	327	70	13.6×
125*	1795	**1466**	2320	0.8×
127	**28**	368	61	13.1×
129	168	173	**104**	1.0×
Hard tasks	**894**	1145	1141	1.3×
Very hard tasks	**1739**	2180	2121	1.2×

Table 3: Mean number of generations until first success per question for SDPO, best-of-k sampling, and the multi-turn sampling. For the mean calculation, values are truncated at the maximum budget of 2750 generations. Very hard tasks (pass@64 < 0.03) are marked with an asterisk (*). Averaged over all questions, SDPO achieves successes faster than the baselines, reaching a speedup of up to $13.6\times$ on individual questions compared to best-of-k sampling.

Question	Initial Teacher Accuracy (%)
1	0.00
3	0.00
10	0.00
43	6.25
46	0.00
59	0.00
69	3.12
74	0.00
86	0.00
91	0.00
92	0.00
95	0.00
100	0.00
103	0.00
111	0.00
120	0.00
125	0.00
127	1.23
129	0.06

Table 4: **Average accuracy of the retrospective teacher at the first step for each question.** These scores represent the percentage of successful solutions generated when the base model is reprompted with feedback in a single-turn interaction. For the majority of these hard and very hard tasks, the teacher accuracy is near or exactly 0%. Despite this, the self-distilled token-level advantages are sufficiently rich for SDPO to iteratively refine its policy and solve these questions over successive updates.

G EXPERIMENT DETAILS

G.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.

G.2 HYPERPARAMETERS

We summarize hyperparameters used for SDPO in Table 5.

Parameters	**TTT** Section 3
General	
Model	Qwen/Qwen3-8B
Thinking	False
Data	
Max. prompt length	2048
Max. response length	8192
Batching	
Question batch size	1
Mini batch size	1
Number of rollouts	16
Rollout	
Inference engine	vllm
Temperature	1.0
Validation	
Number of rollouts	-
Temperature	-
Top-p	-
SDPO loss	
Top-K distillation	20
Distillation divergence	Reverse-KL
Clip advantages	5.0
Teacher-EMA update rate	0.01
Rollout importance sampling clip	2
Training	
Optimizer	AdamW
Learning rate	1×10^{-6} (constant)
Warmup steps	0
Weight decay	0.01
Gradient Clip Norm	1.0

Table 5: Hyperparameters used for **SDPO** for each experimental setup.

H QUALITATIVE EXAMPLES

H.1 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"'
~~~

H.2 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 1: 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 2: Example of feedback “Memory Error” from our code environment in case of a wrong answer, inspired by LeetCode

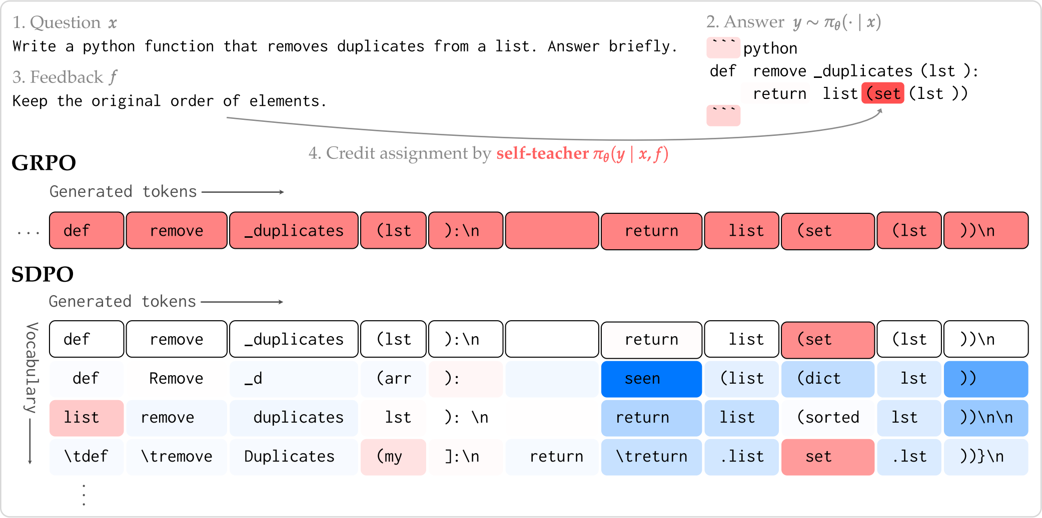
```
Runtime Error
IndexError: list index out of range
Line 28 in sortMatrix (Solution.py)

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

Listing 3: Example of feedback “Index Error” from our code environment in case of a wrong answer, inspired by LeetCode

H.3 ILLUSTRATIVE EXAMPLE

Figure 10 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

...	def	remove	_duplicates	(lst	):\n		return	list	(set	(lst	))\n

SDPO

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 10: **Dense credit assignment through self-teaching in SDPO.** The answer is generated by the 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 $\log\left(\mathbb{P}(\text{self-teacher})/\mathbb{P}(\text{student})\right)$, 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.
```