

Verified On-Policy Self-Distillation

Stanford CS224N Custom Project

Jack Li

Department of Computer Science
Stanford University
jkli@stanford.edu

Sophia Li

Department of Computer Science
Stanford University
sophyli@stanford.edu

Abstract

The ability of models to "learn on their own" is seeing increasing traction. In recent work, on-policy self distillation as a bootstrapping method has been demonstrated to surpass standard supervised fine-tuning, which suffers from "catastrophic forgetting," and reinforcement learning, which is limited by a well-defined reward. However, bootstrapping with self-distillation can run into grounding errors due to the lack of verifiable rewards. We hypothesize that the inclusion of feedback on "failure modes" in self-distillation fine-tuning (SDFT) provides a richer learning signal. In this paper, for the task of formal proof generation on mathematical benchmarks, we propose Verified On-Policy Self-Distillation (VOPSD): an SDFT method with Lean 4 compiler errors and verbalized critic on the student's generation as in-training feedback. We observe that after conditioning the teacher model on both the ground truth and verbalized feedback on errors from the student generation, the student's cumulative pass@1 performance throughout training passes surpasses baseline methods, including standard SDFT, supervised fine-tuning (SFT), and the base Lean-pretrained model. In the long-run, our method, if reproducible when scaled, can lend to broader implications in continual learning by demonstrating a scalable pathway for models to autonomously improve by interacting with formal environments.

1 Key Information to include

- Mentor: Luke Bailey
- External Collaborators (if you have any): N/A
- Sharing project: N/A
- Late Days (if you are using any):

Team member	Late days remaining
Jack Li	3
Sophia Li	3
Total pooled	6
Team size	2
Late days applied for the report	3 each

2 Introduction

Continual learning, in which the agent autonomously acquires and refines knowledge over time, has been increasingly shifting towards bootstrapping methodologies. Here, the model iteratively improves its performance by learning from its own generations. Reinforcement learning (RL) has been the state-of-the-art for bootstrapping. However, in complex reasoning tasks, rewards are often sparse or unavailable, providing a weak learning signal [1].

Alternatively, supervised fine-tuning (SFT), or learning from expert-generated data, remains a dominant paradigm. However, SFT suffers from catastrophic forgetting: because the model is trained on a static distribution of targets, gradient updates may erase prior general capabilities in favor of new task-specific labels [2] [3].

Self-distillation fine-tuning (SDFT) has emerged as a superior post-training framework. In SDFT, the "student" model learns by minimizing the KL-divergence between its own policy and the "optimal" policy generated by the same model instance, conditioned on privileged information (e.g., the solution) [4] [5]. By maintain distributional alignment, SDFT effectively mitigates catastrophic forgetting. However, standard self-distillation lacks insight into failure modes. Grounding training in execution feedback—for example, runtime logs in coding—improves performance [6]. Furthermore, including "negative" trajectories may provide in a stronger reflective signal than learning from successful examples alone [7].

We claim that incorporating verified, execution-based feedback directly into the self-distillation objective strengthens the learning signal. Our task is defined as the generation of Lean-compiled proofs for formal mathematical statements. In particular, Lean provides an ideal verification environment because the compilation is deterministic (every tactic/step is checked against a kernel), and the compiler provides "dense feedback" via specific error messages that pinpoint where a reasoning chain failed.

In this work, we present Verified On-Policy Self-Distillation (VOPSD): an SDFT framework that incorporates Lean compiler feedback in the learning signal via (1) the error messages themselves and (2) verbalized feedback on the student’s generation, given error message—the framework outperforms our baselines. We mathematically simplify the naive reverse KL divergence over the full mathematical response and shift to a token-level KL divergence to yield a dense reward signal. Our results reveal that verbalized feedback provides a richer learning signal, and we conduct theoretical analyses on token-level KL divergence loss to explain failure modes in SDFT with compiler errors.

3 Related Work

3.1 Self-Distillation

Self-distillation enables continual learning by generating dense, on-policy training signals without external rewards or large teacher models. On-policy learning mitigates distribution shifts and catastrophic forgetting inherent in standard, off-policy SFT [8]. [5] proposed On-Policy Self-Distillation (OPSD), in which the teacher model provides dense token-level supervision for the student. Additionally, [9] presented Self-Distillation Policy Optimization (SDPO), which conditions the "feedback-informed" teacher model on the trajectory’s "textual feedback"; they convert standard natural language feedback into a dense learning signal for the identification of localized failure modes.

3.2 Learning from Execution Feedback

Grounding model generations in verifiable environments enhances reasoning and iterative problem solving. [6] presents reinforcement learning with execution feedback (RLEF), in which the model leveraged multi-step execution feedback to achieve state-of-the-art results on competitive programming tasks. In addition to feedback, information on failure modes can robustify reasoning. [7] fine-tuned LLMs on "mistake-correction" data pairs—requiring the model to identify its mistake and failure reasoning—and found that mixing positive examples with correction data significantly outperformed training on a single data source type.

3.3 Formal Math

Integrating Large Language Models (LLMs) with Interactive Theorem Provers (ITPs) like Lean, foundationalized by LeanDojo [10], has established Reinforcement Learning from Proof Assistant Feedback (RLPAF) as a dominant reasoning paradigm. Contemporary models directly optimize via compiler signals: DeepSeek-Prover-V1.5 [11] applies GRPO to binary verification outcomes, Leanabell-Prover-V2 [12] iteratively corrects reasoning using multi-turn error logs, and Kimina-Prover [13] successfully scales these environment-driven reinforcement learning pipelines to 72B parameters. However, RL-based provers remain bottlenecked by the extreme sparsity of successful

proof trajectories. To overcome this data scarcity, recent frameworks employ iterative bootstrapping and self-play, notably Goedel-Prover-V2’s [14] scaffolded curriculum and verifier-guided self-correction, alongside STP’s [15] dynamic conjecturer-prover training. Our methodology directly addresses this sparse reward limitation.

4 Approach

In our approach, we first generate a solution using the student model. Then, we generate compiler information from the Lean4 Compiler which takes an input of the problem statement and a possible solution and outputs error messages, indirectly denoting pass or fail by the existence or nonexistence of errors respectively. Then, we generate feedback from a feedback model using this problem statement, reference solution, student solution, and compiler information. Then we compute the loss using the student model prompted with the problem statement and a copy of the student model also prompted with the problem statement, reference solution, feedback, and the problem statement.

For the loss, a naive definition would be to use reverse KL divergence over the responses. The following math uses problem statement x , student solution y , student model π_θ , teacher model π_ϕ , and privileged information i_p . In our case, i_p represents information like compiler error messages e_{msg} and feedback f .

$$\begin{aligned}
\nabla_\theta L_{\text{naive}}(\theta) &= \nabla_\theta \mathbb{E}_{y \sim \pi_\theta(y|x)} \left[\log \frac{\pi_\theta(y|x)}{\pi_\phi(y|x, i_p)} \right] \\
&= \int \nabla_\theta \left[\pi_\theta(y|x) \left(\log \frac{\pi_\theta(y|x)}{\pi_\phi(y|x, i_p)} \right) \right] dy \\
&= \int \pi_\theta(y|x) \left[\left(\log \frac{\pi_\theta(y|x)}{\pi_\phi(y|x, i_p)} \right) [\nabla_\theta \log \pi_\theta(y|x)] + \left[\nabla_\theta \log \frac{\pi_\theta(y|x)}{\pi_\phi(y|x, i_p)} \right] \right] dy \\
&= \int \pi_\theta(y|x) \left[\left(\log \frac{\pi_\theta(y|x)}{\pi_\phi(y|x, i_p)} \right) [\nabla_\theta \log \pi_\theta(y|x)] \right. \\
&\quad \left. + \underbrace{\int \pi_\theta(y|x) [\nabla_\theta \log \pi_\theta(y|x) - \nabla_\theta \log \pi_\phi(y|x, i_p)] dy}_{\nabla \int \pi_\theta(y|x) dy = \nabla 1 = 0} \right] dy \\
&= \mathbb{E}_{y \sim \pi_\theta(y|x)} \left[\left(\log \frac{\pi_\theta(y|x)}{\pi_\phi(y|x, i_p)} \right) [\nabla_\theta \log \pi_\theta(y|x)] \right].
\end{aligned}$$

This is intractable because it’s not possible to enumerate all possible y . However, given that π_θ and π have the same tokenizer, then we can approximate

$$\mathbb{E}_{y \sim \pi_\theta(y|x)} \approx \mathbb{E}_{y_t \sim \pi_\theta(y_t|x, y_{<t})}$$

where y_t and $y_{<t}$ represents the t -th token and the $t - 1$ previous tokens of a student response y , respectively. Hence, this is the loss we use, yielding

$$\nabla_\theta L(\theta) = \mathbb{E}_{y_t \sim \pi_\theta(y_t|x, y_{<t})} \left[\left(\log \frac{\pi_\theta(y_t|x, y_{<t})}{\pi_\phi(y_t|x, i_p, y_{<t})} \right) [\nabla_\theta \log \pi_\theta(y_t|x, y_{<t})] \right]$$

This is equivalent to doing a token level KL divergence instead of a response level KL divergence. This approximation is tractable because we can enumerate all possible y_t by $y_t = v \in V$ where v is a token in the total vocabulary V . This approximation is reasonable because repeatedly sampling the log of the probabilities a Monte-Carlo sampling method, the standard for approximating otherwise intractably complex expected values, would give a sensible approximation over all of the most probable trajectories in a student’s response.

Another added benefit of our approach is that it explicitly yields a very dense reward signal at the token level (signal on each token) as opposed to the a more sparse reward signal (signal on each response, averaging approximately 6,000 tokens long). This makes possible an analysis of the token level KL divergence for added training interpretability as shown in Section 6.2.

Additionally, for the teacher model, we used a frozen copy of the student weights instead of updating the copy as the student model updates. We do this as an attempt at added stability for training by indirectly biasing towards updates that don’t stray far from the initial model.

For our feedback model, we use Gemini 3.1 Preview Pro in an attempt to get more signal given the limited scale of our post-training.

Our approach is summarized by the following algorithm.

Algorithm 1 Self-Distillation with Feedback and Reverse KL

Require: Dataset $\mathcal{D} = \{(x, y^*)\}$ (Problem Statement x , Solution y^*), Student Model π_θ (Student Solution y), Teacher Model π_ϕ , Feedback Model π_φ (Feedback f), Learning Rate α , Epochs E , Batch Size B , Max Tokens T , Lean4 Compiler LeanCompiler (Error Messages e_{msg}).

- 1: Initialize Student π_θ {Inject LoRA adapters}
- 2: Initialize Teacher π_ϕ {Keep parameters frozen}
- 3: **for** epoch $e = 1$ **to** E **do**
- 4: **for** example $i \in \{1 + (e - 1) \cdot B, \dots, e \cdot B\}, (x, y^*)_i \in \mathcal{D}$ **do**
- 5: $y \sim \pi_\theta(\cdot \mid x)$ {Sample proof up to T tokens}
- 6: $e_{msg} \leftarrow \text{LeanCompiler}(x, y)$ {Verify formal proof}
- 7: $f \sim \pi_\varphi(\cdot \mid x, y^*, y, e_{msg})$ {Sample feedback }
- 8: $\mathcal{L} \leftarrow \sum_{t=1}^T D_{KL}(\pi_\theta(y_t \mid x, y_{<t}) \parallel \pi_\phi(y_t \mid x, y^*, e_{msg}, f, y_{<t}))$
- 9: $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}$ {Backprop + update gradient}
- 10: **end for**
- 11: **end for**
- 12: **return** $\mathcal{M}_\theta$

5 Experiments

We perform all our experiments using Goedel Prover V2 8B as the base student model [14]. See Appendix for details on student and teacher prompts.

5.1 Data

To construct our training set, we select a subsample from the NuminaMath benchmark [16]. To ensure the training data provides a meaningful learning signal, we perform 8 rollouts per problem using the base model and filter out entries for which the model achieves either 0 or 8 passes, as these represent extremes that are either out-of-distribution (too difficult to learn from) or already mastered (too easy), respectively.

- Sample Input: A formalized problem statement in Lean (NuminaMath).

```
import Mathlib
import Mathlib.theorem.number_theory_4550 (n : ℕ) (hn : 0 < n) :
(Nat.factorial (2*n)) < 2^(2*n) * Nat.factorial n^2 := by
```

5.2 Preliminary Experiments

5.2.1 Experimental details

As a pilot study to establish baseline dynamics, we trained on a subsample of 93 problems from our filtered NuminaMath set, restricted to problems where the base model passes between 2-6 out of 8 rollouts, inclusive. We trained using a batch size of 10 samples of 20 epochs with the AdamW optimizer, a learning rate of 10^{-5} , a maximum generation length of 32768 tokens, and temperature 0.7. For each generation produced by the student model, the response was verified by the Lean 4 compiler with a timeout of 120 seconds. If compilation failed, the first 10 lines of raw compiler error were append to the teacher model’s prompt alongside the problem statement and student solution. We fine-tuned the model using Low-Rank Adaptation (LoRA) via PeFT [17] [18] [19].

5.2.2 Evaluation method

We compare against three post-training baselines, in addition to the base model: supervised fine-tuning (SFT) with cross-entropy loss, "vanilla" SFT with no compiler error status or feedback, and cross-model distillation with Goedel Prover V2 32B as the teacher model. Here, we evaluate SFT

with the raw compiler error only, and with verbalized feedback—here, the verbalized feedback is generated from the base model itself (Goedel V2 8B). For the distillation baseline, we used Gemini to generate the verbalized feedback: this setup was intended to serve as our performance upper bound. We evaluated the models on randomly selected subsets of n samples ($n = 20, 50$ from the MiniF2F benchmark, reporting pass@ k ($k = 1, 4$).

5.2.3 Intermediary Results

Our preliminary evaluation results are shown in Figure 1. The high variance and limited sample size in both training and testing preclude definitive conclusions. Although we observe that SDFT with raw compiler errors consistently performs slightly better than other training methods, the differences are marginal, and the base model without any post-training methods performs comparably. Counterintuitively, the distillation baseline showed no measurable improvement. We hypothesize that a combination of the small training set size (93 samples) and train-test generalization gap (MiniF2F problems are structurally distinct and generally more difficult than NuminaMath) masked the underlying learning signals, prompting a shift in our primary experimental design.

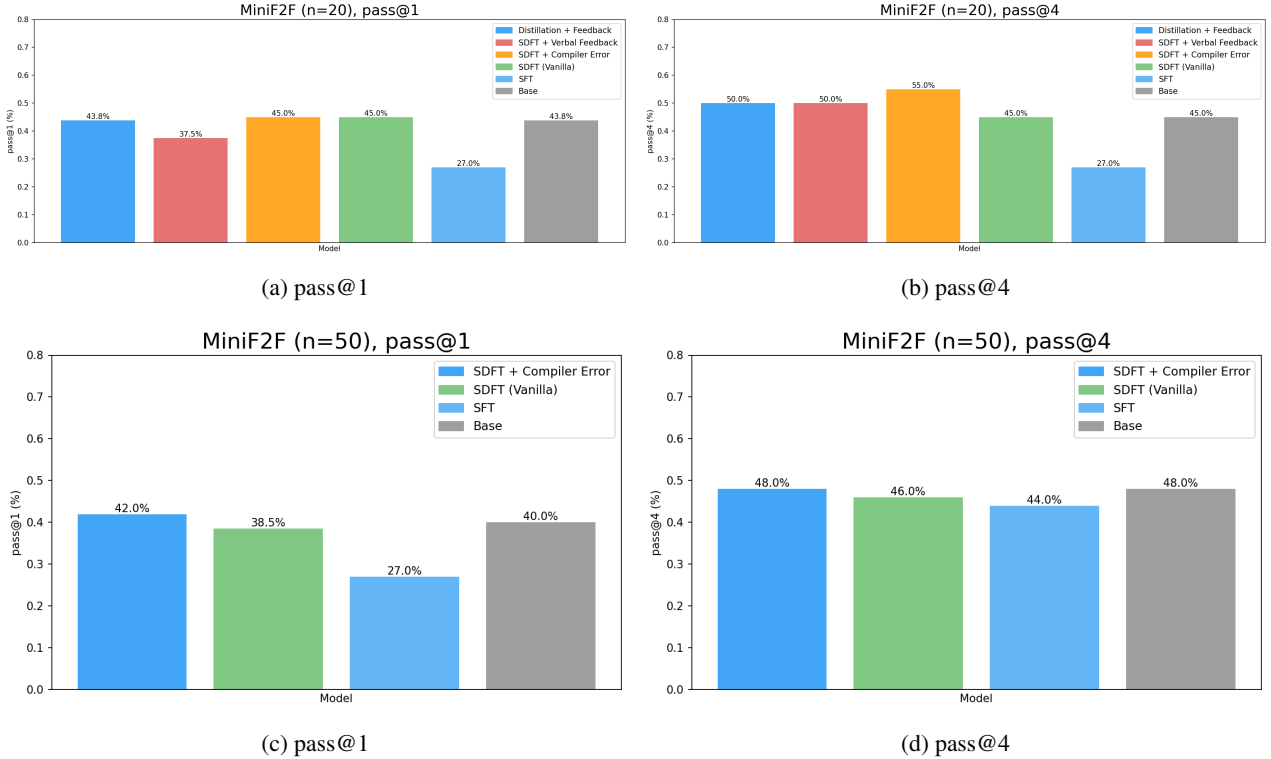


Figure 1: Performance on the MiniF2F dataset for pass@ k ($k = 1, 4$) and test sample size $n = 20, 50$.

5.3 Verified On-Policy Self-Distillation with Verbalized Feedback

5.3.1 Experimental details

Building on the pilot study’s observations, we design our primary experiment to test the explicit impact of external, verbalized feedback within SDFT. To increase data diversity and expand the model’s training distribution, we scaled our dataset to a sample of 200 NuminaMath problems where the base model passes between 1-7 out of 8 rollouts (inclusive). We maintain the AdamW Optimizer, learning rate of 10^{-5} , temperature of 0.7, batch size of 10 over 20 epochs where every single training sample is unique. We halve our maximum generation length to 16384 tokens after qualitative analysis revealed catastrophic cyclical reasoning loops at higher context limits. We verify generation via Lean 4 compilation and finetune using LoRA with PeFT. Notably, unlike the preliminary setup, we use

Gemini, instead of the base model, to verbalize natural language feedback according on failure modes to the compiler errors in SDFT.

5.3.2 Evaluation methods

We evaluated the following post-training methods against the base model: supervised fine-tuning (SFT), "vanilla" SDFT (no generation feedback), SDFT + Compiler Error (with raw compiler error messages as feedback), and SDFT + Verbalized Feedback (with compiler error status and Gemini-verbalized feedback. For each baseline, we analyze the pass@1 of the model across the training epochs (10 evaluations per epoch). At each sample, the success state (compilation pass or fail) of the student's generation was recorded. Because raw training pass@1 exhibits high variance between batches, we utilized temporal aggregation to isolate the performance trends: cumulative pass@1 (global success rate up to a given point as $\{\# \text{ success}\}/\{\text{total}\}$) and exponential moving average $A_t = \alpha \cdot k_t + (1 - \alpha) \cdot A_{t-1}$ where k_t is the raw pass score as epoch t and $\alpha = 0.09$.

5.3.3 Results

Results comparing raw pass@1, a rolling 3-epoch average, cumulative pass@1, and the time-weighted EMA are presented in Figure 2. We observe that SDFT with Gemini-verbalized feedback definitively outperforms all other baselines across the cumulative and EMA estimates, yielding a 3-5% improvement over the base model. Funnily enough, we found that conditioning the teacher model solely on raw compiler error status (SDFT + Compiler Error) aggressively degraded model performance across all training epochs. "Vanilla" self-distillation and SFT performed comparably to the base model, with vanilla SDFT showing marginal gains. The results suggest that while compiler feedback is valuable, it must be translated into natural language reasoning to provide an effective distillation gradient.

5.4 Hardware Configuration and Optimizations

All experiments were run on Modal serverless platform. Most model inference and training loops were executed on $1 \times$ and $2 \times$ NVIDIA H100 (80GB) GPUs, respectively: one to host the student and teacher models if needed for backprop in training, and one to host a vLLM instance for student generation. The Lean 4 compiler feedback loop executed with Lake, parallelized in evaluation on isolated CPU containers to prevent the 1-2 minute compilation latency from bottlenecking GPU execution.

To maximize throughput and manage limited compute, we implemented several pipeline optimizations. To mitigate the inherently slow, sequential nature of autoregressive generation, we utilized vLLM for batched prompt generation. We optimized multi-GPU orchestration by dedicating one GPU's VRAM almost entirely to inference (the primary bottleneck), while storing the teacher model and executing backpropagation for the student model on the second GPU. Finally, to prevent Out-Of-Memory (OOM) errors during the gradient update, we chunked the calculation of the KL divergence. This approach minimized the number of logits stored simultaneously, freeing up critical memory for vLLM without noticeably impacting overall training speed.

6 Analysis

To understand the mechanics of our self-distillation framework, we evaluate the impact of different feedback modalities on model performance and examine the token-level KL divergence signals to interpret how the teacher model guides the student.

6.1 The Role of Feedback Modalities

Our evaluations reveal a distinct performance hierarchy among the feedback methods: relying solely on raw compiler errors degrades performance compared to the base model, whereas augmenting compiler errors with natural language feedback yields the highest performance.

We hypothesize that raw Lean 4 compiler diagnostics, while perfectly accurate and precise, are often misaligned with the reasoning level of the model. Compiler errors frequently highlight syntactic failures (e.g., using syntax of Lean 3 instead of Lean 4) or cascading type mismatches at the end

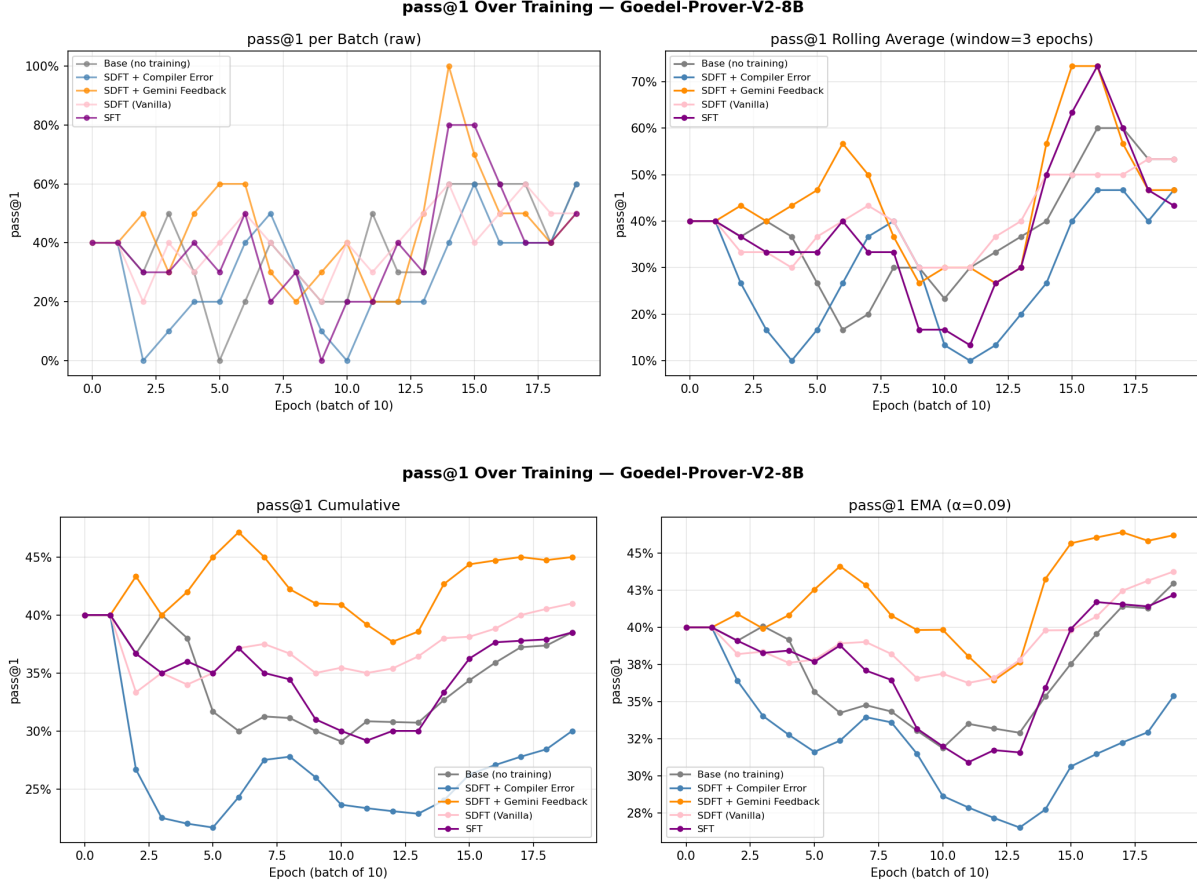


Figure 2: Continuous in-training evaluation of Goedel-Prover-V2-8B. The top charts display raw batch pass@1 and a 3-epoch rolling average. The bottom charts show the cumulative pass rate and Time-Weighted Exponential Moving Average ($\alpha = 0.09$).

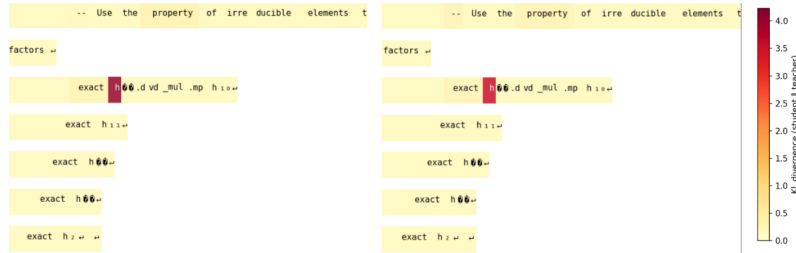


Figure 3: Denser reward signal with Feedback Model’s feedback of “*Wrong typeclass for Euclid’s Lemma: You attempted to use ‘Irreducible.dvd_mul’... the property of dividing a product belongs to the ‘Prime’ typeclass, not ‘Irreducible’.*” illustrated. (With feedback, left, and without feedback, right.)

of a logical chain, rather than the root mathematical flaw. When the teacher policy is conditioned exclusively on these dense, localized errors, it struggles to map the failure back to the overarching proof plan.

Conversely, supplying the teacher model with an LLM-generated natural language critique of the compiler error provides a massively boosts training effectiveness. For example, rather than just seeing a generic Lean type error, the feedback explicitly translates the failure into mathematical concepts. This conceptual feedback effectively bridges the gap between syntactic failure and mathematical logic, allowing the teacher policy to condition on not only a more clear *what* but even the *why*,

translating into a much richer signal for the student. We interpret these results as a form of applying chain-of-thought reasoning to token-level feedback generation as opposed to one-shot feedback.

6.2 Token-Level KL Divergence Case Studies

An additional advantage of our token-level reverse KL divergence approach is the interpretability it affords. By visualizing the density of the loss signal across a generated proof, we observe two distinct behavioral phenomena.

We observe that the learning signal (areas of high KL divergence) is disproportionately dense near the beginning of the student’s generation and dissipates significantly in the later stages. We attribute this to the architectural behavior of the Goedel prover, which is implicitly trained to operate in a reasoning “plan-then-execute” paradigm. The initial tokens predominantly comprise the informal chain-of-thought and proof planning. The teacher model correctly identifies that the ultimate success or failure of the formal code is dictated by this initial plan. Consequently, the distillation signal heavily penalizes flawed logic in the planning phase. This signal dwindles later on in the proof as the teacher is only continuing from the trajectory of the student: a “true” execution of a faulty plan *should* make the plan’s mistakes. The sparsity of the signal later in the sequence is likely compounded by context rot; as generations approach 10,000 tokens, the attention mechanism dilutes, and the teacher model loses the ability to provide high-confidence corrections on deeply nested formal syntax.

Additionally, we observe that explicitly incorrect tokens (e.g., the exact invocation of a non-existent or mathematically invalid lemma) do not always receive the highest KL divergence penalty. Instead, the teacher policy frequently assigns the highest loss to the sequence of tokens leading up to the error. We understand these results as showing that by the time the model outputs the explicit syntax error, it has already committed to an irrecoverable logical path [20]. Hence, the teacher policy recognizes the divergence at the actual inflection points of the flawed decision tree. With this framework, the distillation process is not merely punishing the final erroneous keystroke; it is actively teaching the student model to avoid the conceptual branching path that made the error inevitable.

7 Conclusion

In this work, we presented Verified On-Policy Self-Distillation (VOPSD), a framework that integrates compiler feedback into self-distillation fine-tuning (SDFT). Our experiments demonstrate a significant distinction in performance with regards to feedback form: conditioning the teacher model on raw compiler errors degrades performance, whereas translating the feedback into verbalized, natural-language explanations improves the learning signal.

In addition to our empirical results, our analysis indicates a practical advantage of self-distillation over distillation. Computing token-level KL divergence requires the student and teacher models to share the exact same tokenizer vocabulary, which limits the practical purpose of cross-model distillation. Because self-distillation guarantees tokenizer alignment, it makes dense, token-level optimization highly tractable.

7.1 Limitations and Future Work

The scale of our experiments is heavily restricted—at most 200 training entries and 20 epochs—due to our compute bottleneck. Consequently, our pass@1 evaluation metrics exhibit high variance. Furthermore, compute constraints prevented us from utilizing standard training optimizations, such as gradient update warm-ups. Future work would prioritize scaling the experiments: with increased compute, we aim to expand the training data by an order of magnitude, conduct comprehensive hyperparameter and training parameter sweeps (e.g., learning rates), train until convergence, and report pass@k ($k > 1$) on a separate test set to address evaluation variance.

Architecturally, there are several future directions we look to explore. First, we plan to add a direct pass/fail reward signal to the current token-level KL divergence loss to provide the model with a more concrete, grounding objective. Second, while our top-performing pipeline relies on Gemini to verbalize Lean compiler feedback, we look to test self-correction, where the model verbalizes its own compiler feedback. Finally, applying the framework to a non-Lean-pretrained base model (e.g., Qwen) would verify the reproducibility and broader generalizability of our findings.

Acknowledgements

The authors would like to extend their sincerest utmost gratitude to their mentor Luke Bailey, who is the goat. Additionally, the authors would like to thank their unofficial partners Alex Huang and Adam Banks.

Team contributions (Required for multi-person team)

Sophia focused on conducting preliminary and VOPSD experiments, and analysis of the results. Jack Li focused on performing mathematical analysis and hardware optimizations. Both authors contributed significantly to the codebase, efficiency of the pipeline, experiments (e.g., data filtering, training runs, model evaluations, data analyses) and the final paper.

References

- [1] Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. Metamath: Bootstrap your own mathematical questions for large language models. *arXiv preprint arXiv:2309.12284*, 2023.
- [2] Howard Chen, Noam Razin, Karthik Narasimhan, and Danqi Chen. Retaining by doing: The role of on-policy data in mitigating forgetting. *arXiv preprint arXiv:2510.18874*, 2025.
- [3] Yun Luo, Zhen Yang, Fandong Meng, Yafu Li, Jie Zhou, and Yue Zhang. An empirical study of catastrophic forgetting in large language models during continual fine-tuning. *IEEE Transactions on Audio, Speech and Language Processing*, 2025.
- [4] Idan Shenfeld, Mehul Damani, Jonas Hübötter, and Pulkit Agrawal. Self-distillation enables continual learning. *arXiv preprint arXiv:2601.19897*, 2026.
- [5] Siyan Zhao, Zhihui Xie, Mengchen Liu, Jing Huang, Guan Pang, Feiyu Chen, and Aditya Grover. On-policy self-distillation for large language models. *arXiv preprint arXiv:2601.18734*, 2026.
- [6] 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. *arXiv preprint arXiv:2410.02089*, 2024.
- [7] Shengnan An, Zexiong Li, Zhaoyu Lin, Qian Liu, Bei Chen, Yinhe Zheng, Weizhu Chen, and Nanning Zheng. Learning from mistakes makes llm better reasoner, 2023.
- [8] Idan Shenfeld, Jyothish Pari, and Pulkit Agrawal. RL’s razor: Why online reinforcement learning forgets less. *arXiv preprint arXiv:2509.04259*, 2025.
- [9] Jonas Hübötter, Frederike Lübeck, Lejs Behric, Anton Baumann, Marco Bagatella, Daniel Marta, Ido Hakimi, Idan Shenfeld, Thomas Kleine Buening, Carlos Guestrin, and Andreas Krause. Reinforcement learning via self-distillation. *arXiv preprint arXiv:2601.20802*, 2026.
- [10] Kaiyu Yang, Aidan M Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. Leandrojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems*, 2023.
- [11] Huajian Xin et al. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. *arXiv preprint arXiv:2408.08152*, 2024.
- [12] Xingguang Ji, Yahui Liu, Qi Wang, Jingyuan Zhang, Yang Yue, Rui Shi, Chenxi Sun, Fuzheng Zhang, Guorui Zhou, and Kun Gai. Leanabell-prover-v2: Verifier-integrated reasoning for formal theorem proving via reinforcement learning. *arXiv preprint arXiv:2507.08649*, 2025.
- [13] Moonshot AI and Project Numina. Kimina-prover-preview, 2025.
- [14] Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, et al. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *arXiv preprint arXiv:2508.03613*, 2025.

- [15] Kefan Dong and Tengyu Ma. STP: Self-play LLM theorem provers with iterative conjecturing and proving. In *International Conference on Machine Learning (ICML)*, 2025.
- [16] Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, Jianqiao Lu, Hugues de Saxcé, Bolton Bailey, Chendong Song, Chenjun Xiao, Dehao Zhang, Ebony Zhang, Frederick Pu, Han Zhu, Jiawei Liu, Jonas Bayer, Julien Michel, Longhui Yu, Léo Dreyfus-Schmidt, Lewis Tunstall, Luigi Pagani, Moreira Machado, Pauline Bourigault, Ran Wang, Stanislas Polu, Thibaut Barroyer, Wen-Ding Li, Yazhe Niu, Yann Fleureau, Yangyang Hu, Zhouliang Yu, Zihan Wang, Zhilin Yang, Zhengying Liu, and Jia Li. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. 2025.
- [17] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.
- [18] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, and Sayak Paul. Parameter-efficient fine-tuning of large language models, 2022.
- [19] John Schulman and Thinking Machines Lab. Lora without regret. *Thinking Machines Lab: Connectionism*, 2025. <https://thinkingmachines.ai/blog/lora/>.
- [20] Eric Bigelow, Ari Holtzman, Hidenori Tanaka, and Tomer Ullman. Forking paths in neural text generation, 2024.

A Appendix

B Prompt Templates

B.1 Student Prompt

Complete the following Lean 4 code:
`<formal_statement>`

Before producing the Lean 4 code to formally prove the given theorem, provide a detailed proof plan outlining the main proof steps and strategies. The plan should highlight key ideas, intermediate lemmas, and proof structures that will guide the construction of the final formal proof.

B.2 Feedback Evaluator Prompt

You are an expert in Lean 4 formal mathematics. A student model attempted to prove the following theorem:

Theorem: `<formal_statement>`

Correct proof: `<formal_proof>`

Student’s attempted proof: `<proof_text>`

Compiler result: `<status> <compiler_errors>`

[If compilation PASSED]: *The student’s proof compiled successfully. Provide specific feedback on: (1) any inefficient, brittle, or overly long tactics used, and (2) key techniques from the correct proof worth learning from.*

[If compilation FAILED]: *The student’s proof failed. Provide specific, concise feedback on: (1) what the student did wrong, (2) which Lean 4 tactics or approaches to avoid for this theorem, and (3) what tactics or proof strategies would likely succeed. Focus on concrete Lean 4 mistakes, not general advice.*

B.3 Teacher Prompt

[Context]

Correct proof: <formal_proof>

Previous attempt result: <status> <compiler_errors>

Feedback on what went wrong and what to try instead:
<evaluator_feedback>

[Task]

Complete the following Lean 4 code:

<formal_statement>

Before producing the Lean 4 code to formally prove the given theorem, provide a detailed proof plan... [Continued same as Student Prompt]

C Supplementary Training Dynamics

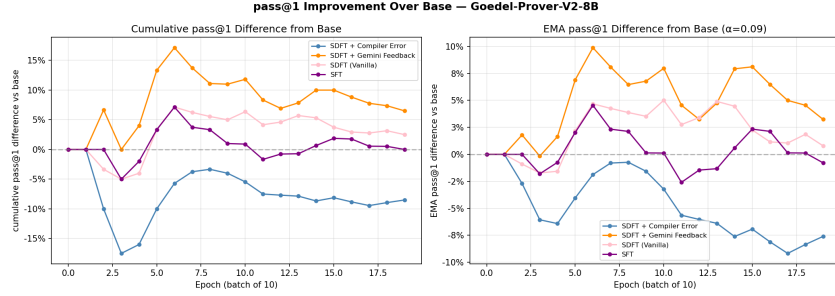


Figure 4: Cumulative and Time-Weighted Exponential Moving Average ($\alpha = 0.09$) pass@1 difference from the base model, isolating performance gains attributable to the post-training methods.

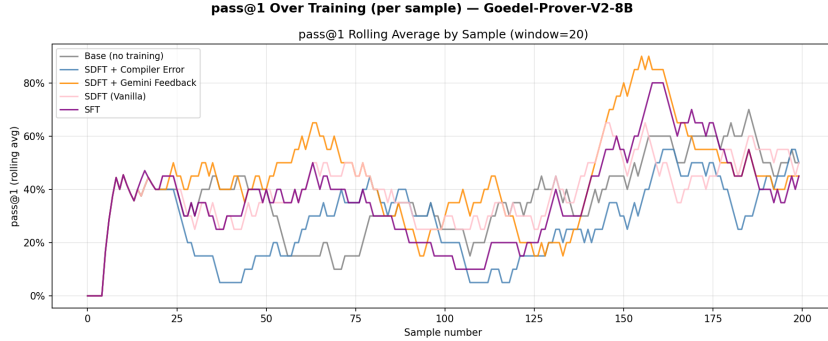


Figure 5: Rolling average (window=20) of pass@1 evaluated per training sample, illustrating the localized volatility and recovery trends of the student model.