

Scaling Test-Time Compute to Improve Formal Reasoning in Lean via Compiler Feedback

Stanford CS224N Custom Project

Adam Banks

Department of Computer Science
Stanford University
banksaj@stanford.edu

Alexander Huang

Department of Computer Science
Stanford University
alexh06@stanford.edu

Abstract

Formal theorem proving provides a rich verifiable environment for large language models. When a proof attempt fails, the Lean compiler returns structured messages describing the errors. However, most existing approaches rely primarily on the sparse success-or-failure result from the verifier, leaving the rich compiler feedback underutilized. We investigate whether this feedback can be used to improve proof generation through leveraging test-time compute by exploring two approaches. First, we implement test-time self-distillation, where compiler errors are used to construct a feedback-conditioned teacher and the student policy is updated by minimizing per-token KL divergence between the student and teacher distributions. Second, we implement multi-turn correction rounds, where the model iteratively revises failed proofs using compiler feedback appended to the prompt without updating weights, relying entirely on in-context learning. We evaluate both methods on the MiniF2F Lean theorem-proving benchmark using Qwen3.5-4B with LoRA adapters and compare them against a pass@k parallel sampling baseline. We find that multi-turn correction performs best overall, especially on the hardest problems. Self-distillation improves early discovery rates most quickly, and we hypothesize that it may surpass multi-turn correction with a larger number of iterations.¹

1 Key Information to include

- Mentor: Luke Bailey
- External Collaborators (if you have any): None
- Sharing project: None

Team member	Late days remaining
Adam Banks	2
Alex Huang	4
Total pooled	6
Team size	2
Late days applied for the report	3

2 Introduction

Recently, large language models have demonstrated strong performance on complex reasoning tasks such as mathematical problem solving and competitive programming. Much of this progress has been

¹The code is available at https://github.com/AlexH2006/test_time_self_distillation_for_lean

driven by reinforcement learning-based post-training methods, in particular Reinforcement Learning with Verifiable Rewards (RLVR), where model outputs can be automatically evaluated for correctness using external verification systems such as unit tests or compilers.

However, a key limitation of RLVR is its reliance on sparse, outcome-level rewards. In most RLVR pipelines, a generated solution only receives a binary signal indicating success (1) or failure (0). This limited feedback provides no information about why an output failed or which parts of the output were responsible for the failure. As a result, policy-gradient methods often struggle to make progress on difficult problems, since it is common for all sampled outputs to fail and therefore provide no learning signal, even when partial progress was made.

One way to address this limitation is to apply RL methods in environments that provide richer feedback signals than a binary verifier, such as formal theorem proving. Unlike many natural language environments, formalization in Lean produces structured compiler feedback when a proof attempt fails, revealing incorrect tactics, type errors, or counterexamples in the proof. These error messages contain significantly more information than a simple success-or-failure signal and may therefore provide useful guidance for improving model outputs.

In this project, we investigate whether compiler feedback can be used to improve formal theorem proving in Lean at inference-time through additional test-time computation. We explore two approaches for leveraging this feedback. First, we implement test-time self-distillation, where the model generates a proof attempt, receives compiler errors from Lean, and updates its policy by matching a feedback-conditioned teacher distribution by minimizing per-token KL divergence. Second, we implement a multi-turn correction pipeline, where the model iteratively revises failed proofs using compiler feedback appended to the prompt while keeping model weights fixed. By comparing these two approaches, we evaluate how different ways of incorporating structured verifier feedback affect theorem-proving performance.

3 Related Work

3.1 Scaling Test-time Compute

In early 2025, *DeepSeek-R1* discovered that large-scale reinforcement learning naturally elicits extended reasoning traces, allowing it to match the reasoning performance of bigger frontier models simply by generating longer chains of thought. [1] Following the test-time scaling paradigm, *s1: Simple test-time scaling* further validated the effectiveness of scaling test-time by showing that small open-source models can exceed frontier models like o1-preview by injecting "wait" tokens to artificially prolong reasoning traces. [2] In January 2026, *Learning to Discover at Test Time* pushed the boundary even further by revealing that test-time scaling is not limited to frozen models. By actively updating a model’s weights via reinforcement learning at test-time, AI systems can dynamically adapt to exceptionally difficult and out-of-distribution tasks. [3]

3.2 Theorem Proving in the Verifiable Environment of Lean

Automated theorem proving in Lean has gained recent attention because its inherently verifiable rewards and compiler feedbacks provide a robust signal for improving mathematical reasoning. *Goedel-Prover-V2: Scaling Formal Theorem Proving with Scaffolded Data Synthesis and Self-Correction*[4] introduces language models trained specifically for formal mathematics in Lean. The system combines scaffolded data synthesis, reinforcement learning, and verifier-guided self-correction to improve proof generation. In particular, the model iteratively revises failed proofs using feedback from the Lean compiler, improving performance on benchmarks such as MiniF2F and PutnamBench.

3.3 Self-distillation

Recent work has explored using self-distillation with richer feedback signals to improve reasoning performance in language models. *Reinforcement Learning via Self-Distillation* [5] introduces Self-Distillation Policy Optimization (SDPO), a framework that converts environment feedback into dense token-level learning signals. Instead of relying on an external teacher model, the method conditions the model on feedback from the environment and distills this feedback-conditioned distribution

back into the student policy. This approach improves learning efficiency compared to reinforcement learning with verifiable rewards (RLVR), which typically provides only sparse outcome-level signals.

Our work is complementary to these existing works: while Goedel-Prover-V2 focuses on post-training stronger prover models through large-scale data synthesis and RLVR, we investigate test-time self-distillation with structured and rich Lean compiler feedback to continually improve with test-time compute.

4 Approach

We propose two main approaches: *test-time self distillation* and *multi-turn self-correction* to leverage rich compiler feedback with significant test-time compute.

4.1 Baselines

As a baseline, we evaluate the model using parallel sampling and report pass@k accuracy. To ensure precise metrics, we sample multiple times and apply Monte Carlo estimation to calculate the final pass@k rate.

4.2 Test-Time Self-Distillation

4.2.1 Approach

We use a single policy model with shared parameters to generate both the student and teacher distributions. The student model is conditioned only on the problem statement (informal statement and formal theorem), while the teacher model is conditioned on the same input, augmented with a maximum of 10 compiler error messages from the most recent Lean verification attempt. As shown in Figure 1, the test-time self-distillation pipeline over one problem can be represented with the following algorithm:

Algorithm 1 Test-Time Self-Distillation (for One Problem)

- Input:** Language model π_θ ; question x in miniF2F dataset; minibatch size M ;
Lean environment to obtain feedback for attempts.
- 1: **repeat**
 - 2: Sample candidate solutions: $\{y_i\}_{i=1}^M \sim \pi_\theta(\cdot | x)$.
 - 3: Parse and compile candidate solutions to obtain feedback f_i .
 - ▷ Self-distillation:**
 - 4: Compute log-probs of self-student
 $\log \pi_\theta(y_{i,t} | x, y_{i,<t})$.
 - 5: Compute log-probs of self-teacher
 $\log \pi_\theta(y_{i,t} | x, f_i, y_{i,<t})$.
 - 6: Compute the $\mathcal{L}_{SD}(\theta)$ and update θ with gradient descent.
 - 7: **until** a successful proof is found OR a maximum number of iterations is reached.
-

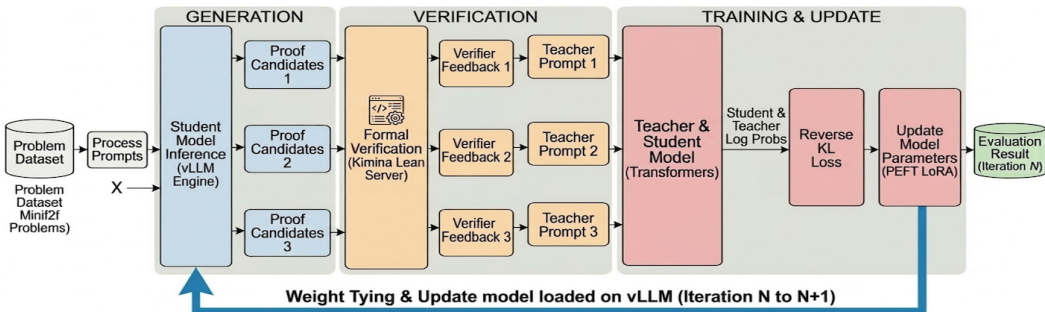


Figure 1: Test-time self-distillation on-policy training pipeline.

4.2.2 Reverse KL Divergence as Self-distillation Loss

We train the model by minimizing the sum of per-token reverse KL divergences between the student and teacher distributions over the generated Lean 4 code. The self-distillation loss is:

$$\mathcal{L}_{SD}(\theta) = \mathbb{E}_{\{y_i\}_{i=1}^M \sim \pi_\theta(\cdot|x)} \sum_t D_{KL}(\pi_\theta(y_{i,t} | x, y_{i,<t}) \parallel \pi_\theta(y_{i,t} | x, f_i, y_{i,<t})) \quad (1)$$

where the student and teacher model share parameters π_θ but are conditioned differently, and $y_{i,<t}$ denotes the tokens generated prior to step t of candidate solution y_i .

We specifically optimize the reverse KL divergence for its mode-seeking properties. While forward KL wastes model capacity by forcing the student to reproduce the teacher’s entire distribution, reverse KL concentrates the student’s probability mass strictly on high-reward, successful proof trajectories [6]. Finally, to avoid the high memory cost of computing this divergence over the full vocabulary, we approximate the calculation using a top-20 plus tail decomposition.

4.2.3 Pipeline Implementation and Hardware-aware Training Optimization

Because of the recency of the test-time self-distillation research, we cannot find off-the-shelf packages and independently implemented the test-time self-distillation on-policy training pipeline.²

1. **Generation:** vLLM generates minibatches of candidate proofs in parallel. To avoid GPU cold starts, kernel caches are persisted on a shared volume.
2. **Verification:** Kimina Lean Server parses and verifies candidate proofs in parallel off-GPU, returning structured feedback to the training node.[7]
3. **Training:** The model processes a teacher prompt (problem + compiler feedback). LoRA updates are computed using the reverse KL loss between the teacher and student distributions.
4. **Weight Tying:** The vLLM inference and QLoRA training models occupy disjoint memory on the same H100 GPU. For efficient on-policy training, merged full-precision weights from the training model are applied in-place to the inference model after each gradient step. This preserves tensor memory addresses, keeping existing CUDA graphs valid without recompilation and enabling immediate generation with the updated policy.

4.3 Multi-Turn Correction

As a second approach, we implement an iterative correction round loop. In this pipeline, the model generates an initial proof attempt conditioned on the problem statement. Until the model generates a correct proof, or for up to k correction rounds, a user message containing a maximum of 10 compiler errors from the most recent attempt along with a prefix is appended to the base prompt, and the model is asked to generate a revised proof attempt. Unlike self-distillation, this approach has the weights frozen, instead relying entirely on in-context learning to produce improvements.

5 Experiments

We evaluate three strategies leveraging test-time compute: parallel sampling, multi-turn correction using compiler feedback, and test-time self-distillation. For each problem, the model is allowed up to k attempts depending on the experiment (e.g., pass@32 or pass@128). Performance is measured using pass@ k , the probability that a correct proof is found within k attempts. All experiments are run on subsets of the MiniF2F-lean4 test split, and results are reported as the fraction of problems successfully solved within the allotted number of attempts.

5.1 Base Model Selection

We observe that models specialized in Lean via large-scale RL (e.g., Kimina-Prover-RL-1.7B and Goedel-Prover-V2-8B) produce few tactic-level errors. They typically either solve problems on the

²The detailed prompts and hyperparameters setting test-time self-distillation are included in the Appendix.

first attempt or get stuck in long reasoning loops without yielding a final proof. Furthermore, these models—particularly smaller ones like Kimina-Prover-RL-1.7B—suffer from significantly degraded in-context learning for natural language tasks.³

In contrast, Qwen3.5-4B retains excellent in-context learning alongside moderate formal reasoning in Lean. Crucially, its proof failures primarily stem from incorrect tactic applications, providing an ideal signal for improvement through self-distillation and multi-turn correction. Therefore, we chose to evaluate Qwen3.5-4B with lightweight LoRA adapters.

5.2 Pass@32 on Subset of miniF2F Subset

5.2.1 Construction of a Representative Subset

Due to the computational constraints of performing a pass@32 evaluation on the entire miniF2F-test dataset, we constructed a representative subset of 50 problems. We began by conducting a pass@8 parallel sampling across all 244 problems in the full test split. Based on these results, we subdivided the problems by their individual pass rates and applied stratified sampling to extract a 50-problem subset that mirrors the overall pass@8 success rate of the full dataset. To validate this selection, we performed 4 independent parallel sampling on both datasets to achieve the Monte Carlo estimation of their pass@k rates. As shown in Figure 2, our selected subset accurately reflects the true distribution of the full miniF2F-test dataset.⁴

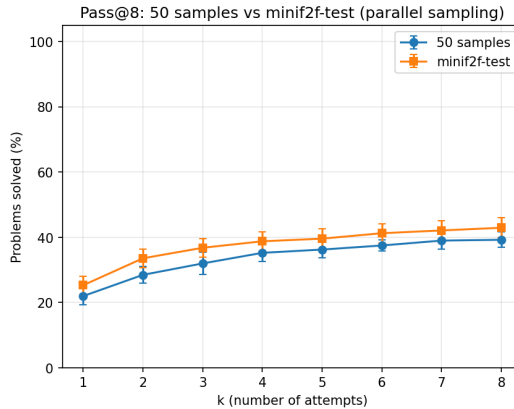


Figure 2: Comparison of pass@8 performance between the selected 50-problem subset and the full MiniF2F test set.

5.2.2 Results

We first evaluated the three pipelines on this subset of 50 problems with a maximum of $k = 32$ attempts per problem. As seen in Figure 3, self-distillation shows improvements in early attempts, though its advantage diminishes after around 10 attempts as the approaches converge in performance. Multi-turn correction ultimately achieves slightly higher final accuracy than both self-distillation and parallel sampling.

5.3 Subset of 10 at pass@128

5.3.1 Construction of a Subset of Difficult Problems

To evaluate performance on the most challenging problems, we then hand-selected the 10 theorems with the lowest pass@32 accuracy from the subset of 50 problems constructed in 5.2.1. The selected problems were rarely solved under parallel sampling, with one problem solved in 4/32 runs, three in 2/32 runs, four in 1/32 runs, and two never solved.⁵

³See Appendix for examples of failure modes in Qwen3.5-4B and Kimina-Prover-RL-1.7B.

⁴The list of the subset of problems we chose can be found in the Appendix.

⁵The list of the subset of problems we chose can be found in the Appendix.

Parallel Sampling vs Self-distillation vs Multi-turn Correction on 50 minif2f Problems

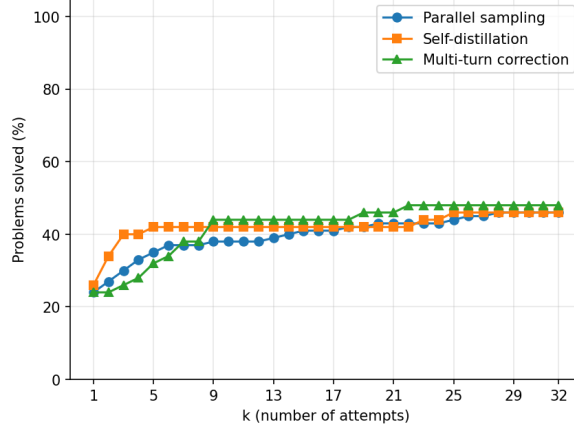


Figure 3: Pass@k curves for parallel sampling, self-distillation, and multi-turn correction on 50 MiniF2F problems with a maximum of $k = 32$ attempts.

5.3.2 Results

We evaluated the three pipelines on this new subset of 10 problems with a maximum of $k = 128$ attempts per problem. In this experiment, we also include a variant where the KL divergence is computed over the full model output rather than restricted to the Lean 4 code block. The results are shown in Figure 4. Overall, we observe that on the most difficult problems, multi-turn correction significantly outperforms the other approaches across the full range of attempts. Additionally, both self-distillation variants surpass the parallel sampling baseline after approximately 50 attempts, however the code-only self-distillation variant is the only one that improves on the baseline prior to that point.

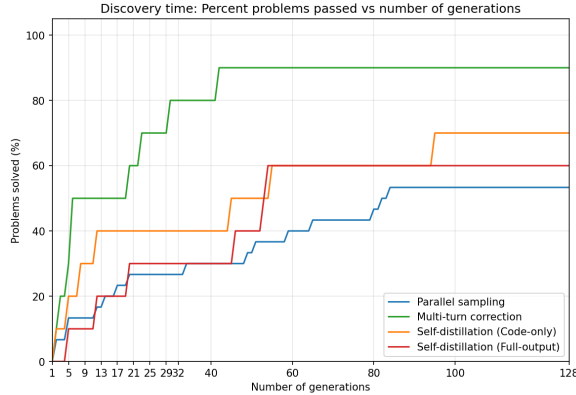


Figure 4: Pass@k curves for parallel sampling, self-distillation (code-only and full-output), and multi-turn correction on 10 MiniF2F problems with a maximum of $k = 128$ attempts.

6 Analysis

6.1 Self-distillation

6.1.1 Novel Techniques to Isolate the Training Signal

A key factor in our method’s effectiveness is concentrating the training signal to prevent distilling random noise into the reasoning model. Based on our training observations, we highlight the following novel improvements:

1. **Masking Starting Tokens:** Due to context differences between the student and teacher, the first few generated tokens often exhibit high KL divergence that is irrelevant to the actual compiler feedback. To mitigate this noise and concentrate the self-distillation signal, we exclude the first 32 tokens of the student’s output from the loss calculation.
2. **Code-Only Outperforms Full-Output:** KL divergence can be calculated over the entire response or restricted to the extracted code blocks. As shown in Figure 4, code-only self-distillation significantly outperforms the full-output approach, particularly during the early stages of generation. We hypothesize that full outputs introduce noise through natural language tokens, which exhibit high KL divergence simply due to minor formatting differences.
3. **Non-Thinking Outperforms Thinking:** Although a rigorous ablation of thinking versus non-thinking modes is left for future work, we qualitatively observe that models in non-thinking mode significantly outperform those that generate extended reasoning traces. Reasoning traces consist primarily of natural language tokens that carry little meaningful learning signal. Furthermore, we empirically note that the further a token is from the error message in the prompt, the less impact the feedback has. By excluding the reasoning trace, the non-thinking mode keeps the code output closer to the prompt’s feedback message, effectively concentrating the signal.

			use		n	/	1	3	4	2
exact	Nat	.d	vd	_iff	_mod	Eq	_zero	.m	pr	h
obtain		(	k	,	hk	)	:=	h	0	0
have	h	0	0	:	0	<	k	:=	by	

Figure 5: Self-distillation successfully identifies the wrong tactic given the compiler feedback of “unknown constant Nat.dvd_iff_modEq_zero.mpr” on mathd_numbertheory_314. Note that darker color represents higher KL divergence.

6.1.2 Self-distillation Successfully Identifies Errors

Specialized Lean models like Kimina-Prover-RL-1.7B often fail to assign meaningful KL divergence to incorrect tactics. In contrast, without specialized RL training, Qwen3.5-4B leverages strong in-context learning to effectively penalize wrong tokens with high KL divergence. Figure 5 illustrates this: the teacher model correctly flags the erroneous tactic causing the “unknown constant Nat.dvd_iff_modEq_zero.mpr” compiler error.

6.1.3 KL Sum Loss Can Cause Length Collapse

By not normalizing the sum of per-token KL divergences by proof length, our pipeline implicitly encourages concise responses. However, in noisy test-time RL environments, this unnormalized loss can lead to reward hacking, where the model converges on generating proofs too short to be successful while still achieving low loss. Figures 6 and 7 illustrate this degeneration on difficult problems like mathd-algebra-332, which succeeded only once in 128 samplings.

6.2 Multi-Turn Correction Beats Parallel Sampling

A clear advantage of multi-turn self-correction is its ability to efficiently recover from small mistakes in otherwise correct proof attempts. In many cases, the initial generation is very close to a valid proof but fails due to a minor issue, such as a type mismatch. Self-correction allows the model to revise these attempts using the compiler feedback, fixing the error while preserving the correct parts of the proof. In contrast, parallel sampling essentially treats each failed attempt as unusable and must generate a completely new proof from scratch. As a result, when attempts are performed sequentially with feedback rather than independently in parallel, self-correction can find correct solutions much more efficiently by iteratively refining close-to-correct proofs instead of repeatedly discarding them.

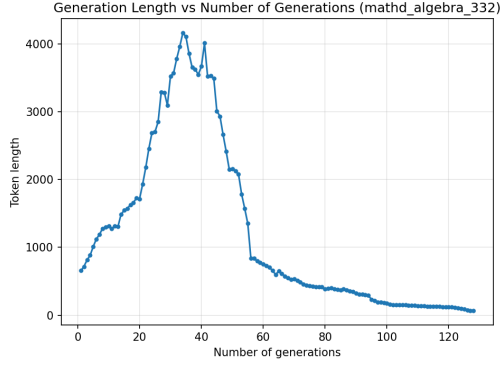


Figure 6: An example of generation length collapse when training with KL sum loss.

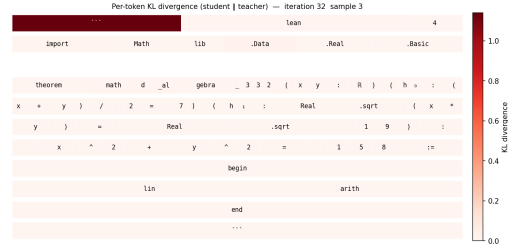


Figure 7: An example of collapsed generation when training with KL sum loss. Note that the model learns to exclude reasoning traces to minimize KL sum loss.

6.3 Comparison of Self-Distillation with Multi-Turn Correction

Across our experiments, we consistently observed that multi-turn correction outperformed self-distillation. One possible explanation is that multi-turn correction explicitly provides the model with the reason for failure through compiler feedback, allowing the model to directly revise incorrect proof attempts through its in-context learning ability. In contrast, self-distillation incorporates this feedback more indirectly by encouraging the student policy to match a feedback-conditioned teacher distribution through token-level KL divergence.⁶

6.4 Effect of Scaling

While self-distillation underperformed multi-turn correction in our experiments, distinct mechanisms of the two methods could yield vastly different performance when scaled up. Multi-turn correction leverages in-context learning with frozen weights, providing strong immediate gains on in-distribution examples. In contrast, self-distillation updates the model parameters, gradually internalizing the underlying distribution to potentially solve harder, out-of-distribution problems. However, we note the robustness of our comparative analysis was ultimately limited by available compute resources.

7 Conclusion

Our project explored whether self-distillation and multi-turn correction rounds can improve the performance of large language models for formal theorem proving at test time using feedback from the Lean compiler. We found that both approaches outperform baseline parallel sampling, with multi-turn correction achieving the strongest results. However, our experiments were limited in scale, and we hypothesize that with sufficient iterations, self-distillation may eventually surpass multi-turn correction since the latter is fundamentally limited by not being able to update the model itself.

Future work should focus on further scaling of test-time compute and evaluating performance over longer runs on more difficult problems. Because self-distillation updates parameters at test-time, additional iterations may help the model progressively internalize corrections. Furthermore, because reverse KL divergence is mathematically equivalent to the advantage function formulation, investigating RL regularization techniques to stabilize test-time training is another promising research direction. [5]

8 Team Contributions

Adam implemented and ran experiments for the multi-turn correction pipeline. Alex implemented and ran experiments for the self-distillation pipeline. All other contributions were roughly equal.

⁶One such example is included in the Appendix.

References

- [1] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.
- [2] Weijia Shi Xiang Lisa Li Li Fei-Fei Hannaneh Hajishirzi Luke Zettlemoyer Percy Liang Emmanuel Candès Tatsunori Hashimoto Niklas Muennighoff, Zitong Yang. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*, 2025.
- [3] Xinhao Li Federico Bianchi Jed McCaleb Xiaolong Wang Jan Kautz Yejin Choi James Zou Carlos Guestrin Yu Sun Mert Yuksekgonul, Daniel Kocaja. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.
- [4] Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. 2025.
- [5] Jonas Hübner, 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. 2026.
- [6] Nicolas Gontier Alexandre Lacoste Laurent Charlin Massimo Caccia Emiliano Penaloza, Dheeraj Vattikonda. Privileged information distillation for language models. *arXiv preprint arXiv:2602.04942*, 2026.
- [7] Haiming Wang Ran Wang Mantas Baksys Mert Unsal Junqi Liu Zhengying Liu Jia Li Marco Dos Santos, Hugues de Saxcé. Kimina lean server: A high-performance lean server for large-scale verification. *arXiv preprint arXiv:2504.21230*, 2025.

A Appendix

A.1 Prompts

System prompt:

You are an expert in mathematics and Lean 4 theorem proving.

User prompt:

Prove the following Lean 4 theorem without using `sorry`. At the very end of your response, output your complete, final solution as exactly one ``lean4`` code block.

```
# Informal problem:
{informal problem statement}

# Lean 4 theorem to prove:
``lean4
{file header and formal theorem statement}
``
```

Self-distillation Teacher Prompt:

Prove the following Lean 4 theorem without using `sorry`. At the very end of your response, output your complete, final solution as exactly one ``lean4`` code block.

```
# Informal problem:
{informal}
```

```
# Lean 4 theorem to prove:
```lean4
{header_and_theorem}
```
```

You MUST avoid the following compiler errors of earlier attempts:

```
{feedback}
```

Correction round user prompt:

The previous proof did not verify in Lean.

Here is the Lean/compiler feedback from your previous attempt:

```
<error>
{feedback}
</error>
```

Please revise your previous solution to fix these errors.

Requirements:

- Keep working on the same theorem.
- Use the verifier feedback above.
- At the very end of your response, output your complete corrected solution as exactly one ```lean4``` code block.
- Do not output multiple final Lean code blocks.
- Do not leave any `sorry` in the final code.

A.2 Problems

For the subset of 50 problems (5.2), the following theorems were selected from the test split:

```
mathd_algebra_44
mathd_numbertheory_1124
imo_1983_p6
mathd_algebra_419
amc12a_2020_p10
imo_1960_p2
mathd_numbertheory_427
mathd_algebra_459
mathd_numbertheory_559
mathd_algebra_24
mathd_numbertheory_447
aime_1997_p9
algebra_apbmpcneq0_aeq0anbeq0anceq0
mathd_numbertheory_765
imo_1959_p1
numbertheory_fxeq4powxp6powxp9powx_f2powmdvdf2pown
imo_1992_p1
imo_1982_p1
aime_1994_p3
mathd_numbertheory_769
mathd_numbertheory_314
mathd_numbertheory_343
mathd_numbertheory_135
mathd_algebra_388
imo_1981_p6
```

```

mathd_algebra_598
mathd_algebra_276
amc12b_2020_p6
mathd_numbertheory_435
algebra_others_exirrpowirrat
mathd_algebra_76
mathd_numbertheory_233
mathd_algebra_17
mathd_algebra_153
induction_pprime_pdvdapowpma
amc12a_2002_p6
mathd_algebra_304
amc12a_2021_p8
amc12_2000_p1
algebra_amgm_sumasqdivbgeqsuma
mathd_numbertheory_320
aime_1988_p8
numbertheory_3pow2pownm1mod2pownp3eq2pownp2
amc12b_2002_p2
mathd_numbertheory_66
amc12_2000_p20
mathd_algebra_302
amc12a_2021_p25
aime_1995_p7
mathd_algebra_441

```

For the subset of 10 problems (5.3), the following theorems were chosen:

```

mathd_algebra_478
numbertheory_4x3m7y3neq2003
aime_1983_p1
amc12_2001_p5
mathd_algebra_141
mathd_numbertheory_3
imo_1969_p2
mathd_algebra_44
mathd_algebra_209
mathd_numbertheory_1124

```

A.3 Self-Correction Example

When trying to solve `mathd_algebra_392`, Qwen generated the following proof (on its first attempt), correct except for a small casting error between $\mathbb{Z}$ and $\mathbb{N}$:

```

...
have h3 : n = 64 := by
  rw [h2]
  <.> norm_cast
...

```

The proof attempt resulted in the following compiler error:

```

tactic 'rewrite' failed, did not find instance of the pattern in the target
expression
  ↑n
  n : ℕ
  h0 : Even n
  h1 : (↑n - 2) ^2 + ↑n ^2 + (↑n + 2) ^2 = 12296
  h2 : ↑n = 64
  ⊢ n = 64

```

After receiving this feedback, the model corrected the casting issue on the next iteration by explicitly converting the integer equality to a natural number equality, while keeping the rest of the proof identical:

```
...
have h3 : n = 64 := by
  have h4 : (n : ℤ) = 64 := h2
  have h5 : (n : ℕ) = 64 := by
    norm_cast at h4 ⊢
    <;> omega
  exact h5
...
```

The second attempt successfully compiled and was counted correct.

A.4 Test-time Self-distillation Training Hyperparameter Settings

Table 1: Hyperparameters used for Test-time Self-distillation Training

Parameter	Default
<i>Generation</i>	
temperature	0.6
top_p	0.95
top_k	20
min_p	0.0
repetition_penalty	1.0
max_new_tokens	8192
use_think_mode	False
<i>Training</i>	
learning_rate	5e-6
distillation_topk	20
feedback_errors_only	True
max_feedback_errors	10
kl_mask_final_code_only	True
<i>LoRA (training)</i>	
use_lora	True
lora_r	16
lora_alpha	32

A.5 Base Model Failure Mode Examples

A.5.1 Annotated Example of a Representative Failed Proof Generated by Qwen3.5-4B

```
import Mathlib.Algebra.BigOperators.Basic
import Mathlib.Data.Real.Basic
import Mathlib.Data.Complex.Basic
import Mathlib.Data.Nat.Log
import Mathlib.Data.Complex.Exponential
import Mathlib.NumberTheory.Divisors
import Mathlib.Data.ZMod.Defs
import Mathlib.Data.ZMod.Basic
import Mathlib.Topology.Basic
import Mathlib.Data.Nat.Digits

open BigOperators
open Real
open Nat
```

```

open Topology
open Nat

theorem amc12b_2021_p3
  (x : )
  (h : 2 + 1 / (1 + 1 / (2 + 2 / (3 + x))) = 144 / 53) :
  x = 3 / 4 := by

  have h : 3 + x 0 := by
    by_contra h
    have : 3 + x = 0 := by simpa using h
    rw [h] at h -- ERROR 1: `h` is a negation `(3 + x 0)`. You
    cannot use `rw` on a negation. The AI meant `rw [this] at h`.
    simp at h -- WARNING: Because the rewrite above failed, `simp`
    and `norm_num` will not find the contradiction they are looking for.
    norm_num at h

  have h : 2 + 2 / (3 + x) 0 := by
    have : 3 + x 0 := h
    field_simp [h] at h -- NOTE: This only clears the innermost fraction, leaving
    a messy non-linear equation.
    <=> norm_num at h
    <=> linarith -- ERROR 2: `linarith` only solves linear equations.
    It completely fails here because variables are still stuck inside unresolved
    fractions.

  have h : 1 + 1 / (2 + 2 / (3 + x)) 0 := by
    have : 3 + x 0 := h
    field_simp [h, h] at h
    <=> norm_num at h
    <=> linarith -- ERROR 3: Fails for the exact same reason as above.
    `linarith` cannot process the remaining nested fractions.

  have h : 1 + 1 / (2 + 2 / (3 + x)) 0 := by -- ERROR 4: This is an exact logical
    duplicate of `h`. The AI lost track of its context.
    have : 3 + x 0 := h
    field_simp [h, h] at h
    <=> norm_num at h
    <=> linarith -- ERROR 5: Fails again due to non-linear fractions.

  have h : 3 + x 0 := h -- ERROR 6: Completely redundant. `h` is already in the
    exact same format in the local context.

  field_simp [h, h, h, h] at h
  ring_nf at h
  linarith -- ERROR 7: Because the intermediate proofs (`h`
    through `h`) failed to compile, this final block will fail. Even if they worked,
    `linarith` would likely struggle if `ring_nf` didn't perfectly linearize `h`.

```

A.5.2 An Example of Circular Reasoning in Kimina-Prover-RL-1.7B on imo_1981_p6

Wait, let's try to compute $f(4,0)$.

To compute $f(4,0)$, we can apply condition 3 with $x = 3$: $f(4,1) = f(3, f(4,0))$.

But we need $f(3,0)$.

Wait, this is getting complicated. Let's try to use Lean to compute the value.

Wait, the theorem statement is given as $f(4,y+1) = 2^{\{f(4,y)+3\}} - 3$.

So if we can prove that $f(4,y+1) = 2^{\{f(4,y)+3\}} - 3$ for all y , then for $y = 1980$, we have $f(4,1981) = 2^{\{f(4,1980)+3\}} - 3$.

But we need to compute $f(4,1981)$.

Wait, but the theorem is given as a statement to be proved. So we need to find the value of $f(4,1981)$.

Wait, but we have $f(4,y+1) = 2^{\{f(4,y)+3\}} - 3$.

So for $y = 0$, $f(4,1) = 2^{\{f(4,0)+3\}} - 3$.

But we need to compute $f(4,0)$.

Wait, this is getting complicated. Let's try to find a pattern.

Alternatively, let's try to compute $f(4,y)$.

[... Repetitive reasoning trace omitted for brevity ...]

Wait, if we can prove that $f(4,y+1) = 2^{\{f($