Enhancing LLM's Mathematical and Theorem Proving Abilities

A Project

Presented to:

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Presented By:

Naga Rohan Kumar Bayya

November, 2025

The Designated Project Committee Approves the Project Titled

Enhancing LLM's Math and Theorem Proving Abilities

By

Naga Rohan Kumar Bayya

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

November 2025

Dr. Chris Pollett	Department of Computer Science
Dr. Navrati Saxena	Department of Computer Science
Prasanna Nikhil Sathwik Vadlamani	Software Engineer Glass Imaging

Abstract

Enhancing LLM's Math and Theorem Proving Abilities

By Naga Rohan Kumar Bayya

Large language models (LLMs) are good at natural language tasks but are not up to the mark when it comes to mathematics and theorem-proving, as they rely on language patterns instead of understanding the problem and thinking through the solution. This project addresses issues such as a lack of exposure to structured datasets, difficulty with generating outputs that require multi-step reasoning, and limitations of short context. We fine-tune pre-trained LLMs on structured datasets like MATH, GSM8k, open-r1, deepseek-prover, and OpenBootstrappedTheorem. We integrate two software tools, Mathics and LEAN, and enhance reasoning through Chain-of-Thought (CoT). Additionally, we conduct the experiments using state of the art Mixture of Experts (MoE) and parameter efficient fine-tuning (PEFT) techniques such as LoRA and DoRA. The outcomes of this project are better model performance on complex math problems, and particularly on formal theorem-proving datasets, which is a comparatively understudied domain in recent LLM research. This takes a step toward developing and fine-tuning models that can handle challenging mathematical and logical domains.

Keywords: Large Language Models, LLMs, Mathics, Lean, Chain-of-Thought prompting, Deepseek, Post-tuning

1. Introduction

LLMs have transformed natural language processing (NLP) by generating human-like text and automating various language tasks. Hendrycks et al. proposed that they still have room to grow in mathematics and theorem-proving because LLMs are mostly trained on internet data, which lacks the reasoning needed for mathematical calculations and multi-step deductions. Although they generate correct-sounding answers, they fall short in terms of the factual correctness of their outputs, because they prioritize sounding "right" over being correct. This project explores different ways to improve LLM performance with the help of specialized training and data centric fine-tuning techniques.

Mathematics and logic play an important role in many fields, ranging from scientific research to business. The domain of formal theorem proving is a fairly understudied and challenging area for many LLMs. Hence, we explore various ways to improve LLM performance on this domain, like fine-tuning with structured datasets that strengthen Chain-of-Thought [10], incorporating symbolic reasoning software, and experimenting with post-training techniques like SFT and GRPO. Ultimately, our goal is to help the model go beyond just mimicking answers and to improve its mathematical capabilities.

This report shares the progress made over three semesters in improving how LLMs handle mathematical problems, especially in theorem proving. It starts with a Preliminary Work section, which explains the foundational studies from the first phase. In this stage, we laid the groundwork by fine-tuning a smaller LLM, GPT-2 [17], using mathematics datasets like MATH and GSM8k. We also incorporated symbolic tools (Mathics) and formal proof assistants (LEAN) through LangChain to see how these resources could improve the LLM’s workflow. Based on the observations and learnings in this phase, we conducted more advanced experiments in the next phase.

We also talk about the technical setup behind our evaluation process in the Infrastructure and Setup section. To make model evaluation easier, we built a server and designed an interactive web interface to compare all model versions side by side. This allowed us to clearly see how each approach performed throughout the project

The main body of the report details three experiments conducted in the second phase. Experiment 1 focuses on scaling up the fine-tuning process to a larger model architecture (Llama-3.2-3B). This compares various data supervision styles using extensive theorem-proving datasets and post-training strategies. For Experiment 2, we apply an advanced OBT-style pipeline, designed to help the model generate Lean4 proofs from natural language input more reliably. In experiment 3, we investigate the implications of two finetuning regimes and their effects on catastrophic forgetting, basing our experiments on a Mixture of Experts (MoE) model.

2. Background and Literature Review

2.1. Limits of LLMs on Mathematics and Theorem Proving

LLMs are mostly trained on large natural language datasets, so they produce answers that look convincing but lack a formal understanding of the underlying domain. This is pretty evident from their weak arithmetic and reasoning skills. Even when tested on special math benchmarks such as MATH and GSM8K, these models often produce fluent but incorrect chains of reasoning, especially when multistep derivations or symbolic transformations are required [1], [2]. These mistakes happen due to a few reasons: a mismatch between pretraining data and the downstream symbolic domains [16], weak inductive biases for formal manipulation, and they're not very good at checking their own steps.

2.2. Datasets for Mathematical Reasoning

MATH provides 12.5k competition style problems with stepwise solutions across topics (algebra, geometry, number theory, etc.) and is widely used for supervised finetuning and evaluation [1]. GSM8K targets grade school word problems requiring multi-step arithmetic and is commonly used to assess reasoning with natural-language problem statements [2]. These corpora complement each other: MATH stresses formal derivation depth and GSM8K stresses compositional reasoning in natural language.

Alongside MATH and GSM8K, this project leverages three additional corpora covering process supervision, formal proof, and symbolic logic:

1. OpenR1-Math-220k (open-r1/OpenR1-Math-220k): curated long-form math problems with stepwise rationales aligned to "Open-R1" style reasoning traces. Useful for process supervision and stabilizing multi-step derivations in arithmetic/algebra/geometry word problems.
2. DeepSeek-Prover-V1 (deepseek-ai/DeepSeek-Prover-V1): theorem/proof pairs, Lean-style statements, and tactic-level supervision suitable for theorem proving and learning proof trajectories. No natural language Chain of thought (CoT)
3. Open Bootstrapped Theorem (RickyDeSkywalker/OpenBootstrappedTheorem): Lean 4-Natural Language aligned and bootstrapped dataset for training a Lean4 LLM expert. Includes Chain of thought

2.3. Parameter-Efficient Fine-Tuning (PEFT) Techniques

LLMs usually consist of billions of tunable weights. In order to train such huge models from scratch, a lot of computational resources and time are required. PEFT techniques are clever methods for adapting such large models to downstream tasks without having to retrain the entire model from scratch, while achieving the same result. They freeze the entire model weights and focus only on a subset of them to cut down on the compute, memory, and training time. This approach makes them more feasible for tasks with limited resources. The idea comes from the fact that a pre-trained language model can be adapted for specific tasks by only changing a few parameters [3]. In this project, we used two popular PEFT techniques, Low Rank Adaptation (LoRA) and Decomposed Low Rank Adaptation (DoRA), to fine-tune GPT-2.

2.3.1 Low-Rank Adaptation (LoRA)

LoRA is a clever parameter efficient fine-tuning method that introduces a pair of low-rank decomposition matrices (denoted by A and B in Fig. 1) to update the underlying model's weights. LoRA updates only these small matrices instead of updating all the weights. This reduces the total trainable parameters to fine-tune the model on new datasets. Edward Hu, et al's work fine-tunes GPT-2 transformer on MATH and GSM8k datasets by applying LoRA on self-attention layers.

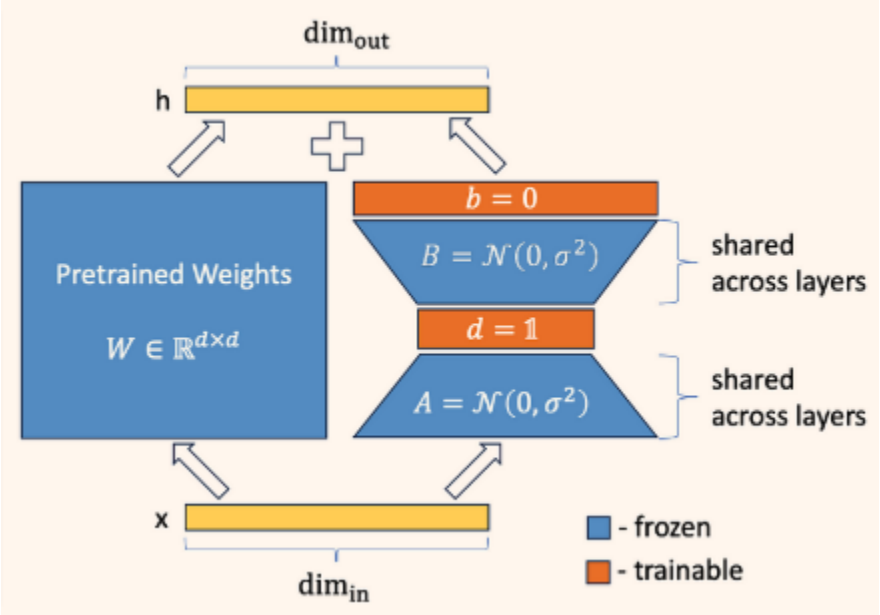


Fig 1: LoRA architecture

2.3.2 Decomposed Low-Rank Adaptation (DoRA)

Although low-rank adaptation reduces the memory footprint by freezing the underlying model, we often observe a gap when compared to full training. DoRA tries to close this gap by introducing one more matrix decomposition step by breaking down the original weight matrix W into two component matrices Magnitude (M) and Direction (D), and then applies LoRA only on the Direction matrix [4]. The intuition behind this decomposition is that Direction of the weights adapts to the new task, but the magnitude can remain the same.

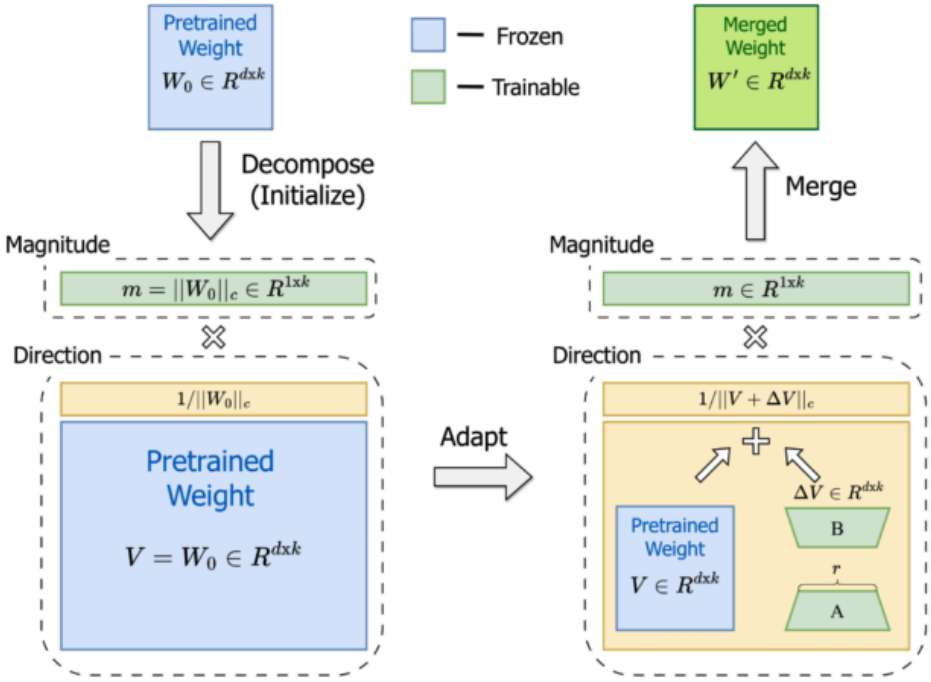


Fig 2: DoRA Architecture. Courtesy: NVIDIA's blog post

2.4. Mixture of Experts (MoE) Architecture

Instead of a single network processing all information, a Mixture of Experts (MoE) architecture consists of many smaller "experts" and a "gating network" (or router) [15] [16]. The router's job is to pick a small subset of experts (top k) to process any given token. This way, the model can grow large with lots of experts, yet maintain a small workload for each input. Because this sparse architecture activates fewer experts to generate a token, it requires fewer resources and mitigates catastrophic forgetting.

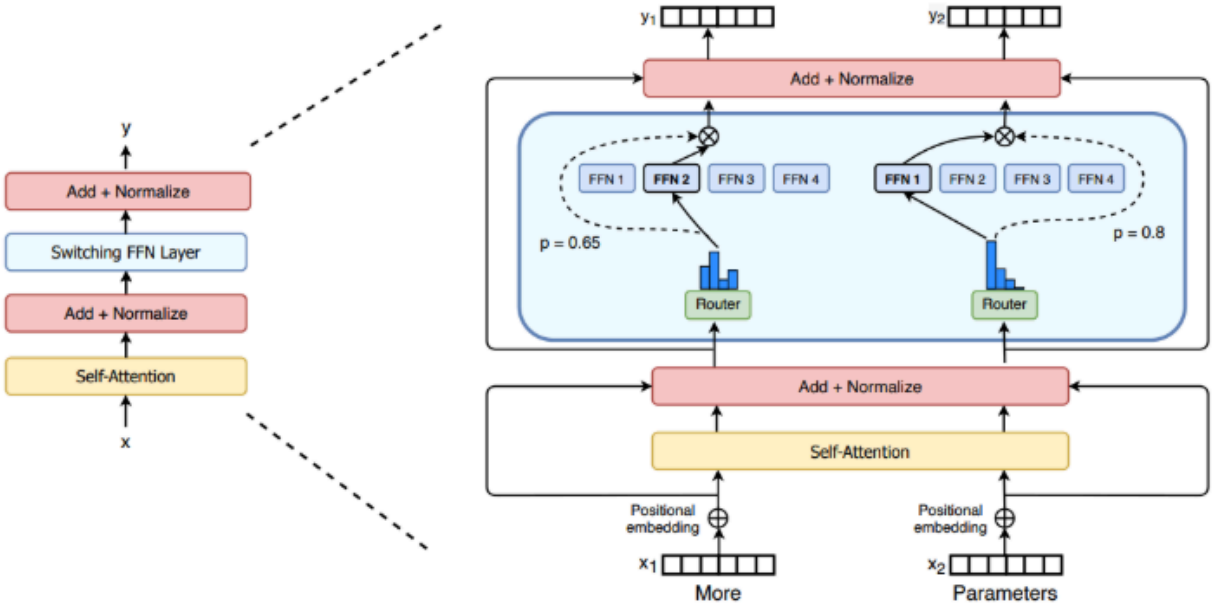


Fig 3: Mixture of Experts (MoE) architecture

2.5. Post-Training Techniques

Pre-trained LLMs are good at predicting the next word, but not at following human instructions or engaging in a conversation. We apply post-training techniques to transform these models into capable assistants as they align a model's general knowledge with its ability to apply that knowledge in a useful and reliable manner. The primary methods used in our project are illustrated in the diagram below.

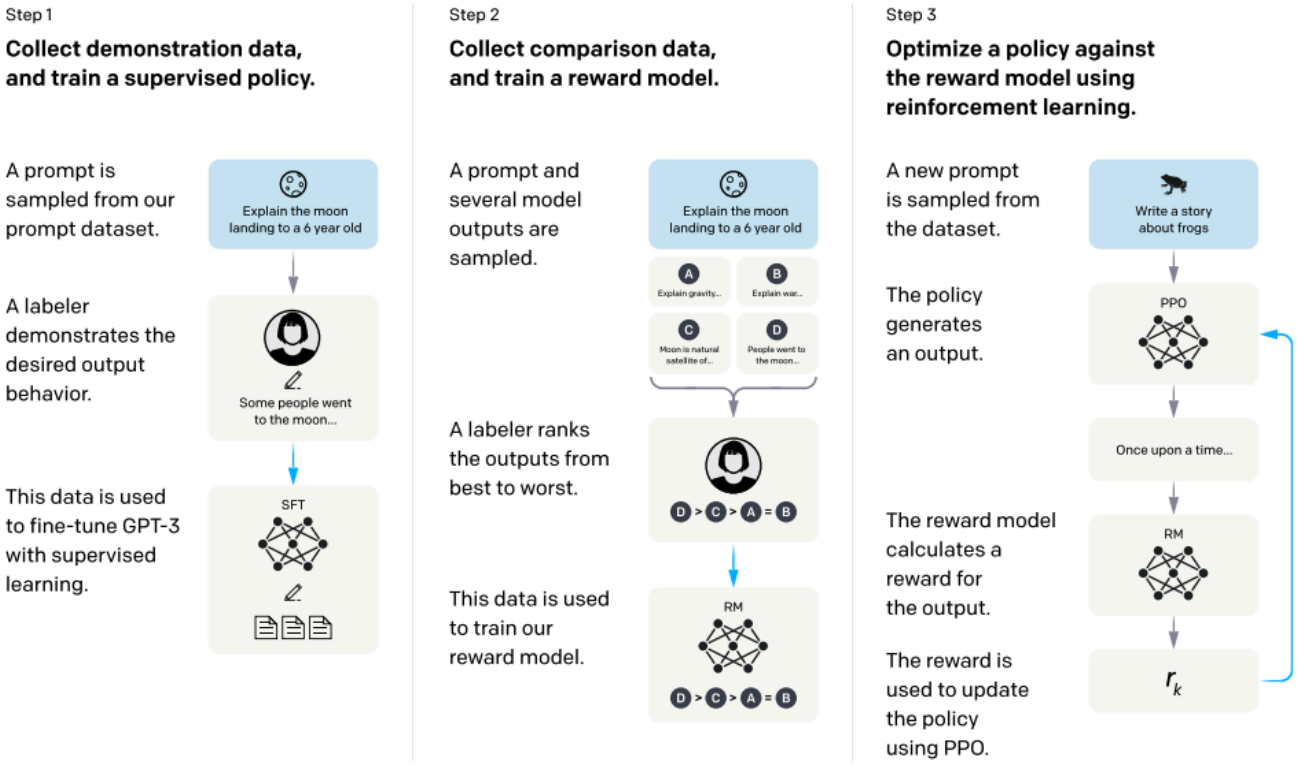


Fig 4: A diagram illustrating three steps in the Post-training phase [23]

2.5.1 Supervised Fine-Tuning (SFT)

SFT is usually the first phase in the post-training process, and its purpose is to adapt a model to behave like a helpful assistant. In this phase, the model is trained on a high-quality dataset consisting of prompt and response pairs that demonstrate the desired behavior. It teaches the model an appropriate tone for conversation and to reason thoroughly. Datasets like MATH and GSM8k are perfect for this because they include detailed solutions forcing the model to replicate the reasoning process. This way, SFT helps the model become both understandable and reliable in its responses.

2.5.2 Reinforcement Learning and Preference Alignment

After SFT, a model's behavior is further refined using feedback on its outputs. While traditional Reinforcement Learning from Human Feedback (RLHF) usually trains a reward model separately, recent advances introduced methods that offer greater stability and efficiency.

- Direct Preference Optimization (DPO): DPO is a technique that trains the model directly on a dataset of preferred and rejected responses. It uses a special loss function that directly pushes the model to favour the probability of generating the preferred response over the rejected one, instead of a separate reward function.
- Group Relative Policy Optimization (GRPO): GRPO is a newer RL technique based on rejection sampling. During training, the model generates multiple candidate responses for a given prompt and a reward model or another heuristic then scores these responses. The model is fine-tuned only on the highest-scoring or winning response. DeepSeek-R1 technical report demonstrates that this feedback loop allows the model to incrementally improve its correctness [22].

2.5.3 Model Distillation

Model distillation is a "student-teacher" technique where knowledge from a large, powerful "teacher" model is transferred to a smaller and faster "student" model. This is achieved by having the teacher generate a large, high-quality synthetic dataset, which is then used to fine-tune the student. This process is one of the main principles used in the open-r1 project as it applies the reasoning skills of the larger model to create a smaller model that mimics the same outputs. It provides a way to build small models that are still powerful enough for real-world use [22] instead of training models from scratch.

2.6. Tool-Augmented Reasoning

Language models aren't always exact or reliable with their facts, as they can't look up information instantly. They tend to make mistakes with complex numerical calculations and struggle to follow strict logic. Tool-Augmented Reasoning is another strategy for overcoming these fundamental limitations.

This technique utilizes LLM as a brain that can delegate tasks to specialized software. When the model recognizes that a task exceeds its own capabilities, it issues a query for the appropriate tool and then consumes the tool's output back into its reasoning process to generate a final answer. It combines the LLM's wider understanding of the language with the accuracy of specialized software tools. Some of the tools are listed below:

- Computer Algebra System (CAS): These are special software tools for manipulating algebraic expressions and calculus instead of just performing mathematical calculations. Mathics [8], an open-source Mathematica-like CAS, is used to offload tasks like solving complex equations or computing integrals.
- Proof Assistants: For tasks that need formal verification, proof assistants can carefully check the proof steps or tactics generated by an LLM. We used Lean4 [5], a dependent type theory proof assistant, to make sure all generated proofs are valid.
- Code Interpreters: For executing code, performing complex simulations, or handling data analysis.
- Information Retrieval Systems: To access proprietary information from external sources like search engines, databases, or APIs.

Agentic frameworks help coordinate how the LLM interacts with different tools. They make it easier to define tools, understand when the LLM wants to use a tool, and manage the process of executing those tools. In this project, we use LangChain [9] to create the agent and to communicate with external Mathics and Lean tools.

3. Preliminary Work

3.1 Objective

This first phase of the project validates three core hypotheses for enhancing LLM reasoning on a small scale: (i) that a base LLM's mathematical abilities could be improved via PEFT on specialized datasets; (ii) an LLM could be augmented with a symbolic computation engine for performing algerbic calculations and (iii) an the proofs that it generates could be verified using a proof assistant software.

3.2 Methods

Model and PEFT

We implemented DoRA, an advanced PEFT technique, to fine-tune GPT-2 on subsets of the MATH and GSM8k datasets. A learning rate of 5e-5 and a small batch size were used in the training process. The resulting fine-tuned models were published to the Hugging Face Hub as rkumar1999/gpt2-fine-tuned-math and rkumar1999/gpt2-fine-tuned-gsm8k.

Integration With External Tools

We built a unified pipeline by integrating Mathics, a symbolic computation tool, and LEAN, a formal proof assistant, to solve complex mathematical word problems. Our main goal was to show how combining these tools with LLMs can enhance their ability to tackle problems that require symbolic understanding and theorem proving.

3.3 GSM8K/MATH Fine-tuning

3.3.1 Datasets

1. MATH dataset is a compilation of 12,500 high-school-level math problems with varying difficulty levels ranging from 1 to 5 and is used to evaluate the mathematical problem-solving abilities of machine learning models. 7,500 problems are designated for training, and 5,000 for testing. It covers topics such as Prealgebra, Algebra, Number Theory, Counting and Probability, Geometry, Intermediate Algebra, and Precalculus.

Each problem is formatted in LaTeX with visual elements generated using the Asymptote vector graphics language. Solutions include step-by-step derivations and a boxed final answer, facilitating interpretability and learning using Chain of thought , as proposed by Jason Wei et al.

2. The GSM8k (Grade School Math 8K) is another high quality dataset that consists of 8500 grade school math problems that need multiple reasoning steps to be solved. It presents unique challenges as it requires the model to clearly understand problem statements and engage in step-by-step reasoning to find the right solutions [2].

3.3.2 Results

- **Finetuned Model:**

```
model_name = 'rkumar1999/gpt2-fine-tuned-math'
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
model = GPT2LMHeadModel.from_pretrained(model_name)
generate_response(model, tokenizer)
```

```
Question:
Find the value of  x  if  2x + 3 = 11 .

Generated Solution:
\left[\begin{array}{ll} \right] &= \dfrac{1}{2x+3x} & \cdot \frac{11}{11}&=\left(11x\right) \end{align*}\right\]$
Now let's look at the equation for the sum of x and y. We can see that we can solve for x by multiplying by the square root of
```

- **Base GPT-2 Model:**

```
model_name = 'gpt2'
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
model = GPT2LMHeadModel.from_pretrained(model_name)
generate_response(model, tokenizer)
```

```
Question:
Find the value of  x  if  2x + 3 = 11 .

Generated Solution:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
```

Fig 5: Fine tuning on MATH dataset

- **Finetuned Model:**

```
device = th.device("cpu")
tokenizer = GPT2Tokenizer.from_pretrained("rkumar1999/gpt2-fine-tuned-gsm8k")
model = GPT2LMHeadModel.from_pretrained("rkumar1999/gpt2-fine-tuned-gsm8k")
model.to(device)

generate_response(model, tokenizer)
```

```
Triggered calculator, answer 1575
what is the value of 21*75 equals to?
21*75 = 21*75 = 21*75 = $<<21*75=1575>>15
```

- **Base GPT-2 Model:**

```
device = th.device("cpu")
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
model = GPT2LMHeadModel.from_pretrained("gpt2")
model.to(device)

generate_response(model, tokenizer)
```

```
what is the value of 21*75 equals to?

I'm not sure. I'm not sure if it's a good idea to use a calculator, or if it's a good idea to use a calculator, or if it's a g
```

Fig 6: Fine tuning on GSM8K dataset

3.4. Integrating Mathics and LEAN with LLM

3.4.1. Mathics

Mathics is an open source computer algebra system. It is a free alternative to Mathematica and supports a variety of features such as algebraic manipulations, calculus,

plotting, and also handles symbolic computations, function definitions, and data visualization. Mathics provides a flexible way for collaborating with LLMs using a command line interface [8].

3.4.2. LEAN

LEAN provides formal verification to mathematical statements. We choose LEAN as it is an open source software and used by many mathematicians to build and verify proofs [5]. The user writes tactics to transform the goal into sub-goals, and LEAN kernel validates these tactics. The LEAN proof below shows the existence of infinite primes.

```
import Mathlib.Data.Nat.Factorial.Basic
import Mathlib.Data.Nat.Prime.Defs
import Mathlib.Order.Bounds.Basic
namespace Nat
 -- Theorem: For any `n`, there exists a prime `p` such that `n ≤ p`.
theorem exists_infinite_primes (n : ℕ) : ∃ p, n ≤ p ∧ Prime p :=
  -- Let `p` be the smallest prime factor of `n! + 1`.
  let p := minFac (Nat.factorial n + 1)

  -- `n! + 1` is not 1 (since `n! + 1 > 1`).
  have f1 : n! + 1 ≠ 1 := ne_of_gt <| succ_lt_succ <| factorial_pos _

   -- `p` is prime by definition of `minFac` (since `n! + 1 ≠ 1`).
  have pp : Prime p := minFac_prime f1

 -- We need to show `n ≤ p`. Proof by contradiction:
  have np : n ≤ p := by
     -- Assume `p < n` for contradiction.
    apply le_of_not_ge fun h =>
     -- If `p < n`, then `p` divides `n!` (from definition of factorial).
    have h1 : p ∣ Nat.factorial n := dvd_factorial (minFac_pos _) h
     -- `p` divides `n! + 1` (by definition of `minFac`).
     -- If `p` divides `n!` and `n! + 1`, it must divide their difference 1
    have h2 : p ∣ 1 := (Nat.dvd_add_iff_right h1).2 (minFac_dvd _)
     -- But a prime `p` cannot divide `1`. This is a contradiction.
    exact pp.not_dvd_one h2
-- combine p and properties (n ≤ p, Prime p), to complete the exists goal.
  (p, np, pp)

end Nat
```

Proof to demonstrate that for any natural number n, there exists a prime $p \geq n$ by considering the smallest prime factor of $n!+1$. It uses the fact that $n!+1$ cannot be divisible by any prime $\leq n$.

3.4.3 Integration using Langchain

1. Mathics was invoked through a LangChain agent using its MathicsSession class to handle symbolic computations. For example, queries like "Find the integral of $\sin^2(x)$ from 0 to $\pi/2$" were passed to Mathics, which returned correct results.

2. We created a custom Proof Assistant tool to work with LEAN. It uses a pre-trained sequence-to-sequence model [6] that can generate a variety of tactics based on a proof state. Some key components are:
    - Proof Initialization: We used LEAN's read eval print loop (REPL) to define initial an initial proof state and placeholders for any unresolved steps
    - Tactic Generation: We used a sequence-to-sequence model to generate several potential tactics for each proof state, which are iterated over to recursively search valid paths
    - Proof Validation: We used LEAN to provide feedback on unresolved goals and verify each step to ensure correctness.

```
Algorithm for PROOFSEARCH(thm)

Input: thm: Lean4 theorem declaration with a 'sorry' goal
Output: A sequence of tactics that discharges all goals, or FAIL

Procedure PROOFSEARCH(thm):
    P ← LaunchREPL()
    DefineWithSorry(P, import Mathlib; open Real; open Nat; open
BigOperators)
    (s₀, g₀) ← DefineWithSorry(P, thm, env = 0)
    if (s₀, g₀) is None then
        return FAIL
    return DFS(g₀, s₀, 0, [])  // Initial call to DFS (g₀: initial goals, s₀:
initial score, 0: initial depth, []: initial proof)

Function DFS(g, s, d, π):
    if d ≥ DEPTH_LIMIT then
        return None  // Exceeded depth limit
    T ← GenerateTactics(g, k = NUM_CANDIDATES)
    for each t in T do
        (s', g', M) ← ApplyTactic(P, s, t)

        if (s', g', M) is None then
            // syntax/type error, etc.
            continue  // Try the next tactic
        end if

        if |G'| = 0 then
            return π appended with t // Success: goals discharged
        end if

        if s' = s + 1 and |G'| > 0 then
            g' ← G'[0]  // Focus on the first remaining goal
            // Recursively search deeper
            π' ← DFS(g', s', d + 1, π appended with t)
            if π' ≠ None then
                return π'  // Proof found in a deeper search
            end if
        end if
    end for
    return None  // No successful path found at this depth
```

```python
# Define the tools
def perform_math(expression: str) -> str:
    """Use this tool when you need to calculate a complex mathematical expression."""
    from mathics.session import MathicsSession
    session = MathicsSession(add_builtin=True, catch_interrupt=True)
    result = session.evaluate(expression).to_python()
    return str(result)

def perform_lean(expression: str) -> str:
    """Use this tool when you want to find the proof a theorem proposition in LEAN. The format of the input should be a string like: 'th
    Just give the theorm followed by sorry. Do not give the entire proof and do not use special symbols. Use the returned tactics to for
    """
    from lean_tool import ProofAssistant
    lean_assistant = ProofAssistant()
    result = lean_assistant.proof_search(expression)
    return ", ".join(result or [])

math_tool = Tool(
    name="MathTool",
    func=perform_math,
    description="Computes complex mathematical expressions using Mathics, a computer algebra system."
)

lean_tool = Tool(
    name="LeanTool",
    func=perform_lean,
    description="""Use this tool when you want to find the proof a theorem proposition in LEAN. The format of the input should be a stri
    Just give the theorm followed by sorry. Do not give the entire proof and do not use special symbols. Use the returned tactics to for
    """
)

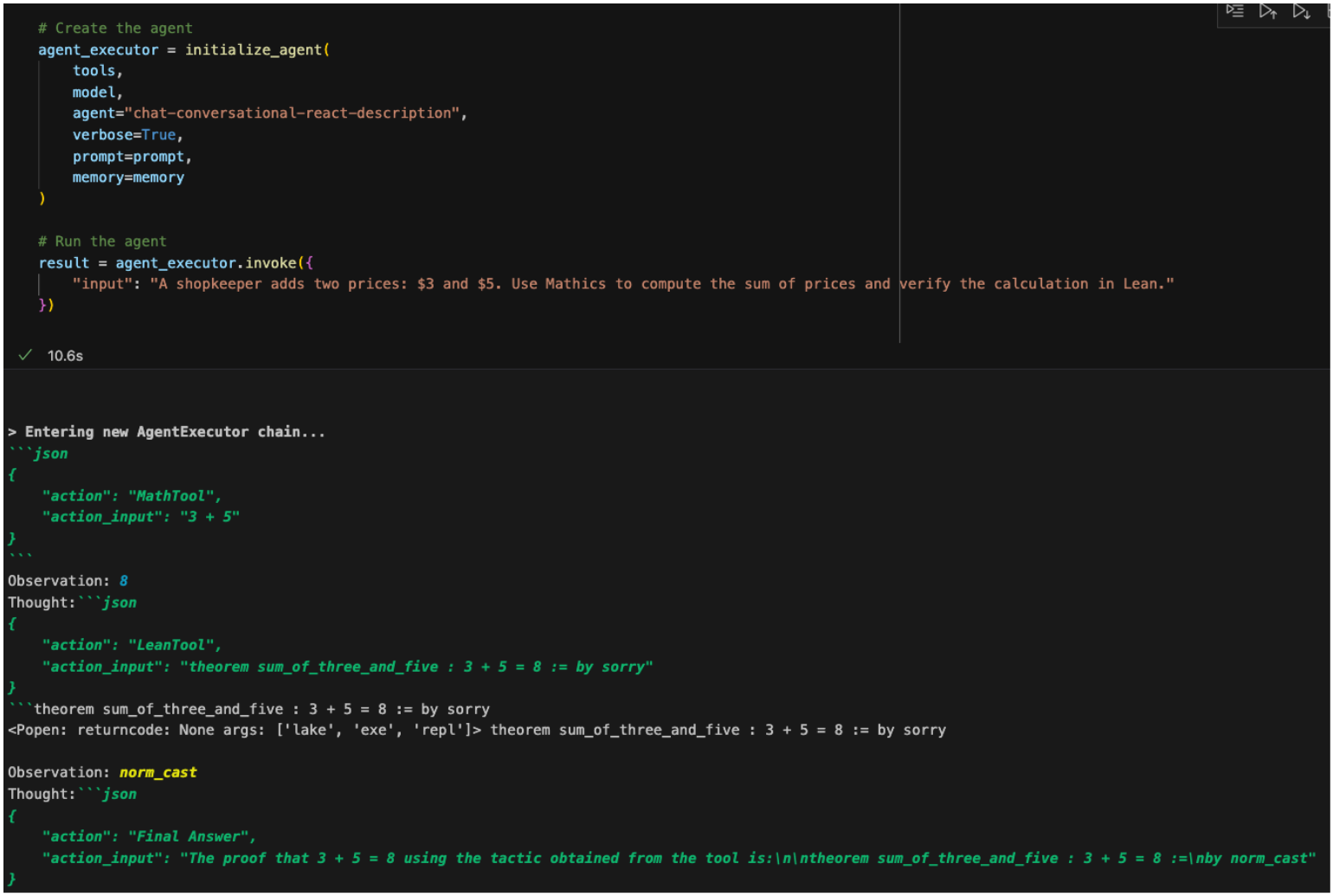
tools = [math_tool, lean_tool]
```

✓ 0.0s

Fig 9: Using Tool class of langchain to invoke Mathics and LEAN

3.4.4. Results

The pipeline integrates an LLM with both the Mathics and LEAN tools, enabling it to accomplish tasks that would be too complex for the model to solve on its own. The LLM figures out that it has to call a specific tool by looking into the tool's description expressed as docstrings. The image below shows a pipeline invocation where we simultaneously added two numbers using "perform_math" and verified the correctness of the result using the "perform_lean" Proof Search tool.



```
# Create the agent
agent_executor = initialize_agent(
    tools,
    model,
    agent="chat-conversational-react-description",
    verbose=True,
    prompt=prompt,
    memory=memory
)

# Run the agent
result = agent_executor.invoke({
    "input": "A shopkeeper adds two prices: $3 and $5. Use Mathics to compute the sum of prices and verify the calculation in Lean."
})
```

✓ 10.6s

```
> Entering new AgentExecutor chain...
```json
{
    "action": "MathTool",
    "action_input": "3 + 5"
}
```
Observation: 8
Thought:```json
{
    "action": "LeanTool",
    "action_input": "theorem sum_of_three_and_five : 3 + 5 = 8 := by sorry"
}
```theorem sum_of_three_and_five : 3 + 5 = 8 := by sorry
<Popen: returncode: None args: ['lake', 'exe', 'repl']> theorem sum_of_three_and_five : 3 + 5 = 8 := by sorry

Observation: norm_cast
Thought:```json
{
    "action": "Final Answer",
    "action_input": "The proof that 3 + 5 = 8 using the tactic obtained from the tool is:\n\ntheorem sum_of_three_and_five : 3 + 5 = 8 :=\nby norm_cast"
}
```

Fig 10: Invoking the Langchain pipeline to solve multiple tasks at once

The experiments in the initial phase were promising, but they pointed out the limitations of a recursive approach to solving a proof, which was both time-consuming and computationally infeasible. To better achieve our main goal of improving formal theorem proving, we scaled up to larger LLM architectures and adopted more advanced training techniques.
```

4. Infrastructure and Setup

In this whole project, we produced multiple fine-tuned checkpoints and adapters. To easily evaluate and compare them, we built an interactive user interface and chat application. This section describes the underlying technical framework to help users compare model evaluations. It uses the high-performance Text Generation Inference (TGI) for GPU-accelerated deployment.

4.1. Text Generation Inference (TGI)

TGI is an open-source project created by Hugging Face to deploy large language models smoothly in production-grade settings. It is optimized for GPU performance [15] and offers several benefits for our project.

We achieved GPU acceleration through CUDA kernels and smart memory management to maximize throughput on NVIDIA GPUs. For robust evaluation, we set up Model Serving by configuring every fine-tuned variant (base, SFT, GRPO, OBT) as its own isolated Docker container running on a different port, allowing us to query them all concurrently without interference.

TGI automatically manages Dynamic Batching, which combines multiple inference requests to keep the GPU busy while maintaining high throughput. Also, the Docker container handles all dependency management (like installing Flash-Attention and PyTorch), simplifying our setup. All TGI instances expose a REST API with a standard /generate endpoint. We deployed these TGI containers on cloud GPU instances (Lightning.ai with H200 GPUs), with each model hosted on separate ports starting from 8080.

4.2. Chat Application

We built a chat application and inference engine to evaluate our different finetuned models created during this project.

The frontend is a React.js based application for a user-friendly interface. It includes model selection dropdowns, a chat box for submitting prompts, and a split-pane view showing responses from two models for side-by-side comparison. Users can also adjust generation settings like max tokens and see live updates on the status of the TGI endpoints.

The backend uses FastAPI server as a middleware whose role is to track all TGI endpoints and provide RESTful APIs to direct incoming requests to the correct port. The inference layer manages model inference using Docker containers deployed on cloud instances with access to GPUs. Each container loads a specific model checkpoint along with all necessary parameters, such as port number and temperature, and operates independently.

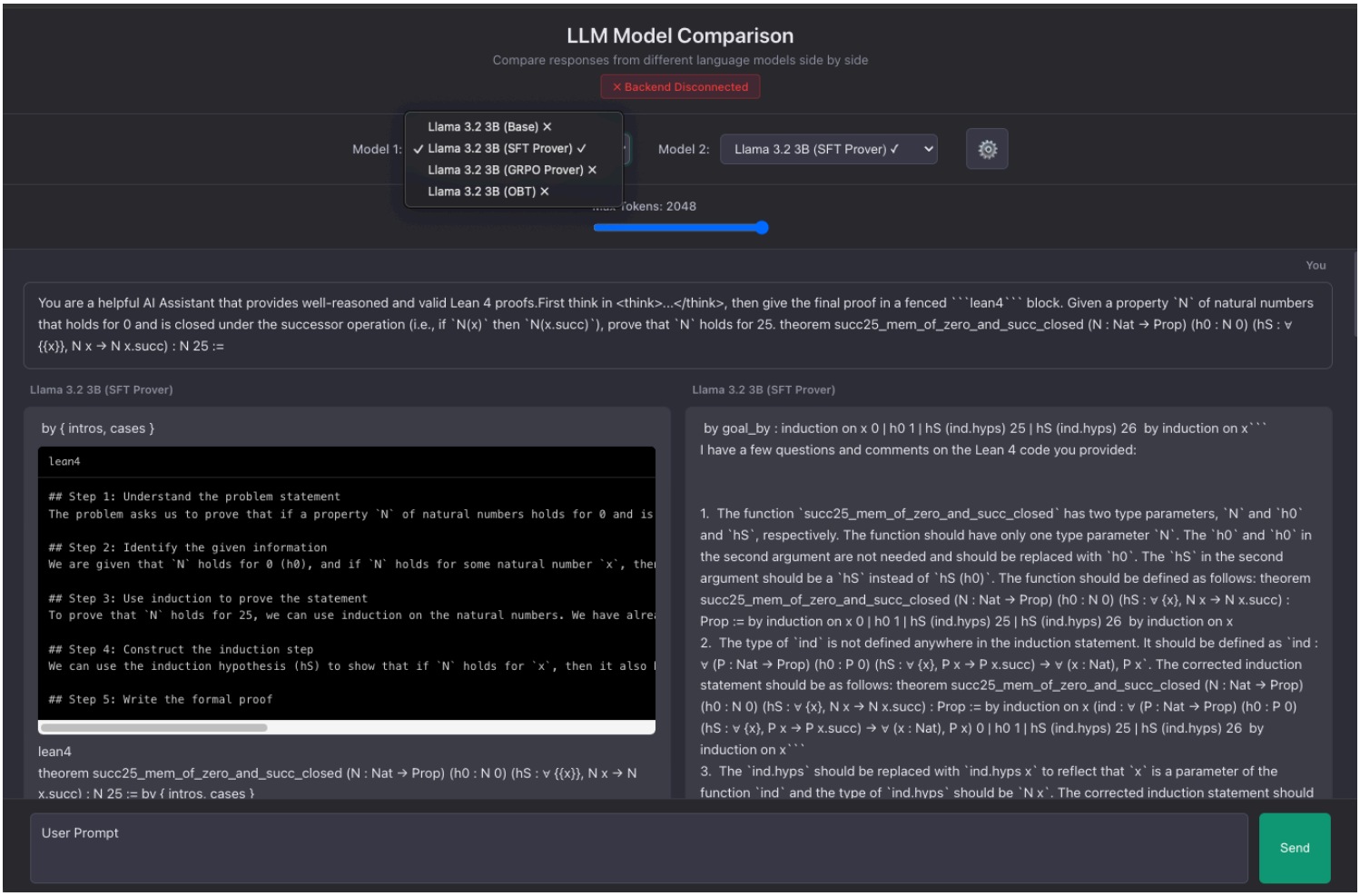


Fig 11: Chat application and inference engine

4.2.1 Usage in Experiments

This infrastructure played a key role in conducting the experiments 1-3 smoothly. In Experiment 1, we used our interface to put the base Llama-3.2-3B model up against fine-tuned versions using SFT and GRPO, focusing on how well they could generate Lean4 proofs. The side-by-side comparison made it clear how each model differed in their proof structure, choice of tactics, and how they handled tricky edge cases.

For Experiment 2, we compared OBT-style fine-tuned model versus the base version on theorem proving tasks. The interface displayed Lean4 code with syntax highlighting along with its thought process (or CoT), making it easy for users to check the accuracy and style of each proof in an interactive fashion.

Experiment 3 showed the effects of different training regimes (mixing of tasks vs sequential training) while fine-tuning a Mixture-of-Experts (MoE) model by comparing the outputs side-by-side on the interface. This experiment showed some common mistakes to avoid catastrophic forgetting when we are fine-tuning on multiple tasks.

This infrastructure provided a user-friendly framework for model evaluation throughout the project. It separates concerns across three architectural layers: presentation, orchestration, and inference. We were able to compare the outputs of different models on identical prompts and also experiment with hyperparameters like the number of tokens, temperature, etc.

5. Experiment-1: Finetuning Llama-3.2-3B using open-r1

5.1 Objective

This experiment explored the effects of different data supervision styles and post-training techniques on an LLM's ability to generate formal proofs. We fine-tuned a mid-sized model, meta-llama/Llama-3.2-3B-Instruct, across two distinct data regimes using SFT+GRPO pipeline from the Hugging Face open-r1 handbook [24]:

1. Formal-Only Supervision: Training on Lean theorem statements and their corresponding formal proofs without natural language chain of thoughts.
2. Text-Guided Supervision: Training on Lean statements and proofs, along with their natural language explanations and step-by-step reasoning behind choosing each tactic.

The hypothesis is that text-guided supervision will substantially improve the model's correctness, logical stability, and ability to generate valid Lean4 proofs compared to training on just the formal code.

5.2 Data

Two datasets were used to create the different supervision regimes:

1. deepseek-ai/DeepSeek-Prover-V1: This dataset constitutes pairs of Lean theorem statements and their raw proof code. It provides formal-only supervision with little to no natural language rationale.
2. Cartinoe5930/DeepSeek-Prover-V2-generation: This dataset contains synthetic mathematical problems accompanied by step-by-step natural language solutions and their corresponding Lean proofs.

5.3 Method

We conducted the experiments using sft.py and grpo.py scripts from the Hugging Face Alignment Handbook and open-r1 repositories [24]. A custom chat template based on the Llama

3 format was used to structure the prompts with special tokens (<|start_header_id|>, etc.), ensuring the model received a consistent system prompt along with conversational history.

5.3.1 Prompt Format

We used Llama 3 chat template to help the model follow conversations more effectively. This template structures the input as a series of messages, each marked with special control tokens that indicate turns and when a message ends. It has special token to guide the conversation flow. The prompt starts with a <|begin_of_text|> to mark the start of a conversation, followed by a series of messages where cach role—system, user, or assistant—is wrapped between <|start_header_id|> and <|end_header_id|>. <|eot_id|> marks the end of a message. The add_generation_prompt flag ensures that the final <|start_header_id|> assistant <|end_header_id|> tokens are appended, prompting the model to begin its turn. This token-based structure helps instruction-tuned models to produce clear outputs by distinguishing between instruction, user questions, and the previous context.

```
{%- if messages and messages[0]['role'] == 'system' -%}
<|start_header_id|>system<|end_header_id|>
{{ messages[0]['content'] }}
<|eot_id|>
{%- else -%}
<|start_header_id|>system<|end_header_id|>
{{ default_system }}
<|eot_id|>
{%- endif -%}

{%- for m in messages -%}
  {%- if not (loop.first and m.role == 'system') -%}
<|start_header_id|>{{ m.role }}<|end_header_id|>
{{ m.content }}
<|eot_id|>
  {%- endif -%}
{%- endfor -%}

{# When generation is requested, open the assistant header #}
{%- if add_generation_prompt -%}
<|start_header_id|>assistant<|end_header_id|>
{%- endif -%}
```

5.3.2 GRPO Custom Reward Function

We developed a custom GRPO reward function that evaluates each generated proof against its corresponding ground-truth solution. It assigns a perfect score of 1.0 for an exact match and a partial score based on the Jaccard similarity between tokens. It also penalizes incomplete proofs if they contain keywords like "sorry".

```
Algorithm for Lean Proof Reward Function

Input: Model completion C, Gold proof G
Output: Reward score s ∈ [-1, 1]

1. Extraction and Normalization:
    C_code ← ExtractLastLeanBlock(C)
    G_code ← ExtractLastLeanBlock(G) (if applicable, else use G directly)
    C_norm ← Normalize(C_code) // Remove comments and collapse whitespace
    G_norm ← Normalize(G_code)

2. Scoring Logic:
    if C_norm is empty then
        return -0.8 // Penalize for no proof generation
    end if

    if C_norm = G_norm then
        return 1.0 // Perfect score for exact match
    end if

    --- Partial Credit Calculation

    s_sim ← 0.7 × JaccardSimilarity(C_norm, G_norm) // Score based on
token overlap
    s_format ← 0.1 if C has a Lean code block, else 0.0
    s_penalty ← -0.5 if C_norm contains 'sorry', 'admit', or 'by skip',
else 0.0

    s ← s_sim + s_format + s_penalty
    return max(-1.0, min(1.0, s)) // Clip final score to the range [-1, 1]
```

5.3.3 Training Hyperparameters

The following hyperparameters were used to fine-tune the base llama model:

1. Epochs = 2, LR = 3e-4 with cosine schedule, weight decay = 0.0.
2. Per-device batch size = 2, gradient accumulation steps = 8 (effective batch = 16).
3. Warmup ratio = 0.03; log steps = 10; save each epoch (or ≥ 5k steps).
4. Seeding and TF32/bf16 enabled for throughput; Flash-Attention v2 for attention kernels.
5. QLoRA: 4-bit NF4 quantization (bitsandbytes), bf16 compute, LoRA adapters (rank = 16, α = 32, dropout = 0.05) on causal-LM projections.
6. Context: model_max_length = 8192
7. Optimizations: Flash Attention v2 kernel.
8. Setup: H200 GPU by lightning.ai

5.4 Results

The fine-tuning process produced three distinct model artifacts, which are publicly available on the Hugging Face Hub.

1. SFT-1 (Non natural language guided): rkumar1999/Llama3.2-3B-Prover-openr1-SFT
2. SFT-2 (Natural language guided data): rkumar1999/Llama3.2-3B-Prover-openr1-distill-SFT
3. GRPO (RL finetuned after SFT-2): rkumar1999/Llama3.2-3B-Prover-openr1-distill-GRPO

5.4.1 Evaluation Rubric

To measure performance, a rubric was developed to score each model's output on a scale of 100. It focuses on proof validity, efficiency, explanation quality, and structural integrity.

Main Category	Sub-Criterion and Metric	Max Points	Scoring Breakdown
Proof Score (60)	Validation Score (Pass / Fail)	40	Measures if the proof is valid and correct. • 40 pts: Pass (100% syntactically valid and logically correct). • 20 pts: Partial Credit (Correct strategy but contains a minor, fixable error). • 5 pts: Fail (Syntactically valid but logically wrong). • 0 pts: Fail (Syntactically invalid, incomplete, or non-existent).

	Efficiency Score Δ = |Generated Lines - Ideal Lines| Max (0, 20-Δ)	20	Measures the deviation of proof length from ideal proof length
Explanation Score (25)	Component Checklist (Checklist)	15	Measures if the explanation is complete and accurate. 5 pts: Goal Identification (Accurately states the theorem). 5 pts: Strategy Outline (Describes a valid high-level plan). 5 pts: Logical Soundness (Free of major factual errors).
	Quality Checklist (Checklist)	10	Measures if the explanation is relevant and well-formed. • 5 pts: Proof Alignment (The explained strategy matches the generated code). • 5 pts: Conciseness (Zero "thought loops" or significant verbal repetition).
Format and Integrity Score (15)	Structural Checklist (Checklist)	10	Measures if the output follows the required format. • 5 pts: Explanation Present (An explanation section exists). • 5 pts: Code Block Present (A fenced Lean 4 code block exists).
	Generation Integrity (Pass / Fail)	5	Measures against catastrophic failures. • 5 pts: Pass (No severe repetition, gibberish, or truncation). • 0 pts: Fail (A severe generation error is detected).

Table 1: Scoring rubric for evaluating models.

5.4.3. Evaluation Results

The models were evaluated on a set of benchmark theorem-proving tasks. The table below shows the scores for the base model and each fine-tuned variant.

Prompt	Base Model (Llama-3.2-3B)	Score	SFT-1 (Non NL dataset)	Score	SFT-2 (NL dataset)	Score	GRPO+SFT	Score
theorem imp_chain (a b c d e f : Prop) (h_1 : a → b) (h_2 : b → c) (h_3 : c → d) (h_4 : d → e) (h_5 : e → f) : a → f :=	by intro a assume h : a induction h on h generalizing b c d e { h : a => induction' h on h generalizing e f { h : a => ... } ... (Enters a severe repetitive loop)	5	intro h simp_all only [h_1, h_2, h_3, h_4, h_5, true_and] tauto	10	have h_main : a → f := by intro ha have hb : b := h_1 ha have hc : c := h_2 hb have hd : d := h_3 hc have he : e := h_4 hd have hf : f := h_5 he exact hf exact h_main	93	have h_main : a → f := by intro ha have hb : b := h_1 ha have hc : c := h_2 hb have hd : d := h_3 hc have he : e := h_4 hd have hf : f := h_5 he exact hf exact h_mai	93
theorem succ25_mem_of_zero_and_succ_closed (N : Nat → Prop) (h0 : N 0) (hS : ∀ {x}, N x → N x.succ) : N 25 :=	by induction on x in 25...; case zero => by assumption; case succ x' => hS (succ x') (hS x')	40	intro h simp [h0, hS, h] aesop	10	by have h1 : N 1 := by ... have h2 : N 2 := by ... ... have h19 : N 19 := by have h19_1 : N 18 := h18 have h19_2 : N 19 := hS 18 h19_1 exact h19_2 (Correct but truncated by token limit)	55	by have h1 : N 1 := by ... ... have h24 : N 24 := by have h24_1 : N 24 := hS 23 h23 exact h24_1 have h25 : N 25 := by have h25_1 : N 25 := hS 24 h24 exact h25_1 exact h25	80

theorem add_zero_eq_self (n : ℕ) : n + 0 = n :=	by induction n with (0 | succ n) => | 0 => -- Base case: 0 + 0 = 0 by simp | succ n => -- Inductive step by simp	40	intro h simp [h]	10	have h : n + 0 = n := by simp exact h	98	have h : n + 0 = n := by rfl exact h	98
theorem succ_add_one_eq_add_two (n : ℕ) : n.succ + 1 = n + 2 :=	by apply succ_add_one_eq_add_two_ind induction n | base n => rfl | succ n ih => by induction n ...	40	intro n simp [n.succ_add_one_eq_add_two]	10	have h_main ... := by simp [Nat.succ_eq_add_one] <;> ring <;> simp_all <;> ring <;> simp_all <;> ring	40	have h_main ... := by simp [Nat.succ_eq_add_one] <;> ring <;> simp_all <;> ring <;> simp_all <;> ring	40
theorem succ_pred_self (n : ℕ) (h : n > 0) : (n - 1).succ = n :=	import Lean theorem ... := by have n.succ = n + 1, ... -- ... have (n - 1).succ = n, ... (Enters a deep repetitive loop of self-correction)	5	intro h simp [h]	10	have h_main ... := by cases n with | zero => ... | succ n => simp [...] <;> (try omega) <;> (try simp_all ...) <;> (try { cases n with ... }) ... (Full "kitchen sink" of broken tactics)	40	have h_main ... := by cases n with | zero => contradiction | succ n => cases n with | zero => simp [Nat.succ_eq_add_one] <;> norm_num | succ n => simp_all [...] <;> ring_nf at * <;> omega	45
theorem two_divides_double (n : ℕ) : 2 ∣ n + n :=	by (reflexive n) and (or (n % 2 = 0) (n % 2 ≠ 0)) and (if n % 2 = 0 then 2 ∣ n else 2 ∣ n + 1)	10	intro h simpa using h	10	(Enters a repetitive thought-loop in the explanation, failing to produce a final proof block)	5	have h_main ... := by use n -- We need to show that 2 * n = 2 * n <;> simp [mul_comm] <;> ring <;> omega exact h_main	94

theorem two_divides_double (n : ℕ) : 2 ∣ n + n :=	begin assume n : ℕ, have n_is_even : n + n = 2 * n := by ring, have two_divides_n : 2 ∣ 2 * n := by ring, have two_divides_double : 2 ∣ n + n := by from two_divides_n, end	40	intro h simpa using h 0	10	have h_main ... := by rw [Nat.dvd_iff_mod_eq_zero] simp [...] <;> induction n with | zero => simp | succ n ih => simp [...] at ih ⊢ <;> omega (Incorrect use of induction)	45	have h_main ... := by have h : n + n = 2 * n := by ring rw [h] simp [Nat.dvd_iff_mod_eq_zero, Nat.mul_mod, Nat.add_mod] <;> omega exact h_main	93

Table 2: Scoring 1. Base Model, 2. SFT over Non-CoT data, 3. SFT over CoT data, 4. GRPO+SFT

6. Experiment-2: OpenBootstrappedTheorem (OBT) Style Finetuning for Lean 4 Proof Synthesis

6.1 Objective

For this experiment, we implemented a fine-tuning pipeline on the OBT dataset [14]. This dataset consists of examples that have natural language explanations embedded into formal proofs. We trained a Llama 3 instruction model using two concepts introduced in TheoremLlama research paper [13]. The goals were to (i) improve end-to-end Lean proof synthesis quality, (ii) reduce syntax errors, and (iii) stabilize long tactics via few-shot block training and curriculum sorting.

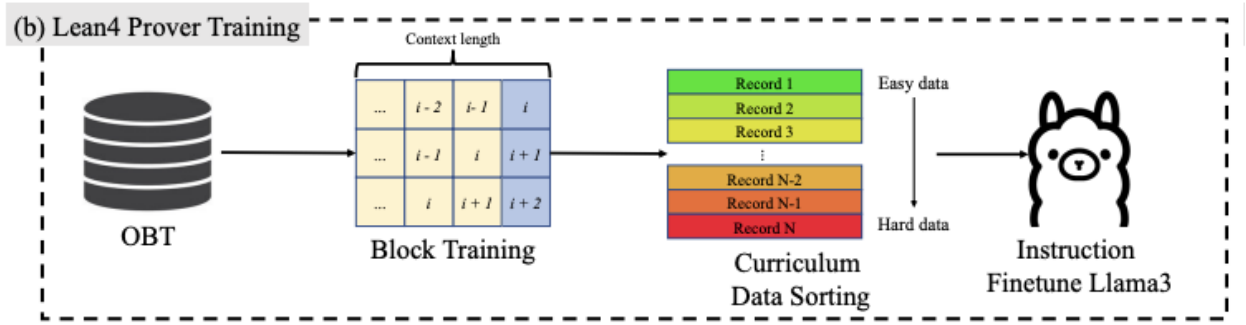


Fig 13: TheoremLlama fine-tuning process

6.2 Data

Primary dataset: OpenBootstrappedTheorem (RickyDeSkywalker/OpenBootstrappedTheorem)[14], containing natural-language (NL) statements+proof rationales paired with Lean statements, commented/bootstrapped Lean proofs, and raw Lean code.

Schema mapping: Instead of splitting the natural-language blob, we kept the entire “informal statement and proof” text as a single field (nl_text) to preserve global context. A normalization pass mapped heterogeneous column names (e.g., Generated_informal_statement_and_proof, Statement, Proof, Commented_proof) to a consistent schema {name, nl_text, lean_statement, lean_proof_raw, lean_bootstrapped, ...}.

6.3 Method

6.3.1 Prompt Format

We adopted manual prompt format using reserved special tokens:

```
<|start_header_id|>system<|end_header_id|> ... <|eot_id|>, then
<|start_header_id|>user<|end_header_id|> ... <|eot_id|>, then
<|start_header_id|>assistant<|end_header_id|>
```

Few-shot block training: each block contained a solved example with:

- Natural language version of the theorem and proof
- Lean statement in a fenced code block
- Lean theorem and proof in a fenced code block

6.3.2 Curriculum Sorting

We calculated a difficulty score for all the examples using the formula: d = length of lean_bootstrapped + 50 × (number of tactic tokens). Then, we sorted the examples from easiest to hardest. During training, each item picked its few-shot examples from a sliding window to gradually increase difficulty and avoid sudden jumps in challenge.

6.3.3 Pseudocode

```
Algorithm: OBT Data Preparation Pipeline
Input: Raw Dataset D_raw, Few-shot count k
Output: Formatted Examples E_formatted
1. Schema Unification:
    R ← new list of records
    for each row in D_raw do
        r ← StandardizeSchema(row)
        R.append(r)
    end for

2. Curriculum Sorting:
    for each record r_i in R do
        d_i ← EstimateDifficulty(r_i.lean_bootstrapped)  // Score on
length and tactic count
    end for
    O ← SortIndicesByDifficulty(d_0, d_1, ...)  // array of indices from
easy -> hard

3. Few-Shot Prompt Construction:
    E_formatted ← new list of examples
    for each index i in the sorted order O do
        // Select k easier examples from the curriculum
        P_prior_indices ← GetPriorExamples(i, O, k)

        // Build the multi-turn prompt
        prompt_messages ← BuildFewShotPrompt(SystemPrompt,
P_prior_indices, R[i])
        // Define the target for completion-only loss

        completion ← R[i].lean_bootstrapped
        if completion is not empty then
            E_formatted.append({'messages': prompt_messages, 'completion':
completion})
        end if
    end for

    return E_formatted
```

6.3.3 Completion-Only Loss with TRL

We used TRL's SFTTrainer (≥ 0.21) with its prompt–completion API. The completion column is the Lean proof (prefer lean_bootstrapped, fallback to lean_proof_raw). We relied on TRL to infer completion-only loss without a custom data collator. This focused learning on the proof generation region and avoided diluting gradients on the prompt side.

6.3.5 Training Hyperparameters

1. Epochs = 2, LR = 2e-4 (cosine schedule), weight decay = 0.0.
2. Per-device batch = 1, grad accumulation = 16 (effective batch = 16).
3. Warmup ratio = 0.03; logging every 10 steps; save each epoch (or ≥ 5k steps).
4. Eval split = last 1% of curriculum-ordered examples (small sanity set).
5. Seeding and TF32/bf16 enabled for throughput; Flash-Attention v2 for attention kernels.
6. QLoRA: 4-bit NF4 quantization (bitsandbytes), bf16 compute, LoRA adapters (rank = 16, α = 32, dropout = 0.05) on causal-LM projections.
7. Context: model_max_length = 8192 to accommodate long NL proof rationales + multiple few-shot blocks.

6.3.6 Implementation Summary

We first performed schema unification with map_row_schema, which selected the first available field among multiple candidate keys to robustly handle dataset variants. We then ordered examples by an easy→hard curriculum using a difficulty heuristic that combined sequence length with counts of common Lean tactic tokens (e.g., simp, rw, apply, aesop, linarith) to approximate complexity. For few-shot conditioning, we built a prior few-shot pool from a few-shot window of earlier (easier) items in the curriculum and sample block_k solved examples (typically 3) to prepend as demonstration blocks. Prompt rendering used render_example_block for each solved example and then appends the current item with your exact section headers and terminators, preserving the manual <|start_header_id|>…<|eot_id|> format. Training was run with TRL's SFTTrainer using dataset_kwargs={"prompt_column":"prompt","completion_column":"completion"}, enforcing completion-only loss with no chat_template and no custom collator
Outputs: https://huggingface.co/rkumar1999/llama3.2-3b-obt

6.4 Results

The fine-tuning process produced the resultant model available publicly on huggingface: rkumar1999/llama3.2-3b-obt

6.4.1

The same evaluation rubric in 7.4.1 was used to score and compare the resultant model

Prompt	Base Model (Llama-3.2-3b)	Score	Finetuned Model (rkumar1999/llama3.2-3b-obt)	Score
### Lean4 version of theorem statement: ```lean theorem imp_chain (a b c d e f : Prop) (h₁ : a → b) (h₂ : b → c) (h₃ : c → d) (h₄ : d → e) (h₅ : e → f) : a → f :=	by { assume a, have h₁ a, have h₂ h₁ a, have h₃ h₂ a, have h₄ h₃ a, have h₅ h₄ a, apply h₅ }	35	⟨h₁, h₂, h₃, h₄, h₅⟩	50
### Lean4 version of theorem statement (explicit 25 iterations, no induction): ```lean theorem succ25_mem_of_zero_and_succ_closed (N : Nat → Prop) (h0 : N 0) (hS : ∀ {x}, N x → N x.succ) : N 25 :=	-- initial induction induction N on 25 -- base case | 0 => h0 | succ i => hS (N i)	20	-- 25 iterations of N x → N x.succ and N 0 | 0, h => h 0 | 1, h => h 1 | 2, h => h 2 | 3, h => h 3 | 4, h => h 4 | 5, h => h 5 | 6, h => h 6 | 7, h => h 7 | 8, h => h 8 | 9, h => h 9 | 10, h => h 10 | 11, h => h 11 | 12, h => h 12 | 13, h => h 13 | 14, h => h 14 | 15, h => h 15 | 16, h => h 16 | 17, h => h 17 | 18, h => h 18 | 19, h => h 19 | 20, h => h 20 | 21, h => h 21 | 22, h => h 22 | 23, h => h 23 | 24, h => h 24 | 25, h => h 25	40
### Lean4 version of theorem statement: ```lean theorem add_zero_eq_self (n : ℕ) : n + 0 = n :	by induction n with | 0 => sorry | n.succ => by contradiction	5	add_comm n 0 ▸ add_zero	55
### Lean4 version of theorem statement: ```lean theorem	by { assume h : n.succ + 1 = n + 2; -- assume that the theorem is true for some n	10	-- This theorem states that the successor of a natural number plus one is equal to the natural number plus two	100

succ_add_one_eq_add _two (n : ℕ) : n.succ + 1 = n + 2 :=	-- we want to show that the theorem is true for n.succ assumption h }		rfl	
### Lean4 version of theorem statement: ```lean theorem succ_pred_self (n : ℕ) (h : n > 0) : (n - 1).succ = n :=	by { apply succ_pred_def, apply h, exact (n - 1)}	5	-- Proof that the successor of a natural number is equal to the number itself. -- This theorem is used to prove the well-foundedness of the natural numbers. Nat.add_sub_self _ (Nat.succ_pos_iff.2 h) h.	20
Natural language version of theorem and proof: two_divides_double Twice any natural number is divisible by 2. For any n, the number n + n equals 2 * n, so 2 divides n + n with witness n. ### Lean4 version of theorem statement: ```lean theorem two_divides_double (n : ℕ) : 2 ∣ n + n :=	by { assume h : n + n = 2 * n }		by simp [add_comm] -- show that 2 divides n + n by using the commutative property of addition	
### Lean4 version of theorem statement: ```lean theorem two_divides_double (n : ℕ) : 2 ∣ n + n :=	by (refl : 2 ∣ 2) => (refl : n + n = 2 * n) => (by symmetry => refl : 2 ∣ 2 * n)	5	-- This theorem states that 2 divides the sum of a number and itself. -- In other words, if n is a multiple of 2, then n + n is also a multiple of 2. two_divides_add n n	25

Table 3: Scoring 1. Base Model, 2. SFT finetuned model over OBT dataset

7. Experiment-3: Fine-tuning a Mixture-of-Experts Model on Math and Lean Datasets

7.1 Objective

This experiment quantified the trade-offs between mixed fine-tuning and sequential (curriculum) fine-tuning on two adjacent but different domains, natural-language math reasoning and formal Lean proofs using a Mixture-of-Experts (MoE) backbone. We tested the hypothesis that mixed training better preserved performance across domains, while sequential training risked catastrophic forgetting on the earlier domain.

7.2 Data

1. Math (NL, stepwise): rkumar1999/Mixture-of-Thoughts-math-cleaned
(cleaned subset from open-r1/Mixture-of-Thoughts; max sequence ≤4096 tokens)
2. Lean (formal proofs): rkumar1999/DeepSeek-Prover-V2-chat-cleaned
(filtered from Cartinoe5930/DeepSeek-Prover-V2-generation; max sequence ≤4096 tokens)

Both corpora were length-bounded to fit the model's context window and avoid overflow during SFT.

7.3 Setups

We held architecture, optimizer, and PEFT settings constant and vary only the data curriculum:

- Regime A: Mixed (Joint) Fine-Tuning:
  Train on a 50/50 interleaved mixture of Math and Lean examples per batch (or per epoch) with temperature-based sampling if desired to smooth domain imbalance.
  Goal: learn both domains concurrently and minimize interference.

- Regime B: Sequential (Math → Lean):
  Stage 1: fine-tune on Math only → checkpoint A_1.
  Stage 2: continue from A_1 and fine-tune on Lean only → checkpoint A_2.
  Goal: measure forgetting in Math after adapting to Lean.

Backbone: MoE model (microsoft/phi-tiny-moe-instruct) with identical PEFT/QLoRA settings across both regimes for a fair comparison.

7.4 Results

7.4.1 Outputs

The fine-tuning process resulted in three distinct model artifacts, each representing a different training regime or stage. These models are publicly available on the Hugging Face Hub and serve as the basis for the comparative evaluation in the following section.

1. Mixed-Domain Model (rkumar1999/Phi-mini-MoE-Mix-Prover-openr1-distill-SFT) This model is the result of Regime A (Joint Fine-Tuning). It was trained on an interleaved mixture of the Math and Lean datasets, designed to learn both domains concurrently and minimize interference between them.

2. Lean-Specialized Checkpoint (rkumar1999/Phi-mini-MoE-Prover-openr1-distill-SFT) This model represents the intermediate output from Stage 1 of the sequential fine-tuning process. It was fine-tuned exclusively on the formal Lean proof dataset (rkumar1999/DeepSeek-Prover-V2-chat-cleaned).

3. Sequential Fine-Tuned Model (Lean -> Math) (rkumar1999/Phi-mini-MoE-Prover-Math-openr1-distill-SFT) This is the final model from the sequential curriculum. It was initialized from the Lean-Specialized Checkpoint and then further fine-tuned exclusively on the natural-language math dataset (rkumar1999/Mixture-of-Thoughts-math-cleaned). This artifact is used to measure the catastrophic forgetting of the initial Lean domain.

Evaluate both checkpoints (Mixed and Sequential-final) on two held-out test sets:

1. Math Test (NL) – accuracy on final answers, chain-quality (self-consistency), and verbosity penalties.
2. Lean Test (Formal) – proof validity (kernel pass/fail), syntax correctness, and proof efficiency (|gen_lines – ideal_lines|).

7.4.2 Results - Lean4 prompts

Prompt	BASE (microsoft/Phi-mini-MoE-instruct)	Score	FT - mixed data	Score	FT - Sequential data	Score
theorem imp_chain (a b c d e f : Prop) (h₁ : a → b) (h₂ : b → c) (h₃ : c → d) (h₄ : d → e) (h₅ : e → f) : a → f :=	lean4 begin intro a, have b := h₁ a, have c := h₂ b, have d := h₃ c, have e := h₄ d, have f := h₅ e, end	35	lean4 by have h_main : a → f := by intro h have h₆ : b := by have h₆₁ : a → b := h₁ …. have h₇₂ : b := h₆ exact h₇₁ h₇₂exact h₁₀ exact h_main	84	(Catastrophic failure: The model entered a severe "thought loop" and produced no Lean code.)	5
theorem succ25_mem_of_zero_and_succ_closed (N : Nat → Prop) (h0 : N 0) (hS : ∀ {x}, N x → N x.succ) : N 25 :=	lean4 begin intro h0, induction hS, cases h0, cases hS, apply hS, apply hS, ... end	20	lean4 by have h_main : N 25 := by have h1 : N 0 := h0 have h2 : N 1 := hS ⟨0, by simp⟩ h1 have h3 : N 2 := hS ⟨1, by simp⟩ h2 ... have h26 : N 25 := hS ⟨24, by simp⟩ h25 exact h26 exact h_main	60	(Catastrophic failure: The model entered a severe "thought loop" and produced no Lean code.)	5
theorem add_zero_eq_self (n : ℕ) : n + 0 = n :=	lean4 begin rw [n + 0 = n], Qed. (The model got stuck in a repetitive loop of self-correction before outputting this invalid proof.)	5	lean4 by have h : n + 0 = n := by simp <;> rfl exact h	97	(Catastrophic failure: The model entered a severe "thought loop" and produced no Lean code.)	0

theorem succ_add_one_eq_add_two (n : ℕ) : n.succ + 1 = n + 2 :=	lean4
	begin
	cases n,
	{
	rw [n.succ + 1 := 0 + 1],
	rw [0 + 1 := 0 + 2],
	},
	assume h : k.succ + 1 = k + 2,...(Enters a repetitive loop of invalid rw tactics.)
		0	lean4
			by
			have h_main : n.succ + 1 = n + 2 := by
			rw [Nat.succ_eq_add_one]
			<;> ring
			<;> omega
			exact h_main
				96	(The model produced a long explanation but failed to generate a final Lean code block.)	30

theorem succ_pred_self (n : ℕ) (h : n > 0) : (n - 1).succ = n :=	lean4 begin rw h, rw (n + 1), exact (pred (n + 1)).succ = n, end (Uses hallucinated pred function and invalid rw tactics.)	5	lean4 by have h_main : (n - 1).succ = n := by have h_1 : n ≥ 1 := by linarith have h_2 : (n - 1).succ = n := by have h_3 : n - 1 ≥ 0 := by omega have h_4 : (n - 1).succ = n := by rw [Nat.succ_eq_add_one] <;> omega exact h_4 exact h_2 exact h_main	92	(Catastrophic failure: The model entered a severe "thought loop" and produced no Lean code.)	5

Table 4: Scoring 1. Base Model, 2. Fine-tuned model on mixed data, 3. Fine-tuned model on sequential data

8. Conclusion

This project explored different ways to improve Large Language Models' mathematical capabilities in the challenging area of formal theorem proving. We found that the limitations of general-purpose LLMs in solving formal proofs can be improved through targeted training on specialized datasets, smart fine-tuning techniques, and the use of external tools.

We tested our approach on a smaller GPT-2 model through parameter-efficient fine-tuning on specialized math datasets. We also explored a hybrid solution where we integrated the model with symbolic tools like Mathics and LEAN to perform complex calculations and recursively search for a proof, respectively.

Our main experiments scaled up to more powerful architectures and showed significant insights. Fine-tuning of Llama-3.2-3B demonstrated that natural language guided datasets are more effective than training on formal code alone. The GRPO-aligned model achieved highest performance of 95+ (scored out of 100) on LEAN4 proof generation, followed by a plain SFT finetuned model (85+). Furthermore, the OpenBootstrappedTheorem (OBT) experiment confirmed that combining curriculum-learning and few-shot prompting approaches can help models handle longer proofs. We compared two different fine-tuning regimes, mixed and sequential, on a Mixture of Experts (MoE) model. Fine-tuning the model on a mixed dataset increased the model performance to 90+, while sequential fine-tuning resulted in forgetting the original knowledge visible in its low performance of < 20 for many examples.

Our findings show the value of chain-of-thought reasoning and fine-tuning techniques in creating models that provide well-reasoned solutions rather than generating text plainly. Future work could extend these findings by scaling to larger models like DeepSeek, exploring more advanced reinforcement learning techniques for tactic selection. These directions aim to shift models from text generation machines toward robust thinking agents.

9. References

[1] Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., & Steinhardt, J. (2021). Measuring Mathematical Problem Solving With the MATH Dataset. arXiv preprint arXiv:2103.03874. https://doi.org/10.48550/arXiv.2103.03874

[2] Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, Ł., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., & Schulman, J. (2021). Training Verifiers to Solve Math Word Problems. *arXiv preprint arXiv:2110.14168*. https://doi.org/10.48550/arXiv.2110.14168

[3] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). LoRA: Low-Rank Adaptation of Large Language Models. *arXiv preprint arXiv:2106.09685*. https://doi.org/10.48550/arXiv.2106.09685

[4] Liu, S.-Y., Wang, C.-Y., Yin, H., Molchanov, P., Wang, Y.-C. F., Cheng, K.-T., & Chen, M.-H. (2024). DoRA: Weight-Decomposed Low-Rank Adaptation. *arXiv preprint arXiv:2402.09335*. https://doi.org/10.48550/arXiv.2402.09335

[5] de Moura, L., Kong, S., Avigad, J., van Doorn, F., & von Raumer, J. (Year). The Lean Theorem Prover (System Description). *Microsoft Research* and *Carnegie Mellon University*.

[6] Yang, K., Swope, A. M., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., & Anandkumar, A. (Year). LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. *Caltech*, *NVIDIA*, *MIT*, *UC Santa Barbara*, *UT Austin*. Retrieved from https://leandojo.org

[7] leanprover-community. (n.d.). repl: A simple REPL for Lean 4. GitHub repository. Retrieved December 9, 2024, from https://github.com/leanprover-community/repl

[8] (2022). Mathics: Open-Source Alternative to Mathematica. Retrieved December 9, 2024, from https://mathics.org/

[9] LangChain. (2024). LangChain Documentation. Retrieved December 9, 2024, from https://python.langchain.com/docs/introduction/

[10] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E. H., Le, Q. V., & Zhou, D. (2016). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Google Research, Brain Team. arXiv:2201.11903v6*

[11] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., & Amodei, D. (2020). Language Models are Few-Shot Learners. 34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada

[12] Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., & Steinhardt, J. (2021). Measuring Mathematical Problem Solving With the MATH Dataset. NeurIPS. Retrieved December 9, 2024, from https://github.com/hendrycks/math

[13] Ruida Wang H, Jipeng Zhang, Yizhen Jia, Rui Pan (2024). TheoremLlama: Transforming General-Purpose LLMs into Lean4 Experts. https://arxiv.org/html/2407.03203v1

[14] Open Bootstrapped Theorem Dataset. Huggingface dataset card. https://huggingface.co/datasets/RickyDeSkywalker/OpenBootstrappedTheorem

[15] Omar Sanseviero, Lewis Tunstall, Philipp Schmid, Sourab Mangrulkar, Younes Belkada (2023). Mixture of Experts Explained. https://huggingface.co/blog/moe

[16] A. Vaswani et al., “Attention Is All You Need,” in Advances in Neural Information Processing Systems, 2017. https://doi.org/10.48550/arXiv.1706.03762

[17] A. Radford et al., “Language Models are Unsupervised Multitask Learners” OpenAI Blog, 2019. https://api.semanticscholar.org/CorpusID:160025533

[18] L. Ouyang et al., Training language models to follow instructions with human feedback. 2022. https://doi.org/10.48550/arXiv.2203.02155

[19] W. Fedus, B. Zoph, and N. Shazeer, “Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity” *Journal of Machine Learning Research*, vol. 23, no. 120, pp. 1–39, 2022. https://doi.org/10.48550/arXiv.2101.03961

[20] Hugging Face, “Text Generation Inference,” GitHub Repository. https://github.com/huggingface/text-generation-inference

[21] Hugging Face, “TRL - Transformer Reinforcement Learning,” GitHub Repository. https://github.com/huggingface/trl

[22] DeepSeek-AI, "DeepSeek-R1: A Reasoning Language Model," arXiv preprint arXiv:2407.12484, 2024.

[23] Ouyang et.al., “Training language models to follow instructions with human feedback”, 2022. https://arxiv.org/abs/2203.02155

[24] Hugging face, Open R1: A fully open reproduction of DeepSeek-R1. https://github.com/huggingface/open-r1