

Fine Tuning An LLM

Andrew Ely

Objective

We seek to take an existing trained LLM and fine-tune it to obtain a model that is useful to mathematics researchers, with a training budget of \$10.

We decided to pick the narrow but fundamental mathematical task: counting integer solutions to linear inequalities over a bounded domain. The model is asked questions of the form “For how many integers x in $[a, b]$ does the inequality $f_1(x) (\leq, \geq, <, >) f_2(x)$ hold?”, and must output the exact number of solutions.

This kind of discrete reasoning appears frequently in many areas of mathematics and theoretical computer science, for example when: checking small cases of a conjecture over a finite range, counting plausible values subject to linear constraints, or verifying that certain inequalities hold for all integer inputs in a bounded interval. More broadly, our goal is to experiment with the process of fine tuning an LLM.

This paper is broken up into two parts. Part 1 outlines our fine tuning process and results for the domain mentioned above. Part 2 explores a far harder domain of fine tuning the model to solve AIME problems. We found in part 1 that our fine tuning methods significantly improved the models ability to solve the problems. However, in part 2 we observe the limitations of lightweight models when attempting to train them to solve very hard problems.

Part I

Model Selection

When considering an LLM to fine tune, we ideally want to pick an LLM so that out of the box it can mathematically reason about these problems at least somewhat well, and then after our fine tuning, it becomes significantly better.

Selecting a model such as GPT-2 would be a bad choice because that model was not designed to have mathematical reasoning, and it performs poorly with math problems in general. So, any improvements are not only unlikely but bounded by a small accuracy at best.

On the other hand, selecting a model like deepseek-R1 which reportedly has 87.5% AIME 2025 accuracy would be a bad choice because for our problems the model will likely have nearly 100% accuracy since they are much easier than AIME problems, and improving to a better accuracy would almost certainly be a small increase and the results will be underwhelming.

Also, our model selection will be restricted to models that are open source and available for us to use. We also want to pick a model that can be trained with affordable compute, so a smaller model will be preferred over a very large model. After considering different models, we decided to use Qwen-2.5-Math-1.5B as our model to fine tune.

Qwen-2.5-Math-1.5B and other related models

The Qwen2.5 models are a family of open-weight LLMs developed by Alibaba as the next generation after Qwen and Qwen2. They come in a range of sizes from 0.5B parameters up to 72B parameters. These models have specialized variants for different domains, namely the Qwen2.5-Math models which are designed for math reasoning. They were trained with $\sim 580K$ math problem with English natural language, and $\sim 500K$ math problems with Chinese natural language. There are also variants of Qwen2.5-Math-Instruct which are already fine tuned versions of the math models, trained with the goal of producing better natural language instruction following (such as stricter formatted outputs). The math variants, like the Qwen2.5-Math-1.5B model we use, are trained on curated mathematical data and are designed to be especially good at symbolic manipulation, problem solving, and competition-style reasoning, while still being small enough to fine-tune with a reasonable budget. We decided to go with this model because we found it to have reasonable results out of the box, and it is relatively light weight to fine tune. Even though the Chinese math problem training, which focused on making the model multi-lingual, does not particularly help with our specific task (our problems will all be in English), this model seems to be a good fit in practice.

Data Selection

LLM’s are trained with extremely large amounts of data, of the order of millions or billions of data points. Since our LLM is already trained with over 1 million math problems, and the goal is to nudge the model into learning from specific new data, we do not need an extremely large data set for fine tuning. For our specific use case, we decided to curate our own dataset using a script. This allowed for us to guarantee consistency in our data and ensure it is accurate. See file “generateData.py” for the data generation.

This type of relatively small data set usage is known as few-shot training. There are two different types

of few-shot training. One involves only cleverly prompting the LLM with questions and answers and hoping the model generalizes from its responses. This does not involve changing the model weights. The other type of few-shot training refers to using a relatively small amount of data to change the models weights. When we refer to using few-shot training, it is important to clarify we are referring to the idea of updating the weights with a small dataset, i.e. our fine tuning process does indeed change the underlying model.

Train Data

During fine tuning, we will need data to feed the model so that it can learn. In general, we want enough data to expose the model to a wide variety of problems, but not too much data that it takes too long to train. On the other hand, too little data can lead to the classical issue of overfitting. When fine tuning a model, it is hard to have an idea beforehand of how much data is appropriate since we have no comparisons to guide this decision. For this reason, we decided to start with 400 train data points. We decided to produce this data algorithmically using a script. This resulted in a dataset with a highly formatted input and output and verifiably correct answers. Each data point consists of a problem as a string, and a solution as a string formatted as a boxed answer. For example:

```
“question”: ”For how many integers  $x$  in the range  $[-50, 50]$  is the inequality  $5x - 10 < -4x - 2$  satisfied?”  
“answer”: “\\boxed{51}.”
```

Validation Data

When experimenting with the fine tuning process and producing different models with different weights, we will want to have a reserved dataset that allows us to reasonably assess how well a model will perform. This is the reason we use a held out validation data set. This consists of 40 problems, all different from the 400 train problems, with the same format as the train data. When we use the validation data, we will not present the model with the final answer, only the question. Then after the model answers each question we compare to the ground truth answers to assess accuracy.

Test Data

After we achieve a high validation accuracy, we will want another dataset that the model has never seen before to assess accuracy on brand new questions. We decided to use 40 data points for the test set. During training, we will not expose the model to test data until we decide on our final model.

Model assessments using our data

It is important to clarify why in our accuracy analyses for our fine tuned models, we do not include train

data accuracy and do not include test data accuracy. We do not evaluate our models with the test data because we do not want to influence our decisions during fine tuning by test data results. This dataset is specifically reserved to assess our final models accuracy, after we establish a final model.

We also do not include train data accuracy because this would require prompting the model with every single train data point which would take way too long to do since there is a lot of train data. During training, the model will output training loss for the objective function, but it does not directly check the models outputs for the train question prompts. This is why our analyses only report training loss, and not training accuracy. It is important to clarify that this loss function value is not a percentage of accuracy, but only the numerical value of the optimization objective that the model is trying to minimize. A lower loss typically correlates with better performance, but it is not directly interpretable as “X% correct”. This is why we rely on separate validation accuracy computed on a held-out set of problems to evaluate how well the model is actually answering questions correctly.

Baseline performance

Before we attempt to make the model better, we should have an idea of what the current ability of the model is. After loading the model, we prompted it with our validation and test sets to assess its accuracy out of the box. Our method of doing this was to prompt the model with each question and instructed it to format its final answer as “boxed{answer}”, so we can extract its answer from the response. If the model does not provide the answer in the specified format, we count it as incorrect. If a response does not create a final answer after 1024 tokens, we count it as incorrect. We set a very high token limit to allow the model to reason as long as possible, preventing cutting answers short before it provides a final answer. This results in the following baseline accuracies.

- Test set baseline accuracy: 0.25
- Validation set baseline accuracy: 0.225

We note that the validation set is slightly harder for the model than the test set, meaning we can expect high validation accuracy to correlate to high test accuracy. These baseline results suggest that the model is at least capable at mathematical reasoning. Even though the accuracy is low, its ability to solve some of the problems is motivation for improvement.

Fine Tuning the Model

Objective Function and Training Setup

Our fine tuning approach is standard supervised fine tuning (SFT) on chat-formatted data. For each training example, we construct a synthetic conversation consisting of a system message requesting step-

by-step reasoning and a boxed final answer, a user message containing the generated inequality-counting question, and an assistant message containing the correct boxed answer.

We then use the tokenizer’s chat template to turn these messages into a single text string. The SFTTrainer from the trl library treats this text as the target sequence and trains the model in a next-token prediction fashion: at each position, the model tries to predict the next token in the combined conversation.

The underlying objective function is the standard cross-entropy loss over the sequence of tokens. We use the Adam optimizer which is common for neural networks. This encourages the model both to produce coherent chain-of-thought reasoning that matches the training examples, and end its response with the correct `\\boxed{n}` answer, where n is the true number of solutions.

We use Unsloth’s `FastLanguageModel.get_peft_model` to add LoRA adapters on the attention and MLP projection layers. These adapters introduce a relatively small number of additional parameters that are trained, while the original base model weights remain frozen. This reduces memory usage and speeds up training. During fine tuning, the number of parameters that we are actually fine tuning is relatively small, only about 5% of the total parameters. However, these parameters are enough to influence the model.

Our main training hyperparameters are:

- LoRA rank $r = 64$ and scaling factor $\alpha = 128$
- batch size 2 with gradient accumulation of 4 (effective batch size 8)
- learning rate 2×10^{-4} with linear warmup and decay
- 3 epochs over the 400 training examples

LoRA and Parameter-Efficient Fine Tuning

LoRA (Low-Rank Adaptation) is a parameter-efficient fine-tuning method that decomposes weight updates into low-rank matrices. Instead of updating the full dense weight matrices in the transformer, we learn two small matrices A and B of rank r such that the effective weight becomes $W + BA$. This allows us to adapt the model to new tasks with far fewer additional parameters and with much lower memory requirements.

In our setup, LoRA is applied to the key, query, value, output, and MLP projection layers. This makes it possible to fine tune the 1.5B-parameter model entirely within the memory constraints of a Colab GPU, especially when combined with 4-bit quantization via Unsloth.

An important factor is that we can experiment with fine tuning without needing to modify or store a full new copy of all model weights. Instead, we only need to store the small LoRA adapter weights, which is useful both for exporting the model and for staying within our storage limits.

Overfitting Considerations

Because our training dataset is relatively small (400 examples), overfitting is a real concern. A model with enough capacity can simply memorize the training distribution rather than learning reusable structure about linear inequalities.

We monitor overfitting primarily via the held-out validation set. During and after training, we measure accuracy on the 40 validation examples using the same evaluation pipeline as for baseline testing. If training loss were to approach 0 while validation accuracy stagnated or decreased, that would be a strong indicator of overfitting.

Analyzing the Results

After fine tuning, we re-evaluated the model on the validation set using exactly the same prompting and answer-extraction pipeline as in the baseline assessment. This ensures that any change in accuracy is due to weight updates alone. We refer to the model outlined above as Model A.

Model A

- training loss (objective function loss): 0.78
- Validation accuracy: 0.50

So, fine tuning nearly doubles exact-answer accuracy on our validation set. This is a substantial improvement given the simplicity of our setup and the relatively small training dataset.

We also examined individual problems before and after fine tuning. We observed several consistent trends. The fine tuned model is more likely to correctly rearrange inequalities (moving terms with x to one side and constants to the other). It appears more reliable at identifying the correct interval of integer solutions and counting the number of integers in that interval. The model more consistently follows the requested output format, ending with a single `\\boxed{answer}`, rather than including multiple candidate answers or failing to box anything.

Also, we frequently observed the following common errors: off-by-one mistakes at the endpoints of the interval, sign errors when moving terms across the inequality, occasional formatting failures where the model did not output a `\\boxed{answer}` pattern at all. These error patterns suggest that the model has learned a reasonably robust procedure for handling these inequalities, but that its internal arithmetic and boundary handling are still imperfect. Importantly, however, the improvement relative to the baseline indicates that the model is not simply memorizing template answers, it is generalizing better across new unseen problems.

Model adjustments

Even though we have relatively high improvements (near double accuracy post fine-tuning), the absolute accuracy of $\sim 50\%$ is still low. We explored other variations of our fine tuning method to attempt to achieve higher accuracy. Below are outlines of our plans and results for adjusting our model.

Model B

In our model A, we used 400 train data points and 3 epochs. The number of epochs is the number of passes over our dataset that the model performs. One possible issue with this configuration is that with a small number of data points, the model can overfit the train data and result in bad generalization. With multiple passes over the data, this becomes a larger concern.

We decided to test the model with an increase in training data points to 2000 data points. We generated the data points using the same generator, and kept the same format. The motivation for this is that if the model is exposed to a larger variety of problems, then there is a better chance that it can learn from this larger set of data and generalize better. We also decreased the number of epochs from 3 to 2. This may seem counterintuitive since we increased the amount of data, but the reason for this change is to prevent the model from seeing the same data points over and over again.

Results

- training loss (objective function loss): 0.75
- Validation Accuracy: 0.725

So the adjustments to our model by simply including a larger amount and variety of train data, and decreasing the number of Epochs resulted in a higher accuracy for validation and test data. This suggests that our first model overfit the small dataset of 400 questions. We also explore fine tuning the hyperparameter to try to achieve even better accuracies.

Model C

Since Model B only made modifications to the data we presented the model, and it resulted in a better accuracy, we decided for Model C to also include these 2000 train data points and 2 Epochs. Furthermore, we explore changing the hyperparameter of our fine tuning by:

- LoRA rank $r = 32$ and scaling factor $\alpha = 64$
- batch size 2 with gradient accumulation of 4 (effective batch size 8)

- learning rate 1×10^{-4} with linear warmup and decay
- 2 epochs over the 2000 training examples

The motivation for these changes is that with 2000 examples, the model already hits 72.5% accuracy, so a lower rank adapter could maybe be used to learn the mappings from inequalities to answers. Reducing the rank enforces a lower capacity, which can reduce overfitting. And overfitting seemed to be our main issue from our first model A compared to model B. When reducing the LoRA rank, we also want to lower the alpha proportionally so we aren't over-amplifying the adapter.

In our original model, we used dropout 0. Adding dropout during training means that the model will leave some of the tunable parameters out during epochs, which can prevent overfitting since it won't update every single parameter for every single training example.

With more data examples, a slightly smaller step size can give more stable convergence by making less dramatic changes in the model parameters during learning.

Results

- training loss (objective function loss): 0.75
- Validation Accuracy: 0.80

So, our changes to hyperparameter resulted in an increased validation accuracy. With this validation score, and the exploration of three different fine tuned models, we decide to go with Model C as our final model.

Now that we have established our final model, we expose it to the test data to observe its accuracy.

- Model C test set accuracy: 0.825

Which is significantly better than our baseline accuracy of 0.25. This suggests that our fine tuning process was a large success, and that future unseen problems in the domain of finding the number of integers that satisfy inequalities will be solved with high accuracy by our model.

Part 1 Conclusion

We fine tuned the open-weight unsloth/Qwen2.5-Math-1.5B model on a narrowly defined but fundamental mathematical task: counting integer solutions to simple linear inequalities over a bounded domain. We compared using a generated dataset of 400 training examples with using a train dataset of 2000 examples. We used parameter-efficient fine tuning with LoRA, and achieved a substantial improvement in performance.

Exact-answer accuracy on a held-out test set improved from 25% in the baseline model to 82.5% after fine tuning.

These results demonstrate that even a small, already math-specialized model can benefit meaningfully from targeted supervised fine tuning on a focused task. Overall, we provided a concrete case study of how to fine tune an open-source LLM on a mathematically well-defined task under a tight budget, and showed that even modest fine tuning can materially improve the model’s usefulness for a specific kind of mathematical reasoning.

Part II

Objective

After achieving positive results for our domain in Part 1, we also wanted to address our exploration with a much harder domain of problems: solving AIME math problems. The American Invitational Mathematics Examination (AIME) is a yearly exam given to high school students, in which their performance determines qualifying for the USA Math Olympiad. Hence, these problems are quite challenging for many humans to solve. More relevantly, LLM’s have historically been bad at solving these problems. AIME problem sets are often used by LLM developers to assess the accuracy of their models.

Therefore, a model that is specifically good at solving AIME problems is naturally a useful tool for mathematicians. In general, having an LLM that is good at solving hard math problems can be useful to solve other hard problems, including potentially unsolved problems that researchers are working on.

We used the same model, Qwen2.5-Math-1.5B and attempted to fine tune it to improve accuracy when solving these AIME problems.

Our results were that the model was incapable at performing well on these problems despite fine tuning.

Baseline performance

We prompted the base model with all 30 of the 2025 AIME questions to assess its accuracy out of the box. Our method of doing this was similar to our methods in part 1. One difference was that we set a max token limit of 4,000 which is very large, but we found the model was often getting cut off before finishing reasoning and wanted to make sure we allowed it to fully work through the problem. This resulted in a baseline accuracy for the AIME 2025 problems of $3/30 = 10\%$.

This baseline accuracy was motivation that we could indeed improve the model, since it was at least able to solve some of the hard AIME problems.

Important considerations after baseline assessment

AIME problems are always integer solutions between 1 and 999. During the baseline test, the model outputs some extraneous answers like “316,234,143,255” or “-1665”. Of course one thing we could do is include in our prompt instructions, a detail that informs the model that answers will be bounded by 1 and 999. However, what we think is a better approach is to hope that the model learns this information from the train data rather than being told it. If the model can learn this fact, then the output structure constraint will result in more reasonable final answers.

Sometimes the model would produce outputs that did not adhere to the instructions of putting the final answer in “boxed{ }”. In a practical sense, this may not seem to be an issue because if the model gives the correct answer, a user can reasonably see what it is regardless of the format. However, for fine-tuning, it is critical that we have a consistent way of extracting the models final answer for every response. If the model can’t follow the instructions of putting the answer in a way we can automatically check, then we have little choice but to mark the answer incorrect since we can’t manually check all responses during training and validation when fine tuning.

Another critical issue we found was that prompting the model to answer the 30 problems was slow, and it took nearly an hour to do so. This is why we decided to use a relatively small validation set, which is described in detail in the next section.

Data Selection

The AIME exam has been administered from 1983-2025, and each exam is 30 questions for a total of 1290 problems in existence. We found a 1983-2024 data set consisting of only questions and final answers with no explained solution, a 2022-2024 data set consisting of questions and full solutions with final answers, as well as the 30 AIME 2025 problems which consists of questions and only final answers. We use all three of these data sets.

One major issue with collecting AIME data to train the model is that generating solutions to these problems is non trivial. Even a powerful LLM like GPT-5.1 would take minutes to produce outputs for each question and was not always correct. Also the generation of AIME level of difficulty problems is also non trivial. So we had no reliable way to ensure the data was very high quality, and no reliable way to generate mass data sets, and had to rely on the data that already existed. We also found that in total the datasets only had 980 problems, resulting from some problems from earlier years exams not being in the dataset, or other findable datasets. This is a key difference for this fine tuning process compared to our other domain in part 1 which consisted of very high quality train data that we prepared algorithmically.

Data Preprocessing

Since our data was not created algorithmically and was taken from three different data sets, we need to create a preprocessing block to ensure that all of our data is in as consistent of a format as possible. The three different data sets we found all have slightly different formats. This led to the need for a data preprocessing block. We create one new dataset that has three columns: 'id' - an integer index of the problem, 'question' - the full AIME problem, 'answer' - the final answer (with explanation if available). If the data had no explanation, we ensured the answer entry was of the format "The final answer is boxed{answer}". If there was an explanation, we ensured that the very last thing in the explanation is "The final answer is boxed{answer}". This enforces the model to learn from a consistent output structure, with hope that it acquires this output structure in its own responses.

Now we need to determine how to use this data in each stage of the fine tuning process. A natural construction of these splits would be to follow a chronological timeline such that we train with old problems, validate with newer problems, and test with the newest 2025 problems. The issue with this data split is that the data of the years 2022-2024 are the only data points that have full explanations. We will want to include these problems in our train data so the model can learn from full explanations. This is the motivation for our non-chronological order train/validation/test split that we use.

Train Data

Since the years 2022-2024 include rigorously reviewed solution explanations, and these are the only problems in our data set that have explanations, we will want to include them in our train data set. It is important that our model sees full explanations and learns from the reasoning. However, we also want our validation set to contain recent problems, and we can not have any overlap from train data and validation data. That is why we selected our train data to consist of the problems from years 1983-2019 and the years 2022-2024.

Also, it would be beneficial if we provided the model with some full reasoning train data throughout training, and not all at the end. That is why we implemented a random data shuffling that scatters the full-reasoning (2022-2024) problems throughout the train dataset. Our total train data set consists of 890 problems.

Validation Data

We want our validation data to be similar in difficulty to our 2025 test set problems. But we can not use the years 2022-2024 since they were selected for our train data set. We also want a validation set that is not too large, so during fine tuning we can relatively quickly assess our models performance. That is why we selected our validation data to consist of the problems from the years 2020, and 2021. These problems

have no explanation, only final answers. We will be evaluating our models performance based only on its final solution, not its explanation.

Test Data

We decided to use the 30 AIME 2025 problems as our test set. We used these problems for our baseline assessment, so after fine tuning the model it is natural to test it with the same data we initially evaluated it to see improvements. During fine tuning, it is important we never test a model with the test data. Only when we have achieved our final model will we perform a test set evaluation.

Fine tuning the model

Baseline validation accuracy

During training, we will be using the validation set to check if the model has reasonably improved, and if we can expect better performance on the test data. Since we will not test the model on the 2025 problems until we achieve our final fine tuned model, we will need a baseline accuracy for the validation set. We prompted the model out of the box with our validation data set using the same method, resulting in a baseline validation set accuracy of $14/60 = 23.33\%$. Note that this accuracy is substantially higher than our baseline 2025 problem accuracy, meaning the validation set problems are easier for the model to solve. It is important to consider during fine tuning that our validation accuracy will likely be higher than our test accuracy. So, we want a very high validation set accuracy in order to expect reasonable improvement with our test accuracy.

Fine Tuning the Model

We attempted similar architecture for fine tuning as in Part 1. Our fine tuning approach is standard supervised fine tuning (SFT) on chat-formatted data. We then use the tokenizer’s chat template to turn these messages into a single text string. The SFTTrainer from the trl library treats this text as the target sequence and trains the model in a next-token prediction fashion.

The underlying objective function is the standard cross-entropy loss over the sequence of tokens. This should motivate the model to produce not only correct, but properly formatted output.

We use Unsloth’s FastLanguageModel to add LoRA adapters on the attention and MLP projection layers. These adapters introduce a relatively small number of additional parameters that are trained, while the original base model weights remain frozen. This reduces memory usage and speeds up training.

Our main training hyperparameters are:

- LoRA rank $r = 64$ and scaling factor $\alpha = 128$

- batch size 2 with gradient accumulation of 4 (effective batch size 8)
- learning rate 2×10^{-4} with linear warmup and decay
- 2 epochs over the 890 training examples

Analyzing the Results

After fine tuning, we re-evaluated the model on both the validation and test sets using exactly the same prompting and answer-extraction pipeline as in the baseline.

Our final results are:

- Validation accuracy: 0.15
- Test accuracy: 0.10

So, our validation accuracy is lower compared to our baseline. We suspected this was due to the train data consisting of many problems that had no explanations. We also tried using a smaller dataset consisting of the 90 problems, using the 2022-2024 dataset with questions with explanations and answers. Our results still ended in lower validation accuracy. We also attempted to adjust our hyperparameter, but we could not achieve a better validation or test accuracy from a fine tuned model.

One reason for this could be that our methods for fine tuning were not ideal. However, from Part 1, we found that the methods did indeed produce positive results for a much easier domain of problems. So, with credit to our Part 1 performance, we reason that the Qwen2.5-Math-1.5B model is not able to solve AIME math problems with high accuracy under our fine tuning methodology and budget constraints.

Part 2 Conclusion

AIME math problems have been historically difficult for LLMs to solve. It has only been with recent, and very large models, that they have achieved substantially high AIME problem accuracy. For a 1.5B parameter model, it is very unlikely that it will have enough mathematical reasoning to solve problems with difficulty on par with AIME problems, regardless of how it is fine tuned.

Final Remarks

In the first part, we focused on a simple domain: counting integer solutions to simple linear inequalities over a bounded interval. Because we could generate large, clean datasets synthetically and check answers exactly, the fine-tuning setup was stable and the model learned the task very well (0.25 accuracy to 0.825). After training, the model consistently produced correct counts and its behavior was easy to analyze: mistakes were rare, and when they occurred, they were usually traceable to clear reasoning or parsing issues. This gave good evidence that, for a narrow, well-structured mathematical task with programmatically generated data and straightforward labels, small-scale fine-tuning can reliably improve performance.

In the second part, using AIME-style contest problems as the domain was much less successful. Even before fine-tuning, the baseline model only solved a small fraction of the AIME test set correctly, and our attempts at fine-tuning on a small number of full solutions did not lead to gains. The problems themselves are long, conceptually dense, and require multi-step creative reasoning, while our training budget and dataset size were limited. In practice, this meant the model often produced plausible-looking but incorrect solutions, and accuracy stayed low.

The two parts suggest that under tight compute and data constraints, fine-tuning is most effective on narrowly defined, well-structured tasks with clean supervision, and much less effective as a way to “teach” a small model to handle highly complex olympiad-style problem solving.

[References]

- [1] Qwen2.5 Technical Report
- [2] LoRA: Low-Rank Adaptation of Large Language Models
- [3] fine tuning large language models
- [4] AIME 1983-2024 dataset
- [5] AIME 2022-2024 dataset
- [6] AIME 2025 dataset