# Llamathlete: Fine-Tuning Llama 3.1 8B for Grading Answers to Math Questions

**Aditya Azad**
NYU Tandon
aa10878@nyu.edu

**Ching Huang**
NYU Tandon
ch4802@nyu.edu

**Allan Thekkepeedika**
NYU Tandon
ajt444@nyu.edu

## 1 Introduction

This Kaggle competition required us to fine-tune Llama 3.1 8B for the task of grading answers to math questions. The provided dataset contained the questions, answers, explanations for those answers, and whether the answer was correct. The challenge was difficult given the weakness of LLMs in mathematical reasoning and the time requirements for fine-tuning them. We addressed these by reasoning about and trialing different training hyper-parameters with the goal of enhancing our models' capacity to mathematically reason. Ultimately, we were able to raise ours models' accuracies from $\sim$50% to $\sim$84%. While a significant improvement, we believe further improvements can be made with training on more data, trialing different prompts and parameter configurations, and adopting QLoRA.

## 2 Adjusting the Knobs

LLMs face challenges in arithmetic tasks, and many reasons for this have been proposed (Loeber, 2024). One, for example, suggests that LLMs struggle to decompose complex math problems into the sub-problems that must be solved first (Li et al., 2023). While another suggests that the positional encoding applied to tokens don't capture numbers as well as they do words (McLeish et al., 2024). We, of course, can't address these challenges, as that requires changing the model's architecture, which was beyond the scope of this competition. But our basic strategy was to adjust the learning parameters to help the model overcome these challenges as much as possible.

### 2.1 LoRA Adapters

Low-rank adaptation (LoRA) for fine-tuning LLMs has proven effective in achieving high accuracy for a given task with relatively short training times and memory consumption (Team, 2024). It leverages the insight that only a subset of weights is essential for capturing a layer's core contributions (Hu et al., 2021). LoRA optimizes two lower-dimensional matrices whose product's dimensions equal those of the targeted weight matrix in the LLM. The product is then added to the unchanged, target weight matrix during the forward pass. This way, LoRA optimizes fewer parameters, leading to lower memory consumption, faster training, and comparable accuracy.

#### 2.1.1 Rank

Rank determines the sizes of the two matrices. Higher the rank, the more parameters that will need to be optimized. While this would lead to accuracy gains, it would consume more memory and time. Given the difficulties LLMs face with math, our intuition was to have as high a rank as possible to increase the model's capacity to mathematically reason. We also thought it unlikely that Llama was trained on extensive math-related data, and so a higher rank would enable better learning in this domain. Indeed, others who've fine-tuned for such tasks used a rank as high as 64 (Liu, 2024). Unfortunately, the highest rank that can fit into memory on the free versions of Google Colab and Kaggle is 32, which is what we went with.

#### 2.1.2 Alpha

The alpha determines the extent to which the product of the two matrices is scaled before adding it to the LLM's weight matrix. A widely-used rule of thumb is to set it to 1 or 2 times the rank value (Labonne, 2024). Because we assumed that Llama 3.1 8B wasn't trained on significant math-related text, we set the alpha to 64, allowing for more significant updates to the final weights. Moreover, we adopted rank-stabilized LoRA, which increases the scaling factor. We did this because it's proven to lead to better performance, especially for higher rank values (Kalajdzievski, 2023).

### 2.2 Prompt Engineering

Prompts direct the output of LLMs and so we paid close attention to how we crafted ours. We considered two things: one, what from the dataset do we include; and two, how do we craft the prompt to get the best results.

At the very least, we needed to include the question and the given answer. We also included the explanation for that answer. Since LLMs have poor mathematical reasoning, we thought that Llama would find it difficult to determine whether the answer was correct from the question alone. We hoped that the explanation would help Llama see the basis for the given answer and thus enable it to better judge the answer's correctness from the correctness of the explanation.

In crafting the language, we reviewed multiple prompting techniques (Schulhoff et al., 2024). Many involved multi-prompting or data augmentation, which would have complicated things and taken more time for us to implement. We did, however, implement a few prompting techniques in our final prompt in A.1:

- **Role prompting:** Assigning the LLM a role or persona can help it better execute the given task, as it forces it to align the output to the nature of that role. In our case, we used "You are a grader".
- **Emotion prompting:** Incorporating phrases of emotional importance to underscore the desired output is also shown to help. In our case, we used "It's very important to grade the Answer accurately".
- **Re-reading:** Urging the LLM to reread or review the prompt has been shown to enhance its reasoning capabilities. In our case, we instructed the LLM to "read" the question and "review" the given answer towards the end, which in effect is asking it to reread.
- **Plan-and-Solve Prompting:** This involved laying out a series of steps the LLM was to follow in executing the prompt, which has been shown to make its reasoning more robust. In our case, we outlined 4 steps the LLM was to take before determining whether the answer was correct.

### 2.3 Training Hyper-Parameters

The traditional training hyper-parameters were the most difficult to optimize. One reason is that setting one parameter correctly often depends on the values of others. For instance, a high learning rate requires a larger batch size to ensure stable updates, unlike a lower learning rate. Another reason is that different circumstances can justify opposing approaches to certain hyper-parameters. As such, while rationales for various configurations were considered, the final settings were ultimately determined through trial and error.

#### 2.3.1 Batch Size and Gradient Accumulation

Larger batch sizes allow for less noisy and more stable updates, as there are more samples to base those updates on. Unfortunately, the memory in Colab and Kaggle wouldn't allow batch sizes of 16 or higher, so the highest we went was 8 (sticking to powers of 2). However, we were able to simulate larger batch sizes by configuring the gradient accumulation step, which calculates and accumulates the gradients for multiple batches before updating the weights. And so by using a step size of 4 and 8, we simulated batch sizes of 32 and 64.

We also trialed small batch sizes in case our model training was stuck in a local minima. Because smaller batch sizes lead to noisy updates, we thought a small batch size of 4 and an accumulation step of 4 would help our model 'jump' out of local minima and find a more optimal spot on the gradient plane.

#### 2.3.2 Learning Rate

The learning rate is a hyper-parameter we largely left to experimentation to determine what was optimal. We noticed that rates in the order of $10^{-3}$ led to very unstable updates, while those in the order of $10^{-5}$ led to longer training times to achieve high accuracy. Ultimately, we used a learning rate of $10^{-4}$ as that struck a good balance between stability and speed in reaching the optimal weights.

#### 2.3.3 Learning Rate Scheduler

The scheduler determines how the learning rate varies as the training progresses. Conventional wisdom holds that the rate should decline as we approach the optimal region of the gradient plane to avoid overshooting. There were two schedulers of this type that we considered: Linear and Cosine.

The linear scheduler decays the LR at a constant rate, while the cosine decays it at a changing rate that's a function of a cosine curve. This means that the cosine scheduler maintains on average a higher LR than the linear during the first half of training,

but then decays it dramatically to a level lower than the linear during the second half.

Because we felt that significant weight updates were needed for the model to overcome the challenges of arithmetic tasks, we suspected that a cosine scheduler would outperform the linear one. Nevertheless, we trialed both in our experiments.

#### 2.3.4 Miscellaneous

Other configurations like the number of training steps and warm-up steps in addition to the weight decay were determined by trial and error.

## 3 Experiment

### 3.1 Data Preprocessing

The dataset contains 1,010,000 data points. Each contains a math question, the given answer, an explanation for the answer, and a boolean value indicating whether the given answer is correct. 10,000 data points were set aside as test data; the remaining million were for training.

Unfortunately, the training data was slightly unbalanced where 60% were negative samples. To prevent a bias towards negative outputs, we rebalanced the training set by removing a random sample of 200,000 negative data points. While this did reduce the potential amount of data we could train on by 20%, in reality it made no difference as our selected combination of max steps and batch size would result in training on much fewer data.

### 3.2 Prompt Generation

As the Llama model expects prompts as inputs, we reformatted each of our data points into a prompt format. The prompt that we settled on is given in Appendix A at A.1.

### 3.3 Exploration

To explore optimal combinations of hyper-parameters, we performed a grid search over several of them, running each model for around 300 steps. From these models, we chose a few promising ones based on their training losses and performed further training and fine-tuning on them. Since computing validation loss during training requires longer training times, we decided to forgo validation. Instead, we used the training loss and the public score on Kaggle to evaluate our models, essentially treating the Kaggle public test set as our validation set. Table 1 in Appendix A shows the list of models that we further trained after the grid search and submitted on Kaggle.

Initially, we hypothesized that larger batch sizes would help the model take more reliable steps toward optimal weights, as smaller batches lead to noisier updates. Thus, we aimed to maintain a large effective batch size of 32 and explored other hyper-parameters, such as the scheduler and learning rate. From observing the training losses and public Kaggle scores, we found that the cosine scheduler outperformed the linear scheduler when all other hyper-parameters were held constant for models trained with 1,200 steps.

Additionally, we observed that a learning rate of $1e^{-5}$ was too small, resulting in lower accuracy on the testing dataset, while a learning rate of $3e^{-4}$ did not significantly improve accuracy. These were compared to $1e^{-4}$, which is recommended as a good learning rate for fine-tuning large language models (LLMs) (Parthasarathy et al., 2024). When we tested higher learning rates, we found that $5e^{-4}$ and $1e^{-3}$ caused exploding gradients. Ultimately, we decided to settle on a learning rate of $1e^{-4}$.

After fixing the model to have LoRA rank 32, alpha 64, an effective batch size of 32, the cosine scheduler, and a learning rate of $1e^{-4}$, we aimed to train the model with additional data. However, after training the model for 1200 and 1500 steps and submitting the inference results to Kaggle, we observed that the model's performance did not improve with more training. In fact, its accuracy decreased, suggesting that the model may have overfitted or got stuck in a local minima.

To address this, we decided to reduce the effective batch size to 16. We supposed that smaller batches would lead to noisier updates, which would help our model better generalize and escape local minima. Indeed, we found that reducing the effective batch size led to improved model performance, which it made one of our candidate models for further training.

## 4 Final Models and Results

Since the competition allowed us to submit three model inferences, we selected the top 3 models based on their performance on the Kaggle test set from Table 1. Each of the three have a LoRA rank of 32, an alpha of 64, a weight decay of 0.001, and a learning rate of $1e^{-4}$. The key, differentiating hyper-parameters for the models are shown in Table 2.

| Model | Warm-up steps | Batch size | Accumulation steps | Scheduler |
|---|---|---|---|---|
| 1 | 5 | 8 | 4 | cosine |
| 2 | 5 | 8 | 4 | linear |
| 3 | 50 | 4 | 4 | cosine |

Table 2: Final Models

Figure 1 below shows the training loss of the final models as well as the max steps they were trained for.

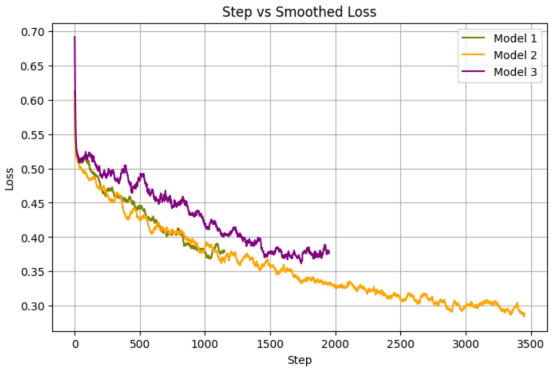


Figure 1: Training Loss of Final Models

### 4.1 Model 1

Model 1 was trained for 1,200 steps. Its training loss is plotted and shown in Figures 1 and 2. We smoothed the loss with a rolling average of 50 steps for clearer visualization of the trend.

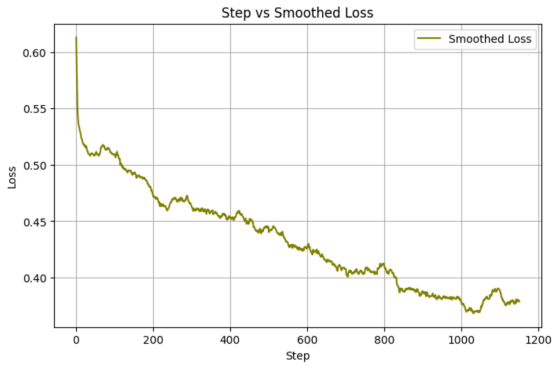


Figure 2: Training Loss for Model 1

Model 1 uses the cosine scheduler, which tends to maintain a higher learning rate than than the linear scheduler during the first half of training. This may be why the training loss does not decrease as smoothly as in Figure 2. Nevertheless, its losses were similar Model 2's for the same steps, as seen in Figure 1.

### 4.2 Model 2

Model 2 was trained for 3,500 steps. Its smoothed training loss is plotted and shown in Figure 2.

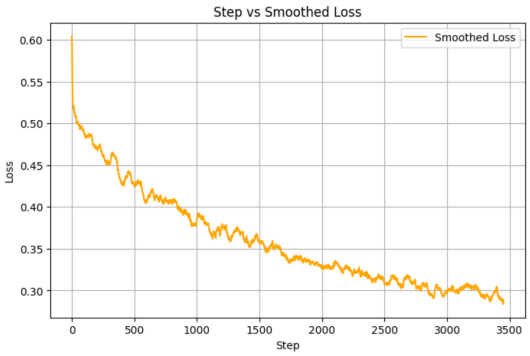


Figure 3: Training Loss for Model 2

Figure 3 demonstrates that the decrease in training loss of Model 2 was the smoothest of all, perhaps because it used a linear scheduler. By the end of training, Model 2 achieved the lowest training loss, which was very likely because it was trained the longest.

### 4.3 Model 3

Model 3 was trained for 2,000 steps, and its training loss is plotted in Figure 3.


Figure 4: Training Loss for Model 3

Model 3 has similar hyper-parameter settings as Model 1 but was trained with smaller batch sizes to address potential over-fitting or getting stuck in local minima. Indeed, figures 1 and 4 show that the decrease in training losses were quite noisy, which we were hoping would help the model better generalize.

### 4.4 Results and Analysis

The final results of our selected models are shown in Table 3 below.

| Model | Kaggle public score | Kaggle private score |
|---|---|---|
| 1 | 0.84228 | 0.83774 |
| 2 | 0.84952 | 0.84171 |
| 3 | 0.84771 | 0.83575 |

Table 3: Results of Final Models

Model 2 has the highest public score, so it was expected that it would probably have the highest private score as well. This was especially likely given that it was trained on the most data, and so had much greater opportunity to optimize its weights and generalize than the other two models. Given that the private accuracies of the other two models are within a percentage point, it's very likely that Model 1 and 3 could have outperformed Model 2 with more training.

Indeed, what stands out the most is by how little Model 2 outperformed the other two given how much more data it was trained on. This suggests that our basic intuition of how the cosine scheduler could help the model overcome obstacles related to mathematical reasoning was valid.

The results, however, don't give much support to our supposition that the model was getting stuck in local minima. This is because Model 3 (with lower batch size) has a marginally lower private score despite being trained on significantly more steps than Model 1, which differed only by having a greater batch size. Of course, we would need to train Model 1 and 3 a lot more to come to any concrete conclusions on this.

# 5 Scope for improvement

Limited time and compute were important factors in the extent to which we could trial various configurations of our models. But in retrospect, a few things could have been done differently.

## 5.1 Model validation strategies

In this project, our strategy was to train models as quickly as possible to experiment with various combinations of hyper-parameters. As a result, we primarily relied on training losses and Kaggle scores to evaluate our models. While this approach allowed us to train many models efficiently, it presented challenges in identifying when models were over-fitting and determining the optimal number of training steps for each model. Consequently, fine-tuning became difficult, as we were unable to significantly improve model performance by adjusting hyper-parameters alone.

To enhance the fine-tuning process in future experiments, we would incorporate validation sets during training. This would provide a clearer indication of over-fitting, help us determine the appropriate point to stop training and enable us to make better judgments on how to adjust our hyper-parameters.

## 5.2 Prompt engineering

In our experiments, we only used one prompt, albeit one that incorporated various prompting techniques. One improvement would be to experiment with more prompts with varying emphasis and implementations of those techniques, and see how they affect performance.

## 5.3 Larger data and LoRA hyper-parameters

Time and resource constraints limited us to training only on a small part of the dataset. However, our best model shows that there is potential for improvements by training models with more data. Therefore, an important future improvement would involve leveraging more data during training and utilizing larger ranks for LoRA, provided additional computational resources are available.

## 5.4 QLoRA

One way to get around compute limitations would have been to use Quantized LoRA adapters (Lambert et al., 2023). QLoRA decreases the precision of the model's weights from 16-bit to 4-bit, which reduces memory consumption during training. It's also been shown not to sacrifice performance significantly. Using QLoRA would've enabled us to support larger batches, potentially leading to better performance.

# 6 Conclusion

In this project, we fine-tuned Llama 3.1 8B to grade answers to math questions. Our experiments focused on optimizing hyper-parameters, including effective batch sizes, learning schedulers, and learning rates. Ultimately, we submitted three models that achieved an accuracy of 0.83–0.84 in the Kaggle competition. Through this process, we gained valuable insights into the impact of various hyper-parameters, the systematic approach to tuning them, and the critical role of frequent model validation in improving performance and preventing over-fitting.

## 7 Models and Notebook

The links to the GitHub Repo containing our notebook and the google drive folder containing our models are embedded. Note that you will need an nyu email to access the Google Drive.

## References


Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *CoRR*, abs/2106.09685.

Damjan Kalajdzievski. 2023. A rank stabilization scaling factor for fine-tuning with lora. *arXiv preprint arXiv:2312.03732*.

Maxime Labonne. 2024. Fine-tuning llama 3 with supervised fine-tuning (sft). https://huggingface.co/blog/mlabonne/sft-llama3. Hugging Face Blog.

Alexander Lambert, Saba Dadi, Tim Dettmers, Mikel Artetxe, Fahim Khan, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*.

Jun Li, Wei Zhang, Xiaohui Chen, et al. 2023. Chatgpt is not all you need: Enhancing large language models with symbolic reasoning. *arXiv preprint arXiv:2305.18654*.

Tiedong Liu. 2024. Goat: A generative model for analysis and tasks. https://github.com/liutiedong/goat. GitHub repository.

Thomas Loeber. 2024. Everything we know about llms. Substack article.

Sean McLeish, Arpit Bansal, Alex Stein, Neel Jain, John Kirchenbauer, Brian R. Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, Jonas Geiping, Avi Schwarzschild, and Tom Goldstein. 2024. Transformers can do arithmetic with the right embeddings. *arXiv preprint arXiv:2405.17399*.

Venkatesh Balavadhani Parthasarathy, Ahtsham Zafar, Aafaq Khan, and Arsalan Shahid. 2024. The ultimate guide to fine-tuning llms from basics to breakthroughs: An exhaustive review of technologies, research, best practices, applied research challenges and opportunities.

Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, et al. 2024. The prompt report: A systematic survey of prompting techniques. *arXiv preprint arXiv:2406.06608*.

Hugging Face Team. 2024. *LoRA Training with Hugging Face Diffusers*. Documentation page.


## A Appendix

### A.1 Prompt for Math Grading Task

"You are a math grader tasked with evaluating whether a given answer to a math question is correct or not. Respond with 'True' if the answer is correct and 'False' if it is incorrect.

Below are the Question, the given Answer, and the Explanation of the Answer.
### Question: {}
### Answer: {}
### Explanation: {}

It's very important to grade the Answer accurately, so you must:
1. Carefully read and understand the Question.
2. Review the given Answer and compare it against the Explanation provided.
3. If the Explanation correctly justifies the Answer, respond with 'True'.
4. If the Explanation is incorrect or does not logically support the Answer, respond with 'False'.

### Grading ('True' or 'False'): {}"

## A.2 Table 1: Models Tested on Kaggle

| Rank | Alpha | Warm-up | Batch size | Accu. steps | Weight decay | Scheduler | Learning rate | Steps trained | Kaggle public score |
|---|---|---|---|---|---|---|---|---|---|
| 32 | 64 | 50 | 16 | 2 | 0.01 | linear | $1e^{-4}$ | 800 | 0.83363 |
| 32 | 64 | 50 | 8 | 4 | 0.0001 | linear | $1e^{-4}$ | 800 | 0.83061 |
| 32 | 64 | 5 | 8 | 4 | 0.001 | linear | $1e^{-4}$ | 1200 | 0.83725 |
| 32 | 64 | 50 | 8 | 4 | 0.001 | cosine | $1e^{-5}$ | 1200 | 0.79621 |
| **32** | **64** | **5** | **8** | **4** | **0.001** | **cosine** | $\mathbf{1e^{-4}}$ | **1200** | **0.84228** |
| 32 | 64 | 5 | 8 | 4 | 0.001 | cosine | $1e^{-4}$ | 1500 | 0.83967 |
| 32 | 64 | 5 | 8 | 4 | 0.001 | cosine | $3e^{-4}$ | 1500 | 0.84047 |
| **32** | **64** | **5** | **8** | **4** | **0.001** | **linear** | $\mathbf{1e^{-4}}$ | **3500** | **0.84952** |
| 32 | 64 | 50 | 4 | 4 | 0.001 | cosine | $1e^{-4}$ | 1500 | 0.84590 |
| 32 | 64 | 50 | 4 | 2 | 0.001 | cosine | $1e^{-4}$ | 2000 | 0.832826 |
| **32** | **64** | **50** | **4** | **4** | **0.001** | **cosine** | $\mathbf{1e^{-4}}$ | **2000** | **0.84771** |
| 32 | 32 | 50 | 4 | 4 | 0.001 | cosine | $1e^{-4}$ | 800 | 0.81734 |

Table 1: Models Tested on Kaggle

The 3 models selected for submissions are bold in the table.