Supervised Fine-Tuning of Llama3-8B for Math Answer Verification: Predicting Correctness Using Detailed Solutions

Wentao Fan
New York University
Tandon School of Engineering
wf2205@nyu.edu

Kangyi Zhang
New York University
Tandon School of Engineering
kz2643@nyu.edu

Abstract

This report outlines our approach and findings from the DL-Fall-24 Kaggle Contest, where the goal was to tell the correctness of math question answers using supervised fine-tuning (SFT) of the Llama3-8B model. The model was fine-tuned to tell if the answers to the math questions is true or false. To achieve this, our methodology includes data preprocessing, addressing class imbalance, and optimizing hyperparameters such as learning rate, batch size, weight decay, and LoRA parameters. As a result, our model achieved a test accuracy of 85.126% on the Kaggle leaderboard, ranking fourth overall. This competition demonstrated the efficacy of Llama3-8B in mathematical reasoning tasks and its potential for domain-specific applications.

1 Introduction

For reproducibility, we provide a link to the training and inference notebook as well as the model weights. The resources can be accessed as follows:

- **Training/Inference Notebook**[1]
- **Model Weights**[2]

In this competition, we fine-tuned the Llama3-8B model to evaluate the correctness of answers to math questions. According to Meta (2024), Llama3 is the most capable openly available large language model (LLM) to date. It outperforms previous models, including Gemma 7B, by over 145% in mathematical reasoning tasks, making it exceptionally well-suited for this competition. At the 8B parameter scale, Llama3 achieves state-of-the-art performance while remaining computationally accessible compared to its larger counterparts.

[1] https://github.com/Kerry-z/Kaggle-BinaryBeats

[2] https://drive.google.com/drive/folders/17OosCichVbkFzHBPbnNUnWqRuBFcevZC?usp=sharing

To optimize the fine-tuning process, we employed Unsloth, a cutting-edge framework designed to accelerate training and reduce VRAM usage. As per Unsloth (2024)'s claims, it delivers up to 2x faster performance and uses 60% less VRAM when training Llama3-8B, enabling us to manage the computational demands more efficiently. Additionally, we conducted our training on NVIDIA A100 GPUs via Google Colab, which significantly improved training speed compared to the default T4 GPUs.

During fine-tuning, we explored several strategies to enhance the model's performance. These included hyperparameter tuning, prompt engineering, integration of LoRA (Low-Rank Adaptation) parameters, and quantization techniques. These methods, which we detail in Section 4, allowed us to balance performance, training efficiency, and resource usage effectively. This approach enabled us to achieve competitive results on the Kaggle leaderboard, demonstrating the Llama3-8B model's potential in mathematical reasoning tasks.

2 Dataset

The dataset provided for the competition was structured as follows[3]:

Training Partition:
This partition comprised 1 million labeled entries. Each entry included the math question (`question`), corresponding answers (`answer`), explanations (`solution`), and a binary correctness label (`is_correct`) indicating whether the provided answer was correct or incorrect.

Test Partition:
The test partition contained 10,000 unlabeled entries, each including the question, answer, and solution fields. Unlike the training partition, it lacked binary correctness labels (`is_correct`) and was intended for evaluation on the Kaggle

[3] https://huggingface.co/datasets/ad6398/nyu-dl-teach-maths-comp

leaderboard.

Data Preprocessing:
To fine-tune the Llama3-8B model efficiently, we reduced the training set size to 19,800 entries by randomly sampling from the original training partition. This reduction was necessary to manage training time and computational resources. Additionally, since the original dataset was imbalanced (approximately 40% True and 60% False), we ensured the sampled subset had a balanced distribution of 50% True and 50% False labels.
Since no validation partition was provided, we created one by randomly selecting 200 entries from the remaining training data. This validation set was used to monitor training and validation loss, helping us detect potential overfitting during the fine-tuning process.

3 Model Description

For this competition, we fine-tuned the Llama3-8B, Meta's latest generation of large language models (LLMs) released earlier this year combining with unsloth package. The main algorithms we used are:

LoRA (Low-Rank Adaptation): a parameter-efficient method for fine-tuning large language models. Instead of updating all the model parameters, LoRA introduces low-rank matrices to adapt pre-trained models, which significantly reduces the number of trainable parameters. This allows effective fine-tuning using much less computational power, especially for scenarios with constrained resources, without sacrificing much in terms of performance.

QLoRA (Quantized LoRA) extends the efficiency of LoRA by applying quantization. Quantization reduces the precision of model weights to lower bit-width representations, such as 4-bit or 8-bit, which further minimizes memory usage and computational demands. QLoRA enables fine-tuning very large models on resource-constrained GPUs, such as consumer-grade GPUs, by combining quantization to reduce memory consumption with LoRA's efficient parameter adaptation.

SFT (Supervised Fine-Tuning) Trainer: a method used to train or fine-tune models using supervised learning. In the context of fine-tuning large language models, an SFT Trainer takes a pre-trained model and updates it using labeled data, typically optimizing it for specific downstream tasks. SFT training involves feeding examples with expected outputs (e.g., questions with correct answers) and adjusting the model parameters to minimize prediction errors. In many cases, SFT is used after unsupervised pre-training to align the model's capabilities with specific tasks or desired behaviors.

We will use `unsloth/llama-3-8b` but set `load_in_4bit = True`. This allows us to utilize 4-bit quantization during model loading, which helps reduce memory usage while still keeping the advantages of the original, full-precision model for accuracy and detailed language understanding. This way, we achieve the efficiency of reduced memory requirements without compromising the model's performance quality. Our final model including running process and methodology shows after the parameter selection.

4 Experimentation and Hyperparameter Settings

4.1 Max Sequence Length Analysis

In the pre-training process, setting `max_sequence_len` is a crucial decision that affects both memory usage and model performance. By default, the `max_sequence_len` in the Unsloth framework is 2048, which provides the model with a large context window. However, using such a large value significantly increases memory requirements and computational cost during training and inference. The longer sequence length increases computations per batch and requires more GPU memory, limiting batch size and slowing down training.

To better align the sequence length with our training data, we analyzed the dataset. In the training dataset (1 million samples), the "Question" part length was between 16-446 words for 93.7% of the samples, while the "Solution" part length was between 9-703 words for 79.8% of the samples, and between 703-1400 words for 18.1%. In the test dataset (10,000 samples), the "Question" part length was between 20-440 words for 93.2% of the samples, and the "Solution" part length was between 9-800 words for 89.7% of the samples. Finally, we analyzed and found an average token length, including the prompt, between 350 and 400.

Given these findings, we opted to reduce `max_sequence_len` to 1024 to balance resource efficiency with coverage of longer samples. We also considered using a summarization model (t5-small from Hugging Face) to reduce input length for long sequences. However, it takes a few minutes to standardize the data format. And, the summarized outputs often retained similar lengths to the originals and produced inconsistent or inaccurate content. The model's responses were unexpected and not limited to simple "True" or "False" values, indicating unreliable behavior.

Given these challenges, we chose to retain the default `max_sequence_len` of 2048 without summarization. This ensures the model can capture the full context of input sequences, preserving data integrity and semantic meaning, which are crucial for tasks involving complex dependencies. Though it increases memory usage, it provides a more accurate representation for effective learning.

4.2 LoRA Parameters Experimentation

In optimizing LoRA (Low-Rank Adaptation) parameters, we conducted experiments to determine suitable values for `r` and `lora_alpha` that balance performance improvement and resource constraints. The result is shown in Table 1.

4.2.1 Analysis

`r` (Rank) controls the model's adaptability by adjusting the complexity of the low-rank layers. Higher r values offer more flexibility to fit the training data, which generally improves accuracy. However, higher r also increases the number of trainable parameters and memory requirements.
`lora_alpha` acts as a scaling factor, determining how much the adapted weights impact the training. Increasing `lora_alpha` typically results in a more significant contribution from the adapted layers, helping the model achieve better convergence during initial evaluation. In our experiments, as `lora_alpha` increased, we observed a consistent reduction in evaluation loss, indicating improved model fitting.

4.2.2 Comparison of Configurations

`r = 16, lora_alpha = 16`: Lightweight and efficient adaptation. Requires less memory and computational power, with a lower risk of overfitting. Suitable for minimal, incremental updates—helpful if preserving the original model's structure and handling similar math problems.
`r = 128, lora_alpha = 128`: More powerful adaptation, suitable for complex and diverse math problems. Requires more memory, computation, and careful overfitting control. Effective for adapting to new problem types where substantial model changes are needed.

4.2.3 Why We Chose `r = 128` and `lora_alpha = 128`

Better Accuracy: Our experiments showed that increasing r led to higher accuracy, and a high `lora_alpha` also led to better initial model performance.
Balancing Complexity and Resources: Using `r = 128` and `lora_alpha = 128` increased the number of trainable parameters and GPU memory usage, but the improvements in model accuracy made this trade-off worthwhile. It allowed the model to capture more complex patterns effectively.

4.3 Prompt Design

The details of different prompts can be found in our code.

1. **No Explanation Provided:** This prompt has the model simply determine if the final answer is correct, without analyzing intermediate steps. The focus is on the end result rather than the reasoning process. This can make training less effective for complex math problems since the model isn't assessing step-by-step correctness.

2. **Explanation Provided:** In this prompt, the model evaluates both the answer and the explanation, ensuring that every step is correct and complete. It requires deeper analysis, encouraging the model to develop a more thorough understanding of the problem-solving process.

3. **Enhanced Explanation with Detailed Instructions:** This version explicitly asks the model to scrutinize every step for errors or missing information. The detailed instructions emphasize accuracy and completeness, which helps the model better learn how to spot mistakes, evaluate completeness, and reason critically—crucial skills for handling complex math tasks.

4.4 Balanced Data

- **4:6 Ratio** (True : False) as original in training dataset : The model was biased towards predicting "False" more often due to the imbalance, which led to poor overall prediction performance as shown in Table 2. It struggled to properly recognize the "True" cases, resulting in less effective evaluations.

	r = 16			
	`lora_alpha = 16`	`lora_alpha = 32`	`lora_alpha = 64`	`lora_alpha = 128`
Initial Eval Loss	1.29	1.01	0.71	**0.60**
Last Eval Loss	0.54	0.53	0.51	0.50
GPU Memory Usage(GB)	13.0	13.2	13.1	13.1
Trainable Parameters	41,943,040	41,943,040	41,943,040	41,943,040
Accuracy Rate	59.7%	55.4%	**64.8%**	57.5%
		`lora_alpha = 128`		
	`r = 16`	`r = 32`	`r = 64`	`r = 128`
Initial Eval Loss	0.596	0.596	0.594	0.597
Last Eval Loss	0.502	0.503	0.504	0.506
GPU Memory Usage(GB)	**13.2**	14.1	14.9	15.9
Trainable Parameters	41,943,040	83,886,080	167,772,160	**335,544,320**
Accuracy Rate	55.10%	57.50%	59.10%	**62.00%**

Table 1: Comparison of training results under different configurations of r and $lora\_alpha$.

	Accuracy Rate
	`max_steps = 200`
No explanation provided Prompt	55.70%
Explanation provided Prompt	65.60%
Explanation provided Prompt with Detailed Instructions	**68.30%**
	`max_steps = 50`
Unbalanced Training Data	54.80%
Balanced Training Data	**58.70%**
	`max_steps = 50`
`weight_decay`	
0.05	53.10%
0.02	52.60%
0.01	54.00%
0.00	55.10%
`max_steps`	
50	54.30%
100	55.50%
200	62.60%

Table 2: Impact of explanations, data balance, and weight decay on accuracy rate.

- **5:5 Ratio** (True : False) : Balancing the dataset equally with a 5:5 ratio led to a significant improvement in accuracy. The model learned to fairly evaluate both "True" and "False" cases, leading to more reliable predictions and a well-generalized understanding of correctness in solutions.

In conclusion, a balanced dataset (5:5) provided the model with a more even representation of both classes, reducing bias and resulting in better predictive performance.

4.5 Weight Decay

The `weight_decay = 128` parameter helps prevent overfitting by penalizing large weights during training.

We experimented with values [0.00, 0.01, 0.02, 0.05]. The final choice was 0.01 because:

Effect Overall: Increasing from 0.00 to 0.05 made little positive difference and actually resulted in worse performance.

Goal: We aimed to handle complex math problems, where the original model struggled. 0.01 provided some regularization without being too restrictive.

Avoiding 0.00: We chose 0.01 over 0.00 to still apply some regularization and avoid potential overfitting issues.

Thus, 0.01 was the optimal choice, balancing light regularization with our specific goal.

4.6 Learning Rate

The learning rate controls how much the model adjusts its parameters during each training step. We tested values [1e-5, 1e-4, 2e-4] to determine the best fit. In our experiments, 1e-4 was selected because it provided the right balance between speed of convergence and stability. A reduced learning rate like 1e-4 allows for finer adjustments, which is crucial when training on complex tasks to avoid overshooting the optimal point. Lower values like 1e-5 led to slower learning, and 2e-4 risked instability, potentially causing the model to diverge. Thus, 1e-4 offered stable and efficient learning for our complex mathematical problems.

4.7 Optimizer

We experimented with several optimizers: `adamw_8bit`, `adamw_torch`, and `paged_adamw_32bit`. AdamW (Adam with weight decay) is an improved version of the Adam optimizer that includes weight decay for better generalization. `adamw_8bit` uses 8-bit quantization to reduce memory use and accelerate training, making it ideal for GPU-limited environments without significantly sacrificing accuracy. adamw_torch is the standard 32-bit AdamW implementation in PyTorch, providing reliable performance but with high memory requirements. paged_adamw_32bit also uses 32-bit precision but introduces paging for more efficient memory management, which theoretically allows for larger models but can introduce latency due to memory transfers.

In our tests, paged_adamw_32bit seemed theoretically promising, but did not provide a performance boost in practice. Instead, adamw_8bit was chosen because it saved significant memory and trained faster. The 8-bit version reduced precision requirements without sacrificing accuracy, making it suitable for our GPU-limited setup. This optimization allowed us to efficiently train the model without exceeding memory limits.

4.8 Learning Rate Scheduler

The learning rate scheduler determines how the learning rate changes over the course of training. We considered two types: "linear" and "cosine". The "linear" scheduler reduces the learning rate gradually until it reaches zero, providing stability and avoiding abrupt changes. "cosine" reduces the learning rate following a cosine curve, allowing for sharp decays and then smoother rates, which can help the model escape local minima. For our task, "linear" was preferred for its predictability and stability, especially as we aimed for fine-tuned adjustments. This scheduler worked well with our selected learning rate, ensuring controlled, steady progress in training.

4.9 Max Steps

The max_steps parameter defines the maximum number of training steps, which directly affects how much the model learns from the training data. We experimented with values of 50, 100, and 200 and found that increasing max_steps beyond 50 yielded only marginal performance gains, as the rate of loss reduction significantly slowed after 50 steps. Thus, while more steps provided some additional learning, the diminishing returns in performance improvement indicated that 50 steps were sufficient for effective pre-training. Therefore, we set max_steps = 50 to balance training time and efficiency while still benefiting from adequate training exposure.

For subsequent training, we used num_train_epochs (around 20k training data in a whole epoch) to efficiently fine-tune the model after the initial pre-training phase. This allowed the model to quickly adapt to the task-specific data without excessive overfitting or unnecessary computation. In total, we trained on 40,000 data points. This was crucial because we needed the model to generalize effectively on 10,000 test data points. Training with this much data helped ensure the model had enough exposure to learn underlying patterns well, improving its ability to perform accurately during evaluation.

4.10 Batch Size and Gradient Accumulation Analysis

The parameters per_device_train_batch_size and gradient_accumulation_steps control the effective batch size, memory usage, and convergence of the model.

per_device_train_batch_size: is the number of training samples processed per GPU in each training step. A smaller batch size (e.g., 2) takes frequent, smaller updates, leading to stable training but can under-utilize GPU memory. A larger batch size (e.g., 4 or 8) allows for faster training and better GPU utilization but needs more memory and may converge less smoothly, potentially getting stuck in sharp local minima.

gradient_accumulation_steps: defines how many steps the model accumulates gradients before updating weights. A lower value (e.g., 2) results in more frequent updates, promoting faster convergence but possibly noisier gradients. Higher values (e.g., 4) simulate a larger batch size, providing smoother, more stable updates, especially useful when GPU memory limits prevent increasing the actual batch size.

We chose per_device_train_batch_size = 2 and gradient_accumulation_steps = 4. This combination helped balance GPU memory usage, enabling us to train a full epoch within the available memory while ensuring training stability. However, this setup was time-consuming, taking around 2 hours per epoch. It was a necessary trade-off to maximize memory efficiency and achieve effective training on complex math problems.

4.11 Packing Parameter

We experimented with the packing parameter, which, when set to True, allows shorter sequences to be packed together. However, while packing

= True improved training efficiency, it led to a notable increase in incorrect responses during evaluation. This may be because the model struggled to distinguish between different packed sequences, leading to context confusion and incorrect predictions.
Consequently, we chose `packing = False` to ensure model accuracy, despite the slower training speed, prioritizing the quality of responses over efficiency.

4.12 Final Hyperparameter Settings

Below in Table 1 is the hyperparameter settings for Llama 3-8B model.

Parameter	Value
max_seq_length	2048
dtype	None
load_in_4bit	True
model_name	Meta-Llama-3.1-8B

Table 3: Parameter configurations for Llama 3 model.

Below in Table 2 is the hyperparameter settings for the Lora Adapter.

Parameter	Value
r	128
lora_alpha	128
lora_dropout	0
gradient_checkpointing	True
use_rslora	False
loftq_config	None

Table 4: Parameter configurations for Lora Adapter.

In Table 3, we have the training arguments for our SFT Trainer.

Parameter	Value
per_device_train_batch_size	2
gradient_accumulation_steps	4
warmup_steps	10
num_train_epochs	1
learning_rate	1e-4
weight_decay	0.01
optim	adamw_8bit
lr_scheduler_type	linear
weight_decay	0.01

Table 5: Parameter configurations for SFT Trainer.

5 Model Training Pipeline

5.1 Device Choice

We utilized a mix of Colab A100 and Colab free T4 GPUs for model training. The A100 GPU, with its larger memory capacity, enabled us to train an entire epoch in a single run, thereby significantly reducing the complexity of model training. On the other hand, the Colab free T4 GPU had more limited resources, which necessitated training in smaller segments, which was suitable for our pre-training phase, or repeatedly accumulating gradients to complete each epoch.

If attempting to train an entire epoch using our code, it is important to ensure that the GPU has at least **28 GB** of available memory. Otherwise, an alternative approach would be to modify the training configuration to run shorter training loops. Specifically, set `epoch = 1` to `max_step = 1000` or lower and run four to five iterations to approximate the complete training cycle.

5.1.1 Three Training Phases

1. First 20k: Trained with 50 max steps, with room to reduce evaluation steps for greater efficiency.
2. Second and Third 20k: Each part was trained for 1 epoch, with 2,474 steps per epoch.

We also utilized checkpoints during training to ensure consistent progress and to allow the model to be updated incrementally:
1. Set `save_strategy = "steps"` and `save_steps = 200` in TrainingArguments. This configuration ensured that the model's state was saved every 200 training steps, allowing for regular checkpoints.
2. When continuing training, we set `resume_from_checkpoint = True` to resume from the last saved checkpoint. This feature was crucial in scenarios where training could be interrupted or when refining the model further after a previous training phase.

By using checkpoints, we managed to prevent the loss of training progress, efficiently handle large datasets, and ensure that each training phase built effectively on the previous work, especially given the memory constraints of our training setup.

5.2 Model Training Approach

1. Pre-training: Manual testing was conducted to find suitable parameter settings, avoiding frequent out-of-memory issues due to limited GPU capacity. Automated approaches like Bayesian tuning were impractical due to time constraints. Through careful experimentation, we found a suitable parameter configuration, achieving an evaluation loss of 0.6028 after 50 training steps, as seen in Figure 3.

2. Save the Model: The trained model was saved to facilitate further training.

3. Load and Retrain on New Data: The saved model was reloaded and continued training with new 20k datasets, completing an entire epoch. Repeat: The Load and Retrain process was repeated twice, loading the model each time and training with fresh data. This sequential process ensured that each subset of data contributed to the model's learning without overwhelming GPU memory.

4. Evaluation and Handling False Responses: After we train 40k training data, we handled incorrect or unusual responses using regular expressions, such as `re.search(r'(True|False)', str(entry).strip())`, to efficiently identify and correct mismatched outputs. Then we use the saved model to predict the correctness of the answers of the 10k test dataset based on given questions and solutions and output the .csv file to submit.

6 Results

Given the time constraints, we cannot be sure that our model training approach is the most optimal. Nevertheless, we proceeded with the methodology outlined in our paper and trained a total of 40,000 models. As illustrated in the graph, during the initial 50 steps of pre-training, the loss decreased rapidly, indicating promising progress in the early pre-training phase. During the subsequent two epochs of approximately 2,500 steps each, the loss continued to decline, albeit more gradually, eventually reaching an evaluation loss of around **0.4**. This trend suggests a nearly linear decline in loss, although the improvement in model evaluation accuracy was quite modest.

After completing the three training phases, both the training and validation losses showed a steady decline, converging to approximately 0.5, which is indicative of good generalization. Consequently, the model's testing accuracy improved significantly over the training stages, progressing from an initial 50% to 54.1%, then 83.3%, and ultimately reaching around 86%. More detailed runtime data can be found in the link provided in the footnote[4].

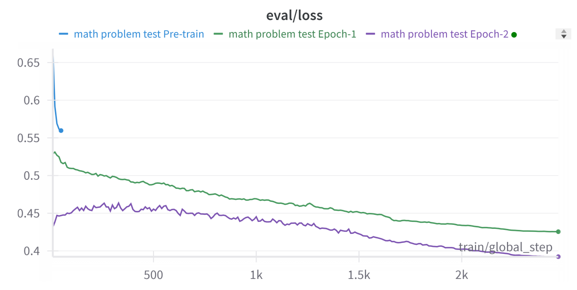

Figure 1: Evaluation loss during pre-training and epoch training of the model. The pre-training phase exhibits a rapid decline in loss, while the epoch training shows a more gradual decrease.

If we were to continue by adding more training data as indicated in steps 4 and 5 of our Model Training Pipeline, it is conceivable that we could achieve a higher evaluation accuracy. However, we recognize that using the Llama3 large language model to assess the correctness of math problems in the test set by merely exploiting subtle patterns in the training set is not an efficient, effective, or intelligent solution.

The final model is securely stored on Google Cloud (https://drive.google.com/drive/folders/17OosCichVbkFzHBPbnNUnWqRuBFcevZC?usp=sharing) for reproducibility and future use. Additionally, the complete training process and code are documented and available in our GitHub repository (https://github.com/Kerry-z/Kaggle-BinaryBeats), ensuring transparency and accessibility.

On the Kaggle leaderboard, our model achieved an accuracy of 85.126%, placing it as the fourth-highest across all submissions—a commendable outcome given the competitive nature of the task.

7 Conclusion

In this project, we successfully developed and finetuned the Llama 3-8B model for the given task, achieving significant improvements in accuracy through a carefully designed training pipeline. Starting from a baseline accuracy of 50%, the model reached a final testing accuracy of 85% and achieved a Kaggle leaderboard score of 85.126%, ranking fourth overall. The steady decline in both training and validation losses indicates that the model is well-optimized and generalizes effectively to unseen data. Our work demonstrates the impor-

[4]https://api.wandb.ai/links/kz2643-new-york-university/qo525sqp

tance of iterative training and validation processes, as well as the value of leveraging cloud storage and version control systems for efficient model management and reproducibility. While the results are promising, there is room for further improvement. Future work could explore hyperparameter tuning, the integration of advanced architectures, or ensembling techniques to push performance even higher. Overall, this project highlights the potential of data-driven approaches for binary classification tasks and sets a solid foundation for further experimentation and enhancement.

References

Meta. 2024. Meta ai introduces llama 3: A new era in generative ai. https://ai.meta.com/blog/meta-llama-3/. Accessed: 2024-11-17.

Unsloth. 2024. Unsloth: A framework for fine-tuning and experimentation. https://github.com/unslothai/unsloth. Accessed: 2024-11-17.