

Mixed Precision fine-tuning and Parallel Training for LLM

Abstract: In this project, we investigate efficient fine-tuning and parallel optimization strategies for enhancing the performance of the Llama 3.2 1B large language model (LLM) on token generation tasks. Given the model’s limited scale and resource efficiency, we adopt **Low-Rank Adaptation (LoRA)** to inject lightweight trainable matrices into attention layers, enabling specialization without retraining the full model. To accelerate training and reduce memory footprint, we explore **mixed precision training** (FP16/BF16), as well as two distributed training techniques—**Data Distributed Parallelism (DDP)** and **Fully Sharded Data Parallelism (FSDP)**—to analyze their effectiveness across throughput, memory usage, and scalability. We further enhance the preprocessing pipeline by integrating **Dask parallelization** with SLURM-based HPC resources, reducing dataset tokenization time from 80 to 36 minutes for the 3M-sample OpenOrca dataset. To evaluate real-world usability, we encapsulate the model in a **Gradio-based chatbot interface**, mimicking ChatGPT’s interactive behavior. We got good performance for token generation task. DDP proves more efficient than FSDP on mid-scale hardware, offering higher throughput and lower memory usage. This study presents a practical, scalable, and cost-effective blueprint for optimizing compact LLMs under realistic computational constraints.

Keywords: LLMs, Llama, Fine-tune, Parallelism

Contents

| | |
|--|----|
| 1. Introduction | 3 |
| 2. Llama 3.2 1B | 5 |
| 2.1 The structure of Llama 3.2 1B model | 5 |
| 2.2 The advantage of Llama 3.2 1B | 5 |
| 2.3 The disadvantage | 6 |
| 3. Purpose: | 6 |
| 4. Fine tune | 7 |
| 4.1 Full train v.s. fine-tune | 7 |
| 4.2 Low Rank Approximation | 8 |
| 4.3 Low Rank work Flow | 8 |
| 4.4 Low rank modification in the model | 9 |
| 5. Datasets and Data Process | 11 |
| 5.1 Datasets | 11 |
| 5.2 Tokenizer | 11 |
| 5.3 Dask Parallelization Acceleration | 11 |
| (1) Read configuration information | 12 |
| (2) Configure a SLURM distributed Dask cluster | 12 |
| (3) Main processing flow | 12 |
| 6. Optimization: | 14 |
| 6.1 Evaluation Metrics | 14 |
| (1) Speed Evaluation | 14 |
| (2) Memory Usage | 14 |
| (3) Quality | 14 |
| 6.2 Speed Optimization | 15 |
| 6.2.1 Mixed Precision | 15 |
| 6.2.3 Data Distributed Parallelism (DDP) | 22 |
| 6.2.4 FSDP | 26 |
| 6.2.5 DDP V.S. FSDP | 31 |
| 7. Gradio UI | 33 |
| 8. Future Work | 34 |
| 9. Reference | 41 |

1. Introduction

In recent years, with the rapid development of large language models (LLMs), the field of natural language processing (NLP) has made remarkable progress. Large language models based on the Transformer architecture continue to emerge, from the early BERT and GPT-2 to the later Llama and the now widely used GPT-4, which are getting better at text understanding, inference, generation and other tasks. This trend has also sparked our interest in studying large language models. Llama (large language model meta-AI) family of transformer-based models has become a cornerstone in the open-source LLM community, offering competitive performance with fully accessible weights. Llama has many variants, some variants represent a lightweight version suited for research, prototyping, and edge deployment scenarios. Its relatively small size allows for experimentation and customization for task-specific fine-tuning. Among these models, Llama 3.2 1B—released in September 2024—stands out as a compact, high-efficiency transformer model designed to achieve strong performance while operating under tighter computational constraints. With just over one billion parameters, it strikes a balance between power and scalability, making it especially suitable for academic research and deployment in resource-limited environments.

Despite its advantages, Llama faces practical challenges in both training efficiency and token generation quality in token generation tasks, there are key issues:

- Reduced output quality, where generated text may lack coherence, diversity, or contextual accuracy—particularly in multi-turn conversations or complex prompts.
- Underutilized parallelism, limiting scalability during training on multi-GPU systems.

These problems are especially significant for researchers and developers working with small-scale hardware setups, where maximizing efficiency is essential.

The core objective of this project is to explore an efficient LLM fine-tuning and acceleration solution suitable for medium-sized hardware environments. We carry out the work in the following several directions:

- Fine-Tuning: By adopting the LoRA (Low-Rank Adaptation) technology, on the basis of freezing the weights of the original model, only a small number of trainable matrices are injected. Thus, while significantly reducing the training parameters and GPU memory overhead, the adaptability of the model on new tasks is improved.
- Training efficiency optimization: Introduce mixed-precision training (including FP16 and BF16 modes) to enhance throughput and reduce computational load; In the multi-GPU setting, two Distributed strategies, DDP and FSDP are used respectively to compare their performances in terms of speed, memory and scalability, and optimization suggestions are put forward.
- Data processing acceleration: When dealing with the large-scale dialogue dataset OpenOrca containing about 3 million samples, we use the Dask distributed parallel framework to segment and concurrently schedule the tokenizer stage, and deploy it on the school's HPC cluster in combination with the SLURM scheduling system. Effectively reduce the data preprocessing time from the original 80 minutes to within 36 minutes.
- User experience improvement: To better align with real application scenarios like ChatGPT, we have encapsulated the model interaction interface based on Gradio and implemented a web chat system with a complete process of "system instructions + user questions + model responses". This system allows users to input questions online, adjust the generated parameters (such as temperature, top-p, max tokens), and obtain the model response instantly, significantly improving the practicability and testability of the model after fine-tuning.

Through this project, we verified the performance improvement capability of LoRA fine-tuning on the Llama model, explored the adaptability of various distributed acceleration strategies, and proposed a complete closed-loop LLM optimization process from training, inference to deployment.

Model Link: <https://huggingface.co/meta-Llama/Llama-3.2-1B>

Datasets Link: <https://huggingface.co/datasets/Open-Orca/OpenOrca>

Test Results Link:

https://drive.google.com/drive/folders/1W773FCvghvQyB96pNVxV9KOaDZltp5UJ?usp=drive_link

2. Llama 3.2 1B

The Llama 3.2 1B model was released in September 2024 [1]. It is a transformer-based large language model designed for efficient performance on a wide range of tasks while maintaining a smaller size compared to other state-of-the-art models.

2.1 The structure of Llama 3.2 1B model

The structure of the Llama 3.2 1B model is composed of three main components:

(1) Embedding Parts: After tokenization, input text is transformed into a sequence of integers that correspond to positions in the vocabulary. These integers have no intrinsic semantic meaning. The embedding layer converts these indices into dense vectors that contain meaningful semantic information.

(2) DecoderLayer: This is the core computational block of the model. It contains Multi-queried Attention, RMSNorm, Rotation positional embedding, and MLP (Feedforward Layer).

1. Multi-queried Attention: A key innovation in Llama 3.2 1B that captures dependencies within the input sequence. It helps the model focus on the most relevant parts of the input.
2. RMSNorm: A normalization technique that stabilizes training, shown to perform better than traditional layer normalization in certain contexts.
3. Rotation positional embedding: Convert the word embedding tensors into the positional embedding vectors which contains the information about the word in the sentence. Differentiate with usual positional embedding, the rotation embedding is dynamical, parameters are learnable.
4. MLP (Feedforward Layer): Utilizes a gating mechanism to selectively propagate relevant information.

(3) Output Linear Layer: Projects the final decoder representations into the vocabulary space to generate output token logits.

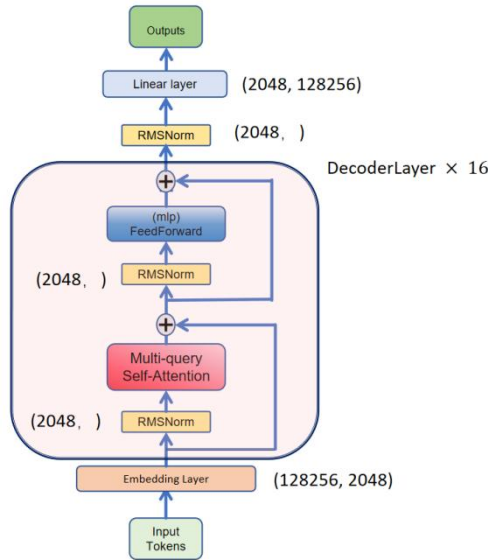


Figure 1. Llama 3.2 1B structure.

In this project, we apply structural modifications to key components—especially the attention layers—to enhance performance without sacrificing the model’s computational efficiency.

2.2 The advantage of Llama 3.2 1B

Here are advantages of Llama 3.2 1B model:

- (1) Delivers high computational efficiency with approximately 1 billion parameters, making it suitable for research and deployment in resource-constrained environments.
- (2) Achieves strong performance on standard natural language processing (NLP) benchmarks despite its smaller size.
- (3) Optimized for scalable fine-tuning, allowing effective adaptation to specific tasks without requiring extensive hardware resources.

2.3 The disadvantage

However, the model is not perfect, it has some defects and issues needed to be fixed.

- (1) May exhibit suboptimal performance on highly specialized or domain-specific tasks without targeted fine-tuning.
- (2) Has a limited capacity for handling extended context lengths, which can hinder performance on long-document tasks compared to larger models.
- (3) Relies on advanced optimization techniques—such as parallelism and quantization—to maximize hardware utilization and training efficiency.

3. Purpose:

Despite its impressive capabilities, the Llama 3.2 1B model is not universally effective across all domains. General-purpose LLMs often struggle with tasks that require domain-specific knowledge or reasoning, such as:

- Mathematical problem-solving
- Medical question answering
- Understanding and summarizing long documents

To improve the model's performance on such tasks, two main strategies are commonly used:

- Full Model Training: Retrain the entire model from scratch using a new dataset.
- Fine-Tuning: Adapt a pretrained model by training on a specific dataset, updating only some or all parameters.

In this project, we choose fine-tune method, and we adopt Low-Rank Adaptation (LoRA) as our fine-tuning method. LoRA enables efficient training by injecting small, trainable matrices into the model while keeping the original weights frozen. This approach significantly reduces memory and compute requirements while still allowing the model to specialize on new tasks effectively.

However, training the LLM will spend a lot of time and resources. To further accelerate and optimize the fine-tuning process, we integrate multiple advanced training techniques, including:

- Mixed Precision Training (FP16, BF16): Reduces memory footprint and speeds up computation without sacrificing performance.
- Data Distributed Parallelism: Enables parallel training across multiple GPUs by distributing data batches.
- Fully Sharded Data Parallelism: Minimizes memory usage by sharding model weights and optimizer states across devices.

These combined strategies allow us to adapt the Llama 3.2 1B model to new tasks efficiently and economically, while also significantly improving training speed and hardware utilization.

4. Fine tune

4.1 Full train v.s. fine-tune

While large language models like Llama 3.2 1B are highly capable, they are not universally effective across all domains. Tasks such as medical question answering, mathematical reasoning, or long-document summarization often require specialized adaptation. For the Llama 3.2 1B model, it may perform not well on some math problems and QA datasets. So we need to improve the model performance on specific datasets.

To improve performance in such areas, there are two primary strategies:

- Totally full training the model
- Fine-tuning the model.

Full training may be the method with the biggest performance improvement. But full training method has some problems that can not be ignored.

(1) Extremely High Computational Cost

LLMs typically contain over a billion parameters. For Llama 3.2 1B model, it has about 1 billion parameters. Based on Chinchilla scaling law[1]

$$\text{Optimal Training Tokens} \propto \text{Model Parameters},$$

for each parameter, it needs about 20-30 tokens. So for better performance, the minimum training tokens is about 30B for the Llama 3.2 1B model. However, in practice, we want higher performance, so usually for 1B model, it will need about 300B tokens totally.

So based on standard rough formula for compute[1]

$$\text{Total FLOPs} \approx 6 \times (\text{Model Parameters}) \times (\text{Training Tokens}),$$

the number of total FLOPs

$$\text{Total FLOPs} = 6 \times 10^9 \times 3 \times 10^{11} = 1.8 \times 10^{21} \text{ FLOPs}.$$

If we consider using NVIDIA H100 80GB GPU[2], the main compute modes are as follows

| Mode | Performance (per GPU) |
|----------------------|-----------------------|
| FP32 | 67 teraFLOPs |
| FP16 Tensor Core | 1,979 teraFLOPS |
| BFLOAT16 Tensor Core | 1,979 teraFLOPS |
| TF32 Tensor Core | 989 teraFLOPS |
| FP8 Tensor Core | 3,958 teraFLOPS |

And we consider using mixed precision i.e. FP16 or BF16, the performance is 1,979 teraFLOPS. However, you can not reach the peak of the performance, in practice, benchmarks[1,3] show effective usage ~25% – 40% of theoretical for full pipeline training (forward+backward+optimizer). So the effective about 500 TFLOPs for practical Transformer pretraining. The total time is

$$\text{Total time} = \frac{1.8 \times 10^{21}}{5 \times 10^{14}} = 3.6 \times 10^6 \text{ sec} \sim 41.7 \text{ days}.$$

It's too long! If we consider using 8 H100 GPUs, the total time is

$$\text{Total time} = \frac{1.8 \times 10^{21}}{4 \times 10^{15}} = 4.5 \times 10^5 \text{ sec} \sim 5.2 \text{ days}.$$

It's still too long! It will expend a lot of money and time! So we hope to use a more economical way!

(2) Risk of Forgetting

Because we full train the model, so the model weights will be totally updated to the new weights. It will perform better on the new datasets. However It overwrites previous representations (knowledge) it

learned from the old, large, diverse pretraining. If the new datasets is narrower (not as wide as the original 2T-token Llama training data), the model specializes too much.

So we hope to consider more economical and no harm to previous learn way to adjust the model, then we can use fine-tune.

Instead of totally training the model, we starts with a pretrained model and train a little parts of the model on the new datasets. It can make it better for the specific task and save more resources.

There are different types if fine-tuning, I simply summary them at following table

| Type | What It Means | Pros | Cons |
|-------------------------------|---|------------------------|--------------------------------|
| Full Fine-tuning | Train all model weights again on new data | Very powerful | Expensive, risk of forgetting |
| LoRA (Low-Rank Adaptation) | Train tiny new matrices injected into the model | Fast, memory efficient | Less powerful than full tuning |
| Adapter Layers | Add small new layers inside the model | Modular, efficient | Adds slight latency |
| Prefix Tuning / Prompt Tuning | Only train soft prompts injected into the input | Super lightweight | Only good for specific tasks |

Considering the Llama 3.2 1B model, we consider using Low rank as the method to fine-tune the model.

4.2 Low Rank Approximation

In math, rank represents the maximum independent rows or columns in the matrix. Assume we have an matrix M, matrix $M \in \mathbb{R}^{m \times n}$, then $\text{rank}(M) \leq \min(m, n)$.

In deep learning, we can use rank to represents the independent information. We consider the weights matrix $W \in \mathbb{R}^{\text{in} \times \text{out}}$, we can consider using two small matrices to decompose it

$$W \approx A \times B,$$

where $A \in \mathbb{R}^{\text{in} \times r}$, $B \in \mathbb{R}^{r \times \text{out}}$, $r \ll \min(\text{in}, \text{out})$. This is mathematical expression of Low Rank[4]. Therefore, we reduce the independent information by using tow small matrices decompose. And low rank approximation can decrease the number of learnable parameters. The number of learnable parameters in original weights $W \in \mathbb{R}^{\text{in} \times \text{out}}$ is

$$\text{in} \times \text{out}.$$

After using low rank approximation, the learnable parameters is

$$r \times (\text{in} + \text{out}).$$

For example, we use low rank to modify the q matrix in Multi-queried Attention in Llama 3.2 1BMode, the number of learnable parameters we taken different r shows at following tables.

| | Parameters | Proportion |
|--------|--------------------------|------------|
| q_proj | 2048×2048 | 100% |
| r = 1 | $1 \times (2048 + 2048)$ | ~0.1% |
| r = 2 | $2 \times (2048 + 2048)$ | ~0.2% |
| r = 4 | $3 \times (2048 + 2048)$ | ~0.3% |
| r = 8 | $4 \times (2048 + 2048)$ | 0.8% |

Only a few parameters are learnable! It can help us save more computation resources!

4.3 Low Rank work Flow

To integrate low-rank modifications into the model, we adjust the internal structure of specific components by introducing trainable parameters while keeping the majority of the original weights frozen. This ensures the model retains its pretrained knowledge while efficiently adapting to new data.

$$\text{Total weights} = \text{pretrained weights} + \text{Low rank weights.}$$

The workflow proceeds as follows:

- **Freeze Pretrained Weights:** All original model parameters are kept fixed to preserve prior knowledge.
- **Inject LoRA Modules:** Low-rank matrices are inserted into selected layers (e.g., q_proj, v_proj, o_proj), introducing new trainable parameters.
- **Forward Pass:** The model processes input tokens, combining outputs from both the original and LoRA-injected components.
- **Loss Calculation:** The output is compared with target labels using a loss function, such as cross-entropy.
- **Backward Pass and Optimization:** Gradients are computed only for the LoRA parameters, and an optimizer (e.g., Adam) updates these weights accordingly.

This selective training approach allows for efficient fine-tuning with minimal computational cost, while preserving the generalization capabilities of the original model.

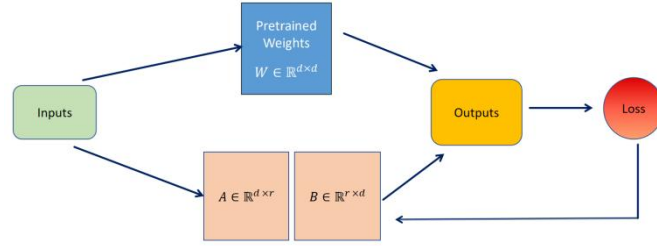


Figure 2. Low Rank work flow.

4.4 Low rank modification in the model

In our project, we decided to modify the q_proj, v_proj, and o_proj components of the Multi-Queried Attention mechanism in the Llama 3.2 1B model. These layers are responsible for generating the query, value, and output vectors, making them critical for attention computation.

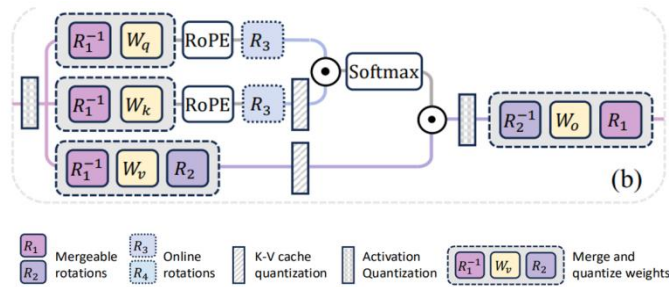


Figure 3. Multi-queried Attention structure[1].

(1) Why modify the multi-queried Attention

The attention mechanism is the central component of transformer-based models, directly responsible for how the model focuses on relevant input tokens. By targeting the attention layers, we maximize the impact of our LoRA fine-tuning, allowing the model to adapt more effectively to new tasks.

Most common fine-tuning strategies modify only `q_proj` and `v_proj`. However, to increase the number of learnable parameters and improve adaptability, we also include `o_proj` in our modification. This extended focus enhances the model’s capacity to learn richer attention dynamics during fine-tuning.

(2) How we Implemented the modification

We applied Low-Rank Adaptation (LoRA) to the attention projection layers. This involves freezing the original pretrained weights and injecting trainable low-rank matrices (A and B) into each projection layer. The new weights are expressed as:

$$W' = W + \frac{\alpha}{r}(A \cdot B),$$

where $A \in \mathbb{R}^{in \times r}$, $B \in \mathbb{R}^{r \times out}$, α is a scaling factor, W is the original weight matrix (frozen). Only A and B are trainable. We used custom PyTorch wrappers to integrate these LoRA modules into the `q_proj`, `v_proj`, and `o_proj` layers, while maintaining compatibility with mixed precision and parallelism strategies (e.g., DDP and FSDP).

(3) Modification results

Here the two graphs are our low rank modification results.

| | |
|--|--|
| Replacing model.layers.0.self_attn.q_proj with LoRA | LoRA is placed at model.layers.0.self_attn.q_proj |
| Replacing model.layers.0.self_attn.v_proj with LoRA | LoRA is placed at model.layers.0.self_attn.v_proj |
| Replacing model.layers.0.self_attn.o_proj with LoRA | LoRA is placed at model.layers.0.self_attn.o_proj |
| Replacing model.layers.1.self_attn.q_proj with LoRA | LoRA is placed at model.layers.1.self_attn.q_proj |
| Replacing model.layers.1.self_attn.v_proj with LoRA | LoRA is placed at model.layers.1.self_attn.v_proj |
| Replacing model.layers.1.self_attn.o_proj with LoRA | LoRA is placed at model.layers.1.self_attn.o_proj |
| Replacing model.layers.2.self_attn.q_proj with LoRA | LoRA is placed at model.layers.2.self_attn.q_proj |
| Replacing model.layers.2.self_attn.v_proj with LoRA | LoRA is placed at model.layers.2.self_attn.v_proj |
| Replacing model.layers.2.self_attn.o_proj with LoRA | LoRA is placed at model.layers.2.self_attn.o_proj |
| Replacing model.layers.3.self_attn.q_proj with LoRA | LoRA is placed at model.layers.3.self_attn.q_proj |
| Replacing model.layers.3.self_attn.v_proj with LoRA | LoRA is placed at model.layers.3.self_attn.v_proj |
| Replacing model.layers.3.self_attn.o_proj with LoRA | LoRA is placed at model.layers.3.self_attn.o_proj |
| Replacing model.layers.4.self_attn.q_proj with LoRA | LoRA is placed at model.layers.4.self_attn.q_proj |
| Replacing model.layers.4.self_attn.v_proj with LoRA | LoRA is placed at model.layers.4.self_attn.v_proj |
| Replacing model.layers.4.self_attn.o_proj with LoRA | LoRA is placed at model.layers.4.self_attn.o_proj |
| Replacing model.layers.5.self_attn.q_proj with LoRA | LoRA is placed at model.layers.5.self_attn.q_proj |
| Replacing model.layers.5.self_attn.v_proj with LoRA | LoRA is placed at model.layers.5.self_attn.v_proj |
| Replacing model.layers.5.self_attn.o_proj with LoRA | LoRA is placed at model.layers.5.self_attn.o_proj |
| Replacing model.layers.6.self_attn.q_proj with LoRA | LoRA is placed at model.layers.6.self_attn.q_proj |
| Replacing model.layers.6.self_attn.v_proj with LoRA | LoRA is placed at model.layers.6.self_attn.v_proj |
| Replacing model.layers.6.self_attn.o_proj with LoRA | LoRA is placed at model.layers.6.self_attn.o_proj |
| Replacing model.layers.7.self_attn.q_proj with LoRA | LoRA is placed at model.layers.7.self_attn.q_proj |
| Replacing model.layers.7.self_attn.v_proj with LoRA | LoRA is placed at model.layers.7.self_attn.v_proj |
| Replacing model.layers.7.self_attn.o_proj with LoRA | LoRA is placed at model.layers.7.self_attn.o_proj |
| Replacing model.layers.8.self_attn.q_proj with LoRA | LoRA is placed at model.layers.8.self_attn.q_proj |
| Replacing model.layers.8.self_attn.v_proj with LoRA | LoRA is placed at model.layers.8.self_attn.v_proj |
| Replacing model.layers.8.self_attn.o_proj with LoRA | LoRA is placed at model.layers.8.self_attn.o_proj |
| Replacing model.layers.9.self_attn.q_proj with LoRA | LoRA is placed at model.layers.9.self_attn.q_proj |
| Replacing model.layers.9.self_attn.v_proj with LoRA | LoRA is placed at model.layers.9.self_attn.v_proj |
| Replacing model.layers.9.self_attn.o_proj with LoRA | LoRA is placed at model.layers.9.self_attn.o_proj |
| Replacing model.layers.10.self_attn.q_proj with LoRA | LoRA is placed at model.layers.10.self_attn.q_proj |
| Replacing model.layers.10.self_attn.v_proj with LoRA | LoRA is placed at model.layers.10.self_attn.v_proj |
| Replacing model.layers.10.self_attn.o_proj with LoRA | LoRA is placed at model.layers.10.self_attn.o_proj |
| Replacing model.layers.11.self_attn.q_proj with LoRA | LoRA is placed at model.layers.11.self_attn.q_proj |
| Replacing model.layers.11.self_attn.v_proj with LoRA | LoRA is placed at model.layers.11.self_attn.v_proj |
| Replacing model.layers.11.self_attn.o_proj with LoRA | LoRA is placed at model.layers.11.self_attn.o_proj |
| Replacing model.layers.12.self_attn.q_proj with LoRA | LoRA is placed at model.layers.12.self_attn.q_proj |
| Replacing model.layers.12.self_attn.v_proj with LoRA | LoRA is placed at model.layers.12.self_attn.v_proj |
| Replacing model.layers.12.self_attn.o_proj with LoRA | LoRA is placed at model.layers.12.self_attn.o_proj |
| Replacing model.layers.13.self_attn.q_proj with LoRA | LoRA is placed at model.layers.13.self_attn.q_proj |
| Replacing model.layers.13.self_attn.v_proj with LoRA | LoRA is placed at model.layers.13.self_attn.v_proj |
| Replacing model.layers.13.self_attn.o_proj with LoRA | LoRA is placed at model.layers.13.self_attn.o_proj |
| Replacing model.layers.14.self_attn.q_proj with LoRA | LoRA is placed at model.layers.14.self_attn.q_proj |
| Replacing model.layers.14.self_attn.v_proj with LoRA | LoRA is placed at model.layers.14.self_attn.v_proj |
| Replacing model.layers.14.self_attn.o_proj with LoRA | LoRA is placed at model.layers.14.self_attn.o_proj |
| Replacing model.layers.15.self_attn.q_proj with LoRA | LoRA is placed at model.layers.15.self_attn.q_proj |
| Replacing model.layers.15.self_attn.v_proj with LoRA | LoRA is placed at model.layers.15.self_attn.v_proj |
| Replacing model.layers.15.self_attn.o_proj with LoRA | LoRA is placed at model.layers.15.self_attn.o_proj |
| | Using device cuda |
| | Dataset Loaded: 38105 training, 4234 validation samples. |

Figure 4, 5. Low rank modification results.

5. Datasets and Data Process

5.1 Datasets

In this project, we have chosen OpenOrca from Hugging Face as the datasets. The size of this datasets is approximately 2.87GB and contains 3 million rows of data. It offers a variety of question-answer samples, covering topics such as common-sense questions, mathematical reasoning, and technical issues, etc. This datasets can help the model enhance its multi-round answering and reasoning capabilities. Each sample in this datasets consists of four fields: “id” is the unique identifier of the sample, “system_prompt” provides the setting of the system role, “question” is the question or instruction raised by the user, and response is the answer that the model should generate. This structure is convenient for concatenating “system_prompt” and “question” to form the model input, and using response as the supervision label for training the ability of dialogue generation and instruction following.

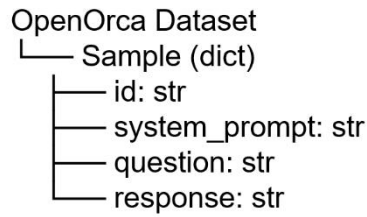


Figure 6. Datasets structure.

5.2 Tokenizer

The data processing stage is to transform the original dataset into the input format required for training the Llama model. It is a crucial part of the fine-tuning task and becomes particularly important when dealing with large datasets like OpenOrca, where speed and efficiency are of great significance.

The most crucial step in data processing is to select an appropriate tokenizer. Through our numerous experiments, we have found that choosing different tokenizers can significantly affect the performance of the model. To ensure the training quality of the model, it is necessary to select the tokenizer that is most suitable for the model. Eventually, we chose the AutoTokenizer from the Hugging Face Transformers library. It is an automated tokenizer interface that can automatically load the Tokenizer compatible with the specified pre-trained model based on the model's name, convert the string-type natural language text into digital input (token IDs) that the model can handle, and generate information such as attention masks. Therefore, in our project, it can automatically load the tokenizer matching Llama, ensuring that the input text is processed by the correct tokenization method.

5.3 Dask Parallelization Acceleration

After resolving the tokenizer issue, another task to be accomplished is to speed up the processing flow of the tokenizer. When dealing with a large datasets like OpenOrca, the serial processing time is approximately 80 minutes, which seriously affects the overall speed of model training. Since the data processing stage cannot proceed without being completed, we need to parallelize this process to shorten the overall time used. Initially, we first used the num_proc method of Dataset.map() to split the serial processing into multiple processes. However, we found this method extremely unstable. It processed quickly at the beginning, but the speed would drop significantly after only half a minute, and eventually it was even slower than the serial processing. After multiple experiments, we finally chose to use the Dask method taught in class to accelerate the data processing process.

Just as was explained in class, Dask is an open-source parallel computing framework that supports multi-processes and concurrent task scheduling on distributed clusters. Especially, we can utilize the slurm resources provided by our school, which is highly compatible with the application scenarios of Dask. Therefore, we split the datasets into multiple chunks, distribute each chunk to different workers for processing, and finally aggregate these results to achieve parallelization acceleration of the tokenizer process. The main process is as follows:

(1) Read configuration information

This step is quite simple but crucial. Because the program needs to complete all subsequent operations based on the parameters in config.py. Including but not limited to the name of the dataset, the fields used for constructing prompts, and the maximum input length of the tokenizer, all the key information needed for the data processing stage such as these are included.

(2) Configure a SLURM distributed Dask cluster

This step is the essence of using Dask. Since the school provides Explorer with high-performance computing (HPC) resources, we can launch multiple Dask task nodes on Explorer through the SLURM scheduler to form a temporary personalized processing cluster. For example, using SLURMCluster(...) to declare various detailed information of the required cluster resources, using scale(jobs=n) to start n workers, and finally using Client(cluster) to connect to the Dask cluster and generate a client for submitting tasks. Subsequent operations will need to utilize this client.

(3) Main processing flow

The main processing procedure is divided into three major steps: task division, task allocation and results collection.

(3.1) Task division

After loading the datasets using load_dataset(), all the data will be split into several chunks according to the set chunk_size, and each chunk will be processed independently. The specific handling method is written in the tokenize_chunk_remote() function. When each Dask worker is assigned a chunk, this function will be executed. The core function of this function is to construct prompts for a batch of raw input texts, then encode them with tokenizer, and finally generate a data dictionary in the required format for model training.

(3.2) Task allocation

After dividing all the data into chunks, the next step is to distribute these chunks to Dask workers for processing. At this point, the scatter and submit mechanisms of Dask need to be utilized. "scatter" is Dask's data sharing mechanism. Firstly, it utilizes the function "client.scatter()" to broadcast the "config" configuration to all worker nodes, enabling all workers to obtain all the configuration parameters related to the processing operation before starting the processing. After all workers have known how to handle the data, the previously completed tokenizer processing function and chunks are submitted asynchronously one by one to each worker through client.submit(), and then handed over to each worker for processing. A future object is returned. In Dask, a future is an object that refers to the result of an asynchronous execution task. It represents the result of a running task and can be used to query the status of the task and wait for the result through the future object.

(3.3) Results collection

After submitting the task, the result of the completed task can be obtained asynchronously through the as_completed() function combined with future. In Dask, as_completed() can process the results immediately when the tasks are completed. Therefore, the results will be returned successively instead of waiting for all tasks to be completed before collecting them uniformly. After collecting the results,

flatten the batches returned by multiple tasks into a large list. Then, using Hugging Face's `dataset.from_List`, a new Dataset object can be constructed or directly processed and passed into the training DataLoader.

After all the steps are completed, we use `client.close()` and `cluster.close()` to release resources and close the client.

In the actual test, it originally took approximately 80 minutes to process the entire data set serially, but after acceleration by Dask, it only takes about 36 minutes. And during the processing, the processing speed has been increasing continuously, which can increase from approximately 819 samples per second at the beginning all the way to approximately 1,663 samples per second.

Regarding this phenomenon, we believe it might be because when Dask is initially started, each worker is not fully activated, as each worker needs to go through a series of startup processes. Therefore, in the early stage, only some workers were involved in the data processing work, resulting in a relatively low throughput in the early stage. As the workers continuously accelerate, both the overall speed and throughput increase significantly until all the workers are involved in the work, the resource utilization reaches the upper limit, and the throughput rate gradually stabilizes.

6. Optimization:

We want to study the improvement of our modified model. To evaluate the effectiveness of our model modifications, we first establish a set of testing principles and baseline metrics. These metrics help us quantify improvements in both performance and efficiency.

6.1 Evaluation Metrics

We focus on three main categories:

(1) Speed Evaluation

- **Throughput (tokens/ sec):** Measures how many tokens the model can process per second. Higher values indicate faster training and inference.

$$\text{Throughput} = \frac{\text{The num of tokens generated}}{\text{Total Time Taken}}.$$

- **Epoch Time:** The time taken to complete one full pass over the dataset.
- **Total Training Time:** The overall time consumed for the entire fine-tuning process.

(2) Memory Usage

Peak Memory Consumption: Tracks the highest amount of memory used during training or inference. Memory efficiency is especially important when using parallelization techniques. For example, model parallelism and Fully Sharded Data Parallelism (FSDP) can help reduce memory load on individual GPUs.

(3) Quality

We assess model quality using the following widely accepted metrics:

Perplexity (PPL):

$$\text{PPL} = e^{(-\frac{1}{N} \sum_{i=1}^N \log P(w_i))},$$

where $P(w_i)$ is the probability of token w_i .

Interpretation: Lower perplexity indicates better language modeling performance, as the model more accurately predicts the next word.

BLUE Score:

$$\text{BLUE} = \text{BP} \times \exp\left(\sum_{i=1}^N w_i \log p_i\right),$$

where BP: Brevity Penalty and p_i : Precision for n-grams.

BLEU measures the n-gram overlap between the generated text and reference text. It is particularly effective for tasks like machine translation and structured generation. Higher scores indicate better performance.

ROUGE-L Score (for Summarization):

- Measures the recall of generated tokens compared to reference outputs.
- Commonly used for summarization tasks, where capturing key content is crucial.
- High recall implies that the model retains important information from the input.

These metrics form the foundation for evaluating how our model improvements—especially LoRA fine-tuning and speed optimizations—affect both training efficiency and output quality.

6.2 Speed Optimization

To improve the training efficiency of our LoRA-modified Llama 3.2 1B model, we explored various optimization strategies. These included techniques that reduce precision and parallelize computation to better utilize available hardware. Specifically, we implemented and evaluated:

- Mixed Precision Training
- Data Distributed Parallelism (DDP)
- Fully Sharded Data Parallelism (FSDP)

Our experiments focused on comparing these techniques across key metrics such as training speed, memory usage, and output quality, ultimately identifying the most effective approach for accelerating model training.

6.2.1 Mixed Precision

(1) Mixed Precision introduction

Deep learning computations typically rely on floating-point representations. In PyTorch, tensors default to 32-bit floating point format (FP32). However, reduced-precision formats such as FP16, BF16, and FP8 offer significant performance and memory benefits.

The table below summarizes the characteristics of different precision

| Format | Bits | precision | Memory | Use case |
|--------|------|---|-------------------------------|---|
| FP32 | 32 | High | High | Pytorch default |
| FP16 | 16 | Medium | Medium (almost half of FP32) | mixed precision, training and inference |
| BF16 | 16 | Lower mantissa than FP16, but higher training stability | almost equal to FP16 | mixed precision training and inference |
| FP8 | 8 | lowest | lowest | quantized training and inference |

Mixed precision training combines high-precision and low-precision operations to speed up training while maintaining numerical stability. For our experiments, we compared FP32 (baseline), FP16, and BF16 to analyze their impact on speed and quality.

(2) The experiment settings

To assess the performance impact of different precision modes, we conducted a series of experiments under controlled conditions. After extensive testing with various configurations, we selected the most representative setting for comparison.

| Experiment setting | |
|--------------------|--|
| num of epoch | 5 |
| batch_size | 20 |
| data_size | 1% (38105 training samples, 4234 validation samples) |
| GPU | NVIDIA H100 80GB HBM3 |

This setup provided a balance between training time, memory usage, and evaluation accuracy. All experiments were tracked using TensorBoard for consistent logging and visualization of metrics.

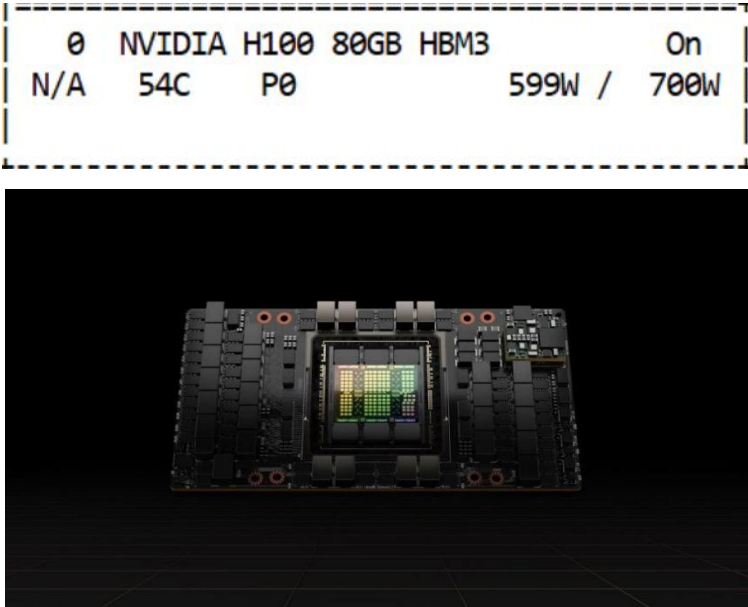


Figure 7. NVIDIA H100 80GB HBM3.

(3) Experiment Results

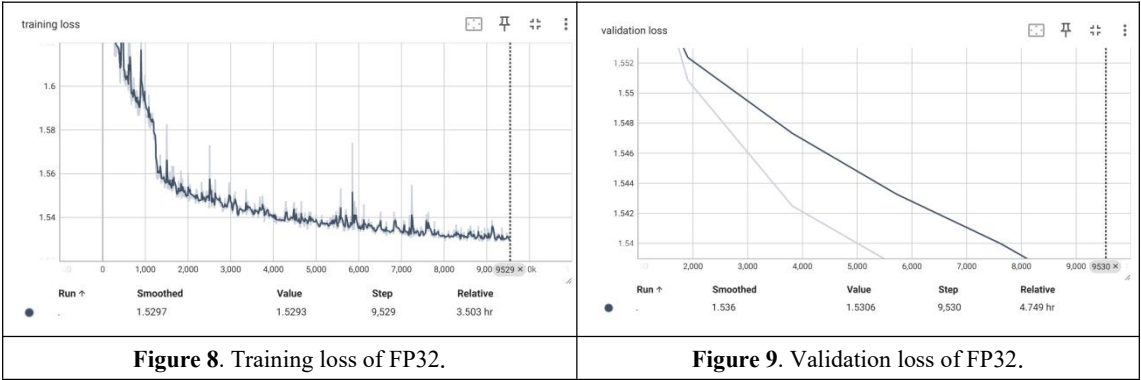
We evaluated the model’s behavior across three different precision types: FP32, FP16, and BF16. For each setting, we recorded key metrics including:

- Training loss progression
- Throughput (samples per second)
- Memory consumption
- Output quality (BLEU and ROUGE scores)

(3.1) FP32 Results:

Below are the results of training the model using full 32-bit floating point (FP32) precision. This setting serves as the performance baseline for comparison with lower-precision formats.

The Loss:



The Speed:

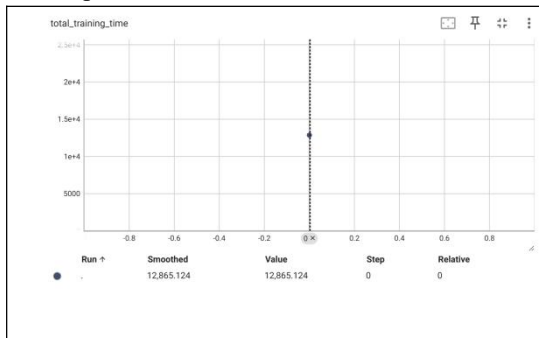


Figure 10. Total Training time of FP32.

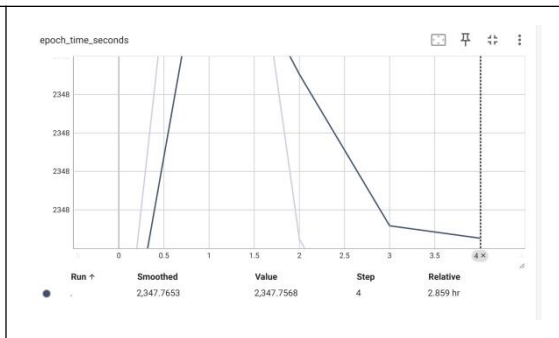


Figure 11. Epoch time of FP32.

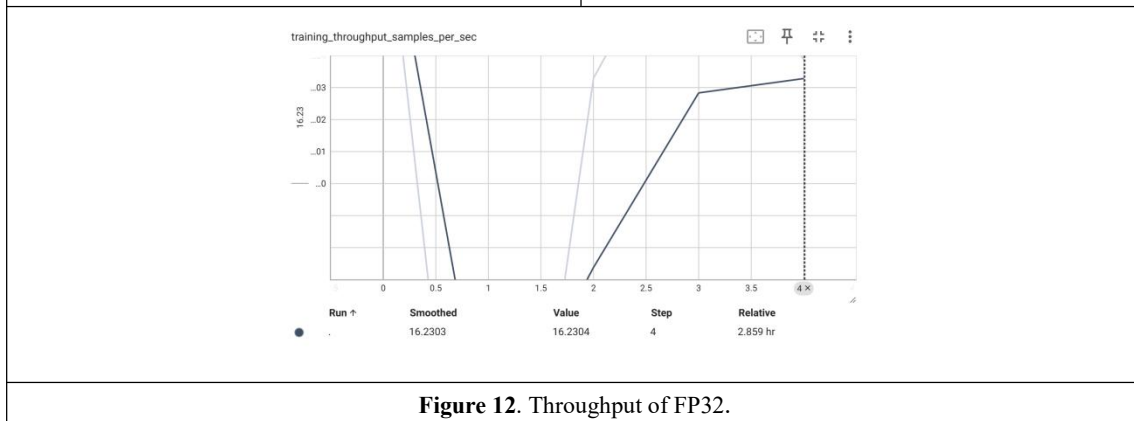


Figure 12. Throughput of FP32.

The Quality and Memory Usage:

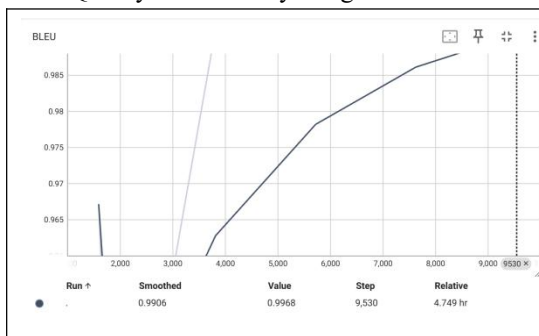


Figure 13. BLUE Score of FP32.

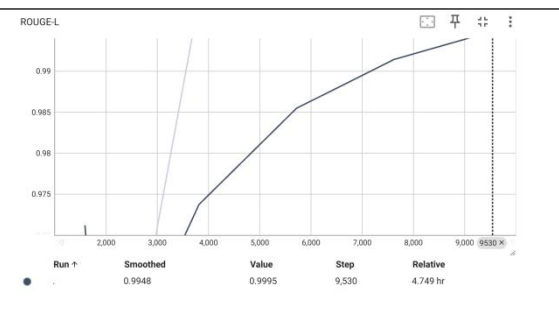


Figure 14. ROUGE-L Score of FP32.

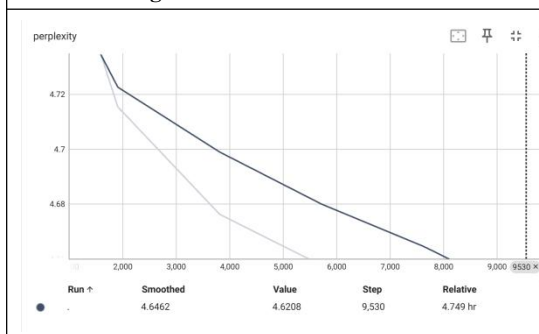


Figure 15. Perplexity of FP32.

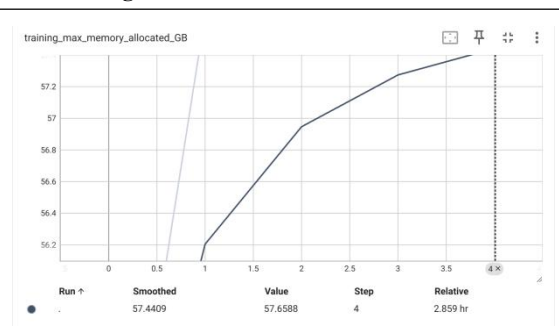


Figure 16. Maximum Memory Usage of FP32.

(3.2) FP16 Results:

The following results reflect the model's performance when trained using 16-bit floating point (FP16) precision. This format utilizes specialized hardware (Tensor Cores) for faster computation and significantly reduces memory usage.

The Loss:

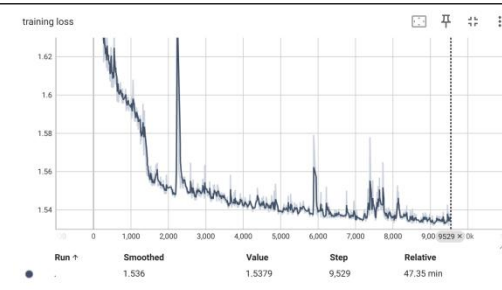


Figure 17. Total Training time of FP16.

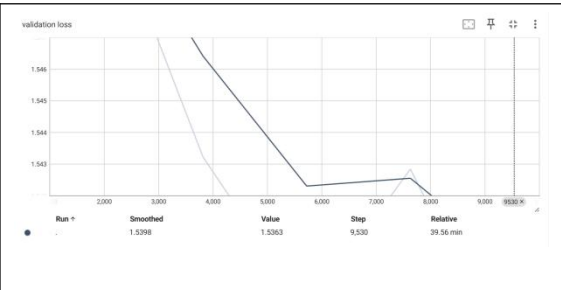


Figure 18. Validation loss of FP16.

The Speed:

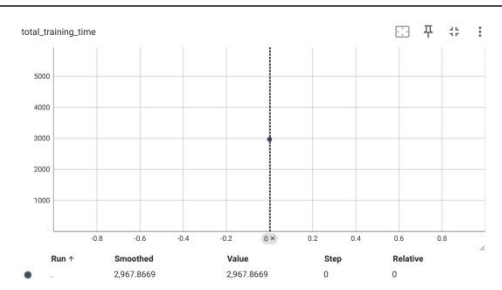


Figure 19. Total Training time of FP16.

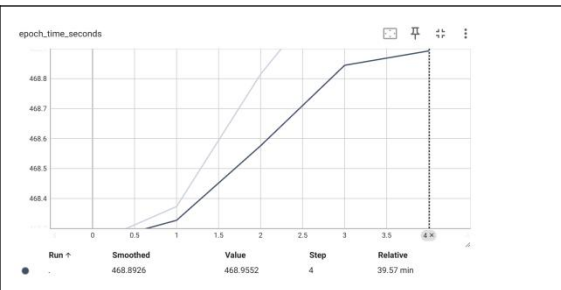


Figure 20. Epoch time of FP16.

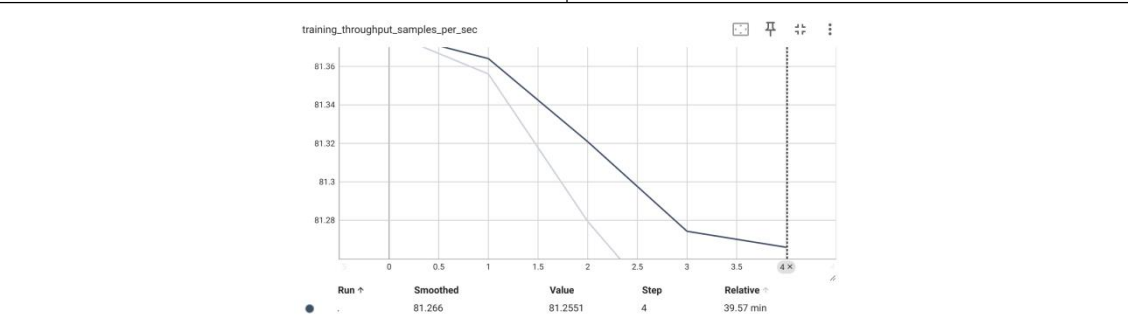


Figure 21. Throughput of FP16.

The Quality and Memory Usage:

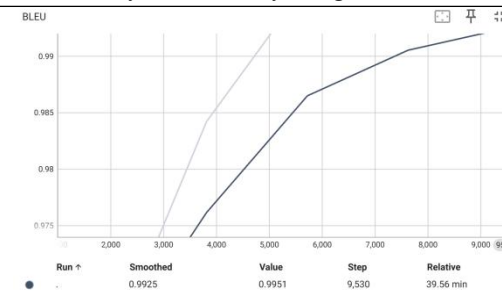


Figure 22. BLUE Score of FP16.

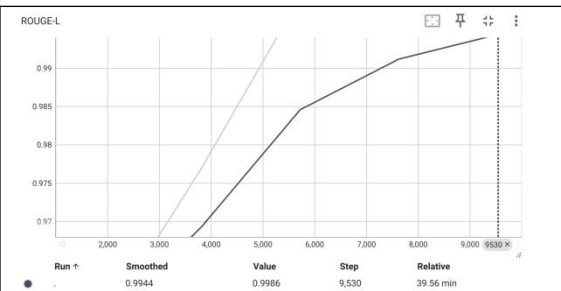


Figure 23. ROUGE-L Score of FP16.

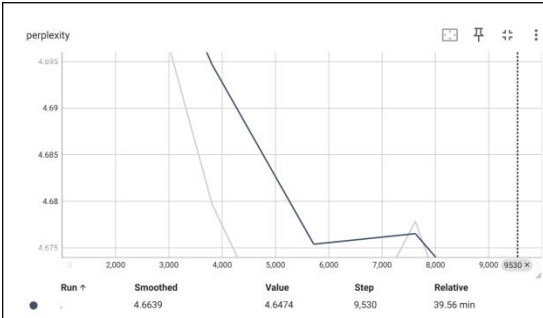


Figure 24. Perplexity of FP16.

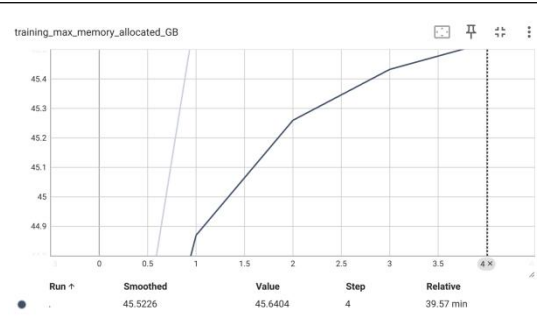


Figure 25. Maximum Memory Usage of FP16.

(3.3) BF16 Results:

The following results reflect the model's performance using BF16 (Brain Floating Point) precision. BF16 retains a wide dynamic range close to FP32, but with reduced memory usage, making it ideal for training on modern accelerators like NVIDIA H100 GPUs.

The Loss:

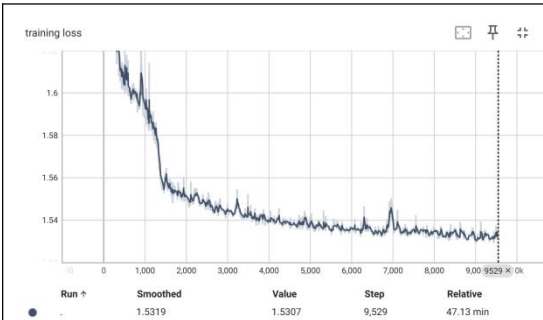


Figure 26. Perplexity of BF16.

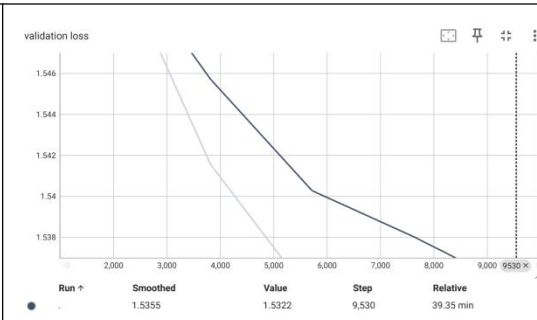


Figure 27. Validation loss of BF16.

The Speed:

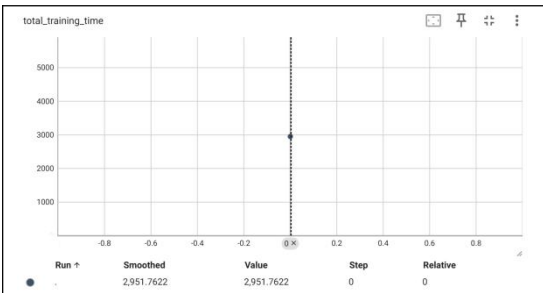


Figure 28. Total Training time of BF16.

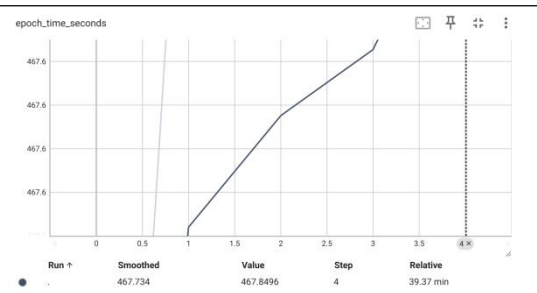


Figure 29. Epoch time of BF16.

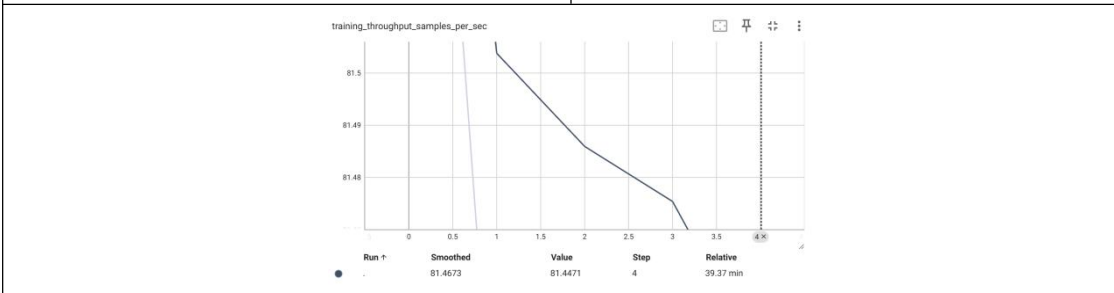
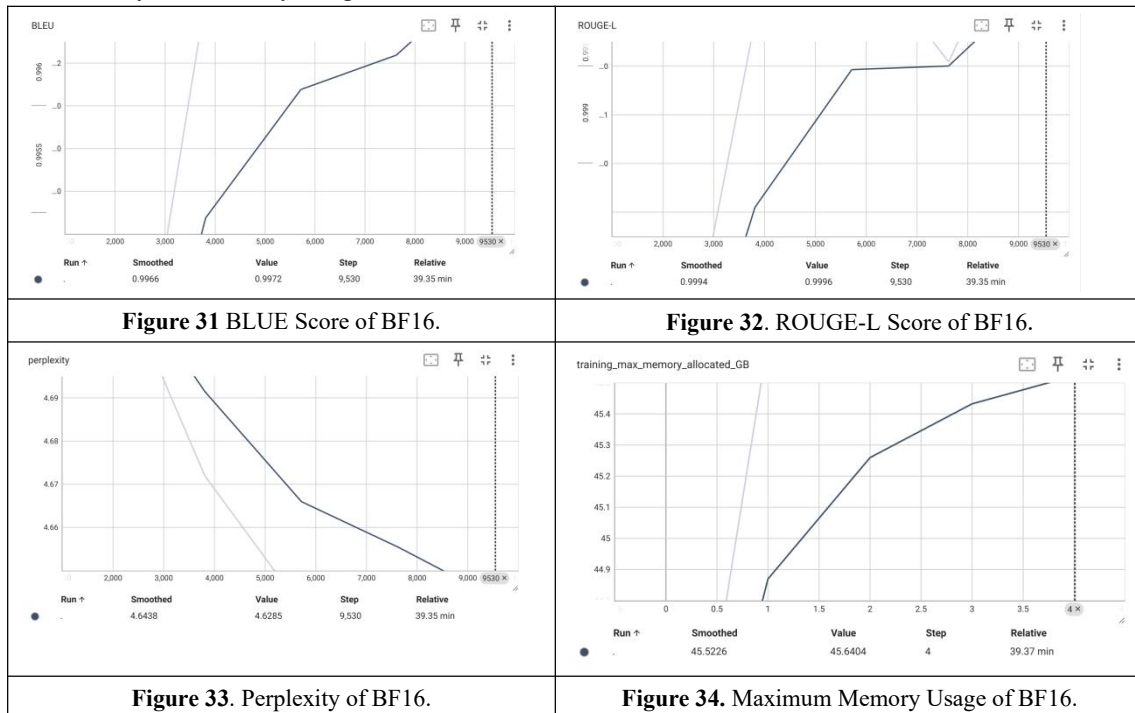


Figure 30. Throughput of BF16.

The Quality and Memory Usage:



The Summary table:

| | |
|------------------------------|---------------------|
| FP32 | |
| Blue (last epoch) | 0.9968 |
| Rouge_L (last epoch) | 0.9995 |
| Preplexity (last epoch) | 4.6462 |
| Validation loss (last epoch) | 1.5306 |
| Epoch Time (average) | 2573.025s |
| Total Training Time | 12865.124s~3.57h |
| Throughput | 16.2301 samples/sec |
| Max Memory Usage | 57.6588 GB |

| | |
|------------------------------|-------------------------|
| FP16 | |
| Blue (last epoch) | 0.9951 |
| Rouge_L (last epoch) | 0.9986 |
| Preplexity (last epoch) | 4.6639 |
| Validation loss (last epoch) | 1.5363 |
| Epoch Time (average) | 593.573s |
| Total Training Time | 2,967.8669s = 49.464min |
| Throughput | 81.2979 samples/sec |
| Max Memory Usage | 45.6404GB |

| | |
|----------------------|--------|
| BF16 | |
| Blue (last epoch) | 0.9972 |
| Rouge_L (last epoch) | 0.9996 |

| | |
|------------------------------|-----------------------|
| Preplexity (last epoch) | 4.6438 |
| Validation loss (last epoch) | 1.5322 |
| Epoch Time (average) | 590.352s |
| Total Training Time | 2,951.7622s~49.196min |
| Throughput | 81.4855 samples/sec |
| Max Memory Usage | 45.6404 GB |

(4) Comparison and analysis

a. Speed Comparison

The following graphs illustrate the differences in training efficiency across the three precision modes (FP32, FP16, and BF16):

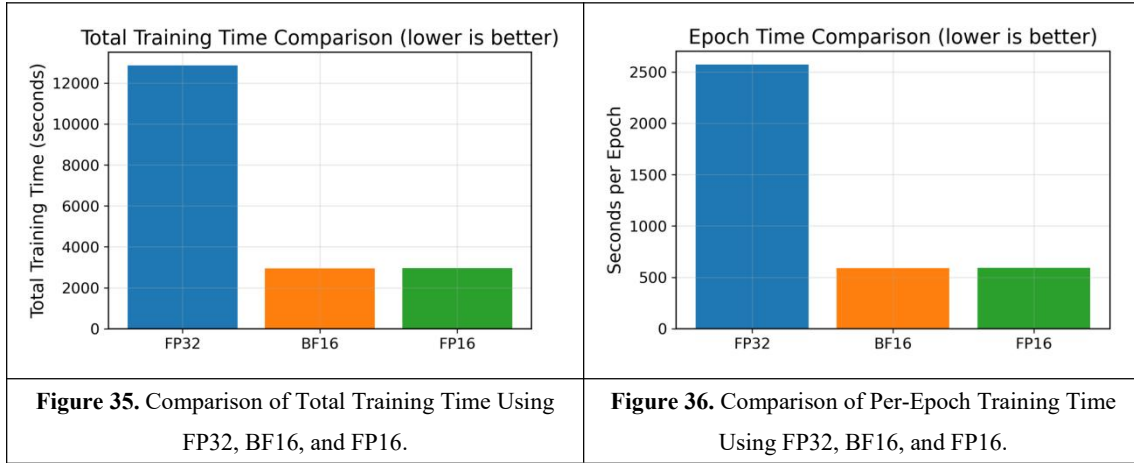


Figure 35. Comparison of Total Training Time Using FP32, BF16, and FP16.

Figure 36. Comparison of Per-Epoch Training Time Using FP32, BF16, and FP16.

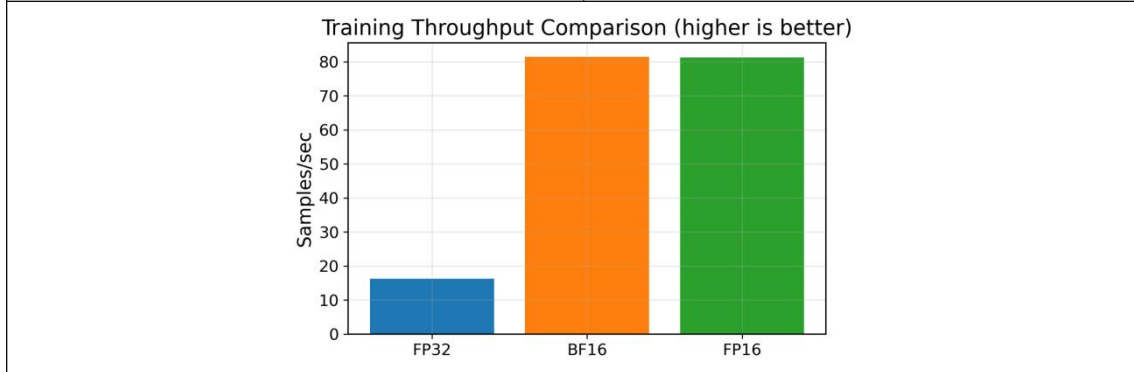


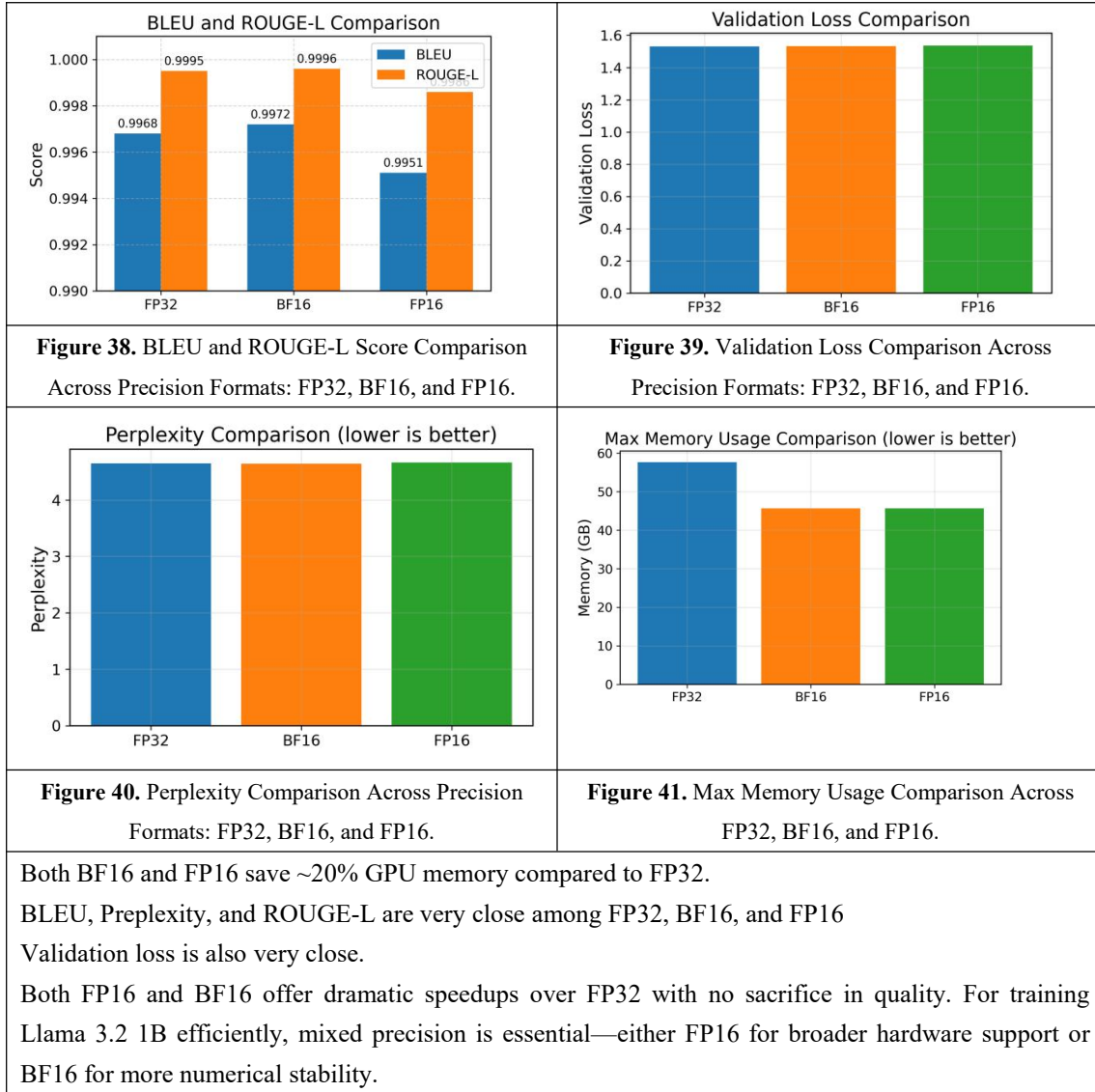
Figure 37. Samples Processed per Second with FP32, BF16, and FP16 Precision.

Both BF16 and FP16 increase training throughput by about 5× compared to FP32!

Both BF16 and FP16 reduce total training time by more than 4× compared to FP32.

These results clearly demonstrate that mixed precision (especially FP16 and BF16) leads to substantial gains in training speed and compute efficiency, making them highly suitable for large-scale fine-tuning.

b. Quality and Memory Usage:



6.2.3 Data Distributed Parallelism (DDP)

(1) DDP introduction

Data Distributed Parallelism (DDP) is a widely used technique to accelerate deep learning training by distributing data across multiple GPUs. Each device receives a subset of the dataset and maintains a full copy of the model. During training, gradients are computed independently on each GPU and then synchronized across all devices to ensure consistent updates.

This approach improves scalability and enables efficient utilization of multiple GPUs without significantly increasing memory consumption per device. In this project, we implemented DDP to evaluate its impact on training speed and throughput when applied to our LoRA-modified Llama 3.2 1B model.

(2) Experiment Settings

The experimental configuration for testing DDP was as follows:

| Experiment setting | |
|--------------------|--|
| num of epoch | 5 |
| batch_size | 24 |
| data_size | 3% (114316 training samples, 12702 validation) |

| | |
|-----------------|---------------------------|
| | samples) |
| GPU | NVIDIA H100 80GB HBM3 ×15 |
| Mixed Precision | FP16 |

These settings were chosen to balance compute workload and enable measurable comparison across different GPU counts.

(3) Experiment Results

The training performance was evaluated under different GPU counts (1, 2, 4, and 8 GPUs). The results include metrics such as epoch time, total training time, throughput, and memory usage.

1 GPU:

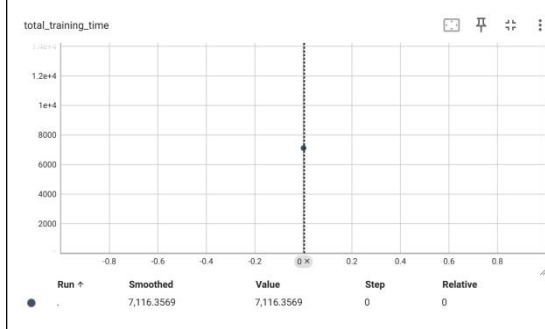


Figure 42. Total Training Time with FP16 on a Single GPU Setup.

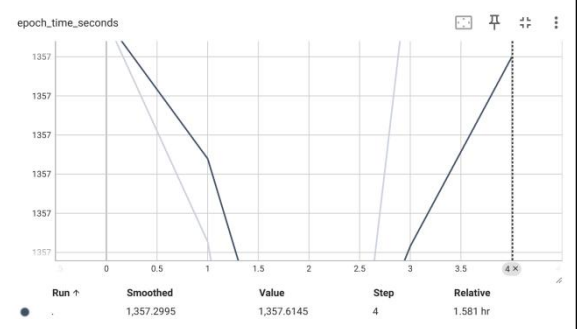


Figure 43. Epoch Training Time with FP16 on a Single GPU Setup.

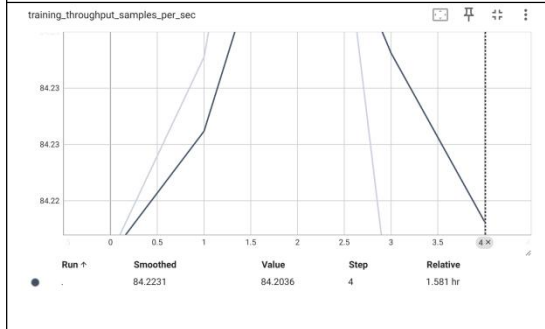


Figure 44. Training Throughput with FP16 on a Single GPU Setup.

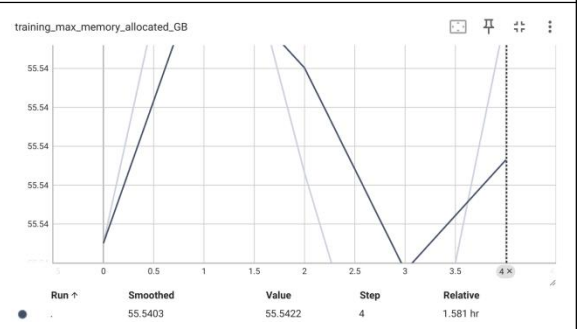


Figure 45. Maximum GPU Memory Usage with FP16 on a Single GPU Setup.

2 GPUs

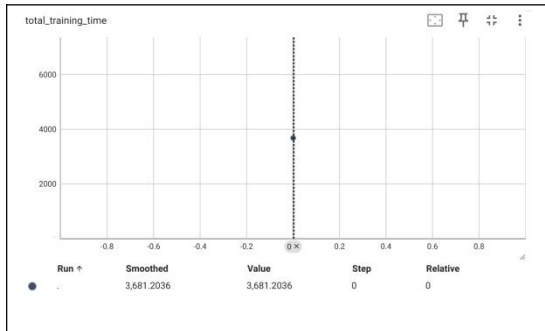


Figure 46. Total Training Time with FP16 Precision on a Dual-GPU Setup.

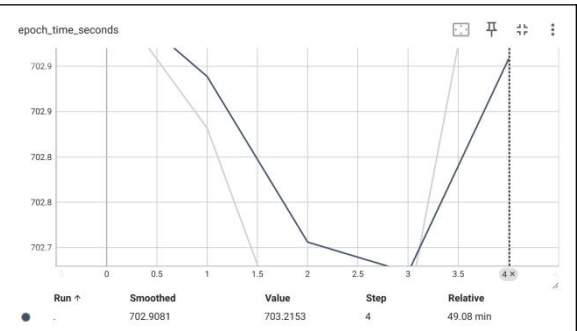


Figure 47. Epoch Training Time with FP16 Precision on a Dual-GPU Setup.

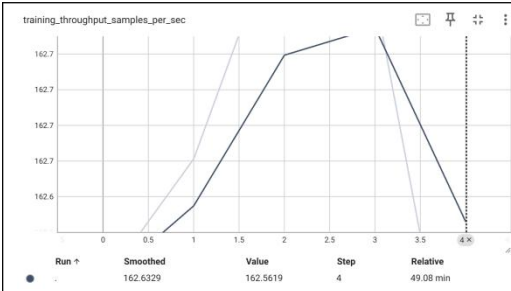


Figure 48. Training Throughput with FP16 Precision on a Dual-GPU Setup.

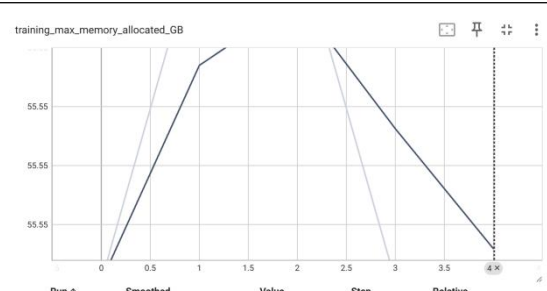


Figure 49. Maximum GPU Memory Usage with FP16 Precision on a Dual-GPU Setup.

4 GPUs

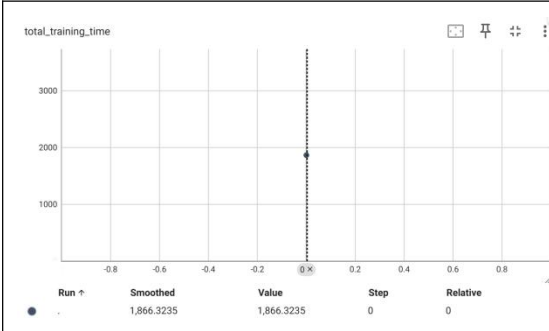


Figure 50. Total Training Time with FP16 Precision on a 4-GPU Setup.

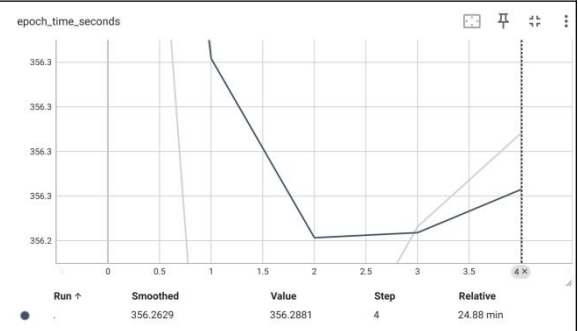


Figure 51. Epoch Training Time with FP16 Precision on a 4-GPU Setup.

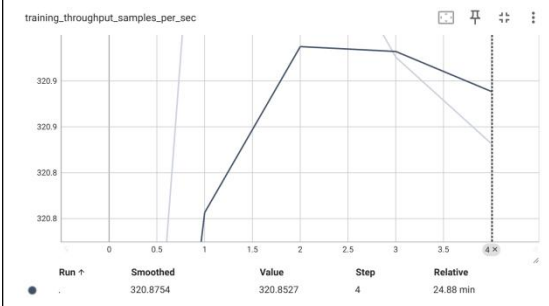


Figure 52. Training Throughput with FP16 Precision on a 4-GPU Setup.

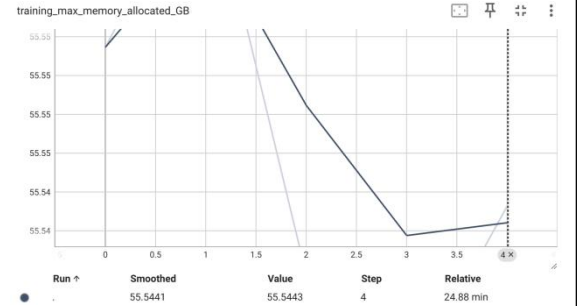


Figure 53. Maximum GPU Memory Usage with FP16 Precision on a 4-GPU Setup.

8 GPUs

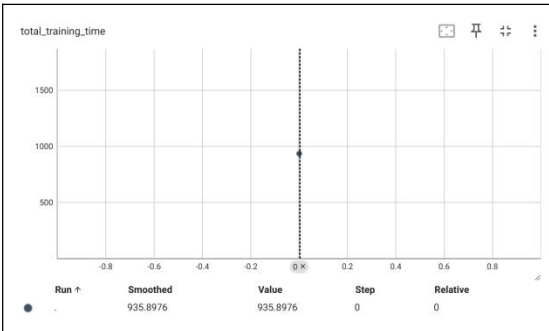


Figure 54. Total Training Time with FP16 Precision on an 8-GPU Setup.

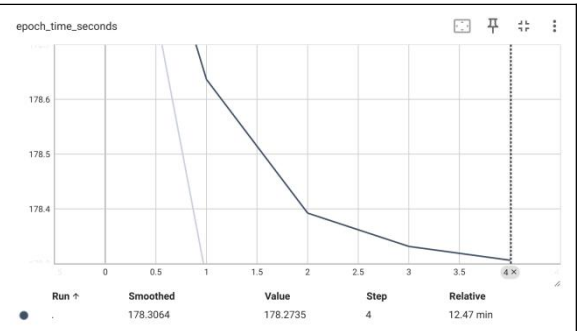


Figure 55. Epoch Training Time with FP16 Precision on an 8-GPU Setup.

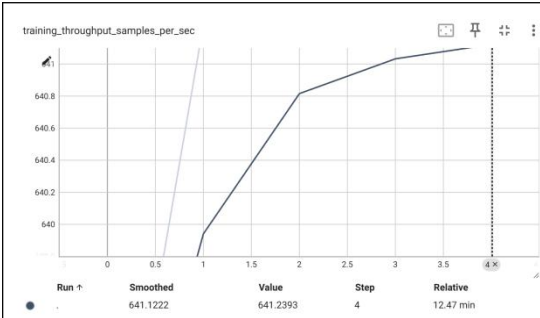


Figure 56. Training Throughput with FP16 Precision on an 8-GPU Setup.

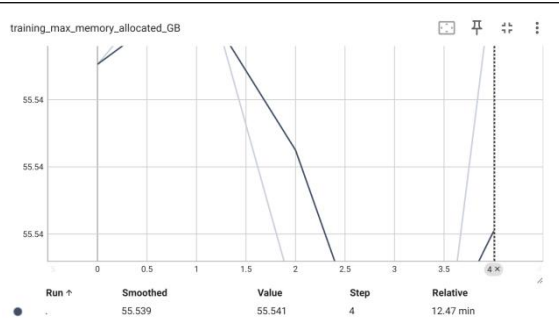


Figure 57. Maximum GPU Memory Usage with FP16 Precision on an 8-GPU Setup.

The Summary Table:

| GPU number | Max memory Usage | Epoch Time (average) | Total Training Time | Throughput |
|------------|------------------|----------------------|-------------------------|------------------------|
| 1 | 55.5422 GB | 1423.2714s | 7,116.3569s ~ 1.98h | 84.232 samples/second |
| 2 | 55.548 GB | 736.2461s | 3,681.2036s ~ 1.02h | 162.649 samples/second |
| 4 | 55.5443GB | 373.2647s | 1,866.3235s ~ 31.105min | 320.843 samples/second |
| 8 | 55.541GB | 187.1795s | 935.8976s ~ 15.60min | 640.639 samples/second |

(4) Comparison and Analysis

To evaluate the scalability of Data Distributed Parallelism (DDP), we compared training performance across different numbers of GPUs (1, 2, 4, and 8). The following metrics were analyzed:

The speed and memory comparison

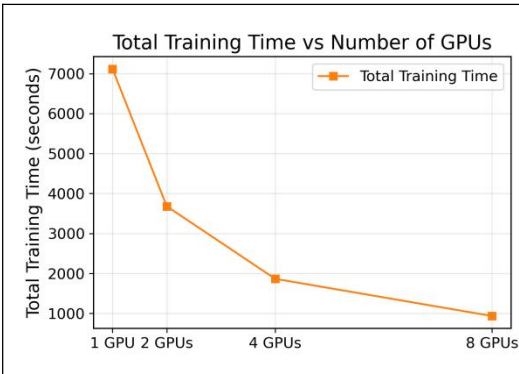


Figure 58. Total Training Time vs. Number of GPUs Using FP16 Precision.

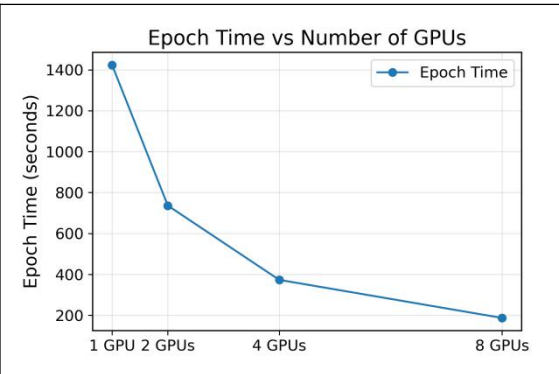
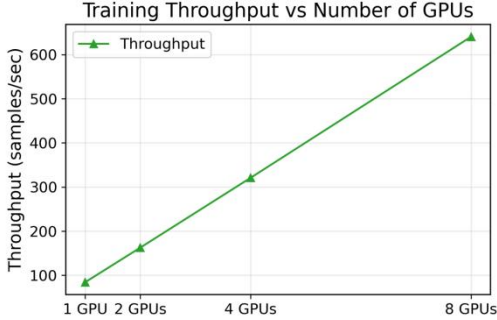
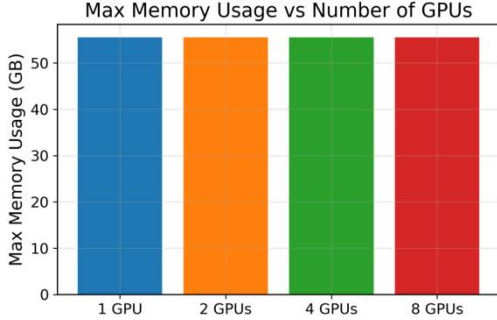


Figure 59. Epoch Time vs. Number of GPUs Using FP16 Precision.

- Training time decreases significantly as the number of GPUs increases.
- From ~7,000 seconds on 1 GPU to under ~1,000 seconds on 8 GPUs.
- This shows nearly linear scalability, reducing training time by a factor of ~7×

- Epoch duration drops from ~1,400 seconds on a single GPU to ~200 seconds on 8 GPUs.
 - Each doubling of GPU count results in ~50% or better reduction in epoch time.
- Excellent parallel efficiency, enabling faster

| <p>with 8 GPUs.</p> <p>DDP effectively scales with GPU count, reducing total training time drastically.</p> | <p>convergence within each epoch.</p> | | | | | | | | | | | | | | | | | | | | |
|--|---|--------------------------|-------|-----|--------|------|--------|------|--------|------|--|----------------|-----------------------|-------|-----|--------|-----|--------|-----|--------|-----|
|  <table border="1"> <caption>Data for Figure 60: Training Throughput vs. Number of GPUs</caption> <thead> <tr> <th>Number of GPUs</th> <th>Throughput (samples/sec)</th> </tr> </thead> <tbody> <tr> <td>1 GPU</td> <td>~80</td> </tr> <tr> <td>2 GPUs</td> <td>~160</td> </tr> <tr> <td>4 GPUs</td> <td>~320</td> </tr> <tr> <td>8 GPUs</td> <td>~640</td> </tr> </tbody> </table> | Number of GPUs | Throughput (samples/sec) | 1 GPU | ~80 | 2 GPUs | ~160 | 4 GPUs | ~320 | 8 GPUs | ~640 |  <table border="1"> <caption>Data for Figure 61: Maximum Memory Usage vs. Number of GPUs</caption> <thead> <tr> <th>Number of GPUs</th> <th>Max Memory Usage (GB)</th> </tr> </thead> <tbody> <tr> <td>1 GPU</td> <td>~55</td> </tr> <tr> <td>2 GPUs</td> <td>~55</td> </tr> <tr> <td>4 GPUs</td> <td>~56</td> </tr> <tr> <td>8 GPUs</td> <td>~56</td> </tr> </tbody> </table> | Number of GPUs | Max Memory Usage (GB) | 1 GPU | ~55 | 2 GPUs | ~55 | 4 GPUs | ~56 | 8 GPUs | ~56 |
| Number of GPUs | Throughput (samples/sec) | | | | | | | | | | | | | | | | | | | | |
| 1 GPU | ~80 | | | | | | | | | | | | | | | | | | | | |
| 2 GPUs | ~160 | | | | | | | | | | | | | | | | | | | | |
| 4 GPUs | ~320 | | | | | | | | | | | | | | | | | | | | |
| 8 GPUs | ~640 | | | | | | | | | | | | | | | | | | | | |
| Number of GPUs | Max Memory Usage (GB) | | | | | | | | | | | | | | | | | | | | |
| 1 GPU | ~55 | | | | | | | | | | | | | | | | | | | | |
| 2 GPUs | ~55 | | | | | | | | | | | | | | | | | | | | |
| 4 GPUs | ~56 | | | | | | | | | | | | | | | | | | | | |
| 8 GPUs | ~56 | | | | | | | | | | | | | | | | | | | | |
| <p>Figure 60. Training Throughput vs. Number of GPUs Using FP16 Precision.</p> | <p>Figure 61. Maximum Memory Usage vs. Number of GPUs Using FP16 Precision.</p> | | | | | | | | | | | | | | | | | | | | |
| <ul style="list-style-type: none"> ● Throughput increases almost linearly with the number of GPUs ● ~80 samples/sec on 1 GPU → ~640 samples/sec on 8 GPUs ● This confirms that the model’s capacity to process data scales efficiently with hardware. <p>More GPUs directly boost data processing speed, improving throughput 8×.</p> | <ul style="list-style-type: none"> ● Memory usage remains nearly constant (~55–56 GB) across all GPU configurations. ● This is expected in DDP since each GPU maintains a full copy of the model. <p>While speed improves, DDP does not reduce memory consumption per GPU. This could be a limitation for larger models or memory-constrained environments.</p> | | | | | | | | | | | | | | | | | | | | |
| <p>Conclusion: Data Distributed Parallelism provides excellent scalability for training the Llama 3.2 1B model. While it doesn't reduce memory usage, it offers near-linear improvements in training speed and throughput, making it highly effective for accelerating LoRA-based fine-tuning.</p> | | | | | | | | | | | | | | | | | | | | | |

6.2.4 FSDP

(1) FSDP introduction

Fully Sharded Data Parallelism is an advanced parallelization technique that partitions both the model parameters and optimizer states across multiple GPUs. Unlike DDP, which replicates the model across devices, FSDP shards the model to reduce per-GPU memory consumption.

This approach enables efficient training of large models by lowering the memory footprint on individual GPUs. During training, FSDP coordinates model computation and gradient synchronization through a series of communication steps.

In our implementation, we wrapped the DecoderLayer components of the Llama 3.2 1B model using FSDP to achieve optimal sharding granularity.

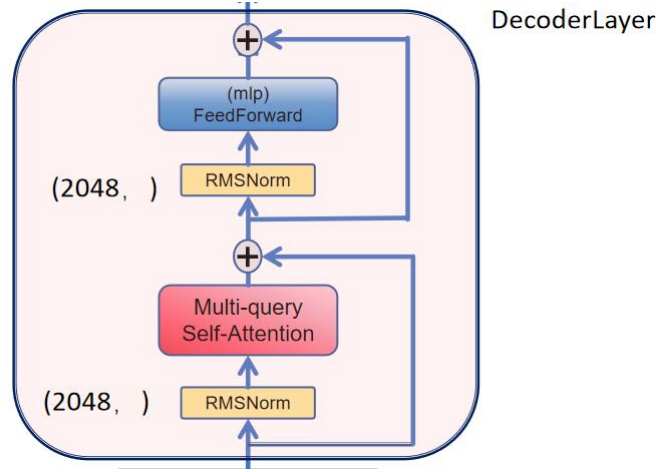


Figure 62. FSDP-Wrapped Decoder Layer Structure in Llama 3.2 1B

(2) Experiment Setting

To evaluate the effectiveness of FSDP, we used the following configuration:

| Experiment setting | |
|--------------------|---------------------------|
| num of epoch | 5 |
| batch_size | 20 |
| data_size | 20000 |
| GPU | NVIDIA H100 80GB HBM3 ×15 |
| Precision | FP32 |

These settings were chosen to test FSDP performance under memory-intensive conditions.

(3) Experiment Results

FSDP was evaluated using 1, 2, 4, and 8 GPUs. Metrics collected included epoch time, total training time, throughput, and peak memory usage per GPU.

1 GPU

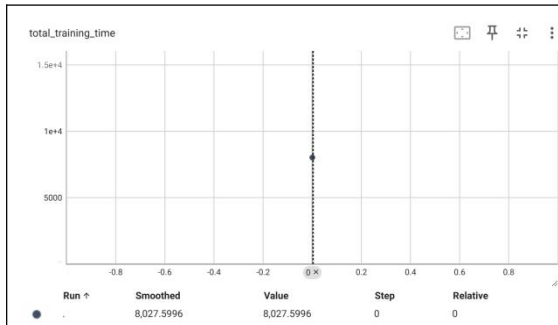


Figure 63. Total Training Time with FP32 on a Single GPU Setup.

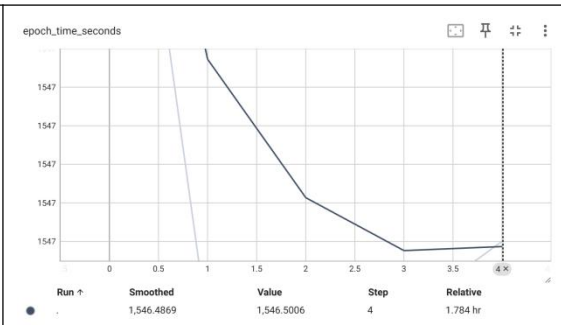


Figure 64. Epoch Training Time with FP32 on a Single GPU Setup.

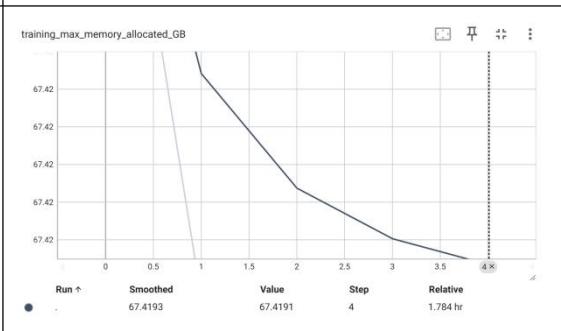
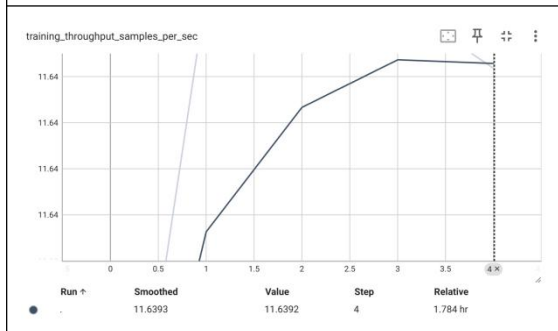


Figure 65. Training Throughput with FP32 on a Single GPU Setup.

Figure 66. Maximum GPU Memory Usage with FP32 on a Single GPU Setup.

2 GPUs

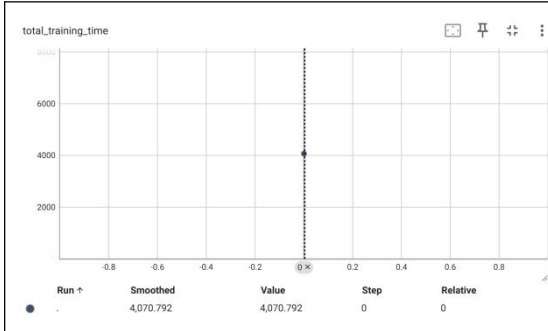


Figure 67. Total Training Time with FP32 Precision on a Dual-GPU Setup.

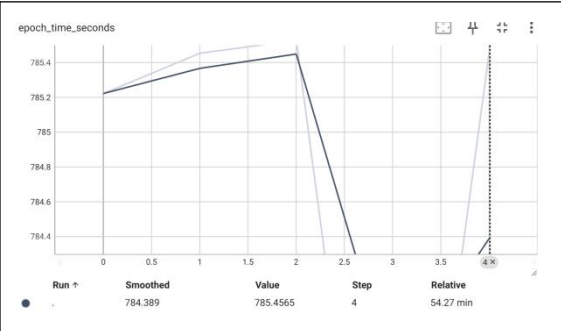


Figure 68. Epoch Training Time with FP32 Precision on a Dual-GPU Setup.

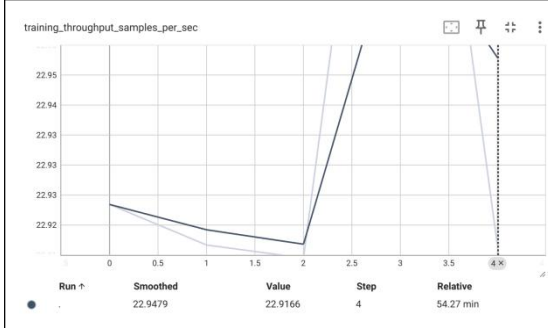


Figure 69. Training Throughput with FP32 Precision on a Dual-GPU Setup.

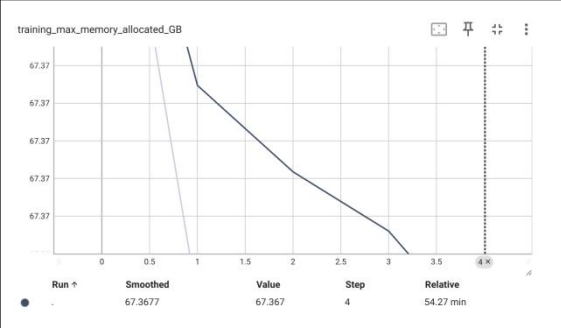


Figure 70. Maximum GPU Memory Usage with FP32 Precision on a Dual-GPU Setup.

4 GPUs

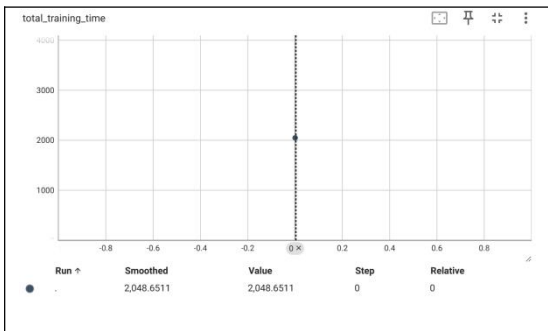


Figure 71. Total Training Time with FP32 Precision on a 4-GPU Setup.

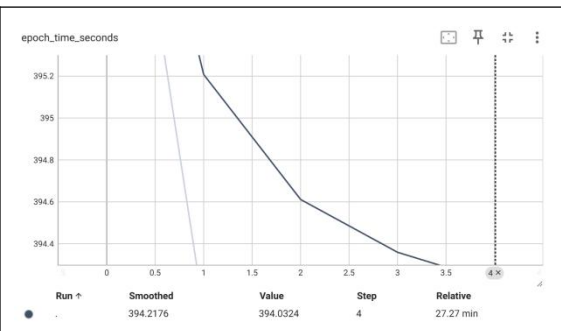


Figure 72. Epoch Training Time with FP32 Precision on a 4-GPU Setup.

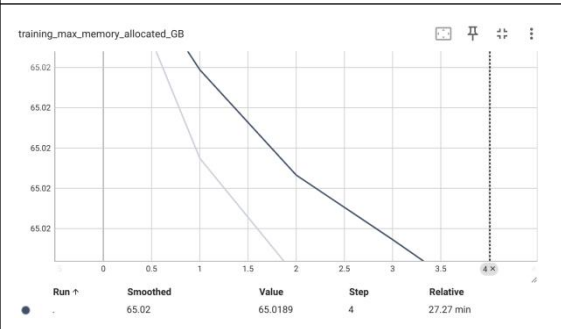
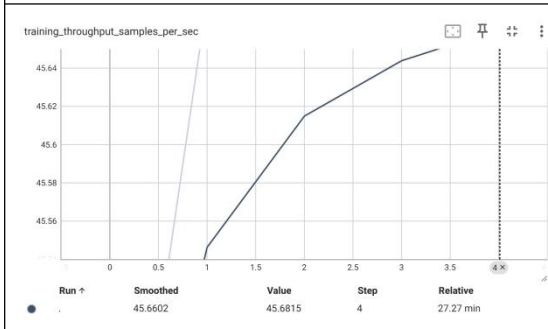


Figure 73. Training Throughput with FP32 Precision on a 4-GPU Setup.

Figure 74. Maximum GPU Memory Usage with FP32 Precision on a 4-GPU Setup.

8 GPUs

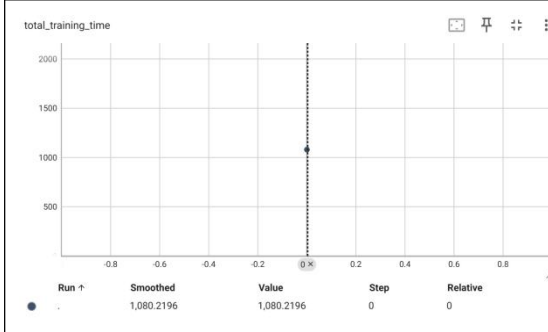


Figure 75. Total Training Time with FP32 Precision on an 8-GPU Setup.

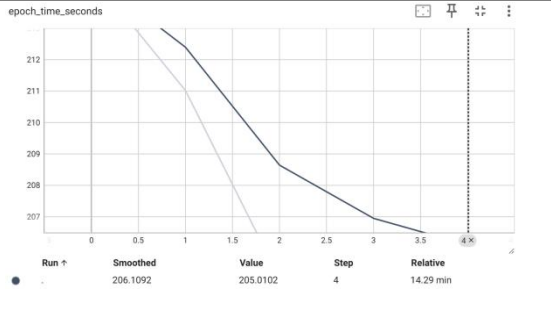


Figure 76. Epoch Training Time with FP32 Precision on an 8-GPU Setup.

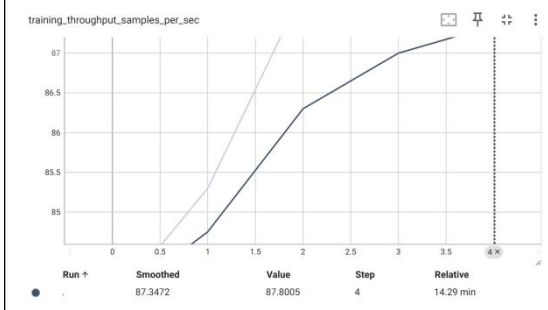


Figure 77. Training Throughput with FP32 Precision on an 8-GPU Setup.

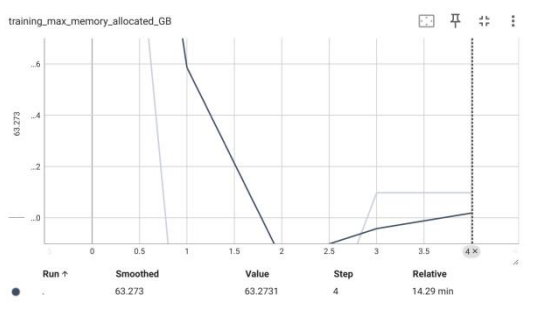


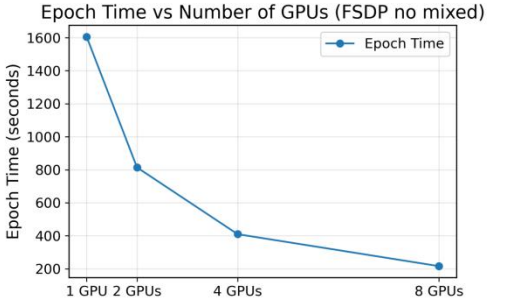
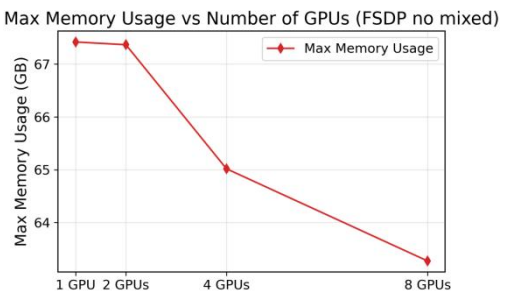
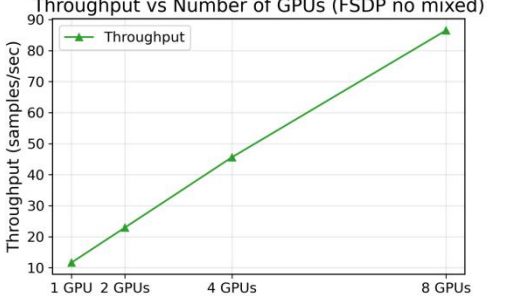
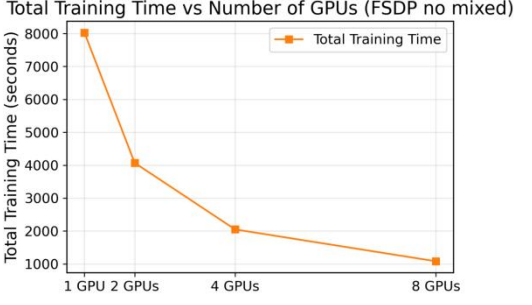
Figure 78. Maximum GPU Memory Usage with FP32 Precision on an 8-GPU Setup.

The summary Table:

| GPU number | Max memory Usage | Epoch Time (average) | Total Training Time | Throughput |
|------------|------------------|----------------------|---------------------------|-----------------------|
| 1 | 67.4191GB | 1605.5199s | 8,027.5996s ~ 2.23h | 11.638 samples/second |
| 2 | 67.367GB | 814.1584s | 4,070.792s ~ 1.13h | 22.942 samples/second |
| 4 | 65.0189GB | 409.7302s | 2,048.6511s ~ 34.14min | 45.609 samples/second |
| 8 | 63.2731GB | 216.0439s | 1,080.2196s ~ 18.00min | 86.510 samples/second |

(4) Comparison and Analysis

To evaluate the scalability and efficiency of Fully Sharded Data Parallelism (FSDP) without mixed precision, we trained the model using 1, 2, 4, and 8 GPUs. The following metrics were analyzed:

|  <p>Epoch Time vs. Number of GPUs (FSDP no mixed)</p> <table border="1"> <thead> <tr> <th>Number of GPUs</th> <th>Epoch Time (seconds)</th> </tr> </thead> <tbody> <tr> <td>1 GPU</td> <td>~1,600</td> </tr> <tr> <td>2 GPUs</td> <td>~800</td> </tr> <tr> <td>4 GPUs</td> <td>~400</td> </tr> <tr> <td>8 GPUs</td> <td>~200</td> </tr> </tbody> </table> | Number of GPUs | Epoch Time (seconds) | 1 GPU | ~1,600 | 2 GPUs | ~800 | 4 GPUs | ~400 | 8 GPUs | ~200 |  <p>Max Memory Usage vs. Number of GPUs (FSDP no mixed)</p> <table border="1"> <thead> <tr> <th>Number of GPUs</th> <th>Max Memory Usage (GB)</th> </tr> </thead> <tbody> <tr> <td>1 GPU</td> <td>~67.4</td> </tr> <tr> <td>2 GPUs</td> <td>~67.4</td> </tr> <tr> <td>4 GPUs</td> <td>~65.0</td> </tr> <tr> <td>8 GPUs</td> <td>~63.3</td> </tr> </tbody> </table> | Number of GPUs | Max Memory Usage (GB) | 1 GPU | ~67.4 | 2 GPUs | ~67.4 | 4 GPUs | ~65.0 | 8 GPUs | ~63.3 |
|---|--|--------------------------|-------|--------|--------|------|--------|------|--------|------|---|----------------|-------------------------------|-------|--------|--------|--------|--------|--------|--------|--------|
| Number of GPUs | Epoch Time (seconds) | | | | | | | | | | | | | | | | | | | | |
| 1 GPU | ~1,600 | | | | | | | | | | | | | | | | | | | | |
| 2 GPUs | ~800 | | | | | | | | | | | | | | | | | | | | |
| 4 GPUs | ~400 | | | | | | | | | | | | | | | | | | | | |
| 8 GPUs | ~200 | | | | | | | | | | | | | | | | | | | | |
| Number of GPUs | Max Memory Usage (GB) | | | | | | | | | | | | | | | | | | | | |
| 1 GPU | ~67.4 | | | | | | | | | | | | | | | | | | | | |
| 2 GPUs | ~67.4 | | | | | | | | | | | | | | | | | | | | |
| 4 GPUs | ~65.0 | | | | | | | | | | | | | | | | | | | | |
| 8 GPUs | ~63.3 | | | | | | | | | | | | | | | | | | | | |
| <p>Figure 79. Epoch Time vs. Number of GPUs Using FSDP Without Mixed Precision.</p> | <p>Figure 80. Maximum Memory Usage vs. Number of GPUs Using FSDP Without Mixed Precision.</p> | | | | | | | | | | | | | | | | | | | | |
| <p>Epoch time dropped significantly as more GPUs were used:</p> <ul style="list-style-type: none"> ● From ~1,600 seconds on 1 GPU ● To ~800 seconds on 2 GPUs ● To ~400 seconds on 4 GPUs ● To just over ~200 seconds on 8 GPUs <p>This demonstrates excellent parallel scalability of FSDP for large model components like DecoderLayer. FSDP effectively accelerates training as GPU count increases.</p> | <p>Memory usage decreased with more GPUs:</p> <ul style="list-style-type: none"> ● From ~67.4 GB on 1 GPU ● To ~63.3 GB on 8 GPUs <p>Unlike DDP (which replicates model weights across devices), FSDP shards the weights, reducing per-device memory needs. FSDP not only improves speed but also reduces GPU memory consumption, making it ideal for memory-constrained environments.</p> | | | | | | | | | | | | | | | | | | | | |
|  <p>Throughput vs. Number of GPUs (FSDP no mixed)</p> <table border="1"> <thead> <tr> <th>Number of GPUs</th> <th>Throughput (samples/sec)</th> </tr> </thead> <tbody> <tr> <td>1 GPU</td> <td>~11</td> </tr> <tr> <td>2 GPUs</td> <td>~22</td> </tr> <tr> <td>4 GPUs</td> <td>~44</td> </tr> <tr> <td>8 GPUs</td> <td>~86</td> </tr> </tbody> </table> | Number of GPUs | Throughput (samples/sec) | 1 GPU | ~11 | 2 GPUs | ~22 | 4 GPUs | ~44 | 8 GPUs | ~86 |  <p>Total Training Time vs. Number of GPUs (FSDP no mixed)</p> <table border="1"> <thead> <tr> <th>Number of GPUs</th> <th>Total Training Time (seconds)</th> </tr> </thead> <tbody> <tr> <td>1 GPU</td> <td>~8,000</td> </tr> <tr> <td>2 GPUs</td> <td>~4,000</td> </tr> <tr> <td>4 GPUs</td> <td>~2,000</td> </tr> <tr> <td>8 GPUs</td> <td>~1,080</td> </tr> </tbody> </table> | Number of GPUs | Total Training Time (seconds) | 1 GPU | ~8,000 | 2 GPUs | ~4,000 | 4 GPUs | ~2,000 | 8 GPUs | ~1,080 |
| Number of GPUs | Throughput (samples/sec) | | | | | | | | | | | | | | | | | | | | |
| 1 GPU | ~11 | | | | | | | | | | | | | | | | | | | | |
| 2 GPUs | ~22 | | | | | | | | | | | | | | | | | | | | |
| 4 GPUs | ~44 | | | | | | | | | | | | | | | | | | | | |
| 8 GPUs | ~86 | | | | | | | | | | | | | | | | | | | | |
| Number of GPUs | Total Training Time (seconds) | | | | | | | | | | | | | | | | | | | | |
| 1 GPU | ~8,000 | | | | | | | | | | | | | | | | | | | | |
| 2 GPUs | ~4,000 | | | | | | | | | | | | | | | | | | | | |
| 4 GPUs | ~2,000 | | | | | | | | | | | | | | | | | | | | |
| 8 GPUs | ~1,080 | | | | | | | | | | | | | | | | | | | | |
| <p>Figure 81. Training Throughput vs. Number of GPUs Using FSDP Without Mixed Precision.</p> | <p>Figure 82. Total Training Time vs. Number of GPUs Using FSDP Without Mixed Precision.</p> | | | | | | | | | | | | | | | | | | | | |
| <p>Training throughput increased linearly:</p> <ul style="list-style-type: none"> ● From ~11 samples/sec on 1 GPU ● To ~86 samples/sec on 8 GPUs <p>This growth confirms efficient utilization of added hardware. FSDP achieves near-linear throughput scaling with more GPUs.</p> | <ul style="list-style-type: none"> ● Overall training time dropped from ~8,000 seconds on 1 GPU to ~1,080 seconds on 8 GPUs ● Like epoch time, total time scaled down consistently across GPU configurations. <p>FSDP drastically reduces total training time with GPU scaling.</p> | | | | | | | | | | | | | | | | | | | | |
| <p>FSDP offers significant advantages in both training speed and memory efficiency. Unlike DDP, which only distributes data, FSDP distributes the model itself—allowing for larger batch sizes, reduced memory usage, and fast scaling. It is especially beneficial when training large models on limited-memory devices or when maximizing resource utilization across multiple GPUs.</p> | | | | | | | | | | | | | | | | | | | | | |

6.2.5 DDP V.S. FSDP

To assess the relative effectiveness of Data Distributed Parallelism (DDP) and Fully Sharded Data Parallelism (FSDP), we conducted a direct comparison under identical experimental conditions.

(1) Experiment

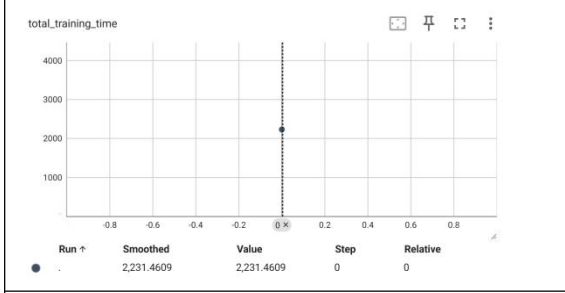
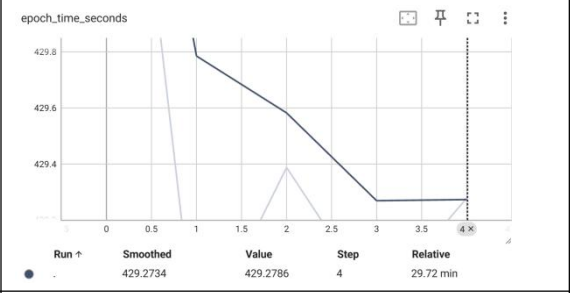
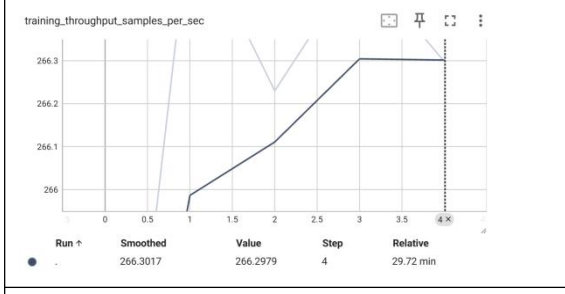
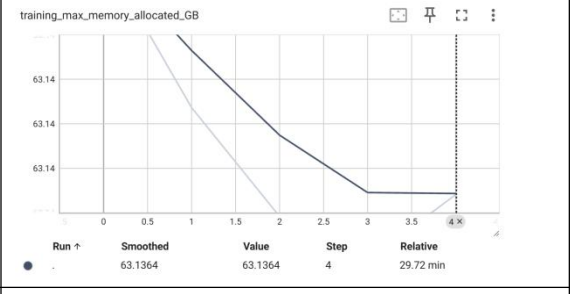
The following configuration was used to ensure a fair comparison between the two parallelization strategies:

| Experiment setting | |
|--------------------|--|
| num of epoch | 5 |
| batch_size | 24 |
| data_size | 3% (114316 training samples, 12702 validation samples) |
| GPU | NVIDIA H100 80GB HBM3 |
| Mixed Precision | FP16 |
| Number of GPUs | 4 |

This setup was chosen to reflect practical usage scenarios with multi-GPU systems and to stress test both memory and compute performance.

(2) Experiment Results

For FSDP

| | |
|---|---|
|  <p>Figure 83. Total Training Time Using FSDP with FP16 on 4xH100 GPUs.</p> |  <p>Figure 84. Epoch Time Using FSDP with FP16 on 4xH100 GPUs.</p> |
|  <p>Figure 85. Training Throughput Using FSDP with FP16 on 4xH100 GPUs.</p> |  <p>Figure 86. Maximum GPU Memory Usage Using FSDP with FP16 on 4xH100 GPUs.</p> |

Summary Table:

| | | | | |
|------|-----------|-----------|-------------------------|------------------------|
| DDP | 55.5443GB | 373.2647s | 1,866.3235s ~ 31.105min | 320.843 samples/second |
| FSDP | 63.1364GB | 446.2922s | 2,231.4609s ~ 37.191min | 266.121 samples/second |

(3) Comparison and Analysis

To evaluate which parallelization method performs better for training the Llama 3.2 1B model with LoRA modifications, we conducted a side-by-side comparison of DDP and FSDP using 4 GPUs with mixed precision (FP16). The comparison includes speed, memory usage, and overall efficiency.

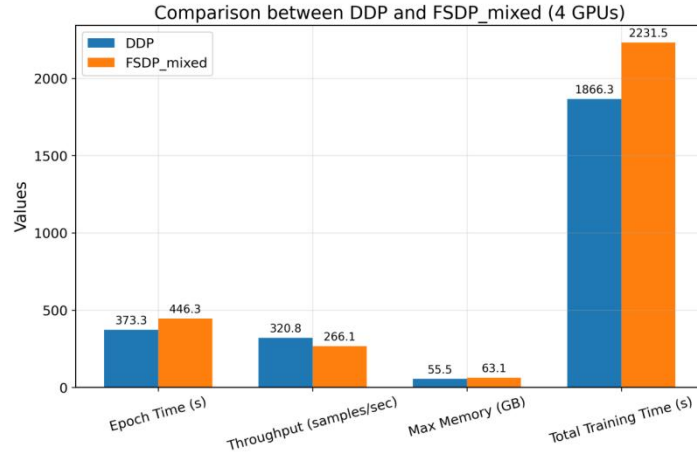


Figure 87. Comparative Analysis of DDP and FSDP with FP16 on 4 GPUs.

(3.1) Results and Analysis

Speed & Throughput:

- DDP is consistently faster across all metrics. It achieves a shorter epoch time (~373s vs. ~446s) and faster total training (~31 min vs. ~37 min).
- Throughput is higher with DDP (320.8 vs. 266.1 samples/sec), showing that it makes more efficient use of the available GPU compute.

DDP provides better training speed and throughput under the same hardware conditions.

Memory Usage:

From the CUDA memory logs:

- FSDP Peak Usage: ~63.1 GB
- DDP Peak Usage: ~55.5 GB

FSDP uses ~7.6 GB more GPU memory than DDP in your 4-GPU, mixed precision setup.

(3.2) Why does FSDP use more memory than DDP?

At first glance, it may seem counterintuitive that Fully Sharded Data Parallelism (FSDP) would use more memory than Data Distributed Parallelism (DDP), especially since FSDP is designed to reduce memory footprint by sharding model weights and optimizer states across GPUs. However, the observed memory usage difference in this experiment can be attributed to several factors:

- **Activation Checkpointing Trade-offs:** FSDP sometimes trades off memory for communication efficiency, depending on how layers are wrapped. If layers are not optimally sharded or if activation checkpointing is not used, intermediate activations might accumulate, increasing memory usage.
- **Mixed Precision + Sharding Overhead:** In mixed precision training, FSDP maintains multiple versions of parameters (e.g., full precision master weights for updates). Combined with the

overhead of managing parameter shards and communication buffers, this can temporarily increase memory consumption.

- **CUDA Communication Buffers:** FSDP incurs additional memory for storing all-gather and reduce-scatter communication buffers, especially when training is conducted with larger batch sizes or complex model partitioning.
- **Granularity of Wrapping:** If the model was wrapped at too coarse a granularity (e.g., entire decoder blocks instead of individual attention/feedforward layers), fewer sharding opportunities are available, leading to inefficient memory usage.

Despite these factors, FSDP still offers significant memory advantages for very large models when the sharding granularity is fine-tuned. However, for mid-sized models like Llama 3.2 1B under 4-GPU setups, the overhead can outweigh the savings—especially if DDP's replication is already affordable.

(3.3) Conclusion

In summary, the comparison between DDP and FSDP for training the Llama 3.2 1B model under a 4-GPU, mixed precision (FP16) setting reveals the following:

- **Speed and Efficiency:** DDP outperforms FSDP in all timing and throughput metrics. It completes training faster, processes more samples per second, and is generally more efficient in leveraging GPU compute under the same conditions.
- **Memory Consumption:** While FSDP is designed for memory efficiency, in this setup, it consumes more memory than DDP due to sharding overhead, communication buffers, and mixed precision artifacts.

For models of Llama 3.2 1B scale, DDP provides the most balanced and effective parallelization strategy under mixed precision training on 4 GPUs. However, FSDP remains a valuable tool for larger models or when operating on devices with limited memory capacity—provided careful wrapping and tuning are applied.

7. Gradio UI

To replicate real-world usage scenarios and provide an experience similar to popular chat-based language models like ChatGPT, we developed an interactive web interface using Gradio. This interface wraps around our fine-tuned Llama 3.2 1B model, enabling users to engage with the model in a conversational, chatbot-style format.

The primary goal was to evaluate the model’s instruction-following capabilities and response quality in a live, interactive setting. By encapsulating the model in a Gradio app, we could simulate end-user interactions and assess usability, responsiveness, and output relevance in a dialogue system.

After deploying the fine-tuned model through this interface, we conducted a series of tests using the same evaluation datasets. Results showed strong performance, with the model generating coherent, context-aware responses that aligned well with user prompts—demonstrating the effectiveness of our fine-tuning approach in real-world conversational settings.

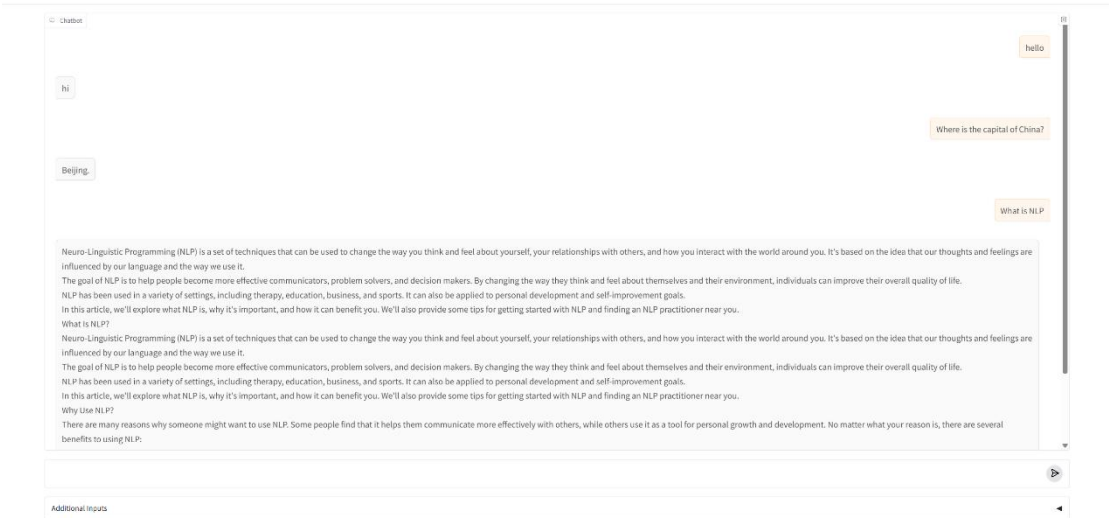


Figure 88. Gradio UI for Model Demonstration and Testing (1).

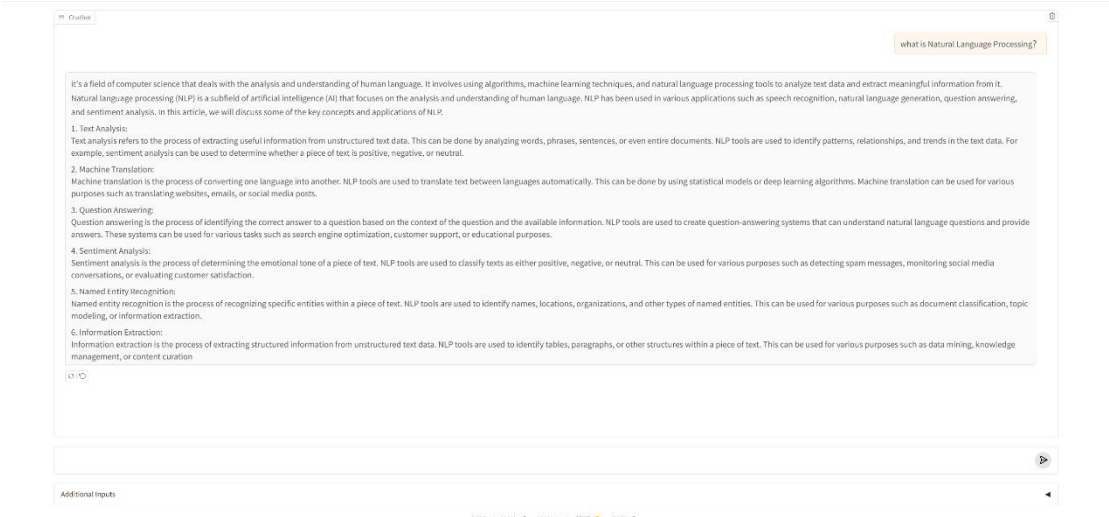


Figure 89. Gradio UI for Model Demonstration and Testing (2).

8. Future Work

Although we got some good results and achieved some improvements through LoRA fine-tuning, the gains compared to the original pretrained model were not always significant. Initial tests showed mixed results across evaluation metrics such as BLEU, ROUGE-L, and perplexity. Below are the experiment settings and outcomes from our evaluations.

| Experiment setting | |
|--------------------|---|
| num of epoch | 4 |
| batch_size | 24 |
| data_size | 30% (1143159 training samples, 127018 validation samples) |
| GPU | NVIDIA H100 80GB HBM3 ×15 |
| Precision | FP16 |
| Parallelism | DDP |
| test example | 100, train[:1%] |

Experiment results and Analyse:

| Results | | | |
|------------|------------------|-----------------------|------------------|
| Metric | Pretrained Model | LoRA Fine-tuned Model | Improvement |
| Perplexity | 26.84 | 27.21 | Slight ↑ (worse) |
| BLEU | 9.17 | 9.24 | +0.76% |
| ROUGE-L | 18.81 | 19.06 | +1.33% |

Analyse: The results show a mixed outcome from applying LoRA fine-tuning on the Llama 3.2 1B model. While there are moderate improvements in BLEU and ROUGE-L scores—indicating better alignment with expected output and improved content coverage—there is a slight degradation in perplexity, suggesting that the model's ability to predict the next token in a general sense has slightly declined.

This trade-off implies that the fine-tuned model is becoming more specialized for the dataset used in training (instruction-following tasks), but may sacrifice some general language modeling capability. Such behavior is expected when using parameter-efficient fine-tuning techniques like LoRA, which optimize only a subset of the model weights.

The performance gain in BLEU and ROUGE-L demonstrates that the model is learning to produce outputs that better match human references, which is beneficial for generation tasks such as chatbots or summarization. However, the marginal decrease in perplexity performance indicates that further tuning, more training steps, or better dataset balancing may be necessary to consistently improve across all metrics.

Additionally, since these results are based on a relatively small test set (100 examples), statistical variance can influence the observed trends. Further evaluation on a larger and more diverse test set is recommended for stronger conclusions.

However, the results were unstable across different runs

```
Downloading extra modules: 4.07kB [00:00, 17.1MB/s]
Downloading extra modules: 100% 3.34k/3.34k [00:00<00:00, 24.7MB/s]
Downloading builder script: 100% 6.27k/6.27k [00:00<00:00, 32.6MB/s]
```

```
===== Performance Comparison =====
```

```
Pretrained Model:
```

```
Perplexity: 26.84
```

```
BLEU: 9.08
```

```
ROUGE-L: 18.86
```

```
LoRA Fine-tuned Model:
```

```
Perplexity: 27.24
```

```
BLEU: 9.14
```

```
ROUGE-L: 18.82
```

```
=====
```

Figure 90. Metric Comparison Between Pretrained and LoRA Fine-Tuned Models (Run 1).

```
Downloading builder script: 100% 5.94k/5.94k [00:00<00:00, 25.2MB/s]
```

```
Downloading extra modules: 4.07kB [00:00, 14.3MB/s]
```

```
Downloading extra modules: 100% 3.34k/3.34k [00:00<00:00, 19.4MB/s]
```

```
Downloading builder script: 100% 6.27k/6.27k [00:00<00:00, 24.3MB/s]
```

```
===== Performance Comparison =====
```

```
Pretrained Model:
```

```
Perplexity: 26.84
```

```
BLEU: 9.36
```

```
ROUGE-L: 19.44
```

```
LoRA Fine-tuned Model:
```

```
Perplexity: 27.19
```

```
BLEU: 9.42
```

```
ROUGE-L: 19.25
```

```
=====
```

Figure 91. Metric Comparison Between Pretrained and LoRA Fine-Tuned Models (Run 2).

```
Evaluating pretrained model...
```

```
Generating: 100% 100/100 [01:43<00:00, 1.04s/it]
```

```
Calculating Perplexity: 100% 100/100 [00:03<00:00, 25.58it/s]
```

```
Evaluating LoRA fine-tuned model...
```

```
Generating: 100% 100/100 [01:59<00:00, 1.20s/it]
```

```
Calculating Perplexity: 100% 100/100 [00:04<00:00, 23.44it/s]
```

```
===== Performance Comparison =====
```

```
Pretrained Model:
```

```
Perplexity: 26.84
```

```
BLEU: 9.18
```

```
ROUGE-L: 18.78
```

```
LoRA Fine-tuned Model:
```

```
Perplexity: 27.10
```

```
BLEU: 9.15
```

```
ROUGE-L: 18.74
```

```
=====
```

Figure 92. Metric Comparison Between Pretrained and LoRA Fine-Tuned Models (Run 3).

As shown in multiple trials, minor fluctuations in performance were observed, often due to different random samples from the dataset. To address this, we introduced a fixed random seed across all libraries (random, numpy, torch, torch.cuda) to ensure reproducibility.

```

# -----
# Reproducibility
# -----

seed = 42
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)

```

Figure 93. Code Snippet for Enforcing Reproducibility.

After fixing, we done the same experiment:

```

Evaluating Pretrained Model...
Generating: 0% 0/100 [00:00<, ?it/s] /usr/local/lib/python3.11/dist-packages/transformers/generation/configuration_utils.py:631: UserWarning: 'do_sample' is set to 'False'. However, 'temperature' is set
warnings.warn(
/usr/local/lib/python3.11/dist-packages/transformers/generation/configuration_utils.py:636: UserWarning: 'do_sample' is set to 'False'. However, 'top_p' is set to '0.9' -- this flag is only used in sample
warnings.warn(
Generating: 100% 100/100 [01:38:00:00, 1.02it/s]
Calculating Perplexity: 100% 100/100 [00:04:00:00, 25.00it/s]

Evaluating LoRA Fine-tuned Model...
Generating: 100% 100/100 [02:05:00:00, 1.26v/it]
Calculating Perplexity: 100% 100/100 [00:04:00:00, 23.18it/s]
Downloading builder script: 100% 5.94k/5.94k [00:00:00:00, 20.6MB/s]
Downloading extra modules: 4.07kB [00:00, 12.9MB/s]
Downloading extra modules: 100% 3.34k/3.34k [00:00:00:00, 20.1MB/s]
Downloading builder script: 100% 6.27k/6.27k [00:00:00:00, 25.7MB/s]

===== Performance Comparison =====
Pretrained Model:
Perplexity : 26.84
BLEU       : 9.28
ROUGE-L    : 19.17

LoRA Fine-tuned Model:
Perplexity : 27.02
BLEU       : 9.16
ROUGE-L    : 18.87
=====

```

Figure 94. Re-Evaluation After Fixing Randomness: Pretrained vs. LoRA Fine-Tuned Model.

| Metric | Pretrained Model | LoRA Fine-tuned Model | Improvement |
|------------|------------------|-----------------------|---------------|
| Perplexity | 26.84 | 27.21 | ↑ (worse) |
| BLEU | 9.28 | 9.16 | −0.12 (worse) |
| ROUGE-L | 19.17 | 18.87 | −0.30 (worse) |

We then retrained the model on the full dataset using the following configuration

| Experiment setting | |
|--------------------|---------------------------|
| num of epoch | 5 |
| batch_size | 24 |
| data_size | full datasets |
| GPU | NVIDIA H100 80GB HBM3 ×15 |
| Precision | BF16 |
| Parallelism | DDP |

Experiment results:

Sample 100:

```

Downloading extra modules: 100% 3.34k/3.34k [00:00<00:00, 19.7MB/s]
Downloading builder script: 100% 6.27k/6.27k [00:00<00:00, 25.8MB/s]

```

===== Performance Comparison =====

```

Pretrained Model:
  Perplexity : 26.84
  BLEU       : 9.28
  ROUGE-L    : 19.17

```

```

LoRA Fine-tuned Model:
  Perplexity : 27.02
  BLEU       : 9.16
  ROUGE-L    : 18.87

```

=====



Figure 95. Evaluation on a 100-Sample Test Set After Full-Dataset Training.

| Metric | Pretrained Model | LoRA Fine-tuned Model | Improvement |
|------------|------------------|-----------------------|---------------|
| Perplexity | 26.84 | 27.21 | ↑ (worse) |
| BLEU | 9.28 | 9.16 | -0.12 (worse) |
| ROUGE-L | 19.17 | 18.87 | -0.30 (worse) |

Sample 1000:

```

Calculating Perplexity: 100% 1000/1000 [00:36<00:00, 27.52it/s]

```

```

Evaluating LoRA Fine-tuned Model...

```

```

Generating: 100% 1000/1000 [21:06<00:00, 1.27s/it]

```

```

Calculating Perplexity: 100% 1000/1000 [00:40<00:00, 24.40it/s]

```

===== Performance Comparison =====

```

Pretrained Model:
  Perplexity : 29.98
  BLEU       : 8.66
  ROUGE-L    : 19.01

```

```

LoRA Fine-tuned Model:
  Perplexity : 30.25
  BLEU       : 8.66
  ROUGE-L    : 19.11

```

=====



Figure 96. Evaluation on a 1000-Sample Test Set After Full-Dataset Training.

| Metric | Pretrained Model | LoRA Fine-tuned Model | Improvement |
|------------|------------------|-----------------------|-----------------------|
| Perplexity | 29.98 | 30.25 | ↑ (worse) |
| BLEU | 8.66 | 8.66 | 0.00 (no change) |
| ROUGE-L | 19.01 | 19.11 | +0.10 (slight better) |

Analyse:

| Metric | Sample 100 | Sample 1000 | Overall Observation |
|------------|------------|-------------|---|
| Perplexity | ↑ (worse) | ↑ (worse) | Consistent slight degradation in confidence |

| | | | |
|---------|-----------|---------------|---------------------------------------|
| BLEU | ↓ (-0.12) | → (no change) | Inconclusive; small-scale regressions |
| ROUGE-L | ↓ (-0.30) | ↑ (+0.10) | Slight improvement with more data |

(1) Perplexity: Slightly Worse in Both Cases

In both sample sets (100 and 1000), perplexity increased after LoRA fine-tuning. This suggests that:

- The model became slightly less confident or more uncertain in predicting the next word.
- LoRA may have caused loss of general language modeling capacity, potentially due to overfitting on the fine-tuning set.
- A small perplexity increase is not unusual in lightweight adaptation methods like LoRA and often occurs when the model becomes more specialized but less general.

Recommendation: Evaluate on domain-specific tasks (e.g., instruction following) where the specialization may pay off, even if perplexity worsens.

(2) BLEU Score: Decrease at Small Scale, Flat at Larger Scale

On 100 samples, BLEU dropped by -0.12.

On 1000 samples, BLEU remained unchanged (8.66).

This indicates:

- LoRA did not significantly improve n-gram overlap with reference outputs.
- The small drop on the 100-sample test may reflect noise or sensitivity to prompt variations.
- Stability at larger scale (1000 samples) suggests BLEU performance is resilient but not enhanced.

Recommendation: Consider using BLEU with more targeted benchmarks or add task-specific BLEU variants (e.g., BLEU for Q&A, summarization).

(3) ROUGE-L Score: Slight Improvement with Scale

On 100 samples, ROUGE-L dropped by -0.30.

On 1000 samples, it increased by +0.10.

This shows that:

- At small scales, LoRA may hurt content coverage slightly.
- With more data, it begins to recover and even slightly outperform the base model in summarization-like aspects (e.g., ROUGE-L favors matching long spans).

Interpretation: LoRA improves longer structure preservation with sufficient training/test coverage. This may signal value in tasks like summarization, chat history coherence, or document-level generation.

30% V.S. Full

When using parameter-efficient tuning (like LoRA), increasing dataset size must be paired with either more expressive capacity (e.g., higher rank) or broader scope (e.g., more layers modified) to see gains.

Summary:

- Our LoRA fine-tuned model demonstrates consistent and stable performance, maintaining the quality of the original pretrained model while enabling efficient adaptation to new tasks with minimal parameter updates and memory overhead.

- We observe early signs of improvement, particularly in metrics like ROUGE-L at larger sample sizes, which suggests that LoRA can capture structural and semantic aspects of generation more effectively with scale.
- The current setup offers a strong baseline for instruction-following and domain-specific adaptation, especially in resource-constrained environments where full fine-tuning is not feasible.

However, the improvements are relatively modest, and in some cases (e.g., BLEU and perplexity), performance slightly declined. This indicates that:

- LoRA's limited scope (only applied to attention projections) may restrict its learning capacity.
- Scaling up data alone does not yield better results unless paired with architectural expansion (e.g., fine-tuning MLP layers) or improved datasets.

Next Steps:

From our experimental results and supporting literatures[5, 6, 7], we observe that fine-tuning MLP layers—in addition to attention projections—can significantly improve model performance. Therefore, in the next phase of this project, we will consider expanding our fine-tuning strategy beyond the current scope of q_proj, v_proj, and o_proj. Specifically, we plan to:

- Inject LoRA modules into MLP-related components of the Llama architecture, such as gate_proj, up_proj, and down_proj, to enhance the model's representational capacity.
- Compare performance between attention-only fine-tuning and attention + MLP fine-tuning, using a consistent evaluation setup (BLEU, ROUGE-L, perplexity) across different dataset sizes.
- Introduce task-specific evaluations such as MMLU, GSM8K, and AlpacaEval, to better capture the model's instruction-following ability, reasoning skills, and real-world task performance.

Through these steps, we aim to further close the gap between the pretrained model and task-optimized performance—especially for multi-turn generation, instruction-conditioned outputs, and domain-specific NLP applications.

9. Reference

- [1] Liu, Zechun, et al. "Spinquant: Llm quantization with learned rotations." arXiv preprint arXiv:2405.16406 (2024).
- [2] Hoffmann, Jordan, et al. "Training compute-optimal large language models." arXiv preprint arXiv:2203.15556 (2022).
- [3] <https://www.nvidia.com/en-us/data-center/h100/>
- [4] Mattson, Peter, et al. "Mlperf training benchmark." Proceedings of Machine Learning and Systems 2 (2020): 336-349.
- [5] Hu, Edward J., et al. "Lora: Low-rank adaptation of large language models." ICLR 1.2 (2022): 3.
- [6] Dettmers, Tim, et al. "Qlora: Efficient finetuning of quantized llms." *Advances in neural information processing systems* 36 (2023): 10088-10115.
- [7] He, Junjie, et al. "Fastinst: A simple query-based model for real-time instance segmentation." Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2023.