**HPC**

**FINAL PROJECT REPORT**

**Fine-Tuning of Large Language Models on HPC Clusters**

**Name: - Nihar Sanjaysinh Dabhi & Daxil Vala**

**Team Number: - 10**

**Professor: - Prof. Handan Liu**

**Introduction**

**What is Fine-tuned LLM: -**

**A diagram of a parrot

AI-generated content may be incorrect.**

Convert a base LLM for specific application or task by train it with the specific kind of dataset which affects the base LLM’s weight and bias, this whole makes new fine-tuned LLM. Initially LLM always train on all different dataset which makes it general purpose application, but sometime this base LLM fails when it’s comes specific application.

**Motivation: -**

As we know now days LLM contributes to Artificial Intelligence at large scale, but there some field where LLM cannot perform as excepted, for instance when it’s comes to pre-trained transformer chat bots it’s often failed to catch human response in specific terms. By giving it specific dataset to train which is not only teach LLM what to response, but in much better way!!!

**Background: -**

Fine-tuning large language models (LLMs) is the process of adapting a pretrained model to perform better on specific tasks or align more closely with human preferences. While LLMs are initially trained on vast, general-purpose text data, they often require fine-tuning to generate more useful, safe, and contextually appropriate responses. This is typically done through supervised fine-tuning using human-written prompt-response pairs or through preference-based methods like Reinforcement Learning from Human Feedback (RLHF), Direct Preference Optimization (DPO), or Online RLHF (ORPO), where the model learns to prefer responses rated better by humans. Fine-tuning enhances the model’s ability to follow instructions, reduces toxic or unhelpful outputs, and aligns its behavior more closely with desired use cases.

**Goal: -**

Our goal is to make a fine-tuned LLM for “**ORPO**: Online Reinforcement Learning from Preferences” by train it on multiple GPU to reduce the time and make the accuracy up to the benchmark.

**Methodology**

**Main Algorithms/Methods: -**

1. **Accelerator: -**

In this project we make a use of accelerator, accelerator developed by hugging face, it is used for large artificial intelligence and machine learning models to speed up the process, accelerator simply divide the workflow across different GPUs or CPUs setup,

Since our project contains large model (around 2 GB Model) to fine tuning the given large language model with large dataset (around 127 MB CSV file) we need accelerator for this purpose so we can speed up the process than the usual time limit, also accelerator automatically divide our model and large csv file into different chosen GPU which take from our HPC Cluster. This library will basically combine the different CPUs or GPUs together like for example it run whole experiment with only one GPU then with the 2 GPUs so on and so forth.

1. **Data Loader: -**

Data Loader is important function in this project since we use accelerator library after we load and splitting the data from the hugging face, we have to efficiently load the that into training section we created batches so data can go one by one efficiently into model, this data loader act as bridge between model and dataset.

After we split and load the data, we want to make sure that each data batch should be consistent, so for we write function “collate\_tokenize” and pass data loader function as parameter, it will take care of combining the ‘prompt’ and ‘question field’ together to ensure that each example goes in ‘proper format’ for training the model. this crucial because we must maintain the ‘tensor’ shape across the training section.

The DataLoader itself is then instantiated with this dataset split, a batch size, shuffling enabled for better generalization during training, and the custom collate\_fn. During training, it yields these tokenized batches one at a time to the training loop. This allows the model to receive data in manageable chunks rather than trying to process the entire dataset at once, which would be impractical, especially for large-scale datasets or memory-constrained environments. Overall, DataLoader helps streamline the data handling process, enabling scalable and efficient model training.

1. **Mixed Precision training and Quantization: -**

In this project we used mixed precision and quantization to reduce the memory consumption during this training session so for that we used mixed precision of 16-bit and 32-bit floating point during training we wrapped our configuration related to the precision into “bits and bytes config ” variable.

The hugging face accelerator handle most of the low level details under the hood, it is automatically casting operation according to precision while ensuring numerical stability, by this you can smoothly run large model or/and large dataset without having memory limit constraint.

Mixed precision work smoothly with the dataloader by accepting batches of tensor that already been properly tokenized. Once the input tensors are loaded into memory and passed into the model, the model and the underlying accelerator infrastructure determine how to handle data in reduced precision, without the need for the developer to manually cast tensors. This makes training both faster and more resource-efficient.

Quantization, on the other hand, refers to reducing the **precision of the model weights themselves**, not just during computation but also in memory. In your script, this is achieved using the BitsAndBytesConfig. This tells the transformers library to load the model in **4-bit quantized format** using the library. This drastically reduces the memory footprint of the model, allowing it to fit into VRAM even when using relatively large transformer architectures.

By using quantization, you're making it possible to train or fine-tune large models on smaller hardware setups or even multiple such models concurrently. Since the quantized model might have specific numeric constraints, using the collate\_tokenize function to create well-structured and consistent input batches ensures that the model can still process the data effectively without running into type mismatches or memory issues.

1. **Gradient Backpropagation: -**

In this code, Gradient Backpropagation is used during the training loop to optimize the model's parameters based on the loss computed for each batch. The purpose of backpropagation is to update the weights of the model in such a way that the loss function is minimized, improving the model's performance on the given task.

After computing the loss by passing the input through the model, the accelerator.backward(loss) function is called. This is where the backpropagation step occurs. It computes the gradients of the loss with respect to each parameter in the model. Essentially, the gradients represent how much each parameter contributed to the error in the model’s predictions. These gradients are calculated using the chain rule of calculus, allowing for the efficient calculation of gradients even in deep neural networks.

Once the gradients are calculated, the optimizer (optimizer.step()) updates the model’s weights in the direction that minimizes the loss. The optimizer, in this case, uses the AdamW algorithm, which adjusts the learning rate for each parameter based on estimates of first and second moments of the gradients, making it more efficient in navigating the loss landscape.

1. **Synchronized Optimizer and Scheduler: -**

the weight update process represents a critical synchronization point that ensures model convergence across the entire computational cluster. Following the synchronized gradient aggregation, the optimizer.step() method executes a coordinated parameter update that applies the accumulated gradients to the model weights according to the specified optimization algorithm's update rule, whether it be SGD, Adam, or another variant. This operation occurs simultaneously across all participating devices, maintaining perfect parameter consistency throughout the distributed system. The learning rate scheduler, which dynamically adjusts the optimization step size based on training progress, is likewise stepped in perfect lockstep across the device fleet, ensuring that all model replicas follow identical learning trajectories regardless of their physical location in the computing infrastructure. This synchronized update mechanism, strategically positioned within the training loop immediately after gradient backpropagation and aggregation, forms the foundation of reliable distributed training by guaranteeing that every model replica experiences identical weight modifications despite processing different data partitions, ultimately enabling linear scaling of training throughput with additional computational resources while preserving the mathematical properties of the underlying optimization process.

**Brief Code Overview: -**

This python demonstrates how you can fine tune Base LLM for dataset for direct preference optimize, where we used “LLAMA 3-8 1B” base LLM model to fine tune it, it around have 2GB size while our data set is about 127 MB. In this project our focus is to “how to divide our in such way that we do parallelism “with data as well for the model so within less time we can train the model with same accuracy as before.

**Steps: -**

**1 Accelerator Initialization and WandB Integration**

the code starts by initializing the accelerator object, which means this accelerator object will be responsible for distributing the model and training dataset on different CPUs and GPUs, and therefore it can significantly reduce the time to train. Also, we initialized the weight and bias integration so we can capture the real time value movement.

**2 Model and Tokenizer Loading with Quantization**

To reduce memory footprint and improve speed, the model is loaded using Hugging Face’s 4-bit quantization via the BitsAndBytesConfig. The model in use is meta-llama/Llama-3.2-1B-Instruct, and it is loaded along with its tokenizer using an authentication token. Padding is configured using the model's end-of-sequence token to avoid padding errors during batch processing.

**3 Dataset Preparation and Tokenization**

The script loads a publicly available ORPO fine-tuning dataset from Hugging Face (mlabonne/orpo-dpo-mix-40k) and limits the training set to a small subset (500 samples) for quicker experimentation. A collate\_tokenize function is defined to dynamically tokenize inputs. It concatenates prompts and questions, tokenizes them using the model's tokenizer, and pads/truncates sequences for uniformity in batch sizes — critical for distributed setups.

**4 Data Loading and Optimizer Setup**

A PyTorch DataLoader is initialized with shuffled training data, and the custom collate function. The optimizer used is AdamW, well-suited for transformer models, along with a StepLR learning rate scheduler that decays the learning rate every 1000 steps to stabilize training.

**5 Training Loop**

The model is trained for two epochs. Each batch is passed through the model with input\_ids and attention\_mask, and a causal language modeling loss is computed using the same input\_ids as both input and label. The loss is backpropagated using accelerator.backward(), followed by optimizer and scheduler steps. Progress is printed after each epoch.

**6 Model Saving and Cleanup**

Once training is complete, the script saves the model checkpoint if the current process is the main one (important in distributed training). It prints the total training time, clears memory by deleting the model and emptying the CUDA cache, and then performs inter-process cleanup with torch.cuda.ipc\_collect().

**System Architecture**

**A diagram of a software development process

AI-generated content may be incorrect.**

The architecture is built to optimize the fine-tuning of large language models on modern GPU hardware using efficient, scalable techniques. At the core is HuggingFace’s Accelerate, which abstracts complex distributed training operations such as device placement, data parallelism, and mixed-precision computation, making multi-GPU training seamless. The model selected—Meta’s LLaMA 3.2B Instruct—is loaded with 4-bit quantization using BitsAndBytesConfig, significantly reducing memory footprint while maintaining performance. Tokenization and padding are handled consistently via AutoTokenizer, ensuring input data is compatible with batch processing and distributed learning. The dataset, sourced from HuggingFace Datasets (mlabonne/orpo-dpo-mix-40k), is preprocessed using a custom collation function and passed through PyTorch’s DataLoader for batching and shuffling. Optimization is driven by AdamW with a scheduled learning rate decay, maintaining training stability. The system is fully integrated with Weights & Biases (wandb) for real-time experiment tracking, capturing metrics like loss, speed, and GPU utilization. Finally, model saving is performed exclusively by the main process to avoid race conditions, followed by an explicit cleanup of GPU memory, ensuring a resource-efficient shutdown. This modular and robust architecture supports both experimentation and production-level scaling for fine-tuning LLMs.

**Dataset Description**

Link: - [dataset](https://huggingface.co/datasets/mlabonne/orpo-dpo-mix-40k)

Size: - 127 M.B

Format: - CSV

The ORPO-DPO-Mix-40k dataset serves as the foundation for the fine-tuning process, consisting of approximately 44,000 curated data records of human-AI interactions stored in CSV format with a total size of 127MB. Each record contains a comprehensive interaction triplet: the original human prompt (question), a preferred response that aligns with human expectations, and a rejected response that demonstrates suboptimal answering patterns. This specialized dataset is specifically designed for alignment-focused fine-tuning of large language models, enabling the Llama-3.2-1B-Instruct model to better distinguish between high-quality and low-quality outputs by learning from human preference patterns. The training architecture leverages these paired examples to enhance the model's ability to generate responses that exhibit key human-aligned qualities—including safety considerations, helpfulness, factual accuracy, and contextual relevance—ultimately producing an inference model that more reliably generates content matching human expectations and ethical guidelines while effectively addressing the core information needs presented in user queries.

**Results And Analysis**

**A graph with numbers and a line

AI-generated content may be incorrect.**

**Execution Time Results and Analysis: -**

The measured execution times show consistent reduction with additional GPUs: 1837.41s (1 GPU), 1486.28s (2 GPUs, -19.1%), 1262.93s (3 GPUs, -15.0% vs 2 GPUs), and 928.46s (4 GPUs, -26.5% vs 3 GPUs). Notably, the largest absolute time reduction (334.47s) occurs between 3 and 4 GPUs, suggesting the workload reaches sufficient scale to better utilize four-way parallelism. However, the diminishing percentage improvements at 2-3 GPUs (19.1% → 15.0%) indicate initial scalability limits, possibly from fixed overheads that become relatively more significant with smaller per-GPU workloads.

**A graph with a line and numbers

AI-generated content may be incorrect.**

**Speedup Results and Analysis**

The achieved speedups progress from 1.24× (2 GPUs) to 1.45× (3 GPUs) and 1.98× (4 GPUs), falling short of ideal linear scaling (2×, 3×, 4× respectively). The 4-GPU configuration shows particularly notable deviation, delivering only 49.5% of the ideal speedup. This pattern suggests two operational regimes: (1) a relatively efficient 2-3 GPU range where speedups scale at ≈72% of ideal, and (2) a 4-GPU regime where efficiency drops to ≈50%, likely indicating communication bottlenecks or memory bandwidth saturation becoming dominant factors.

**A graph with a green line

AI-generated content may be incorrect.**

**Efficiency Results and Analysis**

Parallel efficiency measures confirm these trends numerically: 62% (2 GPUs), 48% (3 GPUs), and 49% (4 GPUs). The stability between 3-4 GPUs (within 1 percentage point) despite adding 33% more resources implies the system reaches a throughput plateau. This plateau occurs at ≈49% efficiency, meaning each additional GPU beyond 3 provides half its theoretical potential. The 2-GPU configuration emerges as the most efficient (62%), making it potentially optimal for energy-conscious deployments where absolute performance is secondary to resource utilization.

**Conclusion**

**The fine-tuning of large language models (LLMs) on HPC clusters presents a powerful approach to adapting general-purpose models for specialized tasks, significantly enhancing their performance in domain-specific applications. This project successfully demonstrated the effectiveness of distributed training techniques in optimizing the fine-tuning process of the Llama-3.2-1B-Instruct model using the ORPO-DPO-Mix-40k dataset. By leveraging Hugging Face’s Accelerate framework, the implementation efficiently managed multi-GPU training, reducing computational overhead while maintaining model accuracy. Key optimizations such as 4-bit quantization (NF4), mixed-precision training (bfloat16), and gradient synchronization across distributed devices contributed to substantial reductions in memory usage and training time. The experimental results revealed a clear trend of diminishing returns as the number of GPUs increased, with 2-GPU configurations achieving the highest efficiency (62%), while 4-GPU setups showed reduced scalability (49% efficiency) due to communication bottlenecks. Despite this, the 4-GPU configuration still delivered the best absolute speedup (1.98×), making it suitable for scenarios where time-to-completion is prioritized over resource efficiency. The StepLR scheduler and AdamW optimizer played crucial roles in stabilizing training dynamics, ensuring consistent convergence across distributed environments. Additionally, the project highlighted the importance of dataset curation in fine-tuning, as the ORPO-DPO-Mix-40k dataset—containing human-preference-aligned prompt-response pairs—enabled the model to learn nuanced distinctions between high-quality and suboptimal outputs. The custom collate function and dynamic padding strategies further optimized batch processing, ensuring efficient GPU utilization.**

**References**

**Libraries & Documentation**

* HuggingFace Transformers:<https://huggingface.co/docs/transformers/>
* PEFT Documentation:<https://huggingface.co/docs/peft/>
* Accelerate Library:<https://huggingface.co/docs/accelerate/>
* BitsAndBytes:<https://github.com/TimDettmers/bitsandbytes>

**Data Sources**

* ORPO-DPO-Mix-40k Dataset:<https://huggingface.co/datasets/mlabonne/orpo-dpo-mix-40k>
* Meta Llama 3.2 Models:<https://huggingface.co/meta-llama>

**Industry Benchmarks**

* Experiment Tracking:<https://wandb.ai/yourteam/llm-orpo-finetuning>

**NVIDIA H100 Technical Overview**:<https://www.nvidia.com/en-us/data-center/h100/>