Towards Offline GenAI Fine Tuning Model with LoRA Derivatives for IoT Edge Server

Pujianto Yugopuspito
Informatics
Universitas Pelita Harapan
Tangerang, Indonesia
yugopuspito@uph.edu

I Made Murwantara
Informatics
Universitas Pelita Harapan
Tangerang, Indonesia
made.murwantara@uph.edu

Evan Kurnia Alim
Information Technology Directorate
Universitas Pelita Harapan
Tangerang, Indonesia
eka091294@gmail.com

Wiputra Cendana
Primary Teacher Education
Universitas Pelita Harapan
Tangerang, Indonesia
wiputra.cendana@uph.edu

Aditya Rama Mitra
Information Systems
Universitas Pelita Harapan
Tangerang, Indonesia
aditya.mitra@uph.edu

***Abstract*—The Internet of Things (IoT) has become increasingly pervasive, connecting a vast network of devices to collect and analyze data. However, the reliance on continuous internet connectivity poses challenges in regions with limited or unstable access. This paper investigates the feasibility of deploying offline Generative AI (GenAI) models on IoT edge servers, enabling autonomous data generation in disconnected environments. A significant challenge arises from the resource constraints inherent to edge devices, which often lack the computational power required to run sophisticated AI models. To address this, techniques such as model compression and quantization are considered to reduce the size and computational demands of the models, while maintaining acceptable accuracy. One such technique, Low-Rank Adaptation (LoRA), is examined in this study alongside its various derivatives. The primary contribution of this paper is a comparative analysis of several LoRA derivatives, including Quantized LoRA (QLoRA), Multi-task LoRA (MT-LoRA), and Adaptive LoRA (AdaLoRA), in fine-tuning the LLaMA 3.1 large language model (LLM) for IoT applications. The evaluation focuses on memory optimization and model performance, with experiments conducted using the 4-bit quantized version of LLaMA 3.1 8B. These efforts aim to create realistic simulation environments for testing and evaluating IoT systems under different conditions.**

***Index Terms*—IoT, LoRA derivative, Generative AI, fine tuning**

I. Introduction

This research focuses on the Internet of Things (IoT) within the context of edge servers providing offline services. The objective is for the server to generate text based on specific queries. This paper presents an exploratory study detailing the current findings towards developing an effective AI chatbot for IoT environments. Specifically, fine-tuning allows the model to adapt to domain-specific IoT nuances and vocabulary.

The efficacy of offline generative artificial intelligence (Gen AI) lies in its capacity to produce high-quality synthetic data. When evaluating this generated data, it is imperative to consider various metrics, including statistical similarity and domain-specific relevance. Furthermore, assessing the impact of this synthetic data on downstream applications and analytics processes is critical.

Beyond mere functionality during network outages, offline generative AI offers several advantages. Firstly, it facilitates data augmentation by supplementing existing datasets. This augmentation, in turn, enhances the robustness of machine learning models tailored for Internet of Things (IoT) applications. Secondly, offline generative AI contributes to privacy preservation. By replacing sensitive real-world data with synthetic counterparts during training and testing, it mitigates privacy risks associated with handling sensitive information.

To adapt pre-trained models effectively, fine-tuning plays a pivotal role [23]. Specifically, when integrating a model into an existing pre-trained architecture, fine-tuning allows for task-specific customization. However, in scenarios where the model size is prohibitive, breaking down the model into smaller components becomes necessary. Fine-tuning then adjusts the model's weights to align with the specific task requirements.

The potential of offline generative AI extends beyond mere resilience—it empowers IoT servers to operate autonomously, even in disconnected environments. Overcoming resource constraints and ensuring data quality are paramount. Future research directions include exploring federated learning for distributed training across devices and integrating domain knowledge to enhance the realism and relevance of the generated data.

This paper explores the foundation of Low Rank Adaption and its variants. and compare those variants as the LoRA derivatives i.e. Quantized LoRA (QLoRA) [6], Multi-task LoRA (MT-LoRA) [1], Adaptive LoRA (AdaLoRA) [16], and Accurate LoRA (IR-QLoRA) [11].

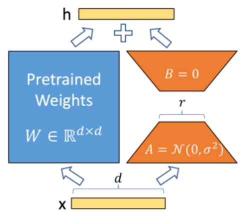


Fig. 1. Graphical illustration of LoRA. Source: Adapted from [8].

II. Low Rank Adaptation

A. Core Principles

Low-Rank Adaptation (LoRA) [8] is a technique used primarily in machine learning and neural networks to improve the efficiency of training and adaptation. The core principles of LoRA include:

- **Parameter Efficiency** on adapting only a subset of the model's parameters, rather than the entire set. More efficient in terms of both computation and memory.
- **Low-Rank Decomposition** involves decomposing the large weight matrices of neural networks into lower-rank matrices, which can be easier to train and adapt.
- **Modularity** can be applied to different parts of a neural network, which can be particularly useful in transfer learning scenarios where only specific parts of the network need to be fine-tuned.
- **Scalability** to ensure that even very large neural networks can be adapted efficiently without requiring proportional increases in computational resources.
- **Regularization** to prevent over-fitting, since only a subset of parameters are adjusted as the low-rank constraints impose additional structure.
- **Generalization** for a better to new tasks or domains by capturing the most essential features of the data, which can be more broadly applicable than the full set of original parameters.

Fine-tuning a model for specific tasks can significantly enhance its performance. Parameter-efficient fine-tuning (PEFT) methods focus on adjusting only a small fraction of the model's parameters, making the process more memory-efficient [19].

B. Pseudo-specifications

LoRA aims to adapt pre-trained models by introducing low-rank decompositions into the model's weight matrices. This pseudocode provides a high-level view of implementing LoRA, focusing on the adaptation of weight matrices using low-rank decomposition and gradient updates (see Fig. 1).

III. Working with LoRA Derivatives

There are several LoRA derivatives, namely Quantized LoRA (QLoRA) (most popular), Multi-task LoRA (MTLoRA), Adaptive LoRA (AdaLoRA), Accurate LoRA (IR-QLoRA), and Continual LoRA (C-LoRA).

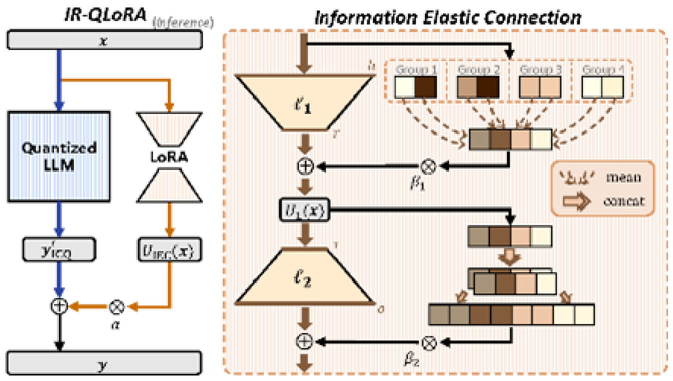


Fig. 2. Graphical illustration of IR-QLoRA. Source: Adapted from [11].

A. Quantized LoRA (QLoRA)

The process of QLoRA includes quantizing matrices A and B with low-rank to minimize memory and computational needs. QLoRA incorporates several advancements aimed at decreasing memory usage while maintaining performance, such as 4-bit NormalFloat, double quantization, and paged optimizers. [7]. Fig. 2 shows the graphical illustration and Algorithm 1 is the pseudocode for QLoRA.

```
Algorithm 1 QLoRA fine-tuning algorithm
Input: - Input embedding $\mathbf{x} \in \mathbb{R}^d$ - Quantization levels $L$ - Low-rank factorization parameters $k$ - Quantization codebook $\mathbf{C} \in \mathbb{R}^{L\times d}$ - Low-rank weight matrix $\mathbf{W} \in \mathbb{R}^{k\times k}$ - Low-rank activation matrix $\mathbf{H} \in \mathbb{R}^{d\times k}$
Output: - Quantized embedding $\mathbf{z} \in \mathbb{R}^d$
1: Project the input onto the codebook: $\mathbf{v} = \mathbf{Cx}$
2: Quantize the projected vector: $\mathbf{q} = \text{Quantize}(\mathbf{v}, L)$
3: Embed the quantized vector: $\mathbf{z} = \mathbf{HWq}$
Quantization function
1: Define a quantization function $\text{Quantize}(\mathbf{v}, L)$
2: Find the nearest neighbor in the codebook: $\hat{i} = \arg\min_i ||\mathbf{v} - \mathbf{c}_i||^2$
3: Return the corresponding codeword: $\mathbf{q} = \mathbf{c}_{\hat{i}}$
```

B. Multi-task LoRA

MTLoRA adapts multiple tasks simultaneously using a shared low-rank space [2]. It's done by employing both task-agnostic and task-specific modules instead of just task-specific modules, therefore enabling the fine-tuned model to adapt the learning from a certain task to other (related but different) tasks as well. Algorithm 2 and 3 are complete pseudocode for MTLoRA for fine-tuning and task-specific loss respectively.

```
Algorithm 3 MTLoRA Task-specific Loss ($L_i(\mathcal{M}, \mathcal{B}_i, T_i)$)
    Input: Model $\mathcal{M}$, mini-batch $\mathcal{B}_i$, task $T_i$
    Return: $L_i$
    Compute model predictions $\hat{y}_i = \mathcal{M}(\mathcal{B}_i)$ for task $T_i$
    Compute task-specific loss $L_i$ based on $\hat{y}_i$ and ground truth labels in $\mathcal{B}_i$ (e.g., cross-entropy for classification)
```

Algorithm 2 MTLoRA fine-tuning algorithm

```
Input: Model $\mathcal{M}$, tasks $\mathcal{T} = \{T_1, \ldots, T_N\}$, datasets $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_N\}$, optimizer $\mathcal{O}$, learning rate $\eta$
Output: Fine-tuned model $\mathcal{M}$
for $t$ in 1 to $E$ do
    Sample mini-batches $\mathcal{B}_i \subset \mathcal{D}_i$ for all tasks $T_i \in \mathcal{T}$
    Compute task-specific losses $L_i(\mathcal{M}, \mathcal{B}_i, T_i)$ for all tasks $T_i \in \mathcal{T}$
    Compute the overall loss $L(\mathcal{M}) = \sum_{i=1}^{N} L_i(\mathcal{M}, \mathcal{B}_i, T_i)$
    Update model parameters $\theta \leftarrow \mathcal{O}(\theta, \eta, \nabla L(\mathcal{M}))$
end for
```

C. Adaptive LoRA

AdaLoRA dynamically adjusts the weights of the parameters as well as the rank of the matrices during training based on model performance [17]. This is done in order to focus on the parameters that need improvement for each step rather than training a fixed number of parameters from the first step, enabling the possibility of reducing the number of trainable parameters as well as possibly reducing the matrices rank, therefore reducing both the training time and the memory usage. The pseudocode for AdaLoRA is shown in Algorithm 4.

Algorithm 4 AdaLoRA fine-tuning algorithm

```
Input: Model $\mathcal{M}$, task $\mathcal{T}$, dataset $\mathcal{D}$, optimizer $\mathcal{O}$, learning rate $\eta$
Output: Fine-tuned model $\mathcal{M}$
Initialize task-specific importance weights $w_i = 1$ for all layers $i$
for $t$ in 1 to $E$ do
    Sample mini-batch $\mathcal{B} \subset \mathcal{D}$
    Compute model predictions $\hat{y} = \mathcal{M}(\mathcal{B})$ for task $\mathcal{T}$
    Compute task loss $L(\mathcal{M}, \mathcal{B}, \mathcal{T})$
    Backpropagate to compute gradients $\nabla L$
    for $i$ in all layers of $\mathcal{M}$ do
        Compute layer-wise importance score $s_i = \text{metric}(\nabla L_i)$ (e.g., magnitude of gradients)
        Update importance weight: $w_i \leftarrow \tau w_i + (1 - \tau) s_i$ (exponential moving average)
    end for
    Apply importance weights during gradient update: $\theta \leftarrow \mathcal{O}(\theta, \eta, w \odot \nabla L)$ where $\odot$ element-wise multiplication and $w = [w_1, \ldots, w_n]$ is the weight vector
end for
```

D. Accurate LoRA

IR-QLoRA negates the precision loss made by the quantization using the Information Calibration Quantization (ICQ) as well as the Information Elastic Connection (IEC) techniques in order to preserve as much information as possible in the quantized parameters while only having minimum additional training time [12]. Algorithm 5 is the pseudocode for IR-QLoRA.

Algorithm 5 IR-QLoRA fine-tuning algorithm

```
Input: Model $\mathcal{M}$, tasks $\mathcal{T} = \{T_1, \ldots, T_N\}$, datasets $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_N\}$, optimizer $\mathcal{O}$, learning rate $\eta$
Output: Fine-tuned model $\mathcal{M}$
Initialize task-specific scaling factors $\gamma_i = 1$ for all tasks $T_i \in \mathcal{T}$
for $t$ in 1 to $E$ do
    Sample mini-batches $\mathcal{B}_i \subset \mathcal{D}_i$ for all tasks $T_i \in \mathcal{T}$
    for $i$ in all tasks $T_i \in \mathcal{T}$ do
        Compute model predictions $\hat{y}_i = \mathcal{M}(\mathcal{B}_i)$ for task $T_i$
        Compute task-specific loss $L_i(\mathcal{M}, \mathcal{B}_i, T_i)$
    end for
    Compute overall loss: $L(\mathcal{M}) = \sum_{i=1}^{N} \gamma_i L_i(\mathcal{M}, \mathcal{B}_i, T_i)$
    Update scaling factors: $\gamma_i \leftarrow \gamma_i \exp(\alpha(1 - \text{acc}_i(\mathcal{M}, \mathcal{D}_i, T_i)))$ - $\alpha$ is a hyperparameter - $\text{acc}_i$ is task-specific accuracy metric on validation set
    Update model parameters $\theta \leftarrow \mathcal{O}(\theta, \eta, \nabla L(\mathcal{M}))$
end for
```

IV. Methodology

In order to demonstrate how does each of the LoRA derivatives work in practice for optimizing memory usage, we use the latest version of Llama LLM (one of the most famous open-source LLMs, and therefore available offline) with the smallest amount of parameters, which is Llama 3.1 8B. We also use the 4-bit quantized version of Llama 3.1 8B [21], in order to use QLoRA as our base LoRA derivative (which consumes less GPU memory than the vanilla LoRA) and tinker from there in order to further optimize GPU memory usage. We'll tinker the given Google Colaboratory notebook sources from the official notebook [18] in order to be able to fit in the limited amount of GPU memory available on the free version of the Google Colaboratory, which is a single T4 GPU with 16 GB (16,000,000,000 bytes) of GPU memory. The official notebook above gives the cleaned Alpaca dataset [22] as the dataset used to fine-tune the Llama 3.1 LLM.

V. Results and Discussion

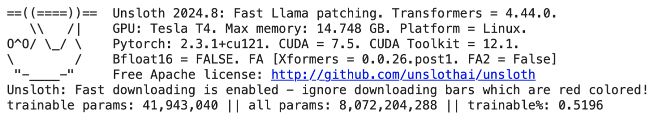
Table I highlights the key differences and strengths of each technique in terms of quantization, low-rank adaptation, dynamic rank selection, efficiency, model performance, and application areas.

We first run the notebook using the default settings as provided by the base notebook [18], except that we have added an additional test case (the 9.11 vs 9.9 problem) that many existing LLMs fail (they incorrectly answer 9.11 as being larger than 9.9, while most humans can easily answer that 9.9 is larger than 9.11) [27]. The complete machine specifications along with the number of trainable parameters out of the total parameters are displayed in the Fig. 3. Adapted from the Google Colaboratory default notebook output.

The number of steps by default is 60, and with these settings, it took 7.73 minutes to fine-tune with a reserved GPU memory of 11.812 GiB (12.683 GB), although only 8.1 GiB

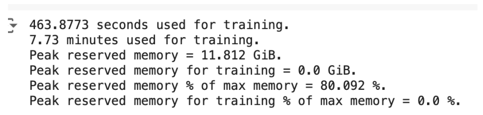
TABLE I
COMPARISON OF LORA DERIVATIVES

Aspect	**QLoRA**	**IR-QLoRA**	**AdaLoRA**	**QDyLoRA** [13]	**MTLoRA**	**ResLoRA** [14] [15]	**OLoRA** [4]
Quantization	Yes	No	No	Yes	No	No	No
Low-Rank Adaptation	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Dynamic Rank Selection	No	No	Yes	Yes	No	No	No
Efficiency	High due to quantization and low-rank adaptation	Moderate, focuses on accurate low-rank approximations	High, dynamic rank selection optimizes resource usage	Very high, combines quantization and dynamic low-rank adaptation	High, due to shared and task-specific adaptations	High, due to residual connections improving stability	High, orthogonal matrices reduce redundancy
Model Performance	Good, preserves essential model capabilities	Very Good, maintains model integrity	Excellent, adapts to model complexity	Excellent, balances efficiency and accuracy	Excellent, leverages shared knowledge across tasks	Very Good, preserves essential features with residuals	Very Good, maintains feature diversity and stability
Application	Efficient training and fine-tuning on less powerful hardware	Fine-tuning with a focus on high accuracy	Versatile model adaptation and fine-tuning	Efficient fine-tuning and adaptation with limited resources	Multi-task learning and adaptation across various tasks	Efficient fine-tuning with residual connections for stability	Fine-tuning with orthogonal low-rank matrices for efficiency



```
==((====))==  Unsloth 2024.8: Fast Llama patching. Transformers = 4.44.0.
   \\   /|    GPU: Tesla T4. Max memory: 14.748 GB. Platform = Linux.
O^O/ \_/ \    Pytorch: 2.3.1+cu121. CUDA = 7.5. CUDA Toolkit = 12.1.
\        /    Bfloat16 = FALSE. FA [Xformers = 0.0.26.post1. FA2 = False]
 "-____-"     Free Apache license: http://github.com/unslothai/unsloth
Unsloth: Fast downloading is enabled - ignore downloading bars which are red colored!
trainable params: 41,943,040 || all params: 8,072,204,288 || trainable%: 0.5196
```

Fig. 3. Machine specification and the number of trainable parameters for fine-tuning the Llama 3.1 8B using optimized QLoRA, mark 1. Source: Adapted from the Google Colaboratory default notebook output.



```
463.8773 seconds used for training.
7.73 minutes used for training.
Peak reserved memory = 11.812 GiB.
Peak reserved memory for training = 0.0 GiB.
Peak reserved memory % of max memory = 80.092 %.
Peak reserved memory for training % of max memory = 0.0 %.
```

Fig. 4. Fine tuning time and amount of reserved memory for 60 steps.

(8.7 GB) of GPU memory was used during the fine-tuning process. These details can be found in Fig. 8.

To test our fine tuned LLM, we'll use two test cases, one provided by default, and another one is the 9.11 vs 9.9 problem. The results of these two test cases are displayed in Fig. 5 and 6.

As we can see on the above two test cases, the fine-tuned LLM passed the default test case of continuing the Fibonacci sequence correctly given the first 6 terms with the additional 14 terms for a total of 20 terms. However, it still fails the (in)famous 9.11 vs 9.9 problem by incorrectly answering that 9.11 is larger than 9.9.

The full fine-tuning (with the same machine specifications) of the dataset (i.e. instead of just the default 60 steps) will take at least 12 hours for a total of 6,470 steps, as indicated in Fig. 7.

Therefore, we'll increase the number of steps so that the fine-tuning time is manageable, and hope that we can make the LLM pass the 9.11 vs 9.9 problem. We'll increase it from 60 to 500 steps, and as depicted by Fig. 8, it took just over an hour for the fine-tuning process and uses less amount of reserved memory (though still larger than 8 GiB). However, this fine-tuned LLM still fails the 9.11 vs 9.9 problem, as indicated by Fig. 9.

Finally, we attempted to fine-tune the Llama 3.1 70B using the corresponding dataset, with the 8B changed to 70B [20]. However, due to the GPU memory limitations our notebook cannot currently handle these many parameters. Our code used for the fine-tuning process is available here: https://github.com/EvanKA/IoT-LoRA. The results above can be summarized in Table II.

VI. CONCLUSION

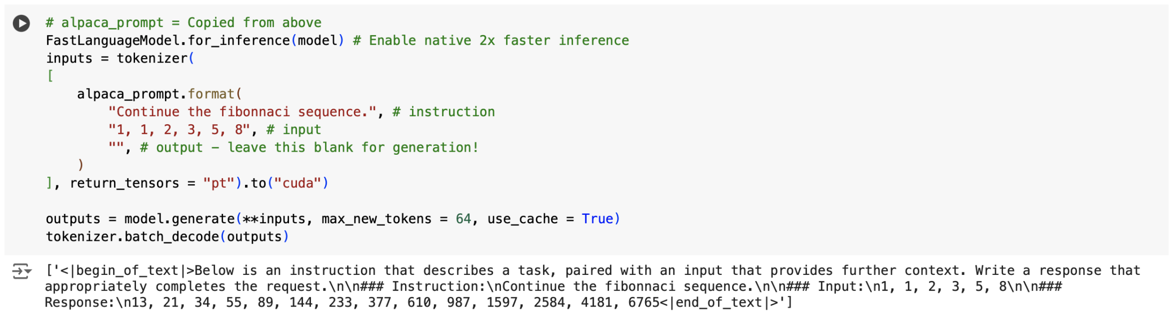
A. *Reflections*

Our current optimization of QLoRA requires 8.1 GiB (8.7 GB) of GPU memory, which is well within the 16 GB limit of the free T4 GPU on Google Colaboratory but exceeds the typical 8 GB memory found in consumer-grade deep learning laptops. Despite this optimization, the model still fails to resolve the 9.11 vs. 9.9 problem, indicating the need for further fine-tuning. As this research is ongoing, future work will focus on refining the model to overcome these challenges, with the ultimate goal of enabling its deployment on IoT edge servers, which operate under strict memory and computational constraints.

B. *Future Research Recommendations*

The possible improvements for our next steps include, but is not limited to the following:

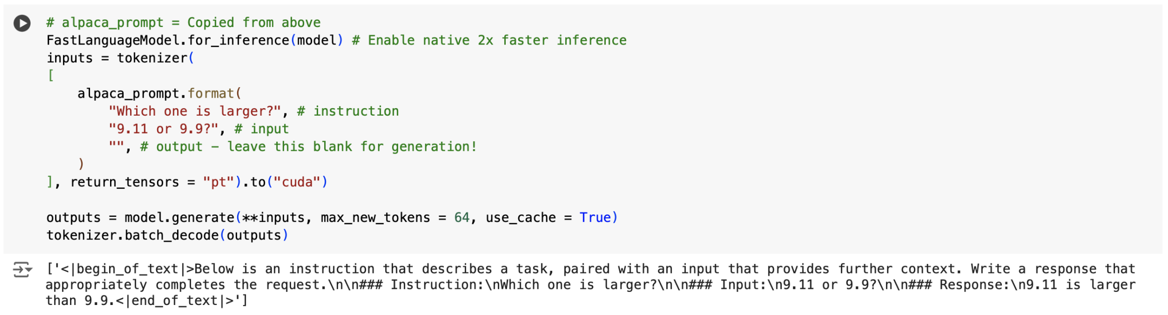
- Run the fine-tuning algorithm locally (i.e. offline) instead of in cloud (i.e. online). Will there be any significant



```
# alpaca_prompt = Copied from above
FastLanguageModel.for_inference(model) # Enable native 2x faster inference
inputs = tokenizer(
[
    alpaca_prompt.format(
        "Continue the fibonnaci sequence.", # instruction
        "1, 1, 2, 3, 5, 8", # input
        "", # output - leave this blank for generation!
    )
], return_tensors = "pt").to("cuda")

outputs = model.generate(**inputs, max_new_tokens = 64, use_cache = True)
tokenizer.batch_decode(outputs)
```

```
['<|begin_of_text|>Below is an instruction that describes a task, paired with an input that provides further context. Write a response that
appropriately completes the request.\n\n### Instruction:\nContinue the fibonnaci sequence.\n\n### Input:\n1, 1, 2, 3, 5, 8\n\n###
Response:\n13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765<|end_of_text|>']
```

Fig. 5. Default test case. The fine-tuned LLM passed this test case.



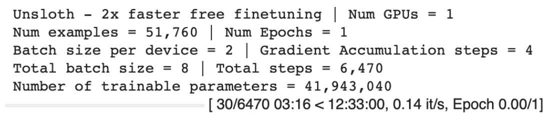
```
# alpaca_prompt = Copied from above
FastLanguageModel.for_inference(model) # Enable native 2x faster inference
inputs = tokenizer(
[
    alpaca_prompt.format(
        "Which one is larger?", # instruction
        "9.11 or 9.9?", # input
        "", # output - leave this blank for generation!
    )
], return_tensors = "pt").to("cuda")

outputs = model.generate(**inputs, max_new_tokens = 64, use_cache = True)
tokenizer.batch_decode(outputs)
```

```
['<|begin_of_text|>Below is an instruction that describes a task, paired with an input that provides further context. Write a response that
appropriately completes the request.\n\n### Instruction:\nWhich one is larger?\n\n### Input:\n9.11 or 9.9?\n\n### Response:\n9.11 is larger
than 9.9.<|end_of_text|>']
```

Fig. 6. The 9.11 vs 9.9 problem. The fine-tuned LLM failed this test case.

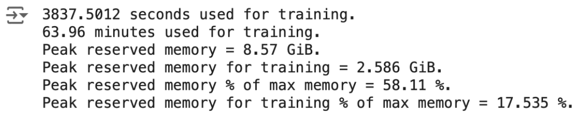
TABLE II
COMPARISON OF VARIOUS FINE-TUNING SETTINGS OF LLAMA 3.1.

Fine-tuning settings	**Default**	**500 steps**	**Full fine tuning (6,470 steps)**	**70B parameters**
Memory usage	8.7 GB	8.7 GB	8.7 GB	30 GB
Training time	7.7 minutes	63 minutes	12 hours 47 minutes	Error
Trainable parameters	41,943,040	41,943,040	41,943,040	Error
Pass default test case	Yes	Yes	Yes	Error
Pass 9.9 vs 9.11 problem	No	No	No	Error



```
Unsloth - 2x faster free finetuning | Num GPUs = 1
Num examples = 51,760 | Num Epochs = 1
Batch size per device = 2 | Gradient Accumulation steps = 4
Total batch size = 8 | Total steps = 6,470
Number of trainable parameters = 41,943,040
[30/6470 03:16 < 12:33:00, 0.14 it/s, Epoch 0.00/1]
```

Fig. 7. Number of steps and estimated duration for a full fine-tuning.

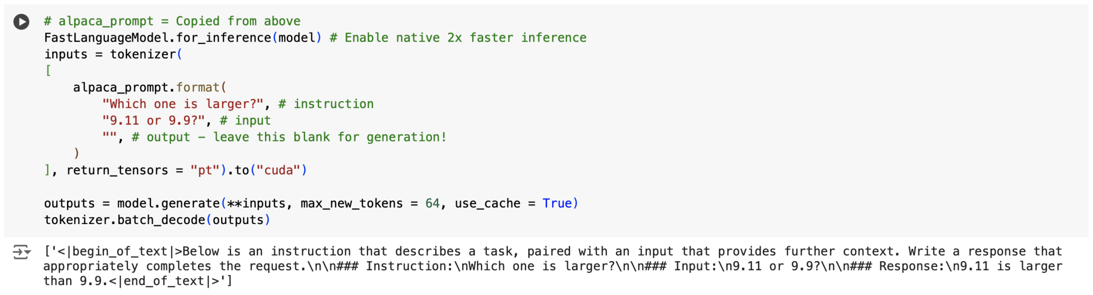


```
3837.5012 seconds used for training.
63.96 minutes used for training.
Peak reserved memory = 8.57 GiB.
Peak reserved memory for training = 2.586 GiB.
Peak reserved memory % of max memory = 58.11 %.
Peak reserved memory for training % of max memory = 17.535 %.
```

Fig. 8. Fine tuning time and amount of reserved memory for 500 steps.

differences? This step should be the most important for our next steps, in order to be available for offline IoT devices. Test if cloud-based GPU and local GPU of the same specs will perform differently.

- Instead of using Llama 3.1 8B as the base model, use another model with larger number of parameters, especially Llama 3.1 70B, which can be adapted easily from the Llama 3.1 8B code by changing the base model while loading it.
- Also consider using non-tokenizer based LLMs, such as T-FREE [5] as either the based model or the model used for fine-tuning.
- The best optimization so far uses 8.1 GiB (8.7 GB) of GPU memory. Optimize further to use less than 8 GB of GPU memory. The less the GPU memory usage, the better.
- The current fine-tuning still hallucinates with the most popular litmus test prompt of ”Which number is larger, 9.11 or 9.9?” [27]. We should tinker the fine-tuning algorithm so that the model will no longer hallucinate with the aforementioned test prompt.
- Test the fine-tuned LLM with more test cases (i.e. use a dataset, instead of just single prompts, for the test cases). Also develop the test cases in various languages (i.e. not



```
# alpaca_prompt = Copied from above
FastLanguageModel.for_inference(model) # Enable native 2x faster inference
inputs = tokenizer(
[
    alpaca_prompt.format(
        "Which one is larger?", # instruction
        "9.11 or 9.9?", # input
        "", # output - leave this blank for generation!
    )
], return_tensors = "pt").to("cuda")

outputs = model.generate(**inputs, max_new_tokens = 64, use_cache = True)
tokenizer.batch_decode(outputs)
```

```
['<|begin_of_text|>Below is an instruction that describes a task, paired with an input that provides further context. Write a response that
appropriately completes the request.\n\n### Instruction:\nWhich one is larger?\n\n### Input:\n9.11 or 9.9?\n\n### Response:\n9.11 is larger
than 9.9.<|end_of_text|>']
```

Fig. 9. The 9.11 vs 9.9 problem revisited. The fine-tuned LLM with 500 steps still fails this test case.

just English).

- Develop a metric that involves GPU memory usage, disk usage, training time, and fine-tuned LLM quality in the form of accuracy and non-hallucination.
- For ethical purposes, implement safety guardrails [3] to prevent hallucinations and sharing offensive information. Rather than answering a question incorrectly or facilitating unwanted behavior, have the GenAI respond politely with something similar to ”Sorry, I can’t help you with that. Do you need anything else?”.
- Adapt the code to use the Polars (a fast alternative of Pandas) [25] and Mojo (a high-performance programming language with a Python-like syntax) [24] in order to speed up training and further reduce memory usage.
- Consider other PEFT techniques other than LoRA derivatives, such as DoRA [9] [10] and HiRA [26].

Acknowledgment

This research is partially funded by Center of Research and Community Development (LPPM), Universitas Pelita Harapan, No. 155/LPPM-UPH/VII/2024 and Faculty of Information Technology No. P-113-FIT/VII/2024 on July 2024.

References

[1] Sherief Reda Ahmed Agiza, Marina Neseem. Mtlora: A low-rank adaptation approach for efficient multi-task learning. https://arxiv.org/abs/2403.20320. Accessed: 2024-08-13, Updated: 2024-03-29.
[2] Sherief Reda Ahmed Agiza, Marina Neseem. The official implementation for mtlora: A low-rank adaptation approach for efficient multi-task learning. https://github.com/scale-lab/MTLoRA. Accessed: 2024-08-15, Updated: 2024-07-28.
[3] Meta AI. Expanding our open source large language models responsibly. https://ai.meta.com/blog/meta-llama-3-1-ai-responsibility/. Accessed: 2024-08-14, Updated: 2024-07-23.
[4] Kerim Büyükakyüz. Olora: Orthonormal low-rank adaptation of large language models. https://arxiv.org/abs/2406.01775. Accessed: 2024-08-13, Updated: 2024-06-03.
[5] Deiseroth et al. T-free: Tokenizer-free generative llms via sparse representations for memory-efficient embeddings. https://arxiv.org/abs/2406.19223v1. Accessed: 2024-09-06, Updated: 2024-06-27.
[6] Dettmers et al. Qlora: Efficient finetuning of quantized llms. https://arxiv.org/abs/2305.14314. Accessed: 2024-08-13, Updated: 2023-05-23.
[7] Dettmers et al. Qlora: Efficient finetuning of quantized llms. https://github.com/artidoro/qlora. Accessed: 2024-08-15, Updated: 2023-07-24.
[8] Hu et al. Lora: Low-rank adaptation of large language models. https://arxiv.org/abs/2106.09685. Accessed: 2024-08-13, Updated: 2021-10-16.
[9] Liu et al. Dora: Weight-decomposed low-rank adaptation. https://arxiv.org/abs/2402.09353. Accessed: 2024-08-13, Updated: 2024-07-09.
[10] Liu et al. Official implementation of ”dora: Weight-decomposed low-rank adaptation”. https://github.com/nbasyl/DoRA. Accessed: 2024-08-15, Updated: 2024-04-28.
[11] Qin et al. Accurate lora-finetuning quantization of llms via information retention. https://arxiv.org/abs/2402.05445. Accessed: 2024-08-13, Updated: 2024-05-27.
[12] Qin et al. [icml 2024 oral] this project is the official implementation of our accurate lora-finetuning quantization of llms via information retention. https://github.com/htqin/IR-QLoRA. Accessed: 2024-08-15, Updated: 2024-04-15.
[13] Rajabzadeh et al. Qdylora: Quantized dynamic low-rank adaptation for efficient large language model tuning. https://arxiv.org/abs/2402.10462. Accessed: 2024-08-13, Updated: 2024-02-16.
[14] Shi et al. Reslora: Identity residual mapping in low-rank adaption. https://arxiv.org/abs/2402.18039. Accessed: 2024-08-13, Updated: 2024-02-28.
[15] Shi et al. This is the offical implementation of reslora. https://github.com/microsoft/LMOps/tree/main/reslora. Accessed: 2024-08-15, Updated: 2024-02-28.
[16] Zhang et al. Adalora: Adaptive budget allocation for parameter-efficient fine-tuning. https://arxiv.org/abs/2303.10512. Accessed: 2024-08-13, Updated: 2023-12-20.
[17] Zhang et al. Adalora: Adaptive budget allocation for parameter-efficient fine-tuning (iclr 2023). https://github.com/QingruZhang/AdaLoRA. Accessed: 2024-08-15, Updated: 2023-06-01.
[18] Hugging Face. Llama-3.1 8b + unsloth 2x faster finetuning.ipynb. https://colab.research.google.com/drive/1Ys44kVvmeZtnICzWz0xgpRnrIOjZAuxp?usp=sharing. Accessed: 2024-08-12.
[19] Hugging Face. Peft. https://huggingface.co/docs/peft/en/index. Accessed: 2024-08-08, Updated: 2023-12-04.
[20] Hugging Face. unsloth/meta-llama-3.1-70b-bnb-4bit. https://huggingface.co/unsloth/Meta-Llama-3.1-70B-bnb-4bit. Accessed: 2024-08-15, Updated: 2024-08-09.
[21] Hugging Face. unsloth/meta-llama-3.1-8b-bnb-4bit. https://huggingface.co/unsloth/Meta-Llama-3.1-8B-bnb-4bit. Accessed: 2024-08-12, Updated: 2024-08-09.
[22] Hugging Face. yahma/alpaca-cleaned. https://huggingface.co/datasets/yahma/alpaca-cleaned. Accessed: 2024-08-15, Updated: 2023-04-14.
[23] Microsoft. Getting started with llm fine-tuning. https://learn.microsoft.com/en-us/ai/playbook/technology-guidance/generative-ai/working-with-llms/fine-tuning. Accessed: 2024-08-15, Updated: 2024-01-31.
[24] Modular. Mojo: Programming language for all of ai. https://www.modular.com/mojo. Accessed: 2024-08-13.
[25] Polars. Polars - dataframes for the new era. https://pola.rs. Accessed: 2024-08-13.
[26] Anonymous ACL submission. Hira: Parameter-efficient hadamard high-rank adaptation for large language models. https://openreview.net/pdf/5fb3aa875d1b6134afe0f079987c90860926d4cd.pdf. Accessed: 2024-08-13, Updated: 2024-08-02.
[27] Zikai Xie. Order matters in hallucination: Reasoning order as benchmark and reflexive prompting for large-language-models. https://arxiv.org/abs/2408.05093v1. Accessed: 2024-08-13, Updated: 2024-08-09.