WhichLoRA
CS 8803: Systems for AI - LLMs

Shiva Ramaswami
Georgia Institute of Technology

Amitrajit Bhattacharjee
Georgia Institute of Technology

Mithilesh Vaidya
Georgia Institute of Technology

1 Introduction

Large Language Models (LLMs) are very popular due to their strong zero-shot performance on a variety of tasks. Their size is key to their performance: models with billions of parameters are trained on billions (or trillions) of tokens of text to obtain these impressive results.

However, we sometimes want to further fine-tune them on a very specific task or a dataset in order to maximise performance. The most natural approach is to load a pre-trained model and further update the weights by fine-tuning the model with a dataset for the task. This can be very expensive (both in terms of time and memory) since we need to update billions of parameters.

Low-Rank Adaptation (LoRA) [9] has emerged as a popular fine-tuning method for LLMs. Instead of fine-tuning billions of parameters, LoRA instead learns a low-rank matrix which is added to the pre-trained weights. Because these low-rank matrices are small compared to the size of the entire model, LoRA adapters can be used to switch between multiple fine-tuned versions of a base model efficiently. In addition, because low-rank matrices are added to the initial weights, there is no addition of latency when performing inference on the fine-tuned model. Low-rank reduces the number of trainable parameters during fine-tuning by multiple orders of magnitude and shows negligible degradation in performance compared to fully fine-tuned models!

LLM inference remains an important and non-trivial problem for optimally serving industrial-scale applications in production. Developers have to navigate through an increasingly large space of model variants - versions of models trained under varying circumstances, all differing in resource footprints, latency, costs, and accuracy. In order to address this, Romero et al. [12] propose INFaaS, an automated model-less system for distributed inference serving. Developers provide their task requirements via an API, while the system uses these to requirements to choose appropriate variants and deploy them on suitable hardware. This saves a lot of time and effort, making it especially amenable to developers who are not in the LLM space.

Combining the insights from INFaaS and LoRA, we propose a framework for automatic generation of LoRA-based fine-tuning and deployment strategies. Developer are often constrained in compute and time resources, and the goal of WhichLoRA is to aid fine-tuning and deployment of LoRA models in an optimal manner within these constraints. We specifically target low resource, single-GPU usecases.

2 Motivation and Background

2.1 Motivation

Fine-tuning using LoRA has become a favored method for developers that have constrained resources or that want to switch between multiple fine-tuned models. The reduced number of trainable parameters when using LoRA can decrease typical training time and peak memory usage when compared to full fine-tuning, and the lightweight nature of the low-rank matrices makes it easy to switch between models. However, deciding hyperparameters for LoRA fine-tuning based on a use case can be a complex task. Depending on the constraints of a task, there are ways to change LoRA's configuration to improve performance in many areas. For example, QLoRA [8] can reduce the peak memory usage at the cost of more time. Parallelism techniques like Fully Sharded Data Parallel [15] can be used to reduce hardware and GPU requirements. Other parameters such as the rank of the matrices that we apply to each layer can pose small trade-offs in training time and accuracy. It can be difficult for a developer to decide which of these frameworks and parameters to use when performing LoRA fine-tuning in order to maximize performance. In addition, maximum accuracy is typically achieved by using the largest pre-trained base model for fine-tuning, so selecting the best base model along with the best LoRA modifications and parameters is a non-trivial decision.

2.2 Flexibility

In the previous subsection, we talked about the challenges with fine-tuning an LLM with LoRA. It involves numerous considerations. However, due to the fast moving nature of the field, any framework which relies on a fixed number of considerations to optimize for will quickly become outdated. For example, although LoRA and QLoRA are the dominant PEFT methods, new methods such as QDoRA [11] have been recently proposed (only a month ago), which could overtake both in adoption. In order to make our framework future-proof, we aim to build a robust API which can easily incorporate such improvements with minimal changes to the codebase. In other words, we aim to abstract away the tedious task of profiling and decouple it from the implementation of the actual LLM and fine-tuning process. In the subsequent sections, we discuss how it is achieved using our API.

3 Design and Implementation

To recap, the goal of our project is to accept some constraints from the user in terms of inference and fine-tuning time and memory, dataset to fine-tune on and list of HuggingFace models[1]. We aim to output the largest possible base model which can be fine-tuned using LoRA (or its variants) and satisfies all the time and memory constraints.

More specifically, we accept the following inputs from the user:

- Inference time per token
- Inference peak memory
- Total Fine-tuning time
- Fine-tuning peak memory
- Dataset to fine-tune on
- Optionally, we also accept a list of base models to be fine-tuned. If not provided, we use some popular open-source LLMs such as Gemma and LLaMa

Our framework outputs the largest base model whose memory requirements are lesser than the peak memory given as input and which can be fine-tuned within the given time constraints. We also output the LoRA variant and the optimal batch size[2]

We break our problem into two independent stages, as shown in Figure 2. All inputs from the developer are accepted through a *config.yaml* file. Different HuggingFace datasets

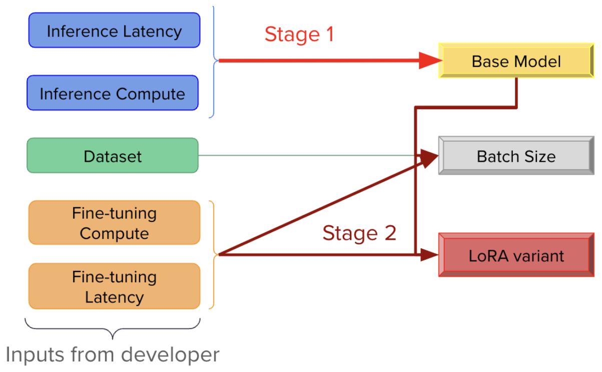
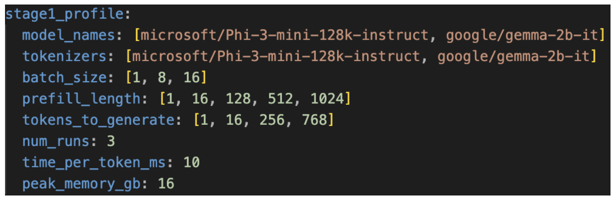


Figure 2: Two stage approach of our framework. Since LoRA does not affect inference performance (both in terms of time and memory), we can first pick the base model purely based on inference constraints. Then, in stage 2, given a base model, we pick the best LoRA configuration which satisfies the fine-tuning constraints.



```
stage1_profile:
  model_names: [microsoft/Phi-3-mini-128k-instruct, google/gemma-2b-it]
  tokenizers: [microsoft/Phi-3-mini-128k-instruct, google/gemma-2b-it]
  batch_size: [1, 8, 16]
  prefill_length: [1, 16, 128, 512, 1024]
  tokens_to_generate: [1, 16, 256, 768]
  num_runs: 3
  time_per_token_ms: 10
  peak_memory_gb: 16
```

Figure 3: Figure depicts a screenshot of the relevant section of the config file for stage 1. It specifies the models to be profiled, along with inference configurations, number of runs to average, maximum allowed time per token and peak memory during inference.

have different pre-processing steps to convert raw data into a tokenized train-validation split. If the user wants to use their own dataset, they need to implement a new class for such a dataset. The framework has a *BaseDataset* class, which needs to be subclassed and it must have two functions. The first function should simply return the train-validation subsets. The second function should convert each row in the dataset into a string, which can be subsequently tokenized.[3]

3.1 Stage 1

In Stage 1, we choose all possible base models which satisfy the inference constraints (both time and memory). The crucial observation is that this choice is **independent** of the LoRA variants and fine-tuning procedure. This is because LoRA weights are merged with the model weights after fine-tuning and hence have no time and memory overhead. This allows us

[1]Due to its popularity, we stick to HuggingFace. Ideally, one can extend the framework to use their own LLM model.

[2]Higher batch sizes increase GPU utilization at the cost of extra memory. Hence, we aim to find the largest possible batch size. Note that the convergence of fine-tuning may be affected by batch size but that consideration is beyond the scope of our work.

[3]e.g. the NVIDIA HelpSteer dataset has a dictionary with keys *query* and *response* per input sample x. The function should return *x['query'] + ['Answer'] + x['response']* and the corresponding attention mask for masked language modelling.

to filter out base models purely based on the inference requirements. Note that returning the best model (best, in most cases, being the largest, considering the scaling hypothesis) is sub-optimal since the best model need not satisfy the fine-tuning constraints. Hence, it is mandatory to return **all** possible base models which satisfy the inference constraints.

Our implementation for Stage 1 involves profiling the forward pass of the model for various batch sizes, prefill lengths and number of tokens to decode (one tuple of the three parameters is hereby referred to as *inference configuration*). These can be either accepted as input from the user or can be run with some default values (such as batch sizes = [1, 2, 4], prefill length = [128, 256, 512] tokens and tokens to decode = [128, 256, 512]). A snippet of the config file for this stage is given in Figure 3.

The user simply submits a predefined PACE script called *script_main.sbatch* (without any changes) to the PACE cluster. This script generates multiple PACE scripts, one for each NVIDIA GPU type found on the cluster. The developer can then submit the desired script, depending on the GPU to profile. Along with the exact command to run Stage 1 and Stage 2, the PACE report from the *script_main.sbatch* job also contains the inference peak memory predictions for each inference configuration, as discussed in Section 3.4. The user can eyeball the numbers to get a rough idea of the required memory.

Once a job is submitted by the developer for a specific hardware, Stage 1 runs profiling, followed automatically by Stage 2. The resulting output is then parsed by our code, which returns plots of inference time and memory for various configurations. The user can visually inspect them to get an intuitive sense for the time and memory characteristic of the models. We also plot the time vs memory on a single plot to get a sense of the time-memory trade-off for various configurations. Examples of such plots are given in Figure 4 and Figure 5. Finally, the script also outputs a list of base models which follow the inference time and memory constraints and are hence potential candidates for fine-tuning using LoRA.

3.2 Stage 2

Similar to Stage 1, our goal with Stage 2 is to take in certain restrictions, and provide the feasible, most optimal solution under those constraints. For Stage 2, our input consists of:

- Total Fine-tuning time
- Fine-tuning peak memory utilisation (as determined by the user's hardware restrictions)
- Dataloader: If it is a dataset previously seen by the system, a dataloader will already be present - if not, the user will need to write custom loader for their dataset.
- The output of Stage 1 i.e. the list of feasible base models which satisfy the inference constraints

Given these restrictions, we perform some profiling in order to the find the ideal fine-tuning strategy i.e. the fine-tuning configuration that operates within the user constraints while also maximising the performance. Note that we do not make considerations for any of the inference constraints, since the assumption is that the Stage 1 process has already accounted for that.

For profiling, we perform fine-tuning using all feasible base models on a small subset of the dataset in order to get an estimate on the peak memory as well as the time taken to process per token. As shown in Figure 6, the fine-tuning time scales linearly with dataset size, and thus the time taken per token can be used to estimate the total fine-tuning time.

Similar to the *inference configuration* defined in Stage 1, we profile for a variety of *fine-tuning configurations*. Each fine-tuning configuration is a tuple consisting of the base model, the batch size, and the LoRA variant. In the scope of our work, we define the LoRA variant to be either LoRA or QLoRA. Although there are other hyperparameters within LoRA (like the r value and alpha value) which can be varied, we decide to fix them to the certain values since the optimal choice for those hyperparameters is still an active research problem which lies outside the scope of this work. As in Stage 1, the batch sizes and LoRA variants can be set to certain default values or can be taken in as input from the user. For example, Figure 7 displays the relevant snippet from the config file where the user provides the batch sizes and the fine-tuning time and memory constraints in order to shortlist the feasible models.

The output of Stage 2 consists of:

- Plots of the fine-tuning time per token for varying batch sizes and LoRA variants across all base models (Figure 8)
- Plots of the peak memory usage for varying batch sizes and LoRA variants across all base models (Figure 8). These plots can be eyeballed by the user to get an intuitive understanding of the memory usage and time taken across models.
- Plots of the memory vs time per token across all base models. These plots showcase the time vs memory trade-offs that come into play on varying both the batch sizes and the LoRA variant - this is further detailed in section 3.3
- Largest base model which satisfies all constraints, along with optimal LoRA variant and batch size - this is the optimal configuration
- A generated PACE script, which the user can use to run the fine-tuning with the optimal settings.

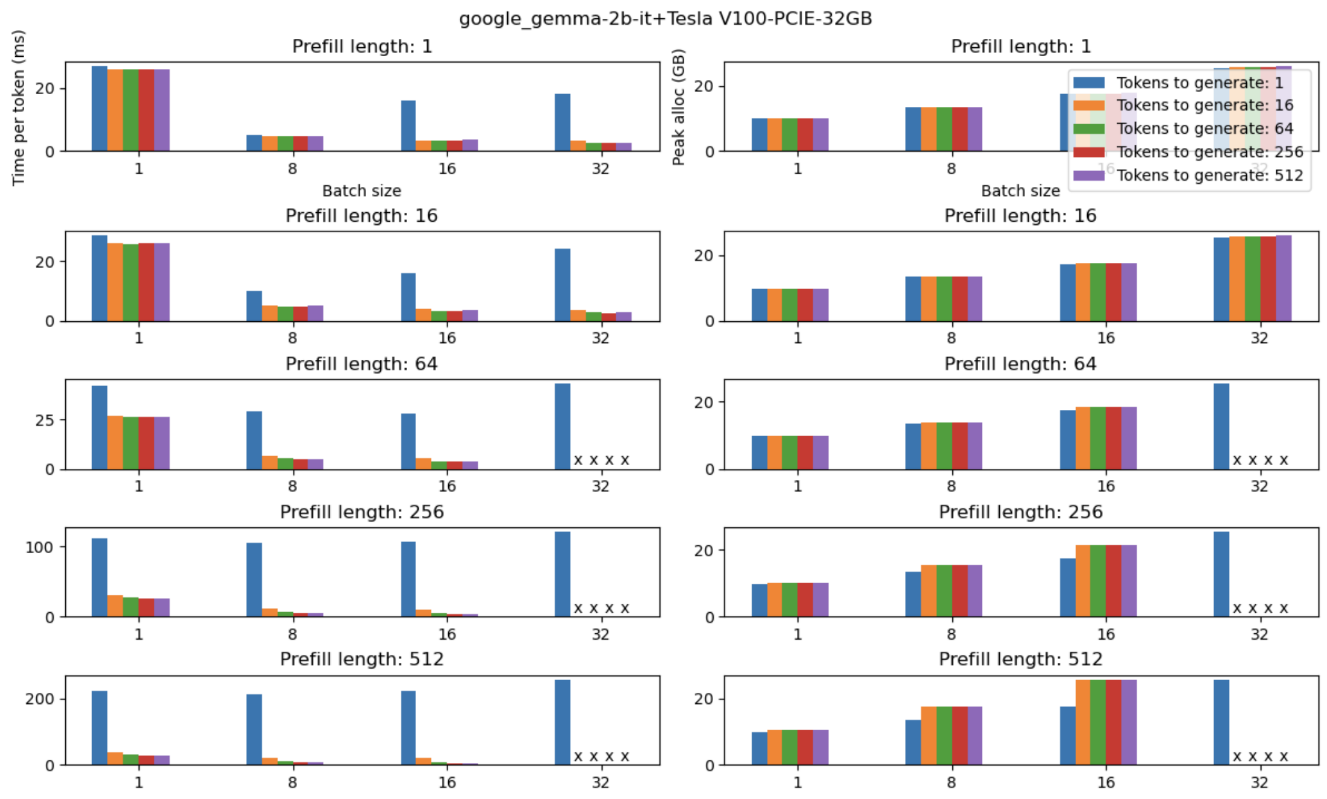


Figure 4: Time per token and peak memory consumption during one forward pass, averaged over 3 runs. A number of interesting observations can be made. Firstly, time shows a non-linear characteristic across various dimensions: with more tokens to decode, the pre-fill cost gets amortized across various autoregressive decode calls. Higher batch sizes reduces time per token since batching increases hardware utilization but only upto a certain limit. Absolute time per token scales linearly with prefill length after PL > 64 since mostly becomes compute-bound thereafter. Secondly, peak memory increases monotonically with pre-fill length, batch size and is less sensitive to the number of tokens to decode since the KV cache does not grow significantly for these configurations. Crosses indicates configurations which led to OOM.

3.3 Time vs Memory trade-offs during Fine-tuning

The key trade-off at the core of the fine-tuning process is the time vs memory trade-off. Both the hyperparameters we vary have a direct effect on these factors:

- If QLoRA is used in place of LoRA, the peak memory usage comes down by 75%, but the tuning time increases by 66% [1]
- If the batch size is decreased, the peak memory usage comes down, but the overall tuning time increases since more training steps must be taken to iterate through the complete dataset

Thus, if the user is extremely constrained in terms of available memory, the ideal configuration is to fine-tune QLoRA with a batch size of 1. In order to visualise these trade-offs, the peak memory usage vs the time per token is plotted for each base model as shown in Figure 9.

We observe that the LoRA variant has the bigger effect on the trade-off as compared to the batch size - fine-tuning with QLoRA will usually require lower memory than with LoRA, irrespective of the batch size. Having said that, if a LoRA model is tuned with a lower batch size and a QLoRA model is tuned with a higher batch size, their memory requirements are quite similar; in fact if a very high batch size is used for QLoRA its memory usage may exceeed the LoRA model's usage. Note also that these trade-offs are highly dependent upon the hardware and the base models as well. For example, in the case of Mixture of Experts (MoE) models - since the number of parameters being tuned is quite high - the advantage of quantisation in QLoRA is so large that the LoRA model almost always has a much higher memory requirement.

3.4 Memory Profiling for inference

As can be seen from Figure 4, time per token is very difficult to predict due to differences in the pre-fill and decode phase

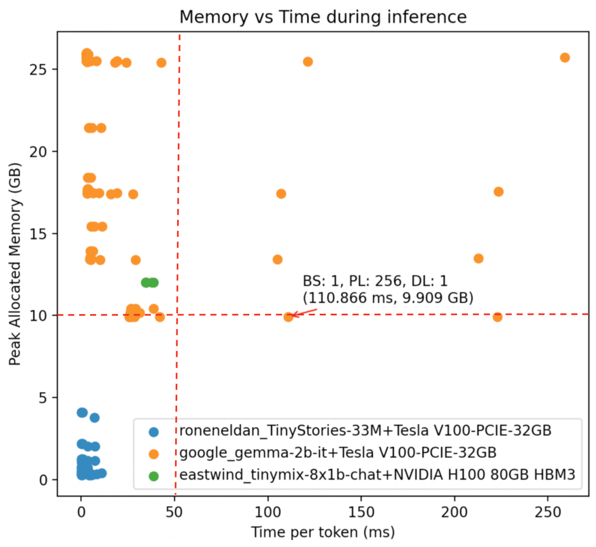


Figure 5: Time-memory trade-off for multiple models. Each dot represents an inference configurations. Each colour represents a particular model-hardware configuration. The plot is interactive, allowing the user to hover over various dots and display the configuration. In the figure, BS, PL and DL refer to the batch size, pre-fill length and number of tokens to decode respectively. The dashed lines represent the max allowed time per token (50ms) and max peak memory (10 GB) specified by the developer.

(leading to varying hardware utilization). Memory, on the other hand, monotonically increases with batch size, pre-fill length and tokens to decode.

Originally, we planned to turn this insight into an additional feature of our framework: **predicting** the memory usage of a model during inference. This can help us avoid profiling runs in cases where the predicted memory usage is higher than the given peak memory constraint. Ideally, such predictions can be carried out by simply keeping track of the sizes of the model weights, activations and the KV cache. However, a simple implementation of such a heuristic deviated from the actual values in certain cases. This can be seen in Figure 10. We debugged the issue and potentially found an explanation: PagedAttention allocates chunks of memory for each input example[4]. Hence, for higher batch sizes, more memory gets allocated than required, especially when number of tokens to generate is low. As a result, our prediction underestimates actual peak memory usage. Refer to Figure 11 for a detailed analysis.

We realised that due to the fast-moving nature of the field, such predictors will quickly get outdated with improvements in quantization techniques, architecture, MoE considerations[5],

[4]Unfortunately, the HuggingFace *generate()* function does not allow us to control the size of the chunk or turn off PagedAttention.

[5]Only a subset of the parameters in a Mixture-of-Experts model are active, thereby complicating the memory consumed during forward pass.

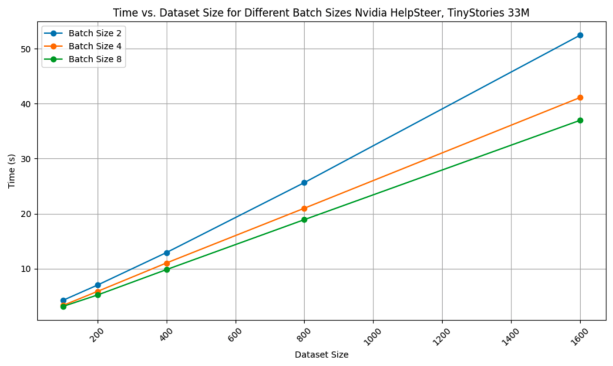
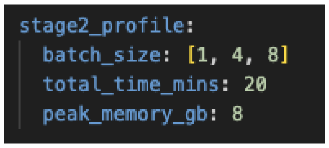


Figure 6: Total fine-tuning time v/s dataset size across multiple batches. As can be observed, the time scales linearly with the size of the dataset.



```
stage2_profile:
  batch_size: [1, 4, 8]
  total_time_mins: 20
  peak_memory_gb: 8
```

Figure 7: Screenshot of the relevant section of the config file for stage 2. It specifies the batch sizes to be tested, as well as the developer constraints in terms of the total fine-tuning time in minutes and peak memory in GB.

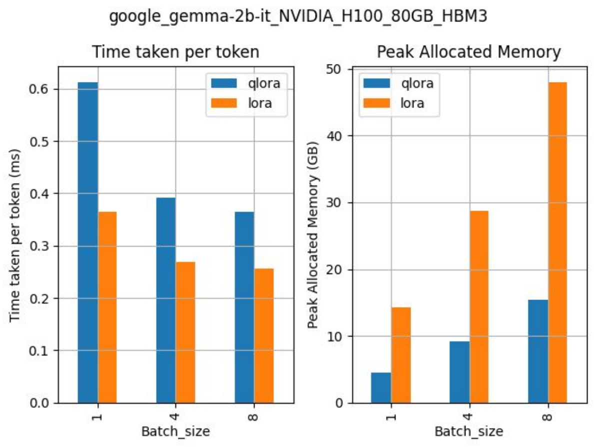


Figure 8: Fine-tuning time taken per token and the peak memory usage plotted for varying batch sizes and LoRA variants for the Gemma 2B model [13]

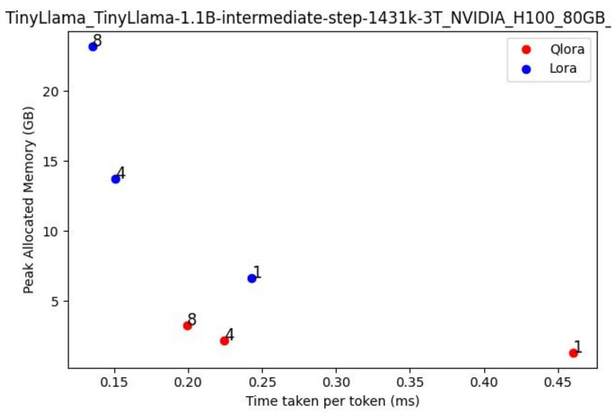
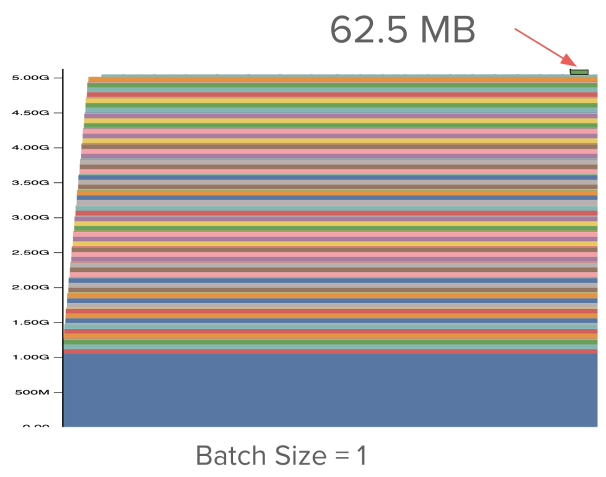
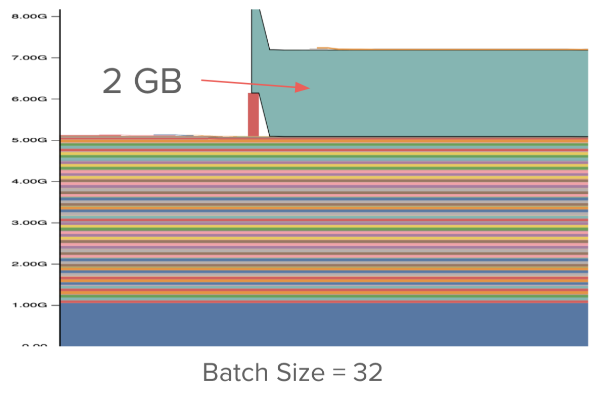


Figure 9: Memory vs Time plot for the TinyLLama-1.1B model [2] at varying batch sizes and LoRA variants. As can be observed, the LoRA variant has the strongest effect on the memory vs time trade-off, but using a low batch size with LoRA might yield similar results to using a high batch size with QLoRA.



(a) (a) When batch size = 1, a 62.5 MB chunk is allocated by PyTorch, as seen from the CUDA profiler snapshot.



(b) (b) When batch size = 32, a 2 GB chunk is allocated by PyTorch, as seen from the CUDA profiler snapshot.

Figure 11: Plot shows PyTorch memory allocation during forward pass of Gemma-2b with prefill length of 32 and tokens to decode set to 1. We can see that a 62.5 MB chunk is allocated when batch size is 1 while a 2 GB chunk is allocated for batch size 32. Since 62.5*32 ≈ 2 GB, we believe that the PagedAttention implementation during inference leads to the observed discrepancy.

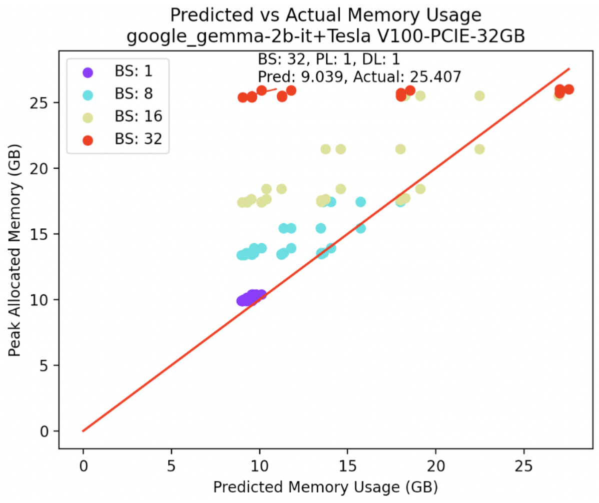


Figure 10: Predicted vs Actual peak memory usage of Gemma 2b for various inference configurations. We can see that the deviation increases with an increase in batch size.

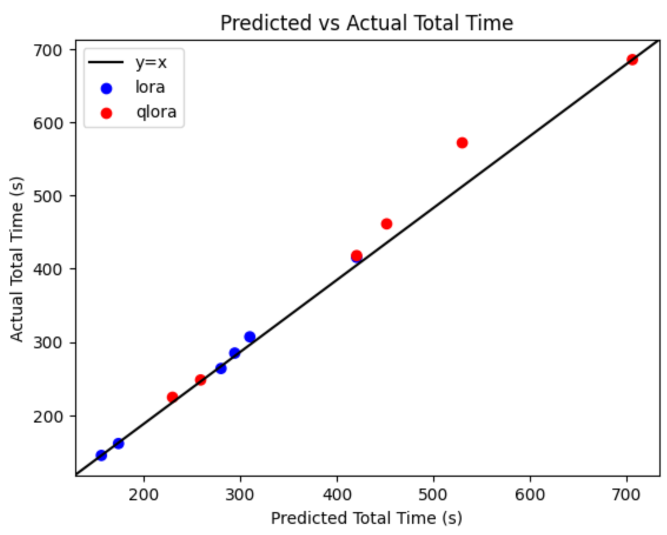


Figure 12: This graph shows very high correlation between the time predicted by Stage 2 and the actual time for fine-tuning.

etc. As a result, we decided to only provide the user an estimate of the memory usage but provide no guarantees and hence profiling all possible configurations. This is a limitation of our work.

3.5 Accuracy Guarantees

It is tempting to consider if such a framework can provide accuracy guarantees (or any other metric suitable for the task at hand; for this discussion, we will refer to it as accuracy but the key idea holds). Before fine-tuning on a particular dataset, can we estimate the gain in accuracy that can be expected? If yes, it could significantly improve the developer experience: instead of specifying fine-tuning time and compute, the developer could simply specify the desired accuracy and the framework could output expected gains for each dataset, thereby making the choice of dataset without any developer intervention). However, some recent literature [14] suggests it might not be possible to do so. The paper shows that for a given model, the fine-tuning performance is significantly affected by size of the pre-training dataset: more the pre-training data, the lesser samples are required during fine-tuning to achieve the same performance as low pre-training data + more fine-tuning. This is just one dimension along which unpredictability can creep into the accuracy guarantees. As a result, we cannot give any accuracy guarantees without actually fine-tuning the given base model.

4 Evaluation

Since no prior work examines the automatic generation of LoRA variants based on user constraints, we do not have a baseline to compare with.

Instead, we compare the actual total time taken by the model to fine-tune using LoRA/QLoRA for varying batch sizes and compare it with the predicted time from our stage 2 profiler. Results are shown in Figure 12.

As expected, since time scales linearly with the dataset size, our predictions are very close to the actual time taken. In summary, by profiling for only a few batches (hyperparameter which can be varied; currently set to 5), we can get an accurate estimate of the fine-tuning time.

5 Related Work

To the best of our knowledge, there is no prior framework which carries out detailed profiling of LLMs. Some blog posts such as [3,4] describe ways to compute the inference time and memory requirements for various hardware but theoretical numbers are hard to match in real-world setups[6].

For easy experimentation with LoRA fine-tuning, there exist some web-based solutions such as [5, 6]. However, they assume that the optimal batch size is known beforehand. In other words, there is no estimate of the time and memory requirements during inference and fine-tuning.

Orthogonal work such as quantization [7, 10] can be integrated into our framework by including them in the *HF-Model.py* file.

6 Conclusion

In this work, we implemented a framework for generating LoRA variants automatically, based on high-level developer inputs which include the list of base models to profile, along with the inference and fine-tuning compute and memory. Batch size, prefill length and number of tokens to generate significantly affect the time-memory characteristics during inference. In Stage 1, we profile all the given models on various inference configurations (or only on the ones specified by the developer) and output all models which satisfy the inference constraints. Next, in Stage 2, we profile various LoRA variants and batch sizes for each base model. We use this as an estimate for the total fine-tuning time and output the largest base model[7] which satisfies the fine-tuning constraints (both memory and time). The two-stage process helps in interpretability. By implementing an end-to-end pipeline, our

[6]As the old adage goes, *In theory, practice and theory are same. In practice, they are different.*

[7]We output a table with all the results but generate the final fine-tuning script for the largest model since larger pre-trained models tend to perform better than smaller pre-trained models due to the scaling hypothesis. However, this may not always hold, as discussed in Section 3.5.

framework outputs a PACE script which can be directly submitted to the PACE cluster for fine-tuning using the optimal configuration.

The field is evolving rapidly. Hence, it is important to make the framework future-proof. Our API accepts base models in the form of HuggingFace strings, thereby allowing newer models (such as MoEs) without any changes in the code. Although we tested LoRA and QLoRA, it is easy to extend our framework to newer works by simply swapping out the PEFT config. By generating PACE scripts for all CUDA devices on PACE, our framework is hardware-agnostic and can be potentially extended to other devices.

Some future work includes accurate memory prediction during both inference and fine-tuning.

Acknowledgments

We would like to thank Prof. Iyer, the TAs and our classmates for the insightful discussions in class.

Individual Contributions

- Amitrajit: Stage 2 profiling, Pipeline, Report, Midterm and Endterm Presentation
- Shiva: Stage 2 experiments on datasets, Pipeline, Report, Midterm and Endterm Presentation
- Mithilesh: Stage 1, Pipeline, Report, Midterm and Endterm Presentation

References

[1] https://cloud.google.com/vertex-ai/generative-ai/docs/models/tune-models#lora-qlora-recommendations-for-llms.

[2] https://huggingface.co/TinyLlama/TinyLlama-1.1B-intermediate-step-1431k-3T.

[3] https://symbl.ai/developers/blog/a-guide-to-llm-inference-performance-monitoring/.

[4] https://www.baseten.co/blog/llm-transformer-inference-guide/.

[5] https://github.com/AUTOMATIC1111/stable-diffusion-webui.

[6] https://github.com/hiyouga/LLaMA-Factory.

[7] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3.int8(): 8-bit matrix multiplication for transformers at scale. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 30318–30332. Curran Associates, Inc., 2022.

[8] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.

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

[10] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration, 2024.

[11] Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. Dora: Weight-decomposed low-rank adaptation, 2024.

[12] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021.

[13] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Pier Giuseppe Sessa, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Amélie Héliou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Clément Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Christian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, Justin Mao-Jones, Katherine Lee, Kathy Yu, Katie Millican, Lars Lowe Sjoesund, Lisa Lee, Lucas Dixon, Machel Reid, Maciej Mikuła, Mateo Wirth, Michael Sharman, Nikolai Chinaev, Nithum Thain, Olivier Bachem, Oscar Chang, Oscar Wahltinez, Paige Bailey, Paul Michel, Petko Yotov, Rahma Chaabouni, Ramona Comanescu, Reena Jana, Rohan Anil, Ross McIlroy, Ruibo Liu, Ryan Mullins, Samuel L Smith, Sebastian Borgeaud, Sertan Girgin, Sholto Douglas, Shree Pandya, Siamak Shakeri, Soham De, Ted Klimenko, Tom Hennigan, Vlad Feinberg, Wojciech Stokowiec, Yu hui Chen, Zafarali

Ahmed, Zhitao Gong, Tris Warkentin, Ludovic Peran, Minh Giang, Clément Farabet, Oriol Vinyals, Jeff Dean, Koray Kavukcuoglu, Demis Hassabis, Zoubin Ghahramani, Douglas Eck, Joelle Barral, Fernando Pereira, Eli Collins, Armand Joulin, Noah Fiedel, Evan Senter, Alek Andreev, and Kathleen Kenealy. Gemma: Open models based on gemini research and technology, 2024.

[14] Biao Zhang, Zhongtao Liu, Colin Cherry, and Orhan Firat. When scaling meets llm finetuning: The effect of data, model and finetuning method, 2024.

[15] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel, 2023.