Making LLMs Work for Enterprise Part 3: GPT Fine-Tuning for RAG

Planning / Implementation

Fine-tuning a language model can tailor it to specific tasks. In this paper, we explore the process of fine-tuning a large language model (LLM) using a custom dataset for retrieval augmented generation (RAG). Specifically, we focus on base model sizing, from efficient inference to hardware requirements, and software options for parameter-efficient fine-tuning.

The deliverables created by following the steps in this paper are:

- Foundation model selected to LoRA fine-tune
- Hardware requirements for fine-tuning and inference
- Software selection for parameter-efficient fine-tuning
- Fine-tuned LLM

Requirements

The following are the hardware and software requirements:

- Hardware
  - Server with 1x NVIDIA A100 80GB GPU minimum (see the Requirements for Parameter Efficient Fine-Tuning (PEFT) section for more information)
- Software - one of the following fine-tuning software options:
  - NeMo Framework Training container from NGC Catalog
  - Python environment with PEFT library installed

Terminology

This section is meant to clarify the meaning of some of the generative AI-specific terms used in this article.

- A **large language model (LLM)** is an AI model used for language tasks that is large (billions of parameters or more) and is typically trained on vast amounts of text data.
- **Parameters** refer to the weights and biases of a neural network model. These values are learned and determine how the model behaves. In fine-tuning, parameters are adjusted to adapt the model to a specific task or domain. The count of parameters in a model is referred to as the model's **size**.
- **Pre-training** is the process of training a language model on a large corpus of text data before fine-tuning it for specific tasks. During pre-training, the model learns general language patters and representations.
- A **base (or foundation) model** is a pre-trained model that can be fine-tuned for many specific tasks. It is the starting point for customization. Creators of foundation large language models, such as OpenAI and Meta, typically release pre-trained versions and versions that are fine-tuned to chat or follow instructions, as this is a very common use case of large language models.
- **Fine-tuning** is the process of adapting a pre-trained model to a specific task or domain using specific data.
- **Parameter-efficient fine-tuning (PEFT)** is a growing collection of fine-tuning techniques that reduce the compute and data needed to effectively tailor a model to a specific domain or task.
- **Low-Rank Adaptation (LoRA)** is a technique for fine-tuning that adds a relatively small number of trainable parameters to a pre-trained model and freezes the initial parameters of the model. This reduces the model's complexity and allows for fine-tuning with limited resources or data.
- **Inference** refers to using a trained model to make predictions or generate output based on input data. During inference, a model applies its learned knowledge to new examples. The computing resources needed for inference are generally much lower than those needed for training.
- **Tokens** are discrete units of text, such as a word or subword. These are the most basic blocks of language that language models ingest and generate.
- **Throughput** measures the rate at which a language model processes input data, typically measured in tokens per second or overall inputs, such as sentences, per second. Optimizing throughput is crucial for real-time applications, especially conversational AI such as chatbots and large-scale language processing.
- **Latency** refers to the time delay between inputting data for inference and receiving the output. High latency can degrade user experience, especially in interactive systems. There is typically a trade-off between optimizing latency, decreasing the wait time for a single response, and optimizing throughput, increasing the total rate of data processing.
- **Batch size** is the number of simultaneous inputs passed to a model. Changing batch size at inference is the lever that balances throughput and latency: lower batch sizes decrease latency and throughput, while higher batch sizes increase them.
- **Tensor parallelism** is a method used to distribute the computational load of large models across multiple GPUs or other processors. It involves splitting the model's tensors - multi-dimensional arrays of parameters - across different devices, allowing for simultaneous computation.

Confirm Model Selection and Sizing

The two most important parts of a RAG application are first, the data used in retrieval, and second, the AI model used to extract and synthesize information from that data and generate a response in the desired way.

In Part 1 of this series, we defined several factors to consider when selecting a generative large language model to fine-tune for RAG: model size, benchmark performance, context size, license terms, and domain. In Part 2, we explained the process for creating a dataset for fine-tuning a language model to perform RAG on enterprise data.

Now that we are prepared with a dataset to customize an LLM through fine-tuning, we should double-check that we will be able to fine-tune this model with the hardware available in development, and that the model can be inferenced with the latency and throughput required for success with the hardware available in production. The most important factor in these considerations is the model's size.

Requirements for Parameter Efficient Fine-Tuning (PEFT)

The GPU requirements for PEFT techniques, such as LoRA, are dependent on the size of the base model.

- Relatively small language models, such as Llama 2 7B, Llama 2 13B, Llama 3 8B, and Mistral 7B require a 1x NVIDIA A100 80GB GPU at minimum.
- Larger language models, such as Llama 2 70B, Llama 3 70B, and Mixtral 8x7B require at least 4x NVIDIA A100 80GB GPUs

Ensure you have the necessary hardware to fine-tune your selected base model.

Requirements for Inference

Additionally, it is important to consider the hardware you will have in production and the speed performance requirements of your language model. The NVIDIA MLPerf AI Benchmarks page shows an MLPerf result for inference throughput of 34,864 tokens/second for Llama 2 70B with the H200 GPU. With more GPUs, you can take advantage of tensor parallelism to improve performance. Additionally, evaluating different batch sizes within each model size-GPU count setup can help find the right balance between throughput and latency. The posted measurements show values for different numbers of GPUs and batch sizes.

Before fine-tuning, consider what latency and throughput will be acceptable in your system. There are some rules-of-thumb to consider when setting latency targets for human-computer interaction:

- Latency below 100 milliseconds is perceived as instantaneous
- Latency below 200 milliseconds is perceived to be acceptable in the flow of a conversation, a relevant threshold for live chats and other real-time assistants
- Latency above 500 milliseconds is perceived as a noticeable delay, important for applications where the user expects a response but is not engaged in a real-time, on-going interaction
- Latency above 1 second can impact user attention and may cause frustration

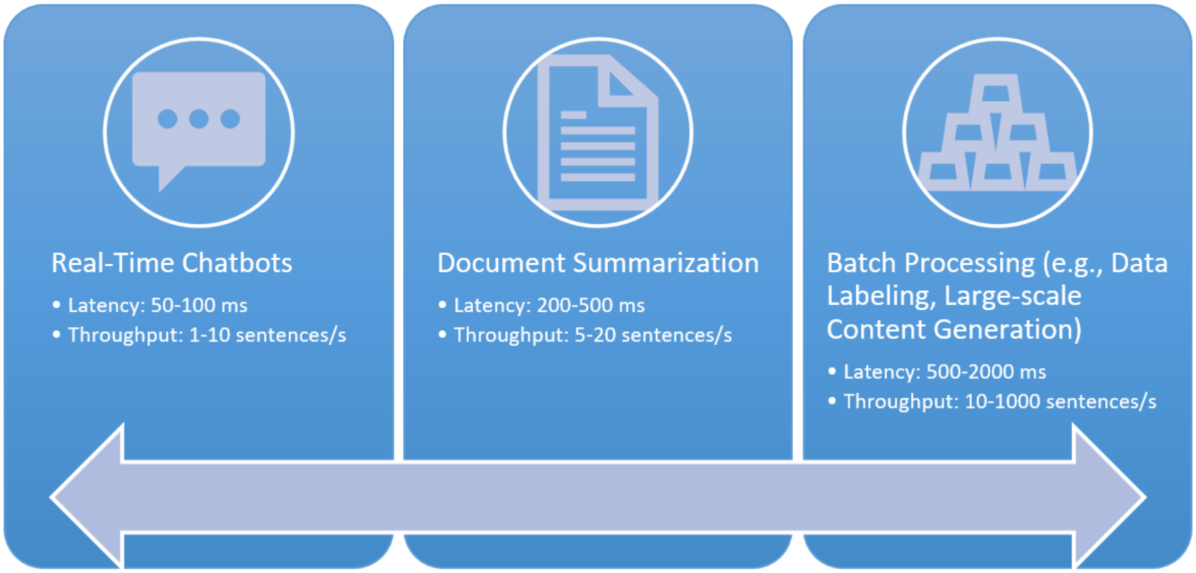
Real-Time Chatbots
• Latency: 50-100 ms
• Throughput: 1-10 sentences/s
Document Summarization
• Latency: 200-500 ms
• Throughput: 5-20 sentences/s
Batch Processing (e.g., Data Labeling, Large-scale Content Generation)
• Latency: 500-2000 ms
• Throughput: 10-1000 sentences/s

Figure 1. Common applications of large language models have varying latency and throughput requirements, from near-instantaneous chatbots to offline, mass data processing

For a given model and hardware, there is a tradeoff between latency and throughput. For real-time conversational applications, like chatbots, minimizing latency is essential to the user experience. Other applications of large language models, like offline data labeling, require high throughout to process large amounts of text, and the high latency that comes with that is acceptable, because no one is waiting for immediate results. Figure 1 shows a few common large language model applications and typical requirements along the latency/throughput spectrum.

After considering these factors, you should have targets of latency and throughput, along with a selected foundation model and the required hardware.

Deliverable: foundation model selected to LoRA fine-tune, hardware requirements for fine-tuning and inference

PEFT Software Implementation

Two software options for performing LoRA fine-tuning are the NVIDIA NeMo toolkit and the HuggingFace PEFT library. There are pros and cons to each option.

- The **NeMo toolkit** integrates seamlessly with the rest of NVIDIA's ecosystem, including TensorRT-LLM for optimized inference. It includes containers for easily building fine-tuning environments.
- The **PEFT library** integrates well with HuggingFace Hub, the largest collection of open source LLMs, and with the Transformers library for inference. There is a large online community for support and documentation.

Overall, the choice of software for fine-tuning should come down to what ecosystem you are already using or familiar.

For either software option, the general steps in fine-tuning will be the same:

1. Download the base model
2. If necessary, change the format of the base model
3. Prepare the data into the required format for fine-tuning (see Part 2 of this series)
4. Perform LoRA fine-tuning, defining the base model to fine-tune, dataset to learn from, and hardware devices on which to compute.
5. Evaluate the resulting model on a hold-out dataset.

Once you have chosen a base model and have your dataset ready, you should follow the documentation for either software option to complete the LoRA fine-tuning process. Both software options have extensive documentation for running LoRA fine-tuning.

We suggest starting with these pages, which walk step-by-step through the process:

- NeMo Framework PEFT Playbook
- LoRA methods guide for the PEFT library

Example Implementation

NeMo makes it very easy to fine-tune popular model architectures like Llama and Mistral. Within the NeMo toolkit, we can use the `megatron_gpt_finetuning.py` script and set some configurations, and we are ready to fine-tune the model. This section describes an example script for fine-tuning the Llama 2 7B model. It is based on an example from NeMo, and we break it down step-by-step.

In this example, our current directory contains two key components for fine-tuning:

- A directory called `datasets` – within this directory is a subdirectory with a descriptive name of the dataset we are fine-tuning on, and within that are JSONL files with splits for training, validation, and testing.
- A `.nemo` model file for the Llama 2 7B model. The model can be downloaded from a repository such as Hugging Face and then converted to .nemo format with the NeMo toolkit.

Here is a look at the directory structure as used in the script:

```
|- datasets
 |- chatbot_questions_and_answers
  |- train.jsonl
  |- valid.jsonl
  |- train.json
|- llama2-7b.nemo
```

Start the script by defining variables that indicate where to find the dataset and model.

```
# This is name of the dataset we are fine-tuning on
MODEL_DATSET_NAME="chatbot_questions_and_answers"

# This is the nemo model we are finetuning
# Change this to match the model you want to finetune
MODEL="./llama2-7b.nemo"

# These are the training datasets (in our case we only have one)
TRAIN_DS="[datasets/"${MODEL_DATSET_NAME}"/train.jsonl]"

# These are the validation datasets (in our case we only have one)
VALID_DS="[datasets/"${MODEL_DATSET_NAME}"/valid.jsonl]"

# These are the test datasets (in our case we only have one)
TEST_DS="[datasets/"${MODEL_DATSET_NAME}"/test.jsonl]"

# These are the names of the test datasets
TEST_NAMES="["${MODEL_DATSET_NAME}"]"
```

Next we define the type of fine-tuning we would like to do. Here we choose a very popular option, LoRA.

```
# This is the PEFT scheme that we will be using. Set to "ptuning" for P-Tuning instead of LoRA
PEFT_SCHEME="lora"
```

If we are using multiple training datasets, we should set the sampling probability for each. Since we are only using one training dataset in this example, we assigned its probability to 1.

```
# This is the concat sampling probability. This depends on the number of files being passed in the train set
# and the sampling probability for each file. In our case, we have one training file. Note sum of concat sampling
# probabilities should be 1.0. For example, with two entries in TRAIN_DS, CONCAT_SAMPLING_PROBS might be
# "[0.3,0.7]". For three entries, CONCAT_SAMPLING_PROBS might be "[0.3,0.1,0.6]"
# NOTE: Your entry must contain a value greater than 0.0 for each file
CONCAT_SAMPLING_PROBS="[1.0]"
```

Next, we set variables according to the hardware on which we are fine-tuning. Tensor parallel size (TP_SIZE) should be set to the number of GPUs available, while pipeline parallel size (PP_SIZE) should stay at 1.

```
# This is the tensor parallel size (splitting tensors among GPUs horizontally
)
TP_SIZE=1

# This is the pipeline parallel size (splitting layers among GPUs vertically
)
PP_SIZE=1

# The number of nodes to run this on
NODE_COUNT=1

# The number of total GPUs used
GPU_COUNT=1
```

The last variable to set is the location for the fine-tuning results to be saved. This will be where the fine-tuned model checkpoints will be stored.

```
# Where to store the finetuned model and training artifacts
OUTPUT_DIR="datasets/"${MODEL_DATSET_NAME}"/results"
```

Finally, we run the training script. This command contains typical hyperparameters for deep learning training, including the maximum number of training steps, batch size, optimizer, and learning rate. At the end of training, the program will output a loss value of the final model. We recommend testing if different learning rates and optimizers can improve the performance at low maximum step values to determine an effective training setup.

```
# Run the PEFT command by appropriately setting the values for the parameters
such as the number of steps,
# model checkpoint path, batch sizes etc. For a full reference of parameter
# settings refer to the config at https://github.com/NVIDIA/NeMo/blob/main/ex
amples/nlp/language_modeling/tuning/conf/megatron_gpt_finetuning_config.yaml

python /opt/NeMo/examples/nlp/language_modeling/tuning/megatron_gpt_finetuni
ng.py \
    trainer.log_every_n_steps=1 \
    trainer.precision=bf16 \
    trainer.devices=${GPU_COUNT} \
    trainer.num_nodes=1 \
    trainer.val_check_interval=10 \
    trainer.max_steps=2048 \
    model.restore_from_path=${MODEL} \
    model.peft.peft_scheme=${PEFT_SCHEME} \
    model.micro_batch_size=1 \
    model.global_batch_size=128 \
    model.tensor_model_parallel_size=${TP_SIZE} \
    model.pipeline_model_parallel_size=${PP_SIZE} \
    model.megatron_amp_O2=True \
    model.activations_checkpoint_granularity=selective \
    model.activations_checkpoint_granularity=selective \
    model.activations_checkpoint_num_layers=null \
    model.activations_checkpoint_method=uniform \
    model.optim.name=fused_adam \
    model.optim.lr=1e-4 \
    model.answer_only_loss=True \
    model.data.train_ds.file_names=${TRAIN_DS} \
    model.data.validation_ds.file_names=${VALID_DS} \
    model.data.test_ds.file_names=${TEST_DS} \
    model.data.train_ds.concat_sampling_probabilities=${CONCAT_SAMPLING_PROB
S} \
    model.data.train_ds.max_seq_length=10000 \
    model.data.validation_ds.max_seq_length=10000 \
    model.data.train_ds.micro_batch_size=1 \
    model.data.train_ds.global_batch_size=128 \
    model.data.validation_ds.micro_batch_size=1 \
    model.data.validation_ds.global_batch_size=128 \
    model.data.train_ds.num_workers=0 \
    model.data.validation_ds.num_workers=0 \
    model.data.test_ds.num_workers=0 \
    model.data.validation_ds.metric.name=loss \
    model.data.test_ds.metric.name=loss \
    exp_manager.create_wandb_logger=False \
    exp_manager.checkpoint_callback_params.mode=min \
    exp_manager.explicit_log_dir=${OUTPUT_DIR} \
    exp_manager.resume_if_exists=True \
    exp_manager.resume_ignore_no_checkpoint=True \
    exp_manager.create_checkpoint_callback=True \
    exp_manager.checkpoint_callback_params.monitor=validation_loss \
    ++exp_manager.checkpoint_callback_params.save_best_model=False \
    exp_manager.checkpoint_callback_params.save_nemo_on_train_end=True \
    model.save_nemo_on_validation_end=False
```

Deliverable: fine-tuned LLM

Conclusion

We have provided guidance on how to ensure the foundation model selected for fine-tuning on a custom dataset will work with the hardware available in development and meet the production performance requirements. Additionally, we have introduced two software options for performing parameter-efficient fine-tuning.

Both the NeMo toolkit and PEFT Python library enable us to customize models with our own datasets and integrate with the NVIDIA and HuggingFace ecosystems, respectively. With either of these software options, we can customize a model through fine-tuning, improving its ability to perform the tasks associated with our RAG use case.

This concludes the series on Making LLMs Work for Enterprise.

- In Part 1, we covered the considerations for selecting the LLMs used in the processes of synthetic dataset generation and evaluation as well as the LLM to fine-tune for an application. We also introduced RAGAS, a framework for continuous evaluation of a RAG system.
- In Part 2, we explained the process for deciding on the scope of an LLM application, compiling a collection of documents for RAG, and creating a synthetic dataset for fine-tuning.
- And finally, in Part 3 (this document), we covered popular methods for fine-tuning an LLM using LoRA. By following this process, you can create a customized LLM with high accuracy and controllability for enterprise use cases.

Authors

David Ellison is the Chief Data Scientist for Lenovo ISG. Through Lenovo's US and European AI Discover Centers, he leads a team that uses cutting-edge AI techniques to deliver solutions for external customers while internally supporting the overall AI strategy for the World Wide Infrastructure Solutions Group. Before joining Lenovo, he ran an international scientific analysis and equipment company and worked as a Data Scientist for the US Postal Service. Previous to that, he received a PhD in Biomedical Engineering from Johns Hopkins University. He has numerous publications in top tier journals including two in the Proceedings of the National Academy of the Sciences.

Chris Van Buren is a Staff Data Scientist at Lenovo. He researches generative AI for enterprise use cases and has developed retrieval augmented generation (RAG) applications with open source, on-premises LLMs.

Related product families

Product families related to this document are the following:

- Artificial Intelligence
- ThinkSystem SR675 V3 Server
- ThinkSystem SR680a V3 Server
- ThinkSystem SR685a V3 Server
- ThinkSystem SR780a V3 Server

Notices

Lenovo may not offer the products, services, or features discussed in this document in all countries. Consult your local Lenovo representative for information on the products and services currently available in your area. Any reference to a Lenovo product, program, or service is not intended to state or imply that only that Lenovo product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any Lenovo intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any other product, program, or service. Lenovo may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

Lenovo (United States), Inc.
8001 Development Drive
Morrisville, NC 27560
U.S.A.
Attention: Lenovo Director of Licensing

LENOVO PROVIDES THIS PUBLICATION ”AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. Lenovo may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

The products described in this document are not intended for use in implantation or other life support applications where malfunction may result in injury or death to persons. The information contained in this document does not affect or change Lenovo product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of Lenovo or third parties. All information contained in this document was obtained in specific environments and is presented as an illustration. The result obtained in other operating environments may vary. Lenovo may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any references in this publication to non-Lenovo Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this Lenovo product, and use of those Web sites is at your own risk. Any performance data contained herein was determined in a controlled environment. Therefore, the result obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

This document, LP1955, was created or updated on November 27, 2024.

Send us your comments in one of the following ways:

- Use the online Contact us review form found at:
  https://lenovopress.lenovo.com/LP1955
- Send your comments in an e-mail to:
  comments@lenovopress.com

This document is available online at https://lenovopress.lenovo.com/LP1955.

Trademarks