How to run performance and scale validation for OpenShift AI

April 30, 2025 | Alberto Perdomo, Kevin Pouget

Related topics: Artificial intelligence

Related products: Red Hat AI, Red Hat OpenShift AI

Share:

Table of contents:

Imagine having the ability to customize a large language model (LLM) to talk like your company, know about your business, and help you fix your specific business challenges with precision. This is not something from the future. It's the current reality of fine-tuning LLMs at scale, a capability that is transforming how organizations use AI to get ahead of their competitors.

In this series, we'll share our latest findings on fine-tuning LLMs with Red Hat OpenShift AI. These insights will be valuable whether you're customizing models for specific use cases or scaling AI operations across multiple cloud environments. In this article, we will introduce our model fine-tuning stack and discuss how we run performance and scale validation of the fine-tuning process.

Using OpenShift AI to build applications

Red Hat OpenShift AI provides enterprises a comprehensive platform for building and deploying impactful AI applications. It is a central management system for coordinating everything from data ingestion to model serving across hybrid cloud environments, allowing organizations to focus on creating value rather than managing infrastructure.

Our team, Performance and Scalability for AI Platforms (PSAP), continues

to advance LLM fine-tuning capabilities among other innovations. We're developing solutions that enable enterprises to leverage AI effectively while maintaining robust security and compliance standards essential for business operations.

AI infrastructure setup has traditionally presented significant challenges. Many ML engineers have encountered difficulties with dependencies and resource orchestration. OpenShift AI addresses these pain points directly, enabling users to fine-tune models and deploy applications with greater efficiency.

This approach transforms AI development into a reliable, enterprise-grade workflow that operates seamlessly across hybrid cloud environments. It eliminates the need to choose between innovation and stability or between speed and security.

Model fine-tuning stack

Imagine a musician carefully adjusting their instrument until it produces the perfect note, or an F1 engineer tweaking the engine for the best performance.

In the AI domain, we adapt powerful, pre-trained models for specialized tasks. It's comparable to providing a talented generalist with expertise in your specific business domain.

The challenge lies in the resources required to train these large models from scratch. The computational and financial demands are substantial, making this approach not feasible for most organizations. Fine-tuning has, therefore, become the preferred method in modern AI development. This impact has been so great that we now refer to initial model training as "pre-training," with fine-tuning representing the important final stage.

OpenShift AI streamlines this process, supporting both custom images and public repository integration. We've incorporated fms-hf-tuning, a robust toolkit developed by our colleagues at IBM Research, which leverages HuggingFace SFTTrainer and PyTorch FSDP. This versatile solution can be utilized as a Python package or implemented in our

experiments as a container image with enhanced scripting capabilities.

Technical note: Our experiments use different fms-hf-tuning images based on the OpenShift AI version (more on that later).

We're going to take a deep dive into exploring three different flavors of fine-tuning on OpenShift AI:

1. **Full parameter fine-tuning**: The conventional method that adjusts all parameters within the network.
2. **LoRA** (low rank adapters): An efficient approach that achieves fine-tuning with significantly reduced resource requirements.
3. **QLoRA** (quantized low rank adapters): One of the latest advancements in resource-efficient fine-tuning techniques.

Each methodology offers specific advantages and limitations. We'll examine when and why you might select one approach over another based on your particular use case.

Full parameter fine-tuning

Think of full parameter fine-tuning as providing your LLM with a comprehensive update rather than a minor adjustment. Unlike the more efficient approaches, LoRA and QLoRA (which we'll discuss later), this method adjusts every parameter in your neural network (see Figure 1).

What makes this different from building a model from scratch? In this approach, model weights aren't randomly initialized. Instead, these weights already encode valuable features. With full parameter fine-tuning, you're working with a model that already has a solid foundation. You're simply adapting it to your specific requirements.

What makes full parameter fine-tuning distinctive?

- **Precision through complete control**: By adjusting every parameter, you're effectively teaching the model to understand your domain thoroughly. Your task-specific dataset becomes the model's comprehensive training program.

- **Resource-intensive but valuable**: This process requires significant computational resources and time. However, for applications where accuracy is essential, this investment often provides the best returns.
- **Enhanced task-specific performance**: Upon completion, your model demonstrates exceptional capability in its specialized task. It effectively transforms a general-purpose AI into one with expertise in your specific domain.

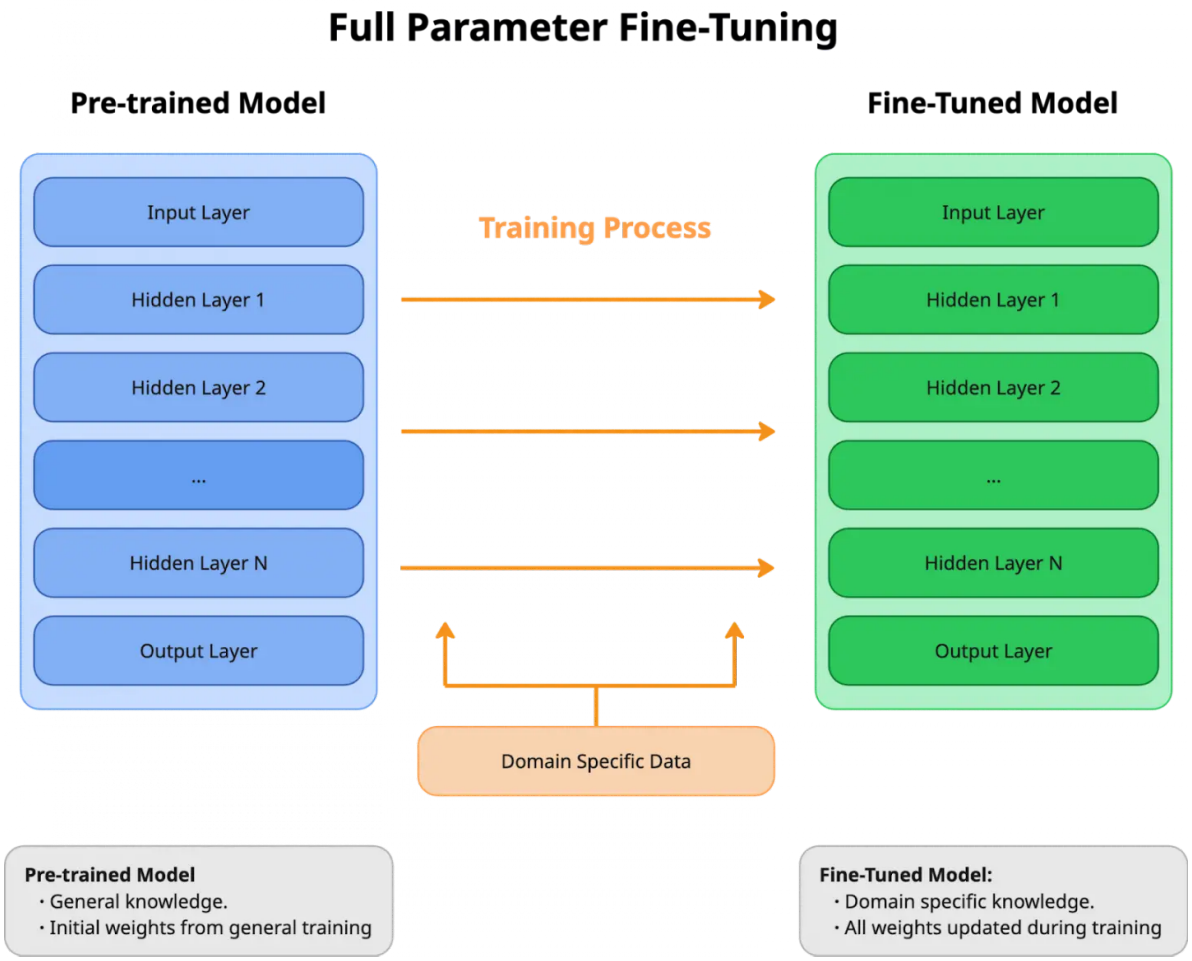


Figure 1: Full parameter fine-tuning diagram.

Low-rank adaptation, LoRA

Low-rank adaptation (LoRA) is an efficient approach within the parameter efficient fine-tuning (PEFT) family of algorithms. While full parameter fine-tuning modifies the entire model, LoRA adopts a more targeted strategy (see Figure 2).

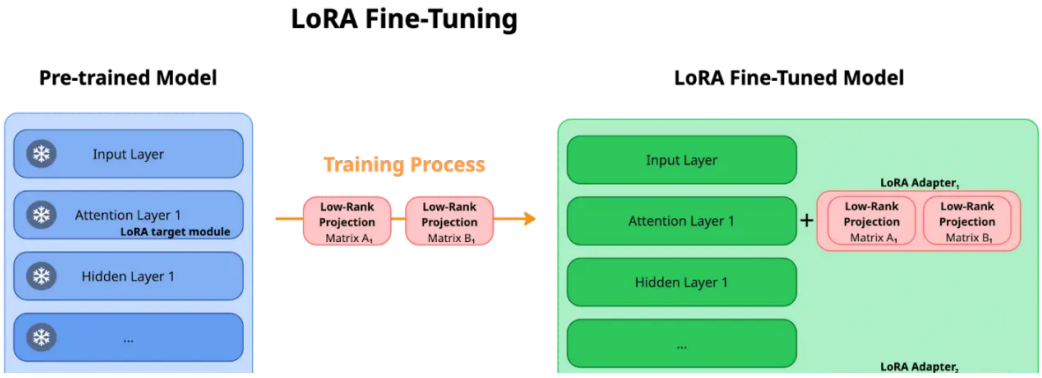
The innovative aspect of LoRA is that it preserves the original model parameters and introduces two compact matrices (e.g., adapters) that work together to create the necessary adjustments. The original model remains unchanged while still providing fine-tuning benefits, offering efficiency without sacrificing effectiveness.

LoRA's effectiveness centers on two key parameters:

- **rank dimension (r):** This represents the complexity of your adjustments. A smaller r-value requires less memory but may offer less precise control.
- **alpha (scaling factor):** This determines the intensity of your adjustments on the final output.

Advantages of the LoRA algorithm:

- **More efficiency:** By focusing on a subset of parameters, LoRA completes training significantly faster than full parameter methods.
- **Resource optimization:** LoRA's streamlined approach reduces GPU memory requirements, enabling fine-tuning of models that would otherwise be too large to process.
- **Versatility:** LoRA allows you to develop different versions of your model for various tasks by simply exchanging adapters, providing flexibility and efficiency.
- **Strategic focus:** LoRA typically concentrates its modifications in the attention layers, where much of the model's critical processing occurs. This targeted approach maximizes impact while minimizing resource usage.



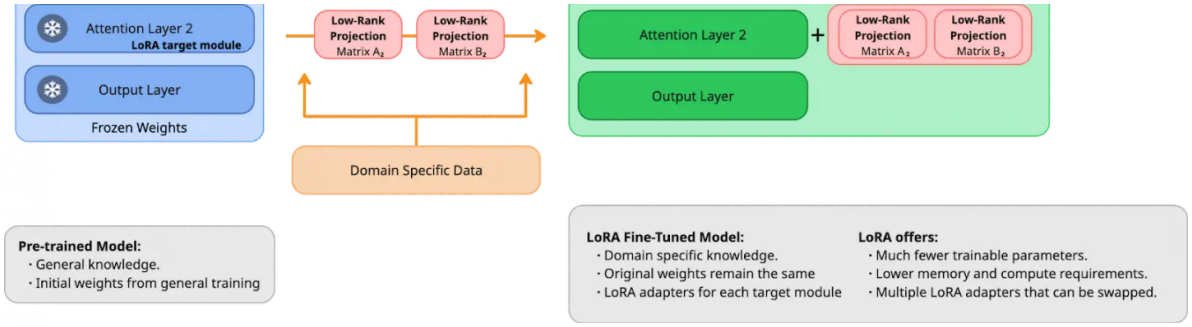


Figure 2: LoRA fine-tuning diagram.

Quantized low-rank adaptation, QLoRA

Quantized low-rank adaptation (QLoRA) is a highly efficient approach to fine-tuning. The key innovation is quantization, storing model weights in 4-bit precision rather than standard full precision (see Figure 3).

However, while the model weights are compressed, QLoRA performs its learning operations in full precision. During each training update, the model temporarily returns to full precision, makes necessary adjustments, and then compresses again, effectively balancing efficiency with accuracy.

Key advantages of QLoRA, similar to LoRA:

- **Memory efficiency**: Substantially reduces GPU memory requirements by keeping the parameters in their quantized form most of the time.
- **Accessibility**: Enables fine-tuning of substantial models on consumer-grade GPUs, making advanced AI development more accessible.
- **Strategic precision**: Maintains full precision for critical low-rank matrices where accuracy is most important.

Processing time is the primary trade-off. The quantization and dequantization processes add additional steps to the training. However, for many applications, this is a reasonable compromise for the ability to work with models that would otherwise require specialized hardware.

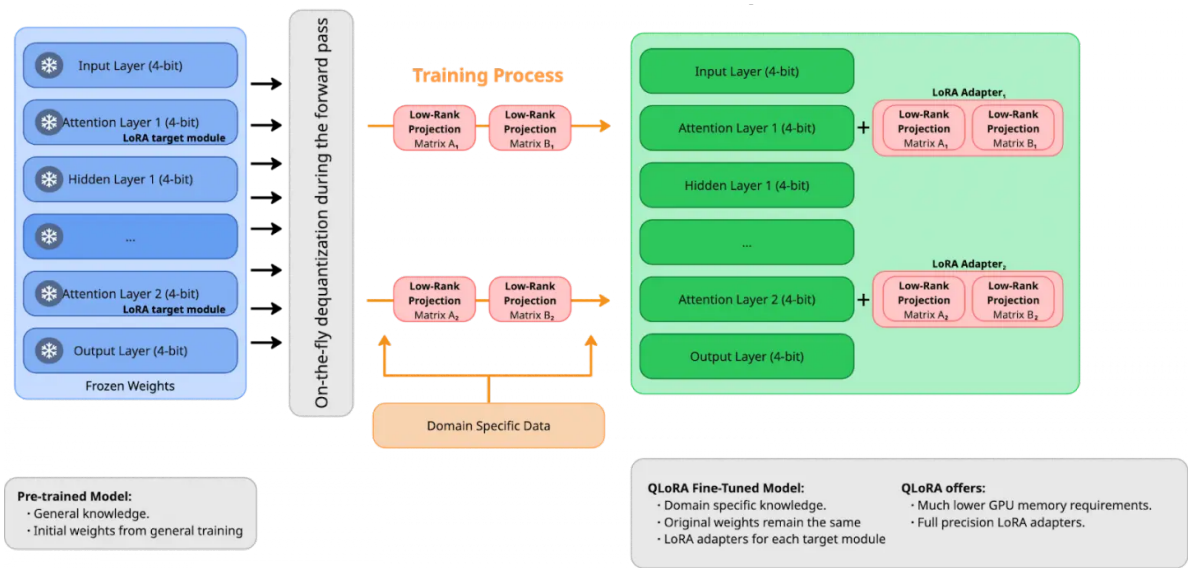


Figure 3: QLoRA fine-tuning diagram.

These tests ran on a Red Hat OpenShift cluster with 4xH100-80GB NVIDIA GPUs. To run these tests in an automated, transparent, and reproducible manner, we use test orchestrator for performance and scalability of AI platforms (TOPSAIL), which uses a combination of Python code for orchestration and Red Hat Ansible Automation Platform roles for cluster control.

Training results

Diving deeper into our journey, we employed the fms-hf-tuning image to conduct our fine-tuning experiments. We put three distinct approaches through their paces: full fine-tuning, LoRA, and QLoRA. Let's pull back the curtain on our test settings and unpack what we discovered.

Technical note: We focused specifically on benchmarking the fine-tuning infrastructure rather than evaluating the model's accuracy through LLM testing. The main objective was to assess the performance of OpenShift AI fine-tuning stack and compare it against internal benchmarks. Think of it as testing the kitchen rather than rating the meal.

For this exploration, we examined results across three OpenShift AI versions:

- **OpenShift AI 2.16** paired with fms-hf-tuning v2.2.1

- **OpenShift AI 2.17** paired with fms-hf-tuning v2.5.0
- **OpenShift AI 2.18** paired with fms-hf-tuning v2.6.0

For the following experiments, we adhered to the default fms-hf-tuning parameters to train a diverse set of models, varying in architecture and size. It's important to note that the performance metrics presented here do not represent optimal performance and could be significantly enhanced by tuning the appropriate fms-hf-tuning parameters.

In upcoming articles, we'll dive into the technical implications of these findings. But for now, we're keeping our spotlight on the automation framework. While we gathered a different kind of metrics during our fine-tuning tests, we chose three that we considered the most informative in terms of performance:

- **Train runtime:** How long the process takes.
- **Train throughput:** How efficiently it processes data.
- **Maximum GPU memory usage across all GPUs:** How resource-hungry the process becomes.

Let's dive into what these numbers tell us.

The first testing setup

When your AI needs a complete wardrobe change, not just a new tie. In the first test scenario, the selected method is full parameter fine-tuning, which is the traditional approach where all model parameters are updated during training. While this method typically achieves the best inference results, it requires significant computational resources since the entire neural network is modified.

This test consists of running full parameter fine-tuning for a given set of models in an OpenShift cluster. The test settings are as follows:

- Dataset: Cleaned Alpaca Dataset
- Replication factor of the dataset (where 1.0 represents the full dataset): 0.2

- Number of accelerators: 4xH100-80GB NVIDIA GPU
- Maximum sequence length: 512
- Epochs: 1

The list of fine-tuned models:

- ibm-granite/granite-3b-code-instruct
- ibm-granite/granite-8b-code-base
- instructlab/granite-7b-lab
- meta-llama/Llama-2-13b-hf
- meta-llama/Meta-Llama-3.1-70B
- meta-llama/Meta-Llama-3.1-8B
- mistralai/Mistral-7B-v0.3
- mistralai/Mixtral-8x7B-Instruct-v0.1

Our earlier full parameter fine-tuning tests revealed intriguing regressions (which we'll dissect in future articles). But for now, let's focus on our comparative analysis across the three OpenShift AI versions. As mentioned, each version comes paired with its own fms-hf-tuning image, and Topsail proved to be our secret weapon, allowing us to automate fine-tuning evaluations while flexibly adjusting GPU counts and hyperparameters to fit various scenarios.

The first thing that one can notice in these initial results is that not all of the models manage to finish the training loop, as shown in Figure 4. The reason behind this is that for full fine-tuning, all of the weights are loaded into memory at full precision and actively participate in both forward and backward passes, receiving gradient updates after each step, triggering OOM errors for very large models (i.e., hitting the hard ceiling of your GPU's VRAM capacity).