Open Source VLM Fine Tuning: LoRA

Kazi Islam

Version 1.0

Prepared for Prof David Smith

Date [11-24-2025]

1. Abstract

This project serves as an introductory exploration of Low-Rank Adaptation (LoRA), a modern fine-tuning technique designed to efficiently adapt large language models (LLMs) without retraining all their parameters. As a beginner, my goal is to understand what LoRA is, how it functions conceptually, and how it can be implemented in practice for the Balanced Blended Space (BBS) project. The study begins by reviewing the core principles of LoRA—its low-rank matrix decomposition and parameter-efficient design—followed by step-by-step experimentation using open-source models such as Qwen2-VL, LLaVA-OneVision, or Idefics-2. Implementation attempts will focus on setting up LoRA environments, identifying compatible model architectures, adjusting key hyperparameters (rank, alpha, and target layers), and testing

outcomes on small BBS datasets. The broader objective is to build foundational knowledge of fine-tuning workflows while assessing whether LoRA can make model customization feasible for small-scale research teams working with limited hardware and open-source tools.

2. Objective

The objectives of this document are-

1. To introduce and explain the core concept of LoRA — what it is, why it was developed, and how it compares to traditional fine-tuning methods.

2. To review the key benefits and trade-offs of LoRA: reduced trainable parameters, lower memory/compute demands, faster turnaround, and portability of adapter weights. Coralogix+2truefoundry.com+2

3. To provide a beginner-friendly, step-by-step implementation guide for applying LoRA on an open-source large language model: covering model selection, data preparation, LoRA configuration (rank r, alpha, target modules), training, saving and loading adapters.

3. Background

The landscape of large-language-model (LLM) deployment has shifted dramatically in recent years. Models with tens or hundreds of billions of parameters are now common, providing remarkable capabilities—but also imposing substantial resource demands. Training such models from scratch is prohibitively expensive for most organizations, often requiring thousands of GPUs and weeks or months of compute. (VLDB) Even when using pretrained base models, classical fine-tuning by updating *all* model weights remains a heavy burden—consuming large amounts of GPU memory, storage space, and energy, and complicating versioning and deployment.

In response to this challenge, the concept of parameter-efficient fine-tuning (PEFT) emerged as a practical and scalable alternative. Techniques under the PEFT umbrella aim to retrofit pretrained models to new tasks by tuning only a limited subset of parameters, leaving the bulk of the original model fixed. (arXiv) Among these, Low-Rank Adaptation (LoRA) has gained prominence for its elegant engineering: instead of modifying the full weight matrices, LoRA injects low-rank trainable modules into key transformer layers, thereby capturing the task-specific adaptation in a compact form. (VLDB)

4. Principles of Low-Rank Adaptation (LoRA)

The core idea behind Low‑Rank Adaptation (LoRA) is that when adapting a large pretrained model to a new task, you don't need to change *all* of its weights. Instead, you freeze the pretrained model's weights and add a pair of much smaller, trainable matrices into certain layers. Those smaller matrices capture how the model needs to change for the new task. Hugging Face+2Databricks+2

In more concrete terms: imagine one big weight matrix that originally has to change to adapt the model. LoRA replaces that full update with two smaller matrices (often called adapter matrices) whose product approximates the change. Because those matrices are much smaller ("low-rank"), you train far fewer parameters. The model still does what it needs to do for the new task, but with much less compute and less memory. Medium+1

In transformer-based models (which many large language models are), LoRA is commonly applied to the query and value projection layers of the attention block. You freeze those original matrices and insert the low-rank adapters only there, which often gives most of the adaptation benefit while keeping the cost low. Medium+1

Another important point is that at inference time, you can merge the adapter matrices back into the original weights (or equivalently incorporate their effect), so you don't incur extra runtime cost. That means you get the benefits of adaptation without making inference slower. Databricks+1

5. How to Use LoRA

Step 1: Choose a pretrained model & freeze its main weights

What it means: You'll start with a large model that's already been trained (on broad/general data). Instead of re-training everything, you keep most of it unchanged.

What you do: Load your base model (for example a transformer-based language model or a vision-language model). Then set the main weights so they are *frozen* (i.e., they won't change during your adaptation). In code for frameworks like PyTorch, this often means for each parameter `param.requires_grad = False`.

Why it matters: The model already has broad knowledge; you don't waste time or resources retraining everything. Freezing main weights saves memory, speed, and makes the adaptation more efficient.

Step 2: Insert a small adapter (trainable module)

What it means: Rather than updating the full model, you add a small extra module (an adapter) in certain layers of the model. This adapter has *fewer parameters* (low-rank) so it's cheaper to train.

What you do: Decide which layers to adapt (commonly in a transformer, the "query" or "value" projection layers in attention). Add the adapter module there — often two small matrices whose product approximates the change to that layer's weights. Use a library (e.g., PEFT with LoRA) which helps inject these modules.

Why it matters: Because you're only training a small extra module, you reduce compute/memory cost while still adapting the model for your specific task.

Step 3: Set up your training configuration

What it means: You define how you will train the adapter — which modules, how big the adapter is, what data you'll use, and what hardware constraints you have.
What you do:

- Choose target_modules: the layer names where you inserted the adapter.
- Choose adapter size (rank *r*), scaling (often "alpha"), dropout if needed.
- Prepare your domain data: prompts, responses (or whatever fits your task). Tokenize, split train/validation.
- Check hardware: ensure your GPU/CPU memory can handle it; set batch size, precision (FP16 or 8-bit) as appropriate.

  Why it matters: These settings determine how efficient and effective your fine-tuning will be. Too large an adapter wastes resources; too small may underfit.

Step 4: Train only the adapter

What it means: You run the training process but only the adapter parameters are updated; the base model weights remain frozen.

What you do: Run your trainer (or custom loop). Ensure optimizer only includes adapter parameters. Monitor training loss & validation metrics. Because you're updating far fewer parameters, memory usage is lower and training can be faster.

Why it matters: This is the core of LoRA's benefit — adapting the model cheaply and quickly without full fine-tuning.

Step 5: Merge adapter (optional) & deploy

What it means: After training, you now have your base model + adapter module. For deployment (making predictions) you can either keep them separate or merge the adapter into the base model weights so you deploy a single model file.

What you do:

- **Separate mode:** At inference, load base model + adapter weights dynamically.

- **Merged mode:** Combine adapter changes into the base model weights (effectively computing $W\_new = W\_base + \Delta W$) and deploy the merged model.

Why it matters: Merging means no extra overhead at inference time — the model runs just like a regular model but with your fine-tuning built in. Keeping them separate gives flexibility to swap adapters for different tasks.

Step 6: Evaluate performance & measure efficiency

What it means: Check how well your adapted model works on your task and how much you saved in resources.

What you do:

- Run evaluation metrics relevant to your task (accuracy, relevance, latency, etc.).
- Measure resource usage: GPU memory, training time, checkpoint sizes (adapter vs full model), inference latency.
- Optionally compare with a baseline (base model without adaptation or full fine-tuning) to quantify benefits.
  Why it matters: You want to prove that using LoRA gave you good task performance and resource efficiency — so you know it worked and you can justify using it.

Key Terms & Jargon (Reference List)

Term	Meaning
Base model / pretrained model	The large model you start with (generic, already trained).
Frozen weights	Model parameters you do *not* update during adaptation (they stay unchanged).

Adapter module	A small trainable component you insert into the model to adapt it.
Low-rank matrices	In LoRA, the two smaller matrices (often A & B) whose product approximates the change to a large weight matrix.
ΔW (delta W)	The change to the original weight matrix: your adapter captures this change.
Rank (r)	A hyper-parameter that defines the size of the adapter matrices (smaller => fewer parameters).
Scaling factor (alpha or α)	A hyper-parameter that scales the effect of the adapter updates.
Target modules / target layers	Specific layers in your model (e.g., “q_proj”, “v_proj”) where you insert the adapters.
Merge (adapter merging)	The process of combining adapter weights into base model weights for deployment.
Parameter-Efficient Fine-Tuning (PEFT)	A class of techniques (including LoRA, adapters, prefix tuning) that adapt models by updating fewer parameters.

6. Comparison with Traditional Fine-Tuning Methods

When adapting a large language model (LLM) or vision-language model (VLM), you essentially have two broad pathways: traditional full fine-tuning and newer parameter-efficient techniques like Low-Rank Adaptation (LoRA). Below is a breakdown of how they compare across key dimensions.

6.1. Scope of Parameter Updates

With traditional fine-tuning, you update all of the model’s parameters to adapt it for your task or domain. That means if your base model has billions of parameters, each one is trainable and subject to change. In contrast, with LoRA you freeze the vast majority of parameters and only train a small fraction (the added low-rank adapter matrices) designed to capture the adaptation. Gradient Flow+3Prem AI+3arXiv+3

6.2. Resource Requirements (Compute, Memory, Storage)

Because full fine-tuning changes everything, it demands high computational resources, large memory (GPU/TPU RAM), and considerable storage for the model and checkpoints. For instance, updating all parameters of a 175 billion-parameter model is extremely costly. arXiv+2Medium+2 In contrast, LoRA's smaller number of trainable parameters dramatically lowers memory/compute needs, and requires storing only the small adapter weights (or merged weights) rather than full model duplicates per task. IKANGAI+1

6.3. Speed of Training & Adaptation Turnaround

With fewer trainable parameters, LoRA typically allows faster fine‑tuning cycles, lower memory bottlenecks, and more efficient iteration for beginners or smaller hardware setups. On the other hand, full fine-tuning can offer more flexibility (since all parameters can shift) but often at the cost of longer training time and heavier logistics. Medium

6.4. Risk of Overfitting & Model Generality

Traditional fine-tuning gives maximum flexibility – it can potentially reach higher accuracy on your specific task because you can shift everything in the model. However, that also means increased risk of overfitting to your domain and possibly degrading the model's performance on other tasks or its general knowledge. By contrast, LoRA often acts as a built-in regulariser: by freezing the bulk of the model and only adding small changes, it tends to preserve the base model's knowledge and maintain better generalisation across tasks. Gradient Flow+2arXiv+2

6.5. Deployment & Modular Use

With full fine-tuning, you typically end up with a separate full model instance for each task or domain (each with its modified parameters). That can lead to large storage costs and maintenance burden. With LoRA, you often store just the base model once and then multiple small adapter weights for each task. These adapters can often be swapped in/out or merged into the model for inference. This modularity is very appealing in practice. Prem AI+1

6.6. When to Use Which?

- If your priority is maximum performance and you have abundant compute & memory resources (and you don't mind creating a distinct model per task), traditional full fine-tuning may still be appropriate. Some studies show that for complex domains (e.g., deep reasoning, code generation), full fine-tuning may outperform LoRA. Gradient Flow

- If your priority is efficient adaptation, limited hardware, iterative experimentation, or maintaining a core model with multiple lightweight task-specific adaptations, then LoRA is very compelling. It strikes a balance of performance + efficiency + modular deployment.

7. Key Advantages & Trade-Offs

7.1. Advantages

Reduced compute and memory requirements: LoRA only trains a small number of additional parameters rather than the entire model. This means you can fine-tune large models on more modest hardware. crunchingthedata.com+2Analytics Vidhya+2

Lower storage and versioning overhead: Since the bulk of the model remains unchanged (frozen weights) and you only save the adapter updates, you avoid needing a separate full-model checkpoint for each fine-tuning run. crunchingthedata.com+1

Preservation of the base model's knowledge (better generalisation): Because you freeze the main model and only add updates, LoRA tends to preserve the original model's broad capabilities and reduce the risk of "forgetting" prior knowledge. Gradient Flow+1

Faster iteration cycles: Training fewer parameters means quicker experiments, faster turnaround from "concept to prototype", which is valuable especially for beginners or small teams. crunchingthedata.com+1

Modular task adaptation: You can maintain one base model and swap in different adapters for different tasks, instead of maintaining separate full-models per task. This makes managing multiple adaptations easier and more lightweight.

7.2. Trade-Offs / Limitations

Potential performance gap in very complex tasks: For tasks requiring deep changes in model behaviour or large domain shifts, LoRA may not reach the same level of performance as full fine-tuning of all parameters. Gradient Flow+1

Inference speed may not improve: LoRA is primarily about reducing training cost and parameters. At inference time, if you keep adapter and base model separate (not merged) you might incur extra overhead; merging removes that but still you don't gain major speed boosts purely from using LoRA. crunchingthedata.com+1

Hyperparameter sensitivity: LoRA introduces additional configuration choices (adapter rank r, scaling factor α, which modules to target) which require tuning. Without careful tuning, you may under-utilize its benefits or degrade performance. Gradient Flow+1

Less flexibility for full model adaptation: Because you are only modifying a subset of the model via low-rank updates, if your task demands large structural changes or full re-adaptation of many layers, LoRA may be less suited compared with full fine-tuning. Medium

8. References

- Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). *LoRA: Low-Rank Adaptation of Large Language Models*. arXiv:2106.09685. arXiv+1
- “What is LoRA (Low-Rank Adaptation)?” IBM Think. IBM. (2025). IBM
- “LoRA: LLM Fine-Tuning Through Low-Rank Adaptation.” Medium. Singh, M. (July 2025). Medium
- Zhang, L. et al. (2024). *Efficient Fine-Tuning of Large Language Models via a Low-Rank Gradient Estimator (LoGE)*. Applied Sciences, 15(1):82. MDPI
- “Fine-Tuning (Deep Learning).” Wikipedia. (2025). Wikipedia

9. Version History

Version	Created on	Created by
1.0	11/24/2025	Kazi Islam