Fine Tuning the model Using LoRA And QLoRA

1)Quantization :

- Quantization is a technique to reduce the computational and memory costs of running inference by representing the weights and activations with low-precision data types like 8-bit integer (int8) instead of the usual 32-bit floating point (float32).
- Reducing the number of bits means the resulting model requires less memory storage, consumes less energy (in theory), and operations like matrix multiplication can be performed much faster with integer arithmetic. It also allows to run models on embedded devices, which sometimes only support integer data types.

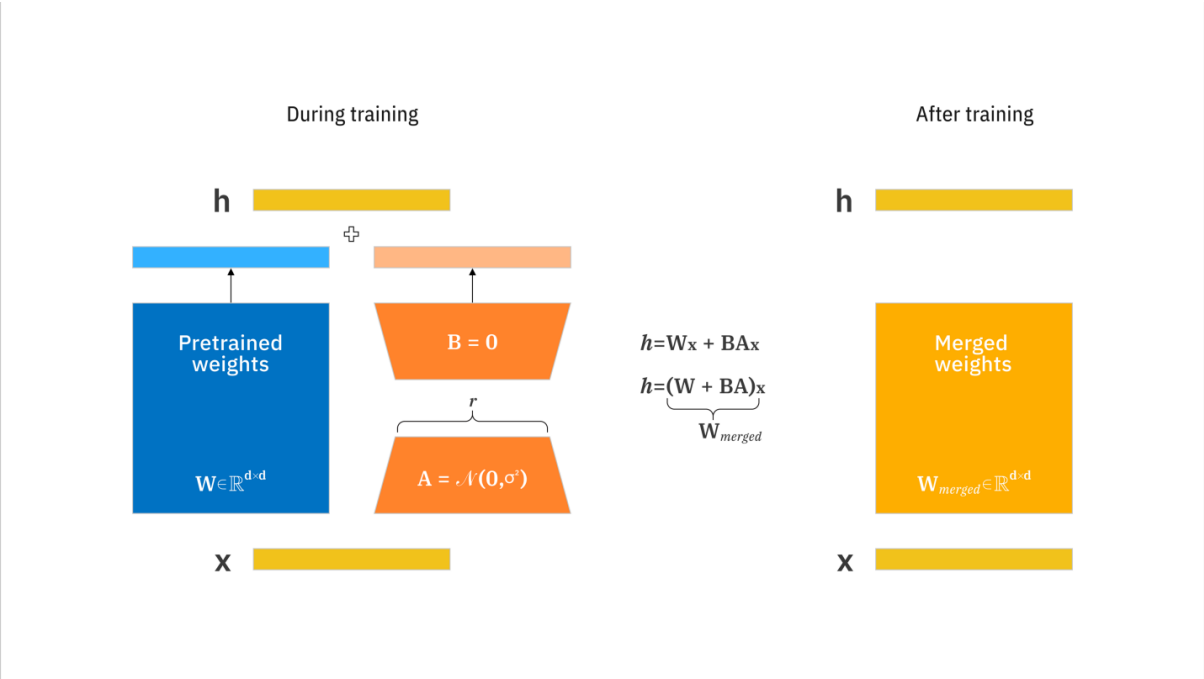
2)LoRA (Low-Rank Adaptation):

- LoRA leverages the concept of lower-rank matrices to make the model training process extremely efficient and fast.
- Large models have a lot of parameters. For example, GPT-3 has 175 billion parameters. These parameters are just numbers stored in matrices. Storing them requires a lot of storage.
- Full fine-tuning means all the parameters will be trained, and this will require an extraordinary amount of compute resources that can easily cost in the millions of dollars for a model size like GPT.

- Unlike traditional fine-tuning that requires adjusting the entire model, **LoRA focuses on modifying a smaller subset of parameters (lower-rank matrices)**, thereby reducing computational and memory overhead.
- LoRA is built on the understanding that large models inherently possess a low-dimensional structure. By leveraging low-rank matrices, LoRA adapts these models effectively. This method focuses on the core concept that significant model changes can be represented with fewer parameters, thus making the adaptation process more efficient.

Working :

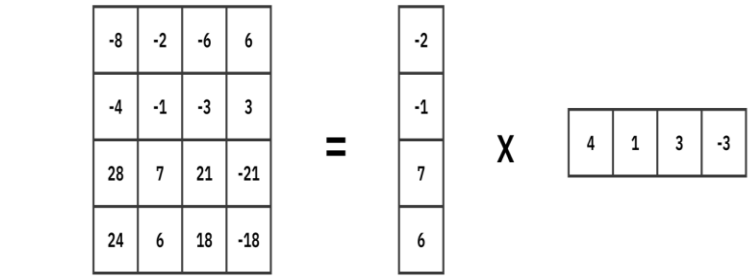
- In LoRA Instead of retraining the whole model, LoRA freezes the original weights and parameters of the model as they are. Then, on top of this original model, it adds a lightweight addition called a low-rank matrix, which is then applied to new inputs to get results specific to the context. The low-rank matrix adjusts for the weights of the original model so that outputs match the desired use case.
- LoRA leverages the concept of lower-rank matrices to make the model training process extremely efficient and fast. Traditionally fine-tuning LLMs requires adjusting the entire model. LoRA focuses on modifying a smaller subset of parameters (lower-rank matrices) to reduce computational and memory overhead.



The diagram shows how LoRA updates the matrices A and B to track changes in the pretrained weights by using smaller matrices of rank r. Once LoRA training is complete, smaller weights are merged into a new weight matrix, without needing to modify the original weight

the pretrained model.

- LoRA is built on the understanding that large models inherently possess a low-dimensional structure. By leveraging smaller matrices, which are called low-rank matrices, LoRA adapts these models effectively. This method focuses on the core concept that significant model changes can be represented with fewer parameters, thus making the adaptation process more efficient.
- In LoRA A high-rank matrix can be decomposed into two low-rank matrices, a 4 x 4 matrix can be decomposed into a 4 x 1 and a 1 x 4 matrix.



- LoRA adds low-rank matrices to the frozen original machine learning model. The low-rank matrices are updated through gradient descent during fine-tuning, without modifying the weights of the base model. These matrices contain new weights to apply to the model when generating results. The multiplied change matrix is added to the base model weights to get the final fine-tuned model. This process alters the outputs that the model produces with minimal computing power and training time.
- To implement LoRA fine tuning with HuggingFace using Python and PyTorch, developers can use the parameter-efficient fine-tuning (PEFT) library to inject the LoRA adapters into the model and use them as the update matrices. This library is

freely available through HuggingFace or GitHub.

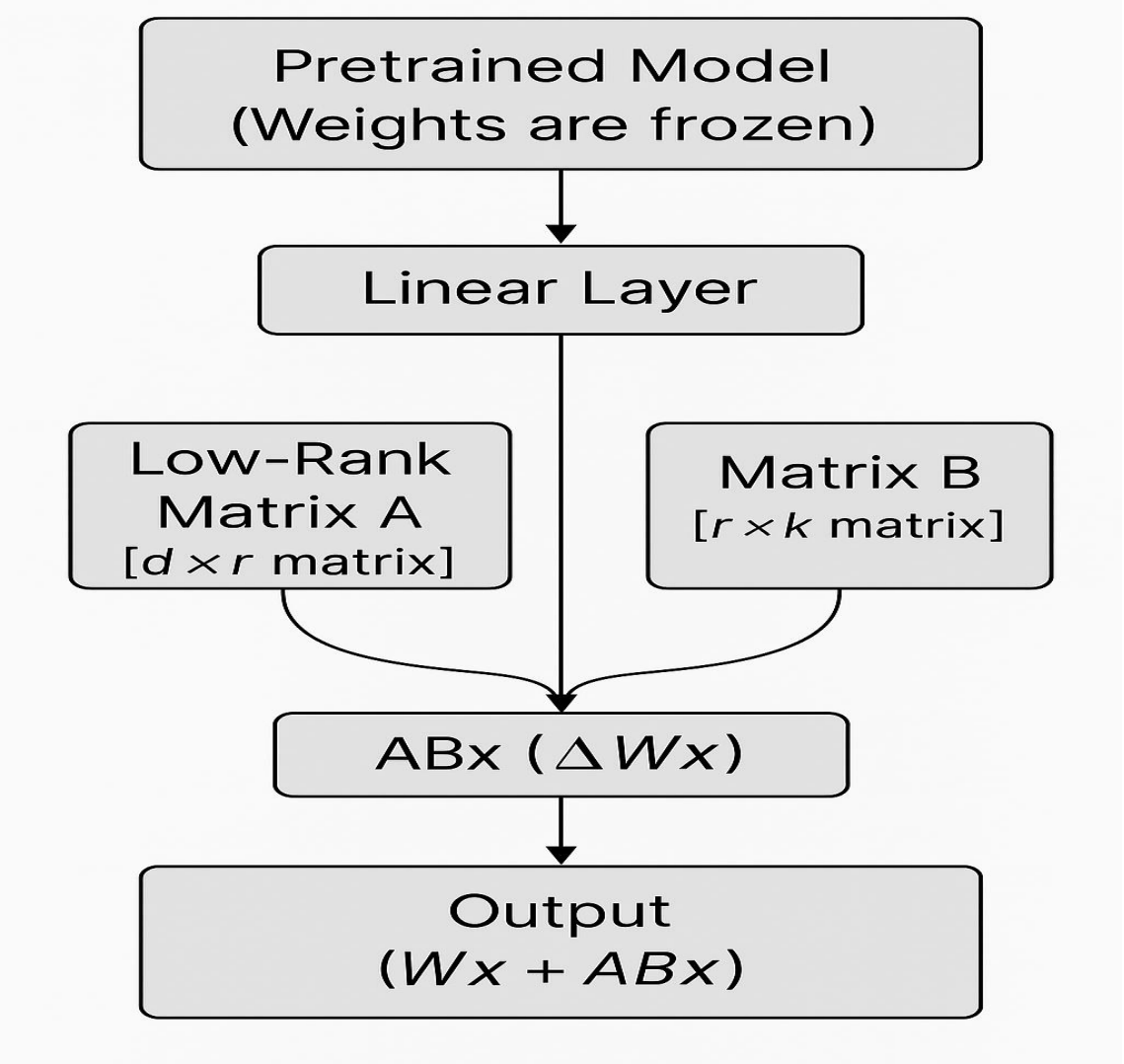
This library provides ways to configure LoRA parameters for your model. Some of the most commonly used parameters are:

- **r**: the rank of the update matrices, expressed in int. Lower rank decomposition results in smaller update matrices with fewer trainable parameters.
- **target_modules**: The modules (for example, attention blocks) to apply the LoRA update matrices.
- **lora_alpha**: LoRA scaling factor.

Steps Training WorkFlow Of LoRA

1. Load a pre-trained model (e.g., LLaMA, GPT).
2. Freeze all parameters of the model.
3. Insert LoRA modules (pairs of trainable low-rank matrices) into target layers (like query/key/value or feed-forward).
4. Train only LoRA parameters on downstream tasks.
5. During inference: compute $Wx+A(Bx)$ instead of modifying W

LoRA Architecture Diagram



Advantages of LoRA

- **Parameter-efficient**: Only a small subset of parameters are trained.
- **Memory-efficient**: Avoids full model updates.
- **Modular**: Can apply to attention, MLP, or any linear layers.
- **Reusable Base**: Same base model can be adapted to multiple tasks by swapping LoRA modules.

3) QLoRA :

- LoRA is a advanced fine-tuning method that quantizes LLMs to reduce memory usage and applies **Low-Rank Adaptation (LoRA)** to train a subset of model parameters.
- QLoRA is the extended version of LoRA which works by quantizing the precision of the weight parameters in the pre trained LLM to 4-bit precision. Typically, parameters of trained models are stored in a 32-bit format, but QLoRA compresses them to a 4-bit format.
- This reduces the memory footprint of the LLM, making it possible to finetune it on a single GPU. This method significantly reduces the memory footprint, making it feasible to run LLM models on less powerful hardware, including consumer GPUs.

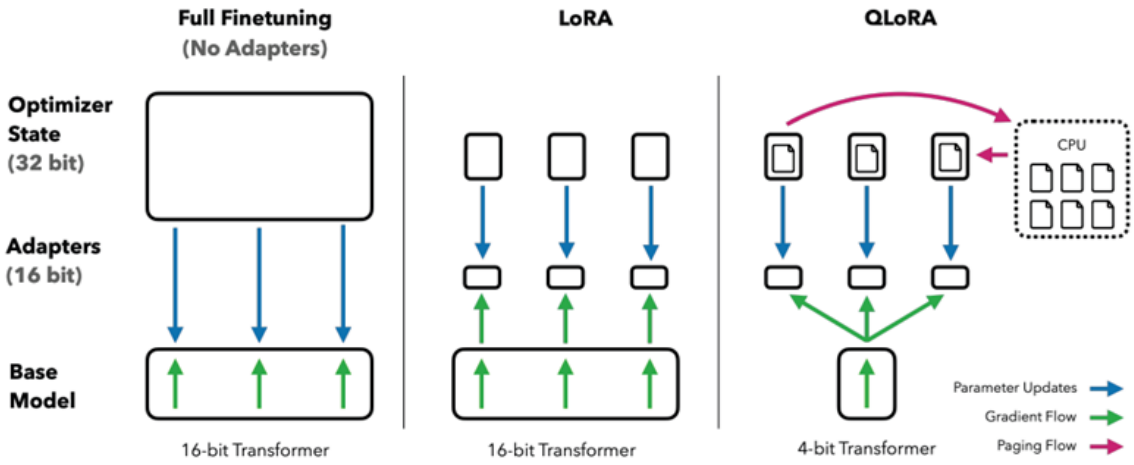
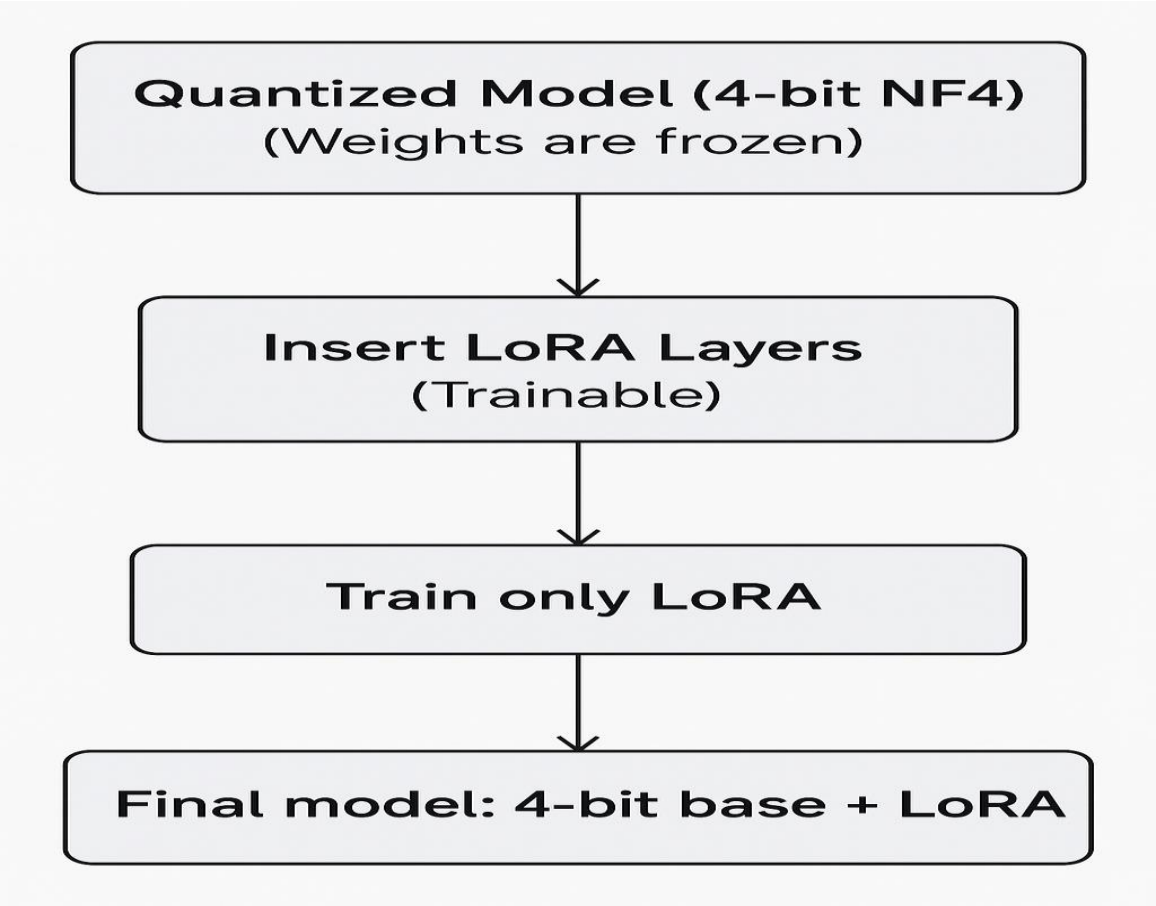


Figure 1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

(Diagram of QLoRA Working)

Working Process of QLoRA :

- **Quantize** the pre-trained model to 4-bit precision using **NF4 (NormalFloat4)**, which preserves distributional properties.
- **Freeze** the quantized model.
- **Insert LoRA modules** into the quantized model (same as before).
- **Train LoRA adapters** (in 16-bit or 32-bit) while keeping the quantized weights fixed.
- Use **paged optimizers** (like paged AdamW) to manage memory efficiently.



Advantages of QLoRA :

- Runs on consumer GPUs (e.g., 24 GB VRAM).
- Minimal performance drop (within 1% of full fine-tuning).
- Combines quantization + parameter-efficient fine-tuning.
- Enables training very large models on single GPUs.

LoRA vs QLoRA – Comparison Table

Feature	LoRA	QLoRA
Base Model Type	Full-precision (FP16 or FP32)	Quantized to 4-bit (e.g., NF4)
Memory Requirement	Moderate (16–24 GB for 7B models)	Low (8–16 GB for 7B models)
Trainable Parameters	Very few (~0.5–2%)	Very few (~0.5–2%)
Training Speed	Faster (no quantization overhead)	Slightly slower (due to quantized math ops)
Accuracy	Near full fine-tuning	Near full fine-tuning

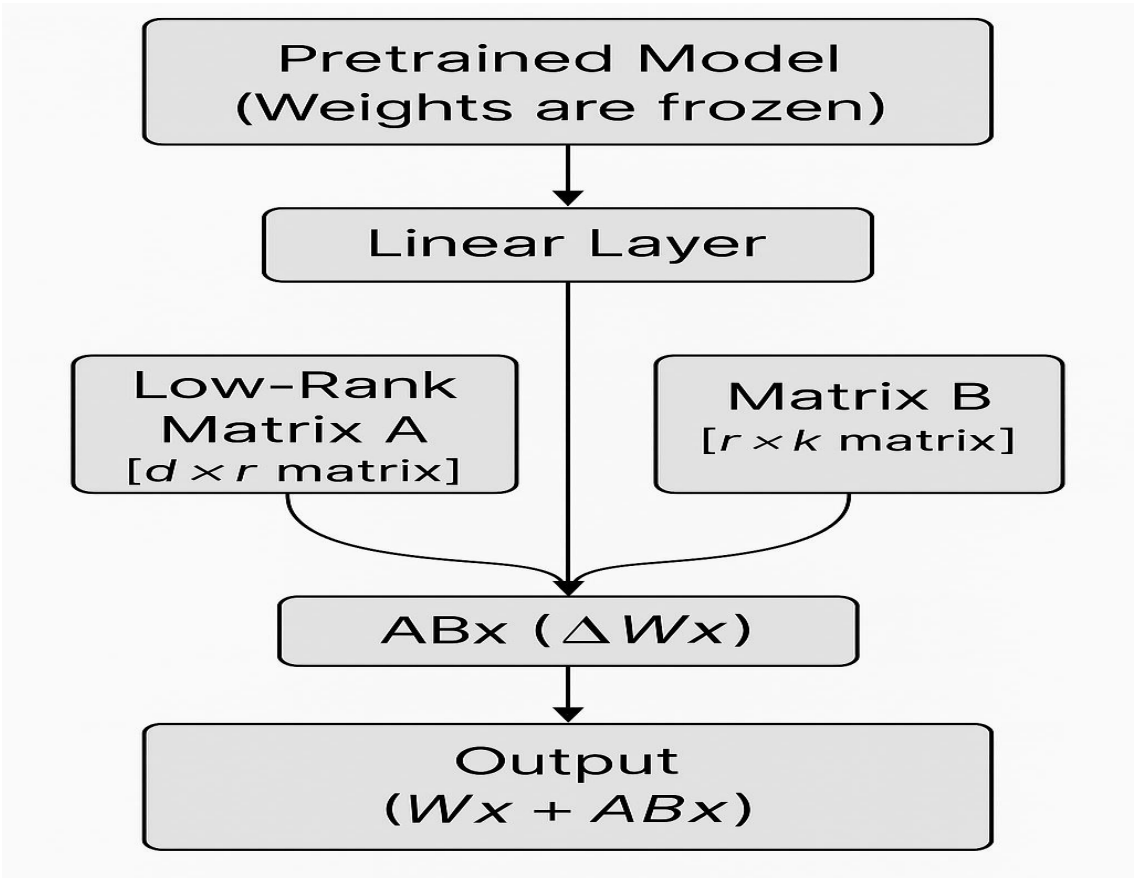
Why Use LoRA and QLoRA?

Training or fine-tuning large language models like GPT, LLaMA, or Falcon is **extremely resource-intensive**. Standard full fine-tuning requires:

- **Huge GPU memory (80–100+ GB)**.
- **High computational cost (multiple A100s)**.
- **Long training times**.
- **Storage overhead** for saving multiple fine-tuned model versions.
- LoRA and QLoRA were developed to **solve these limitations**.

Feature	LoRA	QLoRA
Base Model Type	Full-precision (FP16 or FP32)	Quantized to 4-bit (e.g., NF4)
Memory Requirement	Moderate (16–24 GB for 7B models)	Low (8–16 GB for 7B models)
Trainable Parameters	Very few (~0.5–2%)	Very few (~0.5–2%)
Training Speed	Faster (no quantization overhead)	Slightly slower (due to quantized math ops)
Storage Size	Adapter-only: ~100MB	Adapter-only: ~100MB

Feature	LoRA	QLoRA
Inference Setup	Requires FP16/32 model + adapter	4-bit quantized model + adapter
Use Case	When you have access to decent VRAM	When VRAM is highly limited (e.g., 1 GPU)
Accuracy	Near full fine-tuning	Near full fine-tuning
Deployment Flexibility	Good (modular adapters)	Excellent (small + quantized)
Tools Support	Hugging Face PEFT, 🤗 Transformers, etc.	bitsandbytes, Hugging Face, 🤗 Accelerate


Pretrained Model
(Weights are frozen)
Linear Layer
Low-Rank
Matrix A
[$d \times r$ matrix]
Matrix B
[$r \times k$ matrix]
ABx (ΔWx)
Output
($Wx + ABx$)