Fine-Tuning with LoRA and QLoRA

Introduction

Large Language Models (LLMs) like GPT, BERT, and LLaMA have billions of parameters. Fine-tuning them traditionally requires massive resources. LoRA (Low-Rank Adaptation) and QLoRA (Quantized LoRA) are two efficient techniques that make this process more feasible.

1.LoRA

LoRA doesn't update the full weight matrices of a pre-trained model. Instead, it adds small, trainable low-rank matrices (**A and B**) into specific parts (usually attention layers) and only trains those, saving time, memory, and compute.

What Are A and B Matrices?

- A: Projects input to a low-dimensional space (dim: d → r)
- B: Projects from low-dimensional space back to output space (dim: r → k)

Together, AB acts as a trainable low-rank approximation of the full update you'd make to W, but at a much lower cost.

How it Works

Let's say your model has a linear transformation

$$y=Wx$$

Where:

- $W \in R^{d\times k}$ is the original weight matrix.
- x is the input vector.

Instead of updating W directly, LoRA adds a **low-rank update**:

$$W'=W+\Delta W=W+AB$$

Where:

- $A \in R^{d\times r}$ A
- $B \in R^{r\times k}$ B
- r is the **rank** (usually a small number like 4, 8, or 16)

Only A and B are trainable. W is frozen (not updated during backpropagation).

LoRA Parameters

Parameter	Description
r (rank)	Controls the size of LoRA matrices A & B. Smaller r means fewer trainable params.
alpha	Scaling factor applied to LoRA updates. Helps balance the magnitude of updates.
dropout	Dropout applied before the LoRA output to regularize.
bias	Whether to tune bias terms in addition to LoRA weights. Common values: "none", "all".
target_modules	Which model modules to apply LoRA to (e.g., "q_proj", "v_proj" in transformers).
fan_in_fan_out	Adjusts initialization if layer uses fan-in or fan-out convention.
merge_weights	Whether to merge LoRA weights with base model weights after training.

Parameter Optimization Guide

1. **Rank (r) Selection**

- Start with r=8 for 7B models
- Increase to r=64 for >30B models
- Tradeoff: Higher rank + better performance e more parameters

2. **Alpha Scaling**

   - Default: a = 2r
   - Adjust ratio based on task complexity

3. **Quantization Settings**

   - Optimal: 4-bit NormalFloat (nf4)
   - Enable double quantization for 0.5GB memory saving

4. **Target Modules**

   - LLAMA: ["q_proj", "v_proj"]
   - GPT: ["c_aftn"]
   - BERT: ["query", "value"]

How LoRA Reduces Number of Trainable Parameters

1. **Freezing** all pre-trained weights.
2. **Adding low-rank matrices** A and B only at selected layers.
3. Training only these small matrices.

How to Freeze weights

```
form param in model.parameters():
    param.requires_grad = False
```

Using HuggingFace PEFT:

- When you apply LoRA with PEFT, it **automatically freezes** the base model and enables training only for LoRA layers.

Example: If original layer has 1 million parameters and LoRA injects rank-8 adapters (i.e., 8K total parameters per matrix), you may end up training <1% of the model.

Let's say a normal linear layer has:

- $W \in R^{4096 \times 4096}$ → ~16.7M parameters

LoRA with r=8:

- $A \in R^{4096 \times 8}$ → ~32.8K
- $B \in R^{8 \times 4096}$ → ~32.8K
- Total: ~65.5K trainable parameters**, i.e.,** ~0.39% of original size

Implementation

```
from peft import LoraConfig, get_peft_model, TaskType
lora_config = LoraConfig(
    r=8,
    lora_alpha=16,
    target_modules=["q_proj", "v_proj"],
    lora_dropout=0.05,
    bias="none",
    task_type=TaskType.CAUSAL_LM
)
peft_model = get_peft_model(model, lora_config)
```

LoRA Workflow

1. Load pre-trained model (transformers)
2. Freeze all model weights
3. Configure LoRA (peft. Loraconfig)
4. Inject LoRA layers at target modules (e.g., attention)
5. Train only the LoRA adapters
6. Save adapters or merge with base model (optional)

2.QLoRA

QLoRA enables efficient fine-tuning of large models on limited hardware by combining **4-bit quantization, LoRA adapters,** and **paged optimizers.**

How it works

1. 4-Bit Quantization (NF4)

- **Quantization** reduces model weights from float32 or float16 to **4-bit precision**.
- QLoRA uses **NF4 (NormalFloat4)**, a new 4-bit data type designed for LLMs.

Why NF4

- Unlike older 4-bit formats (e.g., INT4), **NF4 approximates a normal distribution**, matching LLM weight distributions.
- It captures more variance and avoids precision loss in critical ranges.

Implementation

```
load_in_4bit=True,
quantization_config={
  "bnb_4bit_quant_type": "nf4",                # or "fp4"

  "bnb_4bit_compute_dtype": torch.float16,     # or bfloat16
  "bnb_4bit_use_double_quant": True,           # adds extra quantization for stability

}
```

Benefits : A model like **LLaMA 7B** can go from ~28GB (float16) → **~4.5GB** (4-bit NF4)

2. LoRA Adapters

- Trainable **low-rank matrices** $A \in R^{d \times r}$, $B \in R^{r \times k}$ added to key model layers.
- Only these matrices are trained — **base model stays frozen**.
- Typically injected into **transformer attention modules**,(e.g., q_proj, v_proj, k_proj, o_proj)

LoRA Updated Equation

$$W' = W + \Delta W = W + AB$$

- W: Frozen base weight
- A,B: Small trainable matrices
- r: Rank (e.g., 4, 8, 16)

Implementation

```
from peft import LoraConfig

LoraConfig(
   r=8,
   lora_alpha=16,
   target_modules=["q_proj", "v_proj"],
   lora_dropout=0.05,
   bias="none"
)
```

Benefits :

- **Tiny memory footprint** (e.g., just a few MBs per layer)
- Preserves original model knowledge
- Trains fast even on a laptop GPU

3. Paged Optimizers

- **Memory-efficient optimizers** that **offload model data between GPU, CPU, and disk** as needed.
- Allow large-scale training with **low active VRAM**.

Key Optimizer: paged_adamw_32bits

- Included in **bitsandbytes**
- Keeps most data on **CPU or disk**, brings to GPU **only when needed**
- Uses **32-bit optimizer states** for stability, despite 4-bit weights

Implementation

```
from bitsandbytes.optim import PagedAdamW32bit

optimizer = PagedAdamW32bit(
   model.parameters(),
   lr=2e-5
)
```

Benifts:

- Enables **training 33B+ models** on **24GB VRAM GPUs** (e.g., RTX 3090, T4)

Summary

- **QLoRA** is ideal for **fine-tuning massive LLMs on small GPUs**.
- It uses **4-bit quantization + LoRA adapters**.
- Freezes the base model, trains only adapters → fast and cheap training.
- Uses **bitsandbytes**, **peft**, **transformers**, and optionally **trl**.

Combined Effect

Feature	Standard FT	QLoRA
Weight Precision	float32 / float16	4-bit NF4
Memory Usage	~100–300 GB	5–24 GB
Trainable Params	All	LoRA adapters only
Optimizer	AdamW	PagedAdamW32bit (offloading)
Hardware Need	A100/3090+	RTX 2060 / T4 / consumer GPU

Technical Comparison: LoRA vs QLoRA

Feature	LoRA	QLoRA
Precision	16-bit base model	4-bit quantized base
Memory Usage	Higher (16-bit storage)	60-70% lower
Hardware Requirements	24GB+ VRAM	8-16GB VRAM
Training Speed	Faster than full FT	Slightly slower than LoRA
Model Performance	95-98% of full FT	97-99% of LoRA
Parameter Efficiency	0.1-2% of total params	Same as LoRA

Key differences between LoRA and QLoRA

Feature	LoRA	QLoRA
Parameter count	Reduced parameters	Reduced parameters with quantization
Precision	Full precision	4-bit precision
Memory usage	Low	Very low
Performance impact	Minimal	Slightly more efficient

When should you use LoRA or QLoRA?

- **LoRA** is ideal for fine-tuning models where memory is a constraint, but you still want to maintain high precision in terms of the final model.
- **QLoRA** is perfect for scenarios where extreme memory efficiency is required, and you can sacrifice a little precision without significantly impacting performance of the model.

Real-World Applications of LoRA and QLoRA

1. **Chatbot Fine-Tuning**
   LoRA/QLoRA can be used to fine-tune large LLMs for building conversational agents tailored to specific domains such as customer support, HR, finance, and e-commerce, where quick iteration and low-resource deployment are critical.
2. **Domain-Specific Medical/Legal LLMs**
   LoRA enables efficient adaptation of general-purpose models to specialized fields like medicine or law by training on domain-specific corpora. This allows high-quality results with minimal computational overhead.
3. **Multi-Modal Models (e.g., Vision Transformers)**
   In computer vision, LoRA has been applied to models like ViT (Vision Transformers) for tasks involving images and text. It supports quick fine-tuning in applications such as medical imaging diagnostics, satellite image classification, or OCR.

Evaluation Metrics

To measure the effectiveness of fine-tuned models, especially when using LoRA or QLoRA, consider:

- **Perplexity**: Measures how well a language model predicts the next token. Lower is better.
- **Accuracy / F1-Score**: For classification tasks (e.g., sentiment analysis, entity recognition).
- **BLEU / ROUGE / METEOR**: Common for summarization and translation tasks.
- **Inference Time / Latency**: Important for real-time applications.

- **Memory Utilization**: Check improvements in GPU/CPU usage with QLoRA.

Limitations of LoRA and QLoRA

Despite their advantages, these methods also come with trade-offs:

- **Limited Layer Flexibility**
  LoRA adapters are injected into specific modules like attention projections (q_proj, v_proj). Some architectures may require deeper customization or adapter tuning strategies.
- **Quantization Accuracy Trade-Off (QLoRA)**
  While QLoRA enables 4-bit precision for low memory use, this may lead to a **small drop in performance**, particularly in tasks sensitive to numerical precision (e.g., mathematical reasoning, multi-hop QA).
- **LoRA Adapter Merging (Optional)**
  In deployment, merged weights may not always match performance during adapter-based inference unless quantization-aware training is done.

Conclusion

LoRA and **QLoRA** provide resource-efficient alternatives to full-parameter fine-tuning. LoRA focuses on reducing the number of parameters that need updating, while QLoRA takes it further with quantization, making it the most memory-efficient option. Whether you're working with large LLMs for specific tasks or looking to optimize your model fine-tuning process, LoRA and QLoRA offer powerful solutions that save both time and resources.