AI & LLMS

Fine-Tuning and Adapting Open LLMs

LoRA, quantization and instruction tuning on your own data

Houssam Kodad

PDF · DATAFORGE BOOKS

Questions about this sample or the full edition: **support@dataforgebooks.com**

CONTENTS

Table of Contents

CHAPTER 01

When Fine-Tuning Is the Right Tool

When prompting hits its limits, fine-tuning an open model on your own data is the natural next move — and also a reliable way to burn a week and a GPU budget on a model no better than where you started. Fine-tuning is a precision tool, not a default, and knowing when to reach for it is fully half of the skill. This book aims to give you that judgement before it teaches you the mechanics.

This chapter sets expectations honestly, because the field is awash in hype that leads teams to fine-tune when they should retrieve, or to expect tuning to fix problems it cannot. We compare prompting, retrieval and fine-tuning as solutions to genuinely different problems, separate what tuning can fix from what it cannot, and give a sober reckoning of the cost and effort involved — including the parts nobody advertises.

The good news, developed over the following chapters, is that parameter-efficient techniques have made adaptation dramatically cheaper than it was: you can meaningfully adapt a capable open model on a single modest GPU. The sobering news, which we are clear about here, is that the compute was never the main cost. The dataset and the evaluation are, and they are where projects succeed or fail.

Prompting, RAG, or Fine-Tuning

These three approaches address different problems, and confusing them is the most expensive mistake in the field. Prompting changes behaviour through instructions and examples in the context — cheap, instant, but bounded by the context window and by what the base model already knows. Retrieval injects knowledge the model lacks at query time. Fine-tuning bakes behaviour and style into the model's weights themselves.

The decisive question is what you are actually missing. If the model lacks facts, you almost always want retrieval, not fine-tuning — tuning is a poor and costly way to teach facts, and it dates instantly as those facts change. If the model lacks a consistent format, tone, or task-specific behaviour, fine-tuning shines. Reaching for the wrong tool wastes weeks, and we spend real effort on this decision because getting it right saves more than any technique.

What Tuning Can and Cannot Fix

Fine-tuning is excellent at shaping behaviour: enforcing a rigid output format, adopting a domain's tone and conventions, reliably performing a narrow task, or following instructions that are awkward

to specify in a prompt. Given good examples, it makes these behaviours the model's reliable default rather than something you must coax with ever-longer prompts.

It is poor at, and sometimes actively harmful for, injecting fresh factual knowledge, keeping information current, or fixing problems that are really about retrieval. Worse, careless fine-tuning can erode the base model's general capabilities, trading broad competence for narrow skill — a regression that is easy to cause and easy to miss without careful evaluation. Matching the technique to the problem is what keeps fine-tuning a net benefit rather than a step backward.

The Real Cost: Data, Not GPUs

The popular image of fine-tuning is GPU hours, but the dominant cost is almost always building a clean, well-labelled dataset — the unglamorous, decisive work that determines whether the result is any good. A few thousand high-quality, consistent examples beat a hundred thousand noisy ones, and assembling them is where most of the genuine effort and most of the eventual success lives.

Add to the dataset the cost of hyperparameter iteration, rigorous evaluation, and the serving infrastructure for the result, and the true effort dwarfs the training run itself. Going in clear-eyed about where the work actually lies — overwhelmingly in data and evaluation, not in the training loop — is what keeps a fine-tuning project on the rails instead of stalled after an expensive, disappointing first attempt.

Parameter-Efficient Fine-Tuning

Fully fine-tuning a large model updates billions of parameters and demands enormous memory, which put adaptation out of reach for most teams. Parameter-efficient methods, above all LoRA, changed that. Instead of updating the whole model, LoRA freezes the original weights and trains small adapter matrices alongside them, capturing the adaptation in a tiny fraction of the parameters.

The consequences are practical and large. Adapters are small enough to store and swap cheaply, so you can maintain many task-specific adaptations of one base model. Training touches far less memory, bringing serious fine-tuning within reach of a single GPU. Understanding how LoRA works, and which layers to target, is the foundation for everything that follows, and we build it carefully rather than treating it as a magic flag.

```
from peft import LoraConfig, get_peft_model

config = LoraConfig(
    r=16, lora_alpha=32, lora_dropout=0.05,
    target_modules=["q_proj", "v_proj"],   # adapt attention projections
)
model = get_peft_model(base_model, config)
model.print_trainable_parameters()
# trainable: ~0.1% of total — the base weights stay frozen.
```

Quantization for Modest Hardware

Quantization shrinks a model by storing its weights at lower numerical precision — eight or even four bits instead of sixteen — dramatically reducing the memory it occupies with surprisingly small effects on quality. Combined with LoRA, in the technique known as QLoRA, it lets you fine-tune models that would otherwise never fit, on hardware you can actually access.

This combination is what democratised fine-tuning: a model that once required a cluster can be adapted on a single consumer or cloud GPU. The trade-offs — a small quality cost, slower training in some configurations — are real but usually well worth it. We work through quantized fine-tuning end to end, because it is the path most readers will actually take, and the details matter for getting a usable result.

Evaluating a Tuned Model

The most important and most skipped step is honest evaluation. A fine-tuned model must be measured on two axes: did it get better at the target task, and did it get worse at everything else? The second question is the one teams forget, and it is where careless tuning does its quiet damage, trading away general capability for a narrow gain nobody checked against the base model.

Trustworthy evaluation means a task-specific benchmark to confirm the improvement and a broader regression check against the base model to catch capability loss, ideally with human judgement where the task is subjective. Without this, you cannot know whether your fine-tune helped or hurt, and "it seems better" is not a foundation to ship on. We make rigorous before-and-after evaluation the non-negotiable final step of every adaptation.

Serving and Maintaining Adapters

A fine-tuned model is only useful once it serves efficiently, and adapters open up options the full-fine-tuning world lacks. You can merge an adapter into the base weights for a standalone model,

or — more powerfully — serve many adapters on top of a single shared base, switching between them per request to support many tasks from one deployment at a fraction of the memory.

Maintenance is the part the tutorials skip. Models need re-tuning as data and requirements evolve, adapters and datasets need versioning so you can reproduce and roll back, and the licences of open base models impose real obligations you must respect. The chapters ahead build from the foundations here toward a complete, sustainable practice — not a one-off experiment, but a repeatable way to adapt open models and keep them healthy in production.

This is a free sample

You've reached the end of the sample chapter.

Get the complete book — every chapter, fully worked — at **dataforgebooks.com**.

FULL EDITION · 216 PAGES · PDF

Read the full title at **dataforgebooks.com**

Questions? support@dataforgebooks.com