



AI
Bees

Generative AI

On-Premise LLMs & Fine-Tuning

On-Premise Models • RAG vs Fine-Tuning • PEFT / LoRA / QLoRA • Ollama

Student Notes — Theory, Concepts & Practical Setup

1. What Are On-Premise LLMs?

1.1 Definition





An on-premise LLM (also called a local LLM) is a large language model that runs entirely on your own hardware — your laptop, workstation, or company servers — rather than being accessed via a third-party cloud API such as OpenAI, Google Gemini, or Anthropic Claude.

Instead of sending your prompt over the internet to a remote server, you download the model weights once and run inference locally. This fundamentally changes the privacy, cost, and control profile of your AI system.

💡 Key Point: Think of it this way: cloud APIs are like renting a calculator from someone else's building — fast, convenient, but your data leaves your hands. An on-premise LLM is like owning the calculator — it stays in your building forever.

1.2 Why Use On-Premise LLMs?

The main utility of using on-premise LLMs stems from solving the common challenges associated with cloud-based AI APIs. There are four major categories:

Benefit Category	What It Means in Practice
 Data Privacy & Security	Your prompts, documents, and responses never leave your network. Critical for healthcare (HIPAA), finance (PCI-DSS), legal, and government sectors.
 Cost Savings	Zero per-token API fees after the initial download. High-volume usage (millions of queries) becomes dramatically cheaper versus pay-per-use cloud APIs.
 Full Control	You choose the exact model version, fine-tune it for your domain, set the system prompt permanently, and are not affected by provider outages or policy changes.
 No Internet Required	Works in air-gapped environments (classified facilities, remote locations, offline edge devices). No dependency on API availability or latency.

1.3 Enhanced Data Privacy & Security — Deep Dive

Data Sovereignty: The most crucial use, especially for businesses, is keeping sensitive data local. When you use an on-premise LLM, your prompts and proprietary data never leave your internal network or device, ensuring compliance with strict privacy regulations like HIPAA or GDPR.

Confidentiality: It allows you to process confidential documents, patient records, financial data, or trade secrets with an LLM without the risk of exposure inherent in sending that data to a third-party cloud provider.

⚠ Warning: Even "anonymous" data sent to cloud APIs can sometimes be re-identified. For truly sensitive workloads — patient records, legal documents, source code with trade secrets — on-premise is not optional, it is a compliance requirement.

1.4 Cost Savings & Predictable Billing

Zero API Costs: Once the model is downloaded, there are no recurring, usage-based API fees (per-token costs). This makes high-volume or experimental usage significantly cheaper over time.

Metric	Cloud API	On-Premise
Cost model	Pay per token (e.g. \$0.002/1K tokens)	One-time hardware cost; near-zero marginal cost
High-volume usage	Expensive — costs scale linearly	Very cheap — hardware already paid for
Predictability	Variable; hard to budget	Fixed hardware depreciation
Experimentation	Each failed attempt costs money	Free to iterate as many times as needed

1.5 Trade-offs — When NOT to Use On-Premise

On-premise LLMs are not always the right choice. Understanding the trade-offs is important:

- **Hardware cost:** Running large models (7B+ parameters) requires significant GPU VRAM. Consumer GPUs (8–16GB) can run quantized models; full-precision models need A100/H100 class hardware.
- **Setup complexity:** Requires installing runtime software, managing model files, and handling updates manually — more DevOps work than calling an API.
- **Model quality gap:** Open-source models (Llama, Mistral, Gemma) are very capable but for some tasks still lag behind the largest proprietary models (GPT-4o, Claude Sonnet).
- **No automatic updates:** Cloud APIs silently improve their models. With on-premise, you choose when to upgrade.

✓ Key Insight: Rule of thumb: Use cloud APIs for prototyping and low-volume production. Switch to on-premise when you have sensitive data, high query volume, or need offline capability.

2. Ollama — Running LLMs Locally

2.1 What is Ollama?

Ollama is an open-source tool that makes running large language models on your local machine as easy as installing any other software. It handles downloading model weights, loading them into memory, and exposing a simple REST API — all with a single command-line tool.

Ollama supports Windows, macOS (including Apple Silicon), and Linux. On Apple Silicon (M1/M2/M3), it uses Metal GPU acceleration for fast inference without a dedicated GPU.

Property	Details
Website	https://ollama.com
Model Library	https://ollama.com/library (100+ models: Llama, Mistral, Gemma, Phi, Qwen, DeepSeek, etc.)
License	MIT open-source
API format	OpenAI-compatible REST API on localhost:11434 — drop-in replacement for OpenAI SDK calls
Min. hardware	8GB RAM for 3B–7B models; 16GB+ for 13B; 64GB+ for 70B

2.2 Installation

Terminal — Install Ollama

```
# macOS / Linux (one command):  
curl -fsSL https://ollama.com/install.sh | sh  
  
# Windows: download installer from https://ollama.com/download/windows  
# macOS: download .app from https://ollama.com/download/mac
```

2.3 Pulling & Running Models

Terminal — Download and run a model

```
# Pull (download) a model — stored in ~/.ollama/models  
ollama pull llama3.2           # Meta Llama 3.2 3B (2.0 GB)  
ollama pull llama3.1           # Meta Llama 3.1 8B (4.7 GB)  
ollama pull mistral             # Mistral 7B (4.1 GB)  
ollama pull gemma3:4b           # Google Gemma 3 4B (3.3 GB)  
ollama pull phi4                # Microsoft Phi-4 14B (9.1 GB)  
ollama pull deepseek-r1:7b      # DeepSeek R1 7B reasoning model  
ollama pull qwen2.5:7b          # Alibaba Qwen 2.5 7B  
  
# Start an interactive chat in terminal  
ollama run llama3.2  
  
# List downloaded models  
ollama list  
  
# Remove a model  
ollama rm mistral
```

2.4 Using Ollama via Python (LangChain)

Python — Ollama with LangChain

```
# Install
pip install langchain-ollama

from langchain_ollama import OllamaLLM
from langchain_ollama import ChatOllama

# Simple LLM call
llm = OllamaLLM(model="llama3.2")
response = llm.invoke("Explain transformers in simple terms")
print(response)

# Chat model (with message history)
chat = ChatOllama(model="mistral", temperature=0.7)
from langchain_core.messages import HumanMessage, SystemMessage
messages = [
    SystemMessage(content="You are a helpful AI tutor."),
    HumanMessage(content="What is RAG?")
]
response = chat.invoke(messages)
print(response.content)
```

2.5 Using the Ollama REST API Directly

Python — Direct REST API (OpenAI-compatible)

```
import requests


# Chat completion (OpenAI-compatible endpoint)
response = requests.post(
    "http://localhost:11434/v1/chat/completions",
    json={
        "model": "llama3.2",
        "messages": [{"role": "user", "content": "Hello!"}]
    }
)
print(response.json()["choices"][0]["message"]["content"])

# You can also use the OpenAI SDK with Ollama:
from openai import OpenAI
client = OpenAI(base_url="http://localhost:11434/v1", api_key="ollama")
resp = client.chat.completions.create(
    model="llama3.2",
    messages=[{"role": "user", "content": "Summarize this..."}]
)
```

2.6 Popular Models & Where to Find Them

Model	Size (approx.)	Best For	Ollama Pull Command
Llama 3.2	2 GB (3B)	General chat, RAG	<code>ollama pull llama3.2</code>
Llama 3.1	4.7 GB (8B)	Reasoning, coding	<code>ollama pull llama3.1</code>
Mistral 7B	4.1 GB	Fast, general	<code>ollama pull mistral</code>
Gemma 3 4B	3.3 GB	Efficient, multilingual	<code>ollama pull gemma3:4b</code>
Phi-4 14B	9.1 GB	Small but very capable	<code>ollama pull phi4</code>
DeepSeek-R1	4.7 GB (7B)	Step-by-step reasoning	<code>ollama pull deepseek-r1:7b</code>
Qwen 2.5	4.7 GB (7B)	Code generation	<code>ollama pull qwen2.5:7b</code>

Additional model sources: Hugging Face (<https://huggingface.co/models>) hosts thousands of models in GGUF format compatible with Ollama. You can import any GGUF model using a Modelfile.

 **Remember:** Ollama stores models in `~/.ollama/models` on macOS/Linux and `C:\Users\<name>\.ollama\models` on Windows. Make sure you have enough disk space before pulling large models.

3. RAG vs Fine-Tuning vs Re-training — What's the Difference?

3.1 Why This Question Matters


When you want to make an LLM useful for a specific domain (e.g., your company's documents, medical data, customer support), you have three main strategies. Choosing the wrong one wastes significant time, money, and compute. Understanding the differences is fundamental to building production AI systems.

RAG vs Fine-Tuning vs Re-training — Full Comparison

Feature	Retrieval-Augmented Generation (RAG)	Fine-Tuning	Re-training (From Scratch)
Goal	Provide Current/Proprietary Knowledge and reduce hallucinations.	Teach New Skills/Style or improve performance on a specific task.	Create a Fundamentally New Model or drastically update core knowledge.
Model Change	None. The LLM weights/parameters are static.	Partial. The LLM weights/parameters are updated using new data.	Complete. The LLM weights/parameters are trained from scratch.
Knowledge Storage	External. Information is stored in a separate, searchable database (e.g., Vector Database).	Internal. Knowledge is encoded directly into the model's parameters.	Internal. Knowledge is encoded directly into the model's parameters.
Data Requirements	Unstructured data (documents, PDFs) for the knowledge base.	Small-to-medium set of high-quality, labeled examples.	Massive and diverse dataset (terabytes of text/code).
Cost & Time	Low-to-Moderate. Fast to deploy. Ongoing cost for vector DB and inference.	Moderate-to-High. Requires significant compute for training.	Extremely High. Requires massive compute and months of time.
Adaptability	Dynamic. Easily and quickly updated by changing the knowledge base.	Static. New information requires re-tuning the model.	Static. New information requires re-training the model.


3.2 RAG in One Sentence

RAG keeps the model frozen and teaches it to look things up — you add a searchable knowledge base next to the LLM and inject relevant chunks into the prompt at query time. The model's weights never change; only its context window gets richer.

 **Key Point:** Use RAG when: your knowledge changes frequently (news, company docs, product catalogues), you want to cite sources, or you need to add knowledge without the expense of training.

3.3 Fine-Tuning in One Sentence

Fine-tuning updates a subset of the model's weights using your task-specific labelled data, teaching the model new skills, a new tone, or a specific output format — without changing the model's general language understanding.

 **Key Point:** Use fine-tuning when: you need a very specific output style (structured JSON, medical terminology), you want the model to follow a new domain's vocabulary, or you need improved accuracy on a narrow task.

3.4 Re-training in One Sentence

Re-training means building a brand-new model from randomly initialised weights using a massive dataset — this is what companies like Meta (Llama), Google (Gemma), and Mistral AI do. It requires hundreds of GPUs running for weeks and costs millions of dollars.

⚠ Warning: Re-training from scratch is almost never the right choice for an individual developer or small company. It is reserved for organisations with massive compute budgets and billions of training tokens.

3.5 Decision Guide — Which to Use?

Your Situation	Best Approach
You have PDFs/docs you want the LLM to answer questions about	☑ RAG
You want the LLM to always respond in a specific JSON format	☑ Fine-Tuning
Your data changes every week	☑ RAG (update the vector DB, not the model)
You want a customer support bot with your brand voice	☑ Fine-Tuning (+ RAG for product docs)
You want to build a general-purpose assistant from scratch	✗ Re-training (unless you're a well-funded AI lab)
You want to reduce hallucinations using company documents	☑ RAG is the primary solution

4. Methods in LLM Fine-Tuning

4.1 Overview

Full fine-tuning — updating every single weight in the model — is computationally expensive and prone to "catastrophic forgetting" (where the model forgets its general language abilities). The field has developed several Parameter Efficient Fine-Tuning (PEFT) techniques that update only a small fraction of the model's parameters, achieving comparable results at a fraction of the compute cost.

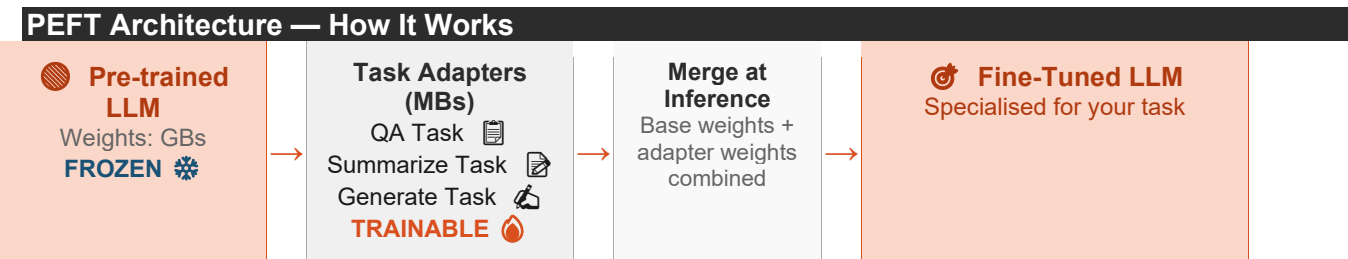
Method	Purpose	Key Benefit	Best For
LoRA	Efficient fine-tuning	Reduces memory usage drastically	Industry-specific LLMs, domain adaptation
QLoRA	Quantized fine-tuning	Enables 4-bit LLM tuning on consumer GPUs	Lower-end GPUs, individual researchers
SFT	Supervised fine-tuning	Trains on labelled input/output pairs	Domain adaptation, instruction following
PEFT	Modular fine-tuning framework	Combines multiple tuning methods (LoRA, Prefix, etc.)	Flexible LLM tuning across many scenarios
OPD	Optimized fine-tuning	Dynamically selects which layers are trainable	Next-generation LLM fine-tuning pipelines

5. Parameter Efficient Fine-Tuning (PEFT)

5.1 The Core Idea

PEFT is a family of techniques that fine-tune LLMs by modifying only a tiny fraction of the model's parameters. The vast majority of the original weights remain frozen (unchanged). Only small, newly inserted or selected parameters are trained.

- Freeze **all original weights** — the pre-trained model's billions of parameters are locked in place.
- Insert small **trainable modules** (adapters) into the architecture — these are the only things that change during training.
- **During training** → **only adapters learn**, base model untouched.
- Result: saves enormous cost, time, and GPU memory — you only store the adapter weights, not a full model copy per task.



Parameter Efficiency	Memory Efficiency	Model Performance	Training Speed	Inference Cost
PEFT key trade-off dimensions — all significantly better than full fine-tuning				

5.2 Why PEFT Matters

Full Fine-Tuning	PEFT (e.g. LoRA)
Update all ~7 billion parameters	Update only ~0.1–1% of parameters
Requires 40–80GB VRAM (for 7B model)	Can run on 8–16GB consumer GPU
Creates full model copy per task (GBs each)	Adapter is ~10–100MB; share the base model
Risk of catastrophic forgetting	Base model preserved; generalisation maintained

6. LoRA — Low-Rank Adaptation

6.1 What is LoRA?

LoRA (Low-Rank Adaptation) freezes the vast majority of a large model's weights **and injects small, trainable low-rank adapter matrices** into specific layers to efficiently learn new tasks with minimal computational cost.

6.2 How LoRA Works — The Math Made Simple

In a normal neural network layer, there is a weight matrix W . During full fine-tuning, you modify W directly. LoRA takes a different approach:

- **Step 1:** Freeze the original weight matrix W — do not touch it.
- **Step 2:** Decompose the weight update ΔW into two **small matrices** A and B , where:
 - $\Delta W = A \times B$ (this is the low-rank decomposition)
 - where A has dimensions $d \times r$ and B has dimensions $r \times k$ (r is the rank, typically 4–16)
- **Step 3:** Train only A & B , not the full W .
- **Step 4:** Final weight at inference = Original $W + \Delta W$ (they are merged back together)

6.3 LoRA Intuition — Why It Works

The key insight is that weight updates during fine-tuning have a low "intrinsic rank" — meaning the meaningful changes can be captured by a much smaller matrix than the full W . Instead of updating a huge matrix, you update two tiny ones that multiply together to approximate the same result.

Full Fine-Tuning	LoRA (rank $r=8$)
Update a 1000×1000 matrix W	Train two matrices: A (1000×8) and B (8×1000)
1,000,000 trainable parameters	8,000 + 8,000 = 16,000 trainable parameters
100% of original weights	1.6% of original weights — 98.4% reduction! 🦋
Stored as full model (~14GB for 7B model)	Stored as tiny adapter file (~50MB)

🦋 **Huge savings in compute + storage, without losing accuracy.** This is why LoRA has become the default method for fine-tuning open-source LLMs.

6.4 LoRA Key Hyperparameter — Rank (r)

The rank r controls the trade-off between quality and efficiency:

Rank (r)	Trade-off	Typical Use Case
$r = 4$	Fewest parameters, fastest training, least expressive	Simple style adaptation, minor tone changes
$r = 8-16$	Good balance — default choice for most tasks	Domain adaptation, instruction following
$r = 32-64$	More expressive, approaches full fine-tuning quality	Complex new task learning, code generation
$r = 128+$	Very close to full fine-tuning — loses most PEFT benefits	Research experiments only

7. QLoRA — Quantized Low-Rank Adaptation

7.1 The Problem QLoRA Solves

Even with LoRA, the base model itself still needs to be loaded into GPU memory. A 7B parameter model in FP16 (16-bit floating point) takes approximately 14GB of VRAM. **Problem:** base model still too big in FP16 → hard to fit in GPU memory for most consumer hardware.

Solution: QLoRA quantizes the base model to 4-bit precision, dramatically shrinking its memory footprint, then applies the standard small LoRA adapter matrices on top (which remain in FP16 for training accuracy).

7.2 Quantization Explained

Quantization reduces the number of bits used to represent each weight in the model. Instead of storing a weight as a 16-bit float (2 bytes), you store it as a 4-bit integer (0.5 bytes) — a 4x reduction in memory:

Precision	Bytes per Weight	Practical Impact
FP32 (full)	4 bytes	Standard training precision — very large
FP16 (half)	2 bytes	Default LLM inference precision
INT8	1 byte	8-bit quantization (bitsandbytes library)
NF4 / INT4	0.5 bytes	QLoRA's default — 4x smaller than FP16!



Real-World Example: Llama 65B Model

FP16: ~130 GB → Requires multiple high-end A100 GPUs (impossible for most people)

4-bit: ~33 GB → Fits on a single A100 80GB GPU ✓

QLoRA made it possible for individual researchers to fine-tune 65B models on a single GPU for the first time.

7.3 QLoRA = LoRA + Quantization

Component	What it Provides	Precision Used
Base model (quantized)	The frozen model you start from	4-bit NF4 (tiny memory footprint)
LoRA adapter matrices (A & B)	The only trainable parameters	BFloat16 / FP16 (full precision for accuracy)
Training gradients	Flow only through adapter, not base model	FP16 (standard)



Key Insight: QLoRA democratizes training huge models on limited infrastructure. It made fine-tuning LLMs accessible to individual developers and researchers on consumer hardware — this is one of the most important recent advances in open-source AI.

7.4 LoRA vs QLoRA — Summary Comparison

Property	LoRA	QLoRA
Base model precision	FP16 (~14GB for 7B model)	4-bit NF4 (~4GB for 7B model)
Min. GPU VRAM needed	16GB+ for 7B models	6–8GB for 7B models — consumer GPU!
Adapter precision	FP16	BF16/FP16
Training speed	Fast	Slightly slower (quantization overhead)
Output quality	Excellent	Comparable to LoRA (minor accuracy loss)
Practical use	Standard server GPUs (16–40GB)	Consumer GPUs (6–16GB) or when RAM is limited

8. Summary — Quick Reference for Students

8.1 Key Terms Glossary

Term	Definition
On-premise LLM	A large language model running entirely on your own hardware — no cloud API, no data leaving your network.
Ollama	Open-source tool that lets you download and run LLMs locally with a single command.
RAG	Retrieval-Augmented Generation — inject relevant documents into the LLM prompt at query time. Model weights stay frozen.
Fine-tuning	Update a subset of model weights using task-specific labelled data to improve performance on a specific domain.
Re-training	Train a new model from randomly initialised weights using a massive dataset. Extremely expensive; only for well-funded AI labs.
PEFT	Parameter Efficient Fine-Tuning — umbrella term for methods that fine-tune models by updating only a tiny fraction of parameters.
LoRA	Low-Rank Adaptation — inserts trainable low-rank matrices ($A \times B$) into model layers; only A and B are trained. ~98% fewer trainable params.
QLoRA	Quantized LoRA — quantizes the base model to 4-bit to save memory, then applies LoRA adapters in FP16. Fine-tune 7B models on 8GB GPU.
Quantization	Reducing the bit-width used to represent model weights (FP32 → FP16 → INT8 → NF4) to reduce memory footprint.
Rank (r)	The LoRA hyperparameter controlling the size of adapter matrices. Lower r = fewer params but less expressive. Typical: r=8 or r=16.
Adapter	A small, task-specific set of trainable weights inserted into a frozen base model. The base model + adapter = fine-tuned behaviour.
Catastrophic forgetting	When full fine-tuning on a new task causes the model to "forget" its previously learned general capabilities. PEFT avoids this.

NF4	Normal Float 4 — the 4-bit data type used by QLoRA to quantize model weights with minimal accuracy loss.
SFT	Supervised Fine-Tuning — training on labelled (input, output) pairs. The most common first step in fine-tuning.
VRAM	Video RAM — GPU memory. The main bottleneck when running or fine-tuning large models.

8.2 Useful Links

Resource	Link
Ollama — Install & Docs	https://ollama.com
Ollama Model Library	https://ollama.com/library
Hugging Face Models	https://huggingface.co/models
Hugging Face PEFT Library	https://huggingface.co/docs/peft
LoRA Original Paper	https://arxiv.org/abs/2106.09685 (Hu et al., 2021)
QLoRA Original Paper	https://arxiv.org/abs/2305.14314 (Dettmers et al., 2023)
bitsandbytes (quantization)	https://github.com/bitsandbytes-foundation/bitsandbytes
Unsloth (fast QLoRA training)	https://github.com/unslothai/unsloth