# Software 3.0: Fine-tuning

Saeed Siddik
Assistant Professor
IIT University of Dhaka

# Developing AIware systems

- Prompt Engineering
- Fine-tuning
- Retrieval Augmented Generation (RAG)

# Fine tuning

- adapting a pre-trained AI-Model to perform better on a specific task or with a domain-specific dataset.
- The goal is to adapt the model's general knowledge to a specialized task, domain, or style.

# Why Fine-Tune? The PM's View

### Training from Scratch

Requires **massive** datasets

Extremely high compute cost

Very high risk of failure

Impractical for 99% of software projects

### Fine-Tuning

Leverages existing knowledge

Requires a smaller, curated dataset (100s-1000s of examples)

Drastically lower cost and time

Faster path to a specialized model

# When NOT to fine-tune

- If non-compliant with privacy/regulatory constraints and model contains sensitive pretraining data you cannot modify.
- When task requires radically different architecture/inductive biases.
- When you must guarantee zero change from pretrained weights (use inference-time adapters instead).
- When compute budget or latency demands prohibit larger models like considering smaller models or distilled variants.

# Types of Fine-Tuning (Part 1)

## 1. Full Fine-Tuning

Updates **all** weights of the pre-trained model. It's the most thorough method but is computationally expensive and creates a full-size copy of the model.

## 2. PEFT

Parameter-Efficient Fine-Tuning. Updates only a **small subset** of parameters (or adds new ones). It's much faster, cheaper, and creates tiny "adapter" files (e.g., 50MB vs 15GB).

# Types of Fine-Tuning (Part 2): PEFT

Popular Parameter-Efficient Fine-Tuning Methods

- **LoRA (Low-Rank Adaptation):** Injects small, trainable "adapter" matrices into the model. The original model weights are frozen. This is the most popular and effective method.

- **QLoRA (Quantized LoRA):** An optimization of LoRA. Uses quantization to reduce the model's memory footprint, allowing fine-tuning of huge models on smaller, cheaper GPUs.

# Project workflow of Fine-Tuning Process

### 1. Task Definition

Classify taregt question and requirement

### 2. Data Prep

Collect & label 1,000 example with the correct label.

### 3. Model Selection

Choose a base model, e.g., 'BERT-base' or 'Llama-3 8B'.

### 4. Tuning

Run the training data. Feed the 1,000 examples using LoRA.

### 5. Evaluation

Test the new model on unseen data. Check its accuracy.

# Example: Task and data

- **Task**: sentiment classification for customer reviews (3 classes: positive, neutral, negative).
- **Pretrained model**: BERT-base (or a Transformer encoder).
- **Dataset**: 10k labeled reviews, 80/10/10 train/val/test.
- **Evaluation metric**: accuracy + F1 (macro).

# Hands-On: What it Looks Like

## Key Libraries (Python)

In a real project, you'd use open-source libraries to handle the complexity.

- **Hugging Face transformers:** To load the pre-trained model.
- **Hugging Face peft:** To easily apply PEFT methods like LoRA.
- **Hugging Face datasets:** To load and process your custom data.

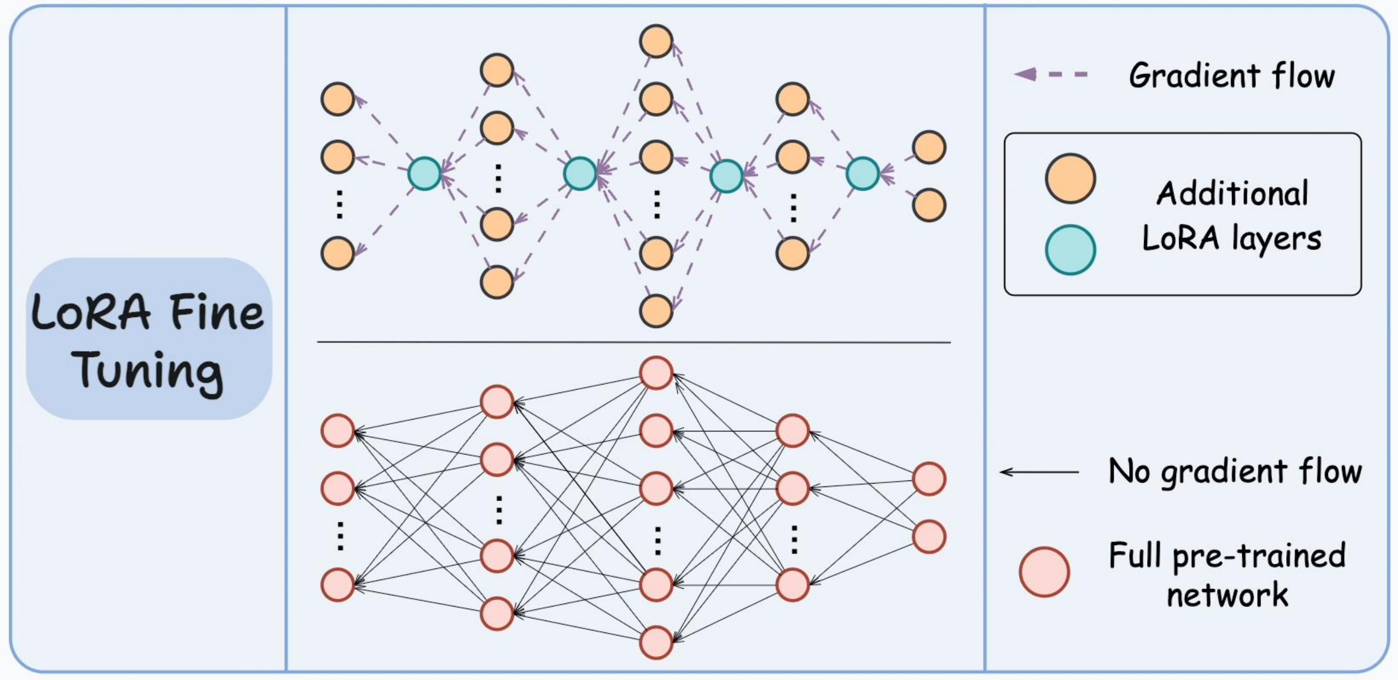
## Conceptual Code

```python
# 1. Import libraries
from transformers import AutoModel, Trainer
from peft import get_peft_model, LoraConfig

# 2. Load base model
model = AutoModel.from_pretrained("meta-llama/Llama-3-8B")

# 3. Define PEFT (LoRA) config
lora_config = LoraConfig(r=8, lora_alpha=16)

# 4. Apply PEFT to the model
peft_model = get_peft_model(model, lora_config)

# 5. Run training
trainer = Trainer(model=peft_model, train_dataset=my_data)
trainer.train()
```

# Low-Rank Adaptation

- LoRA (Low-Rank Adaptation) is a parameter-efficient fine-tuning method that freezes the original model weights and injects small trainable low-rank matrices into specific layers (usually attention layers).
- During training, only these small matrices are updated, not the full model.

# LoRA: Low-Rank Adaptation



# What is QLoRA?

- QLoRA extends LoRA by running the base model in 4-bit quantized form, drastically reducing memory, while still training LoRA adapters in higher precision (NF4/FP16).

# LoRA vs QLoRA

| Aspect | LoRA | QLoRA |
| --- | --- | --- |
| Base model precision | FP16 / BF16 | 4-bit quantized |
| Memory savings | High | Very high (3×–5× more) |
| Accuracy | Near full | Nearto LoRA |
| Training speed | Good | Slightly slower (dequantization cost) |
| Hardware needs | Multiple GPUs for 13B+ | Single 24GB GPU for 65B (with tricks) |
| Use case | Mid-size models | Very large LLMs |

# Building an "AIware" Project with LoRA

### 1. Data Pipeline

The project **is** the data. Your primary focus must be on robust data collection, cleaning, and versioning. Garbage In, Garbage Out.

### 2. Experiment Loop

The 'build' phase is an experiment. You must track the base model, dataset version, hyperparameters, and metrics for **every single run.**

### 3. Model Deployment

The output isn't just code; it's a model artifact (e.g., a 50MB LoRA adapter). This needs its own deployment and serving strategy.

# Project management for fine-tuned AI systems

- **Phases**: discovery → data collection → modeling → integration → testing → deployment → monitoring → maintenance.
- **Roles**: Project Manager, Software Engineer, ML Engineer, Data Engineer, Annotator, QA, DevOps, UX.
- **Deliverables**: dataset spec, baseline, model artifacts, evaluation report, monitoring plan, runbooks.

# Management Challenges (Part 1)

## Data & Cost Management

Acquiring high-quality, **labeled** data is the #1 bottleneck and cost driver.

Compute (GPU) time is your most expensive resource and must be tracked like any other project cost.

## The Versioning Nightmare

You must track **all four** components for reproducibility:

Base Model version (e.g., Llama-3-8B)

Dataset version (e.g., tickets_v1.1)

Training Code version (Git commit)

Resulting Model Artifact

# Management Challenges (Part 2)

## Evaluation is Hard

How do you *know* it's better? Accuracy isn't enough. You must test for bias, fairness, robustness, and specific failure modes. Define success metrics before you start.

## Model Drift & Forgetting

The real world changes. Your model's performance will 'drift' (decay) over time. It may also 'forget' general knowledge after being over-specialized (known as catastrophic forgetting).

# Risk management Challenges

Data risk: low quality or insufficient labels.

Technical risk: model fails at scale, latency issues.

Ethical/legal: bias, privacy infractions.

Operational: costly retraining, lack of monitoring.

Mitigation: pilot studies, audits, staging, KPIs & SLAs.

# End of Fine Tuning