Large Language Model Fine-tuning definition of Key concepts

1. Model Quantization Fundamentals

❖ Quantization definition

Quantization is the process of **reducing the numerical precision** of a model's weights and activations to a lower bit representation, usually from **32-bit floating point (FP32) to 16-bit (FP16), 8-bit (INT8), or even lower**. The goal is to make inference more efficient by reducing memory footprint and computational cost while trying to **minimize the loss of accuracy**.

Mathematically, quantization can be expressed as:

$$q = \text{round}\left(\frac{x}{S}\right) + Z$$

Where:

- x is the original full-precision value
- S (scale factor) determines the range of values
- Z (zero point) shifts the range
- q is the quantized value

❖ Different Numérical precision

Precision	Data Type	Memory Size	Range	Usage
Full Precision	FP32	32 bits	Large range of values	Used in training (high accuracy, high memory cost)
Half Precision	FP16	16 bits	Smaller range than FP32	Used in mixed-precision training to save memory
Integer Precision	INT8	8 bits	Very limited range	Used in quantized inference for speed and efficiency

Key idea:

- **FP32 (float32):** Common in training because it provides high numerical stability.
- **FP16 (float16):** Reduces memory usage but requires careful handling (loss of precision).
- **INT8:** Typically used in inference because integer operations are much faster on hardware like CPUs, TPUs, and NPUs.

❖ Calibration

Calibration is **the process of determining the scale (S) and zero point (Z) parameters** when converting from floating-point to lower-precision representations. It ensures that the new lower-bit representation still retains useful information from the full-precision model.

Mathematical Definition:
The scale factor (S) and zero-point (Z) are computed as:

$$S = \frac{\max(X) - \min(X)}{2^b - 1}$$

$$Z = \text{round}\left(\min(X)/S\right)$$

Where:

- X is the original floating-point tensor
- b is the bit precision (e.g., $b = 8$ for INT8)

❖ Explain and compare types of Quantization

➔ Post-Training Quantization

- Applied **after the model is trained**.
- The trained model is **calibrated**, then quantized.
- **Faster but often leads to accuracy loss** because the model was not trained with quantized weights.

✅ **What happens in PTQ?**

1. Train a model normally in **full precision (FP32)**.
2. **Quantize** the trained model (e.g., convert FP32 weights to INT8).
3. **Infer directly** using the quantized model.

🚀 **Key idea:**

- No additional training after quantization.
- **Fast and simple** but can lead to **accuracy loss**.

➔ Quantization aware Post-Training

- Instead of applying quantization **after training**, the model is trained **while simulating quantization effects**.
- Helps **retain accuracy** since the model learns how to compensate for lower-precision values.
- Used for cases where accuracy loss in PTQ is too high.

✅ **What happens in QAT?**

1. Train a model normally in **full precision (FP32)**.
2. **Quantize the model**, but instead of using it directly, continue training.
3. Fine-tune the quantized model using **custom training data**.
4. **Quantize again** after fine-tuning.
5. Use the new quantized model for inference.

🚀 **Key idea:**

- The model **learns to adapt** to quantization effects.
- **Minimizes accuracy loss** compared to PTQ.
- More computationally expensive because **training is involved**.

❖ Overall comparison

- In **PTQ**, quantization happens **entirely after training**, with no gradient updates to adjust for precision loss.
- In **QAT**, quantization happens in two steps:
    1. **Initial Quantization**: A pre-trained model (FP32) is first quantized to a lower precision (e.g., INT8). This is similar to PTQ, but it's just the starting point.
    2. **Fine-Tuning with Simulated Quantization**: The model is then fine-tuned while applying **fake quantization** (simulated quantization) during forward passes. This means that during training, we still keep FP32 precision for backpropagation, but we introduce quantization effects in forward passes so the model learns to adjust its weights accordingly.

So, while the first quantization (e.g., FP32 → INT8) happens **before** fine-tuning, the key difference is that **the model is then retrained while simulating quantization effects** before final deployment. That's why we say QAT is done "during training"—because fine-tuning adapts to quantization instead of just applying it statically.

🔍 PTQ vs. QAT: Clear Comparison

Feature	Post-Training Quantization (PTQ)	Quantization-Aware Training (QAT)
Training required?	❌ No additional training	✅ Additional training required
Speed & simplicity	✅ Fast & easy	❌ Computationally expensive
Accuracy impact	🔴 Higher accuracy loss	🟢 Better accuracy retention
Ideal for	Fast inference, limited resources	High-accuracy applications

❖ Comparison from a mathematical aspect

3 PTQ vs. QAT - Mathematical Summary

Feature	PTQ (Post-Training Quantization)	QAT (Quantization-Aware Training)
Quantization method	Affine (Linear) quantization	Simulated quantization with STE
Equation (quantization)	$q = \text{round}\left(\frac{x}{S} + Z\right)$	$q = \text{round}\left(\frac{x}{S} + Z\right)$ (STE for gradients)
Equation (dequantization)	$x' = S \cdot (q - Z)$	$x' = S \cdot (q - Z)$
Gradient update	No gradient update for quantized weights	Gradients computed with STE
Accuracy impact	High accuracy loss	Low accuracy loss
Training cost	No additional training	Extra training time required

---

❖ Comparison in term of calibration

1 How Calibration Differs in PTQ vs. QAT

Calibration is the process of **determining the scale factor S and zero point Z** for quantization. The key difference between **PTQ and QAT** is **when and how calibration is done**:

Stage	PTQ (Post-Training Quantization)	QAT (Quantization-Aware Training)
Step 1: Initial Model	Train model in FP32	Train model in FP32
Step 2: Calibration	Use pre-trained weights to compute S and Z	Quantization is simulated **during training** (on-the-fly updates to S and Z)
Step 3: First Quantization	Convert weights **after training** (FP32 → INT8)	Convert weights **before fine-tuning** (FP32 → INT16 or INT8)
Step 4: Fine-Tuning	No fine-tuning	Train model **with quantization effects** applied
Step 5: Second Quantization (if applicable)	Not used	Optionally, further quantize (e.g., INT16 → INT8)
Step 6: Inference	Use quantized model directly	Use the final quantized model after QAT

5 Key Differences in Calibration: PTQ vs. QAT

Feature	PTQ	QAT
Calibration Timing	Done **after** training	Done **during** training
How S, Z are computed?	Computed **from dataset statistics**	Updated **continuously during training**
Quantization Stages	One step (FP32 → INT8)	Can have multiple steps (FP32 → INT16 → INT8)
Fine-Tuning?	No	Yes
Final Model Accuracy	Lower (due to approximation errors)	Higher (model learns to adapt)

❖ Symmetric and Asymmetric Quantization

➔ Symmetric Quantization

- The range of values is **centered around zero**.
- The same scale factor S is used for both positive and negative values.
- **Common in GPUs and TPUs** since hardware accelerators prefer power-of-two scaling.

✅ **Formula:**

$$S = \frac{\max(|X|)}{2^{b-1} - 1}$$

$$q = \text{round}\left(\frac{x}{S}\right)$$

- Since **zero-point** $Z = 0$, the mapping is symmetric around zero.

➔ Asymmetric Quantization

- The range is **not necessarily centered around zero**.
- The scale factor S and zero-point Z are computed to match the full range of the original values.
- **More flexible but slightly more computationally expensive**.

✅ **Formula:**

$$S = \frac{\max(X) - \min(X)}{2^b - 1}$$

$$Z = \text{round}\left(\min(X)/S\right)$$

- **More accurate** for datasets where values are not zero-centered.

❖ How we determine if it symmetric or asymmetric

➔ We could examine the pre-trained Models Weights

Step-by-Step Process:

1. **Load the Pre-trained Model**:
   Load the model you're working with using a deep learning framework (PyTorch, TensorFlow, etc.).
2. **Inspect the Weights**:
   Focus on the weights of the layers (e.g., in a Transformer model, the attention weights and feed-forward layers). The method is similar to checking the weights in any model.
   - **PyTorch Example**:
     In PyTorch, once you load the model, you can easily extract weights using `model.parameters()` or `model.state_dict()`.

➔ Use statistical Metrics for weights distribution

1. **Skewness**:

   Skewness measures the asymmetry of the distribution. For symmetric distributions, skewness is close to zero, while for asymmetric ones, it will be higher or lower.

```python
import scipy.stats as stats

# Calculate skewness for weights
skewness = stats.skew(weights.flatten())
print(f"Skewness: {skewness}")
```

   - **Skewness ≈ 0**: Symmetric distribution.
   - **Skewness > 0**: Positive skew (asymmetric).
   - **Skewness < 0**: Negative skew (asymmetric).

2. **Zero-Centering Check**:

   Check if the mean is near zero. For symmetric quantization, the mean should be near zero.

```python
mean_val = weights.mean()
print(f"Mean: {mean_val}")
```

   - **Mean ≈ 0**: Symmetric.
   - **Mean ≠ 0**: Asymmetric.

---

2. Fine-tuning Approaches and Methods

❖ Type of fine tuning in generale

Fine-tuning methods for **Large Language Models (LLMs)** can be categorized based on how much of the model is modified, the efficiency of training, and the memory requirements. Here's a structured breakdown:

➔ Full fine Tuning

🔹 **Definition**:

In this approach, **all model parameters** are updated during training on a new dataset. It is computationally expensive but allows for full adaptation.

🔹 **Use Case**:

- When you have a **large** dataset similar to the pretraining corpus.
- When you want **maximum adaptability** for a specific task.

🔹 **Mathematical Representation**:

- If θ represents the model parameters, fine-tuning updates θ by optimizing a loss function L using **gradient descent**:

$$\theta' = \theta - \eta \cdot \nabla_\theta L(X, Y)$$

where X is the input data, Y is the target, η is the learning rate.

🔹 **Pros & Cons**: ✅ High performance on the target task.

❌ Expensive in terms of **memory** and **compute**.

➔ Adapter Based fine tuning

🔹 **Definition**:

Instead of modifying all the parameters, adapter-based fine-tuning **adds small trainable layers** (adapters) while keeping the original model frozen.

🔹 **Types**:

- **LoRA (Low-Rank Adaptation)**: Injects **low-rank matrices** into weight updates.
- **Prefix-Tuning**: Adds trainable vectors to **input embeddings**.
- **Adapters**: Introduces extra layers within transformer blocks.

🔹 **Mathematical Representation (LoRA Example)**:

- Instead of updating **all** weights W, we approximate them with **low-rank** matrices A, B:

$$W' = W + AB$$

where A and B are small matrices, reducing memory usage.

🔹 **Pros & Cons**: ✅ Requires **less memory**, enables fast adaptation.

❌ Might be **less effective** than full fine-tuning for drastic domain shifts.

➔ Parameters efficient fine tuning

🔹 **Definition**:

A broader term covering LoRA, adapters, and other methods that fine-tune **only a small subset** of parameters.

🔹 **Techniques**:

- **LoRA** (Low-Rank Adaptation)
- **Prefix-Tuning**
- **Prompt-Tuning** (Optimizing a small prompt-like embedding)
- **BitFit** (Only fine-tuning biases)

🔹 **Pros & Cons**: ✅ **Reduces compute cost**, great for **deployment**.

❌ Some methods may perform **worse than full fine-tuning**.

➔ In context learning

🔹 **Definition**:

Instead of updating model parameters, **examples are provided in the prompt** to guide the model's response.

🔹 **Mathematical Representation**:

- Given **input prompt** X, the model generates output based on **pretrained weights** θ:

  $$P(Y|X,\theta)$$

  No parameter updates occur!

🔹 **Types**:

- **Zero-Shot**: No examples are given.
- **Few-Shot**: Some examples are provided in the prompt.

🔹 **Pros & Cons**: ✅ **No need for retraining**, highly flexible.

❌ **Less reliable** than fine-tuned models for specific tasks.

➔ Reinforcement Learning fine tuning

🔹 **Definition**:

Uses **human feedback** to improve LLMs by optimizing responses.

🔹 **Mathematical Representation**:

- Given a **reward function** $R(x)$, the model is fine-tuned using **policy optimization**:

$$\theta' = \theta + \eta \nabla_\theta R(x)$$

where $R(x)$ comes from human preferences.

🔹 **Pros & Cons**: ✅ Great for **aligning** models with **human values**.

❌ Computationally expensive.

➔ Quantization aware tine tuning

🔹 **Definition**:

- The model is first **quantized** (e.g., from FP32 → INT8).
- Then, **fine-tuning** is done to recover lost accuracy.

🔹 **Mathematical Representation**:

- Quantized weights:

$$W_q = S \cdot (Q(W) - Z)$$

where S is the **scale factor**, $Q(W)$ is the quantized value, and Z is the **zero point**.

🔹 **Pros & Cons**: ✅ Reduces model **size and inference cost**.

❌ Can lead to **accuracy loss** if not done carefully.

➔ General comparaison

Fine-Tuning Type	Trainable Params?	Efficiency	Use Case
Full Fine-Tuning	All	High compute cost	Best for large, custom datasets
Adapter-Based (LoRA, Prefix, Adapters)	Few	Low memory	Works well for domain-specific tasks
PEFT (LoRA, BitFit, Prompt-Tuning)	Few	Very efficient	Low-resource environments
In-Context Learning	None	No fine-tuning	Prompt-based adaptation
RLHF	All	Expensive	Human-aligned responses
QAT	Some	Efficient	Low-power inference

❖ Categorization by Model Modification

We can also categorized into how we adapt the LLM to our needs :**full parameter fine-tuning, domain-specific fine-tuning, and task-specific fine-tuning**—are commonly used in the context of **adapting LLMs:**

➔ Full parameters fine tuning

✅ **Definition**:

In this approach, **all** model parameters are updated using backpropagation on a new dataset. It essentially **re-trains the model** on a specialized dataset.

✅ **Use Case**:

- When you have a **large** dataset similar to the model's pretraining data.
- When you need **maximum adaptation** to a **new domain**.
- Works best with **high computational resources**.

✅ **Mathematical Representation**:

Given an LLM with parameters θ, fine-tuning updates them via gradient descent:

$$\theta' = \theta - \eta \cdot \nabla_\theta L(X, Y)$$

where:

- X is input data, Y is the target,
- η is the learning rate,
- $L(X, Y)$ is the **loss function**.

✅ **Pros & Cons**: ✅ Achieves **the best possible performance**.

❌ **Computationally expensive** (requires GPUs, large memory).

❌ May cause **catastrophic forgetting** (overwrites original knowledge).

➔ Domain specific fine tuning

✅ **Definition**:

Instead of modifying the model **for a single task**, domain-specific fine-tuning adapts it to a **new field or industry** (e.g., finance, medicine, law).

- The model learns **specialized vocabulary, reasoning, and patterns**.

✅ **Use Case**:

- Fine-tuning a **general-purpose LLM** for **medical, legal, or finance** applications.
- When the new domain has **special terminology** not well-covered in pretraining.

✅ **Example**:

- Fine-tuning **GPT-4** for **medical diagnoses** using a large **medical text dataset**.
- Adapting a language model for **legal document understanding**.

✅ **Mathematical Process**: Same as full fine-tuning but on a **domain-specific dataset**:

$$\theta' = \theta - \eta \cdot \nabla_\theta L_{\text{domain}}(X, Y)$$

where L_{domain} is a loss function optimized for domain-specific data.

✅ **Pros & Cons**: ✅ Performs well **across tasks in the target domain**.

✅ **More efficient** than full fine-tuning.

❌ Might not generalize well **outside the domain**.

➔ Tasks Specific fine Tuning

✅ **Definition**:

Fine-tuning a model **for a particular NLP task** like **text classification, summarization, or question answering**.

✅ **Use Case**:

- Fine-tuning **GPT-4** for **sentiment analysis** on customer reviews.
- Training **Llama-2** for **chatbot responses**.
- Adapting a model for **financial fraud detection**.

✅ **Example**: Fine-tuning **a general-purpose LLM** for **text summarization**:

- Dataset: **News articles → summary pairs**
- Task: **Generate accurate summaries**

✅ **Mathematical Process**: For a specific task with dataset D_{task}, the model learns:

$$\theta' = \theta - \eta \cdot \nabla_\theta L_{\text{task}}(X, Y)$$

where:

- L_{task} is the loss function optimized for **a single task**.

✅ **Pros & Cons**: ✅ Highly optimized for **one task**.
✅ **Less expensive** than full fine-tuning.
❌ Limited **generalization**—only works well for **one type of task**.

➔ General comparaison

Fine-Tuning Type	What Changes?	Best For	Pros	Cons
Full Fine-Tuning	All parameters	Any application	High adaptability	Expensive, risk of forgetting
Domain-Specific	Model learns domain knowledge	Medical, legal, finance, etc.	Good for multiple tasks in a field	May not generalize
Task-Specific	Trained for one NLP task	Chatbots, classification, QA	Very effective for its task	Limited to one task

❖ Parameter-Efficient Fine-tuning Techniques

➔ Definition

Parameter-Efficient Transfer Learning (PETL) refers to techniques that allow fine-tuning large language models (LLMs) **without updating all model parameters**. Instead, PETL methods **modify only a small subset** of parameters while keeping the majority of the model **frozen**. This significantly reduces computational and memory costs while still adapting the model to new tasks.

➔ Key PETL techniques in NLP

Method	How It Works?
LoRA (Low-Rank Adaptation)	Injects **low-rank matrices** into Transformer layers and trains only these new matrices.
Adapters	Adds small **task-specific layers** between Transformer layers and trains only these layers.
Prefix-Tuning	Adds trainable **prefix tokens** to the model's input sequence.
Prompt-Tuning	Optimizes **soft prompts** instead of changing model weights.
BitFit	Only **fine-tunes bias parameters** of the model.

➔ Explain LORA technique in depth

✅ **Where It's Used**:

- **Task-Specific Fine-Tuning** (mostly)
- **Domain-Specific Fine-Tuning** (sometimes)
- Used when **full fine-tuning is too expensive**.

✅ **How It Works**:

LoRA **freezes** the original model weights and **adds trainable low-rank matrices** to selected layers.
Instead of updating all parameters (θ), it modifies a smaller low-rank decomposition:

$$\Delta W = AB$$

Where:

- W is the original model weight matrix.
- A and B are **small trainable matrices** (with a rank much smaller than W).
- The final adapted weight is:

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

➔ Explain QLORA technique in depth

✅ **Where It's Used**:

- **Task-Specific Fine-Tuning** (mostly)
- Used when **even LoRA is too expensive** in memory.

✅ **How It Works**:

- First, it **quantizes** the LLM to **4-bit precision** to save memory.
- Then, it **applies LoRA** on top of the quantized model.
- It still **only trains small low-rank matrices** but with **less memory overhead**.

✅ **Mathematical Process**:

1. **Quantization**: Convert model weights W to **4-bit format**.
2. **LoRA Adaptation**: Train small **LoRA matrices** ($\Delta W = AB$).
3. **Final Model**:

$$W'_{\text{quantized}} = W_{\text{quantized}} + \Delta W$$

❖ Evaluating the Fine-Tunes LLMs

➔ Evaluation Metrics

1. **Accuracy on Fine-Tuning Data**
   - **Test Set**: After fine-tuning, evaluate the model on a **test set** (a separate labeled dataset) to assess if it correctly applies what it learned. For example, if you fine-tuned the model for a **QA task**, evaluate its **accuracy**, **F1 score**, or **exact match** with the correct answers.
2. **Cross-Validation**
   - Use **k-fold cross-validation** to ensure that the fine-tuned model doesn't overfit to the fine-tuning dataset and can generalize well across different subsets of the data.
3. **Task-Specific Metrics**:
   - Depending on the task, you'll use specific metrics:
     - **Classification**: Accuracy, Precision, Recall, F1 Score.
     - **Translation or Summarization**: BLEU, ROUGE, METEOR.
     - **Question Answering**: Exact Match (EM), F1 Score.
     - **Text Generation**: Perplexity, Diversity (e.g., n-gram overlap).

➔ Ensuring the fine tuned Model's responses are accurate

1. **Output Inspection**:
    - Manually or programmatically inspect the model's outputs to see if it reflects the fine-tuned task. For example:
        - **For a QA task**, check if the model retrieves and responds correctly to answers based on your fine-tuned knowledge base.
        - **For a specific domain (e.g., law or medicine)**, verify that the responses reflect the specialized knowledge it learned from the fine-tuning dataset.
2. **Fine-Tuning Monitoring**:
    - Keep track of performance changes during fine-tuning. For example, monitor:
        - **Loss curves**: If the loss plateaus or reduces significantly during fine-tuning, it might indicate the model is learning the task well.
        - **Validation vs. Training Accuracy**: If the validation accuracy is similar to or better than training accuracy, it indicates good generalization.

❖ Fine-Tuning vs RAG

- **Fine-Tuning** involves adjusting a pre-trained model on a specialized dataset so that it can better understand the domain-specific knowledge, context, or tasks you're interested in. The fine-tuned model can give you answers **directly** based on that specialized training, without needing an external knowledge source at inference time.
- **RAG (Retrieval-Augmented Generation)** systems, on the other hand, combine a language model with an external knowledge base (like a database, search engine, or document set). The model retrieves **relevant information** from this external source during inference and then uses that information to generate a more informed response.

❖ Relation between Fine tuning and Transfer Learning

➔ What is Transfer Learning ?

Transfer learning is a **machine learning technique** where a model trained on one task is **reused or adapted** for a different but related task. Instead of training a model from scratch, which requires a large dataset and significant computational resources, we leverage a **pre-trained model** and modify it for a new task.

➔ How Transfer Learning works ?

1. **Pre-training Phase:**
   - A model is trained on a **large general dataset** (e.g., ImageNet for images, Common Crawl for text).
   - This model learns useful features that can be applied to various tasks (e.g., edges in images, syntax in text).
2. **Adaptation Phase (Fine-Tuning or Feature Extraction):**
   - The pre-trained model is **adapted** to a new, domain-specific task with a smaller dataset.
   - This adaptation can happen in different ways, leading to different types of transfer learning.

Modification	When to Use It?
Replace only the output layer	When the task is similar to the original training task, and you have a small dataset.
Modify or add layers before the output	When the new task is different, and the existing features are not enough.
Fine-tune earlier layers	When domain-specific knowledge is needed, and you have enough data.

➔ What are the Types of Transfer Learning ?

1. Feature Extraction (Frozen Pre-trained Model)

- We **freeze** the weights of the pre-trained model and only train a new **classifier or head layer** on top of it.
- The pre-trained model acts as a **fixed feature extractor**.
- Example: Using ResNet trained on ImageNet to classify medical images by adding a new classification layer.

Mathematically:
If $f_{\text{pretrained}}(x)$ is the output of the frozen pre-trained model for an input x, we train a new head $g(x, \theta)$:

$$y = g(f_{\text{pretrained}}(x), \theta)$$

Here, only θ (parameters of the new head) are trained.

2. Fine-Tuning (Updating Pre-trained Weights)

- We allow **some or all** of the pre-trained model's layers to be updated on the new dataset.
- Fine-tuning helps the model specialize for the new task while still leveraging the pre-trained knowledge.
- We may use **full parameter fine-tuning** (all layers are updated) or **partial fine-tuning** (only some layers are updated).

Mathematically:
Instead of freezing $f_{\text{pretrained}}(x)$, we update its parameters $\theta_{\text{pretrained}}$:

$$y = g(f_{\text{pretrained}}(x, \theta_{\text{pretrained}}), \theta_{\text{head}})$$

Here, **both** $\theta_{\text{pretrained}}$ and θ_{head} are trained.

➔ Relationship Between Transfer Learning and Fine-Tuning

Fine-tuning is a **type of transfer learning** where we **partially or fully update** a pre-trained model for a new task.

- **Transfer learning = Reusing a pre-trained model**
- **Fine-tuning = Updating the pre-trained model's parameters for a new task**