Quantization in fine-tuning is a technique used to reduce the memory and computational requirements of a model by approximating the model's parameters with lower precision. Typically, models are trained with high precision (e.g., 32-bit floating-point numbers), but quantization allows these parameters to be represented using lower precision formats, such as 8-bit integers, without significantly compromising the model’s accuracy.

**Key Aspects of Quantization in Fine-Tuning**

1. **Purpose**: The main goal is to make the model more efficient in terms of speed and memory usage, which is especially important for deploying models on resource-constrained devices like mobile phones or edge devices.
2. **How It Works**: During quantization, weights and sometimes activations are converted from higher precision (e.g., 32-bit floats) to lower precision (e.g., 8-bit integers). This conversion may occur either before fine-tuning or as part of the fine-tuning process to adjust the model’s performance and stability with reduced precision.

**Benefits**

* **Reduced Model Size**: Using lower precision reduces the amount of memory needed to store the model.
* **Faster Inference**: Models often run faster, especially on specialized hardware optimized for lower precision arithmetic.
* **Energy Efficiency**: Quantized models consume less power, which is crucial for mobile and edge applications.

**Challenges**

* **Potential Loss of Accuracy**: The main trade-off is that quantization may lead to a slight drop in model accuracy, especially for tasks that require very precise computations.
* **Complexity in Implementation**: Ensuring stability and performance during quantization-aware training can be complex.

FP32🡪FP16(Half Precision)  
Full precision

**How to perform Quantization**

It has 2 Types

1)Symmetric Quantization

Symmetric means numbers are qually distributed.\_\_\_\_\_\_\_\_\_\_\_-2\_\_-1\_0\_1\_2\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

🡪Symmetric quantization is done via Batch normalization.

Technique 1:Symmetric uint 8 quantization

Lets say weights of model are in form of floating point number [0,1000] and we wanna convert to [0,255](Since its unsigned).

Done using Min-Max Scaling:

0.0--🡪0

1000--🡪255

0\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1000(x)

To

0\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_255(q)

Scale=(X(max)-X(min))/q(max)-q(min)

=1000-0/255-0

=1000/255=3.92

Q=round(w/s)

W is original weight we wanna compress.

Let's try quantizing a few sample weights:

For W=0

Q1=round(0/3.92)=0

Q2=round(1000/3.92)255

 **Original Range**: [0, 1000]

 **Quantized Range**: [0, 255]

 **Scale Factor**: s≈3.9216s \approx 3.9216s≈3.9216

 **Quantization Formula**: q=round(ws)q = \text{round}\left(\frac{w}{s}\right)q=round(sw​)

 **Clamping**: Ensure qqq stays within the range [0, 255].

2)Asymmetric Quantization

Asymmetric means in number line when Wts are plotted they are either left skewed or right skewed.

\*In asymmetric quantization, the quantized range does not have to be centered around zero. This is in contrast to **symmetric quantization**, where the quantized range is typically centered around zero.

**Quantization Formula**: The formula for quantizing a value www is:

q=round(ws)+zero\_point

Let S= [-20.0 to 1000.0] & we wanna convert to Unsigned int 8 i e.[0 to 255]

Scale factor=s=1000+20/255=4.0

Round(-20/4)-🡪-5

But my distribution should strat from 0.

So we’ll add+5 to make Zero.This +5 is Zero point

**1. Symmetric Quantization**

* **Definition**: In symmetric quantization, the range of the floating-point values (weights or activations) is centered around zero, and the quantization levels are equally spaced on both the positive and negative sides.
* **Scale Factor**: The same scale factor sss is used for both positive and negative values.
* **Zero Point**: The zero point is typically fixed at zero, meaning that the integer value representing zero in the quantized space is exactly zero.
* **Quantization Formula**: The formula for quantizing a value www is: q=round(ws)q = \text{round}\left(\frac{w}{s}\right)q=round(sw​) where sss is the scale factor.
* **Use Cases**: Symmetric quantization is simpler and computationally efficient, making it suitable for scenarios where weight values are distributed relatively evenly around zero.

**Example**

* If the original weight values range from −1000-1000−1000 to 100010001000, a symmetric quantization would map these values to a fixed integer range, such as [−127,127][-127, 127][−127,127] for INT8 quantization, using a single scale factor.

**2. Asymmetric Quantization**

* **Definition**: In asymmetric quantization, the range of the floating-point values is not centered around zero. The quantization levels are not evenly distributed on both sides of zero.
* **Scale Factor and Zero Point**: Asymmetric quantization uses a different zero point, which is an integer value that maps to zero in the quantized space. This allows the quantization to better handle cases where the data range is skewed or does not include zero.
* **Quantization Formula**: The formula for quantizing a value www is: q=round(ws)+zero\_pointq = \text{round}\left(\frac{w}{s}\right) + \text{zero\\_point}q=round(sw​)+zero\_point where sss is the scale factor and zero\_point is the integer that represents zero in the quantized space.
* **Use Cases**: Asymmetric quantization is often used for activations, where the data distribution might be skewed or strictly non-negative (e.g., after ReLU activations).

**Example**

* If the original weight values range from 000 to 100010001000, asymmetric quantization would map these values to [0,255][0, 255][0,255] for UINT8 quantization, using both a scale factor and a zero point to handle the range efficiently.

**Key Differences**

1. **Zero Point**:
   * **Symmetric**: The zero point is always at zero. The quantization grid is symmetric around zero.
   * **Asymmetric**: The zero point is not necessarily zero. It can be adjusted to accommodate the distribution of the values, making it more flexible.
2. **Scale Factor**:
   * **Symmetric**: Uses a single scale factor for both positive and negative ranges.
   * **Asymmetric**: Uses a scale factor and a zero point, allowing for more accurate representation of skewed distributions.
3. **Complexity**:
   * **Symmetric**: Simpler to implement and generally more efficient.
   * **Asymmetric**: Slightly more complex due to the need for an additional zero point, but can provide better accuracy for certain distributions.
4. **Accuracy**:
   * **Symmetric**: May lose accuracy when the range of values is not centered around zero.
   * **Asymmetric**: Generally provides better accuracy for non-centered or skewed distributions, such as activations after a ReLU layer.

**Practical Considerations**

* **Hardware Support**: Some hardware accelerators are optimized for symmetric quantization, making it preferable in those cases.
* **Performance vs. Accuracy**: Symmetric quantization is faster but may sacrifice accuracy, especially when dealing with data that is not centered around zero. Asymmetric quantization provides better accuracy at the cost of slightly increased computational complexity.

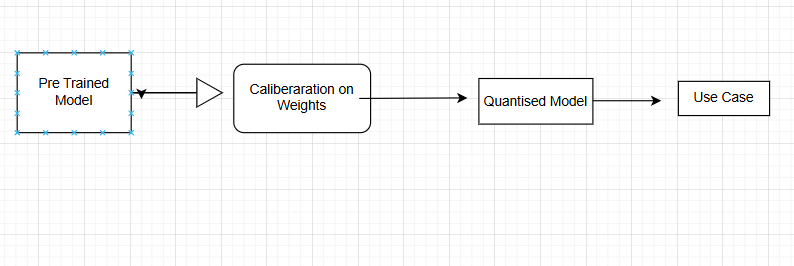
**When to Use Each Type**

* **Symmetric Quantization**: Suitable for weights that are evenly distributed around zero, such as in convolutional or fully connected layers.
* **Asymmetric Quantization**: Better for activations, especially when the values are non-negative or highly skewed.

Caliberation:Process of Squissing Weights from Higher value format to lower value format or process of Quantization is called Caliberation

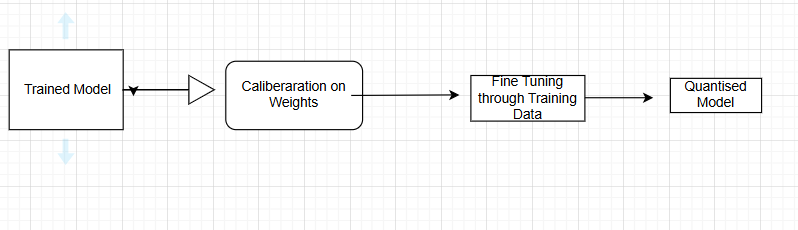
Modes Of Quantization:

* **Post-Training Quantization (PTQ)**: Quantization is applied to a pre-trained model without any additional training. This is simple but may result in a drop in accuracy.



This result in loss of Data

* **Quantization-Aware Training (QAT)**: The model is trained with quantization in mind, often giving better accuracy but requiring additional training.

s

* Mostly We’ll be doing QAT

So When we’ll do fine tuning we’ll basically changing the paerameters of model.Imagine 175B Parameter changing.Which will lead to resource constraint.To Achieve this WE HAVE TECHNIQU LIKE LORA & QLORA.

I] LORA[Low Order Rank Adaptation]

* Instead OF UPDATE All weights it track the changes

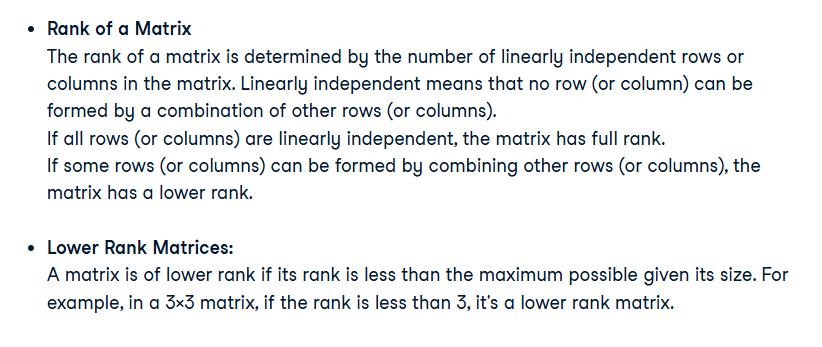
**(Datacamp LORA website)**

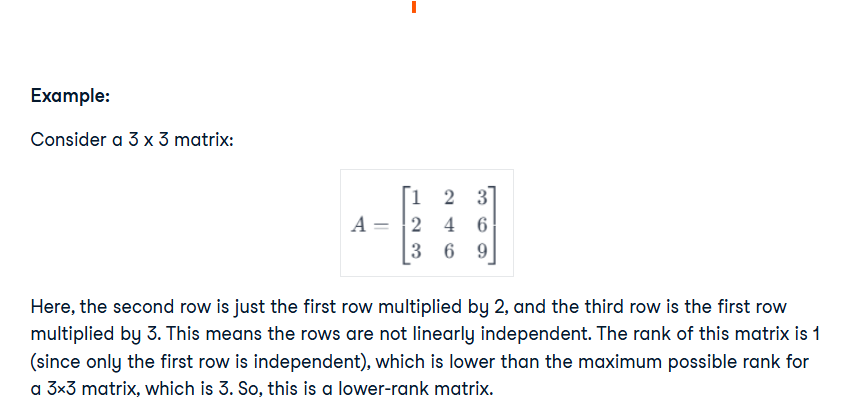
**How LoRA Works**

* At a high-level here is how LoRA works:
* It keeps the original model unchanged and adds small, changeable parts to each layer of the model. This significantly reduces the trainable parameters of the model and reduces the GPU memory requirement for the training process, which is another significant challenge when it comes to fine-tuning or training large models.
* For example, Full fine-tuning of the GPT-3 model will require us to train 175 billion parameters. Using LoRA, the trainable parameters for GPT-3 will be reduced roughly by 10,000 times and GPU memory requirements by three times.

In essence, LoRA solves these problems:

1. **Speed** - because less trainable parameters mean faster training
2. **Compute Resources**- less trainable parameters mean less compute resources required for the training process, making it financially viable to fine-tune large models.
3. **Memory efficiency**- less trainable parameters mean we can cache them in memory, eliminating the need for disk reads, which are inefficient compared to reading from memory.





Lower-rank matrices are significant in various applications like data compression, where reducing the rank of a matrix helps to compress the data while preserving as much information as possible.

The rank in a matrix applies equally to both rows and columns. The crucial point to understand is that the rank of a matrix is the same whether you calculate it based on rows or columns. This is because of a fundamental property in linear algebra known as the Rank-Nullity Theorem.

In simpler terms, the theorem states that the dimensions of the row space (space spanned by the rows) and the column space (space spanned by the columns) of a matrix are equal. This common dimension is what we refer to as the rank of the matrix.

WHAT IS LORA?

In very simple words, LoRA leverages the concept of lower-rank matrices to make the model training process extremely efficient and fast.

Large models have a lot of parameters. For example, GPT-3 has 175 billion parameters. These parameters are just numbers stored in matrices. Storing them requires a lot of storage.

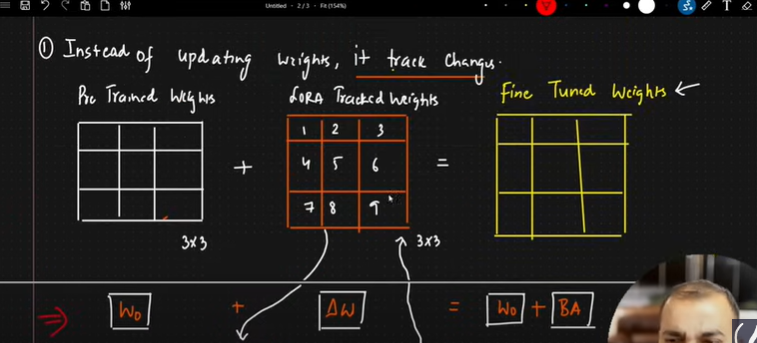
Full fine-tuning means all the parameters will be trained, and this will require an extraordinary amount of compute resources that can easily cost in the millions of dollars for a model size like GPT.

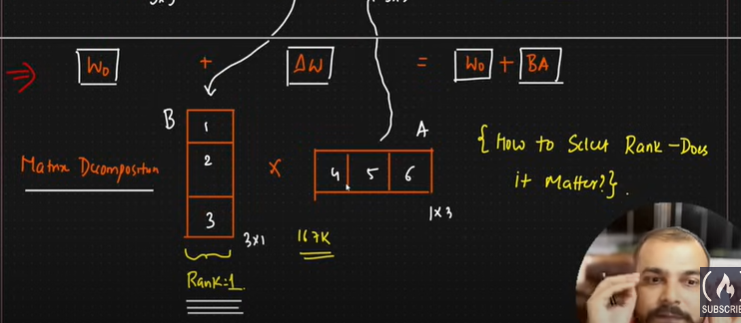
Unlike traditional fine-tuning that requires adjusting the entire model, **LoRA focuses on modifying a smaller subset of parameters (lower-rank matrices)**, thereby reducing computational and memory overhead. We do not change any parameters for a pre-trained model. Instead, only train lower-rank matrices, which happen relatively very quickly because of fewer parameters.

LoRA is built on the understanding that large models inherently possess a low-dimensional structure. By leveraging low-rank matrices, LoRA adapts these models effectively. This method focuses on the core concept that significant model changes can be represented with fewer parameters, thus making the adaptation process more efficient.

IMPORTANT QUESTION:When to use high rank?

=>We we want to make model learning more complex on more parameter.





II]QLORA(Quantised LORA)

LORA (Low-Rank Adaptation) is an effective and efficient method for fine-tuning large language models, but it has limitations when applied to **extremely large models** (e.g., 30B+ parameters). These limitations prompted the development of **QLORA** to address specific challenges:

**Limitations of LORA:**

1. **High Memory Requirements for Large Models**:
   * LORA reduces the number of trainable parameters but still requires storing the original model weights in full precision (16-bit or 32-bit floating point).
   * For very large models (e.g., 30B–65B parameters), the memory footprint of these weights becomes prohibitive, even with LORA.
2. **Hardware Constraints**:
   * While LORA is efficient, it still demands GPUs with substantial VRAM for ultra-large models.
   * This makes it inaccessible to researchers or practitioners with consumer-grade GPUs.
3. **Scalability Issues**:
   * As model sizes grow, the memory overhead associated with storing and operating on full-precision weights becomes a bottleneck.
   * LORA does not inherently address this, focusing only on reducing the trainable parameter count.

**How QLORA Addresses These Limitations:**

1. **4-bit Quantization**:
   * QLORA introduces **4-bit quantization** for model weights, drastically reducing the memory footprint of the base model.
   * This allows even very large models to fit into the memory of commodity GPUs, making fine-tuning accessible on resource-constrained hardware.
2. **Retaining LORA's Efficiency**:
   * QLORA still uses low-rank matrices for adaptation, combining this with quantization to maintain fine-tuning efficiency.
   * This dual approach reduces both trainable parameters (via LORA) and memory requirements (via quantization).
3. **Improved Accessibility**:
   * By significantly lowering the resource requirements, QLORA democratizes access to large language model fine-tuning for smaller labs and individual researchers.

