Parameter-Efficient Fine-Tuning for Large Language Models: LoRA and QLoRA

1. Introduction

The advent of large language models (LLMs) has revolutionized natural language processing (NLP). These models, trained on vast datasets, excel at understanding and generating human-like text. However, fine-tuning these massive models for specific tasks poses challenges due to the high computational costs and memory requirements.

To address this, researchers have developed Parameter-Efficient Fine-Tuning (PEFT) techniques, which aim to achieve high task performance while minimizing the number of trainable parameters. This allows for efficient adaptation of LLMs to specific tasks without compromising performance.

2. Pretrained LLMs and Fine-Tuning

Pretrained LLMs are trained on massive amounts of general-domain data, making them adept at capturing rich linguistic patterns and knowledge. Fine-tuning involves adapting these pretrained models to specific downstream tasks, leveraging their knowledge to excel at specialized tasks. This involves training the pretrained model on a task-specific dataset, usually smaller and more focused than the original training data. During fine-tuning, the model's parameters are adjusted to optimize its performance for the target task.

3. PEFT Techniques

PEFT methods offer an efficient approach to fine-tuning pretrained LLMs while significantly reducing the number of trainable parameters. These techniques balance computational efficiency and task performance, making it feasible to fine-tune even the largest LLMs without compromising quality.

3.1 - Advantages of PEFT

PEFT brings several practical benefits, such as:

- **Reduced Memory Usage :** PEFT methods require less memory for training and inference.
- **Reduced Storage Cost :** Fine-tuned models are typically the same size as the original pretrained model. However, with PEFT, we add small trained weights on top of the pretrained model. This means the same pretrained model can be used for multiple tasks without replacing the entire model, reducing storage requirements.
- **Reduced Inference Latency :** PEFT often results in faster inference times, making it suitable for real-time applications.
- **Efficient Task Switching :** PEFT allows the pretrained model to be shared across multiple tasks, minimizing the need to maintain separate fine-tuned instances for each task. This facilitates quick and seamless task-switching during deployment, reducing storage and switching costs.

3.2 - Various Peft Techniques Approaches.

- Adapter, LoRA, Prefix Tuning, Prompt Tuning, P-Tuning, IA3
- Remember, these techniques help fine-tune models efficiently, like finding the right tools, adjusting layers, and tuning the orchestra for optimal performance! 🚀

1. **Sparse Fine-Tuning (SFT)**:
    - **Definition**: SFT selectively fine-tunes only a subset of the model's parameters, leaving the rest fixed. It identifies important layers or neurons and updates them while keeping others unchanged.
    - **Simple Explanation**: Imagine you have a big toolbox, but you only need a few specific tools for a task. SFT is like using just those essential tools without touching the others.
2. **Layer Drop (LD)**:
    - **Definition**: LD randomly drops certain layers during fine-tuning. It encourages the model to rely on remaining layers effectively.

- **Simple Explanation**: Think of a layered cake. LD removes some layers, leaving a lighter cake that still tastes great.

3. **Layer-wise Learning Rate Scaling (LLRS)**:
   - **Definition**: LLRS adjusts the learning rate for each layer during fine-tuning. Layers closer to the input receive smaller updates, while deeper layers get larger ones.
   - **Simple Explanation**: It's like tuning different instruments in an orchestra—each layer gets its own adjustment for harmony.

4. LoRA: Low-Rank Adaptation

LoRA (Low-Rank Adaptation) is a PEFT technique designed to efficiently fine-tune pre-trained language models by injecting trainable low-rank matrices into each layer of the Transformer architecture. LoRA aims to reduce the number of trainable parameters and the computational burden while maintaining or improving the model's performance on downstream tasks.

4.1 - How LoRA Works / LoRA's key principles are :

- **Starting Point Preservation :** LoRA assumes the pretrained model's weights are already close to the optimal solution for downstream tasks. Thus, LoRA freezes the pretrained model's weights and focuses on optimizing trainable low-rank matrices instead.
- **Low-Rank Matrices :** LoRA introduces low-rank matrices, represented as matrices A and B, into the self-attention module of each layer. These low-rank matrices act as adapters, allowing the model to adapt and specialize for specific tasks while minimizing the number of additional parameters needed.
- **Rank-Deficiency :** LoRA leverages the observation that the weight changes during adaptation can be effectively represented with a much lower rank than the original weight matrices. This allows for significant parameter efficiency.

4.2 - Advantages of LoRA

- **Reduced Parameter Overhead :** LoRA significantly reduces the number of trainable parameters, making it much more memory-efficient and computationally cheaper.
- **Efficient Task-Switching :** LoRA enables the pretrained model to be shared across multiple tasks, minimizing the need to maintain separate fine-tuned instances for each task.
- **No Inference Latency :** LoRA's design ensures no additional inference latency compared to fully fine-tuned models, making it suitable for real-time applications.

5. QLoRA: Quantized Low-Rank Adaptation

QLoRA (Quantized Low-Rank Adaptation) is an extension of LoRA that further enhances parameter efficiency by introducing quantization. It builds on the principles of LoRA while introducing 4-bit NormalFloat (NF4) quantization and Double Quantization techniques.

5.1 - How QLoRA Works

- **NF4 Quantization :** NF4 quantization leverages the inherent distribution of pre-trained neural network weights, typically zero-centered normal distributions with specific standard deviations. By transforming all weights to a fixed distribution that fits within the range of NF4 (-1 to 1), NF4 quantization effectively quantifies the weights without requiring expensive quantile estimation algorithms.
- **Double Quantization :** Double Quantization addresses the memory overhead of quantization constants. It significantly reduces the memory footprint without compromising performance by quantizing the quantization constants themselves. This process involves using 8-bit Floats with a block size of 256 for the second quantization step, resulting in substantial memory savings.

5.2 - Advantages of QLoRAFurther Memory Reduction:

- QLoRA achieves even higher memory efficiency by introducing quantization, making it particularly valuable for deploying large models on resource-constrained devices.

- **Preserving Performance :** Despite its parameter-efficient nature, QLoRA retains high model quality, performing on par or even better than fully fine-tuned models on various downstream tasks.
- **Applicability to Various LLMs :** QLoRA is versatile and applicable to different language models, including RoBERTa, DeBERTa, GPT-2, and GPT-3, enabling researchers to explore parameter-efficient fine-tuning for various LLM architectures.

Conclusion

Parameter-efficient fine-tuning techniques like LoRA and QLoRA are crucial for making large language models more accessible and practical for real-world applications. These methods offer a promising avenue for deploying LLMs in diverse settings, making NLP more accessible and efficient than ever before.

Fine-Tuning for Large Language Models: LoRA and QLoRA

Introduction

The advent of large language models (LLMs) has revolutionized natural language processing (NLP). These models, trained on vast datasets, excel at understanding and generating human-like text. However, fine-tuning these massive models for specific tasks poses challenges due to the high computational costs and memory requirements.

To address this, researchers have developed Parameter-Efficient Fine-Tuning (PEFT) techniques, which aim to achieve high task performance while minimizing the number of trainable parameters. This allows for efficient adaptation of LLMs to specific tasks without compromising performance.

Fine-Tuning LLM Models with Custom Datasets.

In this document, we delve into the intricacies of fine-tuning various machine learning models using custom datasets. Our focus will be on understanding the underlying concepts and mathematical equations. Specifically, we'll employ the "llama 2" model, leveraging techniques such as parameter-efficient transfer learning and Low-rank adaptation of large language models (LoRA).

Key Points :

1. **Custom Dataset :** We'll work with a custom dataset tailored to our specific problem domain.
2. **Transfer Learning :** We'll explore parameter-efficient transfer learning, a powerful technique for adapting pre-trained models to new tasks.
3. **LoRA :** Low-rank adaptation of large language models (LoRA) allows us to fine-tune our model effectively.
4. **Mathematical Details :** We'll provide mathematical explanations to deepen your understanding.
5. **Code Implementation :** Expect code examples to accompany the theoretical concepts.

Remember, while this topic can be complex, we'll strive to simplify explanations wherever possible. Feel free to explore the code and concepts—it'll be valuable for both your studies and real-world applications.

▼ **PEFT**

- **PEFT** stands for **Parameter-Efficient Fine-Tuning**. It's a method that allows you to make small, targeted updates to a large model without having to retrain the entire thing. This saves a lot of computational resources and time.

▼ **LoRA**

- **LoRA** stands for **Low-Rank Adaptation**. It's a technique used within PEFT where you only update a small part of the model's weights. These updates are done in a way that captures the essence of the changes needed for the new task, without altering the entire model.

▼ **QLoRA**

- **QLoRA** is an extension of LoRA that includes **quantization**. Quantization is a process of reducing the precision of the model's weights, which can significantly decrease the model's size and speed up computation. QLoRA applies this to the LoRA updates, making the fine-tuning process even more efficient.

In Simple Terms : Think of a large language model like a huge library of books. Normally, if you wanted to update the library's collection for a new subject, you'd have to replace a lot of books. But with PEFT, you're just adding a few key books to strategic locations. LoRA is like choosing very thin books that fit perfectly in small gaps on the shelves, and QLoRA is like printing these books on thinner paper, so they take up even less space and are quicker to read.

These methods are particularly useful when you want to adapt a model to a new task or domain but don't want to spend a lot of resources on training from scratch. They're part of the broader effort to make AI more accessible and sustainable.

Let's break down the concepts of **accelerators** and **bitsandbytes** in the context of **PEFT (Parameter-Efficient Fine-Tuning)**.

▼ **Accelerators**:

- **Accelerate** is a library designed for distributed training and inference on various hardware setups, including GPUs, TPUs, and Apple Silicon.
- In the context of PEFT, Accelerate plays a crucial role in making large models more accessible. Here's how:
  - **Training Efficiency**: PEFT methods fine-tune only a small number of additional model parameters (instead of all parameters) to adapt large pretrained models to specific downstream tasks. This significantly reduces computational costs.
  - **Storage Efficiency**: Fine-tuned models are typically the same size as the original pretrained model. However, with PEFT, we add small trained weights on top of the pretrained model. This means that the same pretrained model can be used for multiple tasks without replacing the entire model.
  - **Integration**: Accelerate seamlessly integrates with PEFT, making it convenient to train large models or use them for inference even on consumer hardware with limited resources1.

▼ **Bitsandbytes**:

- **Bitsandbytes** refers to a specific technique used within PEFT to further optimize memory usage during fine-tuning.
- It involves **4-bit quantization**, which reduces the precision of model weights. Here's how it works:
  - **Quantization**: Normally, model weights are stored as 32-bit floating-point numbers (FP32). In 4-bit quantization, we represent weights using only 4 bits (hence the name). This reduces memory requirements.
  - **Sign, Exponent, and Mantissa**: Each 4-bit weight is divided into three parts:
    - The **sign bit** represents the sign (+/-).
    - The **exponent bits** determine the base (e.g., 2 raised to the power of the integer represented by the bits).
    - The **fraction or mantissa** is the sum of powers of negative two corresponding to each bit that is "1" (active).
    - ***Benefits*** : By employing 4-bit quantization, we can effectively load models, conserve memory, and prevent machine crashes, especially when working with large-scale models like the OPT-6.7b (6.7 billion parameters) in resource-constrained environments23.
    - URL - https://huggingface.co/blog/4bit-transformers-bitsandbytes
      - To understand know more about **Making LLMs even more accessible with bitsandbytes, 4-bit quantization and QLoRA**

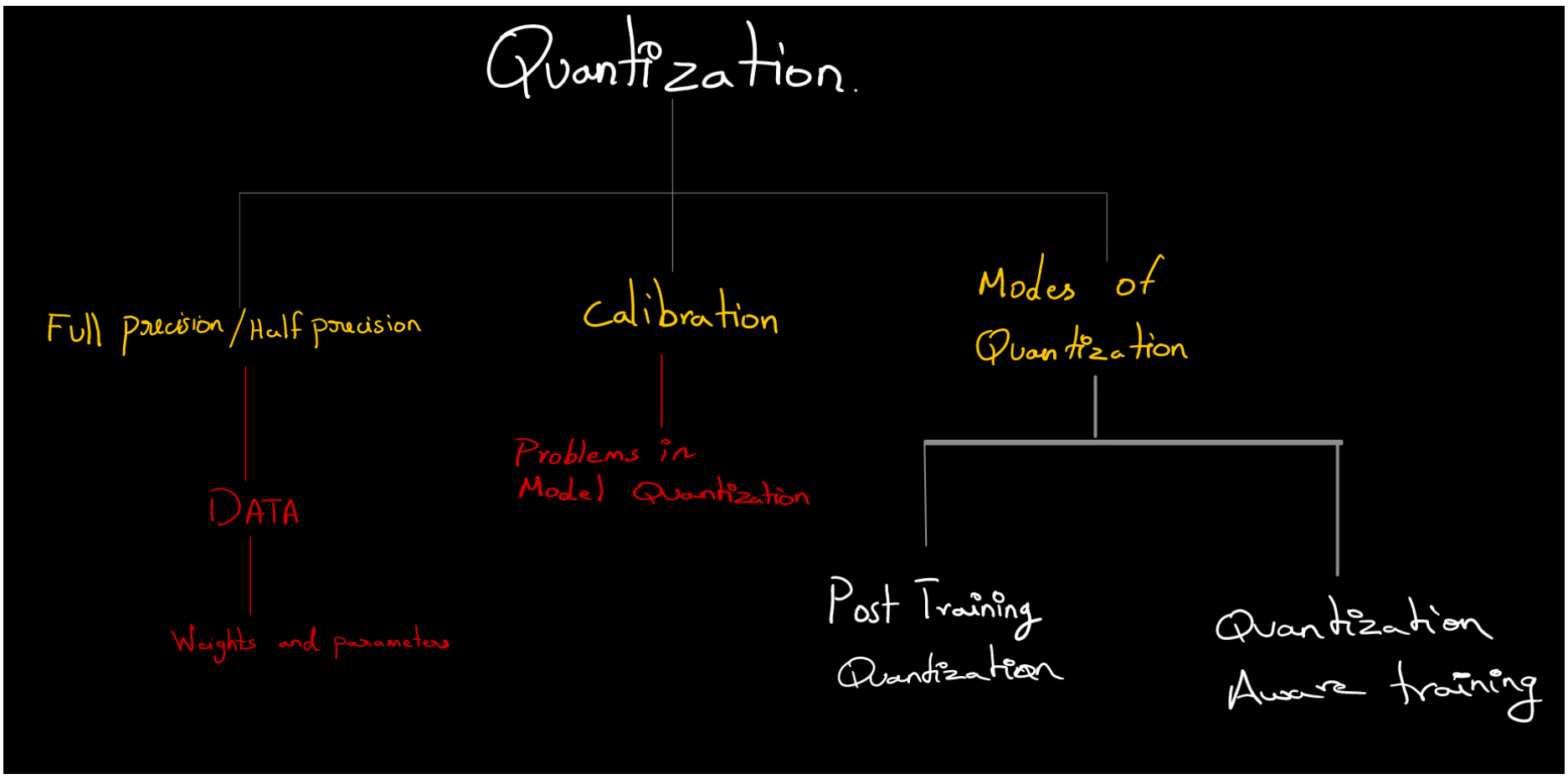
In summary, accelerators and bitsandbytes enhance the efficiency of PEFT by reducing computational costs, storage requirements, and memory usage, making large language models more accessible and practical for various downstream tasks.

Fine Tuning Quantization InDepth Intuition.

Introduction

This document delves into the essential concept of quantization, a technique used to optimize the memory footprint and inference speed of large language models (LLMs) like ChatGPT. We'll explore the nuances[’it means the detailed and often delicate aspects of understandin’] of Full Precision, Half Precision, and how Quantization enables efficient deployment on diverse hardware. Additionally, we'll discuss the crucial role of Calibration and the two primary quantization modes: Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT).

Quantization.



Detailed Examination :

QUANTIZATION : conversion from Higher memory formate to a lower memory formate.

Full Precision / Half Precision : is somthing related to data types , How data is been stored in the memory,

In LLM models its about weights and parameters, Bcz llm are deep learning neural networks in the form of transformers or bert.

Calibration : in Model Quantization their will be some problem how we can do calibration

Their are 2 types of **modes to quantize.**

---

1. Full Precision / Half Precision.

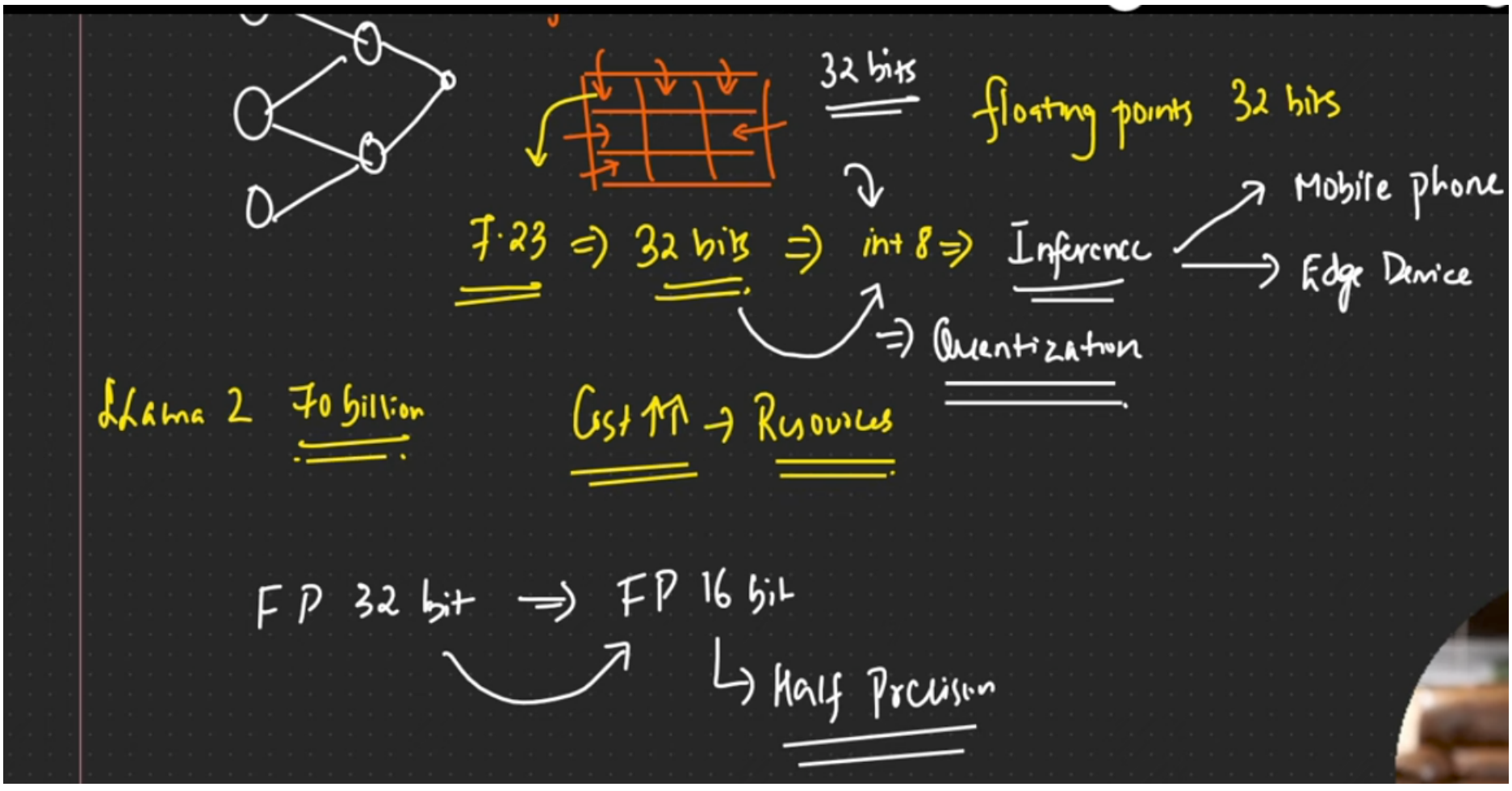
1.1 - Understanding Precision Levels

In the realm of deep learning, weights and parameters within neural networks are often represented using floating-point numbers. These numbers can be stored with varying levels of precision, which significantly impacts memory usage and computational efficiency.

Full Precision (FP32): This format utilizes 32 bits to store each value, providing the highest level of accuracy but also consuming the most memory.

Half Precision (FP16): This format uses 16 bits per value, offering a balance between accuracy and memory efficiency. Many GPUs today support FP16 operations, making it a popular choice for training and inference.

Detailed Examination :



Any neural network or any data, when train neural network what parameter probabilly involved over here is **Weights,** Now Weights are usually in the form of **Matrix.**

in that matrix eg, we have 3×3 matrix, every value is probabilly stored in the memory in the form of **32bits,** we can aslo say it as floating point 32 but **FP** as diff meaning

we also denote it as **FP32.** we can also consider it as **Full Precision / Single Precision.** (in Short it is like Folating Piont number)

1.2 - The Memory Challenge of Large LLMs

Large language models (LLMs) like LLaMa 2, with their billions of parameters, pose a considerable memory challenge. These parameters, encompassing weights and biases, must be stored in memory, leading to massive storage requirements.

Standard Hardware Limitations: Standard GPUs and systems with limited RAM struggle to accommodate these memory demands, rendering direct fine-tuning or inference on such systems impractical.

Cloud Solutions (but at a Cost): While cloud resources can address this limitation, they come with a significant cost associated with the computing power and storage needed.

Detailed Examination :

eg. i have **number 7.23** is stored based on **32bits** in memory.

Understand when you have very big neural network or LLM models parameters keeps on increasing.

consider LLama 2 model with 70 billion parameters. it as 70b parameters in term of weights and bias, now it’s not possible to around do a fine tunning w.r.t the normal GPU or very limited RAM in system

cannot directly download the specific model and put it in my ram or load it in Vram,

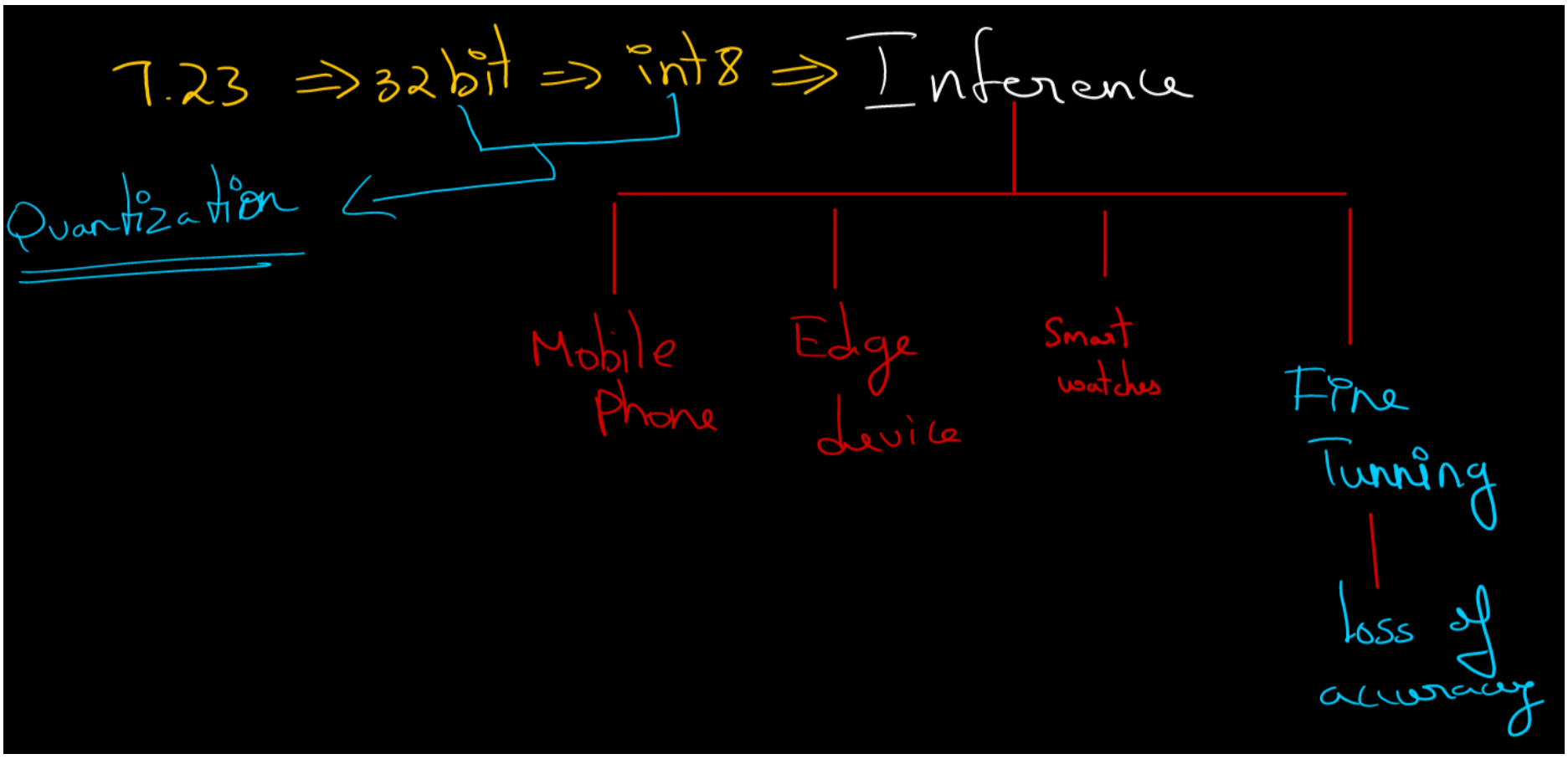
we can use cloud space but it will have lot of cost involved based on the resources, what we can do.. ?

1.3 - Quantization: Reducing Memory Footprint

Quantization provides a solution for overcoming the memory constraints associated with LLMs. It involves converting the high-precision floating-point numbers (FP32) used for weights and parameters to a lower-precision format, typically 8-bit integers (INT8).

Benefits: This conversion drastically reduces the memory footprint of the model, making it feasible to store and deploy it on less powerful devices like mobile phones, embedded systems, and even personal computers.

Trade-Off: Potential Accuracy Loss: However, this conversion can lead to some loss of accuracy due to the reduced precision. It's crucial to weigh the trade-off between memory savings and potential accuracy loss when choosing a quantization method.



we can covert this **32bit into int8 and then download and use the model**,with in my system i will be able to Inference it. for fine tunning a new data set i would need Gpu. If i consider w.r.t **Inferencing** it becomes quite easy all my value that are stored in **32bits it will be stored in the form of 8bits** what we are specifically doing is we are **converting from high memory formate to low memory format** and this is what called as **Quantization.**

a very important thing with quantization we will be able to inference it quickly.

▼ **1.4 - Inference**

1.4.1 - Inference in LLMs (Large Language Models):

Definition: Inference refers to running a trained language model on new data (usually in the form of a prompt,Question, input) and generating an output (such as a response, prediction, translation).

Explanation: Imagine you have a well-trained language model like ChatGPT. When you give it a sentence or a question, it processes that input and produces a meaningful answer. This process of generating a response based on the input is what we call inference.

This is how we leverage the knowledge encoded in LLMs for real-world tasks.

1.4.2 - Why Is Inference Important?

Inference allows LLMs to be useful in real-world applications. Once a model is trained, its true value lies in its ability to provide relevant and accurate outputs when faced with new data.

Inference allows us to apply the specialized knowledge acquired during fine-tuning to real-world tasks.

It ensures that our LLM performs well on specific problems, such as question answering, translation, or sentiment analysis.

Example : Suppose you input the prompt: “Translate the English sentence ‘Hello, how are you?’ into French.”The LLM would perform inference by processing this prompt and generating the translated output: “Bonjour, comment ça va ?”

1.4.3 - Why Quantization Helps with Inference?

Quantization significantly accelerates inference by reducing the amount of data that needs to be processed during calculations. This is particularly beneficial on devices with limited computational resources.

1.4.4 - Inference During Fine-Tuning:

Fine-Tuning

Fine-tuning is a crucial step in training large language models (LLMs) like ChatGPT. Here's how it works:

- First, a pre-trained LLM (such as GPT-3 or BERT) is created using a massive amount of text data.
- Then, this pre-trained model is fine-tuned on a smaller, domain-specific dataset to adapt it to a specific task or application.
- Fine-tuning involves training the model further on task-specific data, adjusting its weights and parameters to specialize it for a particular purpose.

When we fine-tune a language model, we use it for inference on the task-specific data.

During inference, the fine-tuned model processes input examples (such as sentences or prompts) and produces relevant outputs (such as predictions or answers).

For example, if we fine-tune a model for sentiment analysis, inference involves inputting a sentence and predicting whether it has a positive or negative sentiment.

1.4.5 - Fine-Tuning vs. Pre-training Inference:

Pre-training (the initial training on a large corpus) and fine-tuning (the specialized training) both involve inference.

However, **during pre-training, the model learns general language patterns**, **while during fine-tuning, it adapts to specific tasks or domains.**

1.4.6 - When to Use Inference?

- Use inference whenever you need your fine-tuned LLM to process new data and provide meaningful outputs.
- For example, if you've fine-tuned an LLM for sentiment analysis, inference helps classify new sentences as positive or negative.

1.4.7 - Uses of Inference in LoRA and QLoRA

- **Question Answering**: Given a question, the fine-tuned LLM can infer an accurate answer.
- **Text Generation**: Inference generates coherent text, summaries, or translations.
- **Classification**: It predicts labels (e.g., sentiment, topic) for input data.

In summary : inference in LoRA and QLoRA fine-tuning ensures that our LLM performs effectively on specific tasks by generating meaningful outputs based on new input data.

Detailed Examination **:**

Let's say i have a LLM model it i give any i/p to that i will be able to get the o/p (Response), now when i give any i/p all the calculation w.r.t different weight will happen, when i have bigger GPU this Inferencing will happen quickly. If i have GPU with less core's the calculation will take time

If i convert my 32bit to 8bit now every weights are basically converted to 8Bits

Now the calculation their will be difference it will happen much more quicker

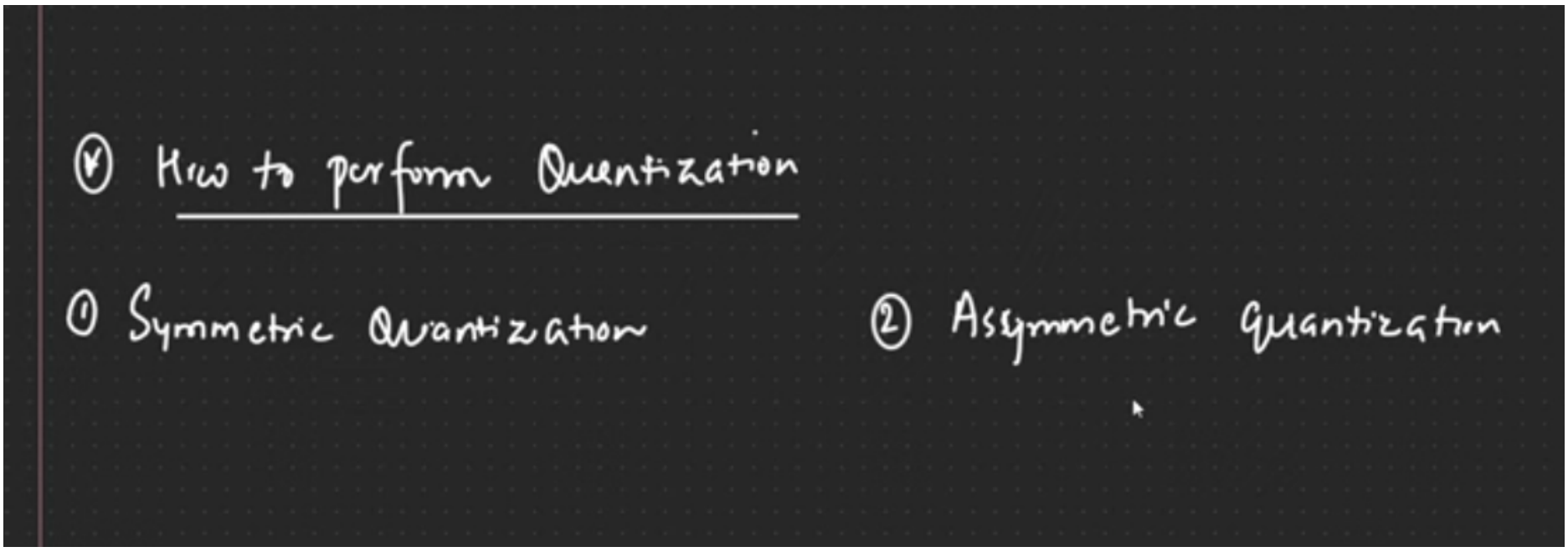
So Quantization is very much important for inferencing

This Inferencing we can use it in phone,watches etc. now with the help of quantization we can perform finetunnig their is a disadvantage

when we perform this quantization since we convert 32bits to int 8 their is some loss of info also and because of this we have loss of accuracy, their are diff techniques that we can overcome on it

---

Quantization Techniques: Symmetric vs. Asymmetric



Detailed Examination :

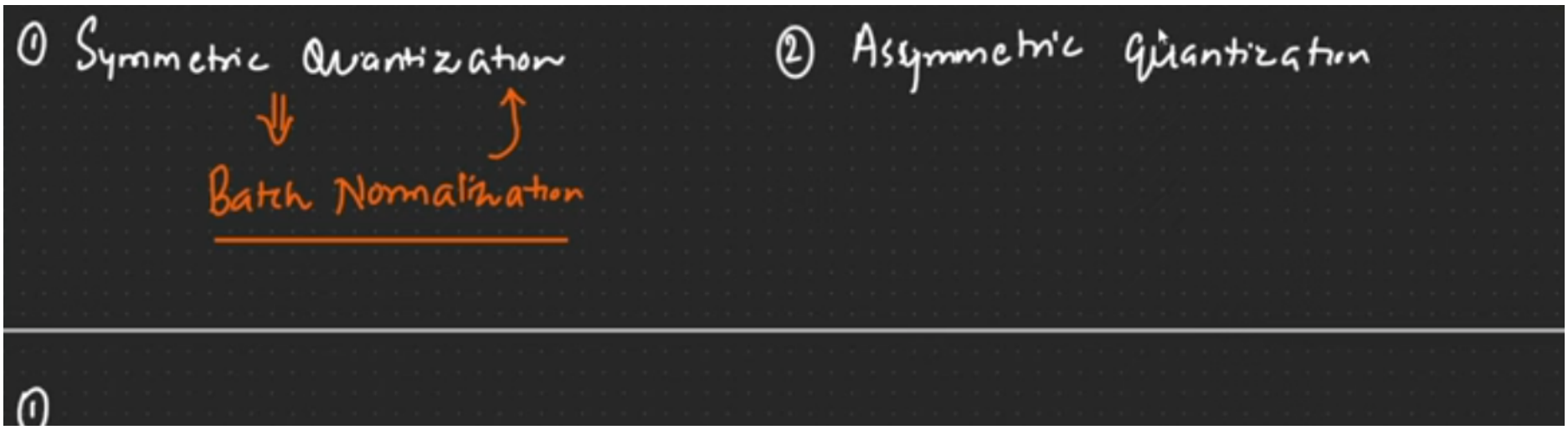
Lets go a head and try to understand how to perform quantization, Two primary techniques are used for quantization

1. Symmetric Quantization
2. Asymmetric Quantization

see, with the help of tensor flow and 4 lines of code we can do this

but its necessary to know that we can do it manually

1. Symmetric Quantization



Concept : In symmetric quantization, the range of values to be quantized is centered around zero, with equal positive and negative ranges.

Example : If we want to convert values in the range of -1000 to 1000 to 8-bit integers (0 to 255), the symmetric approach would be to map -1000 to 0, 0 to 127, and 1000 to 255.

Benefits : This method is often simpler to implement and can provide good results in situations where the data distribution is relatively centered.

Detailed Examination :

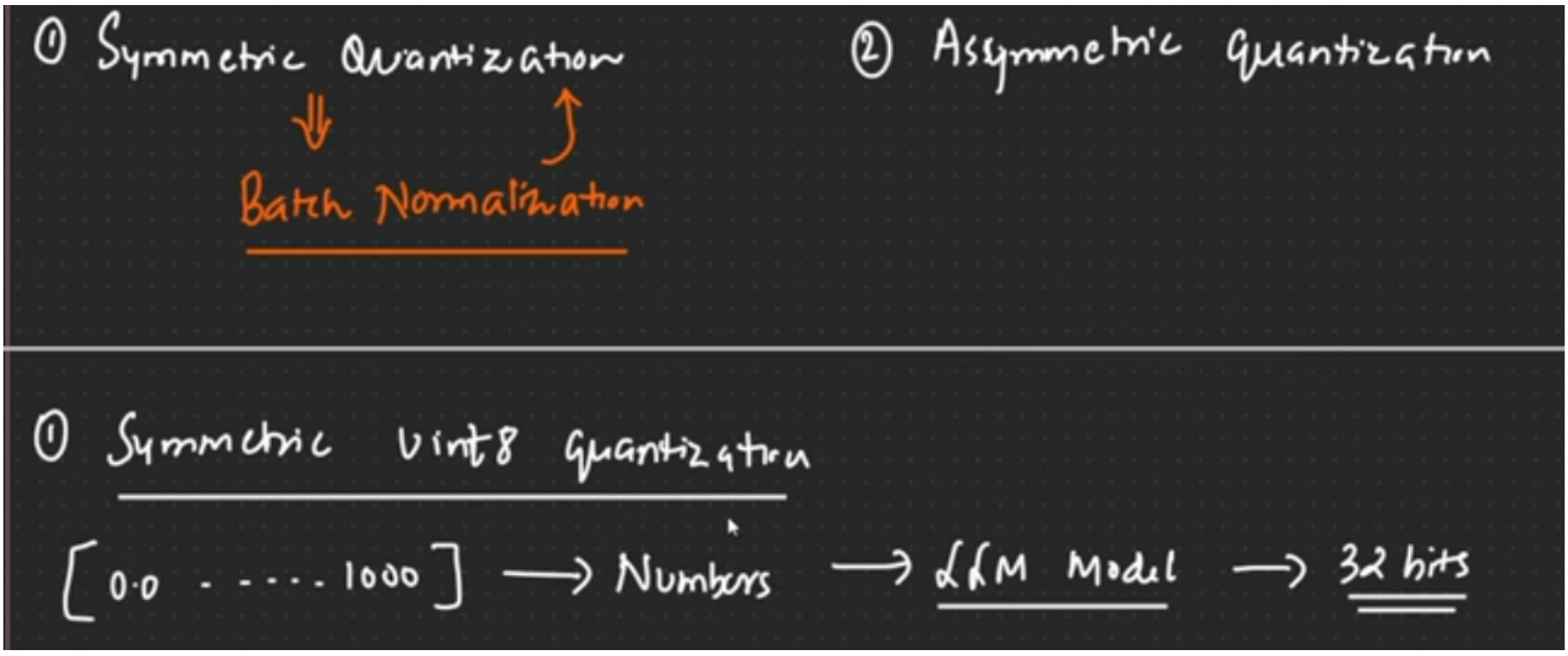
lets say their is a task in this 1st task is r.w Symmetric quantization.

In **Deep Learning** their is **Batch Normalization,** this Batch_normalizer is a technique of symmetric quantization

every time we will be able to see that during **forward and backward propagation** between all the layers we apply batch normalization, so that all our weight are **Zero Centered [Entier distribution of weight is centered near Zero]**

Symmetric Unsigned int8[Uint8] Quantization

Unsigned int8[Uint8]: Uint8 means it will not take any negative number it will be between 0 to 255.



Detailed Examination :

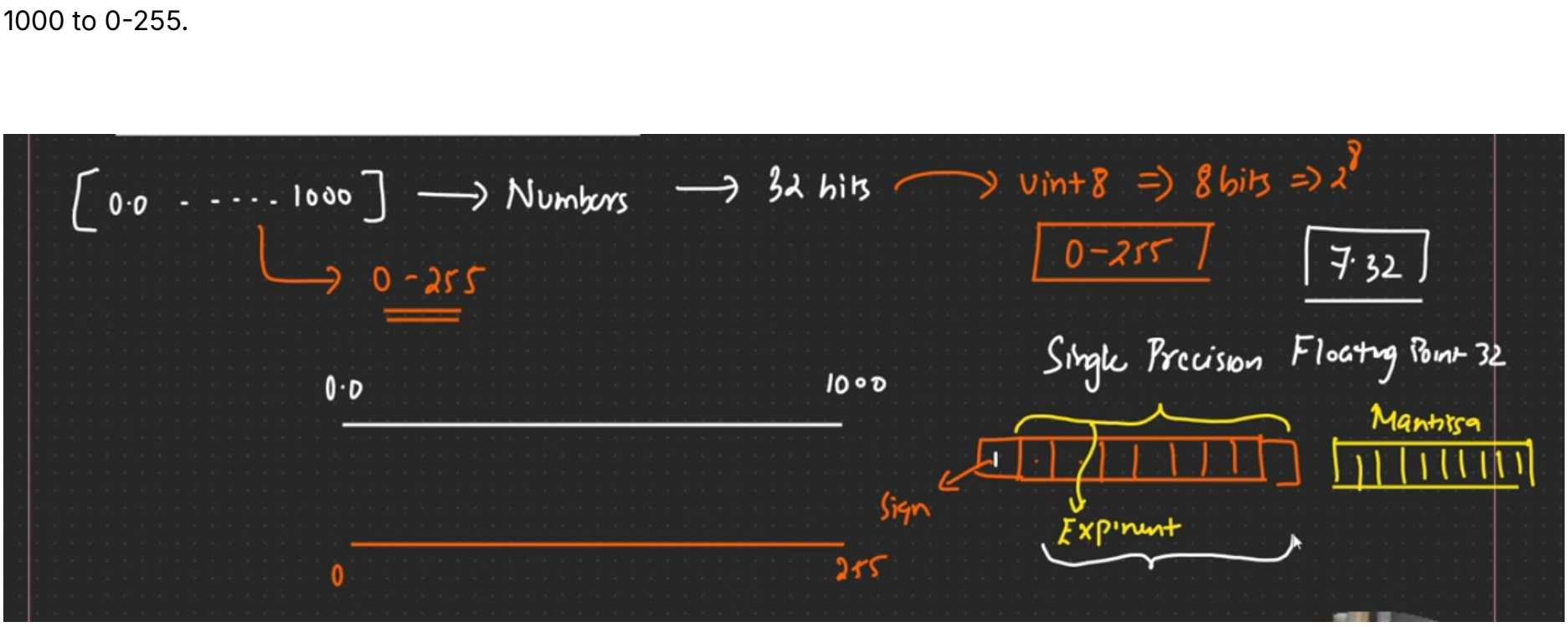
Lets say I have a floating point number between zero to 1000 → weights (Numbers)

and this are the weight for my larger model any LLM Model you have lot of parameters

and all the Numbers [weight] are stored in 32bits and the weights will not be in that range it will be in a minimalistic weights

This numbers are stored in form of 32 bits now main aim is to convert this into Unsigned int8 - [uint8] that basically means 8 bits which 2^8= 256

But when we say U - Unsigned that basically means my value will be ranging from 0-255. So i want to quantize from the 0-1000 to 0-255.



important thing. If we have this Single precision floating point 32. you have a number how that stored.

the first one bit is specifically used for sign or unsign value('+', '-')

then next 8 bit are stored for exponent and remaining 23 bits will basically will save for mantissa(this is specifically for fraction)

▼ **Mantissa** (also known as the **fractional part** or **significand**):

- In scientific notation, especially when representing floating-point numbers, the mantissa refers to the part after the decimal point.
- For example, in the number

  $$12345.6789.$$

  The mantissa is the part after the decimal point (in this case, 0.6789). It holds the significant digits that give the number its value.

  In binary representation, the mantissa is the fraction part of a floating-point number, typically stored in a fixed number of bits (such as 23 bits for single-precision floating-point numbers).

Exponent:

- **Definition**: The exponent represents the power to which a base (usually 2 or 10) is raised.
- **Simple Words**: Think of it as a "zoom factor." If you have a small number like

  $$0.00123$$

  , the exponent (say,

  $$-3$$

) tells you how many times to multiply it by

  $$10$$

  . So,

  $$0.00123 \times 10^{-3}$$

  Putting it all together, when you write a number like

  $$3.14 \times 10^{2}$$

The **mantissa** is (the significant part).

$$3.14$$

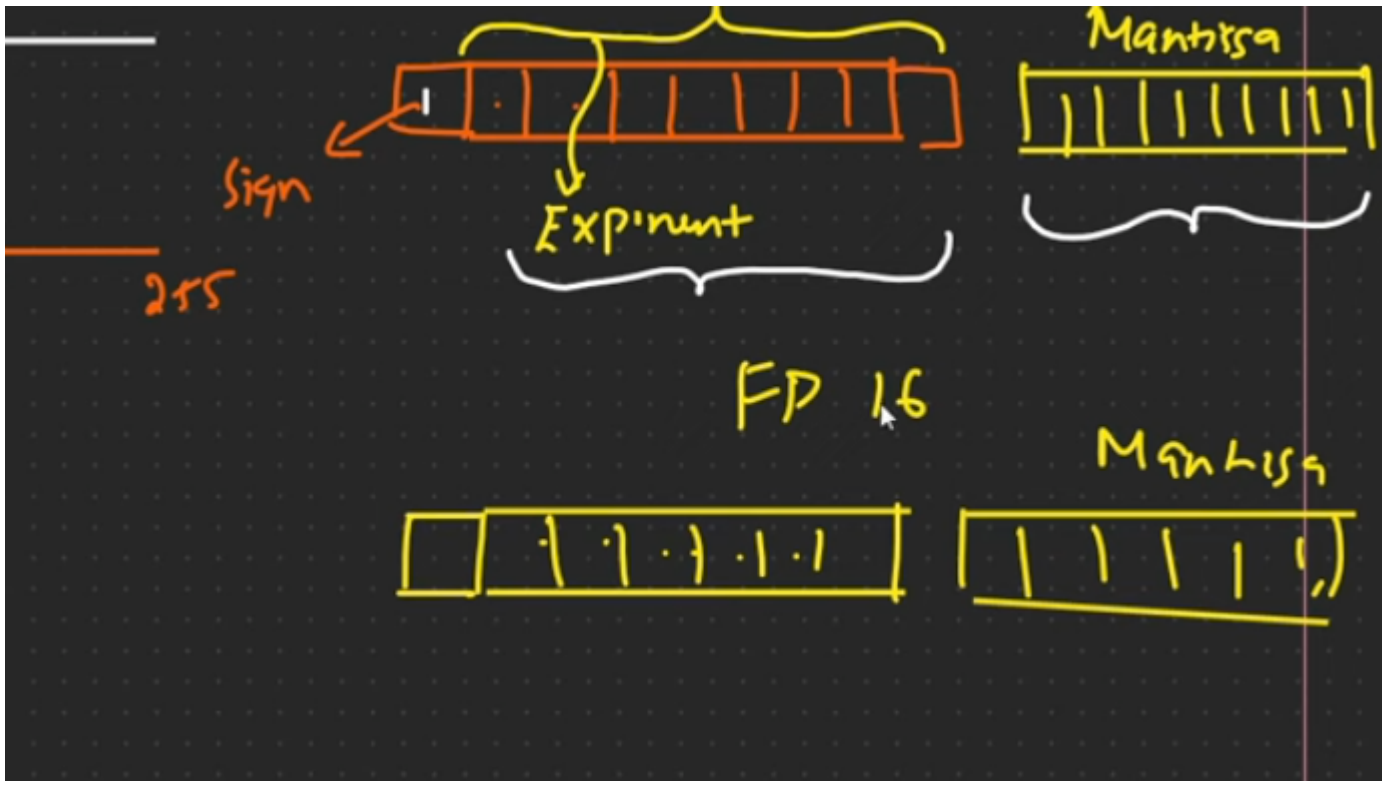
The **exponent** is (the zoom factor).

$$2$$

💡 mantissa holds the significant digits of a number, while the exponent determines its scale. Together, they form the scientific notation or floating-point representation. 🧱✨

e.g. if I have a number 7.32 it in a positive number, so the sign bit that will be a +ve value and then the 7 will probabilly stored in the 8 bit and remaining .32 will be put up in mantison **[we basically say this as fraction any number that comes after the decimal]** .

This is how the numbers are basically stored in memory.



lets see with FP16 → Half Precision floatingpoit 16bit

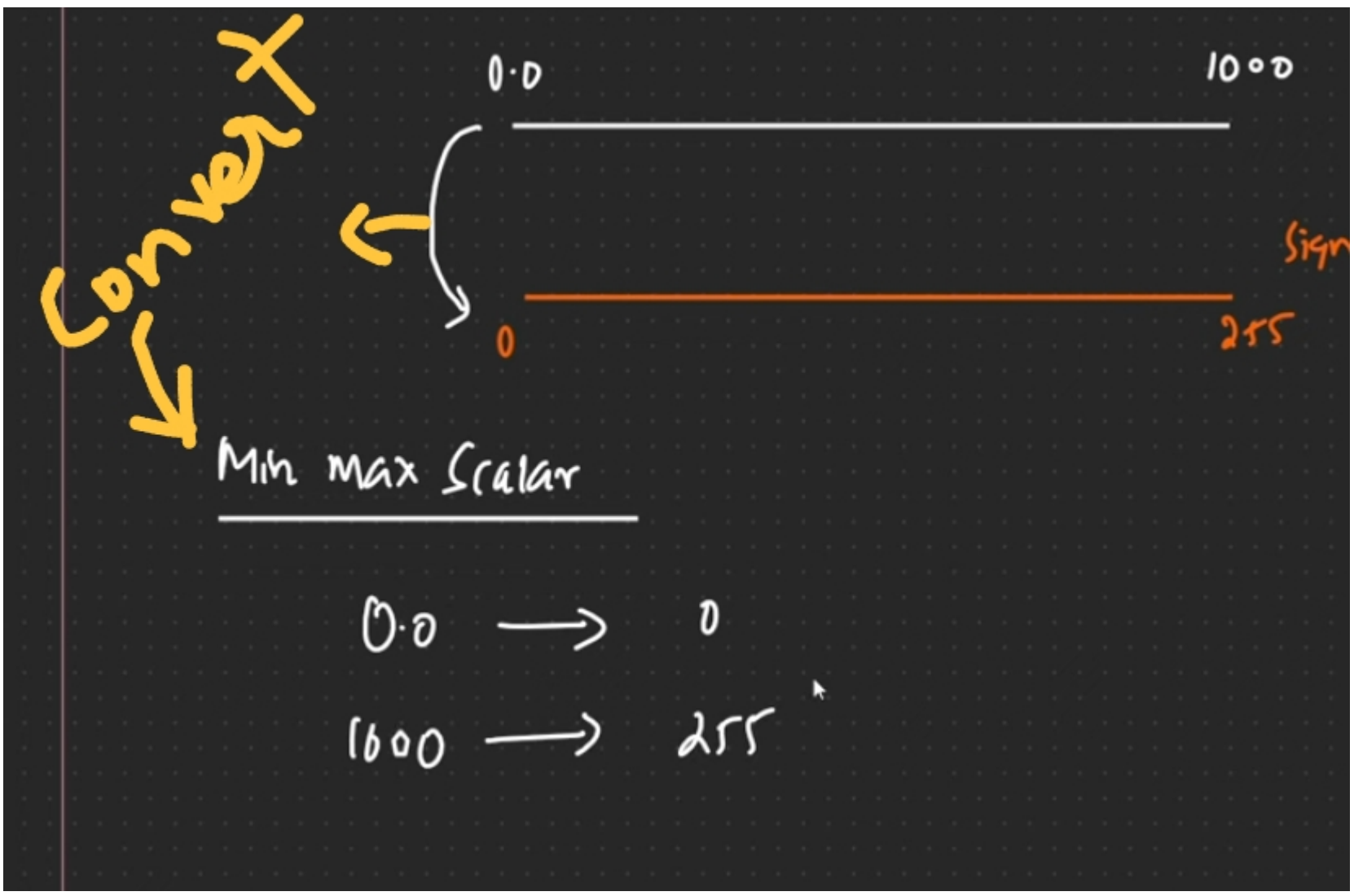
their will be 1 bit for Sign number, then 5 bit for exponent and remaining 10 bit will be saved w.r.t mantison.

know that FP16 takes less memory and FP32 has high memory

What is our main AIM here ?

- We have 32bit number
- It need to convert to range of Unsigned int8 [U int8 - means it will not take any negative numbers, it will be between 0 - 255 (256)]
- This is what we really want to do

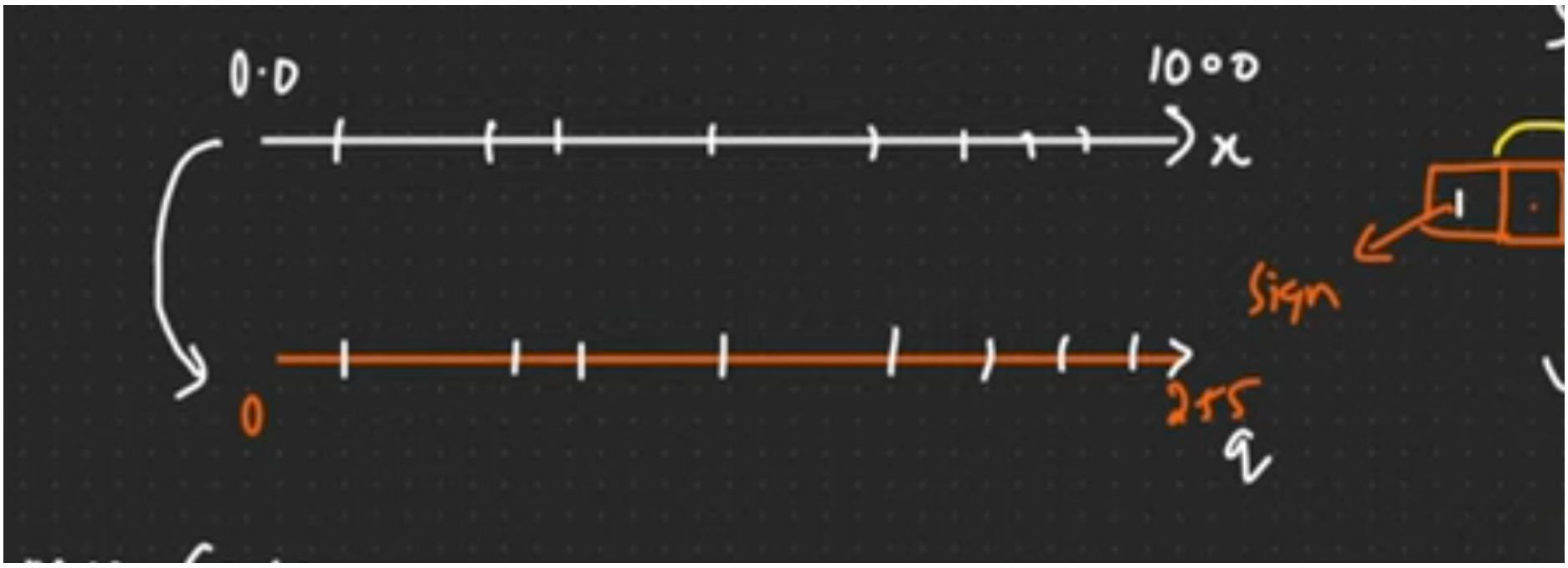
Now for this what will be the equation.



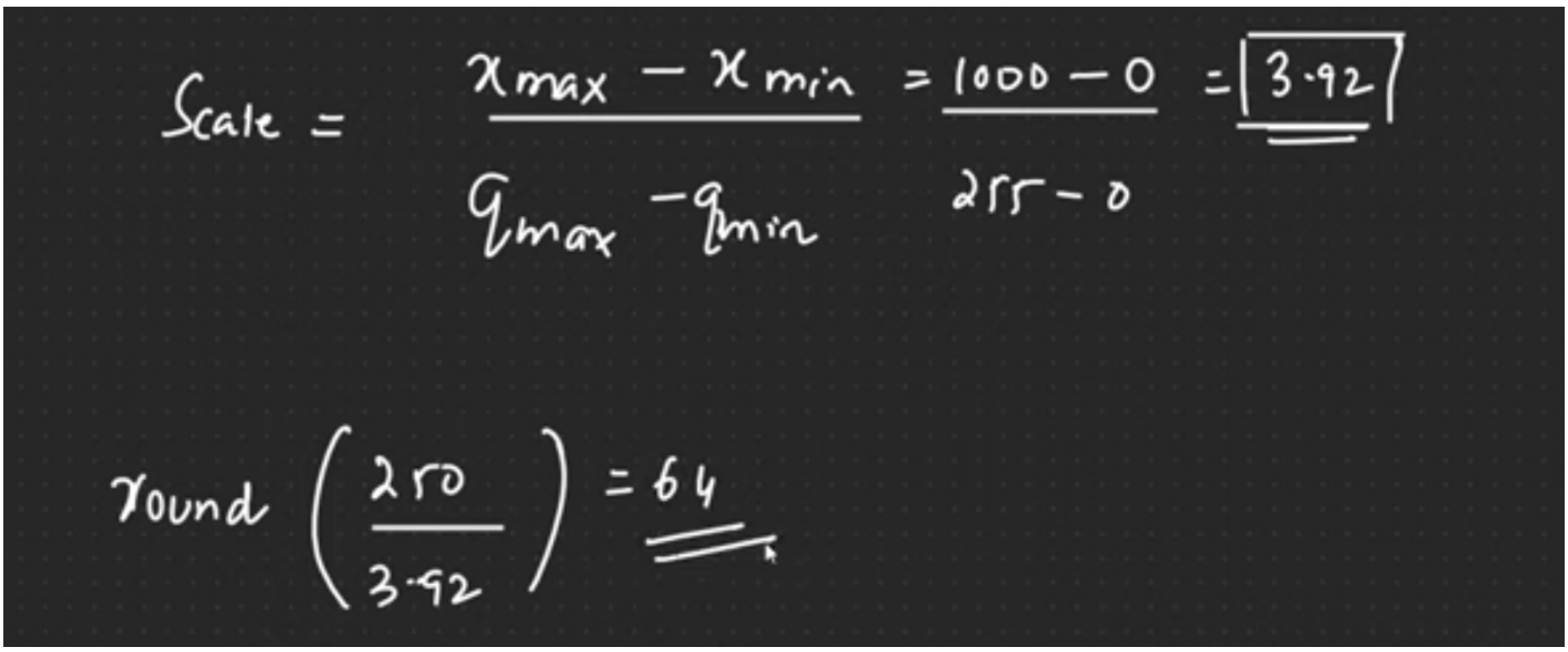
MIN Max scalar from any number from 1000fp to 255fp

The bits are decreasing from 1000-255 **Quantization basically happening**

we probabilly have to come up with the scale factor



here 1000 is my Xmax and 255 is my Qmax, showing **Quantization happens in symmetric distribution [Symmetrics basically means all the data are evenly distributed]** .



Here scale formula will be that scale

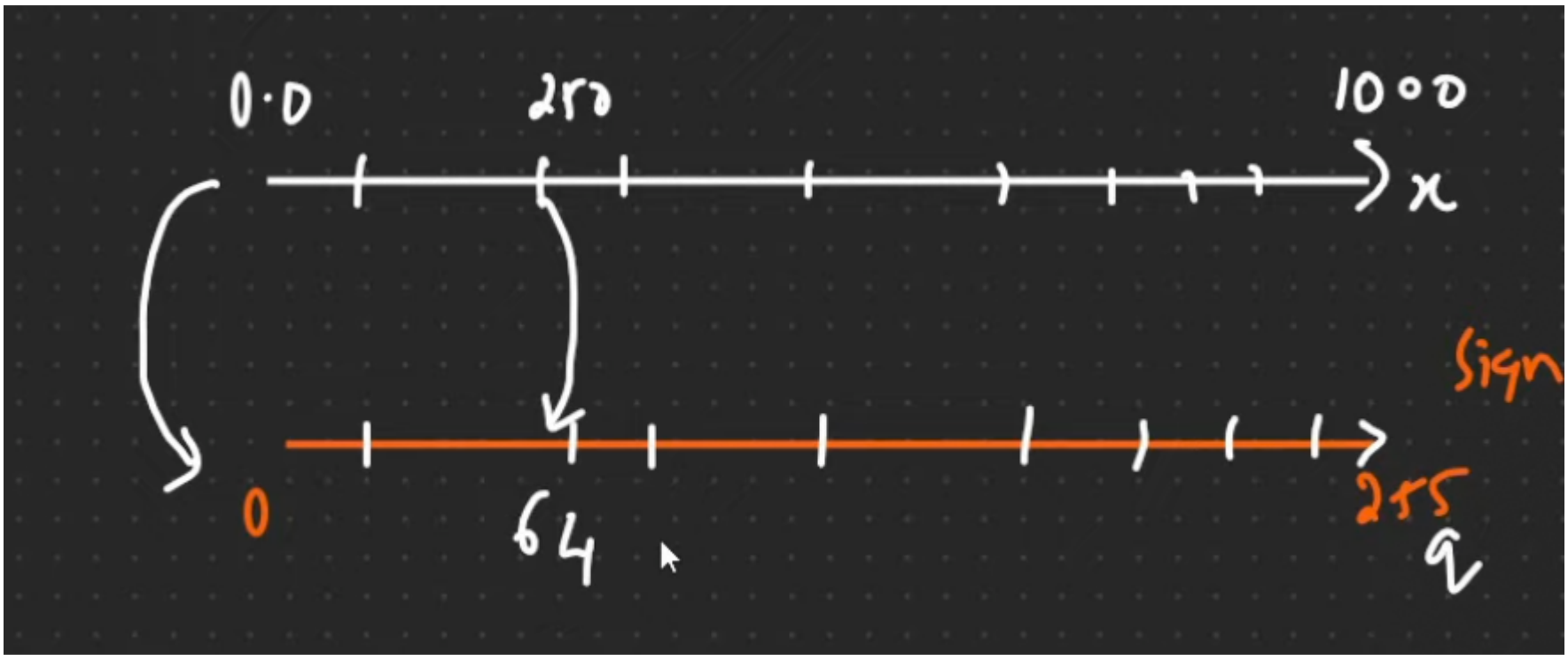
if i go with this specific division the value that i will be having is 3.92 which is

Scale Factor.

Any number that i have over xmax if i want to convert it from FP32bit to Uint8 i just need to use this scalling along with formula round

When i round with any member between xmax lets consider 250

250/3.92 = 63.77 if i do rounding it will be 64



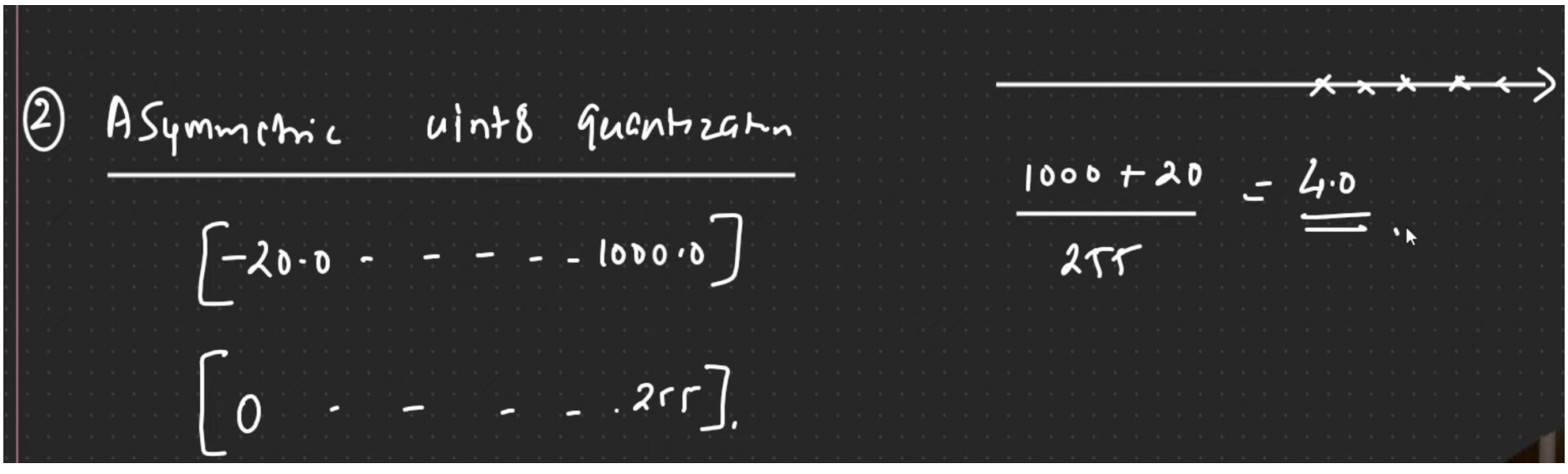
in short any number over Xmax let say 250 this will get convert to quantize value of 64. The same thing the code will be also doing and this for symmetric Uint 8 quantization

2. Asymmetric Quantization

Concept : Asymmetric quantization allows for a more flexible mapping of values, especially when the data distribution is skewed (not centered around zero).

Example : If we have values in the range of -20 to 1000, we might map -20 to 0, 0 to 127, and 1000 to 255. This means the positive range is expanded to cover more values than the negative range.

Benefits : This method can be more effective when dealing with data that has a clear bias towards positive or negative values.

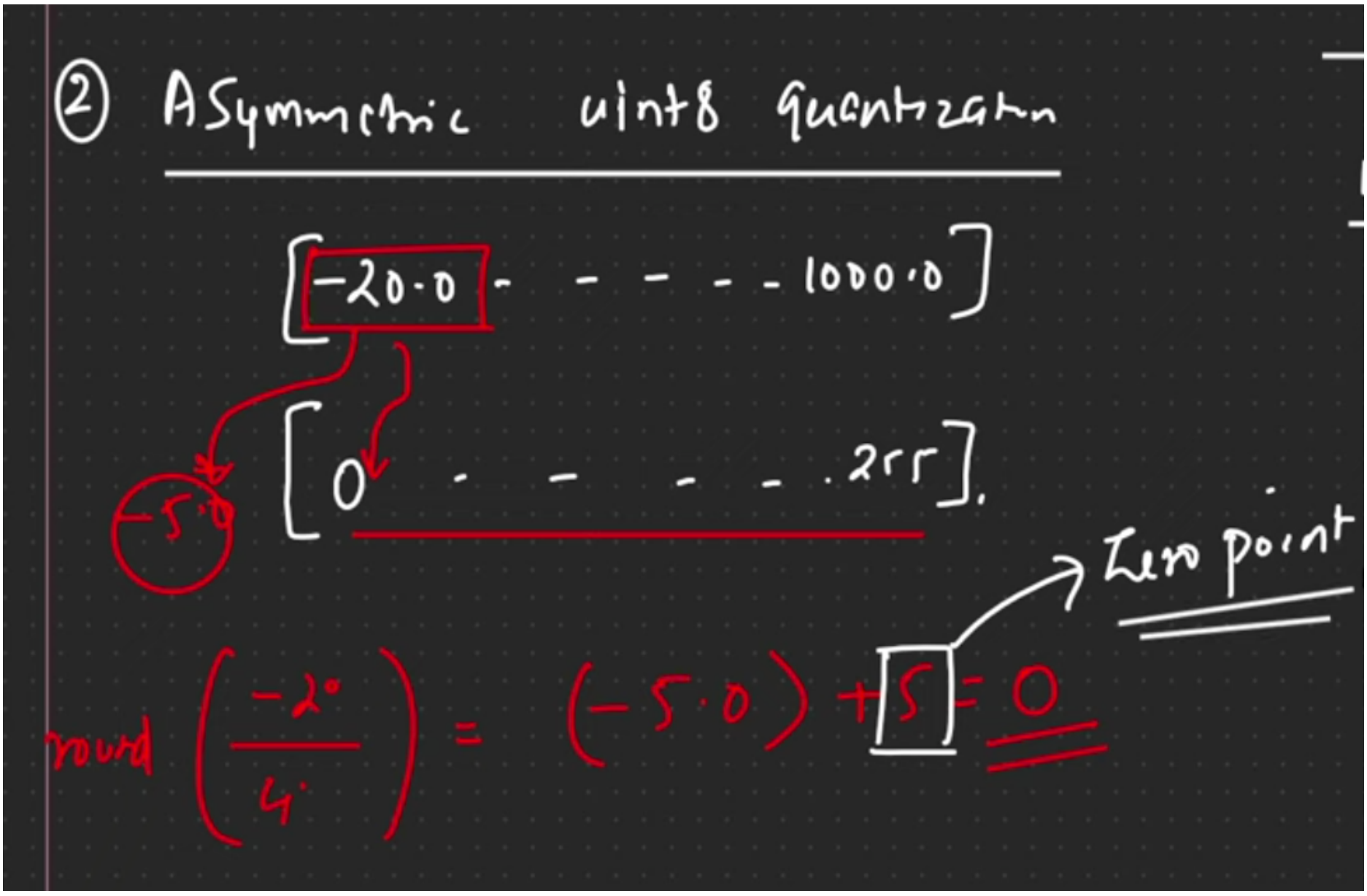


Detailed Examination :

If i want to perform this quantization lets say value lies between -20.0 to 1000 these are my floating point. now i have to perform quantization and convert this in to 0 to 255

in case of asymmetric, what happen in real number section the numbers are not really distributed it may be right or left sqewed

in this scenario my value ranges between -20 to 1000 and i need to convert this from this 0 to 225. here if same formula is been applied xmax - x min i will get scale factor 4.0.

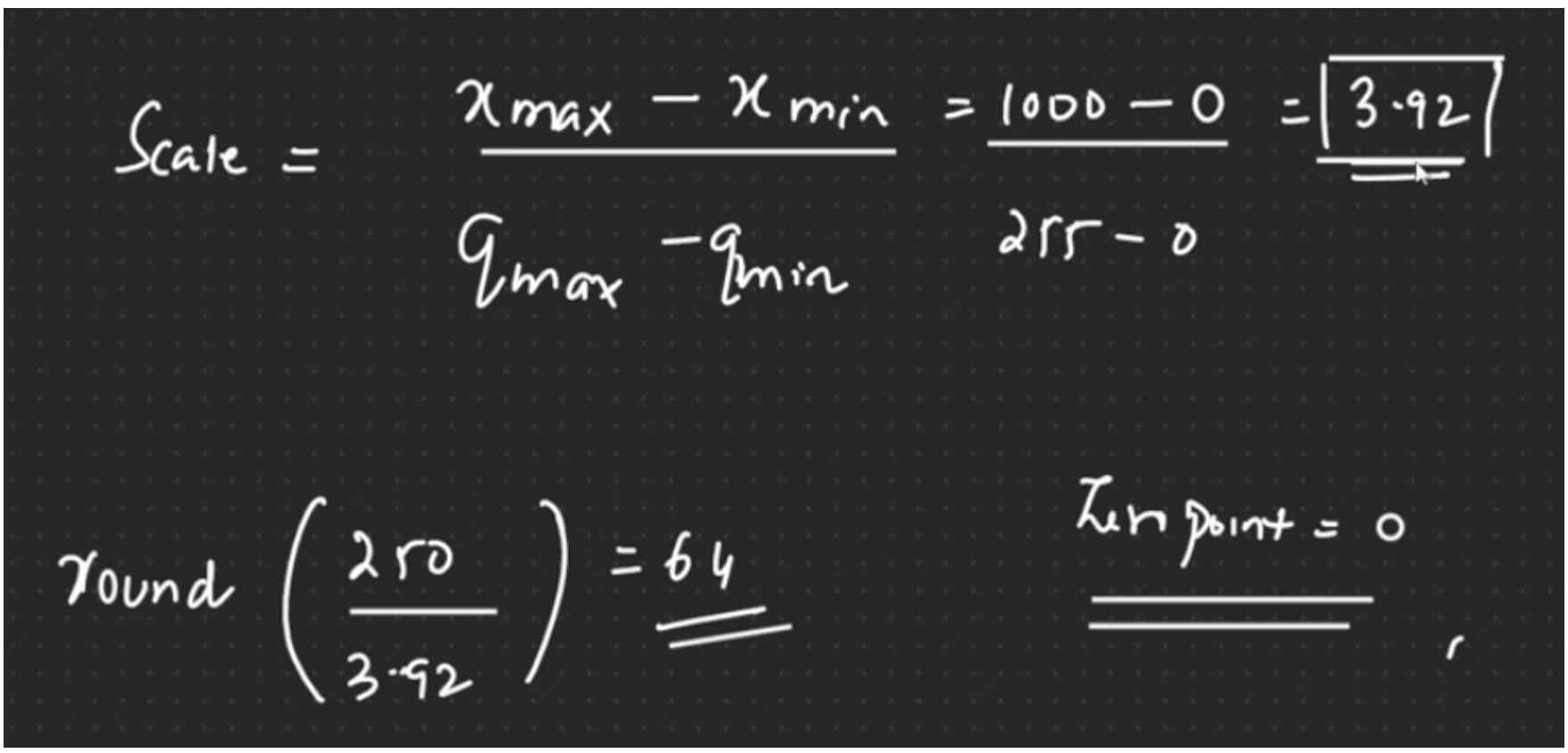


this -20.0 if i quantize it i will get -5 now you can understand this -20.0 is converted to -5.0 but you can see the distribution starts with 0 - 255

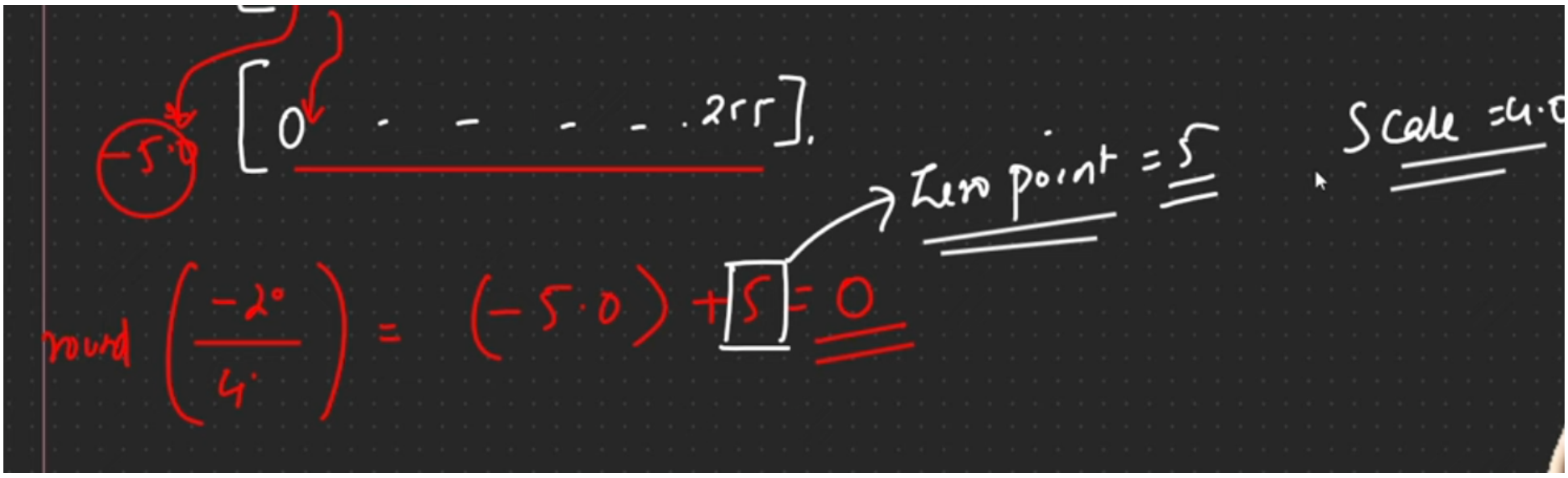
so, the thing is how can the -20.0 can forcefully make it to 0.0

we can add the same number as the answer came +5 by this we will get zero

In this case the number that we added +5 is zero point



for the above 1 which is the symmetrical distribution here the zero point was 0 only and scale was 3.92.

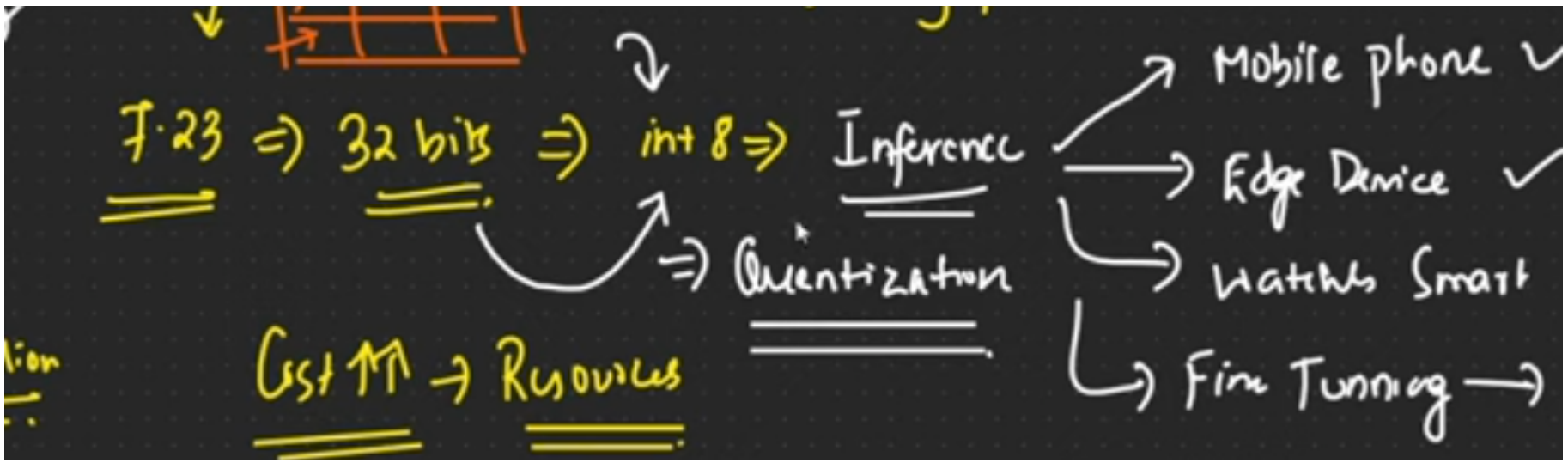


In this particular case Asymmetrical distribution here we have zero point = 5, scale = 4.0

👉🏾 **So, this 2 parameters Zero Point, Scale. we specially required for Quantization.**

2. Calibration

Calibration: Ensuring Accuracy - A Calibration is basically means how we will be able to **convert this 32bit into int8**



Calibration plays a vital role in minimizing accuracy loss during quantization. It involves adjusting the quantization process to optimize for the specific characteristics of the model and data.

The "Squeezing" Analogy: Imagine you have a range of values from 0 to 1000, and you want to squeeze them into the range of 0 to 255. Calibration helps determine the best way to "squeeze" these values while preserving as much accuracy as possible.

Mathematical Intuition: Calibration involves calculating a scale factor and a zero point that determine the mapping between the original floating-point values and their quantized integer counterparts.

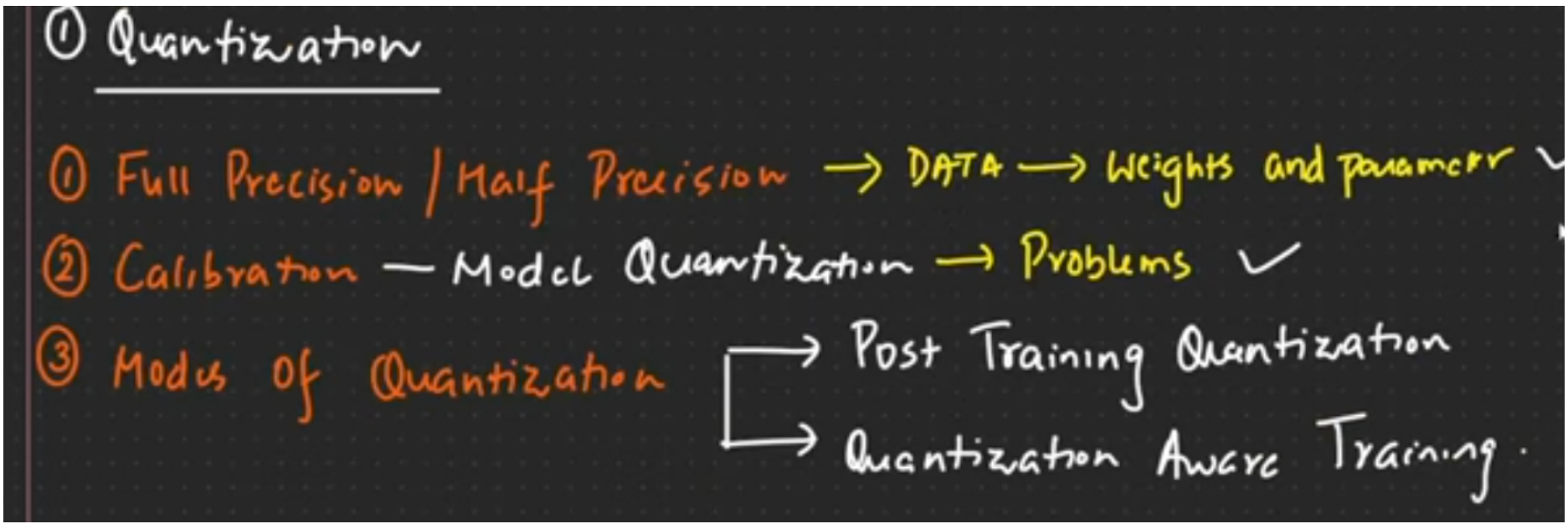
Quantization in Practice

TensorFlow: TensorFlow provides convenient functions and tools for performing quantization, simplifying the process.

Manual Quantization: While TensorFlow simplifies the process, understanding the underlying concepts and calculations is essential for fine-tuning and optimizing the quantization process for your specific use cases.

---

3. Modes of Quantization



1. Post Training Quantization
2. Quantization Aware Training

1. Post Training Quantization (PTQ).

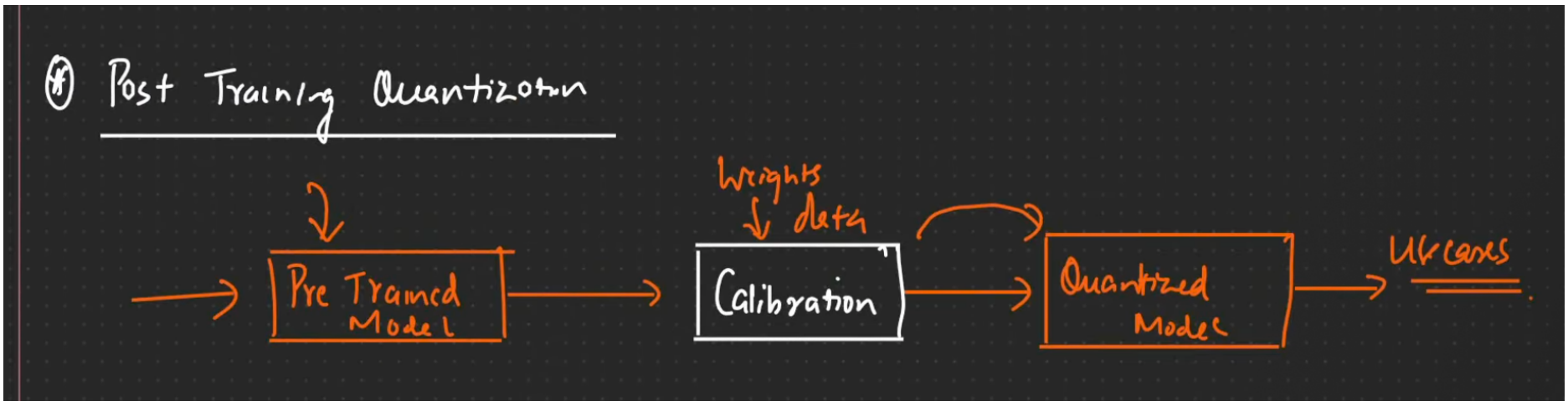
Concept : This method quantizes the model after it has been fully trained.

Process : PTQ involves applying calibration to a pre-trained model to determine the optimal mapping for quantization. This calibrated model is then converted into a quantized model, which can be used for inference.

Benefits : PTQ is a simpler approach, as it doesn't require retraining the model.

Limitations : PTQ can lead to greater accuracy loss compared to QAT because it doesn't account for quantization during the training process.

Detailed Examination :



Here we have a pre trained model , if i want to use it, yes the weights will be high.

we apply Calibration (Squeezing the value) from higher to lower formate and then after performing the calibration, we take the weights data. what ever data is their in the pre-trained model we convert this in to a quantized model by applying this calibration process

then we can use this entier model for any use cases. This is a simple mechanism w.r.t Post Training quantization

whats happening : (Have a pre trained model were weights are fixed. i don’t need to change those weights, i will take this weights data, apply calibration and convert this in to a quantized model)

2. Quantization Aware Training

Concept : This method involves training the model with quantization in mind. It adjusts the weights during training to optimize for the quantized format.

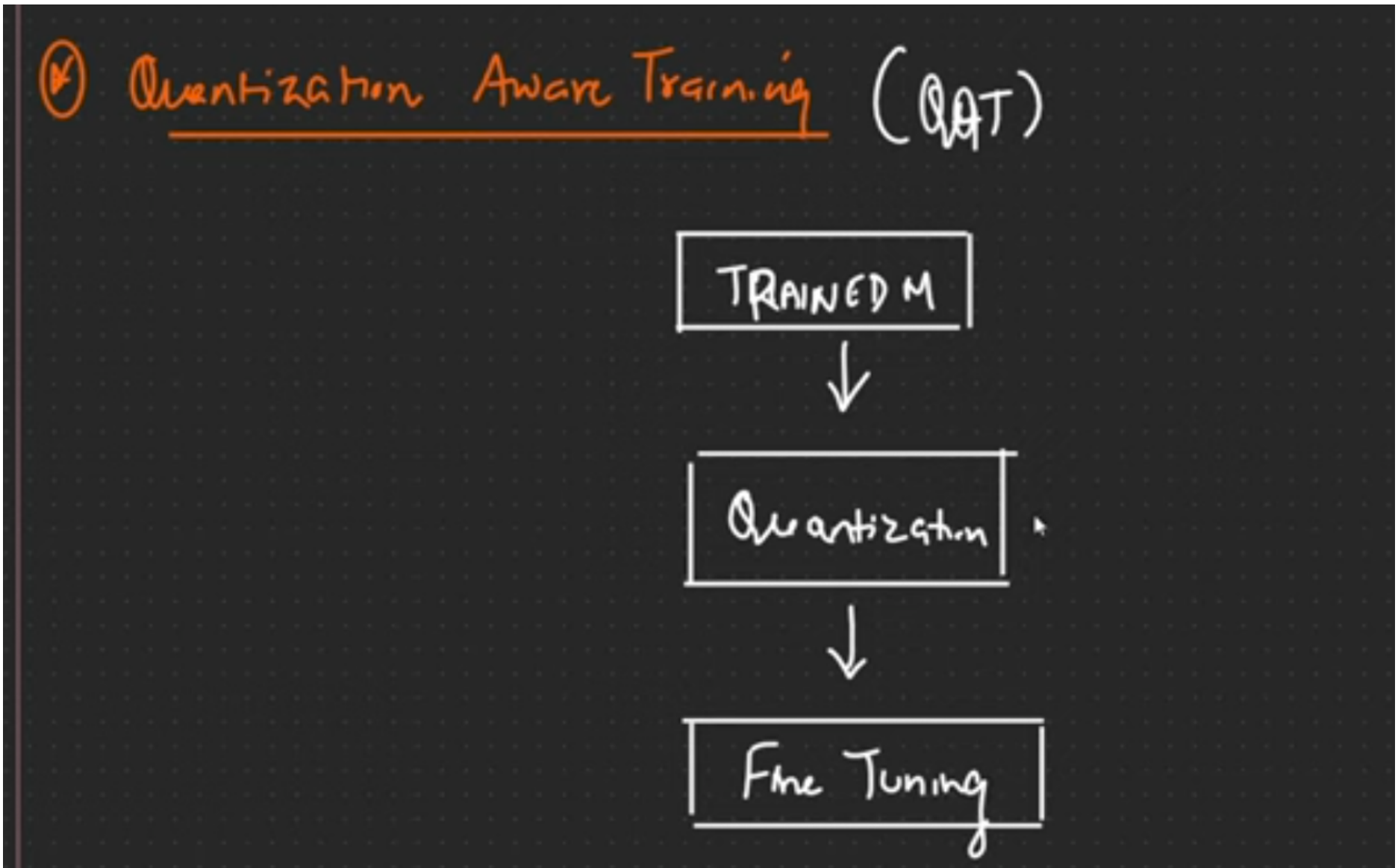
Process : QAT involves introducing quantized operations during the training process, allowing the model to adapt to the lower precision. This results in a quantized model that has been trained to minimize accuracy loss.

Benefits : QAT can lead to better accuracy than PTQ because it accounts for quantization during the training phase.

Considerations : QAT requires additional training time and resources.

Detailed Examination :

over there, the problem is if i perform calibration and create a quantized model their is a **loss of data because of this the accuracy will also decrease** for any use cases



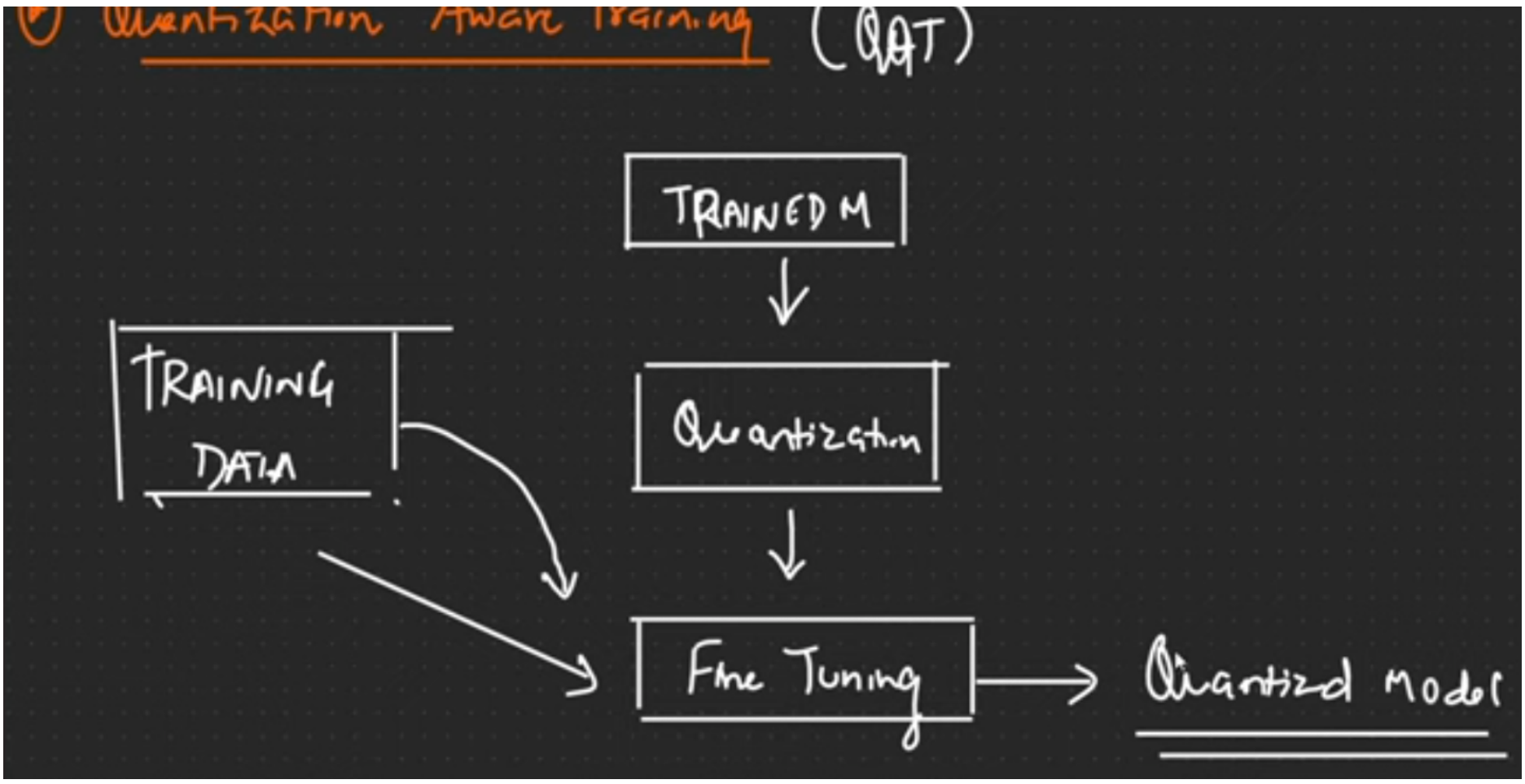
In case of QAT:

We will be taking our train model what ever the train model we use

Perform quantization = same calibration process apply.

Next step we will perform fine tuning.

We know that w.r.t PTQ some loss of data and accuracy will be thier



Here, w.r.t fine tunning we will take new training data. once we specifically take new training data we will be able to fine tune this model and then we create a quantized model

W.r.t to any fine tunning we dont use Post Training Quantization PTQ, we specifically use QAT_Technic.

That basically means we are just not losing data over here we are inturn adding more data for the training purpose through this we will fine tunning our data and then we create our quantized models.

This is the basics difference, all the fine tunning will be this type QAt so that we will not loss much data and accuracy

> 👉🏽 Their are 2 important technique we need to understand QLoRA, LoRA. We nee to understand specifically w.r.t Fine Tunning.

Summary

Quantization is a powerful technique for optimizing the memory efficiency and inference speed of LLMs. By understanding the different methods, calibration techniques, and the trade-offs involved, you can effectively apply quantization to deploy LLMs on various platforms and enhance their performance.

Important Considerations

Accuracy Trade-offs : Quantization always involves a trade-off between memory savings and accuracy. You need to evaluate the impact of quantization on your model's performance and select the appropriate method for your specific needs.

Hardware Compatibility : Ensure that the hardware you're targeting supports the chosen quantization format (e.g., INT8) to avoid compatibility issues.

Fine-Tuning Strategies : For fine-tuning LLMs after quantization, it's generally recommended to use QAT to minimize accuracy loss.This revised document offers a clearer and more structured explanation of quantization.

LoRA, QLoRA: In-Depth Mathematical Intuition - Fine-Tuning LLM Models

Introduction

Fine-tuning large language models (LLMs) like GPT-4 and GPT-4 Turbo is a crucial aspect of adapting them to specific tasks and domains. However, traditional full-parameter fine-tuning methods present significant challenges due to memory constraints and computational resources. This is where techniques like LoRA (Low-Rank Adaptation) and QLoRA (Quantized LoRA) come into play. These techniques offer efficient ways to fine-tune LLMs by selectively updating parameters.

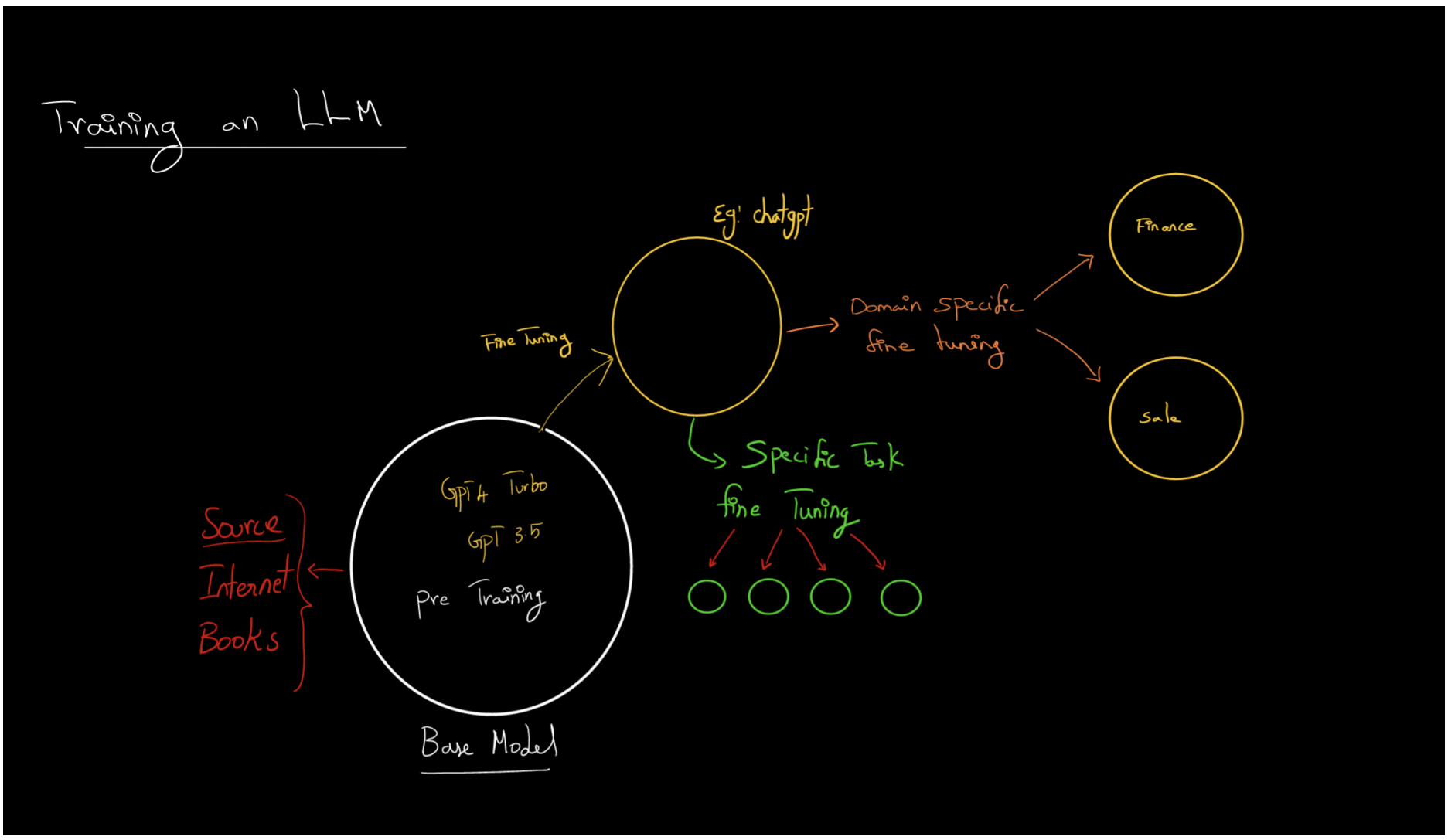
Why LoRA and QLoRA is used?

Low Rank Adaptation(LoRA) is specifically used in Fine Tuning LLM Models.

Initially when ever their is a pre-trained LLM model basically means Gpt4 or Gpt4-Turbo and this model is been created by OpenAi

We basically say this model as Base Model and this model is trained with huge amount of data, So the data sources can be internet, it can be book, can be multiple sources.

all this model they can be measured with token and words like it support 1.5 million token, its been trained with this many number of words. What happens is all this modles probably to predict the next word it will have the context of all those tokens and then it will be able to give response.



these all models are basically base model and we also say this as Pre-Trained Model

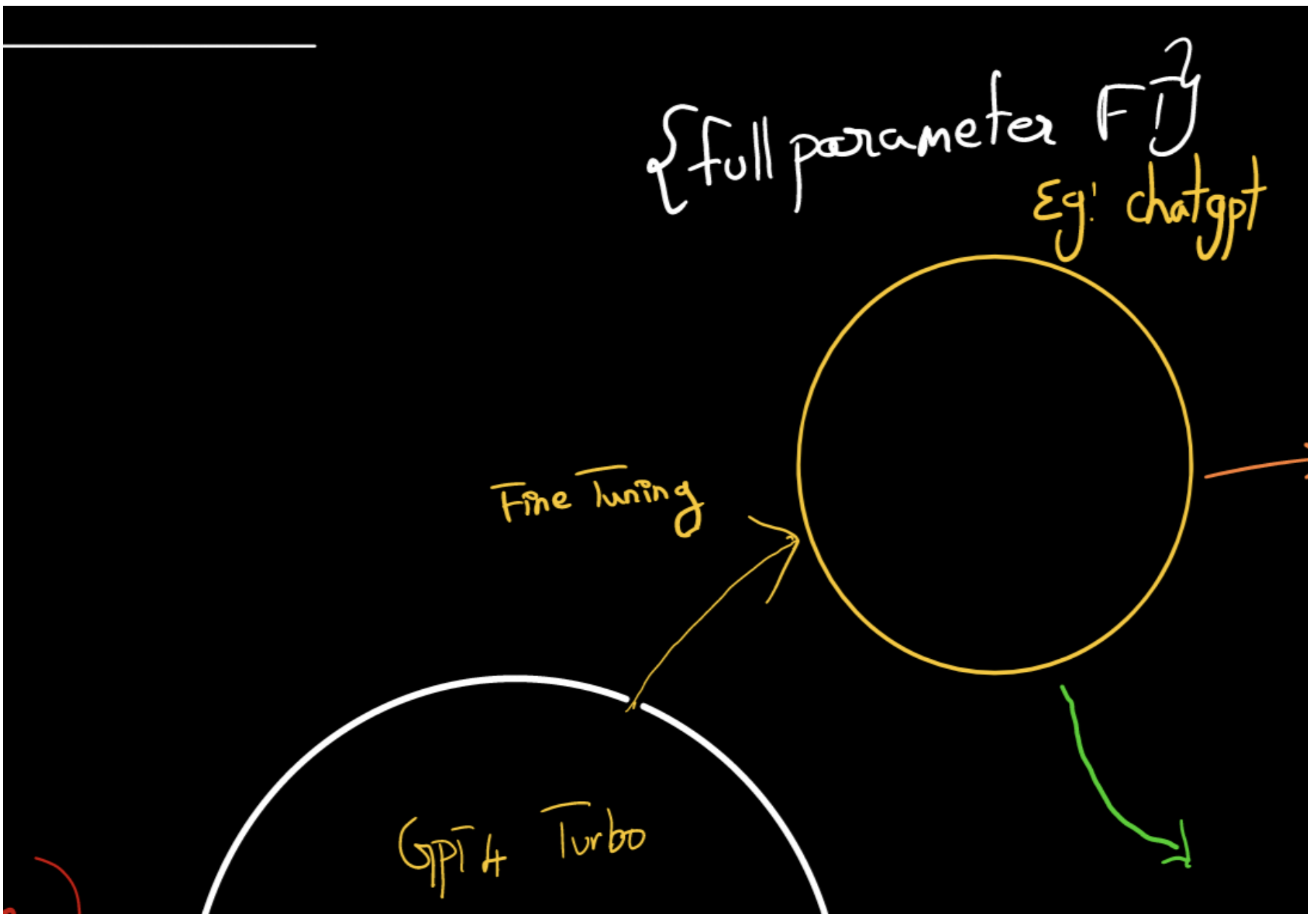
some of the E.g:- Gpt4, Gpt4 Turbo, Gpt 3.5, Gemini, Gemini 1.4.. all these models are specifically pre-trained models

1. The Fine-Tuning Landscape

👉🏾 Before diving into LoRA and QLoRA, let's understand the different approaches to fine-tuning LLMs.

Their are various way for fine tuning it

▼ Full Parameters Fine-Tuning



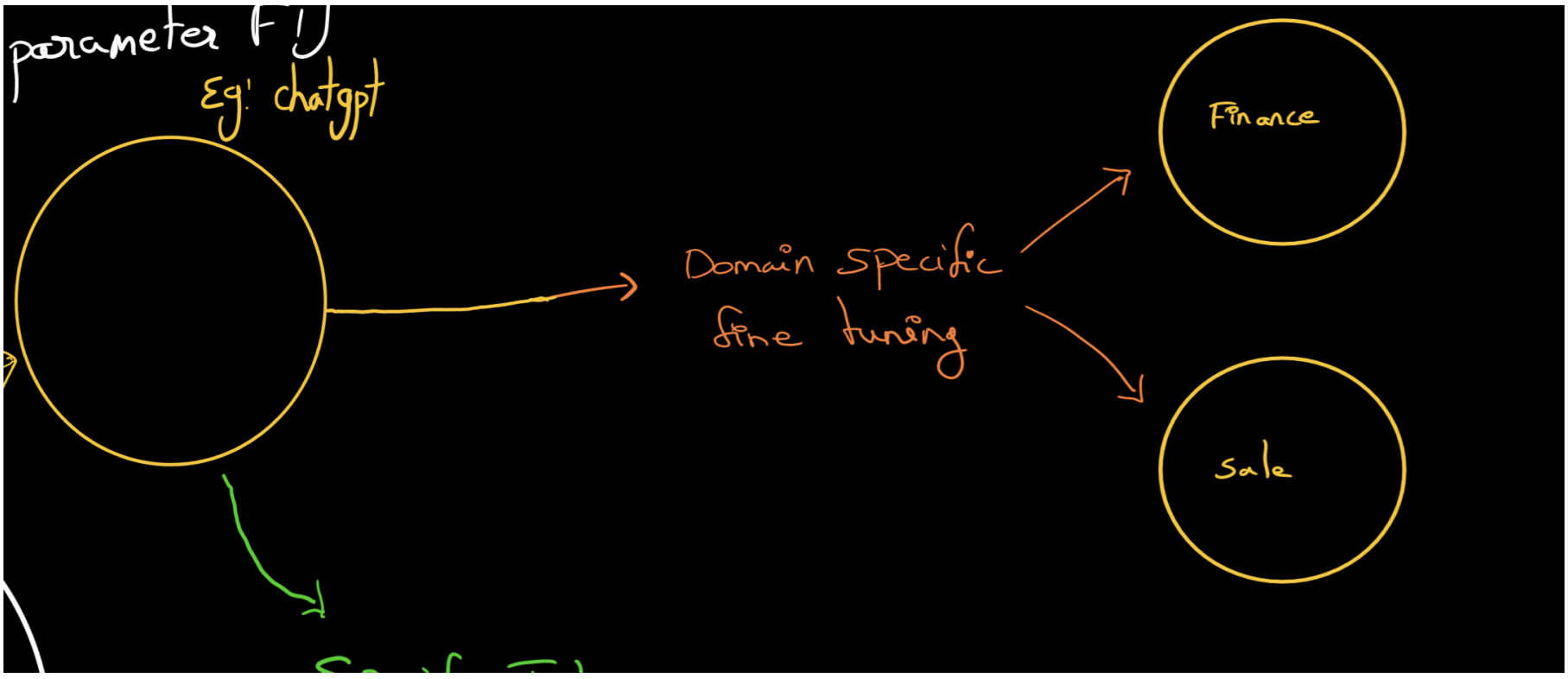
Full Parameter Fine-Tuning: This involves updating all weights within the pre-trained LLM model. This method achieves high accuracy but requires substantial computational resources and memory.

Detail Examination:

This fine tuning is done on all the weights of this specific base model

And, Some of the application we may generate is like Chatgpt, cloudy chatgpt, this is one way of fine tuning were we train all parameters

▼ Domain Specific Fine Tuning

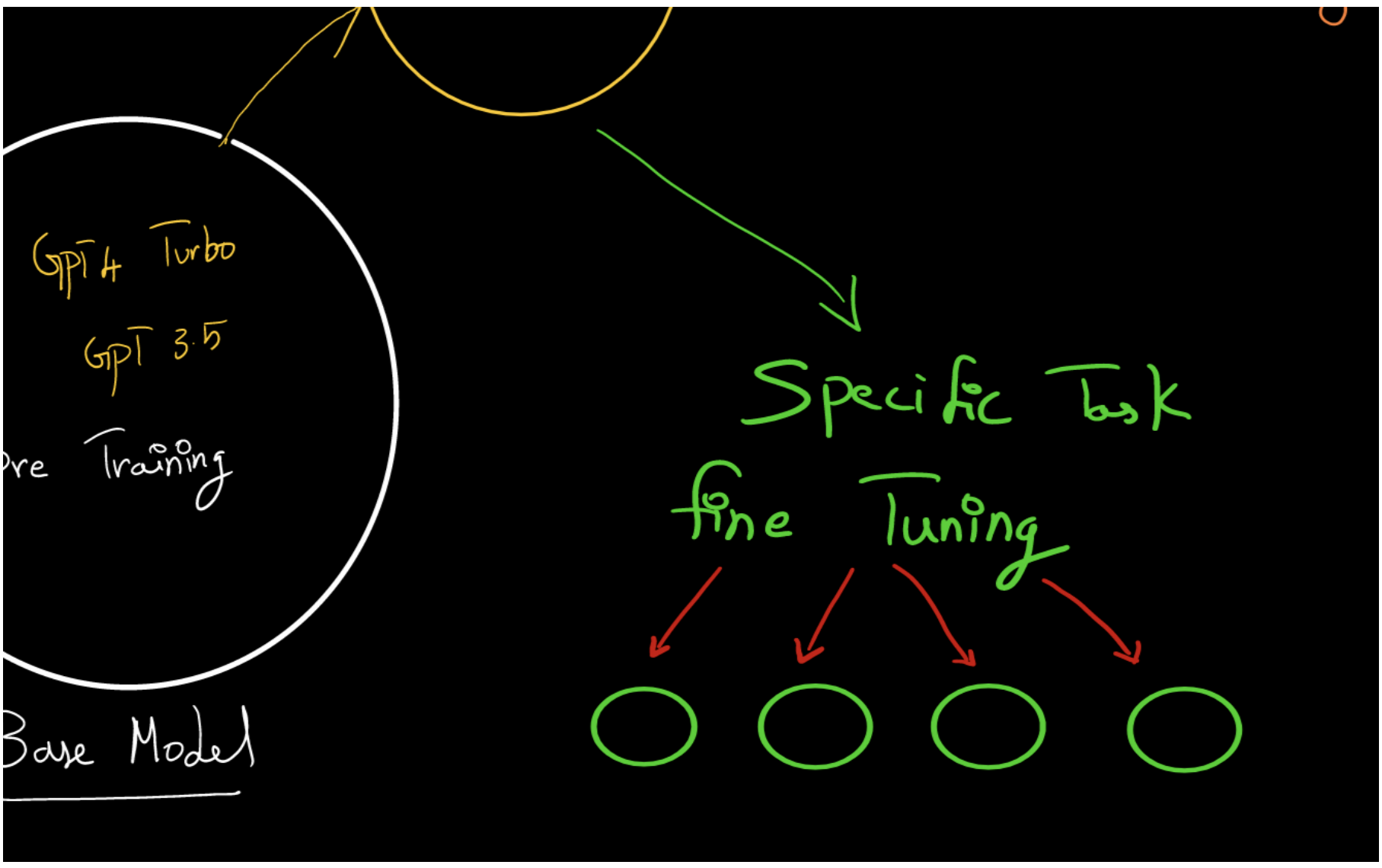


Domain-Specific Fine-Tuning: This approach tailors the pre-trained model to specific domains (e.g., finance, healthcare, legal) by fine-tuning it on a dataset relevant to that domain.

You can also take this model and perform domain specific fine tuning, lets say fine tuning a chatbot model for finance, sales, for different domain itself

---

▼ Specific Task fine Tuning



Specific Task Fine-Tuning: This focuses on training the model for specific tasks, such as question answering, text summarization, or translation.

in case of specific task fine tuning this are my different task , like task A, B, C, D.

A can something related to Q&A Chatbot, B can be document Q&A Chatbot.

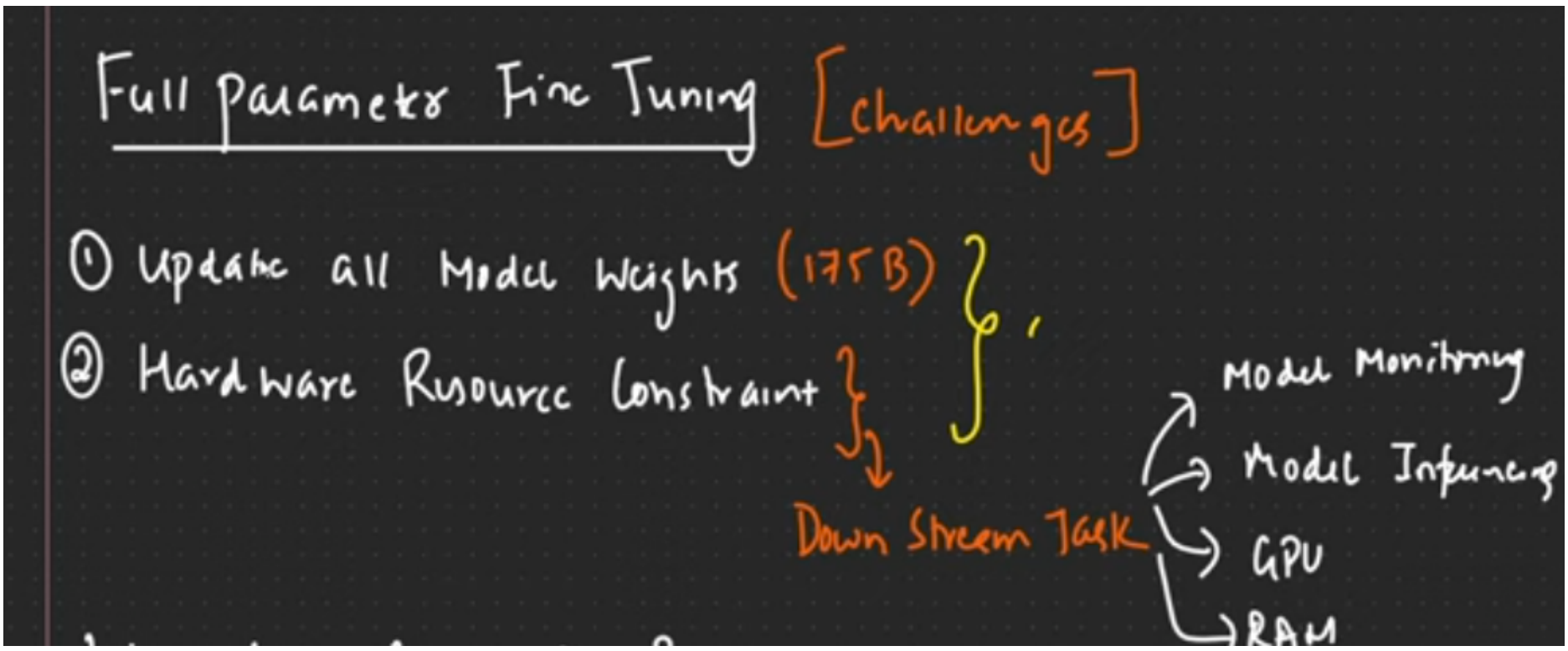
---

2. The Challenges of Full Parameter Fine-Tuning

Full parameter fine-tuning often presents significant challenges due to hardware resource constraints.

Memory Limitations: LLMs typically have billions of parameters, and updating all of them requires a massive amount of memory. Standard GPUs and systems with limited RAM may struggle to handle this.

Downstream Task Challenges: Fine-tuning can make deploying the model for downstream tasks (like model monitoring, inference, and real-time applications) more difficult due to the increased memory requirements and computational complexity.



Detail Examination :

let's see Full Parameter Fine Tuning and challenges with it. That is were LoRA is used.

In full parameter fine tuning the **major challenge is to update the model weight**.

Let's say i have a model which is some way around ***175B parameters that basically means 175 Billion weights.*** In this particular case when ever i fine tune the model i need to update all the weights.

"why it can be a challenge?.. Because their will be Hardware Resource Constraint".

w.r.t different task if really want to use this particular model i need more Gpu and Ram for inferencing purpose, so for **downstream task** it becomes very difficult

Downstream task E.g:- *Model Monitoring, Model Inferencing, similarly the Gpu & Ram Constraint we may have, so we may face multiple challenges when we have this full parameter fine tunning*

> 👉🏽 In order to overcome this fact we will be using LoRA and QLoRA.

3. Introducing LoRA: Efficient Fine-Tuning

LoRA (Low-Rank Adaptation) addresses these challenges by introducing a more efficient way to fine-tune LLMs. Instead of updating all parameters, LoRA selectively updates a smaller set of parameters that represent changes from the pre-trained model.

A. How LoRA Works / What Does LoRA do ?

Parameter Tracking : LoRA doesn't directly modify the original pre-trained weights. Instead, it tracks the changes in those weights that occur during fine-tuning.

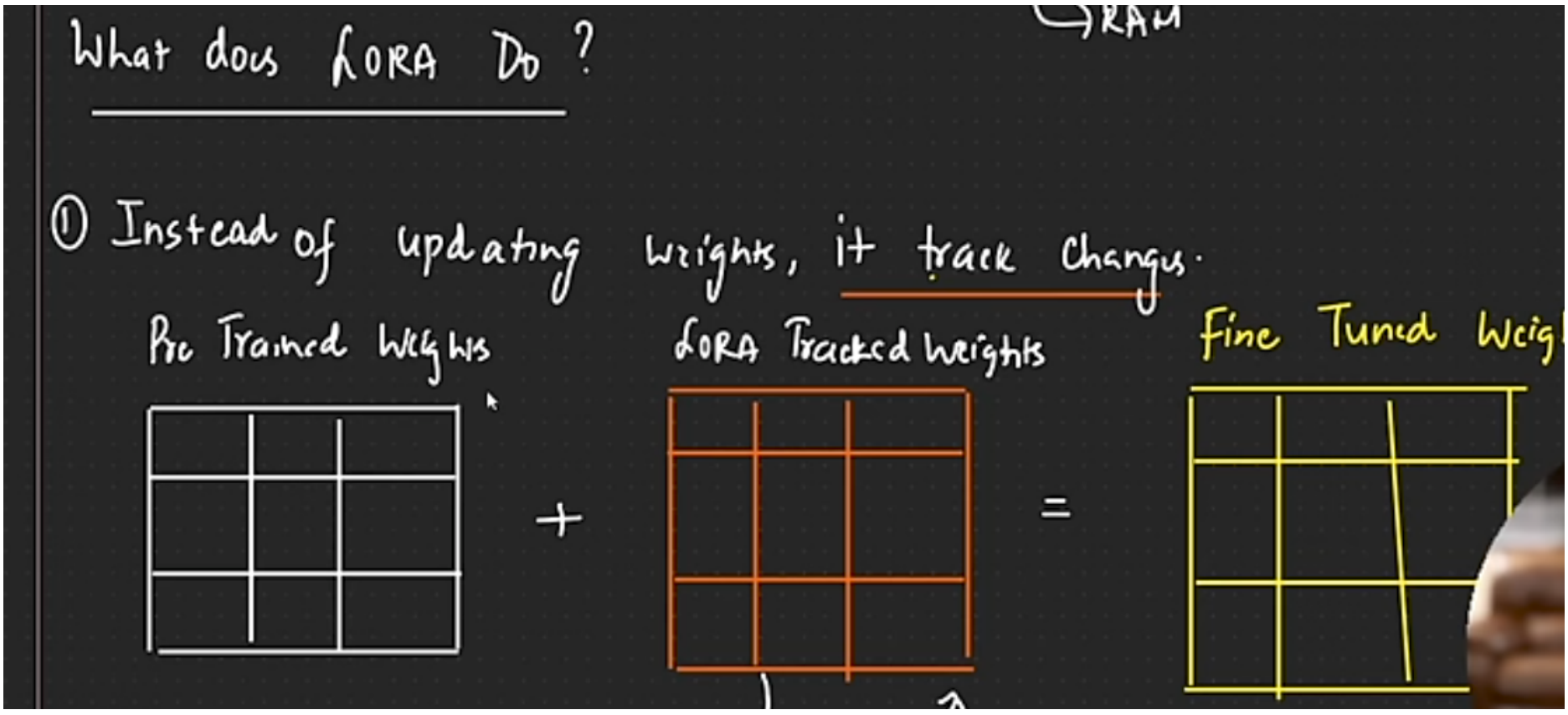
Matrix Decomposition : LoRA achieves this efficient tracking by utilizing matrix decomposition. A large weight matrix (representing the changes) is decomposed into two smaller matrices (often of a lower rank). This decomposition reduces the number of parameters that need to be stored and updated.

Combining Changes : The decomposed matrices are then combined with the original pre-trained weights to create the final fine-tuned model.

Detail Examination :

LoRA says that Instead of updating all the weights in Full parameter fine tunning, **it will not update them Instead it will track the changes**

[Now what changes it will be tracking] == **It tracks the changes of the new weight based on fine tuning**



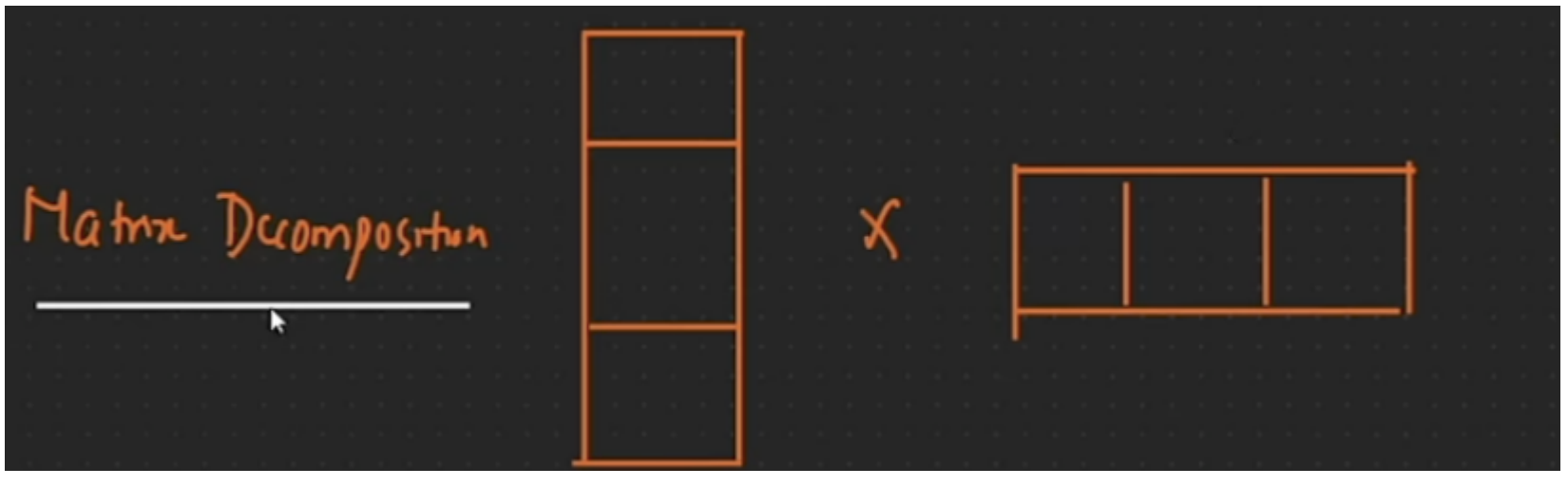
So, Based on the new weights how we will be combining these weights with pre-trained weights.

From above you can see the Pre-Trained Weights from base model(Like lama2). if we performe fine tune using LoRA , the **LoRA will Track the new Weights** from above image which will be of same size. lets say it's 3×3 then the new weight's when it probably do the forward and backward propagation those new weights will be tracked in a separate matrix **3×3** and then this 2 weights 3X3 and **3×3** will combine and we wil get the **Fine Tuned Weights.**

In this way what happens is that, this tracking will happen in a separate way

you can see here also we are Updating all the Weights here also the resource constraint will definitely happens. Yes right its about 3×3 just think of weights and parameters of 175 billion or 7 billion that time i will be having a huge matrix.

At this scenario we need to understand how LoRA works because these **Weights how it's getting Tracked it will just not get tracked in 3×3 matrix instead all this weights been tracked**



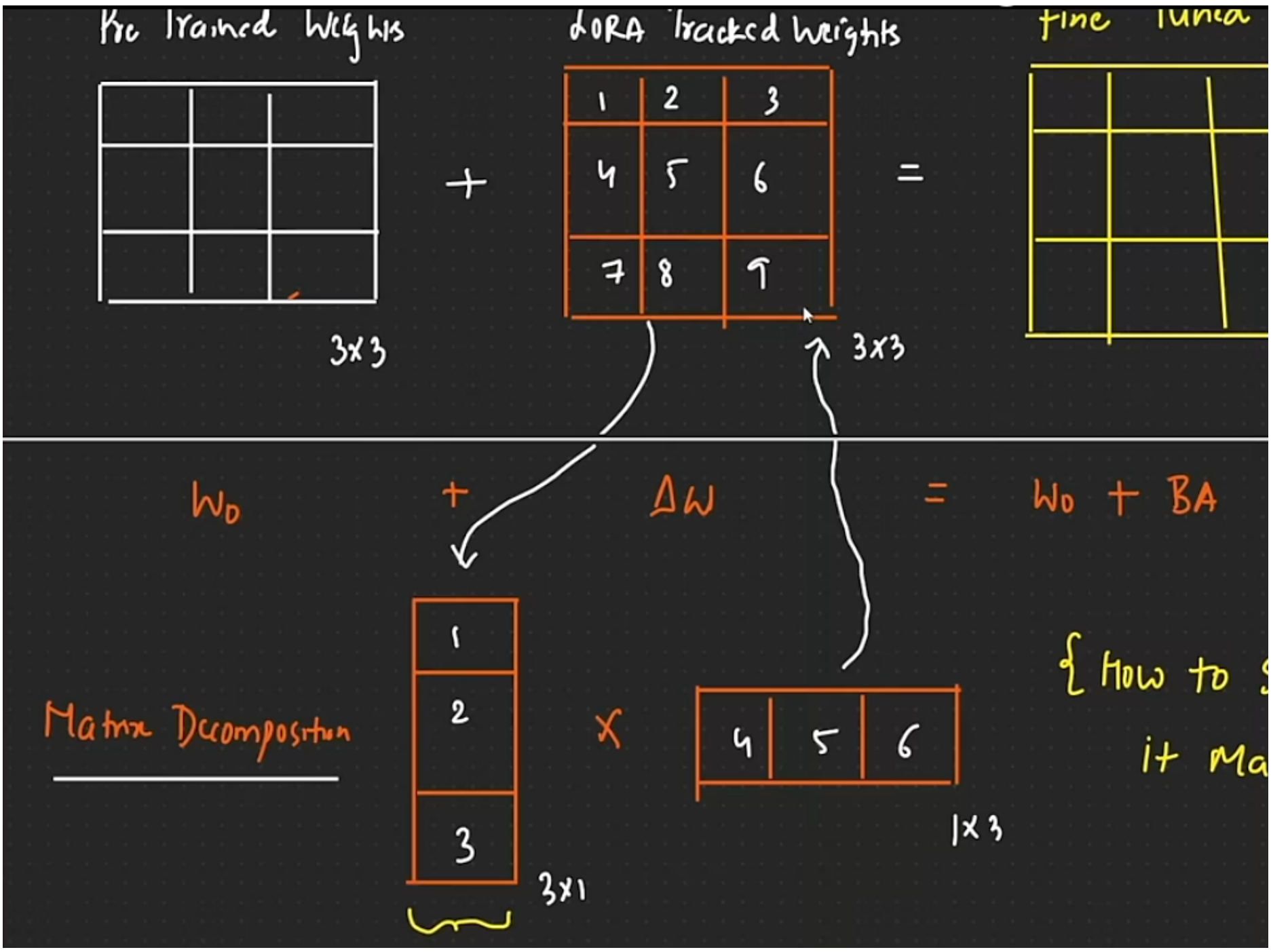
Their a simple mathematical equation will happen or technique called Matrix Decomposition

That means the same 3×3 matrix is saved in two smaller matrix, How . .?

Example : Imagine a 3×3 weight matrix representing changes. LoRA decomposes this into two matrices: a 3×1 matrix (B) and a 1×3 matrix (A). When multiplied, these smaller matrices reconstruct the original 3×3 change matrix.

Detail Examination :

we have a **3×3** we can save it as 3×1 —- 1×3 when we Multiply this two matrix we will be able to find that tracked weights 3×3



let’s consider around 9 weights you can see from above image we will be able to get those 9 weights from 6 number of parameters. When we Multiply this we will be able to get all these 9 parameters or weights

Inshort what LoRA is doing is that it is performing this Matrix Decomposition were the big matrix and this matrix can be of any size is decomposed in to 2 smaller matrix based on the parameter called as Rank.

B. Understanding Rank and its Impact :

Rank : The rank of the decomposed matrices influences the complexity of the changes that LoRA can capture. A higher rank allows LoRA to learn more complex relationships, but it also requires more parameters to represent those changes.

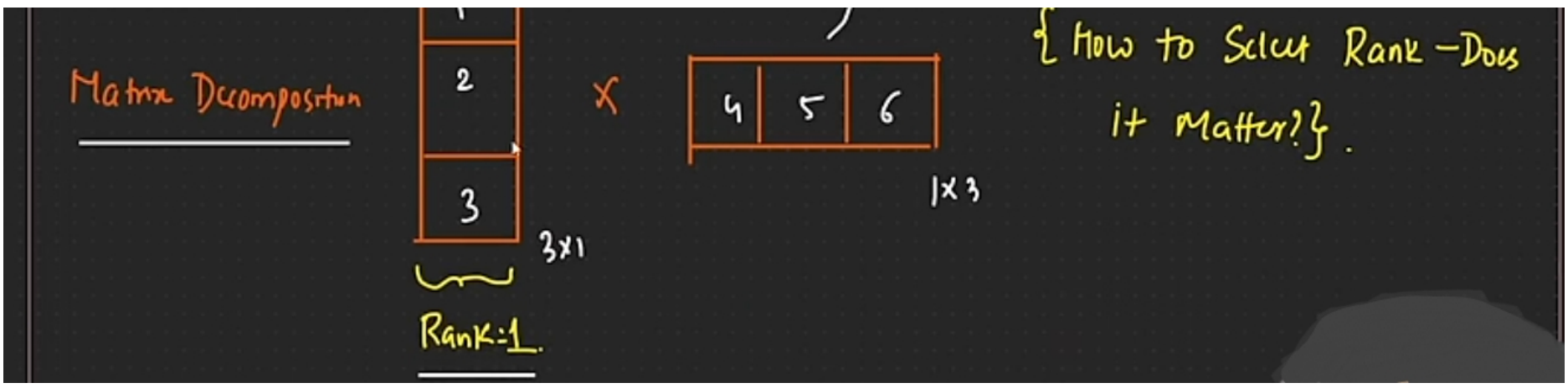
Balancing Efficiency and Complexity : LoRA aims to find the optimal rank that balances efficiency (reduced memory and computational cost) with the ability to learn the necessary changes during fine-tuning.

Research Findings : Research indicates that LoRA performs well with ranks typically ranging from 1 to 8.

Detail Examination :

How to calculate a rank of matrix?

It is a simple Algebraic equation based on Transpose of a matrix how we calculate the Rank

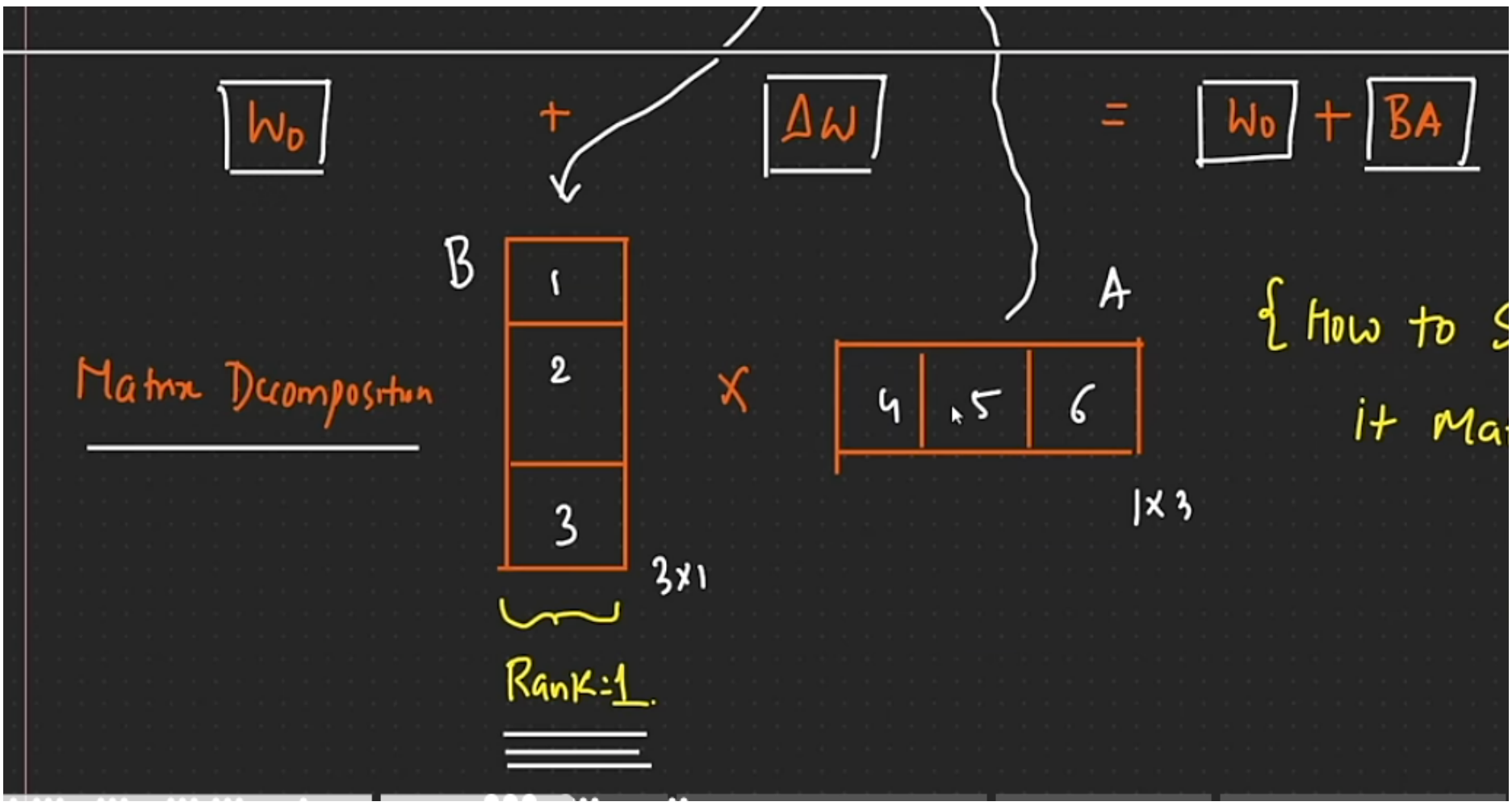


let's say that this matrix that i have which is a 3×3 one over here the rank of this particular matrix is 1 and if i use this 2 matrix you can see the number of parameters that im storing over here is less when compared to this 3×3

"Yes, their will some loss in precision but it's making sure that when we combine both this matrix we will be able to get the entier updated weight"

just imagine lets say i have 7 billion parameters now i am trying to perform fine tuning on those parameters when ever i track those weights this huge matrix which we saw(eg 3×3) will be decomposed into 2 smaller matrix

when we are decomposing this Updated Tracked Weights matrix into 2 smaller matrix obviously we will be needed less parameters to store those values, and by this way your fine tuning will become very much efficient and this really solve Resource constraint



and obviously this required less parameters if you decomposing our bigger matrix in to 2 smaller matrix less parameters is required

C. When to Use High Rank :

Complex Relationships : In cases where the fine-tuned model needs to learn more complex relationships, a higher rank might be necessary to represent those changes effectively.

Beyond the Base Model's Capabilities : If the base model has limited capacity to learn specific tasks or behaviors, a higher rank can help the fine-tuned model overcome these limitations.

IF THE MODEL WANTS TO LEARN COMPLEX THINGS THE WE CAN SPECIFICALLY USE HIGH RANK ***[let's say some of the model is not trained to interact or perform some of the behaviour at that point of time those complex things can be handled when you are probably increasing the number of ranks] .***

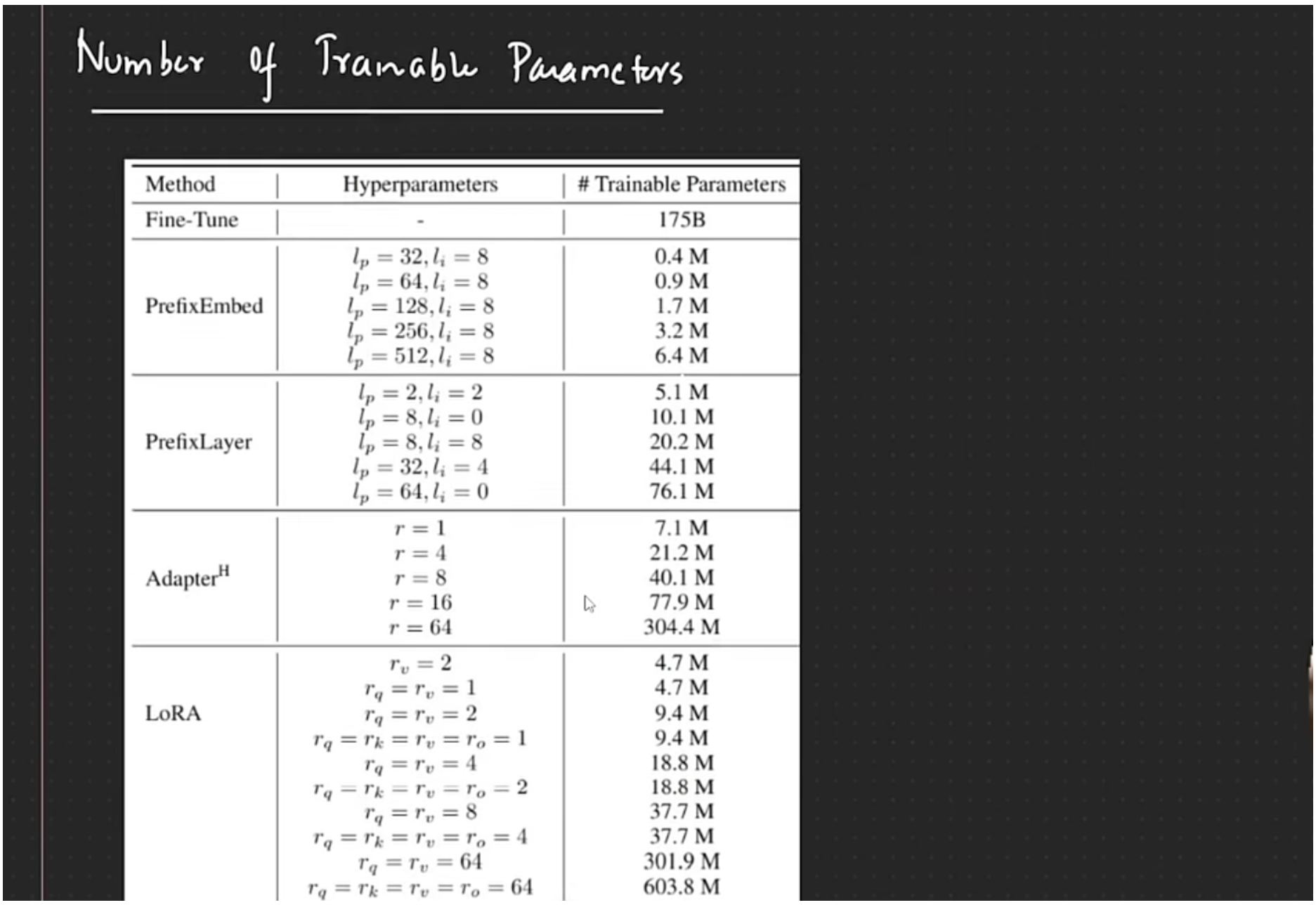
Now what will happen if we keep on increasing the Rank?

Detail Examination :

if you keep on increasing the rank, A&B parameters will also get Increasing but it will always be less than **LoRA Tracked Weights**.

> 👉🏾 *eg. IF I HAVE 7_BILLION PARAMETERS IF DECOMPOSED IN TO 2 SMALLER MATRIX WITH INCREASING RANK THEN ALSO THE PARAMETERS REQUIRED WILL BE LESS..*

How i am saying this because in research paper they have tried with multiple trainable parameters



Method	Hyperparameters	# Trainable Parameters
Fine-Tune	-	175B
PrefixEmbed	$l_p = 32, l_i = 8$	0.4 M
	$l_p = 64, l_i = 8$	0.9 M
	$l_p = 128, l_i = 8$	1.7 M
	$l_p = 256, l_i = 8$	3.2 M
	$l_p = 512, l_i = 8$	6.4 M
PrefixLayer	$l_p = 2, l_i = 2$	5.1 M
	$l_p = 8, l_i = 0$	10.1 M
	$l_p = 8, l_i = 8$	20.2 M
	$l_p = 32, l_i = 4$	44.1 M
	$l_p = 64, l_i = 0$	76.1 M
Adapter^{H}	$r = 1$	7.1 M
	$r = 4$	21.2 M
	$r = 8$	40.1 M
	$r = 16$	77.9 M
	$r = 64$	304.4 M
LoRA	$r_v = 2$	4.7 M
	$r_q = r_v = 1$	4.7 M
	$r_q = r_v = 2$	9.4 M
	$r_q = r_k = r_v = r_o = 1$	9.4 M
	$r_q = r_v = 4$	18.8 M
	$r_q = r_k = r_v = r_o = 2$	18.8 M
	$r_q = r_v = 8$	37.7 M
	$r_q = r_k = r_v = r_o = 4$	37.7 M
	$r_q = r_v = 64$	301.9 M
	$r_q = r_k = r_v = r_o = 64$	603.8 M

they are multiple techniques of fine tuning some of the technique that are their in column method fine tune in above tabular image here Adapter is very famous that is probabilly used befor LoRA

In Hyperparameter & Trainable Parameters we can find that as the Rank is increasing the parameters are decreases compared with Billions of parameters

Initially the trainable parameters are 175B but when i use technique like Adaptor it decreases to Million, initially it is 7.1M with Rank = 1 as i keep on Increasing the Rank the T_parameters will also Increase

we can compare 175B to 7.1M Weights the Percentage is very less

ll’y in LoRA because of Matrix Decomposition we can find as the Rank increases weights will also increase with less percentage compared with 175B

LoRA	$r_v = 2$	4.7 M
	$r_q = r_v = 1$	4.7 M
	$r_q = r_v = 2$	9.4 M
	$r_q = r_k = r_v = r_o = 1$	9.4 M
	$r_q = r_v = 4$	18.8 M
	$r_q = r_k = r_v = r_o = 2$	18.8 M
	$r_q = r_v = 8$	37.7 M
	$r_q = r_k = r_v = r_o = 4$	37.7 M
	$r_q = r_v = 64$	301.9 M
	$r_q = r_k = r_v = r_o = 64$	603.8 M

The Rank q, k, v. This 3 parameters Q, K, V

there only the all multiplication matrix happen w.r.with 3 parameters

> 👉🏽 *Compare from 175B to 4.7M parameters how it was possible ?*
>
> *Ans:- Matrix Decomposition, it's making sure that parameters are not much compared with 175B*

As we keep on increasing the rank from above image take 4, 8, 64 the parameters are obviously increasing but comparing with 175B it is very less 18.8M, 37.7M, 603.8M

▼ *Note* :

- MicroSoft came up with this LoRA technique in one of the Research paper and the have used Rank = 8 to do the Fine Tuning and it as performed absolutely well.
- Most of the time we select Rank 1 - 8 but at E.O.D how to select this Rank ?
- It won't matter because the parameters are increasing very less number over here as we go ahead so usually we can select Rank 1 - 8 to perform fine tuning their will be a scenario were when should we use High Rank

---

At EOD this equation will bee seen in most of the research paper

what LoRA is doing is, All the Track Weights are Decomposed into 2 smaller matrix with different Ranks

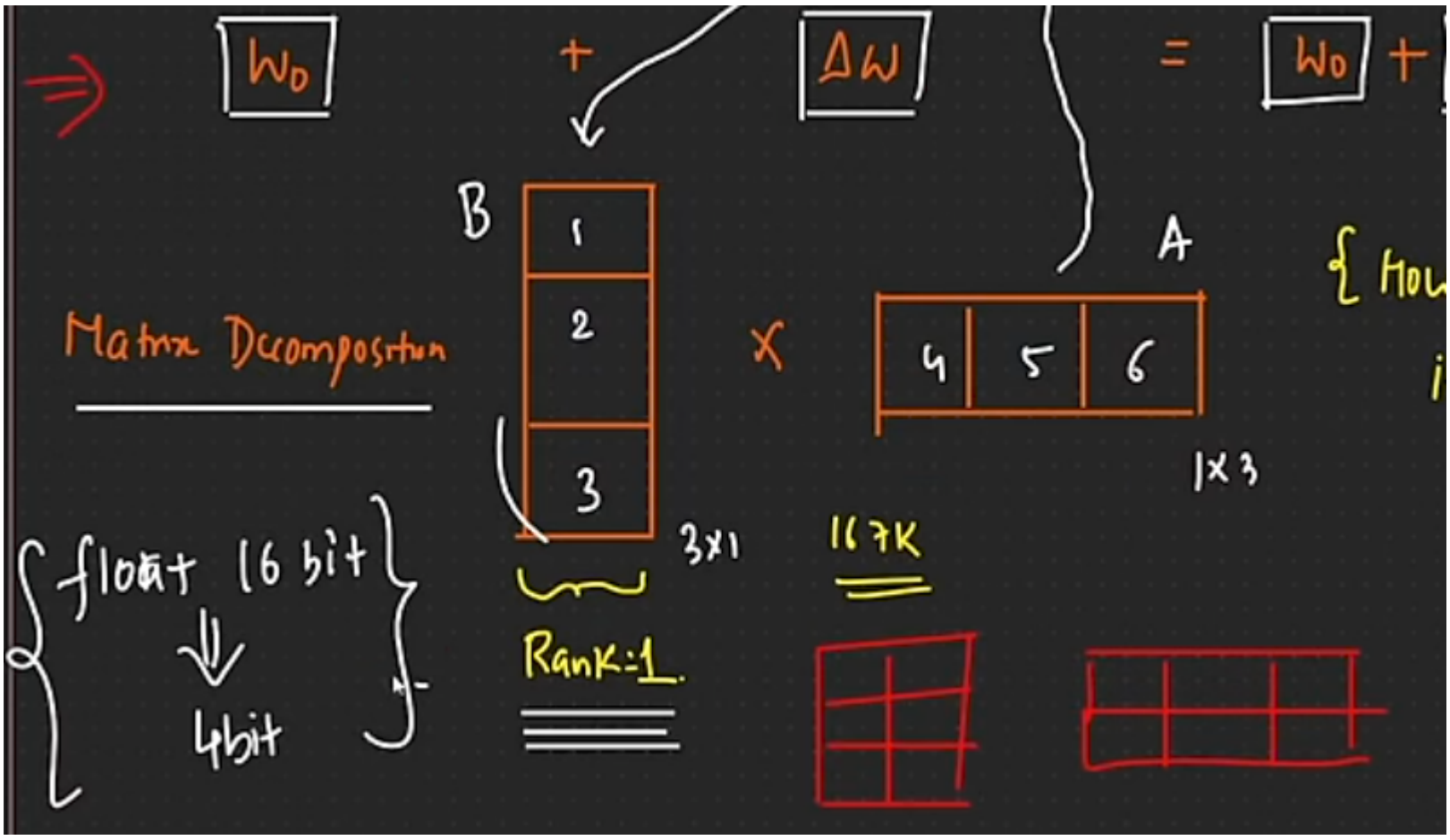
When Fine Tuning the first thing is you really need to set Rank

Because of this technique the Fine Tuning, parameters is done less and this is how the main Resource Constraint is done w.r.t all the DownStream task it becomes very much easy.

"Ok this a LoRA, what about 2.0 ⟹ QLoRA ?"

Quantized LoRA (QLoRA): Further Optimization

QLoRA builds upon LoRA by introducing a quantization step, further reducing the memory footprint of the fine-tuned model.



Concept : The decomposed matrices (B and A) within LoRA are typically stored in FP16 (half-precision). QLoRA converts these matrices to INT4 (4-bit integers), further reducing the memory required to store them.

Precision Trade-off : While QLoRA achieves significant memory savings, it involves a slight reduction in precision. This reduction in precision might lead to a small decrease in accuracy, but this is often offset by the significant memory savings and the ability to deploy the model on devices with limited memory.

Benefits : QLoRA allows for the deployment of fine-tuned LLMs on devices with extremely limited memory, making it suitable for mobile applications and edge computing.

Detail Examination :

Quantized LoRA what happens here is in Matrix Decomposition B&A that are stored in Float 16 bit we will try to convert this into 4 bit, by this we will reduce the precision and reduce the values also by this we won't require more memory that is the reason we say QLoRA technique.

Summary : LoRA and QLoRA are valuable techniques for fine-tuning LLMs while overcoming memory limitations. By efficiently tracking parameter changes using matrix decomposition and leveraging quantization, these methods empower developers to deploy fine-tuned LLMs on diverse platforms and for various real-world applications.

Key Takeaways :

- *Memory Efficiency* : LoRA and QLoRA significantly reduce memory requirements for fine-tuned LLMs, making them more practical for deployment on various devices.
- *Computational Efficiency* : The reduced number of parameters in LoRA and the compressed representation in QLoRA contribute to improved computational efficiency during inference.
- *Accuracy Considerations* : While LoRA and QLoRA offer memory advantages, they might lead to a slight reduction in accuracy. It's crucial to carefully evaluate the trade-off between memory savings and potential accuracy loss.
