# Fine Tuning LLMs

Pavlos Protopapas

# Outline

- Instruction-tuning
  - Full Parameter
  - PEFT
- LoRA
- QLoRA

# Outline

- **Instruction-tuning**
  - **Full Parameter**
  - PEFT
- LoRA
- QLoRA

# Instruction-tuning (Full Parameter)

Fine-tuning very often means instruction fine-tuning.

An instruction dataset, comprising pairs of instructions, answers, and sometimes context, is required for such fine-tuning.

# Instruction-tuning (Full Parameter)

| Instruction | Context | Output |
|---|---|---|
| Suggest a good restaurant | Los Angeles, CA | In Los Angeles, CA, I suggest Rossoblu Italian Restaurant |
| Rewrite the sentence with more descriptive words | The game is fun | The game is exhilarating and enjoyable |
| Calculate the area of the triangle | Base: 5cm; Height: 6cm | The area of the triangle is $15\ cm^2$ |

This is an example of what an instruction dataset looks like.

# Instruction-tuning (Full Parameter)

**Task-specific fine-tuning:**

This particular process involves training the model on a smaller, task-specific dataset.

For e.g.: Summarize this, translate that, etc

This allows the model to learn the nuances, and specialized vocabulary relavant to the task.

# Instruction-tuning (Full Parameter)

For e.g., if you train a model specifically for question answering:

Notice, how it answers requests, starting with 'Sure...'.
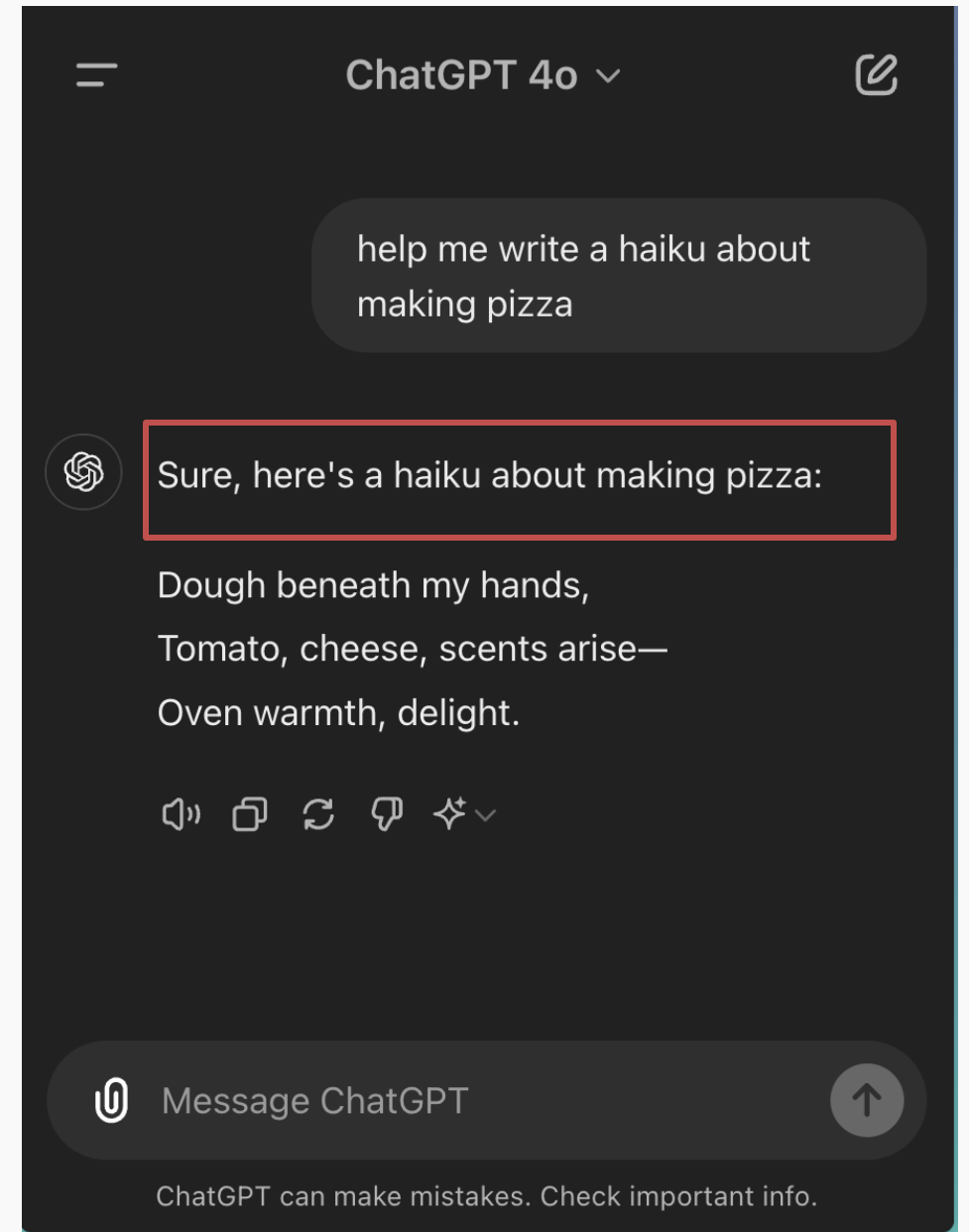
# Instruction-tuning (Full Parameter)

For e.g., if you train a model specifically for question answering:

Notice, how it answers requests, starting with 'Sure...'.



This is opposed to how language models are trained (next-word prediction), according to which the answer should just include the haiku directly.
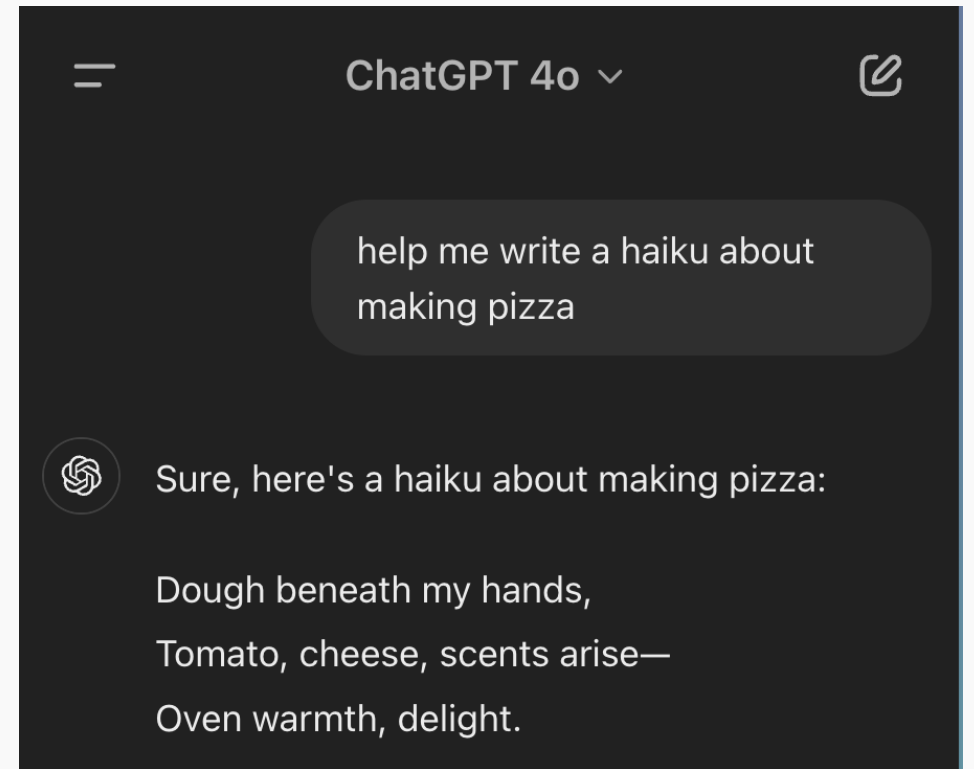
# Instruction-tuning (Full Parameter)

We have to be careful while doing task-specific finetuning to avoid catastrophic forgetting.

**Catastrophic forgetting** refers to the phenomenon where a model **loses its ability to perform previously learned tasks** when it is being fine-tuned on new tasks.

The key idea of catastrophic forgetting is that as the model learns new tasks, it may overwrite what it previously learned, leading to a loss in performance on earlier tasks.

# Instruction-tuning (Full Parameter)

To mitigate the problem of catastrophic forgetting, we need to do multi-task finetuning.

This requires a lot of data, and training resources.

# Instruction-tuning (Full Parameter)

- We need to update all the parameters while finetuning.
  - For a 7B model, we need to update 7 billion weights. For a 13 billion model, we need to update 13 billion weights and so on.

- Storing and updating these weights require a lot of GPU memory.

> 🎉 **Fun Fact:** Did you know, training GPT-4 involved **~25,000 A100 GPUs** over **~90-100 days**, costing OpenAI nearly **$100 million**!

# Instruction-tuning (Full Parameter)

**Not so fun Fact:**

🎉 ~~Fun Fact:~~ Did you know, training GPT-4 involved **~25,000 A100 GPUs** over **~90-100 days**, costing OpenAI nearly **$100 million**!
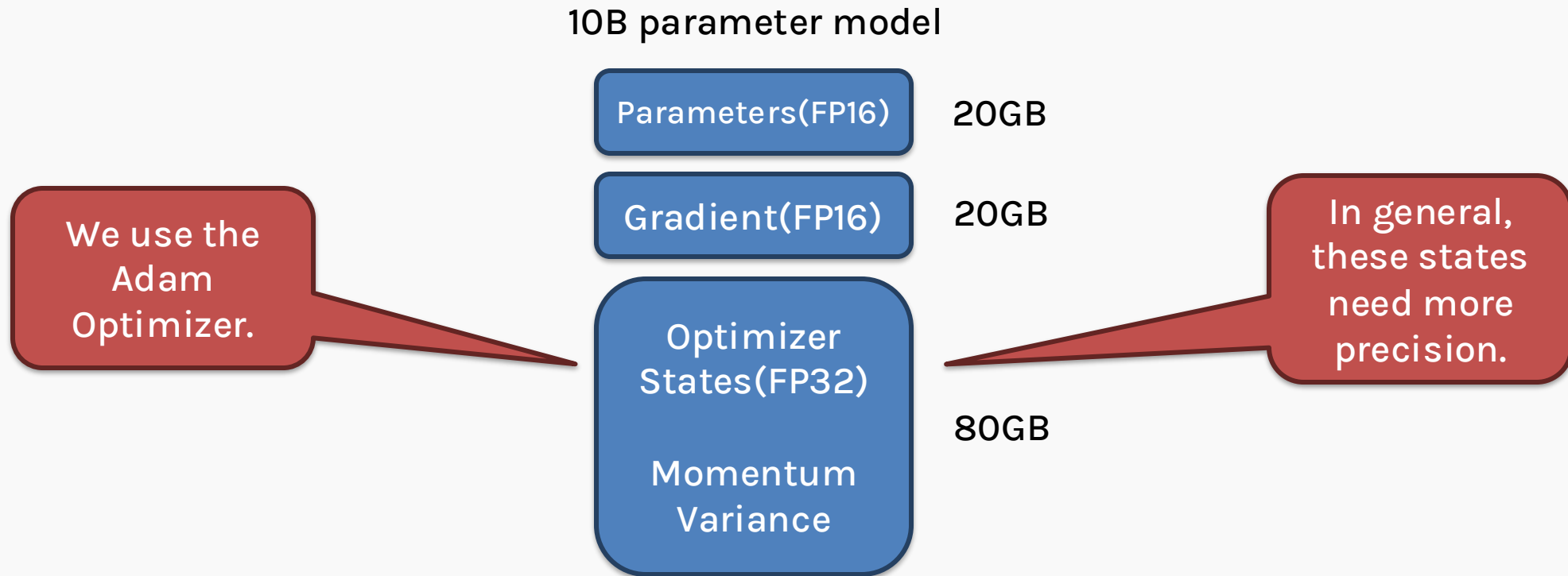
# Instruction-tuning (Full Parameter)

Let's take a fine-tuning example now.

Say we want to finetune a 10 billion parameter model. Let's see how that looks in memory.

Assuming, we're working with FP16 (half precision), which takes approximately 2 bytes per parameter.
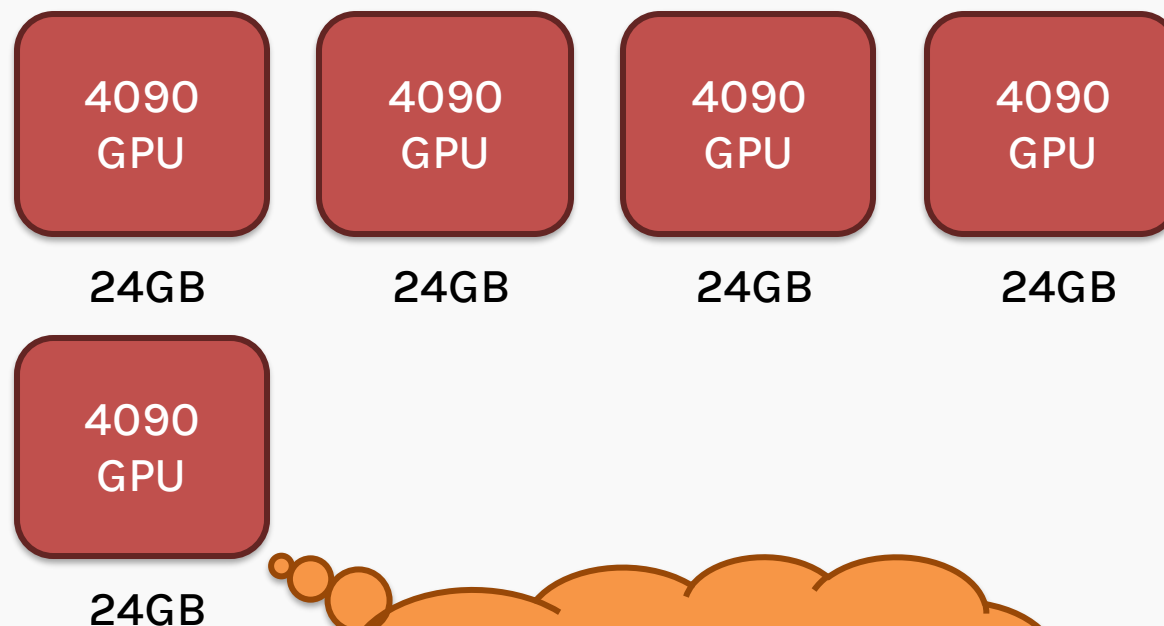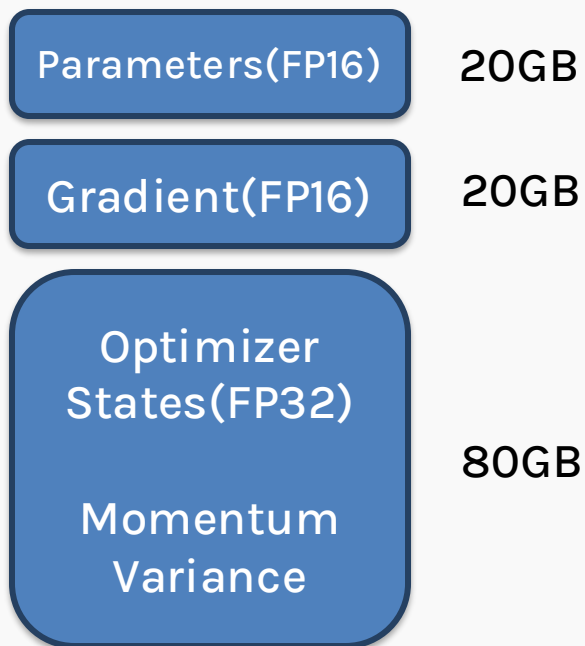
# Instruction-tuning (Full Parameter)

Assuming, we're working with FP16 (half precision), which takes approximately 2 bytes per parameter.

10B parameter model

Parameters(FP16)     20GB

We use the Adam Optimizer.

Gradient(FP16)     20GB

Optimizer States(FP32)

Momentum Variance

80GB

In general, these states need more precision.

# Instruction-tuning (Full Parameter)

Assuming, we're working with FP16 (half precision), which takes approximately 2 bytes per parameter.

10B parameter model

| Parameters(FP16) | 20GB |

| Gradient(FP16) | 20GB |

Optimizer States(FP32)

Momentum Variance

80GB

4090 GPU — 24GB

4090 GPU — 24GB

4090 GPU — 24GB

4090 GPU — 24GB

4090 GPU — 24GB

This model needs at least 5 top-of-the-line consumer-grade GPU's to finetune.

# Instruction-tuning (Full Parameter)

This makes full parameter finetuning inaccessible to normal folks like us.

So, what can we do?

# Outline

- **Instruction-tuning**
  - Full Parameter
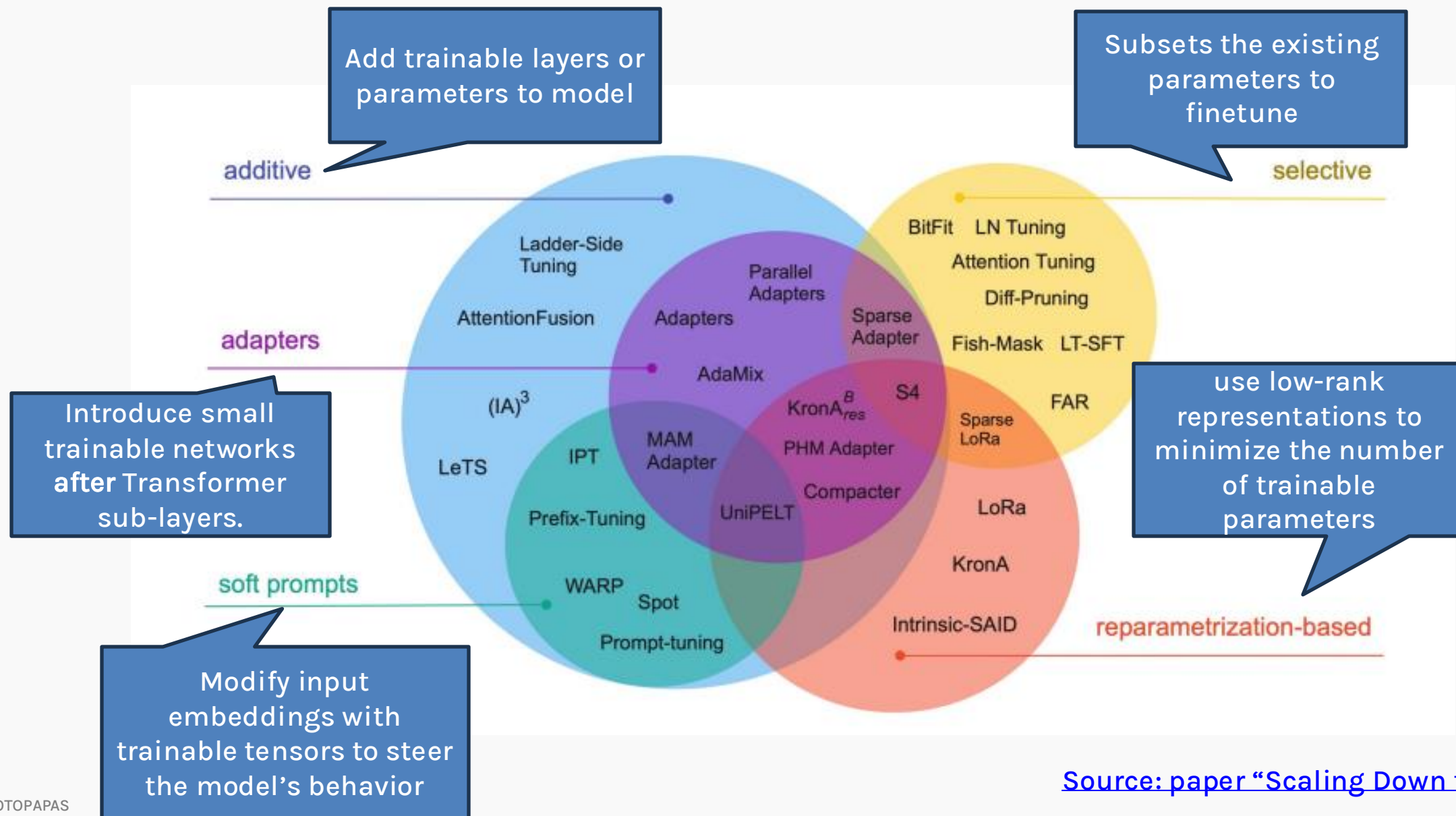  - **PEFT**
- LoRA
- QLoRA

# Instruction-tuning (PEFT)

PEFT stands for Parameter Efficient Finetuning.

Unlike full parameter finetuning, PEFT preserves the vast majority of the model's original weights.

There are majorly three methods to do PEFT.

1. Additive
2. Selective
3. Reparameterization

# Instruction-tuning (PEFT)



Add trainable layers or parameters to model

Subsets the existing parameters to finetune

Introduce small trainable networks **after** Transformer sub-layers.

use low-rank representations to minimize the number of trainable parameters

Modify input embeddings with trainable tensors to steer the model's behavior

additive

selective

adapters

soft prompts

reparametrization-based

Ladder-Side Tuning
AttentionFusion
$(IA)^3$
LeTS
IPT
MAM Adapter
Prefix-Tuning
WARP
Spot
Prompt-tuning
UniPELT
Parallel Adapters
Adapters
AdaMix
Sparse Adapter
$KronA_{res}^B$
S4
PHM Adapter
Compacter
BitFit    LN Tuning
Attention Tuning
Diff-Pruning
Fish-Mask    LT-SFT
FAR
Sparse LoRa
LoRa
KronA
Intrinsic-SAID

Source: paper "Scaling Down to Scale Up" (arxiv.org)

# Instruction-tuning (PEFT)

There are a lot of techniques. We're interested in LoRA, which is one of the most popular.
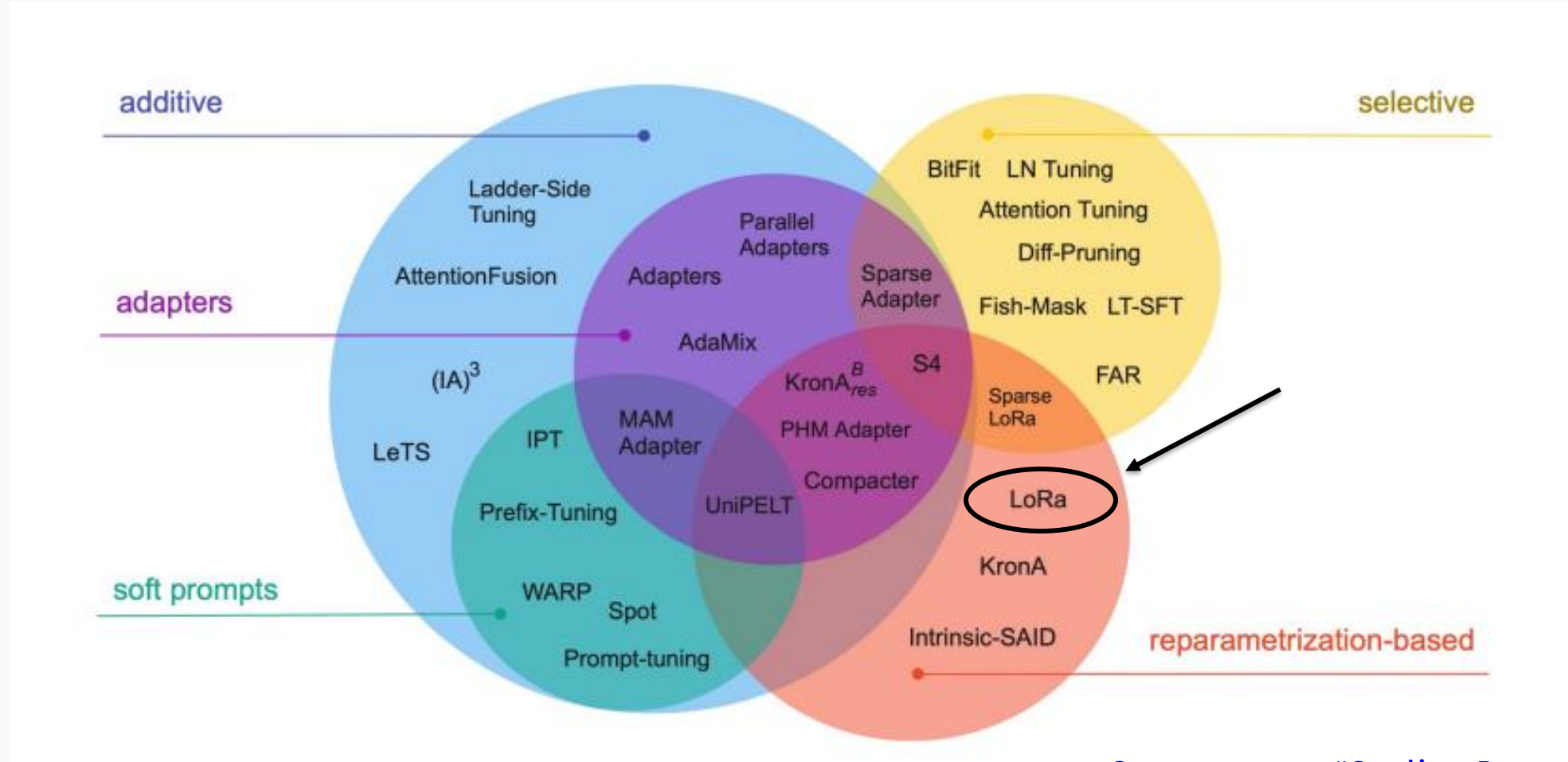
Source: paper "Scaling Down to Scale Up" (arxiv.org)

# Outline

- Instruction-tuning
  - Full Parameter
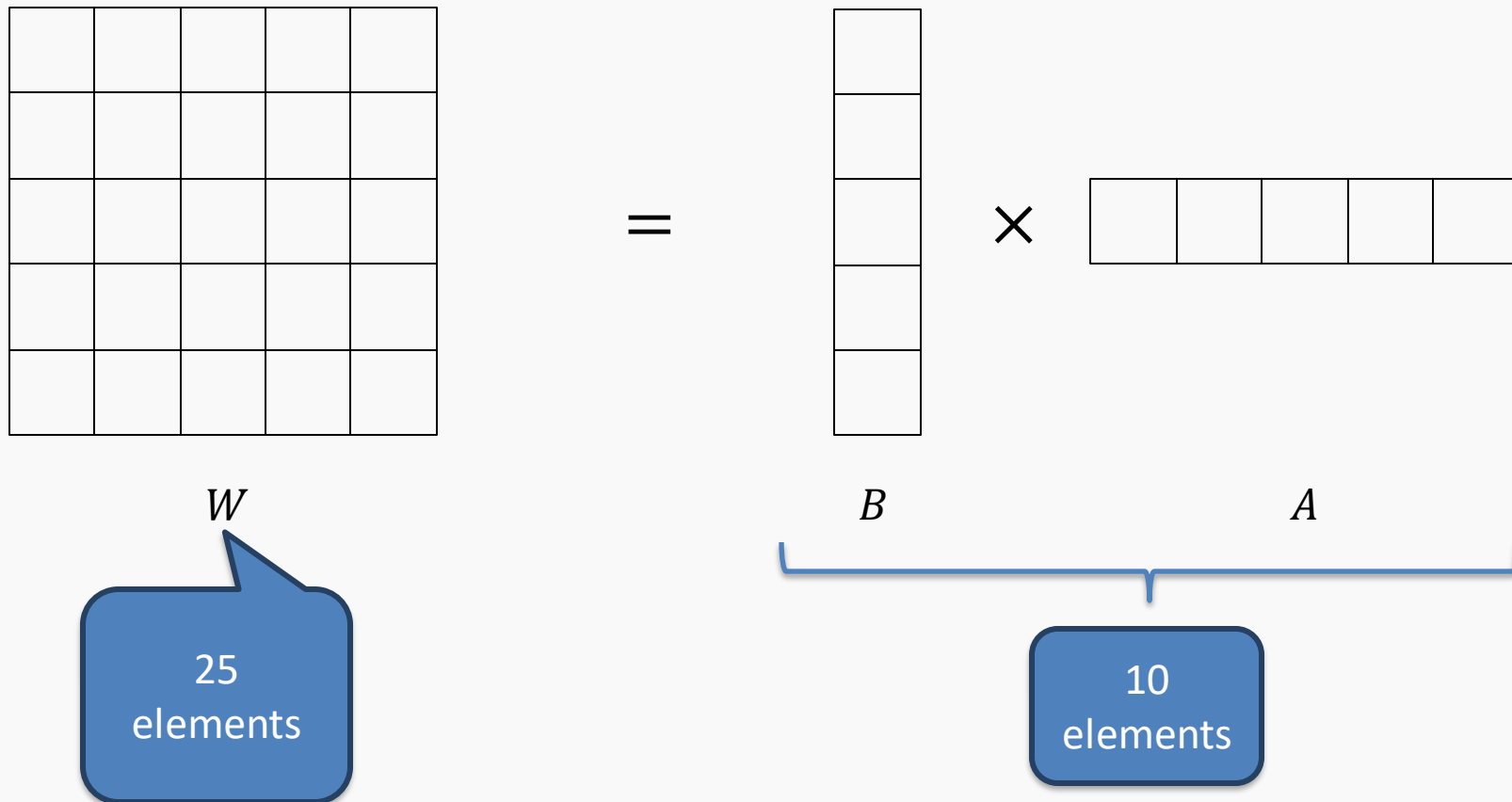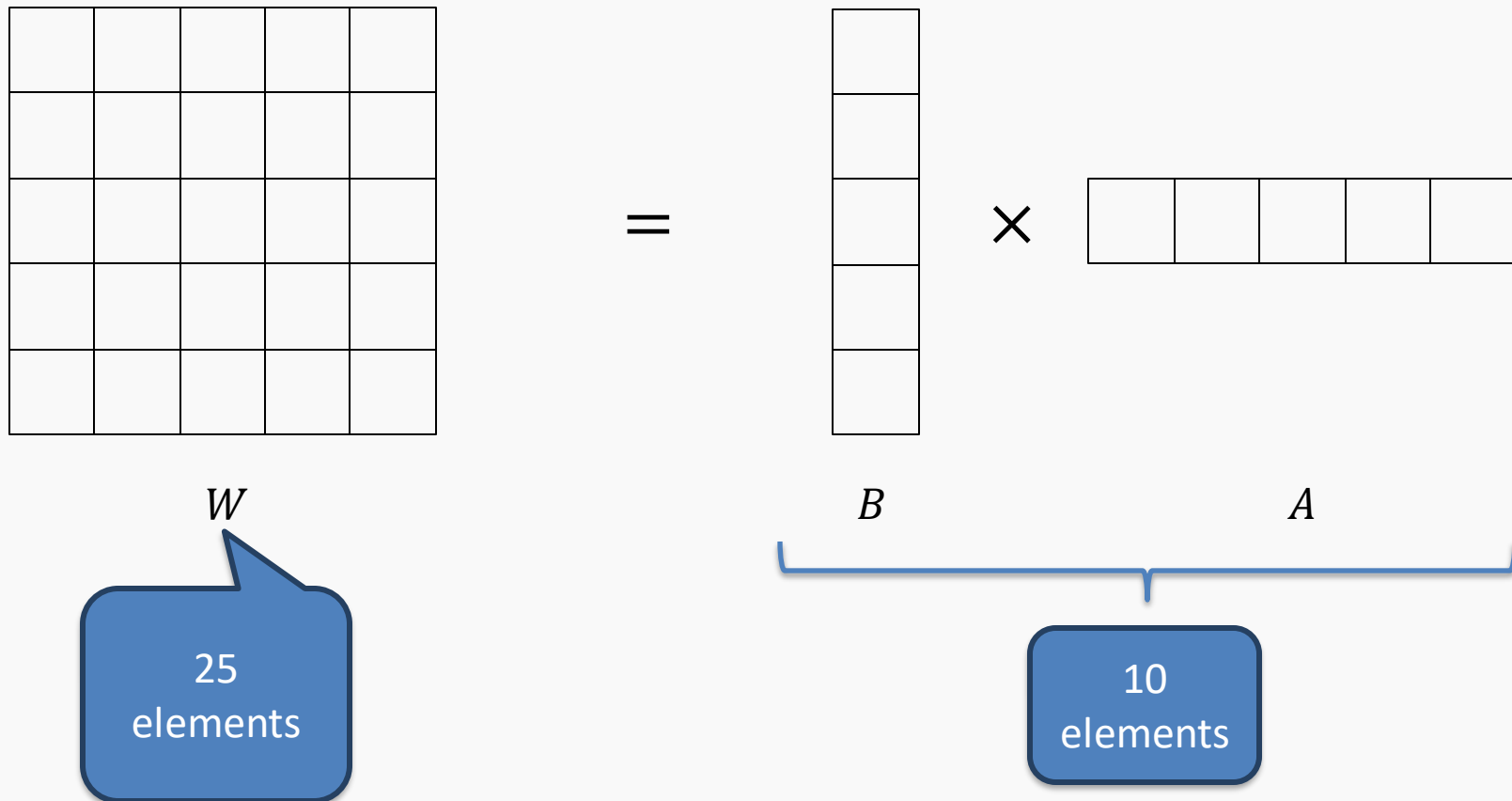  - PEFT
- **LoRA**
- QLoRA

# LoRA - Intuition

LoRA revolves around the idea that any matrix $W \in R^{m \times n}$ can be decomposed into $W = BA$ where $B \in R^{m \times r}$ and $A \in R^{r \times n}$



$W \qquad\qquad B \qquad\qquad\qquad A$

Rank 1

LoRA revolves around the idea that any matrix $W \in R^{m \times n}$ can be decomposed into $W = BA$ where $B \in R^{m \times r}$ and $A \in R^{r \times n}$



$W$

$B$      $A$

25 elements

10 elements

We can even increase the rank to get better performance.



$W$

$B$       $A$

25 elements

10 elements

# LoRA - Working

Now, we use the same concept of matrix decomposition while finetuning an LLM.

Scaling parameter

Update matrix

Decomposed matrices

Initial LLM Weights

$$W_0 + \Delta W = W_0 + \frac{\alpha}{r} BA$$

Rank of $BA$

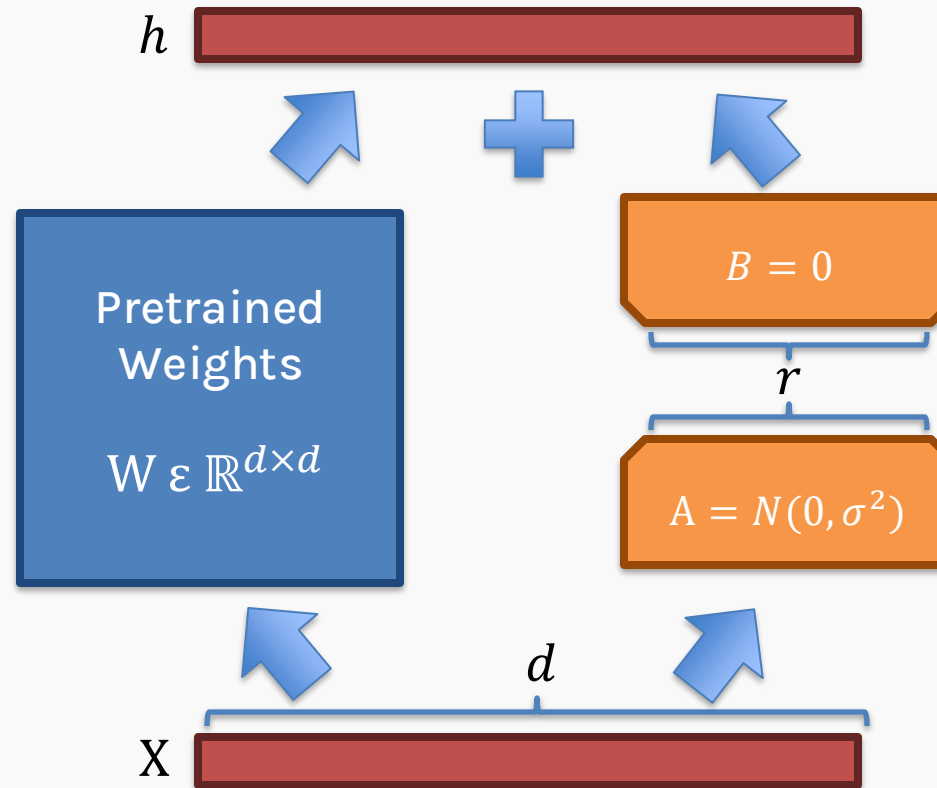Remember, we are decomposing the update matix ($\Delta W$), and not the original weights $W_0$.

# LoRA - Working

$$W_0 + \Delta W = W_0 + \frac{\alpha}{r} BA$$

We initialize B using a zero matrix, and A using a normal distribution.

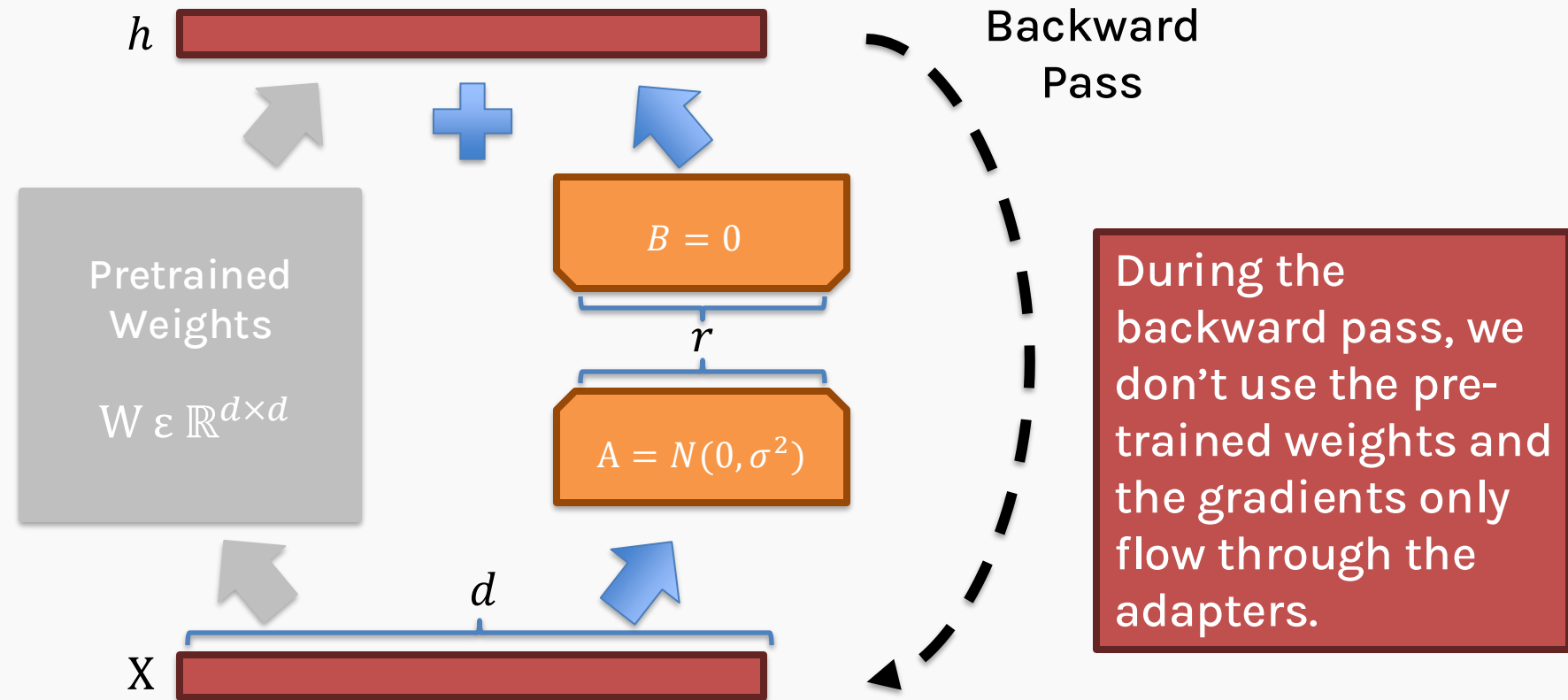Now, let's look at this diagrammatically.

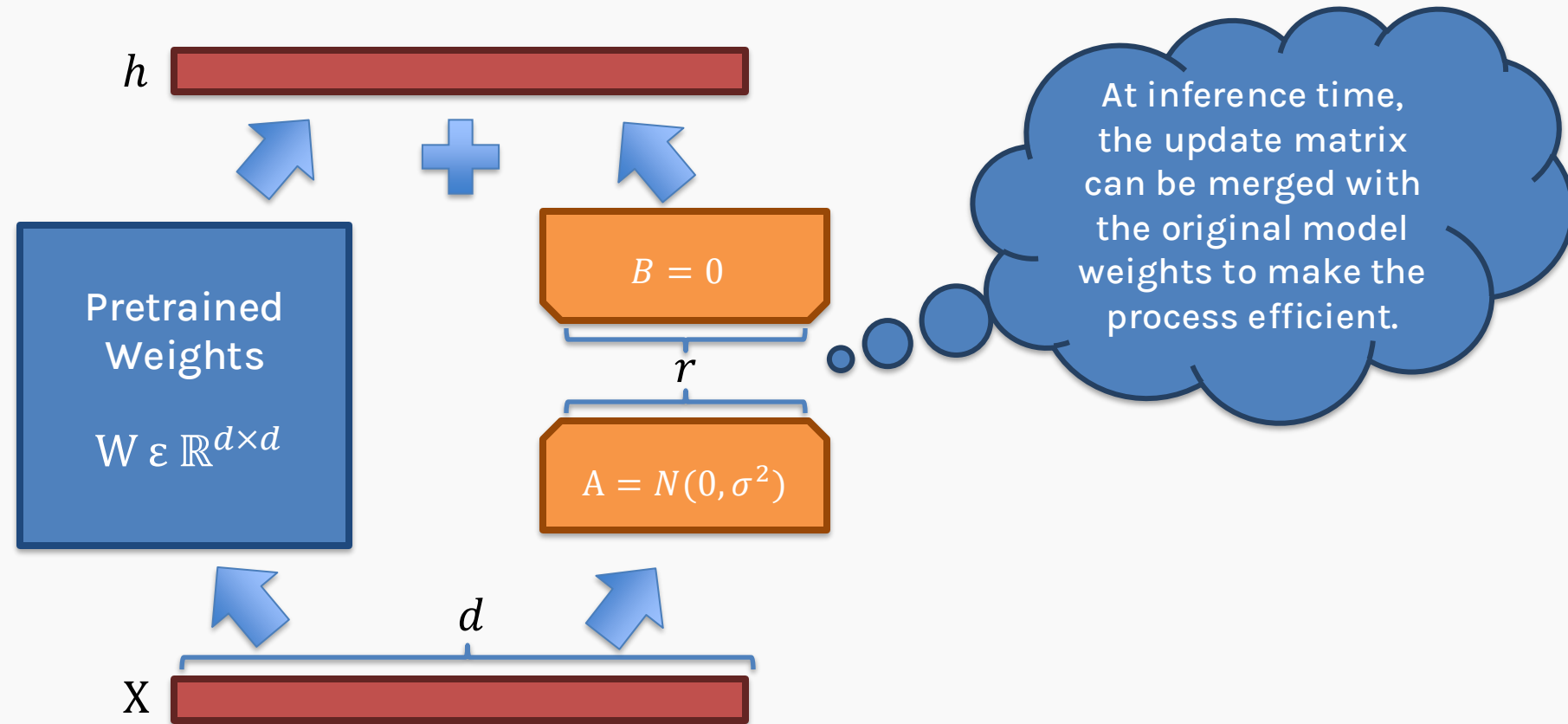Notice how the reparameterization (LoRA) runs parallel to the original model.

# LoRA - Working



Backward
Pass

$h$

Pretrained
Weights

$W \, \varepsilon \, \mathbb{R}^{d \times d}$

$B = 0$

$r$

$A = N(0, \sigma^2)$

$d$

X

During the
backward pass, we
don't use the pre-
trained weights and
the gradients only
flow through the
adapters.

Notice how the reparameterization (LoRA) runs parallel to the original model.

# LoRA - Working



Notice how the reparameterization (LoRA) runs parallel to the original model.

Let's explore the scale at which **LoRA** can help reduce the number of parameters needed to achieve comparable performance!

# LoRA - Intuition

## Number of trainable parameters

| Rank | Model 7B | Model 13B | Model 70B | Model 180B |
|------|----------|-----------|-----------|------------|
| 1    | 167K     | 228K      | 529K      | 849K       |

# LoRA - Intuition

## Number of trainable parameters

| Rank | Model 7B | Model 13B | Model 70B | Model 180B |
|------|----------|-----------|-----------|------------|
| 1 | 167K | 228K | 529K | 849K |
| 2 | 334K | 456K | 1M | 2M |

# LoRA - Intuition

## Number of trainable parameters

| Rank | Model 7B | Model 13B | Model 70B | Model 180B |
|------|----------|-----------|-----------|------------|
| 1    | 167K     | 228K      | 529K      | 849K       |
| 2    | 334K     | 456K      | 1M        | 2M         |
| 8    | 1M       | 2M        | 4M        | 7M         |

# LoRA - Intuition

## Number of trainable parameters

| Rank | Model 7B | Model 13B | Model 70B | Model 180B |
|------|----------|-----------|-----------|------------|
| 1 | 167K | 228K | 529K | 849K |
| 2 | 334K | 456K | 1M | 2M |
| 8 | 1M | 2M | 4M | 7M |
| 16 | 3M | 4M | 8M | 14M |
| 512 | 86M | 117M | 270M | 434M |
| 1024 | 171M | 233M | 542M | 869M |
| 8192 | 1.4B | 1.8B | 4.3B | 7B |
| Full | 7B | 13B | 70B | 180B |

⚠️ This is a generalization considering an LLM of one layer. LLMs are made up of multiple layers.

# LoRA - Advantages

Compared to full parameter finetuning, LoRA has the following advantages:

1. Much faster
2. Finetuning can be achieved using less GPU memory
3. Cost efficient
4. Less prone to "catastrophic forgetting" since the original model weights are kept the same.

# LoRA – Isn't it enough?

Full Parameter
Fine Tuning

Optimizer
State
(FP32)

Base Model
(FP16)

10B → 120GB

# LoRA – Isn't it enough?

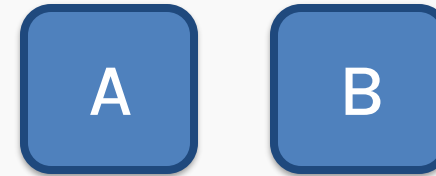## Full Parameter Fine Tuning

Optimizer State (FP32)

Base Model (FP16)

10B → 120GB

## LoRA

Optimizer State (FP32)

A    B

LoRA Adapter (FP16)

A    B
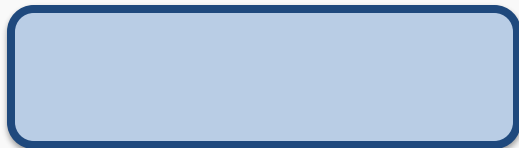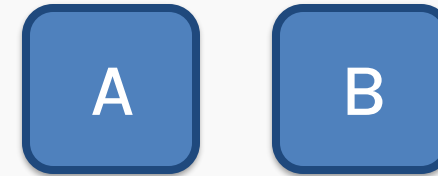
Base Model (FP16)

10B → ~40GB

# LoRA – Isn't it enough?

# LoRA – Isn't it enough?

As we can see below, LoRA's performance is comparative to full parameter fine-tuning and, in some cases, even outperforms it.

| Model & Method | # Trainable Parameters | BLEU | NIST | MET | ROUGE-L | CIDEr |
|---|---|---|---|---|---|---|
| | | | | E2E NLG Challenge | | |
| GPT-2 M (FT)* | 354.92M | 68.2 | 8.62 | 46.2 | 71.0 | 2.47 |
| GPT-2 M (Adapter$^L$)* | 0.37M | 66.3 | 8.41 | 45.0 | 69.8 | 2.40 |
| GPT-2 M (Adapter$^L$)* | 11.09M | 68.9 | 8.71 | 46.1 | 71.3 | 2.47 |
| GPT-2 M (Adapter$^H$) | 11.09M | $67.3_{\pm.6}$ | $8.50_{\pm.07}$ | $46.0_{\pm.2}$ | $70.7_{\pm.2}$ | $2.44_{\pm.01}$ |
| GPT-2 M (FT$^{Top2}$)* | 25.19M | 68.1 | 8.59 | 46.0 | 70.8 | 2.41 |
| GPT-2 M (PreLayer)* | 0.35M | 69.7 | 8.81 | 46.1 | 71.4 | 2.49 |
| GPT-2 M (LoRA) | 0.35M | $\mathbf{70.4}_{\pm.1}$ | $\mathbf{8.85}_{\pm.02}$ | $\mathbf{46.8}_{\pm.2}$ | $\mathbf{71.8}_{\pm.1}$ | $\mathbf{2.53}_{\pm.02}$ |
| GPT-2 L (FT)* | 774.03M | 68.5 | 8.78 | 46.0 | 69.9 | 2.45 |
| GPT-2 L (Adapter$^L$) | 0.88M | $69.1_{\pm.1}$ | $8.68_{\pm.03}$ | $46.3_{\pm.0}$ | $71.4_{\pm.2}$ | $\mathbf{2.49}_{\pm.0}$ |
| GPT-2 L (Adapter$^L$) | 23.00M | $68.9_{\pm.3}$ | $8.70_{\pm.04}$ | $46.1_{\pm.1}$ | $71.3_{\pm.2}$ | $2.45_{\pm.02}$ |
| GPT-2 L (PreLayer)* | 0.77M | 70.3 | 8.85 | 46.2 | 71.7 | 2.47 |
| GPT-2 L (LoRA) | 0.77M | $\mathbf{70.4}_{\pm.1}$ | $\mathbf{8.89}_{\pm.02}$ | $\mathbf{46.8}_{\pm.2}$ | $\mathbf{72.0}_{\pm.2}$ | $2.47_{\pm.02}$ |

Table 3: GPT-2 medium (M) and large (L) with different adaptation methods on the E2E NLG Challenge. For all metrics, higher is better. LoRA outperforms several baselines with comparable or fewer trainable parameters. Confidence intervals are shown for experiments we ran. * indicates numbers published in prior works.

These metrics are used for performance evaluation.

# LoRA - Summary

- LoRA reduces the trainable parameters and memory requirements while maintaining good performance.

- LoRA adds pairs of rank decomposition weight matrices (called update matrices) to each layer of the LLM.

- Only the update matrices, which have significantly fewer parameters than the original model weights, are trained.

# Outline

- Training Cycle – LLM
- Instruction-tuning
  - Full Parameter
  - PEFT
- LoRA
- **QLoRA**

# QLoRA

- QLoRA is the extended version of LoRA which works mainly by quantizing the precision of the original network parameters.

- Before we dive into what QLoRA is, let's look at what quantization is.

Think of quantization as ' splitting range into buckets '.

# QLoRA

Think of quantization as ' splitting range into buckets '.
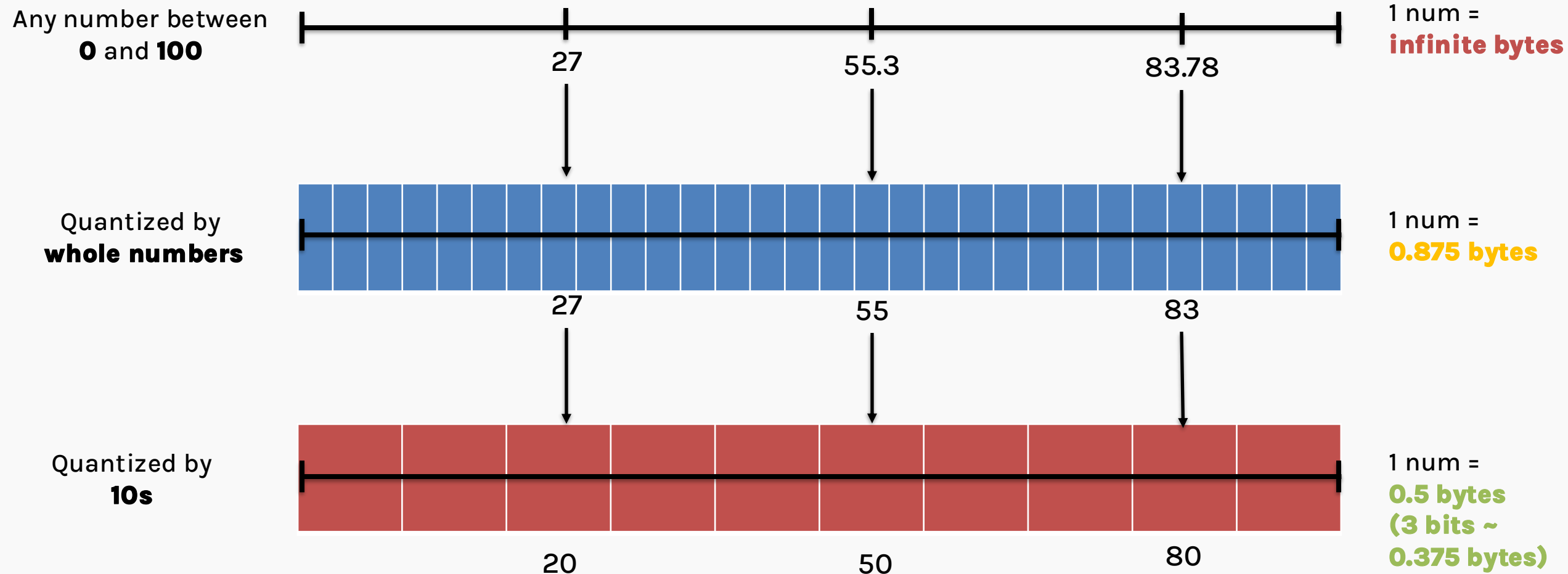
# QLoRA

Let's look at an example!

Let $X^{FP32}$ be an array of values.

Here, FP32 refers to a 32-bit floating-point number.

| 1.5 | 2.3 | 3.7 | 4.1 | 5.6 | 6.8 | 7.9 | 8.4 | 9.2 | 10.2 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|

What if we want to quantize from FP32 to Int8?

# QLoRA

So, to quantize $X^{FP32}$ to $X^{Int8}$ :
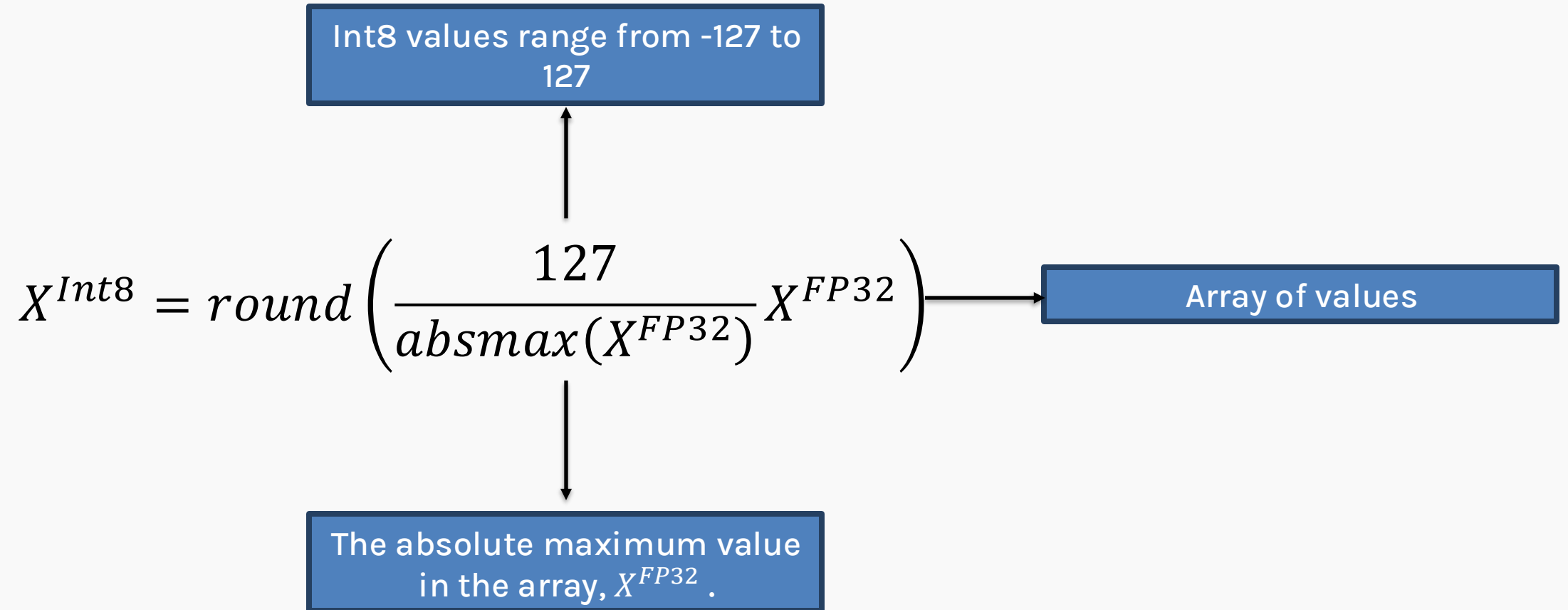
Int8 values range from -127 to 127

$$X^{Int8} = round\left(\frac{127}{absmax(X^{FP32})} X^{FP32}\right)$$

Array of values

The absolute maximum value in the array, $X^{FP32}$ .

# QLoRA

So, to quantize $X^{FP32}$ to $X^{Int8}$ :

$$X^{Int8} = round\left(\frac{127}{absmax(X^{FP32})}X^{FP32}\right)$$

$$\downarrow$$

$$X^{Int8} = round(c^{FP32}X^{FP32})$$

# QLoRA

So, to quantize $X^{FP32}$ to $X^{Int8}$ :

$$X^{Int8} = round\left(\frac{127}{absmax(X^{FP32})}X^{FP32}\right)$$

$$\downarrow$$

$$X^{Int8} = round(c^{FP32}X^{FP32})$$

# QLoRA

$$X^{Int8} = round(c^{FP32}X^{FP32})$$

In our example,

$X^{FP32}$ →

| 1.5 | 2.3 | 3.7 | 4.1 | 5.6 | 6.8 | 7.9 | 8.4 | 9.2 | 10.2 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|
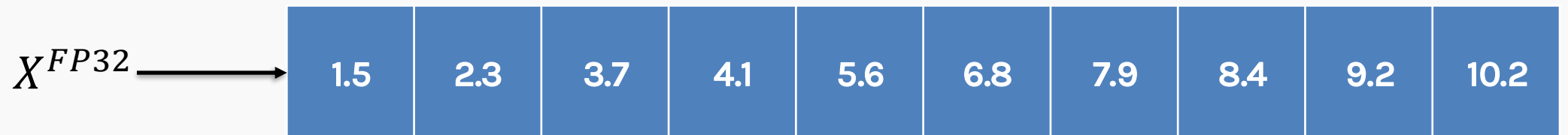
$$c^{FP32} = \frac{127}{absmax(X^{FP32})} = \frac{127}{10.2} = 12.4509$$

Now, we combine the formula and the values that we have

# QLoRA

$$X^{Int8} = round(12.4509 \text{ x } \boxed{1.5 \mid 2.3 \mid 3.7 \mid 4.1 \mid 5.6 \mid 6.8 \mid 7.9 \mid 8.4 \mid 9.2 \mid 10.2})$$

$$X^{Int8} = \boxed{18 \mid 29 \mid 46 \mid 51 \mid 69 \mid 85 \mid 98 \mid 105 \mid 115 \mid 127}$$

Voila! That's how we quantize from FP32 to Int8 using the formula:

$$X^{Int8} = round(c^{FP32}X^{FP32})$$

# QLoRA
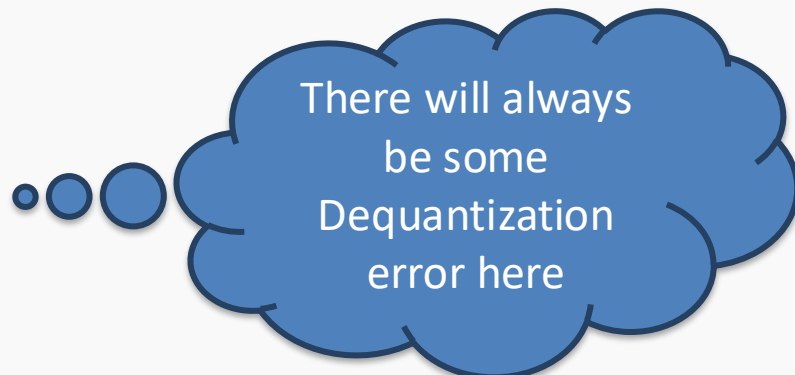
$$X^{Int8} = round(c^{FP32} X^{FP32})$$

What if we want to dequantize and get back the original array, $X^{FP32}$?

To dequantize:

$$X^{FP32} = \frac{X^{Int8}}{c^{FP32}}$$

There will always be some Dequantization error here

Now that we know what quantization is, let's look at how QLoRA works!

# QLoRA – The Pizza

Imagine QLoRA to be a mouthwatering pizza.



Now, to make a pizza, we need to gather a few key ingredients!

# QLoRA – The Ingredients
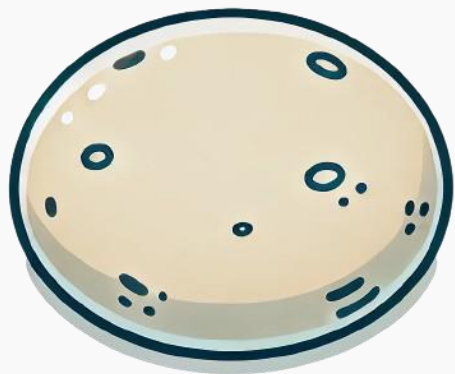
There are 3 key ingredients which helps us make QLoRA:



**4-Bit NormalFloat**
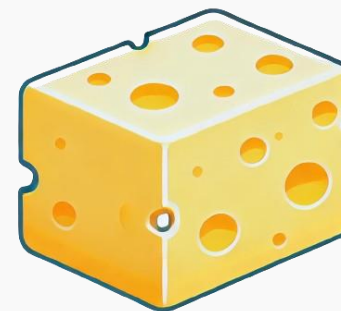


**Double Quantization**



**Paged Optimizer**

## 4-bit NormalFloat

4-bit NormalFloat is a clever way to split the buckets.

4-bit means we have $2^4 = 16$ possible buckets for quantization.
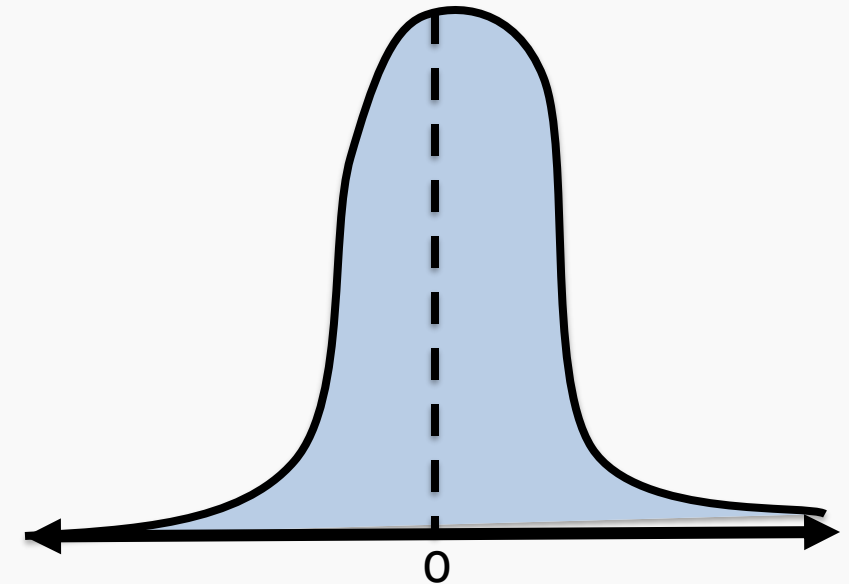
Equally spaced buckets

Equally sized buckets

This is an enhanced version of **quantile quantization**.
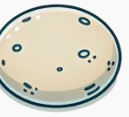
## Why use 4-bit NormalFloat

Designed for efficient storage and computation in machine learning.

Most datasets in machine learning are normally distributed and precision around the mean is valuable.
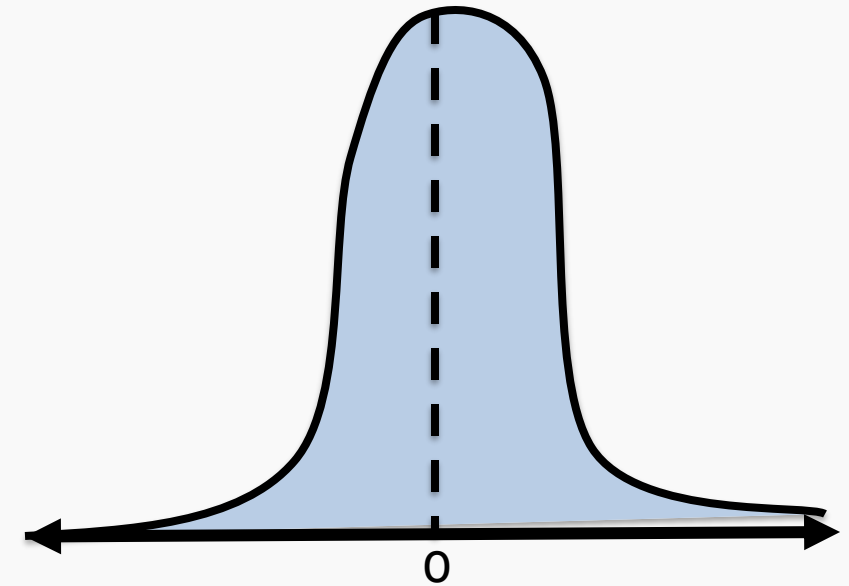


Equally spaced buckets

Equally sized buckets

This is an enhanced version of **quantile quantization**.

# QLoRA – The Ingredients
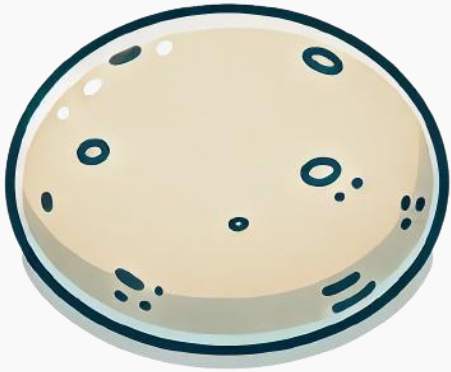
There are 3 key ingredients which helps us make QLoRA:



**4-Bit NormalFloat**       **Double Quantization**       **Paged Optimizer**
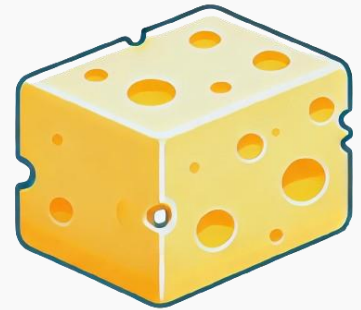
Remember this formula?

Takes up memory

$$X^{Int8} = round(c^{FP32} X^{FP32})$$

Which is not an issue, as it's just 1 constant. Right?

Now, if we think about this in terms of neural networks....

# QLoRA – Ingredient 2: Double Quantization

Now, if we think about this in terms of neural networks....

Let's take a 5x5 matrix to be the weights in a neural network:

Weight Tensor

| | | | | |
|------|------|------|------|------|
| -0.7 | -0.3 | 0.0 | -0.4 | 0.3 |
| -1.0 | 0.2 | 0.7 | 1.7 | -0.9 |
| -0.1 | -1.5 | -0.1 | 0.8 | 0.5 |
| 1.2 | -1.7 | -0.9 | -0.3 | 0.7 |
| 0.4 | 0.1 | -1.4 | 2.2 | -1.1 |

Now, if we think about this in terms of neural networks....

Weight Tensor

| -0.7 | -0.3 | 0.0 | -0.4 | 0.3 |
|------|------|------|------|------|
| -1.0 | 0.2 | 0.7 | 1.7 | -0.9 |
| -0.1 | -1.5 | -0.1 | 0.8 | 0.5 |
| 1.2 | -1.7 | -0.9 | -0.3 | 0.7 |
| 0.4 | 0.1 | -1.4 | 2.2 | -1.1 |

Rescale all parameters

$using\ c$

Rescaled Weight Tensor

| -4 | -2 | 0 | -22 | 16 |
|------|------|------|------|------|
| -60 | 10 | 40 | 99 | -57 |
| -5 | -88 | -9 | 48 | 27 |
| 72 | -100 | -50 | -18 | 40 |
| 22 | 8 | -81 | 127 | -66 |

If we bring back the formula: $round(W^{FP32}c^{FP32}) = W^{Int8}$

Now, if we think about this in terms of neural networks….

**Weight Tensor**

| | | | | |
|---|---|---|---|---|
| -0.7 | -0.3 | 0.0 | -0.4 | |
| -1.0 | 0.2 | 0.7 | 1.7 | |
| -0.1 | -1.5 | -0.1 | 0.8 | |
| 1.2 | -1.7 | -0.9 | -0.3 | |
| 0.4 | 0.1 | -1.4 | 2.2 | -1.1 |

**Rescaled Weight Tensor**

| | | | | |
|---|---|---|---|---|
| -4 | -2 | 0 | -22 | 16 |
| | | | 99 | -57 |
| | | | 48 | 27 |
| | | 50 | -18 | 40 |
| 22 | 8 | -81 | 127 | -66 |

⚠️ **We quantize to Int8 for simplicity but when we implement QLoRA we use 4-bit Normal Float.**

If we bring back the formula: $round(W^{FP32} c^{FP32}) = W^{Int8}$

Let's see how the weight tensors look like on the graph.



+10000

0
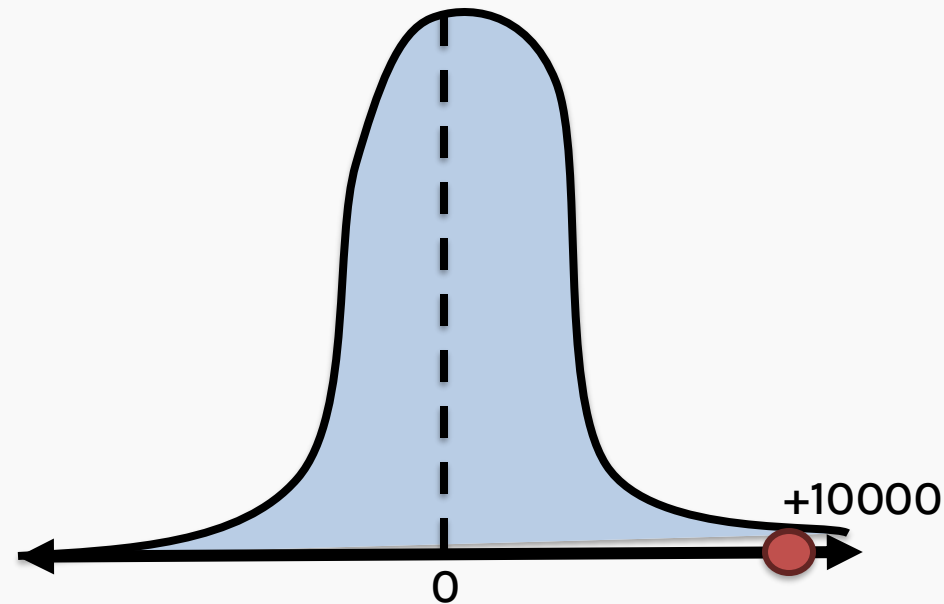
This is unbounded and could take up any maximum value (an outlier!).

$$W^{Int8} = round(\frac{127}{absmax(W^{FP32})}W^{FP32})$$

Let's see how the weight tensors look like on the graph.



+10000

0
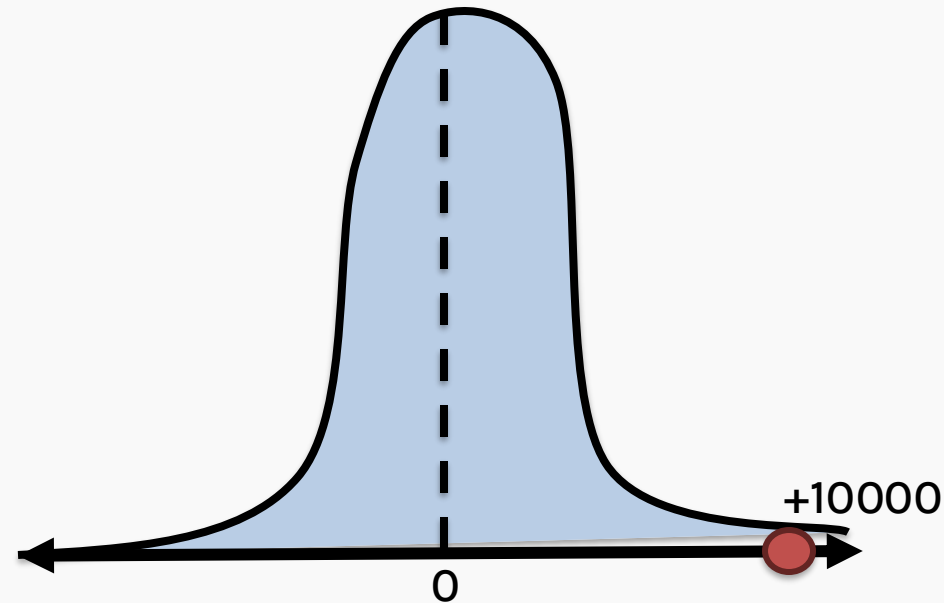
This is unbounded and could take up any maximum valu

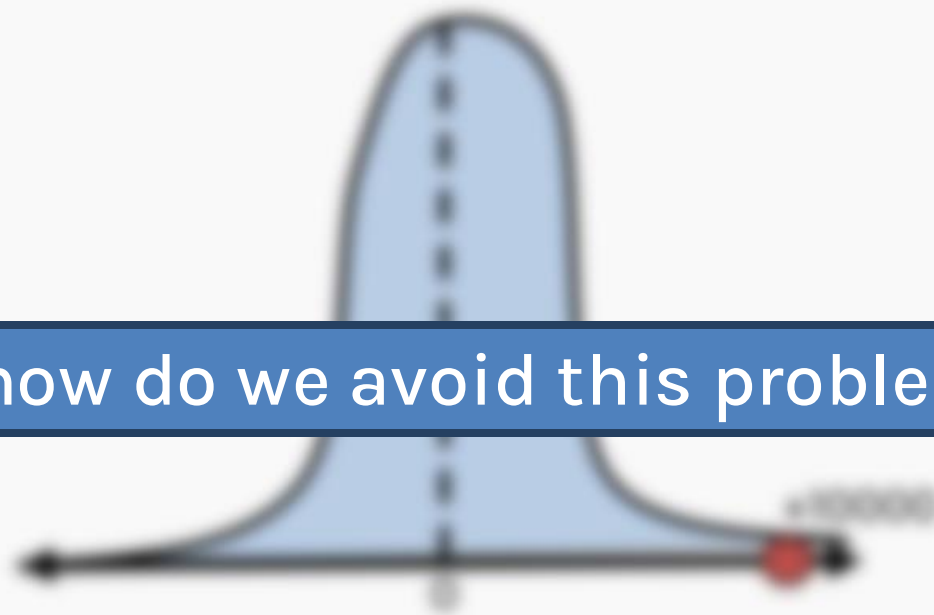This could introduce **bias** in our quantization process

$$W^{Int8} = round(\frac{127}{absmax(W^{FP32})}W^{FP32})$$

The answer to that is: **Block-wise Quantization,** which is the first step in Double Quantization!

Let's look at an example to understand this concept.

We take the weight tensor that we saw in the previous slides.

Weight Tensor ($W^{FP32}$)

| | | | | |
|---|---|---|---|---|
| -0.7 | -0.3 | 0.0 | -0.4 | 0.3 |
| -1.0 | 0.2 | 0.7 | 1.7 | -0.9 |
| -0.1 | -1.5 | -0.1 | 0.8 | 0.5 |
| 1.2 | -1.7 | -0.9 | -0.3 | 0.7 |
| 0.4 | 0.1 | -1.4 | 2.2 | -1.1 |

# QLoRA – Ingredient 2 : Double Quantization

We flatten the matrix as follows:

Now we divide it up into different blocks.

We calculate the quantization constants for each block.

Weight Tensor ($W^{FP32}$)

| -0.7 | -0.3 | 0.0 | -0.4 | 0.3 | -1.0 | 0.2 | 0.7 | 1.7 | -0.9 | -0.1 | -1.5 | -0.1 | 0.8 | 0.5 | 1.2 | -1.7 | -0.9 | -0.3 | 0.7 | 0.4 | 0.1 | -1.4 | 2.2 | -1.1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$c_i$ $\qquad c_j \qquad c_k \qquad c_l \qquad c_m$
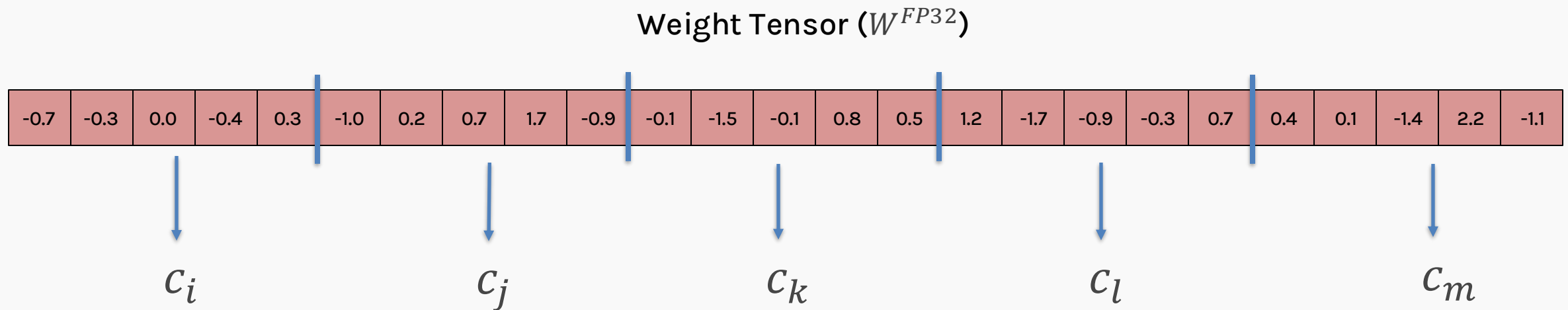
If there are any outliers in a block, they won't affect the quantisation in the other blocks.

$$C^{Int8} = round(\frac{127}{absmax(C^{FP32})}C^{FP32})$$

**Weight Tensor ($W^{FP32}$)**

| -0.7 | -0.3 | 0.0 | -0.4 | 0.3 | -1.0 | 0.2 | 0.7 | 1.7 | -0.9 | -0.1 | -1.5 | -0.1 | 0.8 | 0.5 | 1.2 | -1.7 | -0.9 | -0.3 | 0.7 | 0.4 | 0.1 | -1.4 | 2.2 | -1.1 |

$c_i$

181.4

$c_j$

74.7

$c_k$

84.7

$c_l$

74.7

$c_m$

57.7

We now rescale all the parameters per block.

| -127 | -54 | 0 | -73 | 54 | -77 | 13 | 51 | 127 | -74 | -9 | -127 | -12 | 70 | 40 | 93 | -127 | -64 | -24 | 52 | 22 | 8 | -81 | 127 | -66 |

**Rescaled Weight Tensor ($W^{Int8}$)**
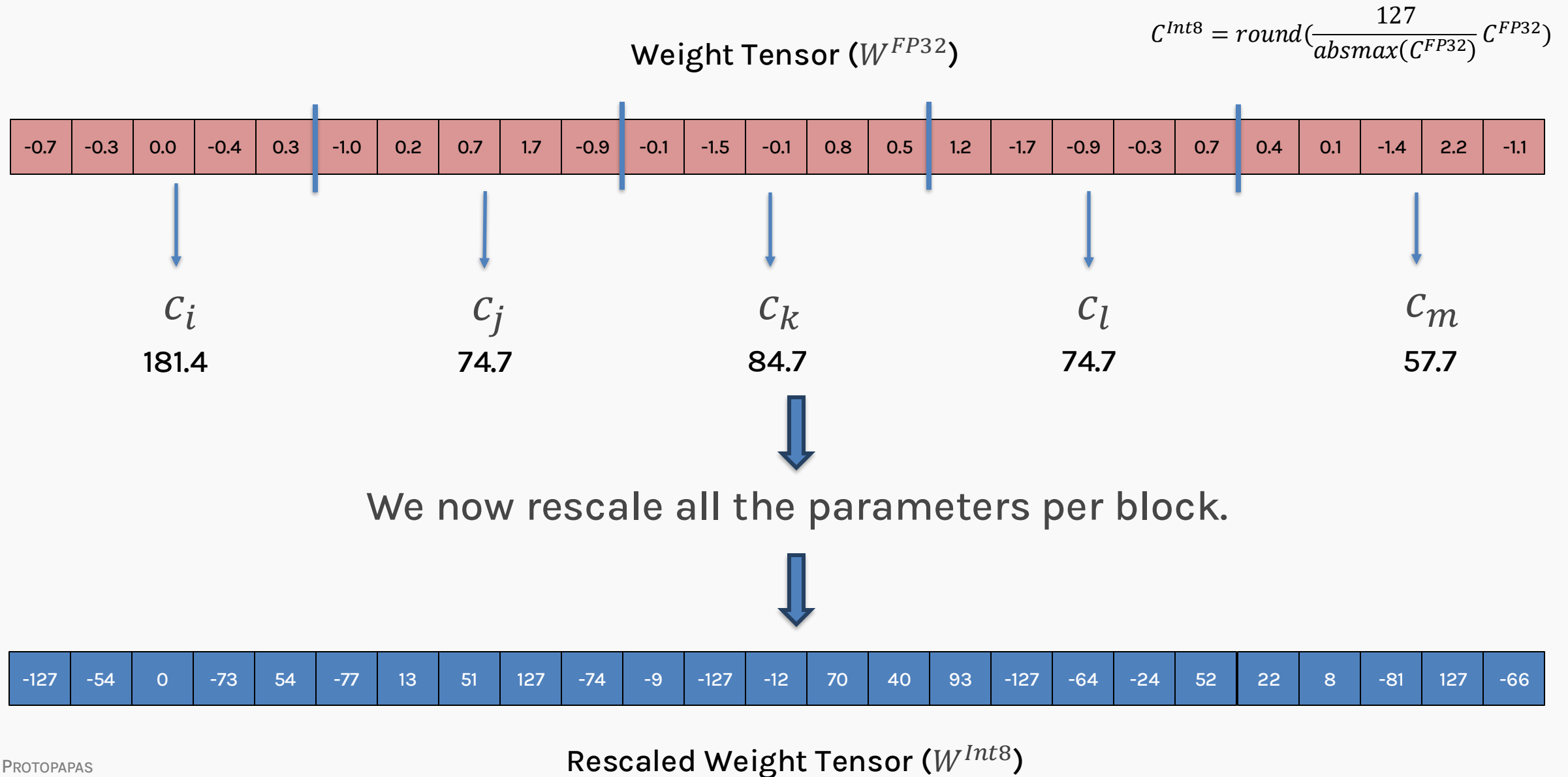
We now have a new array:

$$c_1^{FP32} \longrightarrow$$

| $c_i$ | $c_j$ | $c_k$ | $c_l$ | $c_m$ |
|---|---|---|---|---|
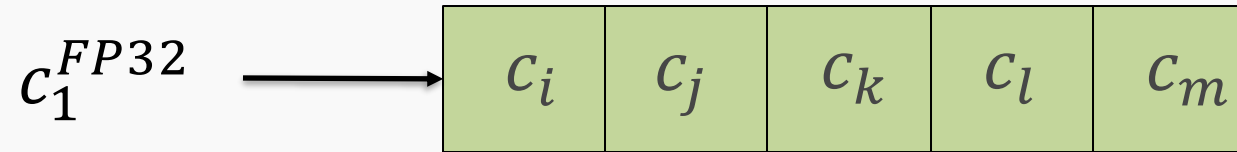
$c_1^{FP32}$ is an array of all the constants from each block of the Weight Tensor.

Now, we repeat the same process of quantization for the quantization constants.

$$c_1^{Int8} = round(\frac{127}{absmax(c_1^{FP32})} c_1^{FP32})$$

$$c_1^{Int8} = round(c_2^{FP32} c_1^{FP32})$$

Double Quantization

$$c_1^{Int8} = round(c_2^{FP32} c_1^{FP32})$$

Let's see the difference in memory usage before and after

Double Quantization.

# QLoRA – Ingredient 2 : Double Quantization

## Before

All we had was a weight matrix containing FP32 values.

In our example, we had a 5x5 matrix.

Each value was 4 bytes in size.

So, the total memory used was:        **25x4=100 bytes**

| -0.7 | -0.3 | 0.0 | -0.4 | 0.3 |
|------|------|-----|------|-----|
| -1.0 | 0.2 | 0.7 | 1.7 | -0.9 |
| -0.1 | -1.5 | -0.1 | 0.8 | 0.5 |
| 1.2 | -1.7 | -0.9 | -0.3 | 0.7 |
| 0.4 | 0.1 | -1.4 | 2.2 | -1.1 |

Weight Tensor ($W^{FP32}$)

Next, let's look at the memory usage **after** Double Quantization.

# QLoRA – Ingredient 2 : Double Quantization

**After**

| -127 | -54 | 0 | -73 | 54 |
|------|-----|-----|-----|-----|
| -77 | 13 | 51 | 127 | -74 |
| -9 | -127 | -12 | 70 | 40 |
| 93 | -127 | -64 | -24 | 52 |
| 22 | 8 | -81 | 127 | -66 |

Rescaled Weight Tensor
($W^{Int8}$)

25x1=25 bytes.

| $c_i$ | $c_j$ | $c_k$ | $c_l$ | $c_m$ |
|-------|-------|-------|-------|-------|

$c_1^{Int8}$

5x1=5bytes.

$c_2^{FP32}$

4 bytes

So, in total:

25 + 5 + 4 = **34 bytes**

**After**

| | | | | |
|---|---|---|---|---|
| -127 | -54 | 0 | -73 | 54 |
| -77 | 13 | 51 | 127 | -74 |
| -9 | -127 | -12 | 70 | 40 |
| 93 | -127 | -64 | -24 | 52 |
| 22 | 8 | -81 | 127 | -66 |

| $c_i$ | $c_j$ | $c_k$ | $c_l$ | $c_m$ |
|---|---|---|---|---|

$$c_2^{FP32}$$

25 + 5 + 4 = **34 bytes**

**Before**

| | | | | |
|---|---|---|---|---|
| -0.7 | -0.3 | 0.0 | -0.4 | 0.3 |
| -1.0 | 0.2 | 0.7 | 1.7 | -0.9 |
| -0.1 | -1.5 | -0.1 | 0.8 | 0.5 |
| 1.2 | -1.7 | -0.9 | -0.3 | 0.7 |
| 0.4 | 0.1 | -1.4 | 2.2 | -1.1 |

25x4=100 bytes

That is an approximate 70% reduction in memory usage!!

# QLoRA – The Ingredients

There are 3 key ingredients which helps us make QLoRA:
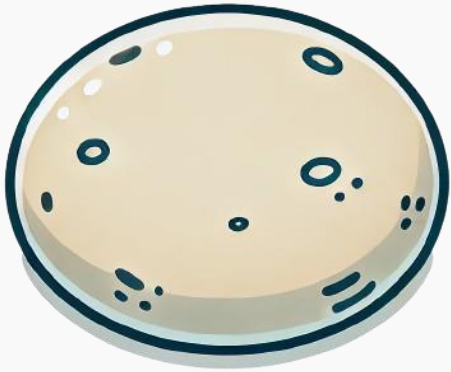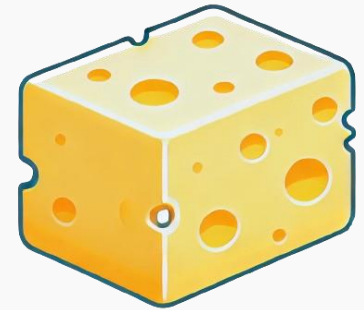
4-Bit NormalFloat          Double Quantization          Paged Optimizer
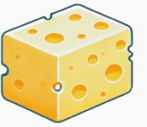
Before we talk about the third ingredient in QLoRA, let's talk about a problem.

A problem which all of us have faced while training a Neural Network

Running Out of Memory!

So, how do we train a modern Neural Networks without taking a hit on the memory?

We use gradient checkpointing.
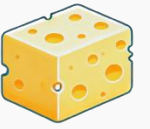
Imagine this simple neural network
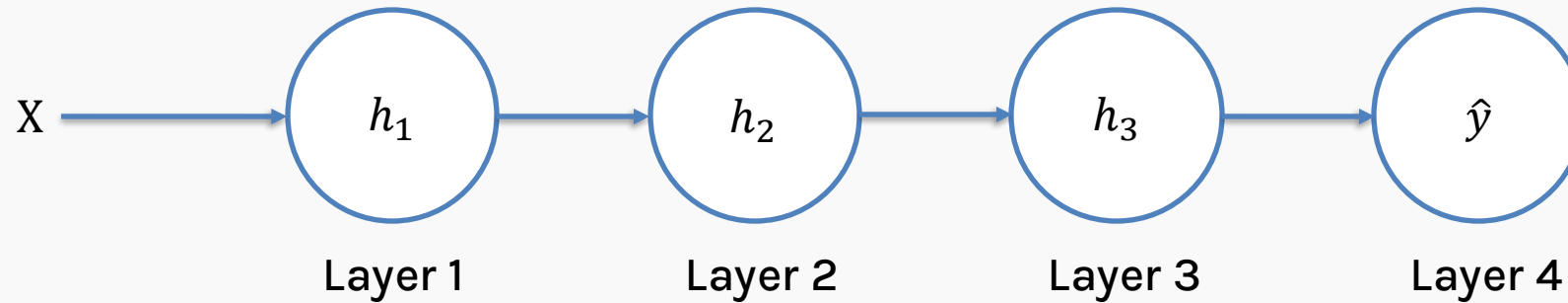


When we do a forward-pass, we calculate the activations for each layer.
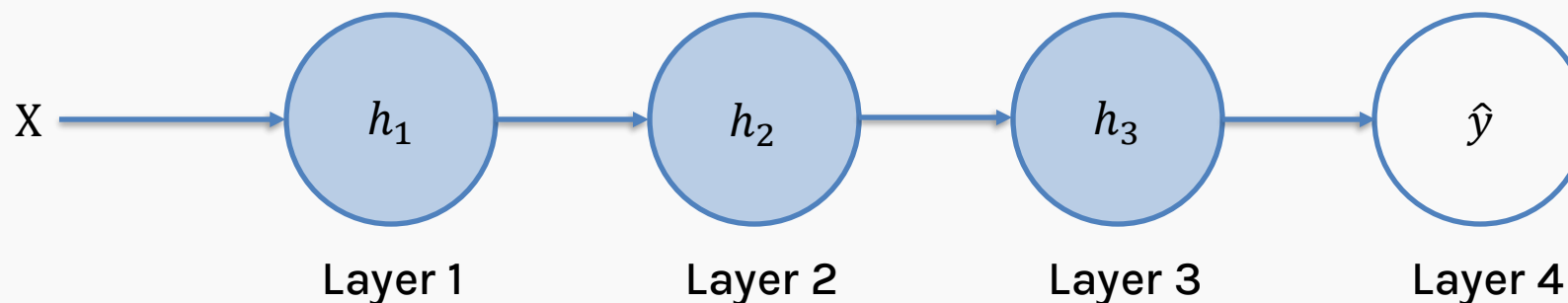
However, this takes up precious memory.

Modern-day computers have become very efficient at parallel processing. What they lack is memory.

We don't need to store all the hidden states.
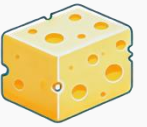
We only store in memory what is needed at the moment.

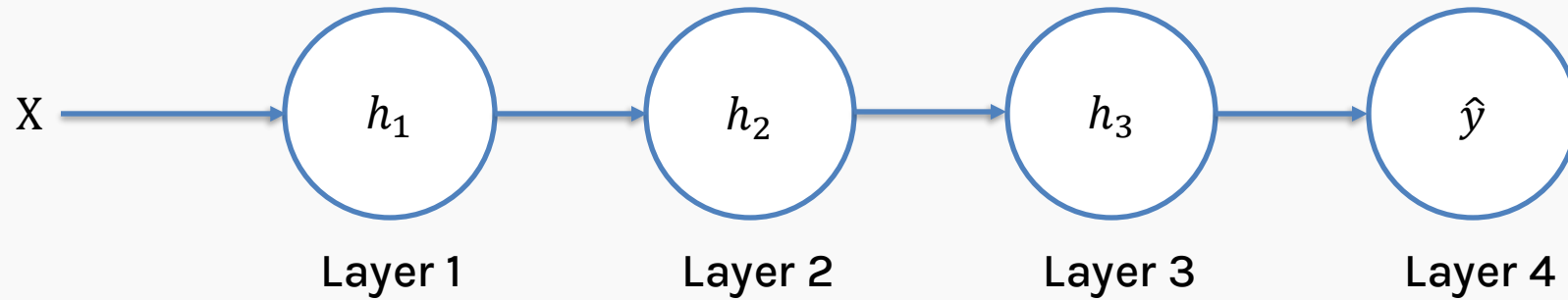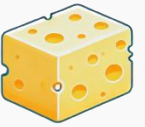We keep discarding activations that have already been used to calculate the next dependent hidden state's activation.

So, let's see how it looks!



$X \rightarrow$ Layer 1: $h_1$ → Layer 2: $h_2$ → Layer 3: $h_3$ → Layer 4: $\hat{y}$

X → Layer 1 → Layer 2 → Layer 3 → Layer 4 ($\hat{y}$)
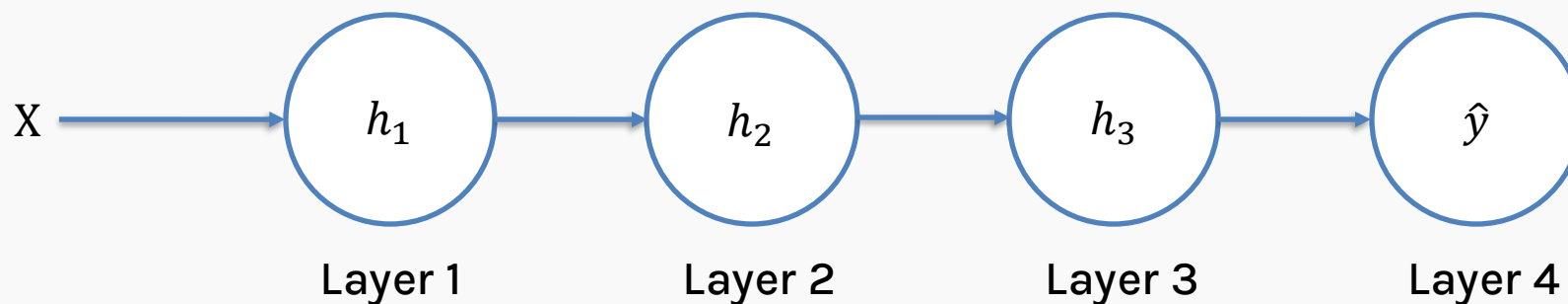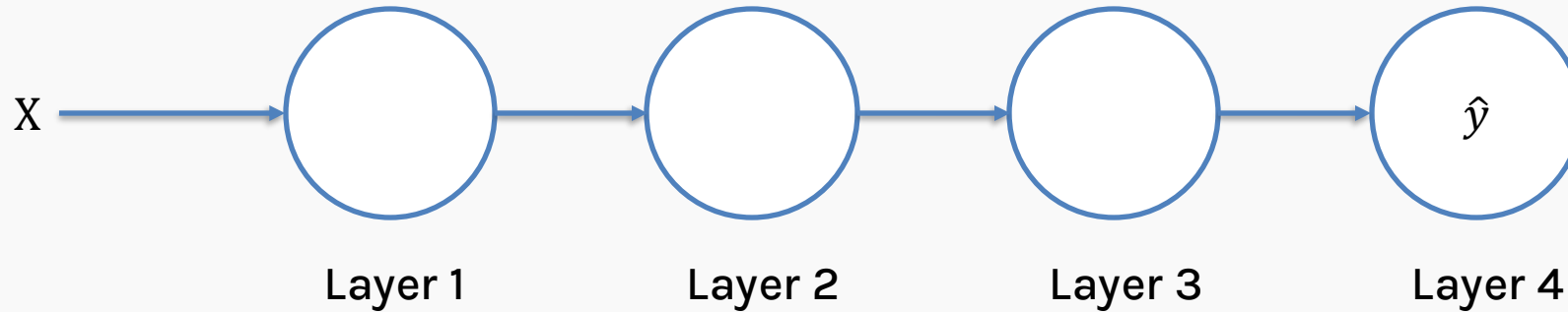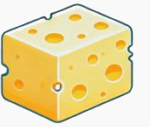
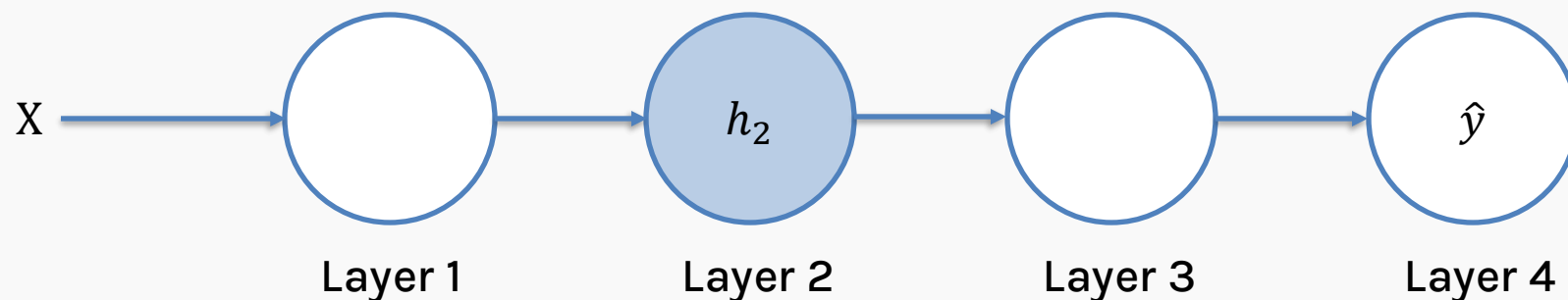During backpropagation, we must recompute all the discarded activations.

To manage this, we introduce checkpoints in the middle.

Checkpoints are usually placed at every $\sqrt{n}$ layer, considering we have a n-layer neural network.
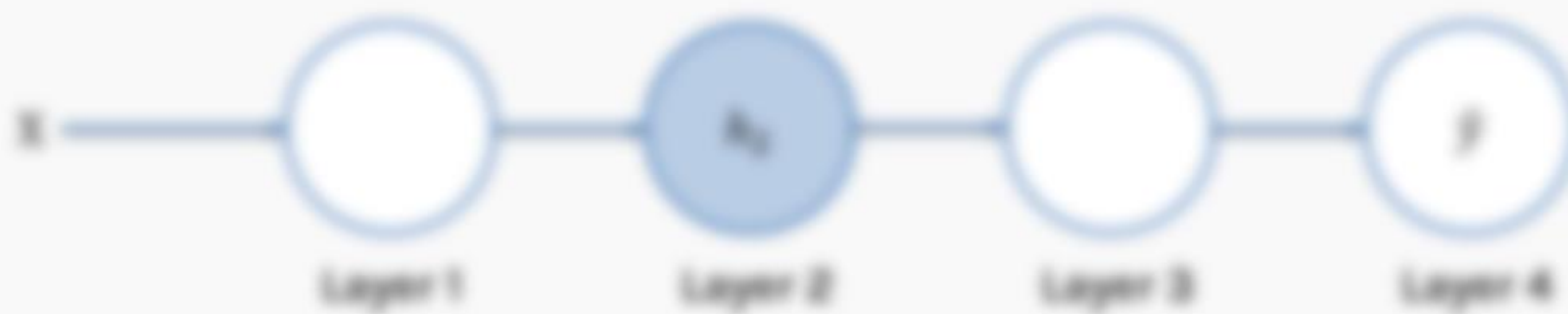
So, now when we re-compute the activations for backward pass, we don't have to start from the beginning!

This allows us to mitigate the OOM (Out of memory) error to some extent, but it doesn't get rid of it!

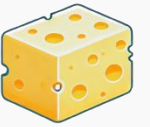We still see some memory spikes especially when we pass in long sequences in the batch.

This is where our third ingredient comes in!

This allows us to mitigate the OOM (out of memory) error to some extent, but it doesn't get rid of it!

We still see some memory spikes especially when we pass in long sequences in the batch.

# QLoRA – Ingredient 3 : Paged Optimizer

**Paged Optimizer -** Looping in your CPU

Paging is a memory management technique, where RAM is divided into fixed-size blocks called 'pages'
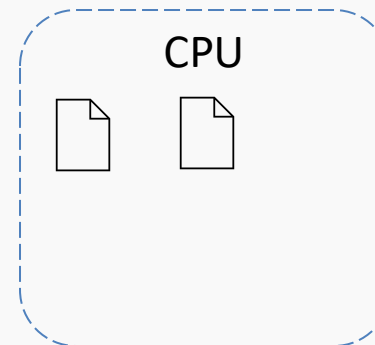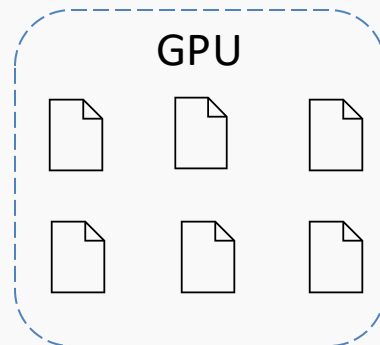
It does automatic page-to-page transfers between CPU and GPU

Avoids the gradient checkpointing memory spikes that occur when processing a mini batch with a long sequence length.
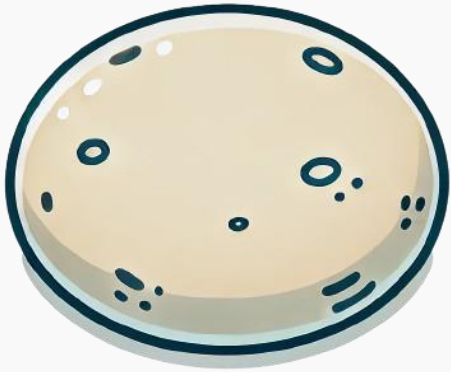
GPU Memory has space now.

GPU

CPU

Now that the GPU has space, when a page moved to CPU is required, we move it back to GPU for computation.
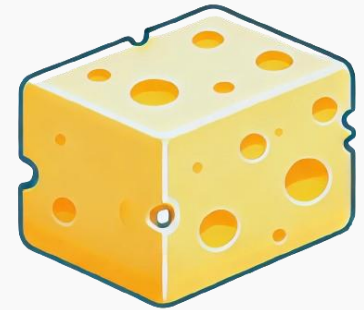
# QLoRA – The Ingredients

We saw the 3 key ingredients needed to make QLoRA:



**4-Bit NormalFloat**


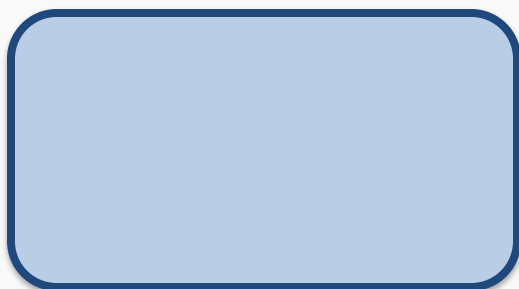
**Double Quantization**



**Paged Optimizer**

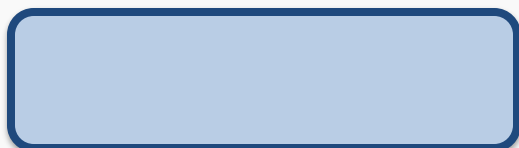Let's bring it all together.

# QLoRA – Putting it all together

## Full Parameter Fine Tuning

Optimizer State (FP32)

Base Model

FP16

10B => 120GB

## LoRA

Optimizer State (FP32)

| A | B |

LoRA Adapter (FP16)

| A | B |

Base Model

FP16

10B => ~40GB

Before we talk about the 3 ingredients, there is another key difference that we should know.

In QLoRA we use BF16 (BrainFloat16) as compared to FP16 in LoRA.

This leads to a change in precision which is tailor-made for deep learning tasks.

$h$

$B = 0$

**BF16**

Pretrained Weights

$W \; \varepsilon \; \mathbb{R}^{d \times d}$
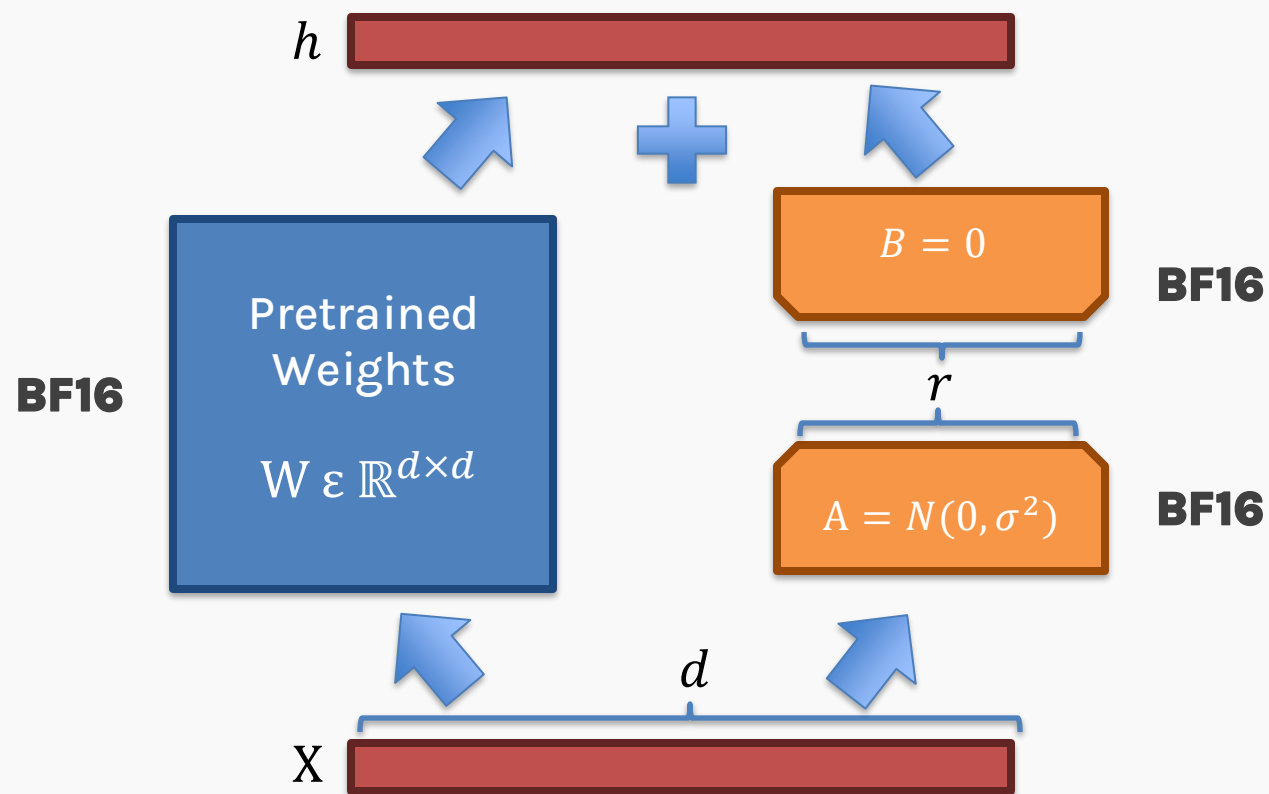
**BF16**

$r$

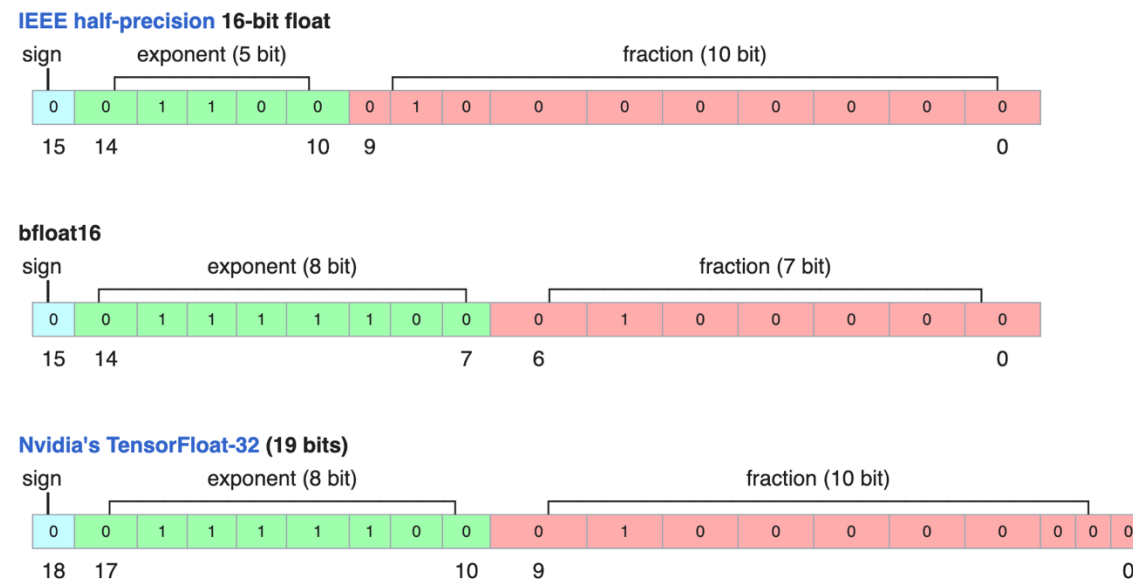$A = N(0, \sigma^2)$

**BF16**

$d$

X
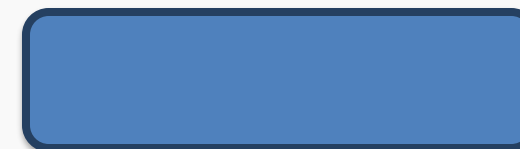
# QLoRA – Putting it all together

Before we talk about the 3 ingredients, there is another key difference that we should know.

In QLoRA we use **BF16 (BrainFloat16)** as compared to FP16 in LoRA.

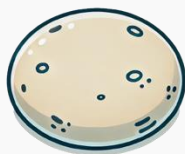This leads to a change in precision which is tailor-made for deep learning tasks.



**IEEE half-precision** 16-bit float

sign    exponent (5 bit)    fraction (10 bit)

| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

15  14          10  9                                        0

**bfloat16**

sign    exponent (8 bit)    fraction (7 bit)

| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

15  14                7  6                        0

**Nvidia's TensorFloat-32** (19 bits)

sign    exponent (8 bit)    fraction (10 bit)

| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

18  17                10  9                                    0

A 16-bit floating-point format keeping a wide range like 32-bit floats but with lower precision to speed up training.

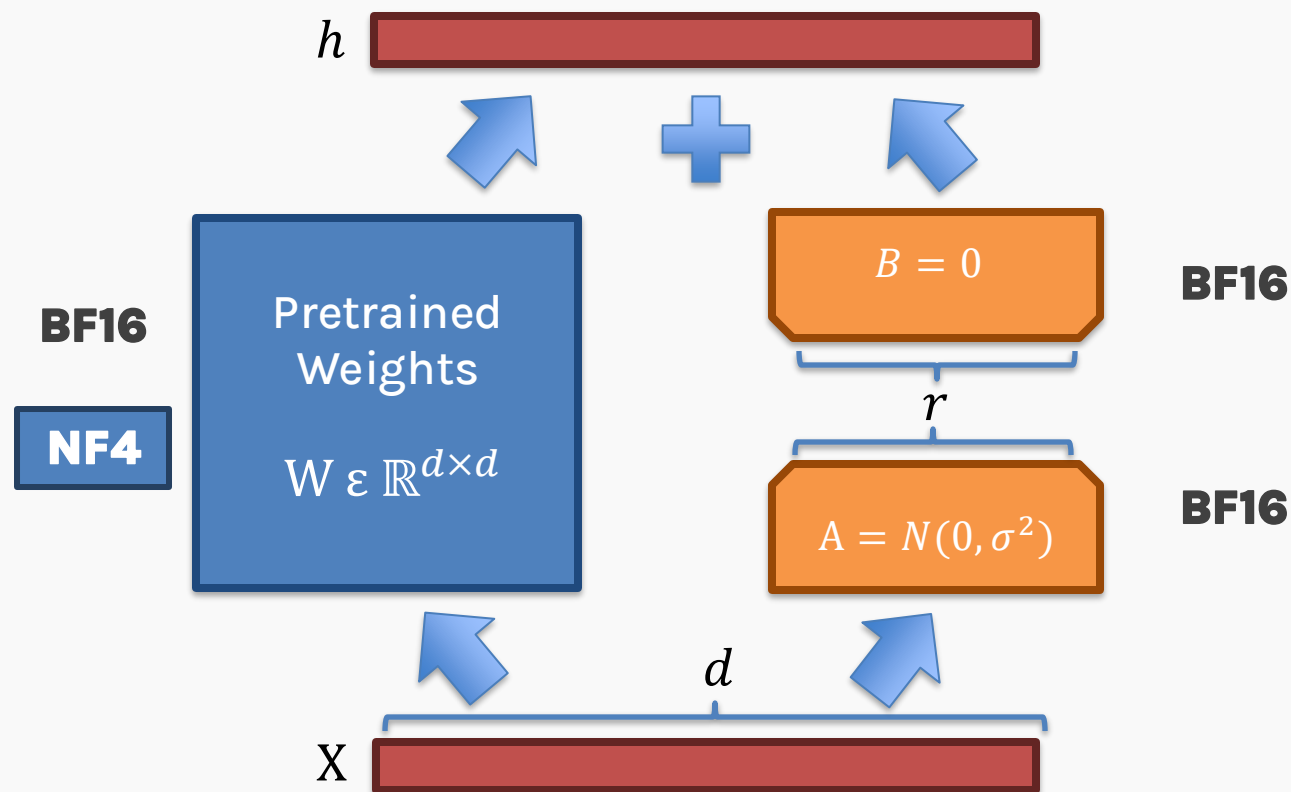Image Source

# QLoRA – Putting it all together

**Ingredient 1:**

4-Bit NormalFloat

**Ingredient 2:**

Double Quantization

$h$

**BF16**

**NF4**

Pretrained
Weights

$\mathbb{W} \, \varepsilon \, \mathbb{R}^{d \times d}$

$B = 0$     **BF16**

$r$

$A = N(0, \sigma^2)$     **BF16**

$d$

X

To convert and store, we make use of
Double Quantization!

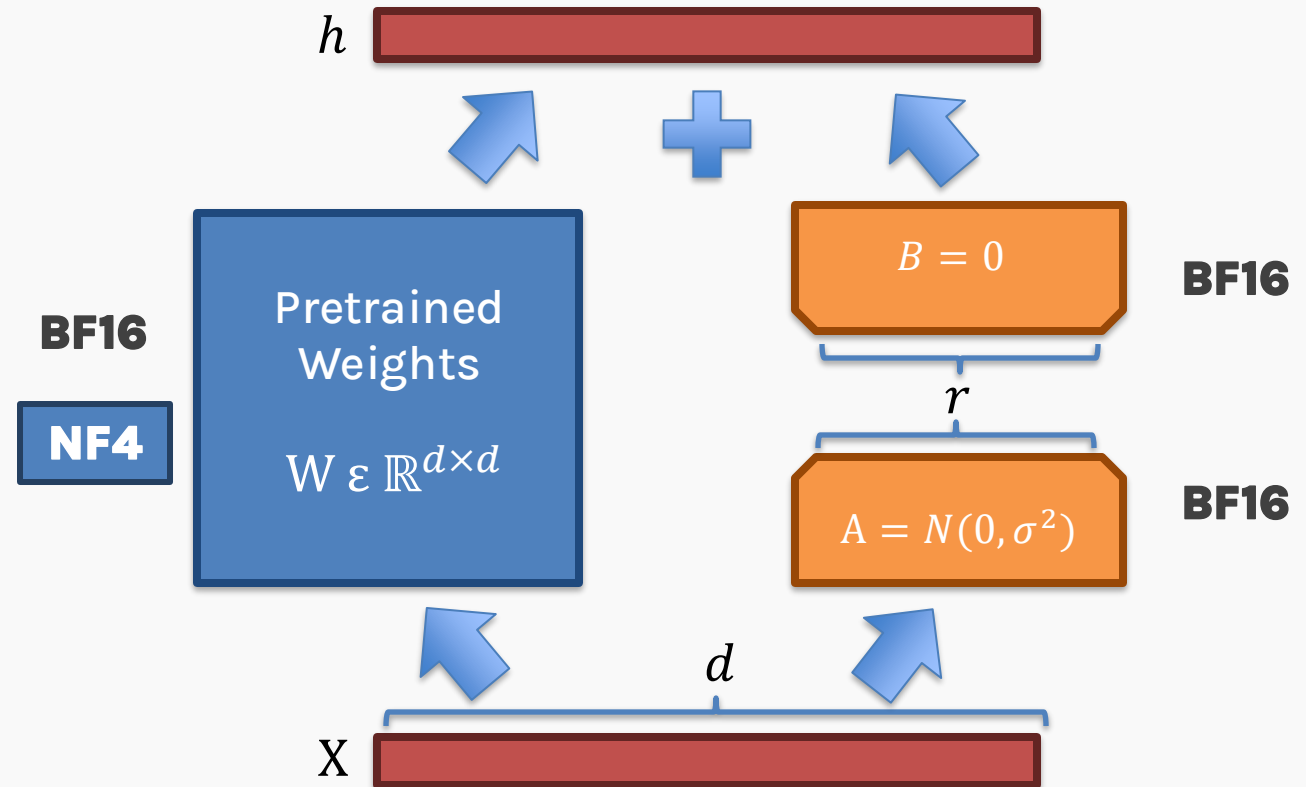We store $\mathbb{W}$ (original model parameters) as 4-Bit NormalFloat

## Forward Pass

During the forward pass, we first dequantize the W weights from NF4 to BF16 for computation.

We then use the BF16 values of W, A and B to perform the required calculations.

The BF16 values of W is then deleted to save on storage!



$h$

$B = 0$    **BF16**

**BF16**

**NF4**

Pretrained Weights

$\mathrm{W} \, \varepsilon \, \mathbb{R}^{d \times d}$
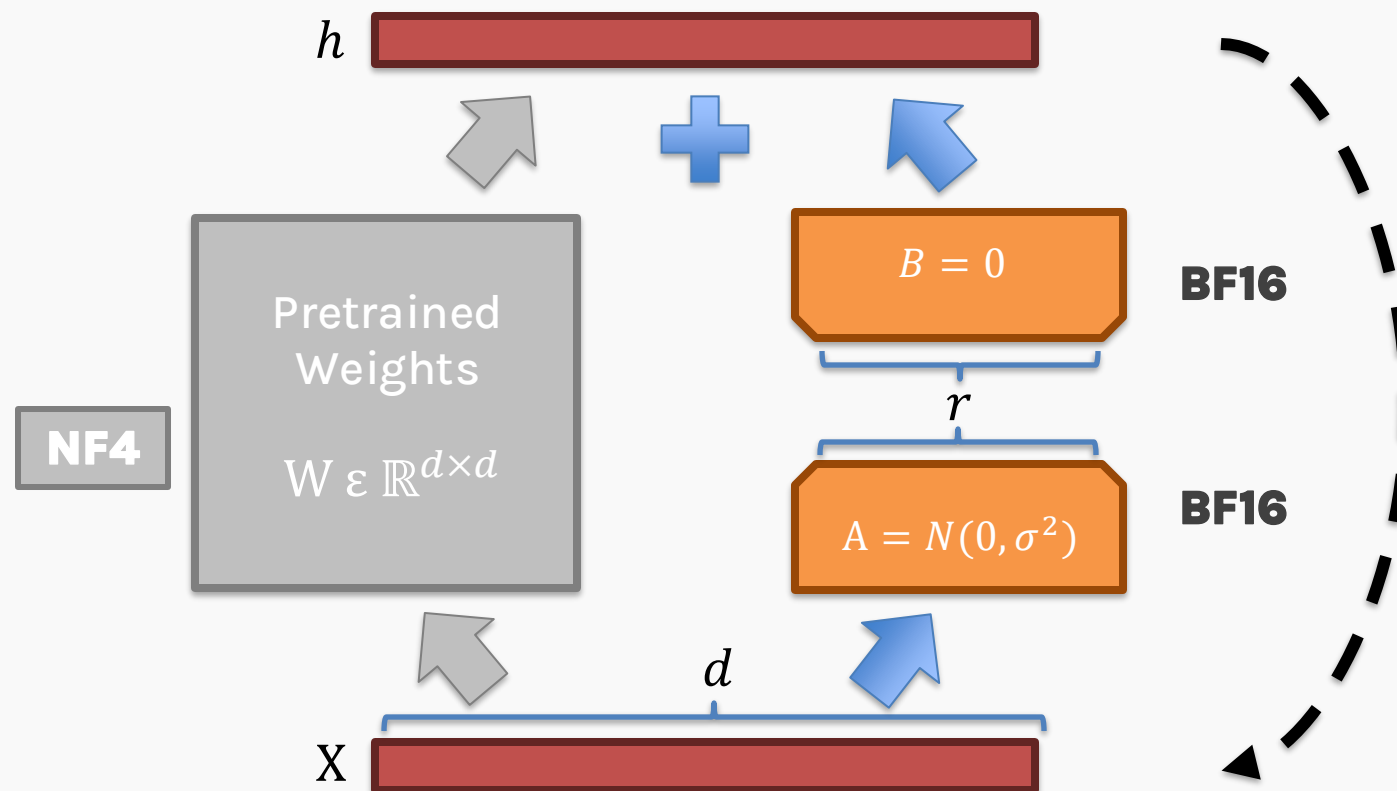
$r$

$A = N(0, \sigma^2)$    **BF16**

$d$

X

# QLoRA – Putting it all together

## Backward Pass

As in LoRA, we keep W weights frozen and allow the gradients to only flow through the adapters.

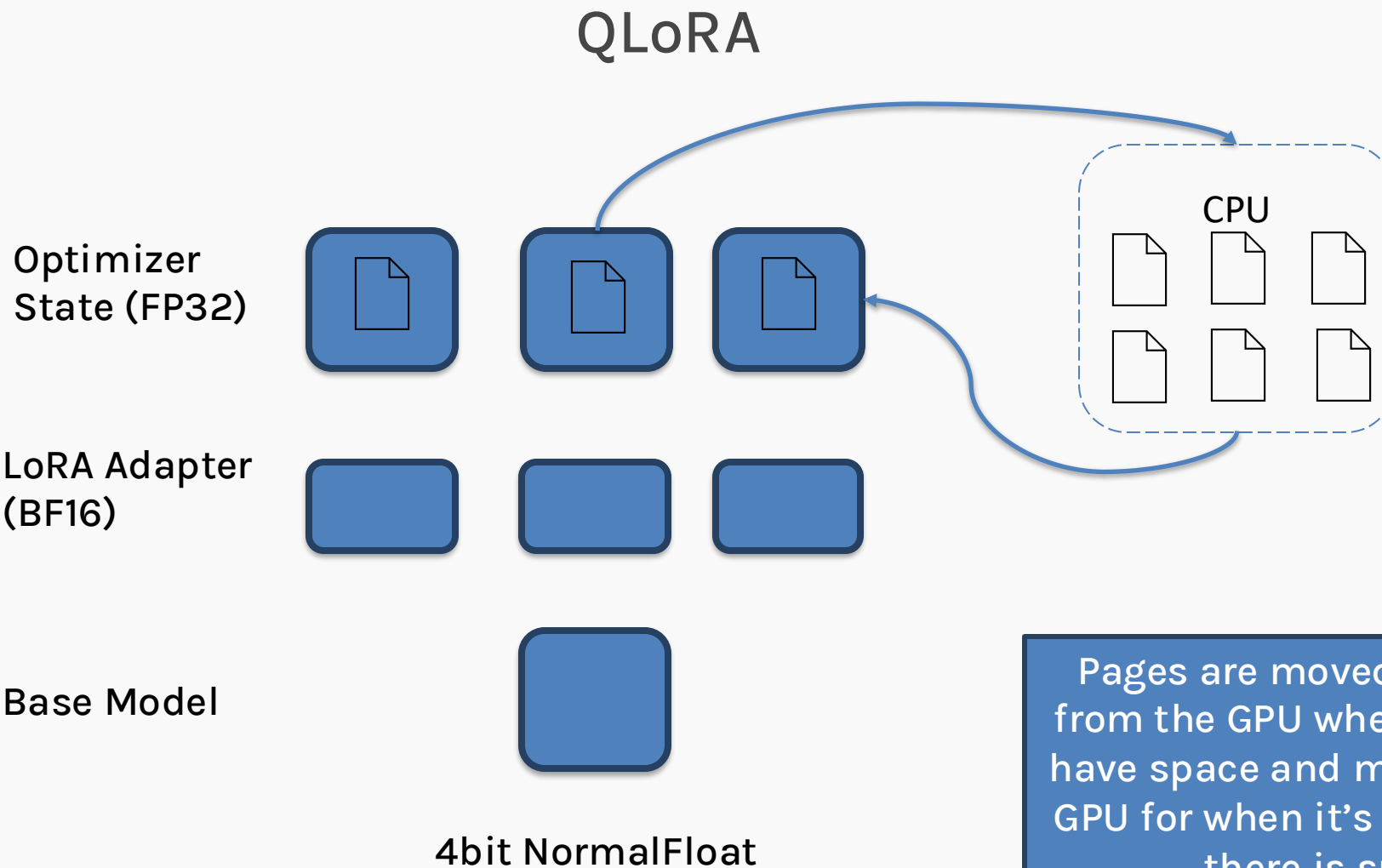We then repeat the cycle of forward and backward passes till a minima is reached.



NF4

Pretrained Weights

$W \in \mathbb{R}^{d \times d}$

$B = 0$

BF16

$r$

$A = N(0, \sigma^2)$

BF16

$h$

$d$

X

# QLoRA – Putting it all together

**Ingredient 3:**

Paged
Optimizer

## QLoRA

Optimizer
State (FP32)

CPU

LoRA Adapter
(BF16)

Base Model

4bit NormalFloat

10B => ~12GB

Pages are moved to the CPU from the GPU when it does not have space and moved back to GPU for when it's required and there is space.

Putting it mathematically,

Let's start with LoRA:

Initial LLM Weights

Scaling parameter

Decomposed matrices

**Weights** of LoRA: $W_0 + \dfrac{\alpha}{r} BA$

Rank of B and A

**Forward pass** in LoRA:  $Y = XW_0 + \dfrac{\alpha}{r} XBA$

$$Y = XW_0 + \frac{\alpha}{r} XBA$$

Let's expand the formula and see how it looks!

$$Y^{BF16} = X^{BF16} doubleDequant\left(c_1^{FP32}, c_2^{k-bit}, W_o^{NF4}\right) + \frac{\alpha}{r} X^{BF16} B^{BF16} A^{BF16}$$

where $doubleDequant\left(c_1^{FP32}, c_2^{k-bit}, W_o^{NF4}\right) = dequant\left(dequant\left(c_1^{FP32}, c_2^{k-bit}\right), W_o^{4bit}\right)$
$$= W^{BF16}$$

# After completing Finetuning:



# Thank you!