AI VIETNAM
All-in-One Course

# NLP Course

# Quantization – QLoRA
# NeurIPS LLM Efficiency Challenge

Nguyen Quoc Thai

Year 2023

# CONTENT

# 1 – Quantization

## Floating Point Number

➢ A floating point number is a positive or negative whole number with a decimal point

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ***man*** → | 0.6 | −0.2 | 0.8 | 0.9 | −0.1 | −0.9 | −0.7 |
| ***woman*** → | 0.7 | 0.3 | 0.9 | −0.7 | 0.1 | −0.5 | −0.4 |
| ***king*** → | 0.5 | −0.4 | 0.7 | 0.8 | 0.9 | −0.7 | −0.6 |
| ***queen*** → | 0.8 | −0.1 | 0.8 | −0.9 | 0.8 | −0.5 | −0.9 |

# 1 – Quantization

## Tensor

- Tensor: multidimensional array

```
import torch
data = torch.rand(3,3)
data
```

```
tensor([[0.5123, 0.6089, 0.1713],
        [0.2419, 0.8776, 0.8224],
        [0.8413, 0.7830, 0.7792]])
```

# 1 – Quantization

## Tensor Properties

- Shape
- Device: CPU (-1), GPU (Cuda:0,…)
- Data Type

```
data.dtype
```

```
torch.float32
```

```
data.shape
```

```
torch.Size([3, 3])
```

```
data.get_device()
```

```
-1
```

| Data type | dtype | CPU tensor | GPU tensor |
|---|---|---|---|
| 32-bit floating point | `torch.float32` or `torch.float` | `torch.FloatTensor` | `torch.cuda.FloatTensor` |
| 64-bit floating point | `torch.float64` or `torch.double` | `torch.DoubleTensor` | `torch.cuda.DoubleTensor` |
| 16-bit floating point | `torch.float16` or `torch.half` | `torch.HalfTensor` | `torch.cuda.HalfTensor` |
| 8-bit integer (unsigned) | `torch.uint8` | `torch.ByteTensor` | `torch.cuda.ByteTensor` |
| 8-bit integer (signed) | `torch.int8` | `torch.CharTensor` | `torch.cuda.CharTensor` |
| 16-bit integer (signed) | `torch.int16` or `torch.short` | `torch.ShortTensor` | `torch.cuda.ShortTensor` |
| 32-bit integer (signed) | `torch.int32` or `torch.int` | `torch.IntTensor` | `torch.cuda.IntTensor` |
| 64-bit integer (signed) | `torch.int64` or `torch.long` | `torch.LongTensor` | `torch.cuda.LongTensor` |
| Boolean | `torch.bool` | `torch.BoolTensor` | `torch.cuda.BoolTensor` |

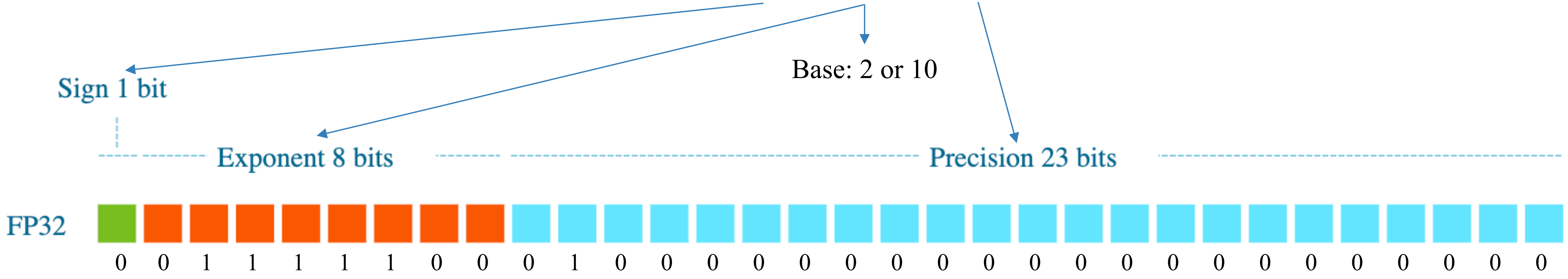
# 1 – Quantization

## FP32: Single Precision Floating Point

- 1 bit sign
- 8 bits exponent
- 23 bits fraction (precision)

- FP32: default => Weights, activations and other values in Neural Networks

$$+0.15625 = (+1)\times 2^{-3}\times 1.25$$



# 1 – Quantization

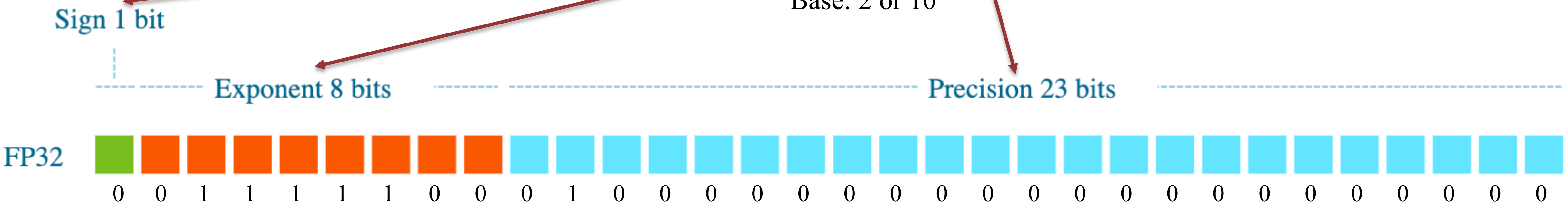
## FP32: Single Precision Floating Point

- Backward Propagation
- FP32: default => Weights, activations and other values in Neural Networks

Use a data type with fewer bits to represent floating point

$$+0.15625 = (+1)\times 2^{-3}\times 1.25$$

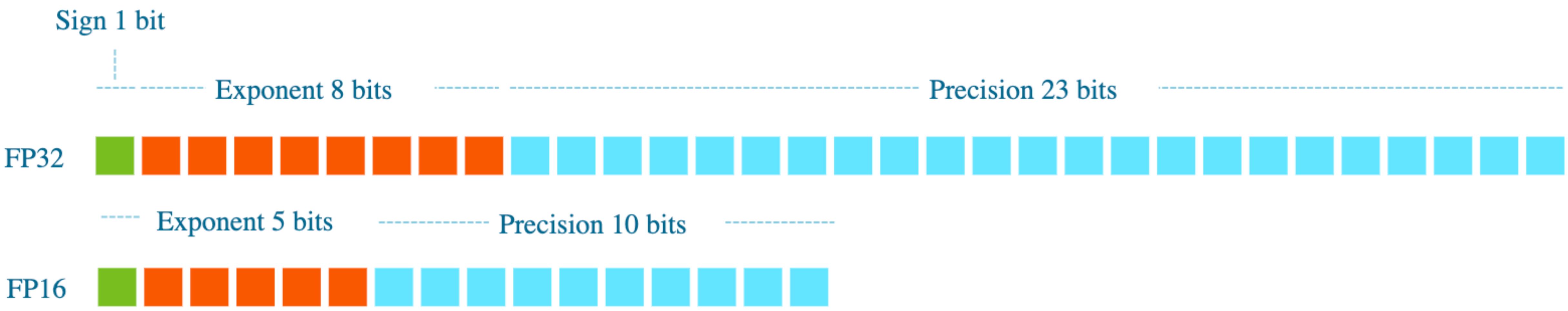
Base: 2 or 10



# 1 – Quantization

## FP16: Half Precision Floating Point

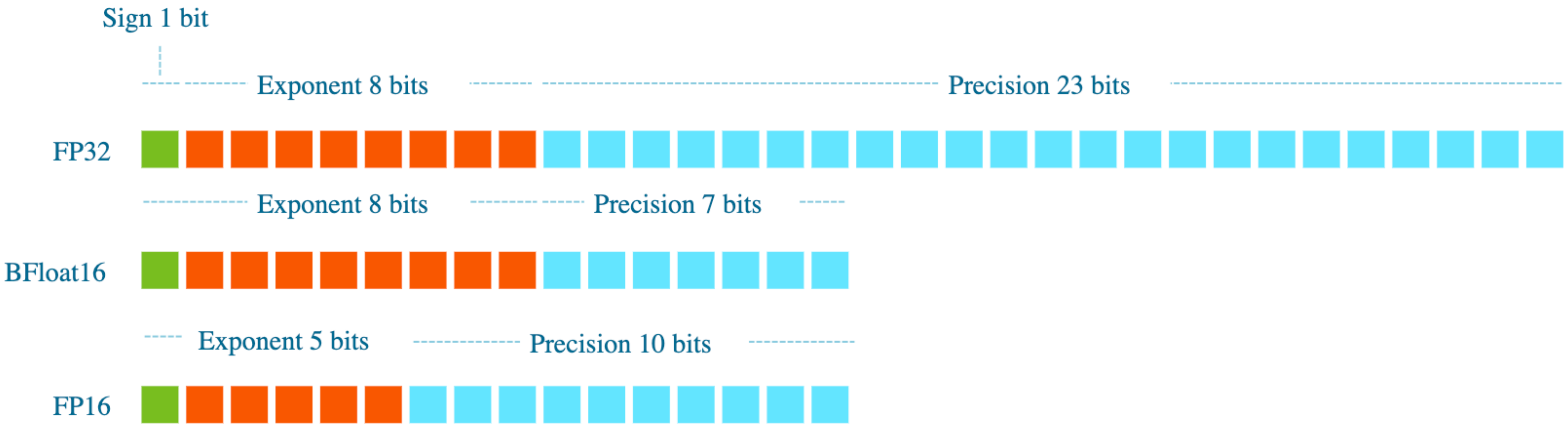
- 1 bit sign
- 5 bits exponent
- 10 bits fraction (precision)



# 1 – Quantization

## BFLOAT16: Brain Floating Point

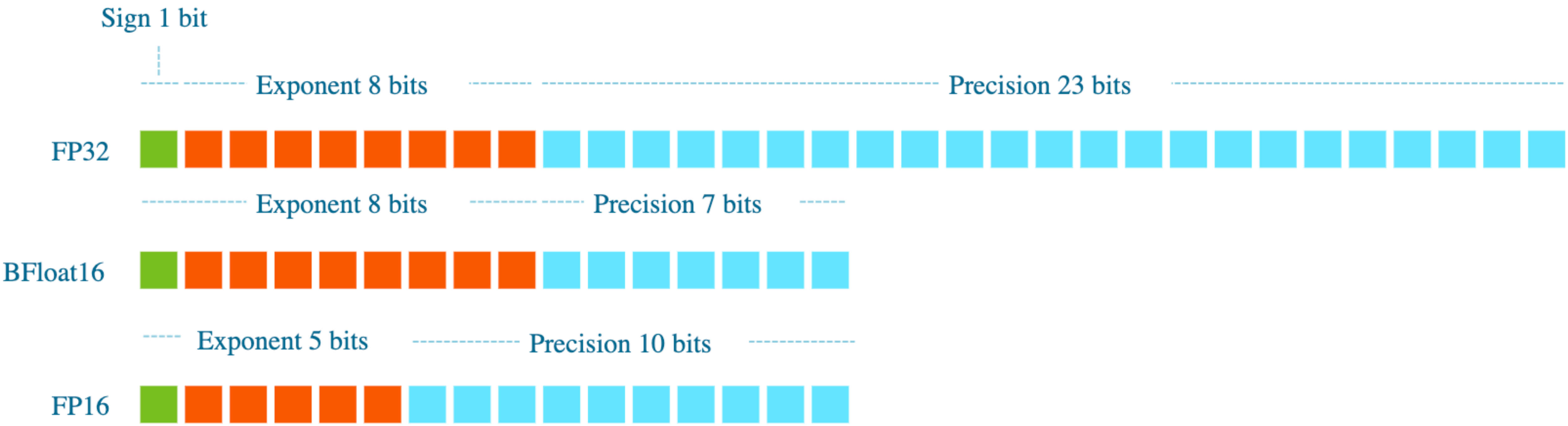
- 1 bit sign
- 8 bits exponent
- 7 bits fraction (precision)



# 1 – Quantization

## Quantization

- Quantization: mapping input values from a large set (often a continuous set) to outputs values in a (countable) smaller set.
- Ex: Rounding and truncation



# 1 – Quantization

## Quantization

- Quantize from source dtype FP32 to target dtype INT8
- INT8: [-127, 127]

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

c: constant

FP32 | 0.1 | 0.2 | 0.4 | → $c = \frac{127}{0.4} = 317.5$ → | 32 | 64 | 127 | INT8

# 1 – Quantization

## Quantization

- Dequantize from target dtype INT8 to source dtype FP32
- INT8: [-127, 127]

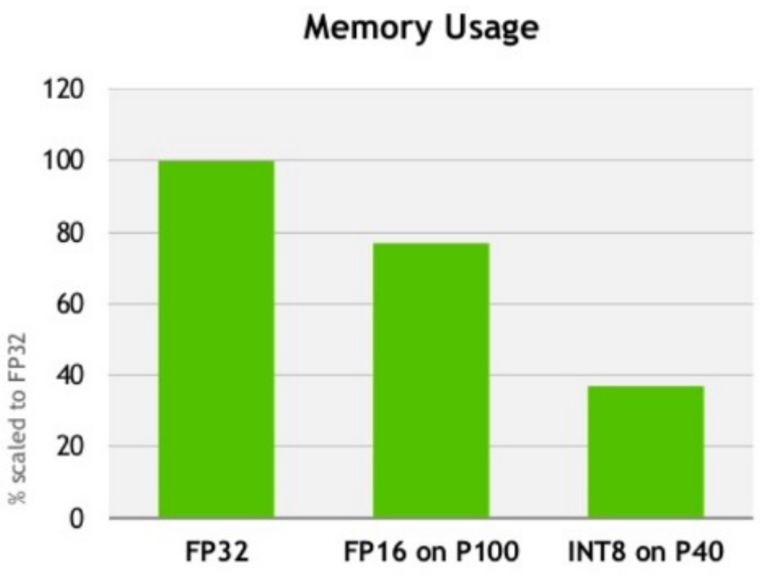
$$\text{dequant}\left(c^{\text{FP32}} X^{\text{FP32}}\right) = \frac{X^{\text{Int8}}}{c^{\text{FP32}}} = X^{\text{FP32}}$$

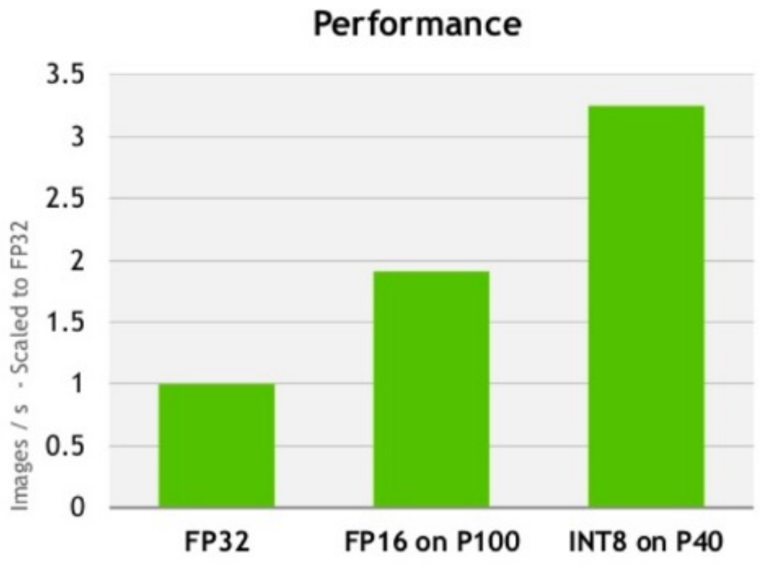
$$C = \frac{127}{0.4} = 317.5$$

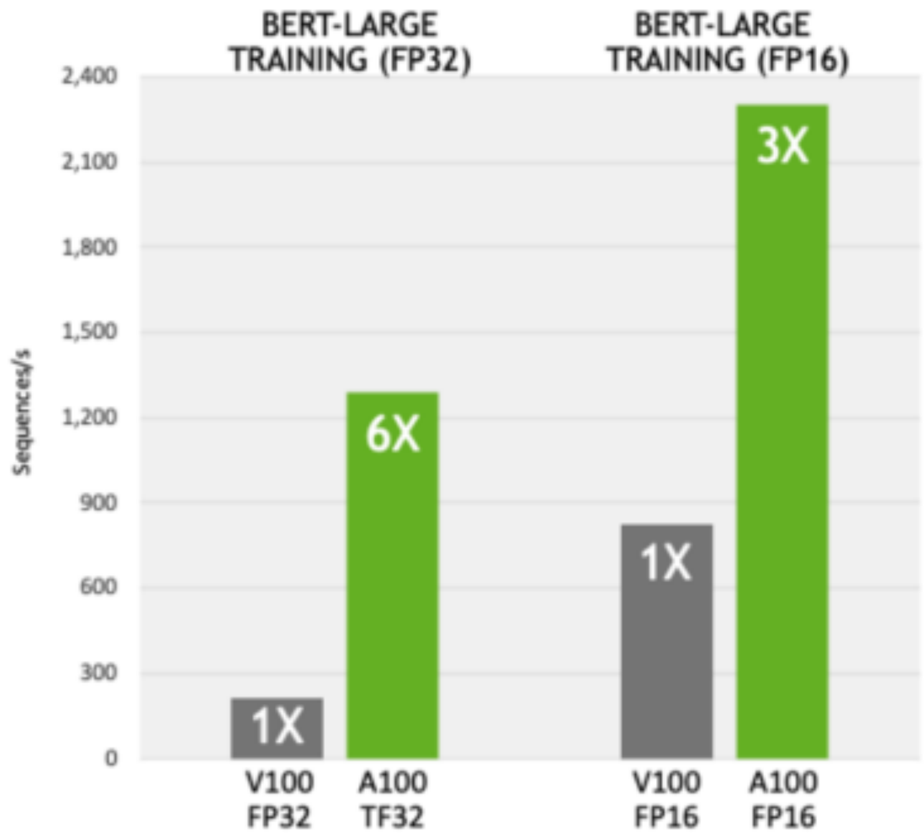
| | | | | |
|---|---|---|---|---|
| INT8 | 32 | 64 | 127 | |
| | 0.1 | 0.2 | 0.4 | INT8 |

# 1 – Quantization

## Smaller and Faster



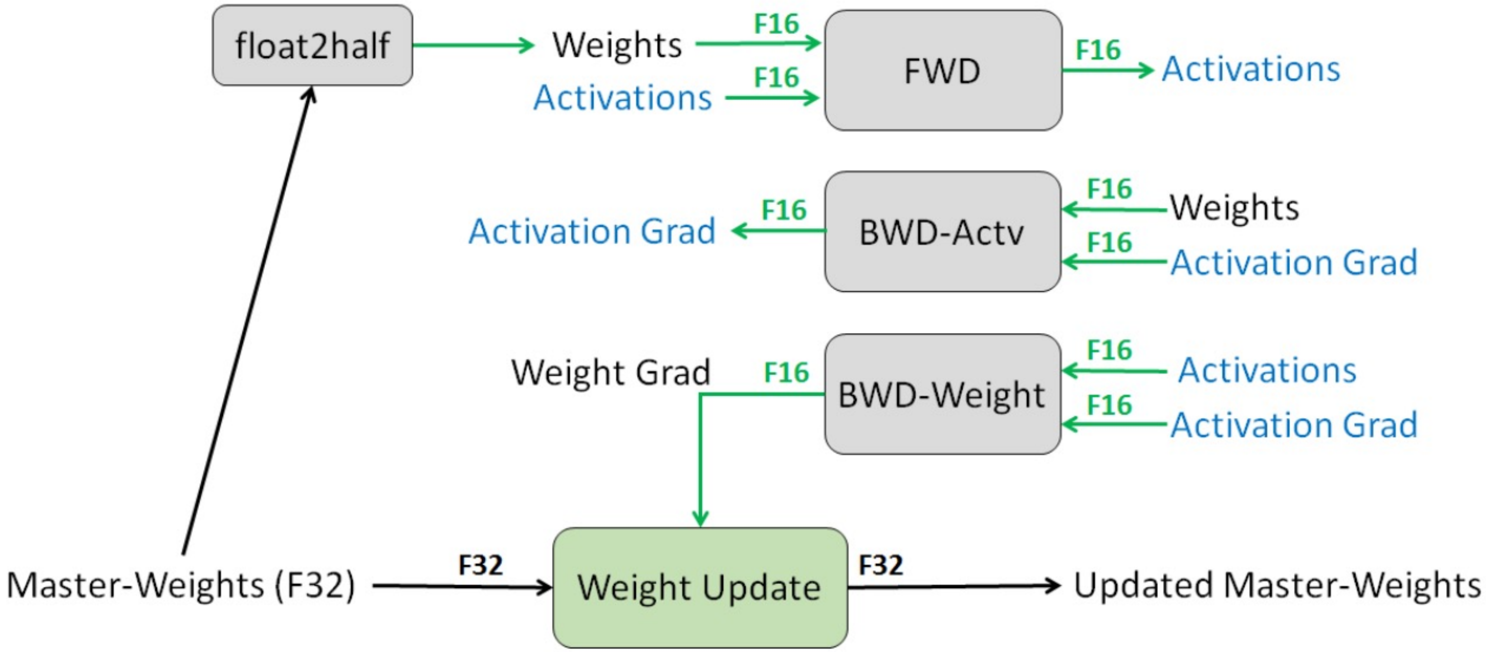



# 1 – Quantization

## Mixed Precision Training

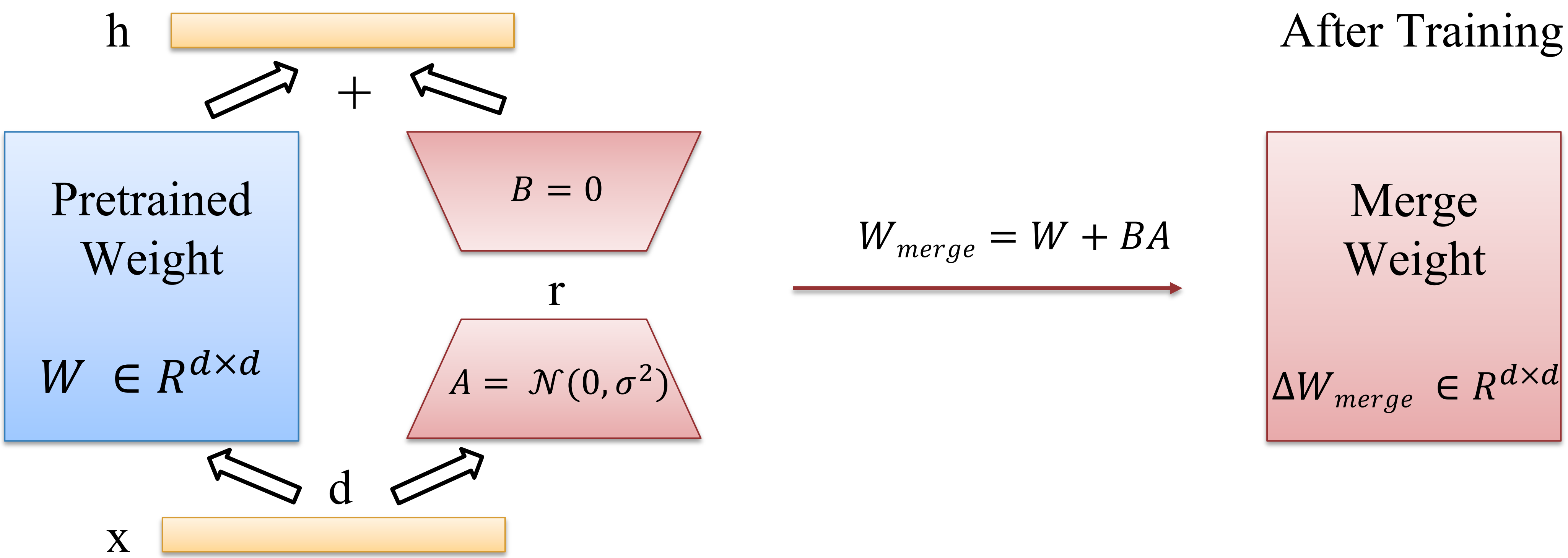
- Mixed Precision Training: Not a floating point data type but a method
- Use a combination of FP16 and FP32 to reduce the memory and math bandwidth



# 2 – QLoRA

## LoRA: Low-Rank Adaptation

- Freezes the pretrained model weights and injects trainable rank decomposition matrices into each layer of the Transformer architecture



# 2 – QLoRA

## LoRA: Low-Rank Adaptation

➤ LoRA can even outperform full finetuning training only 2% of the parameters

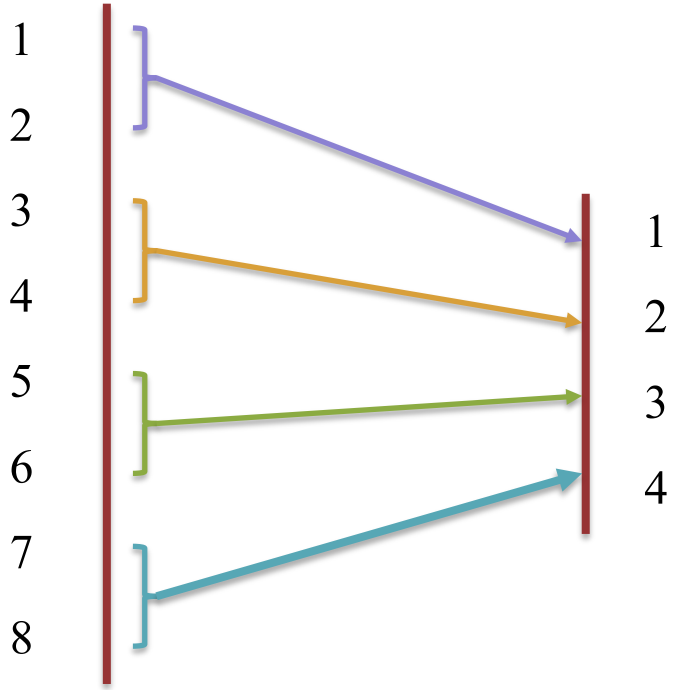
Full finetuning
Only tune bias vectors
Prompt tuning
Prefix tuning
ROUGE scores

| Model&Method | # Trainable Parameters | WikiSQL Acc. (%) | MNLI-m Acc. (%) | SAMSum R1/R2/RL |
|---|---|---|---|---|
| GPT-3 (FT) | 175,255.8M | **73.8** | 89.5 | 52.0/28.0/44.5 |
| GPT-3 (BitFit) | 14.2M | 71.3 | 91.0 | 51.3/27.4/43.5 |
| GPT-3 (PreEmbed) | 3.2M | 63.1 | 88.6 | 48.3/24.2/40.5 |
| GPT-3 (PreLayer) | 20.2M | 70.1 | 89.5 | 50.8/27.3/43.5 |
| GPT-3 ($\text{Adapter}^H$) | 7.1M | 71.9 | 89.8 | 53.0/28.9/44.8 |
| GPT-3 ($\text{Adapter}^H$) | 40.1M | 73.2 | **91.5** | 53.2/29.0/45.1 |
| GPT-3 (LoRA) | 4.7M | 73.4 | **91.7** | **53.8/29.8/45.9** |
| GPT-3 (LoRA) | 37.7M | **74.0** | **91.6** | 53.4/29.2/45.1 |

# 2 – QLoRA

## Challenges of Quantization Method

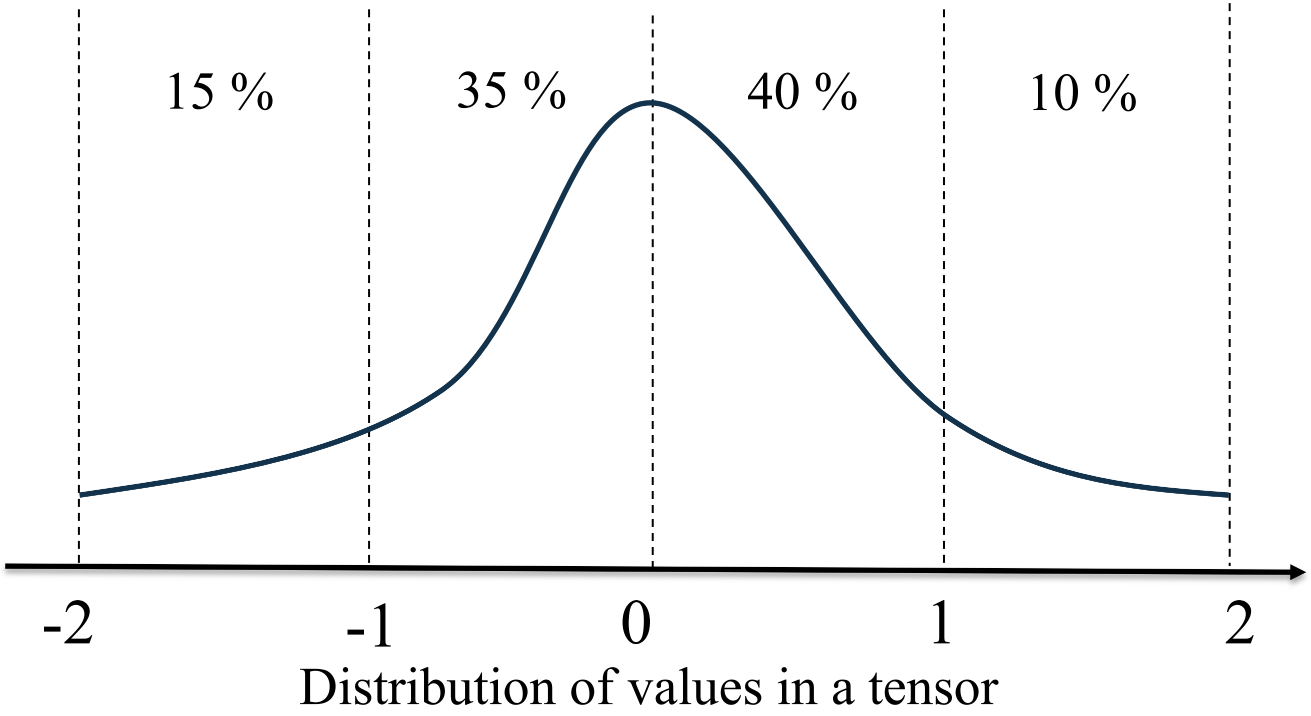
- Information Loss
- Example: quantize from INT3 to INT2




# 2 – QLoRA

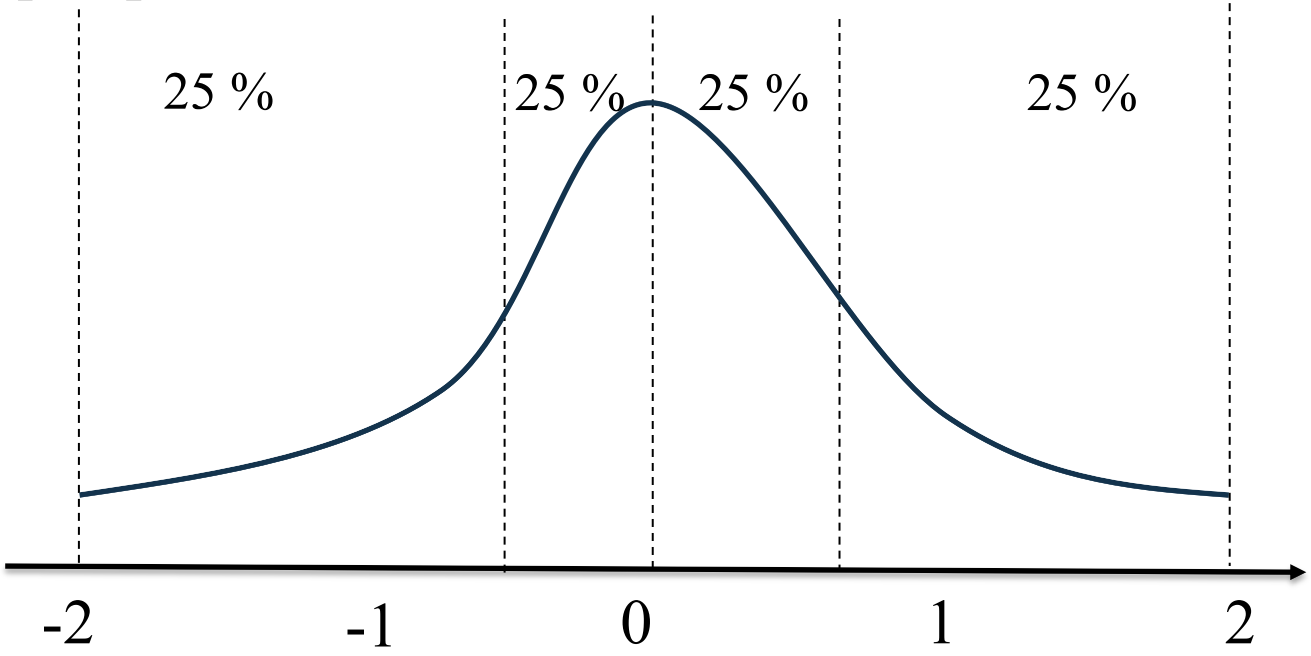
## Challenges of Quantization Method

- Linear Quantization (Ignore the distribution on the source data type)
- Example: $$X^{\text{Int8}} = \text{round}\left(\frac{127}{absmax(X^{\text{FP32}})} X^{\text{FP32}}\right) = \text{round}\left(c^{\text{FP32}} X^{\text{FP32}}\right)$$



## Challenges of Quantization Method

- Quantile Quantization
- Quantiles: cut points dividing the range of a probability distribution into continuous intervals with equal probabilities

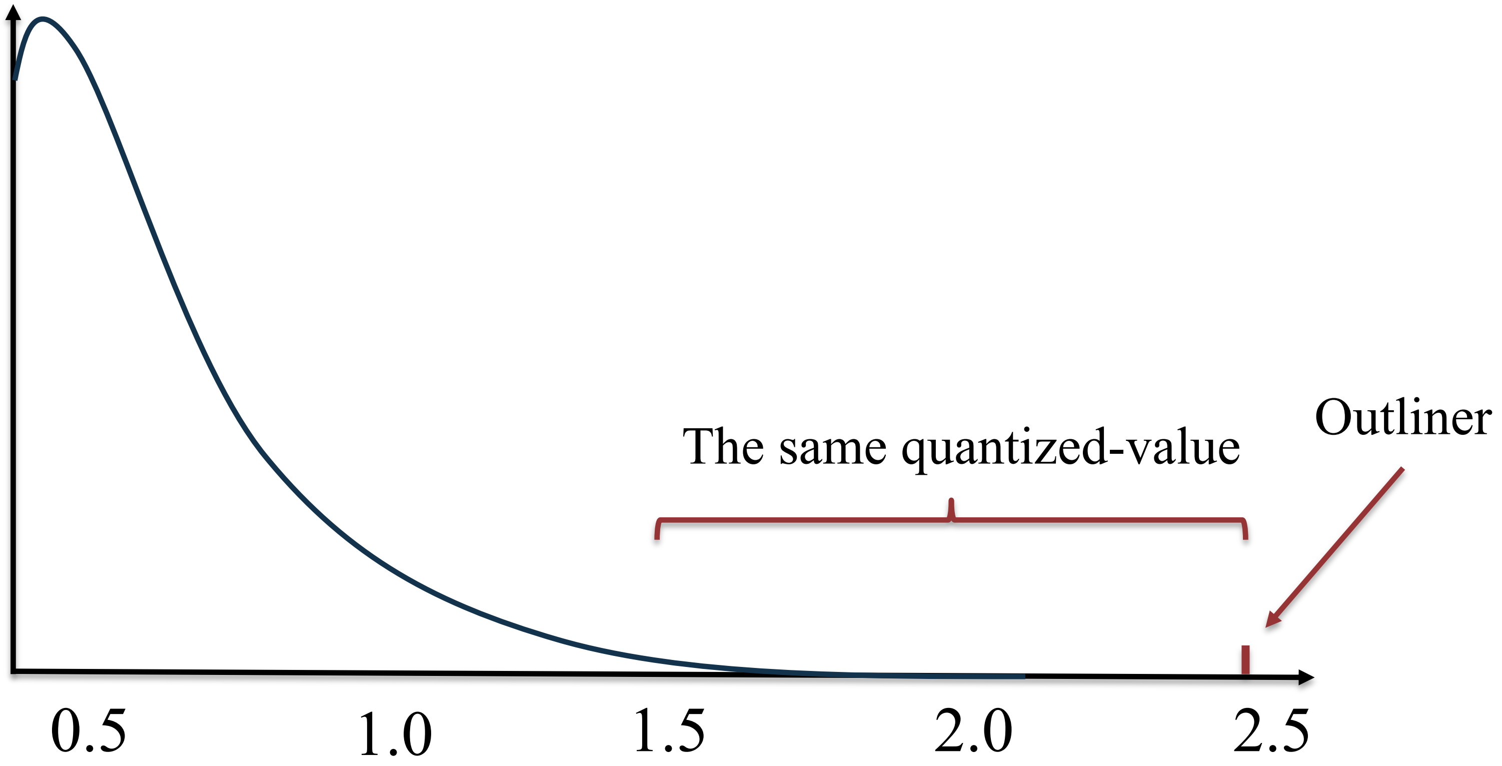



Distribution of values in a tensor

# 2 – QLoRA

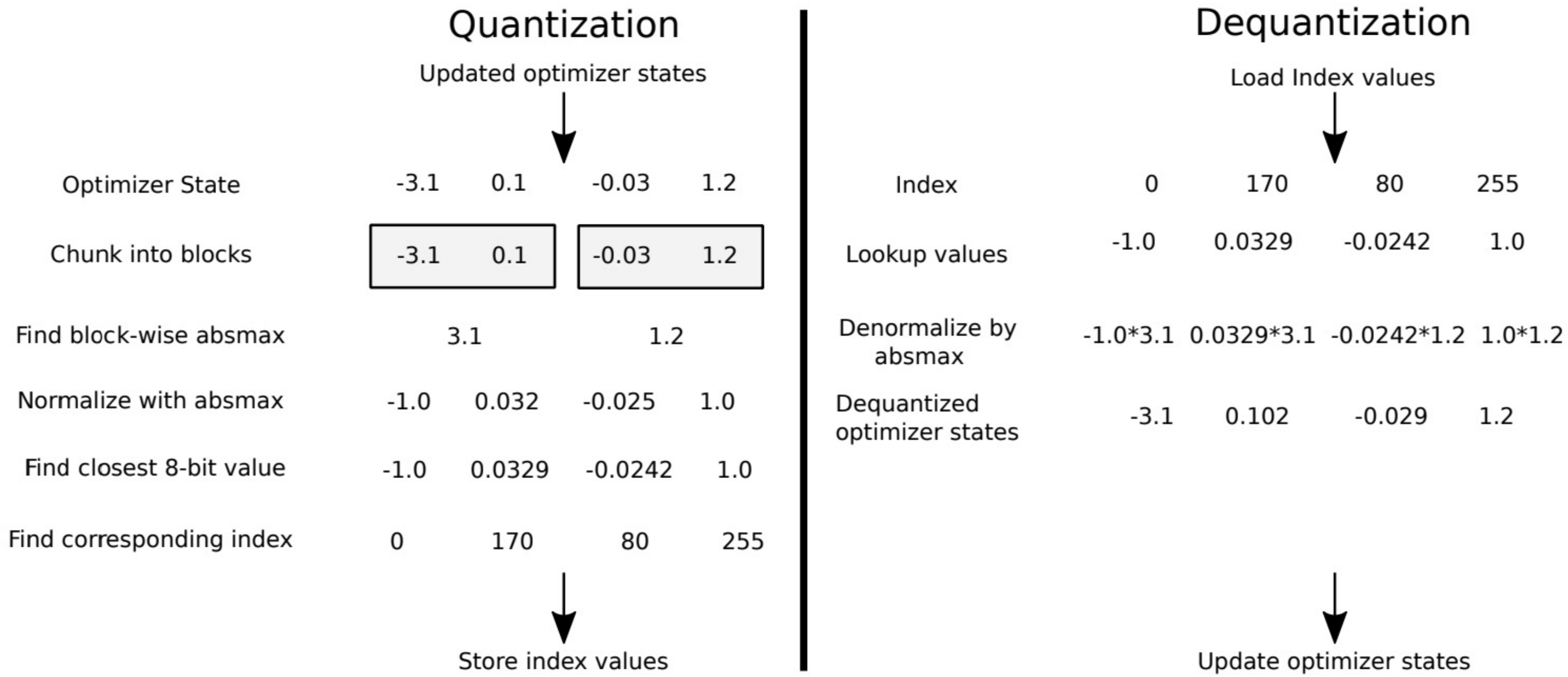
## Challenges of Quantization Method

- Outliner in Quantization: appear few times but is far away from other values
- Outliner often very important (Attention score)



## Challenges of Quantization Method

- Block-wise Quantization: split a tensor into many chunks, quantize individual chunks
- Block size: number of elements in a chunk



# 2 – QLoRA

! QLoRA: Efficient Finetuning of Quantized LLMs

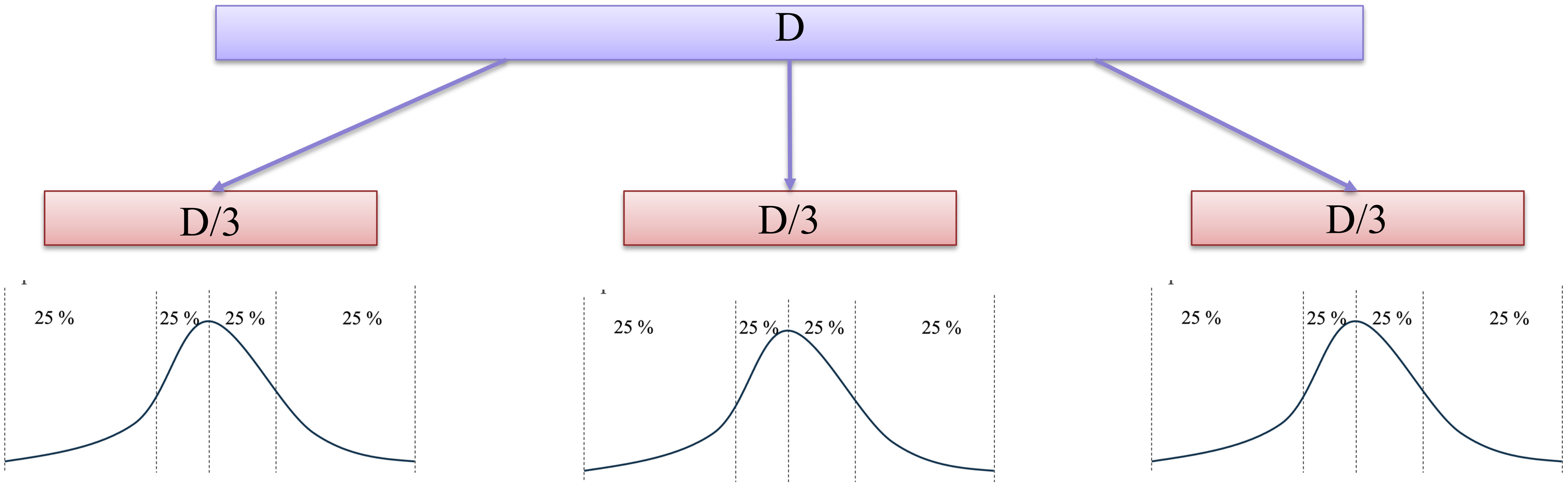
**QLoRA: save memory without sacrificing performance**

- 4-bit NormalFloat (NF4) via Block-wise Quantization
- Double Quantization
- Paged Optimizers
- Combined with LoRA

# 2 – QLoRA

## 4-bit NormalFloat (NF4)

➢ Step 1: Find quantiles in each chunks

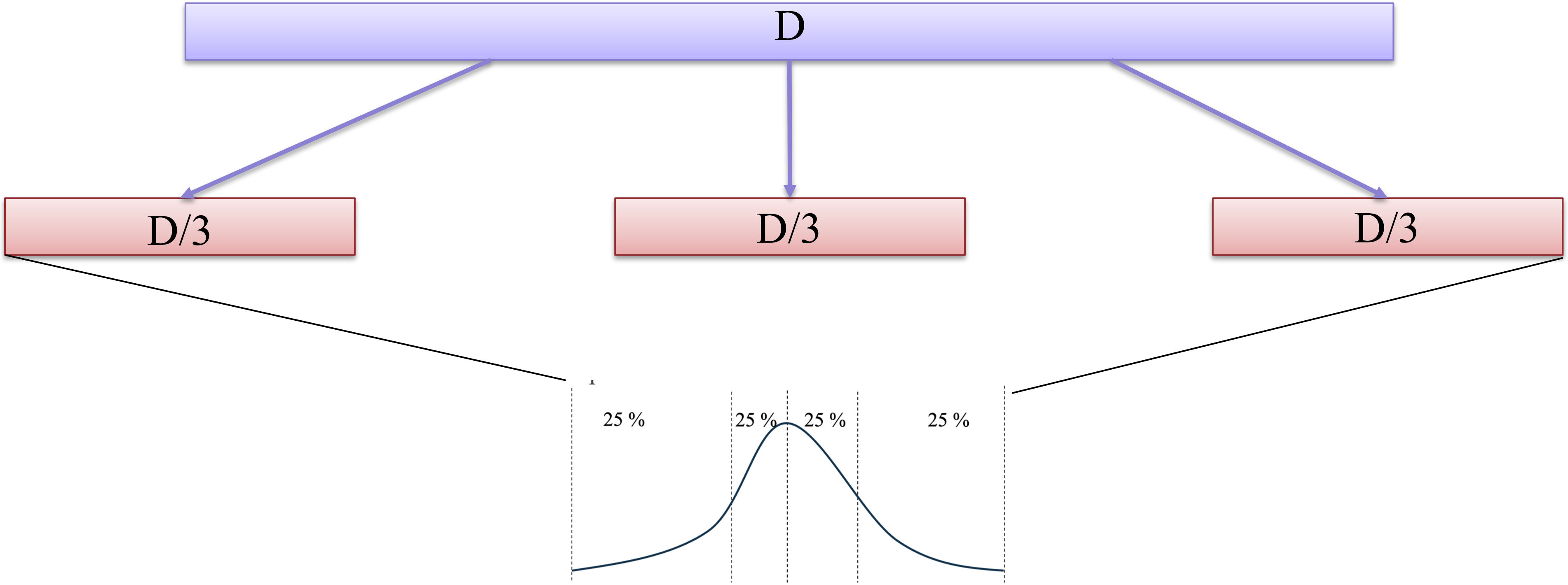



=> The main limitation of quantile quantization: process of quantile estimation very expensive

# 2 – QLoRA

## 4-bit NormalFloat (NF4)

- Step 1: Find quantiles in each chunks
- Use fixed distribution: zero-mean normal distribution with standard deviation σ



## 4-bit NormalFloat (NF4)

- Step 1: estimate the $2^k + 1$ quantiles of a theoretical N(0, 1) distribution to obtain a k-bit quantile quantization data type for normal distributions as follows:

$$q_i = \frac{1}{2}\left(Q_X\left(\frac{i}{2^k+1}\right) + Q_X\left(\frac{i+1}{2^k+1}\right)\right)$$

QX: the quantile function of the standard normal distribution N(0,1)

# 2 – QLoRA

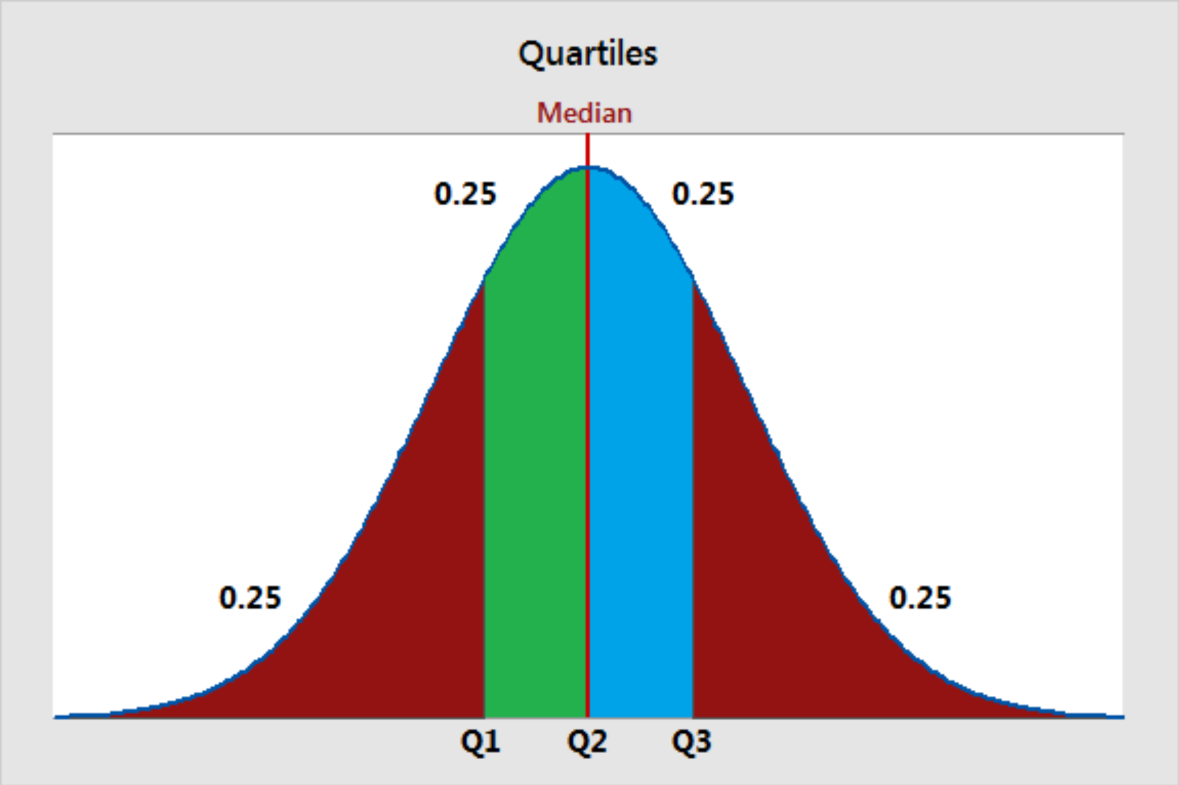
## 4-bit NormalFloat (NF4)

- Step 1: Estimate the $2^k + 1$ quantiles of a theoretical N(0, 1) distribution to obtain a k-bit quantile quantization data type for normal distributions
- Step 2: Take this data type and normalize its values into the $[-1, 1]$ range
- Step 3: Quantize an input weight tensor by normalizing it into the $[-1, 1]$ range through absolute maximum rescaling

# 2 – QLoRA

## 4-bit NormalFloat (NF4)

- Problem: Quantiles not have an exact representation of zero (Symmetric)
- Important property to quantize padding and other zero-valued elements with no error



## 4-bit NormalFloat (NF4)

Solution: create an asymmetric data type by estimating quantiles qi of two range:

- $2^{k-1}$ for the negative part
- $2^{k-1} + 1$ for the positive part
- Then unify these sets of $q_i$ and remove one of the two zeros that occurs in both sets

=> **K-bit NormalFloat (NFk) data type**

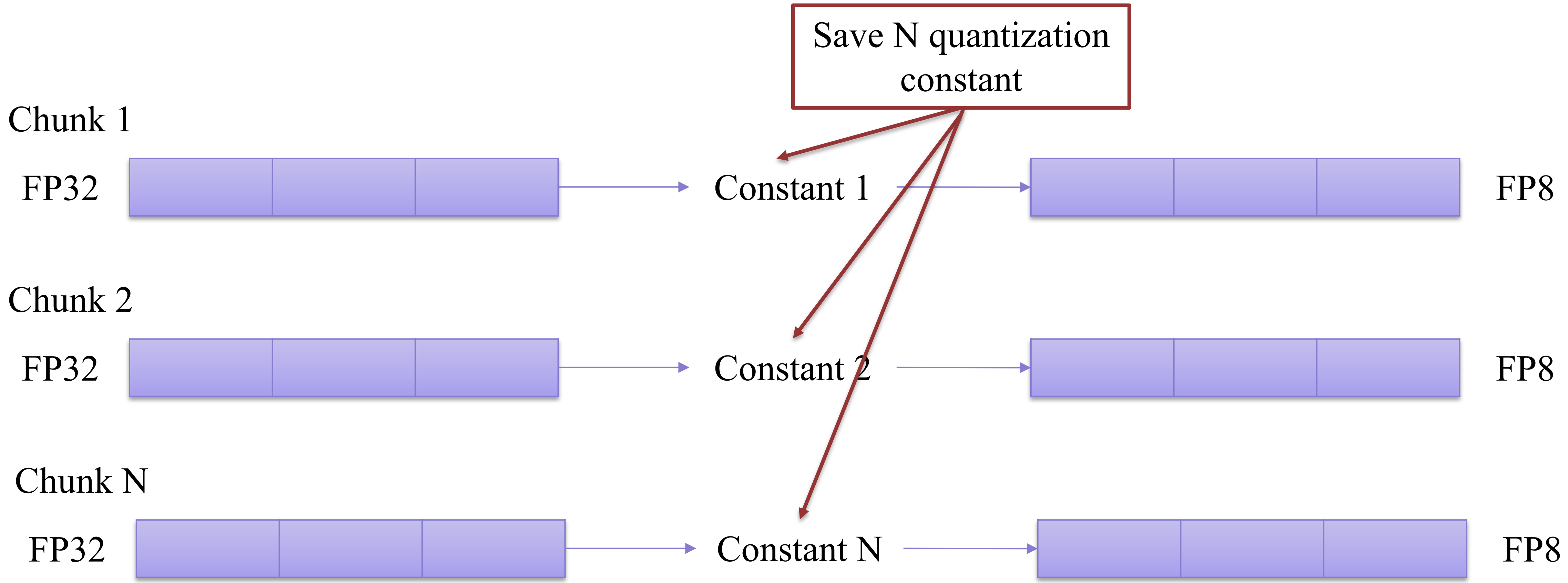
# 2 – QLoRA

## 4-bit NormalFloat (NF4)

- Use 4 bits to representation
- Normalize into [-1, 1] range
- An asymmetric data type: an exact representation of zero
- Quantiles based on zero-mean normal distribution with standard deviation σ

# 2 – QLoRA

## Double Quantization



## Double Quantization

- The process of quantizing the quantization constants for additional memory savings

Chunk

FP32 → Constant → FP8

$c_2^{FP32}$

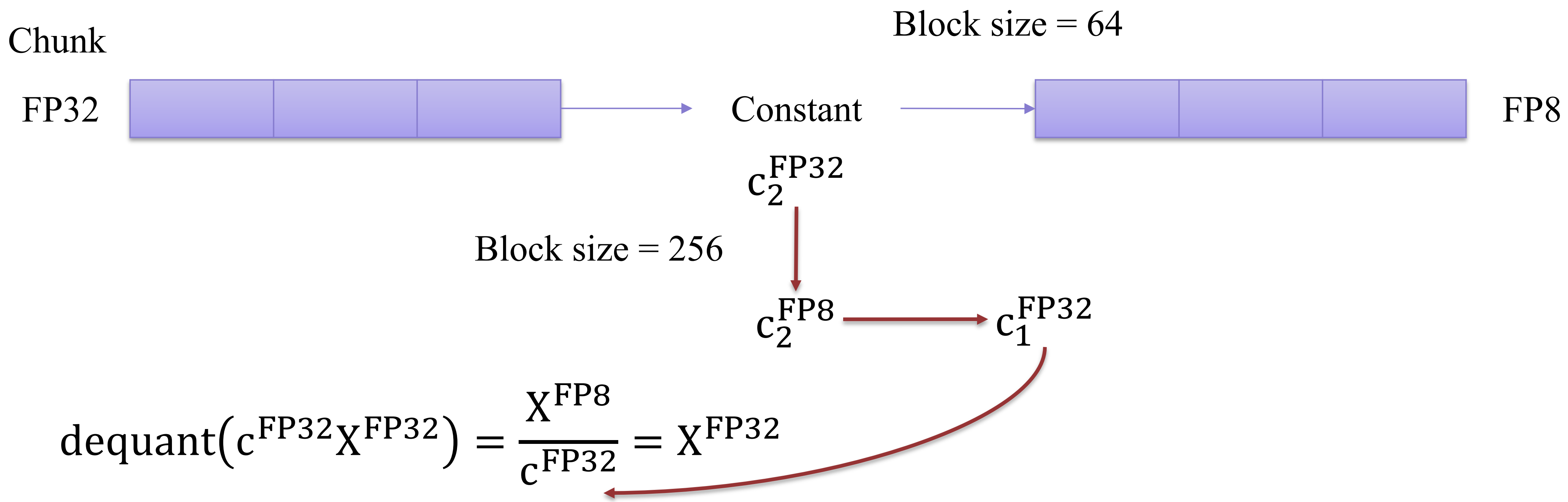
$$c_2^{FP32} \rightarrow c_2^{FP8} \rightarrow c_1^{FP32}$$

$$\mathrm{dequant}\left(c^{FP32} X^{FP32}\right) = \frac{X^{FP8}}{c^{FP32}} = X^{FP32}$$

# 2 – QLoRA

## Double Quantization

➤ The process of quantizing the quantization constants for additional memory savings




$$\text{dequant}\left(c^{FP32}X^{FP32}\right) = \frac{X^{FP8}}{c^{FP32}} = X^{FP32}$$

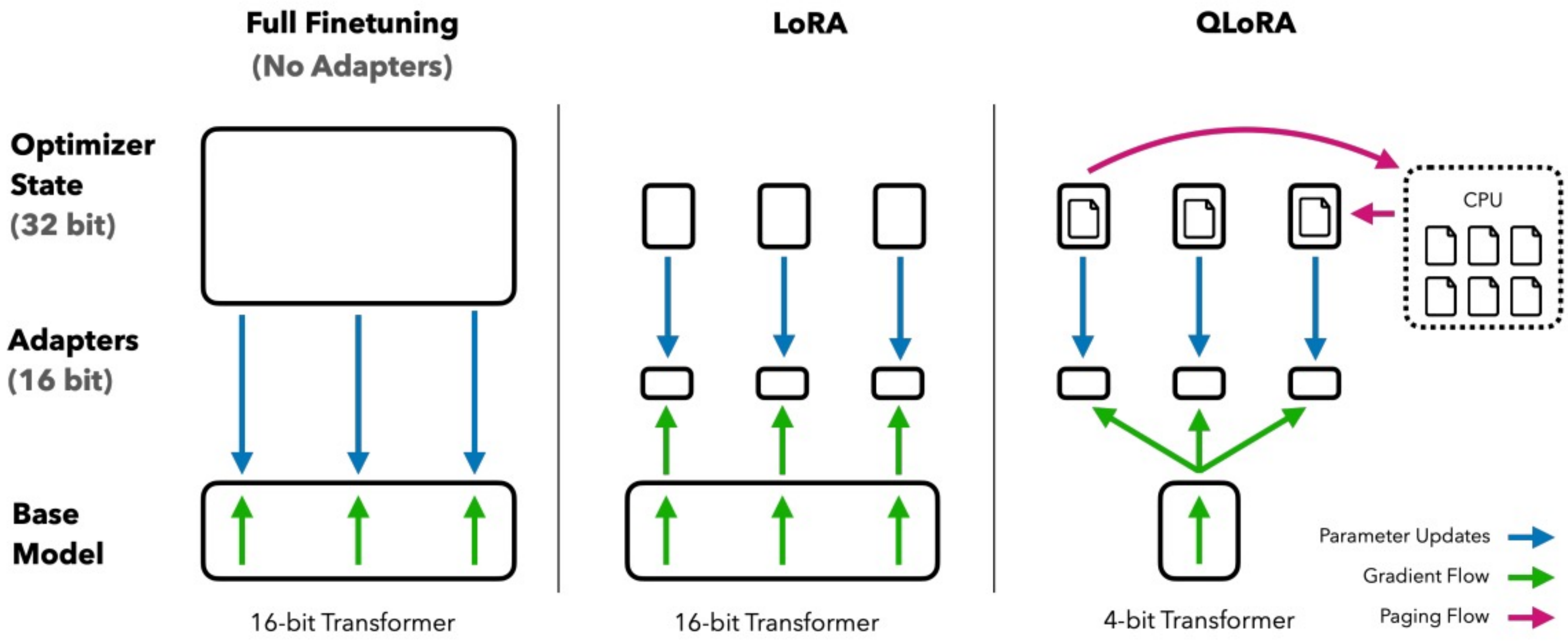
# 2 – QLoRA

## Paged Optimizers

- Page Optimizers to manage memory spikes
- Allocate paged memory for the optimizer states which are then automatically evicted to CPU RAM when the GPU runs out-of-memory and paged back into GPU memory when the memory is needed in the optimizer update step

# 2 – QLoRA

## QLoRA



# 2 – QLoRA

## QLoRA

- Given a projection $XW = Y, X \in \mathbb{R}^{b \times h}, W \in \mathbb{R}^{h \times o}$, LoRA computes:

$$Y = XW + sXL_1L_2$$

$L_1 \in \mathbb{R}^{h \times r}, L_2 \in \mathbb{R}^{r \times o}$, s is a scaler

# 2 – QLoRA

## QLoRA

- Quantized base model with a single LoRA adapter:

$$Y^{BF16} = X^{BF16}\text{doubleDequant}\left(c_1^{FP32}, c_2^{k-bit}, W^{NF4}\right) + X^{BF16}L_1^{BF16}L_2^{BF16}$$

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

NF4 for W and FP8 for c2

A block size of 64 for W for higher quantization precision

A block size of 256 for c2 to conserve memory

# 2 – QLoRA

## Result

| Model / Dataset | Params | Model bits | Memory | ChatGPT vs Sys | Sys vs ChatGPT | Mean | 95% CI |
|---|---|---|---|---|---|---|---|
| GPT-4 | - | - | - | 119.4% | 110.1% | **114.5%** | 2.6% |
| Bard | - | - | - | 93.2% | 96.4% | 94.8% | 4.1% |
| **Guanaco** | 65B | 4-bit | 41 GB | 96.7% | 101.9% | **99.3%** | 4.4% |
| Alpaca | 65B | 4-bit | 41 GB | 63.0% | 77.9% | 70.7% | 4.3% |
| FLAN v2 | 65B | 4-bit | 41 GB | 37.0% | 59.6% | 48.4% | 4.6% |
| **Guanaco** | 33B | 4-bit | 21 GB | 96.5% | 99.2% | **97.8%** | 4.4% |
| Open Assistant | 33B | 16-bit | 66 GB | 91.2% | 98.7% | 94.9% | 4.5% |
| Alpaca | 33B | 4-bit | 21 GB | 67.2% | 79.7% | 73.6% | 4.2% |
| FLAN v2 | 33B | 4-bit | 21 GB | 26.3% | 49.7% | 38.0% | 3.9% |
| Vicuna | 13B | 16-bit | 26 GB | 91.2% | 98.7% | **94.9%** | 4.5% |
| **Guanaco** | 13B | 4-bit | 10 GB | 87.3% | 93.4% | 90.4% | 5.2% |
| Alpaca | 13B | 4-bit | 10 GB | 63.8% | 76.7% | 69.4% | 4.2% |
| HH-RLHF | 13B | 4-bit | 10 GB | 55.5% | 69.1% | 62.5% | 4.7% |
| Unnatural Instr. | 13B | 4-bit | 10 GB | 50.6% | 69.8% | 60.5% | 4.2% |
| Chip2 | 13B | 4-bit | 10 GB | 49.2% | 69.3% | 59.5% | 4.7% |
| Longform | 13B | 4-bit | 10 GB | 44.9% | 62.0% | 53.6% | 5.2% |
| Self-Instruct | 13B | 4-bit | 10 GB | 38.0% | 60.5% | 49.1% | 4.6% |
| FLAN v2 | 13B | 4-bit | 10 GB | 32.4% | 61.2% | 47.0% | 3.6% |
| **Guanaco** | 7B | 4-bit | 5 GB | 84.1% | 89.8% | **87.0%** | 5.4% |
| Alpaca | 7B | 4-bit | 5 GB | 57.3% | 71.2% | 64.4% | 5.0% |
| FLAN v2 | 7B | 4-bit | 5 GB | 33.3% | 56.1% | 44.8% | 4.0% |

# 3 – LLM Efficiency Challenge

!

## Challenge

**NeurIPS Large Language Model Efficiency Challenge:**

**1 LLM + 1GPU + 1Day**

NeurIPS 2023 Challenge

# 3 – LLM Efficiency Challenge

## Challenge

➢ Approved Base Models

- Falcon
- LLaMA or Llama 2
- OpenLLaMA
- Red Pajama Base (not instruction tuned models)
- MPT
- OPT
- Bloom
- GPT Neo, J, NeoX, Pythia
- GPT2
- T5 (not Flan-T5)
- BART
- DeBERTa
- RoBERTa
- BERT
- ALBERT
- DistilBERT
- Electra
- UL2
- Cerebras (btlm, GPT)

➢ Approved Base Models

- Databricks-Dolly-15
- OpenAssistant Conversations Dataset (oasst1)
- The Flan Collection
- AllenAI Dolma
- RedPajama-Data-1T
- LIMA

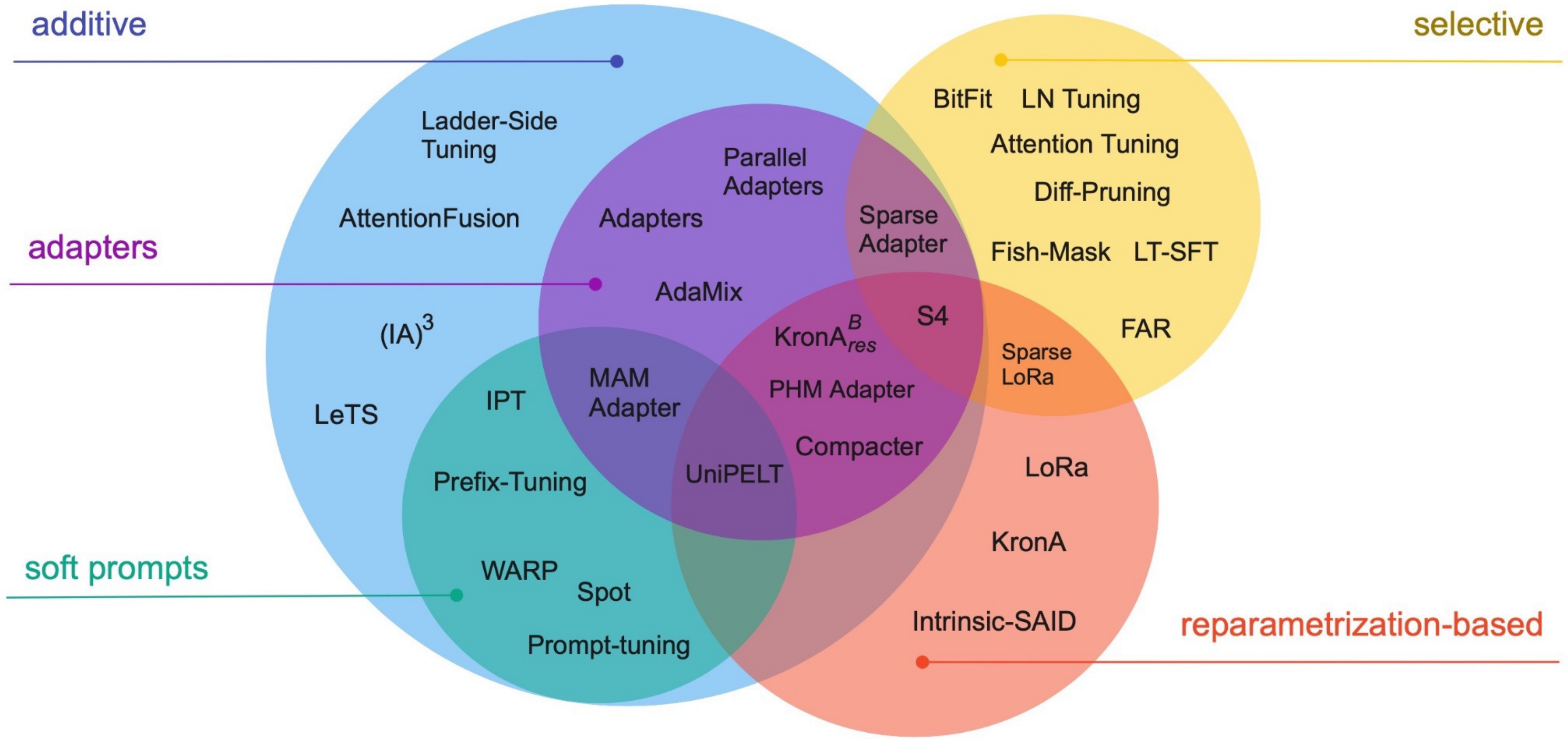
# 3 – LLM Efficiency Challenge

## Methods and tools for efficient training on a single GPU

| Method/tool | Improves training speed | Optimizes memory utilization |
|---|---|---|
| Batch size choice | Yes | Yes |
| Gradient accumulation | No | Yes |
| Gradient checkpointing | No | Yes |
| Mixed precision training | Yes | (No) |
| Optimizer choice | Yes | Yes |
| Data preloading | Yes | No |
| DeepSpeed Zero | No | Yes |
| torch.compile | Yes | No |

# 3 – LLM Efficiency Challenge

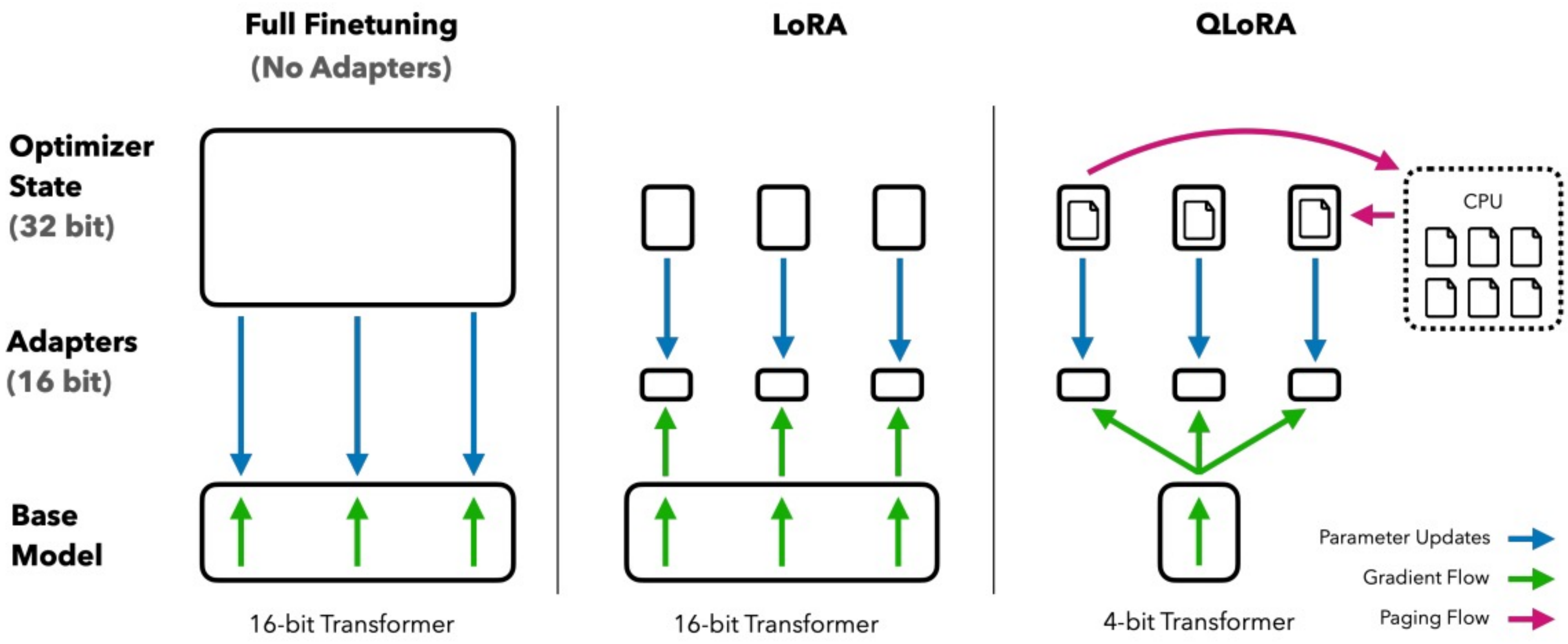
## Parameter-Efficient Fine-Tuning



AI VIET NAM
@aivietnam.edu.vn

# 3 – LLM Efficiency Challenge

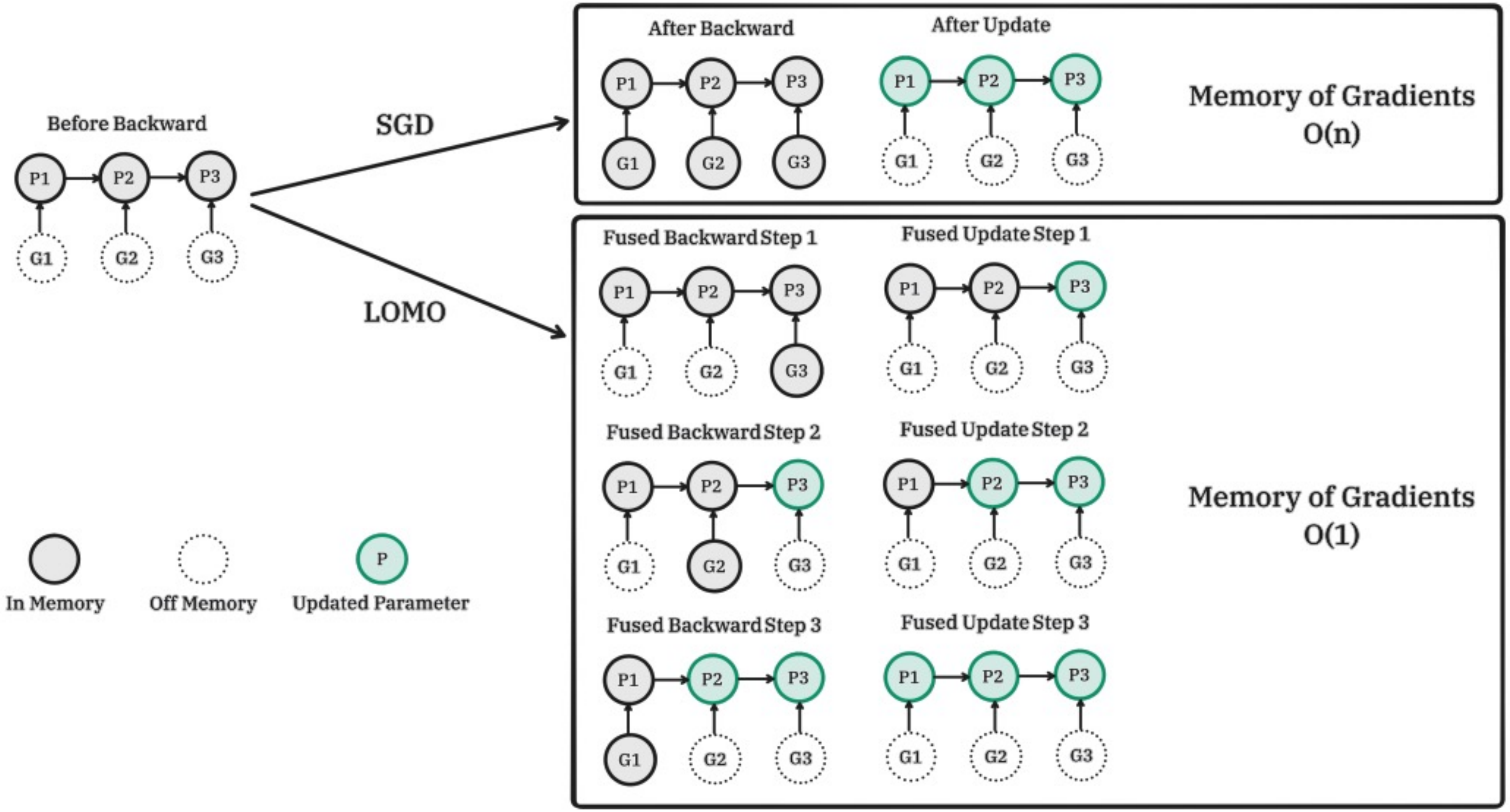
## Quatization



# 3 – LLM Efficiency Challenge

!

## Low-Memory Optimization (LOMO)



# 4 - Experiment

!

## Source code

# Thanks!

## Any questions?