① Quantization

→ full precision / half precision

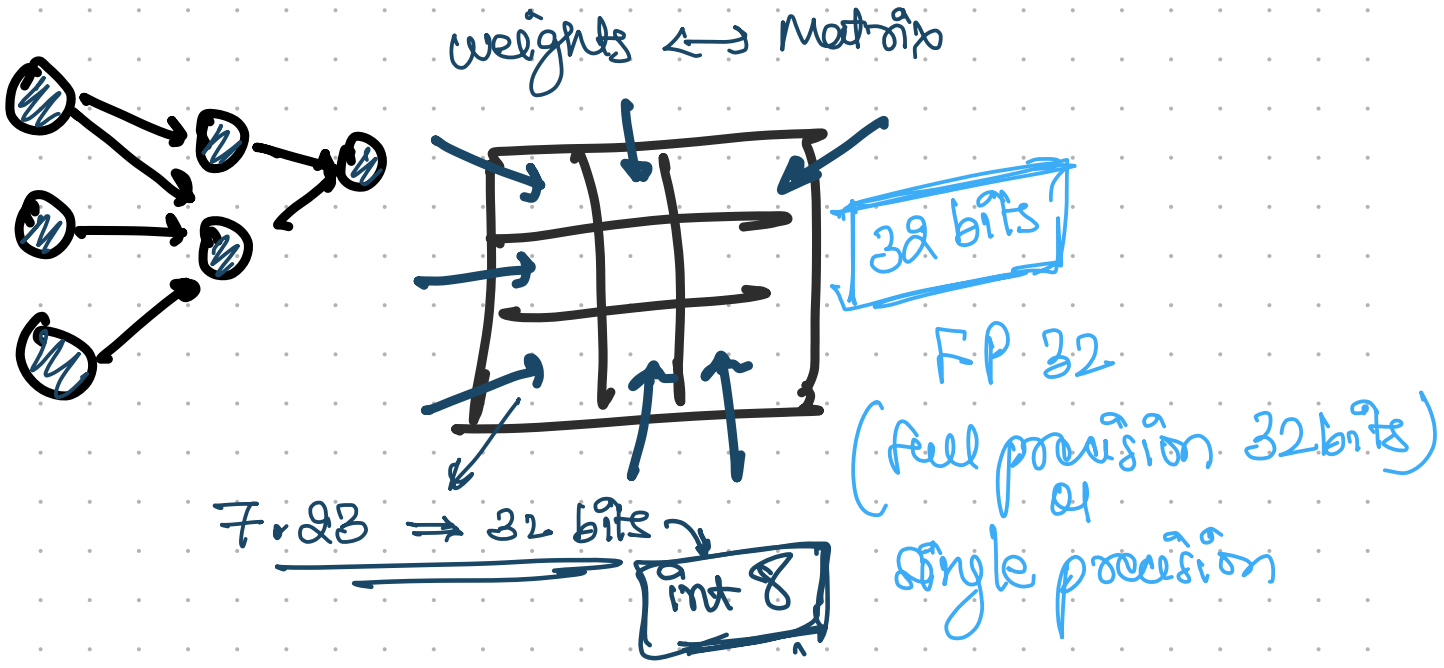
→ Calibration → (Model Quantization)

→ Modes of Quant. ① Post training Quantization

② Quantization-Aware training

→ Data → weights and parameters.

Quantization: conversion from linear memory format to a lower memory format.



LLaMA-2 70 billion ⇒ can't be used in GPUs

↳ Lexico ≈ 28+ GiB per prompt.

vLLM ⇒ 16 GiB

phi-2 occupied (2B) around 12 GiB.

So let's say convert the 32 bit model to 8 bit and then use it for inferences.

But what's the catch?

Accuracy !!!

So let's say:-

we can

FP (32 bit) → FP (16 bit)

full precision → half precision

How to perform quantization: → Symmetric

→ Asymmetric.

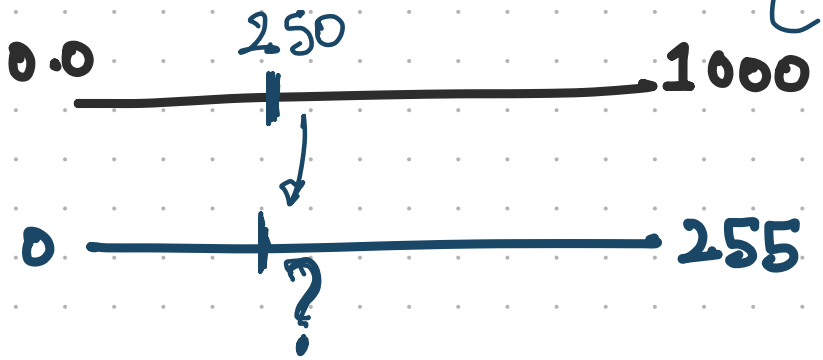
(A) Symmetric: unsigned int8 quantization.

[0.00 ..... 1000] → weights values → 32 bits

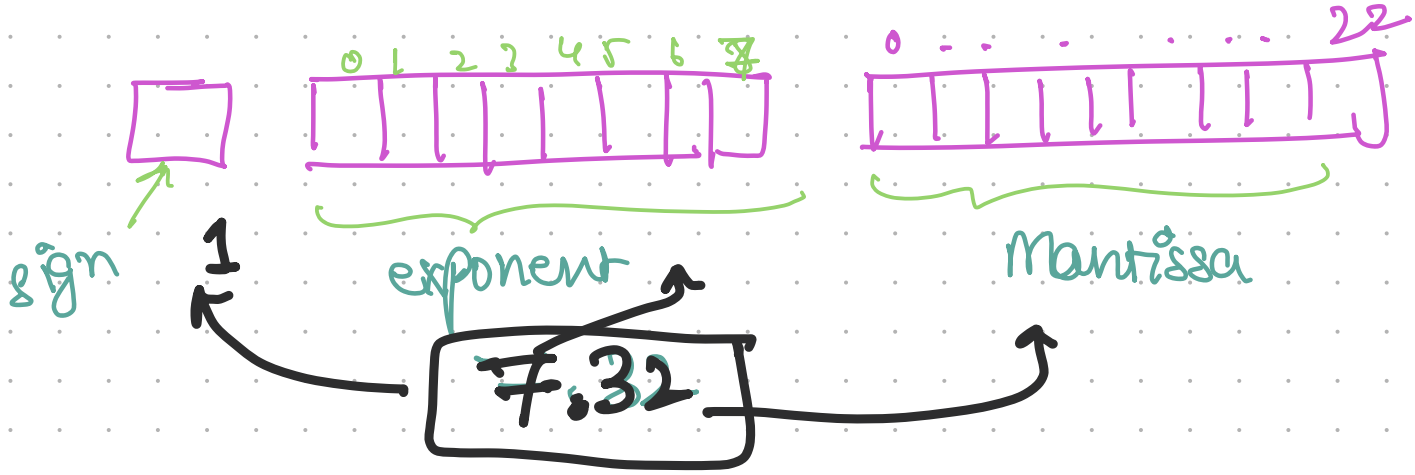
↓

uint8 → 8 bits ⇒ 2^8

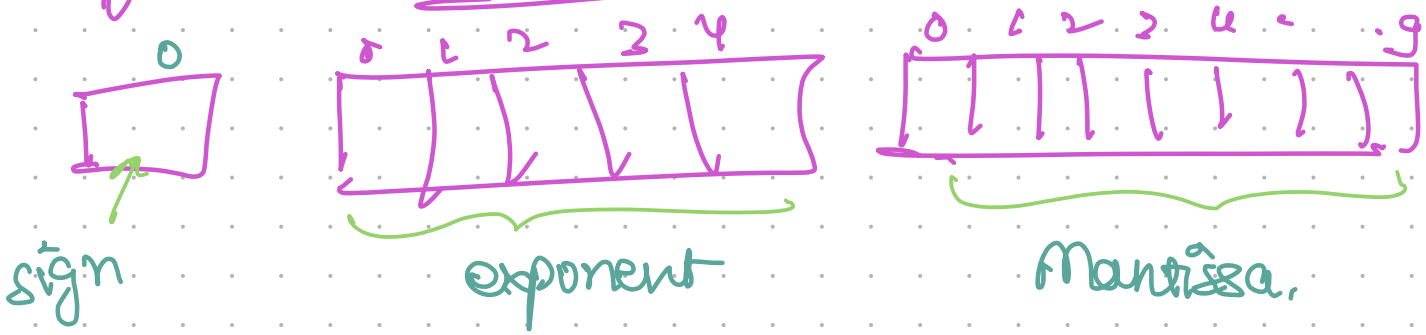
[0-255]



Single precision floating point 32 (1-8-23)



Half precision FP 16 (1-5-10)



How to do it & map the two scales?

Min-Max scaler

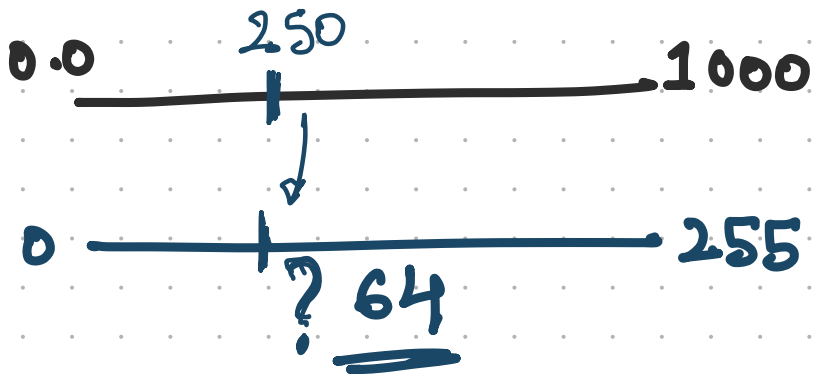
0.0 → 0

1000 → 255

Scale factor $= \frac{x_{max} - x_{min}}{y_{max} - y_{min}}$

So now, $\frac{1000 - 0}{255 - 0} = 3.92$ ← scale factor

round $\left(\frac{250}{3.92}\right) = (63.77) \sim 64$



(2) Asymmetric uint8 quantization

[−20.0 ..... 1000]

unsigned: [0 ..... 255]

signed: | -127 to 128 |

$$\text{scale} = \frac{x_{max} - x_{min}}{q_{max} - q_{min}} = \frac{1000 - (-20)}{255}$$

$$\frac{1000 + 20}{255} = 4.0$$

scale factor.

[-20.0 ..... 1000] -20 ——— 1000

[0 ..... 255] 0 ——— 255

round $\left(\frac{-20}{4}\right)$ = -5.0 → But our distribution starts from '0'.

So we can, (-5.0) + 5 = 0

5 → zero point. [for symmetric, the zero point was '0'.]

scale = 4.0
zeropoint = 5

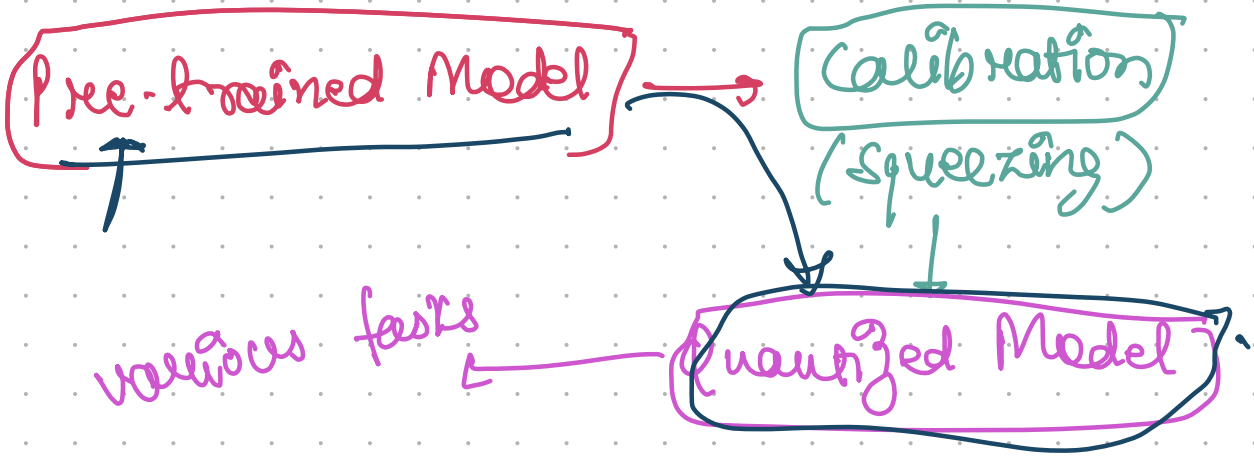
for signed bits, it will be -128 to 128.

So the scale looks like

-128 ——— 128

and the formula changes!!!

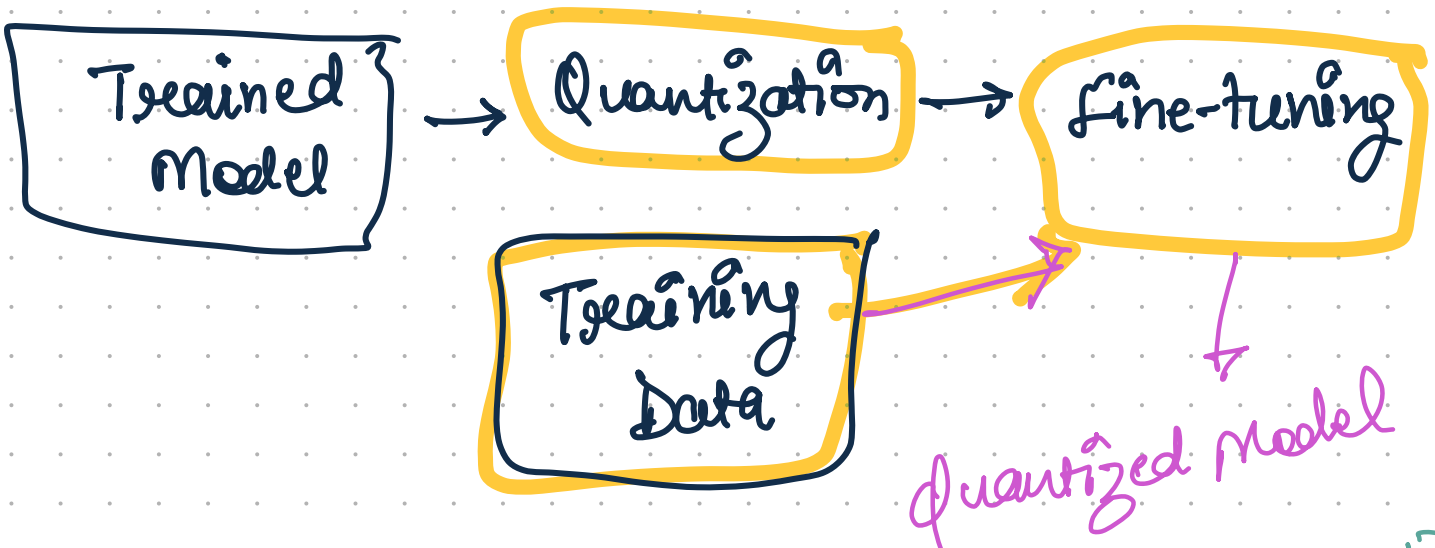
(A) Post-training quantization :- (PTQ)



vLLM library → LLAMa, Mixtral, Multi-modals (LLAVA)

loss of data ⟹ Accuracy ↓

(B) Quantization-Aware Training (QAT)



[Accuracy is usually preserved!]

Low-Rank Adaptation of LLMs

LoRA: Low-Rank Adaptation of Large Language Models

Edward Hu* **Yelong Shen*** **Phillip Wallis** **Zeyuan Allen-Zhu**
Yuanzhi Li **Shean Wang** **Lu Wang** **Weizhu Chen**
Microsoft Corporation
{edwardhu, yeshe, phwallis, zeyuana,
yuanzhil, swang, luw, wzchen}@microsoft.com
yuanzhil@andrew.cmu.edu
(Version 2)

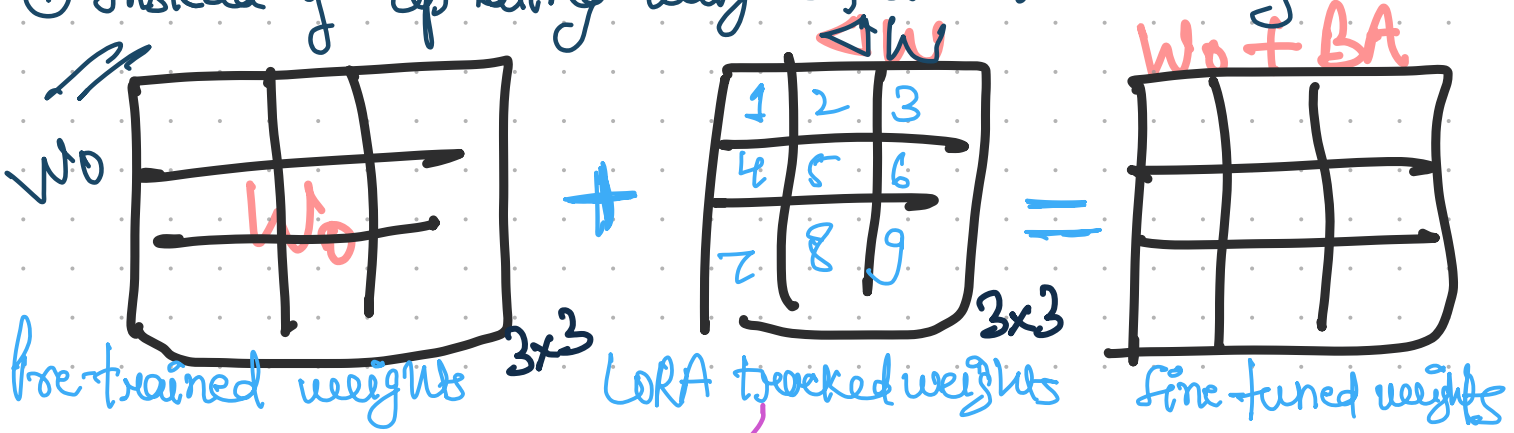
Microsoft
ICLR 22?

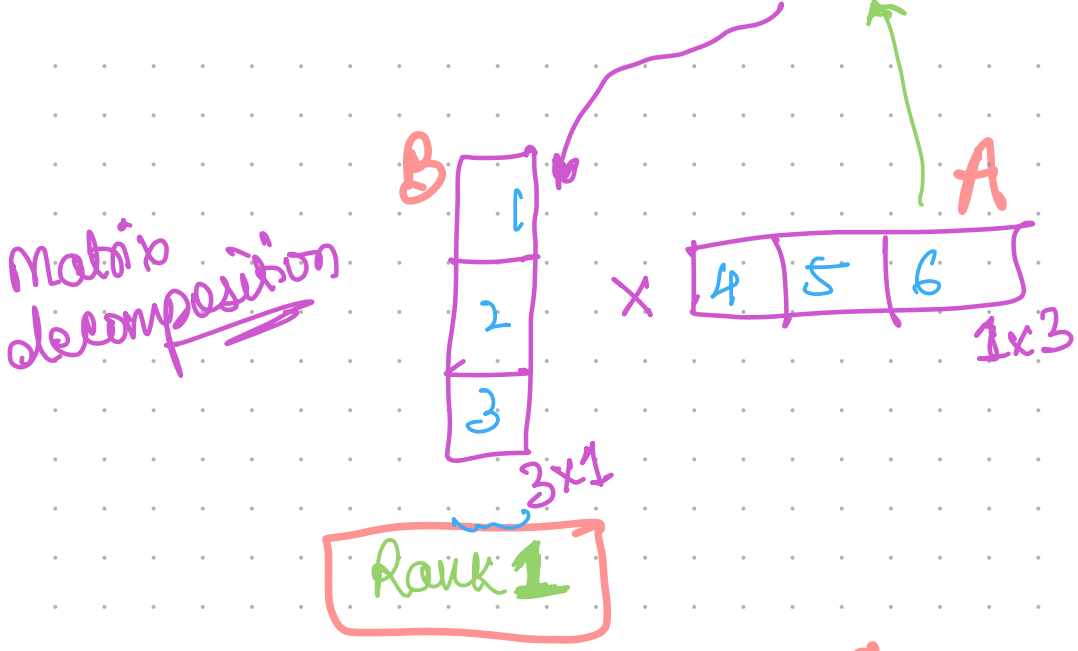
Abstract

An important paradigm of natural language processing consists of large-scale pre-training on general domain data and adaptation to particular tasks or domains. As we pre-train larger models, full fine-tuning, which retrains all model parameters, becomes less feasible. Using GPT-3 175B as an example – deploying independent instances of fine-tuned models, each with 175B parameters, is prohibitively expensive. We propose **Lo**w-**R**ank **A**daptation, or LoRA, which freezes the pre-trained model weights and injects trainable rank decomposition matrices into each layer of the Transformer architecture, greatly reducing the number of trainable parameters for downstream tasks. Compared to GPT-3 175B fine-tuned with Adam, LoRA can reduce the number of trainable parameters by 10,000 times and the GPU memory requirement by 3 times. LoRA performs on-par or better than fine-tuning in model quality on RoBERTa, DeBERTa, GPT-2, and GPT-3, despite having fewer trainable parameters, a higher training throughput, and, unlike adapters, *no additional inference latency*. We also provide an empirical investigation into rank-deficiency in language model adaptation, which sheds light on the efficacy of LoRA. We release a package that facilitates the integration of LoRA with PyTorch models and provide our implementations and model checkpoints for RoBERTa, DeBERTa, and GPT-2 at https://github.com/microsoft/LoRA.

What exactly LoRA do?

① Instead of updating weights, it track changes.





If we keep on increasing ranks, the final B' & A' will be lesser than actual ΔW.

	Hyperparameters	# Trainable Parameters	WikiSQL	MNLI-m
Fine-Tune	-	175B	73.8	89.5
PrefixEmbed	$l_p = 32, l_i = 8$	0.4 M	55.9	84.9
	$l_p = 64, l_i = 8$	0.9 M	58.7	88.1
	$l_p = 128, l_i = 8$	1.7 M	60.6	88.0
	$l_p = 256, l_i = 8$	3.2 M	63.1	88.6
	$l_p = 512, l_i = 8$	6.4 M	55.9	85.8
PrefixLayer	$l_p = 2, l_i = 2$	5.1 M	68.5	89.2
	$l_p = 8, l_i = 0$	10.1 M	69.8	88.2
	$l_p = 8, l_i = 8$	20.2 M	70.1	89.5
	$l_p = 32, l_i = 4$	44.1 M	66.4	89.6
	$l_p = 64, l_i = 0$	76.1 M	64.9	87.9
Adapter^H	$r = 1$	7.1 M	71.9	89.8
	$r = 4$	21.2 M	73.2	91.0
	$r = 8$	40.1 M	73.2	91.5
	$r = 16$	77.9 M	73.2	91.5
	$r = 64$	304.4 M	72.6	91.5
LoRA	$r_v = 2$	4.7 M	73.4	**91.7**
	$r_q = r_v = 1$	4.7 M	73.4	91.3
	$r_q = r_v = 2$	9.4 M	73.3	91.4
	$r_q = r_k = r_v = r_o = 1$	9.4 M	74.1	91.2
	$r_q = r_v = 4$	18.8 M	73.7	91.3
	$r_q = r_k = r_v = r_o = 2$	18.8 M	73.7	**91.7**
	$r_q = r_v = 8$	37.7 M	73.8	**91.6**
	$r_q = r_k = r_v = r_o = 4$	37.7 M	74.0	**91.7**
	$r_q = r_v = 64$	301.9 M	73.6	91.4
	$r_q = r_k = r_v = r_o = 64$	603.8 M	73.9	91.4
LoRA+PE	$r_q = r_v = 8, l_p = 8, l_i = 4$	37.8 M	75.0	91.4
	$r_q = r_v = 32, l_p = 8, l_i = 4$	151.1 M	**75.9**	91.1
	$r_q = r_v = 64, l_p = 8, l_i = 4$	302.1 M	**76.2**	91.3
LoRA+PL	$r_q = r_v = 8, l_p = 8, l_i = 4$	52.8 M	72.9	90.2

Rank ↑ ΔW ↑ decompose

Highly decomposed Matrix

603 <<< 175B

Table 15: Hyperparameter analysis of different adaptation approaches on WikiSQL and MNLI. Both prefix-embedding tuning (PrefixEmbed) and prefix-layer tuning (PrefixLayer) perform worse as we increase the number of trainable parameters, while LoRA's performance stabilizes. Performance is measured in validation accuracy.

Rank ∝ # of trainable parameters (Δw) also increases

From paper!
Optimal rank is 8 and that gave good results ☺.

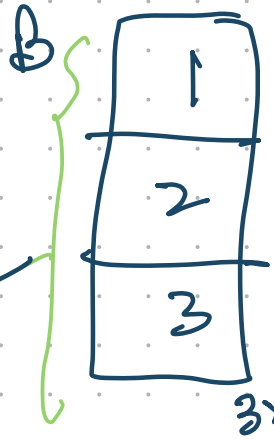
High rank ↑ → complex things. ↑
where [precision is required]

2.2

QLoRA (Quantized LoRA)

NeurIPS 24'

Matrix decomposition



Float 16 bit → Half → 4 bit

approximately 3GB for 65B

So Quantization with F.T.

LLAMA 2 → 7 Billion

FP 32 bit → int 8 bits → 4 bit → 1.58 bit

Can we Reduce it even further?

***

The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits

Shuming Ma* **Hongyu Wang*** **Lingxiao Ma** **Lei Wang** **Wenhui Wang**
Shaohan Huang **Li Dong** **Ruiping Wang** **Jilong Xue** **Furu Wei◇**
https://aka.ms/GeneralAI

Abstract

Recent research, such as BitNet [WMD+23], is paving the way for a new era of 1-bit Large Language Models (LLMs). In this work, we introduce a 1-bit LLM variant, namely **BitNet b1.58**, in which every single parameter (or weight) of the LLM is ternary {-1, 0, 1}. It matches the full-precision (i.e., FP16 or BF16) Transformer LLM with the same model size and training tokens in terms of both perplexity and end-task performance, while being significantly more cost-effective in terms of latency, memory, throughput, and energy consumption. More profoundly, the 1.58-bit LLM defines a new scaling law and recipe for training new generations of LLMs that are both high-performance and cost-effective. Furthermore, it enables a new computation paradigm and opens the door for designing specific hardware optimized for 1-bit LLMs.

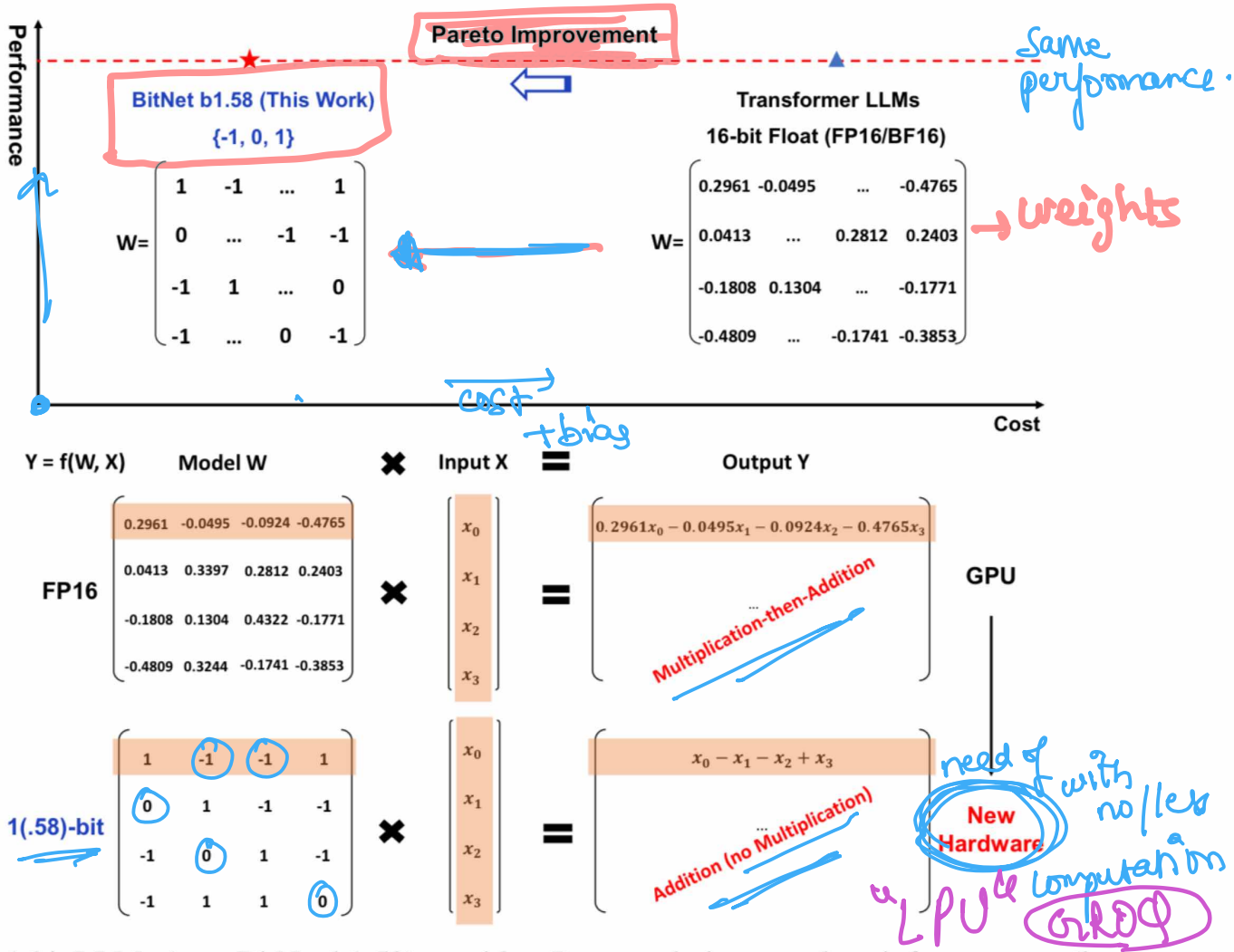


Figure 1: 1-bit LLMs (e.g., BitNet b1.58) provide a Pareto solution to reduce inference cost (latency, throughput, and energy) of LLMs while maintaining model performance. The new computation paradigm of BitNet b1.58 calls for actions to design new hardware optimized for 1-bit LLMs.

Trend has moved from:

32 bits → 16 bits → 4 bits → 1.58 bits.

Major computations floating point addition & multiplication

They need an additional value of '0' of the original 1 bit BitNet, resulting in 1.58 bits in binary system.

Helpful in:

① explicit support for feature filtering.

$x \times 0 = 0$ Input is multiplied with 0.

② full precision (i.e. FP16) performance can be matched for perplexity & end-task performance.

How?

So the quantization function: They adapt "absmean".

① first scale the weight matrix by its avg. absolute value.

② Round each value to nearest integer among $\{-1, 0, +1\}$.

2 BitNet b1.58

BitNet b1.58 is based on the BitNet architecture, which is a Transformer that replaces *nn.Linear* with *BitLinear*. It is trained from scratch, with 1.58-bit weights and 8-bit activations. Compared to the original BitNet, it introduces some modifications that we summarize below.

Quantization Function. To constrain the weights to -1, 0, or +1, we adopt an *absmean* quantization function. It first scales the weight matrix by its average absolute value, and then round each value to the nearest integer among {-1, 0, +1}:

$$\widetilde{W} = \text{RoundClip}\left(\frac{W}{\gamma + \epsilon}, -1, 1\right), \tag{1}$$

$$\text{RoundClip}(x, a, b) = \max(a, \min(b, \text{round}(x))), \tag{2}$$

$$\gamma = \frac{1}{nm}\sum_{ij}|W_{ij}|. \tag{3}$$

The quantization function for activations follows the same implementation in BitNet, except that we do not scale the activations before the non-linear functions to the range $[0, Q_b]$. Instead, the

Models	Size	Memory (GB)↓	Latency (ms)↓	PPL↓
LLaMA LLM	700M	2.08 (1.00x)	1.18 (1.00x)	12.33
BitNet b1.58	700M	0.80 (2.60x)	0.96 (1.23x)	12.87
LLaMA LLM	1.3B	3.34 (1.00x)	1.62 (1.00x)	11.25
BitNet b1.58	1.3B	1.14 (2.93x)	0.97 (1.67x)	11.29
LLaMA LLM	3B	7.89 (1.00x)	5.07 (1.00x)	10.04
BitNet b1.58	3B	**2.22 (3.55x)**	**1.87 (2.71x)**	**9.91**
BitNet b1.58	3.9B	**2.38 (3.32x)**	**2.11 (2.40x)**	**9.62**

Table 1: Perplexity as well as the cost of BitNet b1.58 and LLaMA LLM.

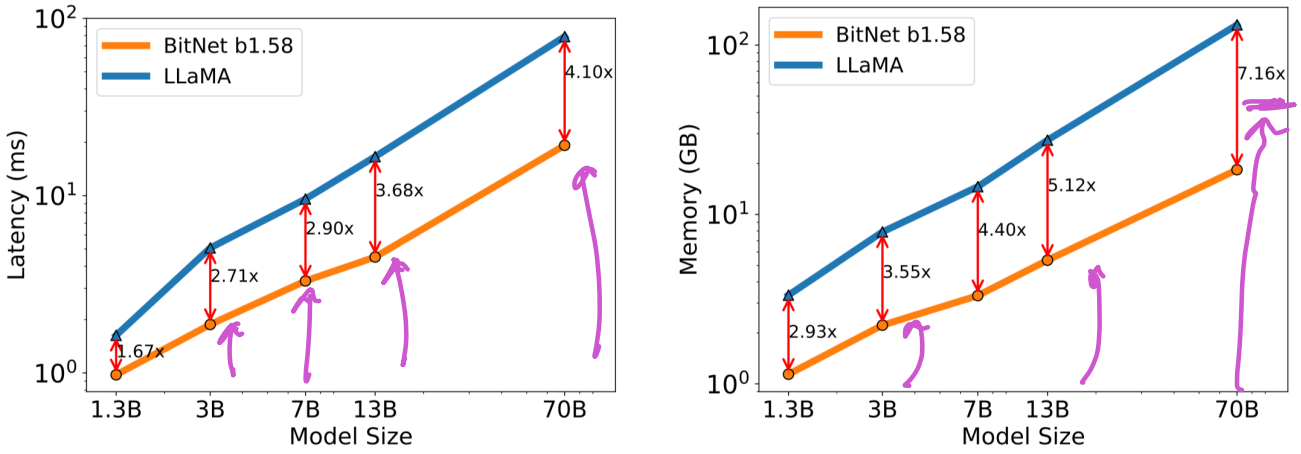


Figure 2: Decoding latency (Left) and memory consumption (Right) of BitNet b1.58 varying the model size.