Scaling Laws and Model Quantization

AI with Deep Learning

EE4016

Prof. Lai-Man Po

Department of Electrical Engineering

City University of Hong Kong

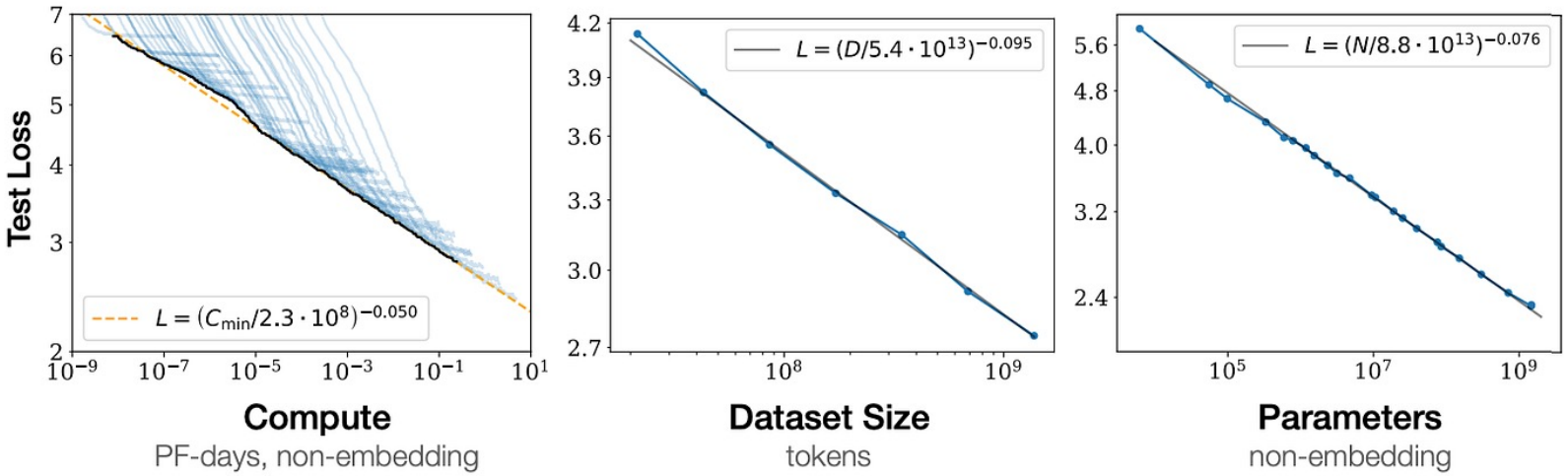
Content

- **LLM Scaling Laws**
- **Closed and Open LLMs**
- **Why Model Quantization?**
- **Numerical Data Types:**
  - Integer, Fix-Point, Floating-Point (FP32, FP16, BF16, FP8, etc)
- **Linear Quantization**
- **Quantization Aware Training (QAT)**
- **Post Training Quantization (PTQ)**
  - GGUF, AWQ, GTPG and NF4
- **The Era of 1.58-Bit LLMs**

OpenAI LLM Scaling Laws (2020)

OpenAI Scaling Laws (2020) — *Kaplan et al., "Scaling Laws for Neural Language Models"*

- Through large-scale experiments, OpenAI found that a model's **test loss** L — a measure of performance — decreases smoothly and predictably as **model parameters** (*N*), **dataset size** (*D*), and **compute** (*C*) increase.



On a log-log plot, this manifests as a straight line ($\log x^{\alpha} = \alpha \log x$) — a hallmark of power-law behavior — making performance extrapolation possible.

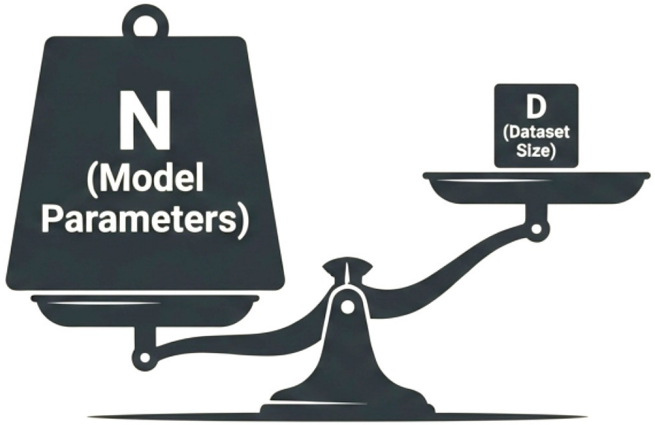
Kaplan Scaling Laws: Power-Law Relationships

In essence, as we scale model size *N* (Parameters), Dataset size *D*, or compute *C*, performance improves in a predictable, logarithmic fashion.

- **Power-Law Relationships**:
  - Mode Size: $L(N) = \frac{N_c}{N^{\alpha}} + L_{\infty}$
  - Data Size: $L(D) = \frac{D_c}{D^{\beta}} + L_{\infty}$
  - Compute: $L(C) = \frac{C_c}{C^{\gamma}} + L_{\infty}$

Fitted slope: $\alpha \approx 0.076, \beta \approx 0.095, \text{and } \gamma \approx 0.05$

Under fixed compute, these coefficients suggested the most efficient are to **prioritize parameters above all else**.



Even with modest data, compute was best spent maximizing parameters. This birthed massive, chronically "starved" monuments like GPT-3 (175B).

The Kaplan Scaling Laws dictating "Bigger is Better"

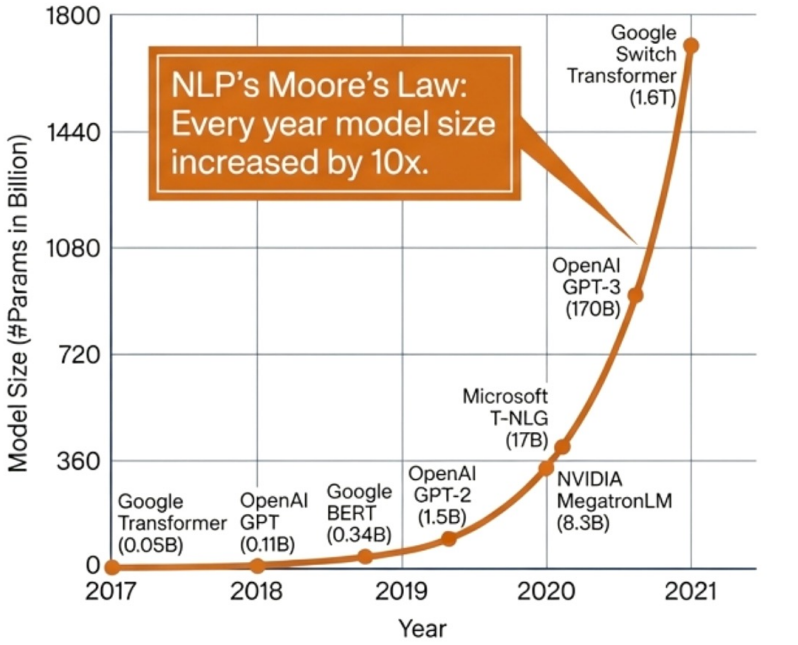
Model Parameter Constant

$$L(N) = \frac{N_c}{N^{\alpha}} + L_{\infty}$$

Model Parameters

$\alpha \approx 0.076$

Kaplan's 2020 laws proved performance improves smoothly with scale. The most efficient use of compute was to prioritize the number of parameters above all else.



Archetype: GPT-3 reaches an unprecedented 175 Billion parameters.

Chinchilla Scaling Laws (2022)

Chinchilla Scaling Laws (2022) — from DeepMind's paper "Training Compute-Optimal Large Language Models" — **showed that most large models like GPT-3 were undertrained**.

The corrected joint scaling law is

$$L(N, D) = E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}}$$

with $\alpha \approx 0.34$, $\beta \approx 0.28$, and E the irreducible loss.

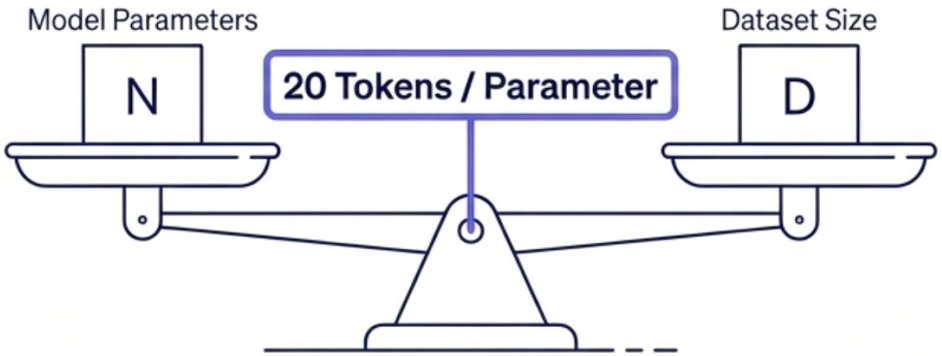
- The Chinchilla Rule states that for a fixed compute budget, optimal performance occurs at roughly **20 tokens per parameter**.
- The 70B-parameter Chinchilla model, trained on 1.4T tokens, proved that smaller, data-rich models outperform larger, under-trained ones.

Chinchilla Model Performance Gains

- The **Chinchilla model (70B parameters, 1.4T tokens)** matched or beat **GPT-3 (175B, 300B tokens)** across many benchmarks—using *4× less compute at inference*

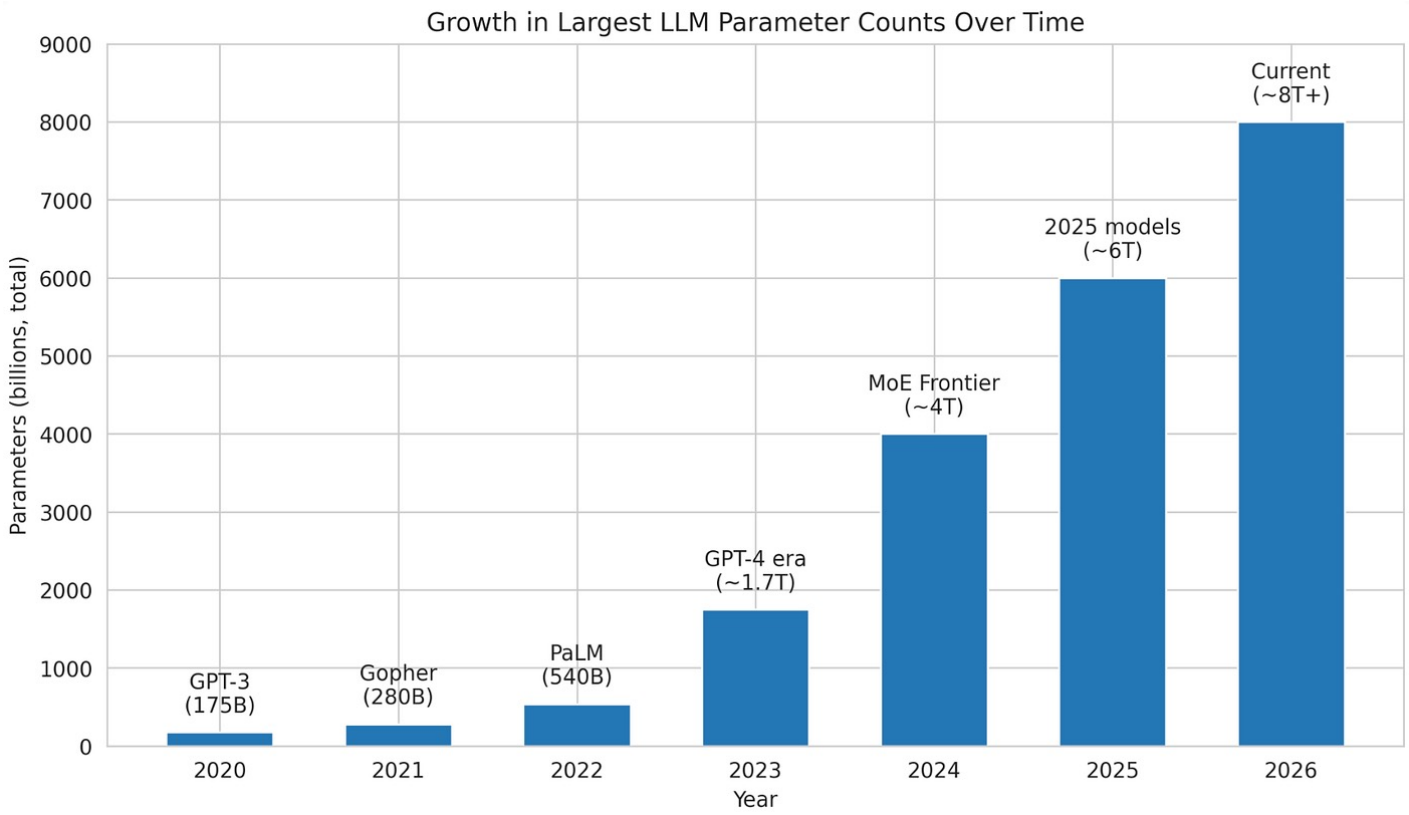
Model	Size (# Parameters)	Training Tokens
LaMDA (Thoppilan et al., 2022)	137 Billion	168 Billion
GPT-3 (Brown et al., 2020)	175 Billion	300 Billion
Jurassic (Lieber et al., 2021)	178 Billion	300 Billion
Gopher (Rae et al., 2021)	280 Billion	300 Billion
MT-NLG 530B (Smith et al., 2022)	530 Billion	270 Billion
Chinchilla	70 Billion	1.4 Trillion

The Chinchilla Pivot Discovered the Optimal Balance



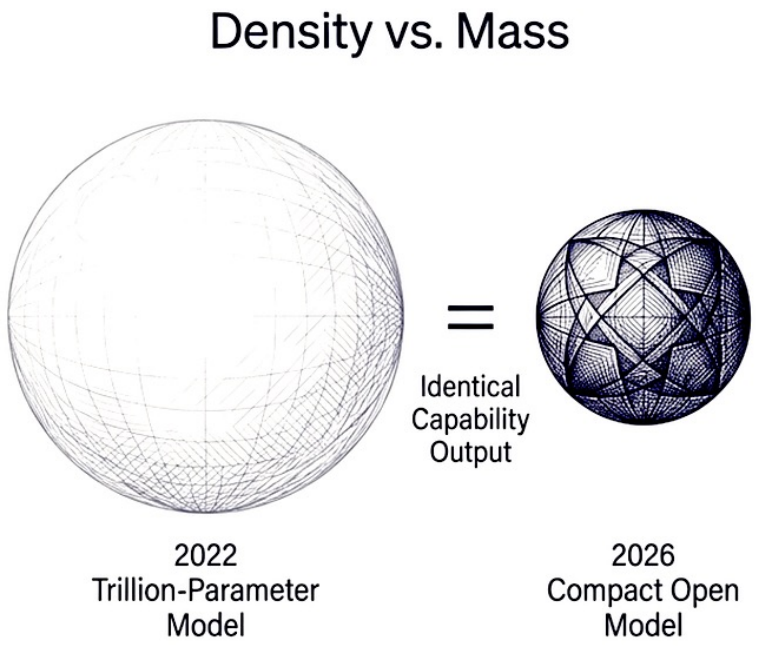
Philosophies of Scale	Kaplan Era (2020)	Chinchilla Pivot (2022)
Primary Goal	Maximize Model Size (N)	Balance Size (N) and Data (D) equally
Optimal Ratio	Favors size regardless of data	The Rule of Thumb: ~20 tokens per parameter
Model Health	Chronically “undertrained”	“Data-optimal” training ($C \approx 6ND$)
Resulting Archetype	GPT-3 (Undertrained, Ratio ~1.7:1)	Llama 2 (Overtrained, Ratio ~29:1 to 50:1)

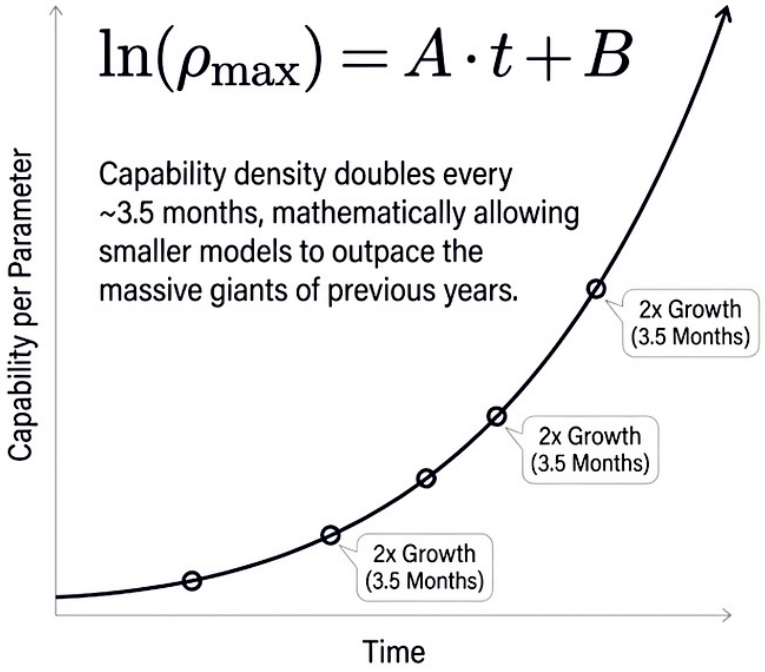
Growth in Largest LLM Parameter Counts Over Time



The Densing Law (2025)

By 2025, the industry began prioritizing “capability density”— the performance we extract per parameter.

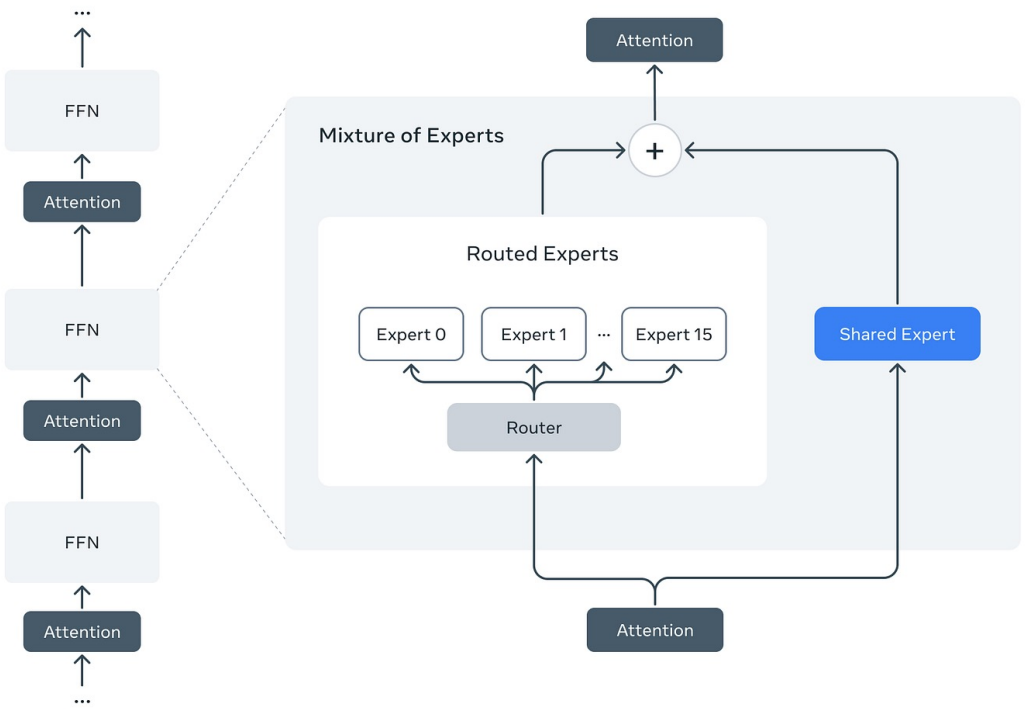




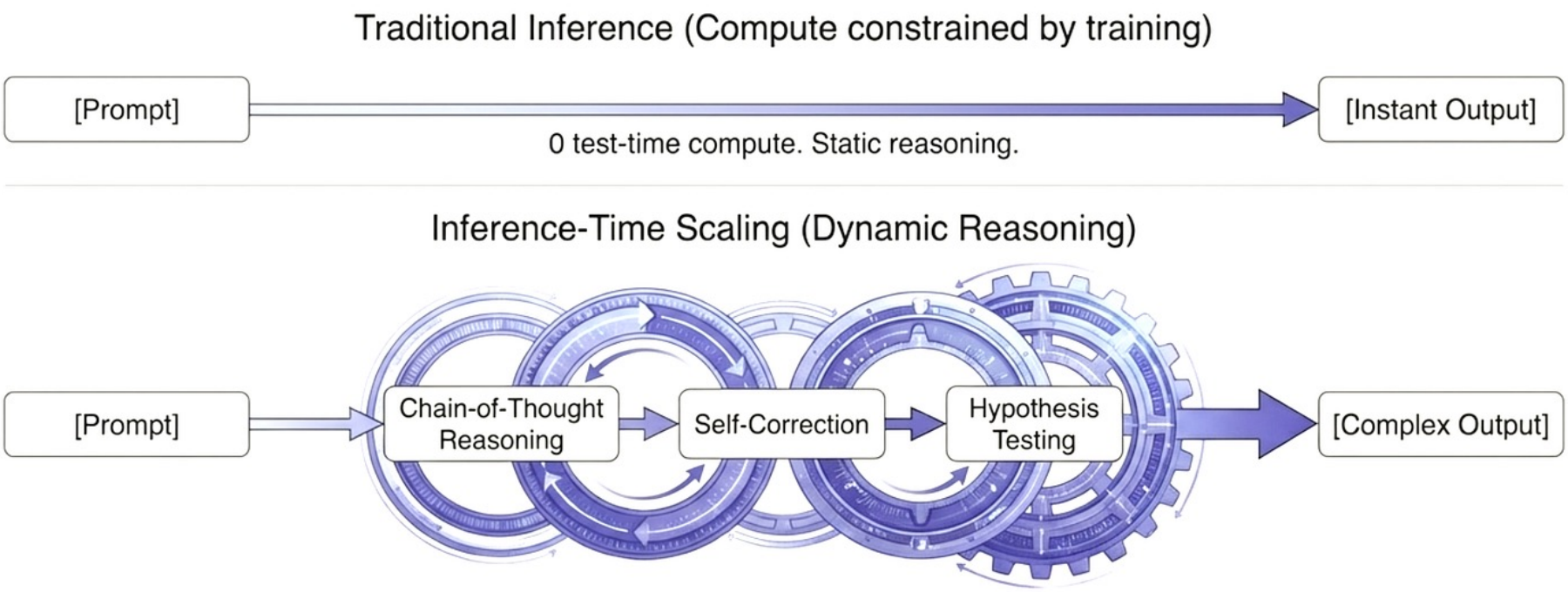
Engine 1: Activating Only the Necessary Pathways

Mixture of Experts (MoE) keeps "knowledge capacity" massive by storing total parameters but keeps compute costs low by only activating a fraction of them per token.

2026 Real-World Specs
Llama 4 Scout: 109B Total Parameters / Only 17B Active
DeepSeek-V3.2: 671B Total Parameters / Only ~37B Active

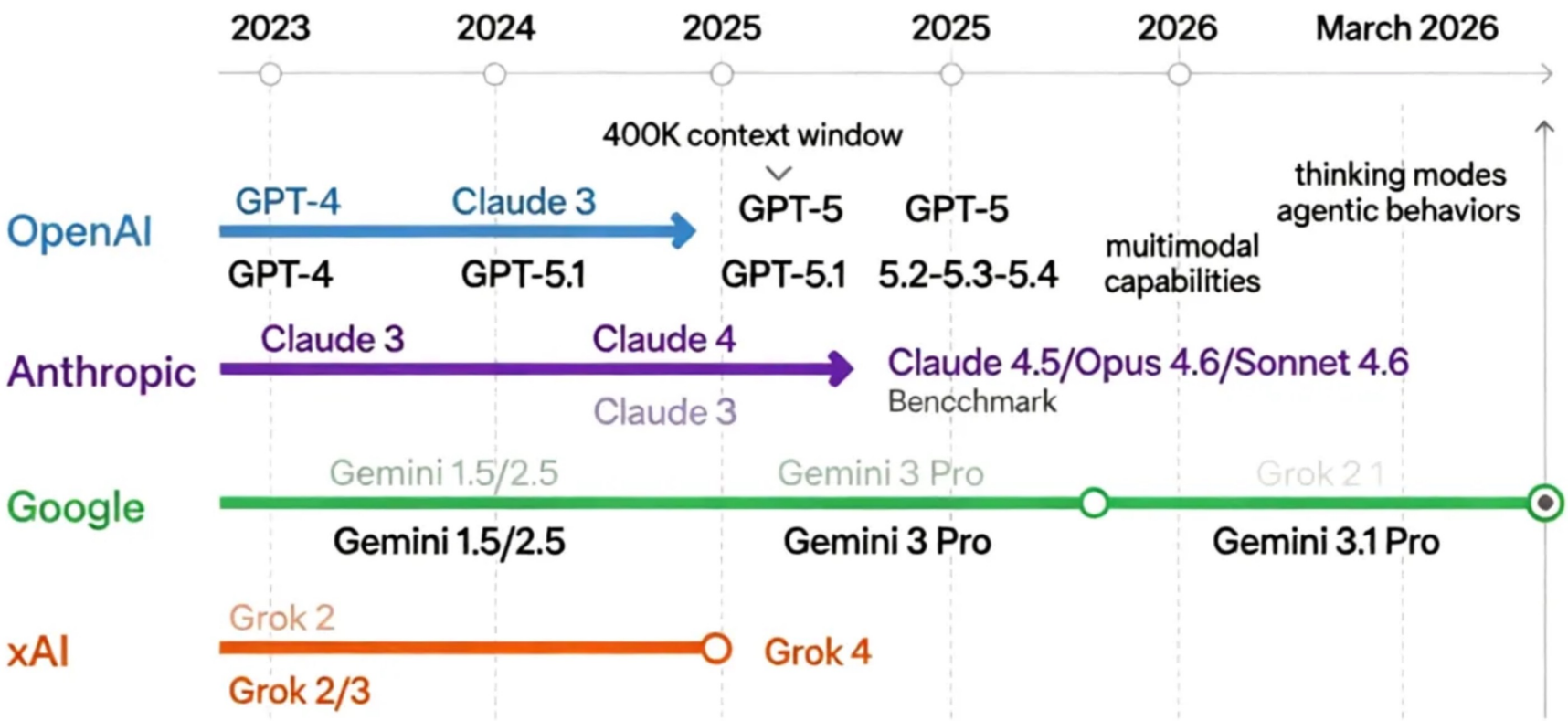


Engine 2: Unlocking Complexity Through Test-Time Compute



Instead of delivering instant responses, the model dynamically allocates extra compute during the answer phase. This entirely bypasses traditional scaling limits, allowing smaller models to emulate the reasoning depth of much larger systems.

The Closed-Source Frontier: Proprietary LLM Development



Closed labs drove the scaling race with secret architectures and massive compute.

The Open-Weight Revolution Shattered the Capability Ceiling

Meta's Llama 4 Herd

The Multimodal Standard

- First natively multimodal open-weight family.
- Standout: Llama 4 Scout featuring a 10M token context (industry-leading).
- Outperforms Gemma 3 and Mistral 3.1 across broad benchmarks.

DeepSeek-V3.2

The Efficiency Master

- Matches GPT-5 and Gemini 3 on advanced math and coding (AIME, HMMT).
- Utilizes sparse attention and RL enhancements.
- Released under an MIT license for extreme accessibility.

Alibaba's Qwen3 Series

The Multilingual Powerhouse

- Exceptional code and long-context capabilities.
- Variants like Qwen3-122B consistently beat GPT-5-mini on multiple tasks.
- Apache 2.0 open-source license.

The 2026 Frontier Operates at Complete Parity

	Closed Giants (OpenAI, Anthropic, Google)	**Open Powerhouses** (Meta, DeepSeek, Alibaba)
Primary Architecture	Massive Unified Systems & Secret MoEs	Highly efficient sparse MoEs (e.g., 37B active parameters)
Context Window Limit	400K to 1M+ Tokens	Up to 10M Tokens (Llama 4)
Deployment Model	API Rental Only (Zero code visibility)	Downloadable Weights / 100% Self-Hosted
Primary Strategic Advantage	Controlled rollout, fully managed enterprise integrations.	Full parameter inspection, localized fine-tuning, absolute data privacy.

LLM Quantization

Quantization: Making LLMs Lighter

- **Large language models (LLMs) are memory hungry.** Model Quantization is to convert high-precision floating-point numbers in neural networks to low-precision representations.
- This process, essentially a function mapping, offers several advantages:
  1. **Smaller model size**
  2. **Reduced memory usage** during inference
  3. **Faster inference** on low-precision optimized processors

Model	Original Size (FP32)	Quantized Size (INT4)
LLaMA3.1-8B	38.4 GB	4.8 GB
LLaMA3.1-70B	336 GB	42 GB
LLaMA3.1-405B	1,994 GB	243 GB

Estimating VRAM needs for LLaMA3.1 models: 8B requires moderate, 70B needs more, and 405B demands substantial capacity for efficient inference.

A Guide to Estimating VRAM for LLMs

The Role of Lower Precision in Deep Learning

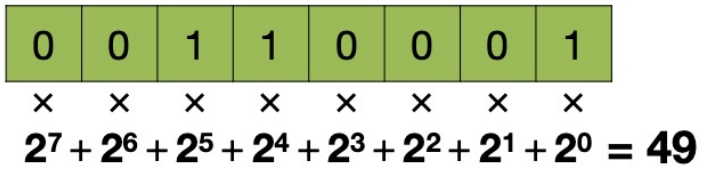
- High precision in neural networks typically means better accuracy and more stable training, but it requires expensive, computationally intensive hardware. However, research by **Google and NVIDIA has shown that lower precision can be effective for certain operations**.
- Both companies have developed hardware and frameworks to support lower precision. For example:
  - **NVIDIA's T4** accelerators offer improved efficiency with Tensor Cores technology
  - **Google's TPUs** introduced the bfloat16 data type, optimized for neural networks.
- The idea behind lower precision is that **neural networks don't always need the full range of 64-bit floats to perform well**. By using lower precision, neural networks can achieve good results while reducing computational costs.

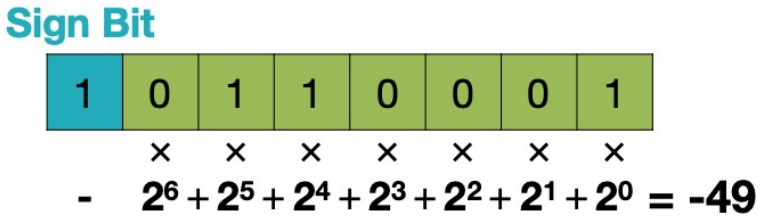
Numeric Data Types

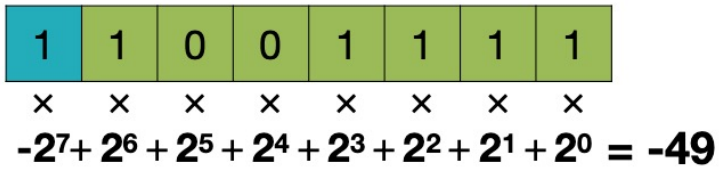
- How is numeric data represented in modern computing systems?
  - **Integer**
  - **Fixed-Point**
  - **Floating-Point**
    - IEEE Standards: FP64, FP32, FP16, FP8
    - Google: BF16
    - Nvidia: FP8 (E4M3), FP8 (E5M2)

Integer Representations

- **Unsigned Integer**
  - n-bit Range: $[0, 2^n - 1]$
- **Signed Integer**
  - **Sign-Magnitude Representation**
    - n-bit Range: $[-2^{n-1} - 1,\ 2^{n-1} - 1]$
    - **Two zeros:** Both 000…00 and 100…00 represent 0
  - **Two's Complement Representation**
    - n-bit Range: $[-2^{n-1},\ 2^{n-1} - 1]$
    - 000…00 represent 0
    - 100…00 represent -2^{n-1}



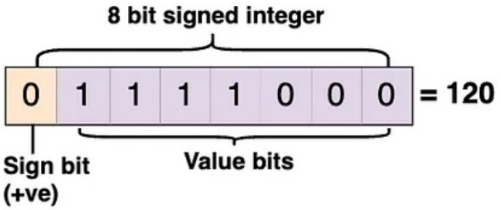


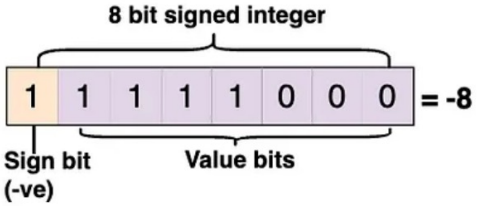


https://hanlab.mit.edu/courses/2023-fall-65940

UINT8 and INT8

- **UINT8: Unsinged Integer with Range (0, 256)**





Calculation: 2's compliment

$= 0 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0$

$= 0 + 64 + 32 + 16 + 8 + 0 + 0 + 0$

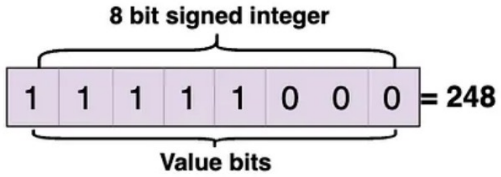
$= 120$

$= -1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0$

$= -128 + 64 + 32 + 16 + 8 + 0 + 0 + 0$

$= -8$

- **INT8: Signed Integer with Range (-128, 127)**



Two's Complement Representation

$= 1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0$

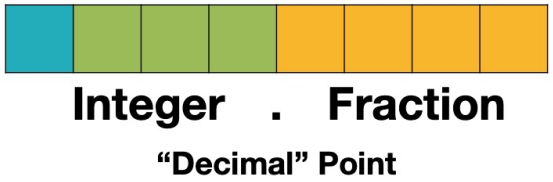
$= 128 + 64 + 32 + 16 + 8 + 0 + 0 + 0$

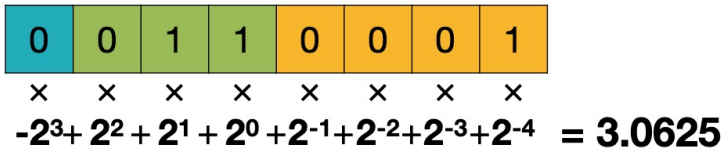
$= 248$

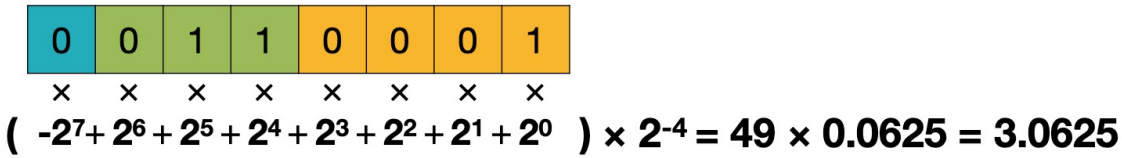
https://medium.com/@tamangmilan/want-to-learn-quantization-in-the-large-language-model-57f062d2ec17

Fixed-Point Number

- Using 2's complement representation, while **a fixed number of bits are allocated for the integer and fractional parts of a number**







https://hanlab.mit.edu/courses/2023-fall-65940

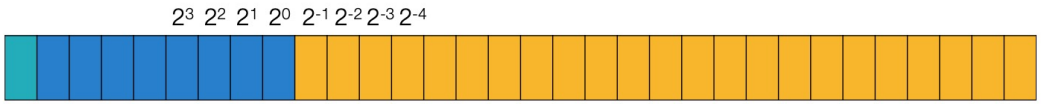
Floating-Point Number

- Floating-point representation uses a more complex format that separates a number into three components: sign, exponent, and mantissa (aka fraction and significand).

$$(-1)^{\text{sign}} \times 2^{\text{exponent}-127} \times (1 + \text{mantissa})$$

  - the sign indicates whether the number is **positive or negative**,
  - the exponent determines the **scale of the number**,
  - the mantissa holds the **fraction digits**.
- This format allows for a wider range of numbers and higher precision compared to fixed-point representation.
  - The **range** is determined by the **exponent**, which can adjust the scale of the number, allowing for very large or very small values.
  - The **precision** is determined by the number of bits allocated to the **mantissa**, which represents the significant digits.

32-bit Floating-Point Number in IEEE 754

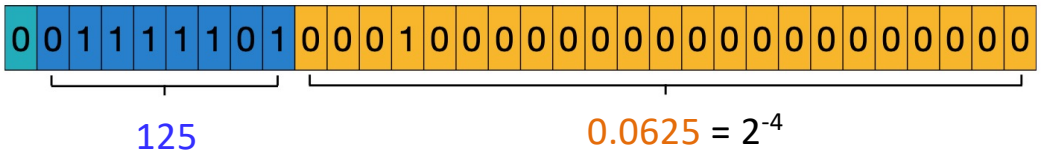


Sign **8-bit Exponent** **23-bit Mantissa**

$$(-1)^{sign} \times 2^{exponent-127} \times (1 + mantissa)$$ — Exponent Bias = 127 = $2^{8-1}-1$

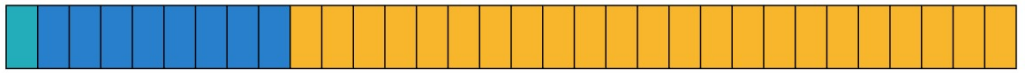
mantissa: fraction or significant

$$0.265625 = (-1)^0 \times 2^{-2} \times 1.0625 = (-1)^0 \times 2^{125-127} \times (1 + 0.0625)$$



https://hanlab.mit.edu/courses/2023-fall-65940

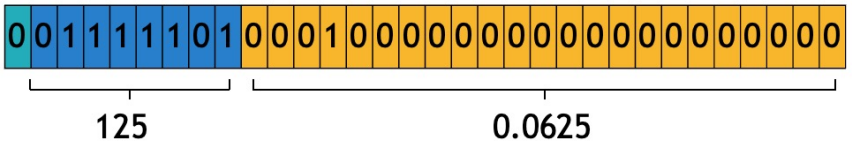
Floating-Point Zero Value as Subnormal



Sign 8-Bit Exponent 23-Bit Mantissa

$(-1)^{sign} \times 2^{exponent-127} \times (1 + mantissa)$

(Normal Numbers, Exponent ≠ 0)

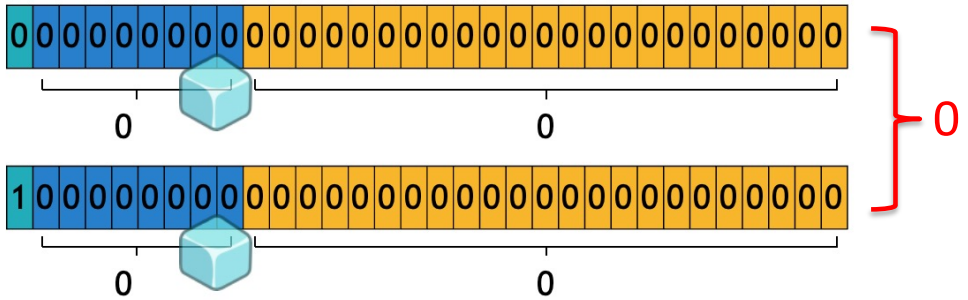


$0.265625 = 2^{-2} \times 1.0625 = 2^{125-127} \times (1 + 0.0625)$

Should have been $(-1)^{sign} \times 2^{0-127} \times (1 + mantissa)$

But we force to be $(-1)^{sign} \times 2^{1-127} \times mantissa$

(Subnormal Numbers, Exponent = 0)



$0 = (-1)^{sign} \, 2^{1-127} \times 0 = (-1)^{sign} \, 2^{-126} \times 0$

https://hanlab.mit.edu/courses/2023-fall-65940

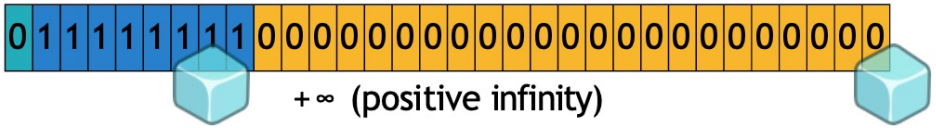
Positive and Negative Infinities Representation

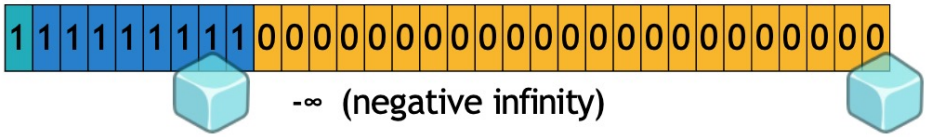
Sign 8-Bit Exponent **23-Bit Mantissa**

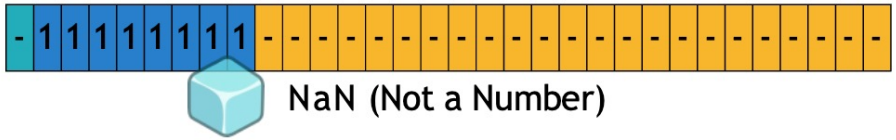
$(-1)^{sign} \times 2^{exponent-127} \times (1 + mantissa)$

(Normal Numbers, Exponent ≠ 0)

$Exponent = 256_{10} = 11111111_2$







much waste. Revisit in fp8.

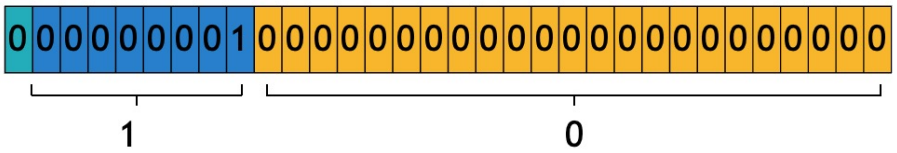
https://hanlab.mit.edu/courses/2023-fall-65940

Minimum Positive Values

Sign 8-Bit Exponent **23-Bit Mantissa**

$(-1)^{sign} \times 2^{exponent-127} \times (1 + mantissa)$

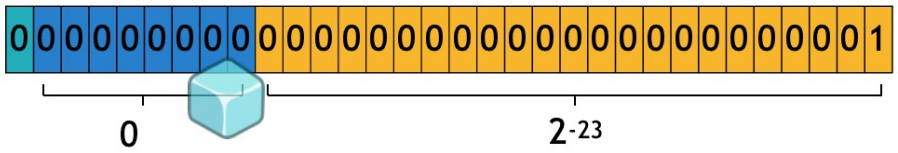
(Normal Numbers, Exponent ≠ 0)



$2^{-126} = 2^{1-127} \times (1 + 0)$

$(-1)^{sign} \times 2^{1-127} \times mantissa$

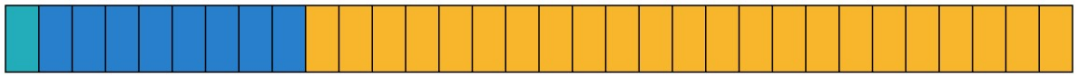
(Subnormal Numbers, Exponent = 0)



$2^{-149} = 2^{1-127} \times 2^{-23}$

https://hanlab.mit.edu/courses/2023-fall-65940

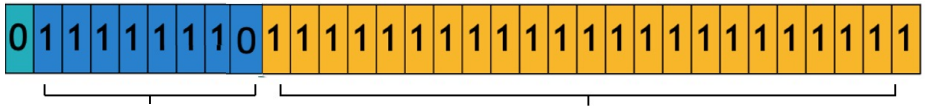
Maximum Positive Values



Sign 8-Bit Exponent **23-Bit Mantissa**

$(-1)^{sign} \times 2^{exponent-127} \times (1 + mantissa)$

(Normal Numbers, Exponent ≠ 0)

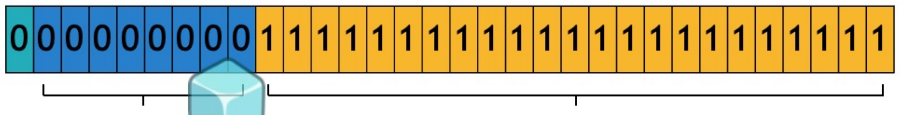


254 $2^{-23} + 2^{-22} + \ldots + 2^{-23} = 1 - 2^{-23}$

$2^{127} \times (1 + 1 - 2^{-23})$

$(-1)^{sign} \times 2^{1-127} \times mantissa$

(Subnormal Numbers, Exponent = 0)

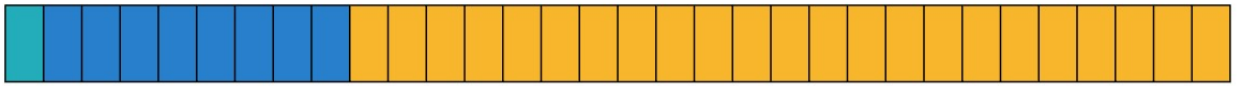


0 $2^{-23} + 2^{-22} + \ldots + 2^{-23} = 1 - 2^{-23}$

$2^{-126} \times (1 - 2^{-23})$

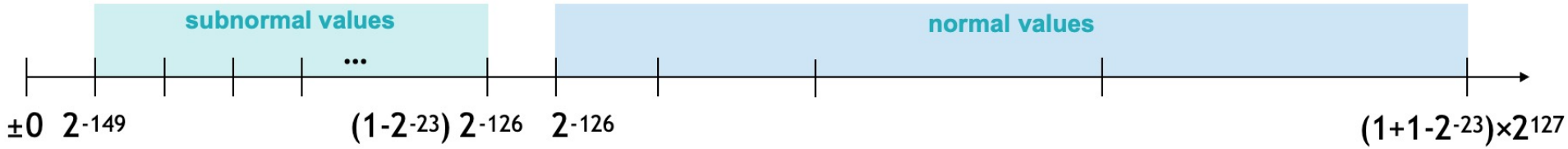
https://hanlab.mit.edu/courses/2023-fall-65940

Summary: FP32 of IEEE 754



Sign 8-Bit Exponent 23-Bit Mantissa

Exponent	Fraction=0	Fraction≠0	Equation
$00_H = 0$	±0	subnormal	$(-1)^{sign} \times 2^{1-127} \times mantissa$
$01_H \ldots FE_H = 1 \ldots 254$	normal		$(-1)^{sign} \times 2^{exponent-127} \times (1 + mantissa)$
$FF_H = 255$	±INF	NaN	



https://hanlab.mit.edu/courses/2023-fall-65940

Floating-Point Number Formats

- **IEEE 754** Single Precision 32-bit Float (IEEE FP32)

- **IEEE 754** Half Precision 16-bit Float (IEEE FP16)

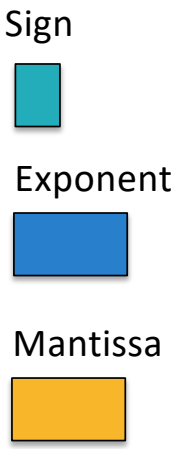
- **Google** Brain Float (BF16)

- **Nvidia** FP8 (E4M3)

  - FP8 E4M3 does not have INF, and $S.1111.111_2$ is used for NaN.
  - Largest FP8 E4M3 normal value is $S.1111.110_2 = 448$.

- **Nvidia** FP8 (E5M2) for gradient in the backward

  - FP8 E5M2 does not have INF ($S.11111.00_2$), and NaN ($S.11111.XX_2$).
  - Largest FP8 E5M2 normal value is $S.11110.11_2 = 57344$.



INT4 and FP4

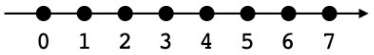
INT4

S			
0	0	0	1
0	1	1	1

=1

=7

-1,-2,-3,-4,-5,-6,-7,-8
0, 1, 2, 3, 4, 5, 6, 7



-1,-2,-3,-4,-5,-6,-7,-8
0, 1, 2, 3, 4, 5, 6, 7

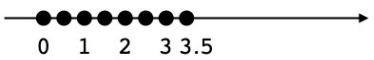
FP4 (E1M2)

S	E	M	M
0	0	0	1
0	1	1	1

$=0.25\times2^{1-0}=0.5$

$=(1+0.75)\times2^{1-0}=3.5$

-0,-0.5,-1,-1.5,-2,-2.5,-3,-3.5
0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5



-0,-1,-2,-3,-4,-5,-6,-7
0, 1, 2, 3, 4, 5, 6, 7 ×0.5

FP4 (E2M1)

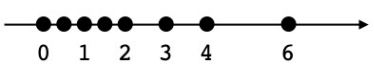
S	E	E	M
0	0	0	1
0	1	1	1

$=0.5\times2^{1-1}=0.5$

$=(1+0.5)\times2^{3-1}=1$

no inf, no NaN

-0,-0.5,-1,-1.5,-2,-3,-4,-6
0, 0.5, 1, 1.5, 2, 3, 4, 6



-0,-1,-2,-3,-4,-6,-8,-12
0, 1, 2, 3, 4, 6, 8, 12 ×0.5

FP4 (E3M0)

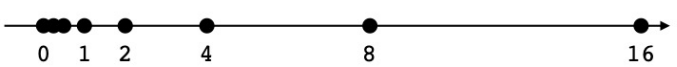
S	E	E	E
0	0	0	1
0	1	1	1

$=(1+0)\times2^{1-3}=0.25$

$=(1+0)\times2^{7-3}=16$

no inf, no NaN

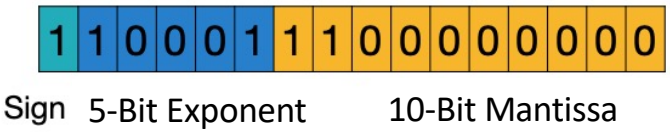
-0,-0.25,-0.5,-1,-2,-4,-8,-16
0, 0.25, 0.5, 1, 2, 4, 8, 16



-0,-1,-2,-4,-8,-16,-32,-64
0, 1, 2, 4, 8, 16, 32, 64 ×0.25

FP16 Example

- What is the following IEEE half precision (IEEE FP16) number in decimal?



Exponent Bias = 15_{10}

$$(-1)^{sign} \times 2^{Exponent-15} \times (1 + Mantissa)$$

  - Sign: $(-1)^1 = -1$
  - Exponent: $10001_2 = (2_{10})^4 + (2_{10})^0 = 16_{10} + 1_{10} = 17_{10}$
  - Mantissa: $1100000000_2 = (2_{10})^{-1} + (2_{10})^{-2} = 0.75_{10}$
  - **Decimal Answer** = $(-1) \times 2^{17-15} \times (1 + 0.75) = 2^2 \times -1.75 = -7.0_{10}$

BF16 (16-Bit Brain Float) Example

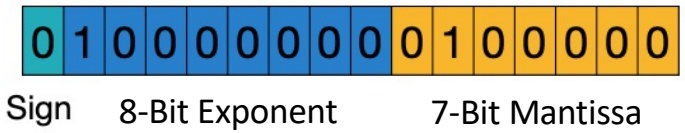
- What is the decimal 2.5 in **BF**16?

$$2.5_{10} = 2^{1} \times 1.25_{10}$$

Exponent Bias = 127_{10}

$$(-1)^{sign} \times 2^{Exponent-127} \times (1 + Mantissa) = (-1)^{0} \times 2^{128-127} \times (1 + 0.25)$$

  - Sign: $1 = (-1)^{0}$
  - Exponent Binary: $1_{10} + 127_{10} = 128_{10} = 10000000_{2}$
  - Mantissa Binary: $0.25_{10} = 2^{-2}{}_{10} = 0100000_{2}$
  - **Binary Answer**



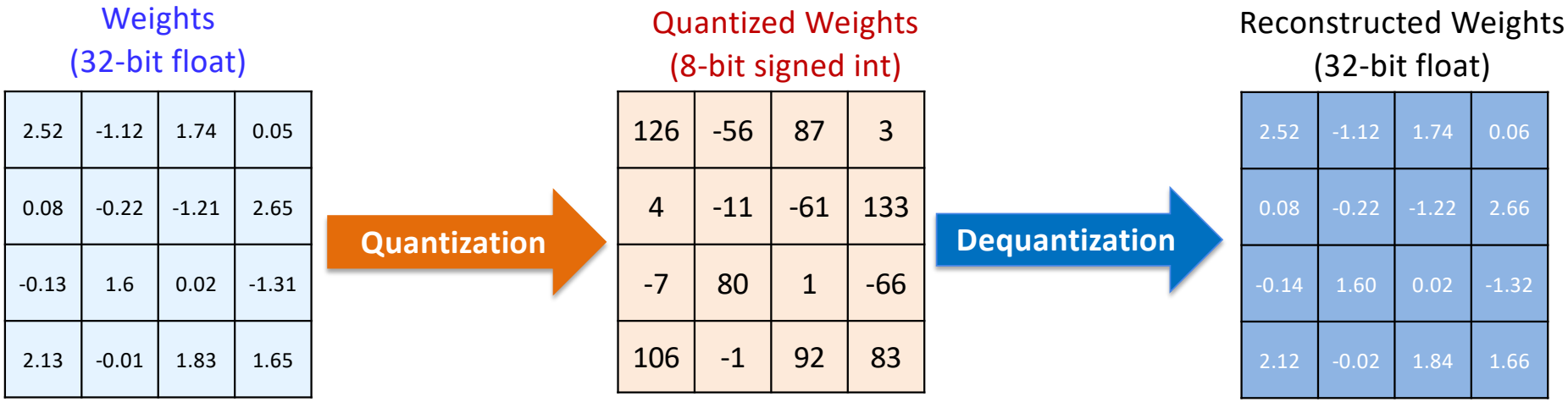
The Limitations of Traditional LLMs

- Traditional LLMs are typically designed as either 32-bit or 16-bit models, where each parameter is represented by a 32-bit or 16-bit floating-point number.
- **While this precision is beneficial, it comes at a significant cost**: **substantial computational power and memory are required, making LLMs resource-intensive and less accessible to a wider range of users.**
- As LLMs have grown in size, reaching more than 175B parameters, **quantization has emerged as a key approach to achieve lower precision representations**.

Quantization of LLMs

What is Quantization?

- **Quantization** is a model compression technique that reduces the precision of weights and activations in large language models.
- It converts high-precision data to lower-precision data, such as from 32-bit floating-point numbers to 8-bit integers.
- This reduction in bits leads to a significant decrease in model size, making LLMs more memory-efficient, requiring less storage space, and consuming less energy.



Quantization: Making LLMs Lighter

- **Large language models (LLMs) are memory hungry.** Model Quantization is to convert high-precision floating-point numbers in neural networks to low-precision representations.
- This process, essentially a function mapping, offers several advantages:
  1. **Smaller model size**
  2. **Reduced memory usage** during inference
  3. **Faster inference** on low-precision optimized processors

Model	Original Size (FP32)	Quantized Size (INT4)
LLaMA3.1-8B	3.84 GB	4.8 GB
LLaMA3.1-70B	336 GB	42 GB
LLaMA3.1-405B	1944 GB	243 GB

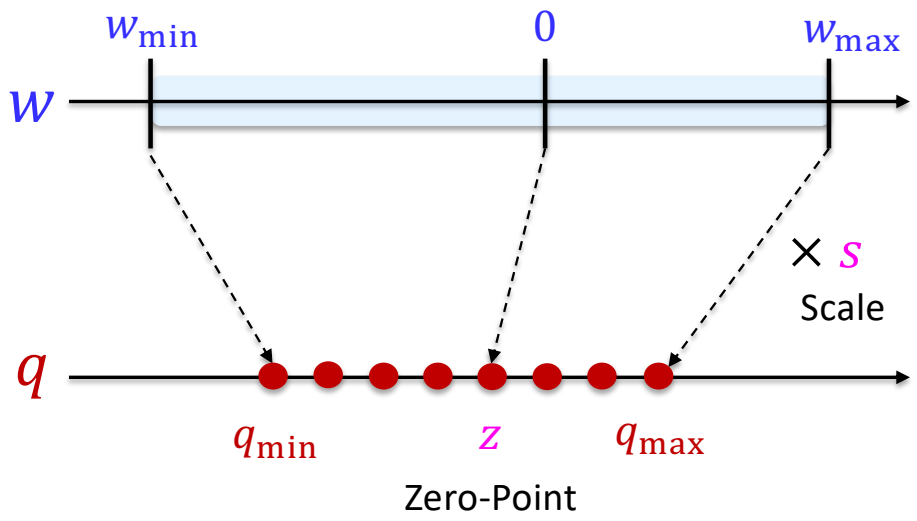
To estimate VRAM usage: **Parameters × (Precision / 8) × 1.2**. LLaMA3.1 models require: 8B (moderate), 70B (more), and 405B (substantial) capacity.
https://medium.com/@lmpo/a-guide-to-estimating-vram-for-llms-637a7568d0ea

Linear Quantization

- Linear quantization is a common method for compressing LLMs by mapping floating-point weights to fixed-point values.
- The process involves four steps:
  1. Determine the range of values for each tensor.
  2. Calculate scale and zero-point to fit values into the target integer range.
  3. Quantize floating-point values to lower-precision integers.
  4. Dequantize during inference for precise calculations.
- This technique significantly reduces model size while maintaining computational accuracy.

Asymmetric Linear Quantization

- **Linear quantization** assigns real values w to a fixed set of quantized values q, dependent on the range and number of bits, using an **affine mapping** of integers to real numbers.
- The mapping from integers to real numbers is linear, meaning it can be represented by a straight line.



Scale Factor: $s = \dfrac{q_{\max} - q_{\min}}{w_{\max} - w_{\min}}$

Zero-Point: $z = q_{\min} - \text{round}(s \cdot w_{\min})$

Quantized Value: $q = \text{round}(s \cdot w + z)$

Dequantized Value: $\hat{w} = \dfrac{q - z}{s}$

Linear Quantization Process

1. Calculate Minimum and Maximum Values:

$$[w_{\min} = \min(w),\ w_{\max} = \max(w)]$$

$$[q_{\min} = -2^{n-1}, q_{\max} = 2^{n-1} - 1]$$

8-bit Quantization

2. Compute Scale and Zero-Point:

Scale Factor: $s = \dfrac{q_{\max} - q_{\min}}{w_{\max} - w_{\min}}$

Zero-Point: $z = q_{\min} - \text{round}(s \cdot w_{\min})$

3. Quantization:

$$q = \text{round}(s \cdot w + z)$$

4. Dequantization:

$$\hat{w} = \frac{q - z}{s}$$

Quantization Parameters

- $w = (q - z)/s$
- **Scaling Factor:** s
  - Floating point number for scaling to a common range between $w_{\min}$ and $w_{\max}$

$$\left.\begin{aligned} w_{\max} &= (q_{\max} - z)/s \\ w_{\min} &= (q_{\min} - z)/s \\ w_{\max} - w_{\min} &= (q_{\max} - q_{\min})/s \end{aligned}\right\} \quad s = \frac{q_{\max} - q_{\min}}{w_{\max} - w_{\min}} = \frac{2^n - 1}{w_{\max} - w_{\min}}$$

n-bit Quantization

- **Zero-Point:** z
  - Integer number number for allow real number $x = 0$ be exactly representable by a quantized integer z

$$\left.\begin{aligned} w_{\min} &= (q_{\min} - z)/s \\ z &= q_{\min} - s \cdot w_{\min} \end{aligned}\right\}$$

Dealing with rounding:
This means that the quantities of $s \cdot w_{\min}$ must be integer numbers:

$$z = q_{\min} - \text{round}(s \cdot w_{\min})$$

Example: 2-Bit Asymmetric Linear Quantization

2.52	-1.12	1.74	0.05
0.08	-0.22	-1.21	**2.65**
-0.13	1.6	0.02	**-1.31**
2.13	-0.01	1.83	1.65

$w_{\max} = 2.65$

$w_{\min} = -1.31$

$$s = \frac{q_{\max} - q_{\min}}{w_{\max} - w_{\min}} = \frac{1 - (-2)}{2.65 - (-1.31)} = 0.76$$

$$z = q_{\min} - \text{round}(s \cdot w_{\min})$$
$$= -2 - \text{round}\big(0.76(-1.31)\big) = -1$$

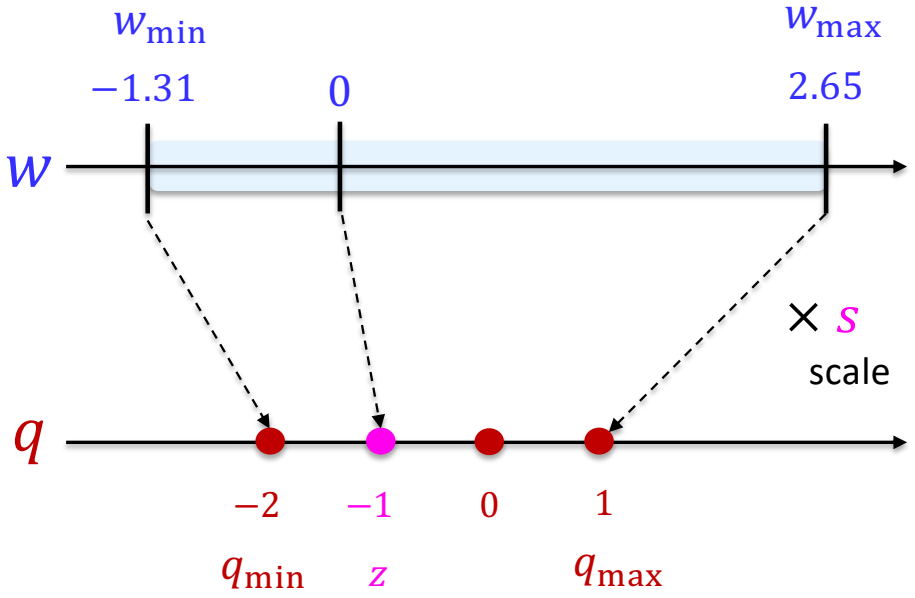
2-bit Quantization

Binary	Decimal
01	1
00	0
11	-1
10	-2

$q_{\max} = 1$

$q_{\min} = -2$

(2-bit signed int)



2-Bit Linear Quantization Example

Weights
(32-bit float)

2.52	-1.12	1.74	0.05
0.08	-0.22	-1.21	2.65
-0.13	1.6	0.02	-1.31
2.13	-0.01	1.83	1.65

Calibration Data

$s = 0.76 \qquad z = -1$

$$q = \text{round}(s \cdot w + z)$$

2-bit Quantization

$$q = \text{round}(0.76 \cdot w - 1)$$

Quantized Weights
(2-bit signed int)

1	**-2**	**0**	**-1**
-1	**-1**	**-2**	**1**
-1	**0**	**-1**	**-2**
1	**-1**	**0**	**0**

2-Bit Linear Dequantization Example

Quantized Weights
(2-bit signed int)

1	-2	0	-1
-1	-1	-2	1
-1	0	-1	-2
1	-1	0	0

$s = 0.76$ $z = -1$

$$\hat{w} = \frac{q - z}{s}$$

2-bit Dequantization

$$\hat{w} = \frac{q - (-1)}{0.76}$$

Reconstructed Weights
(32-bit float)

2.64	-1.32	1.32	0.0
0.0	0.0	-1.32	2.64
0.0	1.32	0.02	-1.32
2.13	0.0	1.32	1.32

Quantization Error

Weights
(32-bit float)

2.52	-1.12	1.74	0.05
0.08	-0.22	-1.21	2.65
-0.13	1.6	0.02	-1.31
2.13	-0.01	1.83	1.65

$q = \text{round}(s \cdot w + z)$

Quantization

$q = \text{round}(0.76 \cdot w + 1)$

Quantized Weights
(2-bit signed int)

1	**-2**	**0**	**-1**
-1	**-1**	**-2**	**1**
-1	**0**	**-1**	**-2**
1	**-1**	**0**	**0**

$\hat{w} = \frac{q - z}{s}$

Dequantization

$\hat{w} = \frac{q - (-1)}{0.76}$

Reconstructed Weights
(32-bit float)

2.64	-1.32	1.32	0.0
0.0	0.0	-1.32	2.64
0.0	1.32	0.02	-1.32
2.64	0.0	1.32	1.32

$$s = \frac{q_{\max} - q_{\min}}{w_{\max} - w_{\min}} = 0.76$$

$$z = q_{\min} - \text{round}(s \cdot w_{\min}) = -1$$

Dequantization Affine Map

Binary	Decimal	Dequantized value
01	1	2.64
00	0	1.32
11	-1	0.0
10	-2	-1.32

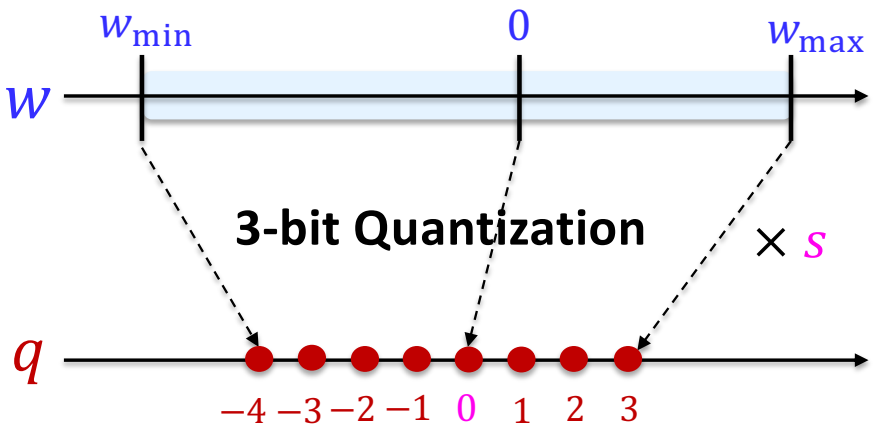
(2-bit signed int)

Quantization Error

-0.12	0.2	0.41	0.0
0.08	-0.22	0.11	-0.01
-0.13	0.28	0.02	0.01
-0.51	-0.01	0.51	0.33

$(w - \hat{w})$

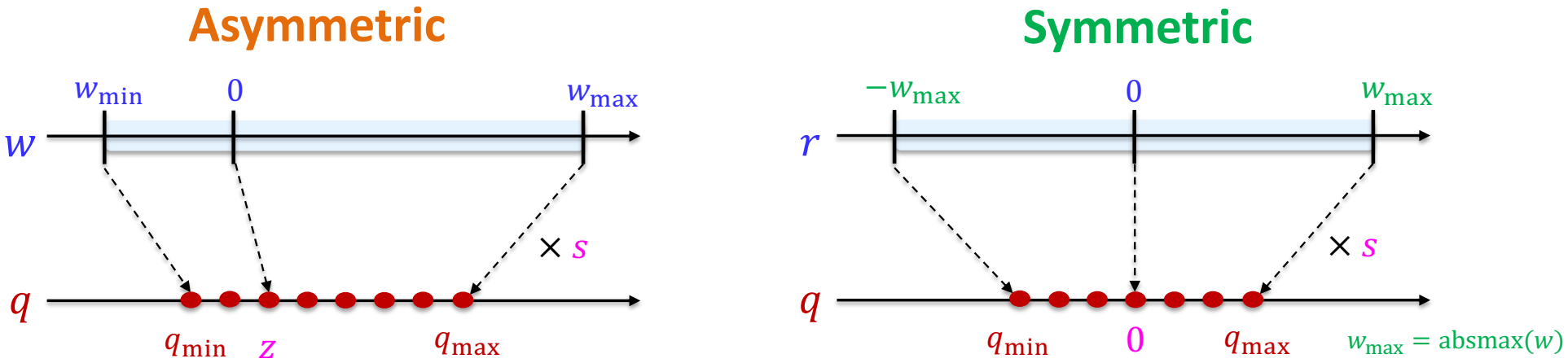
More Linear Quantization Examples



Bit Width	q_{min}	q_{max}
2	-2	1
3	-4	3
4	-8	7
N	-2^{N-1}	$2^{N-1}-1$

Asymmetric vs Symmetric Quantization

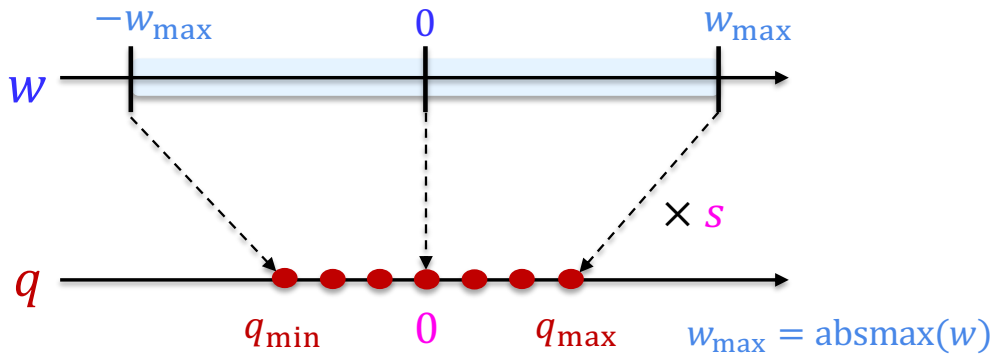
- **In asymmetric linear quantization**, the quantization intervals are **not symmetric around zero**. This means that the positive and negative values are represented with different numbers of bits, and the zero point is not at the midpoint of the range.



- **Asymmetric quantization allocates bits based on the signal's distribution, assigning more bits to the values that need them most.**

Asymmetric vs Symmetric Quantization

- **In asymmetric linear quantization**, the quantization intervals are **not symmetric around zero**. This means that the positive and negative values are represented with different numbers of bits, and the zero point is not at the midpoint of the range.



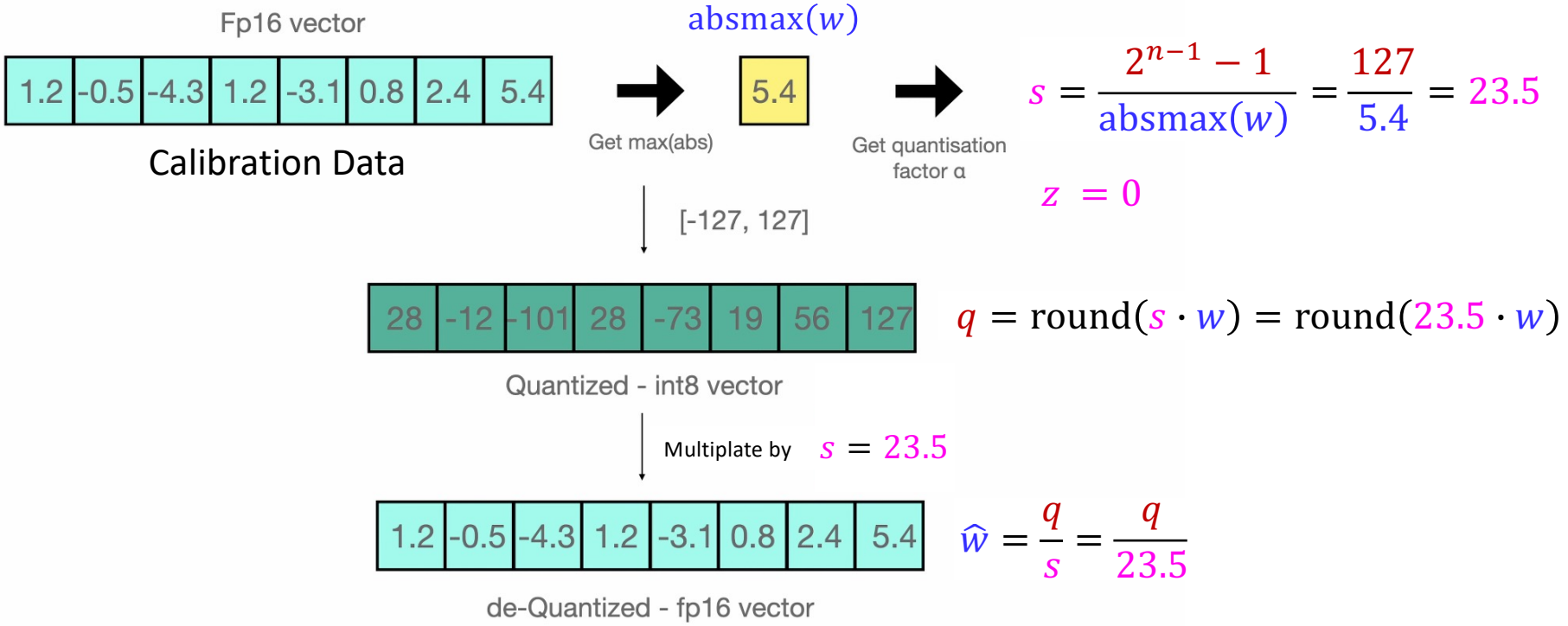
$$s = \frac{127}{\text{absmax}(w)} \qquad q = \text{round}(s \cdot w) \qquad \hat{w} = \frac{q}{s}$$

Symmetric Quantization

- To achieve symmetric quantization with *n*-bit, we usually sacrifice -2^{n-1} which allows to map a series of floating-point numbers in the range $[-\text{absmax}(w), \text{absmax}(w)]$ into another in the range $[-(2^{n-1}-1), 2^{n-1}-1]$.
  - For example, by using 8 bits, we can represent floating-point numbers in the range $[-127,127]$.
- Since the zero-point value is always equal to zero, the scale factor, quantization, and dequantization equations can be simplified.

$$s = \frac{2^{n-1}-1}{\text{absmax}(w)} \qquad q = \text{round}(s \cdot w) \qquad \widehat{w} = \frac{q}{s}$$

Calibration of Symmetric Linear Quantization



$$s = \frac{2^{n-1} - 1}{\text{absmax}(w)} = \frac{127}{5.4} = 23.5$$

$$z = 0$$

$$q = \text{round}(s \cdot w) = \text{round}(23.5 \cdot w)$$

$$\hat{w} = \frac{q}{s} = \frac{q}{23.5}$$

Example: 8-Bit Symmetric Linear Quantization

Weights
(32-bit float)

2.52	-1.12	1.74	0.05
0.08	-0.22	-1.21	2.65
-0.13	1.60	0.02	-1.31
2.13	-0.01	1.83	1.65

Quantization

Quantized Weights
(8-bit signed int)

121	-54	83	2
4	-11	-58	127
-6	77	1	-63
102	0	88	79

$[-127, 127]$

$n = 8$

We usually sacrifice -128 to obtain a symmetric range

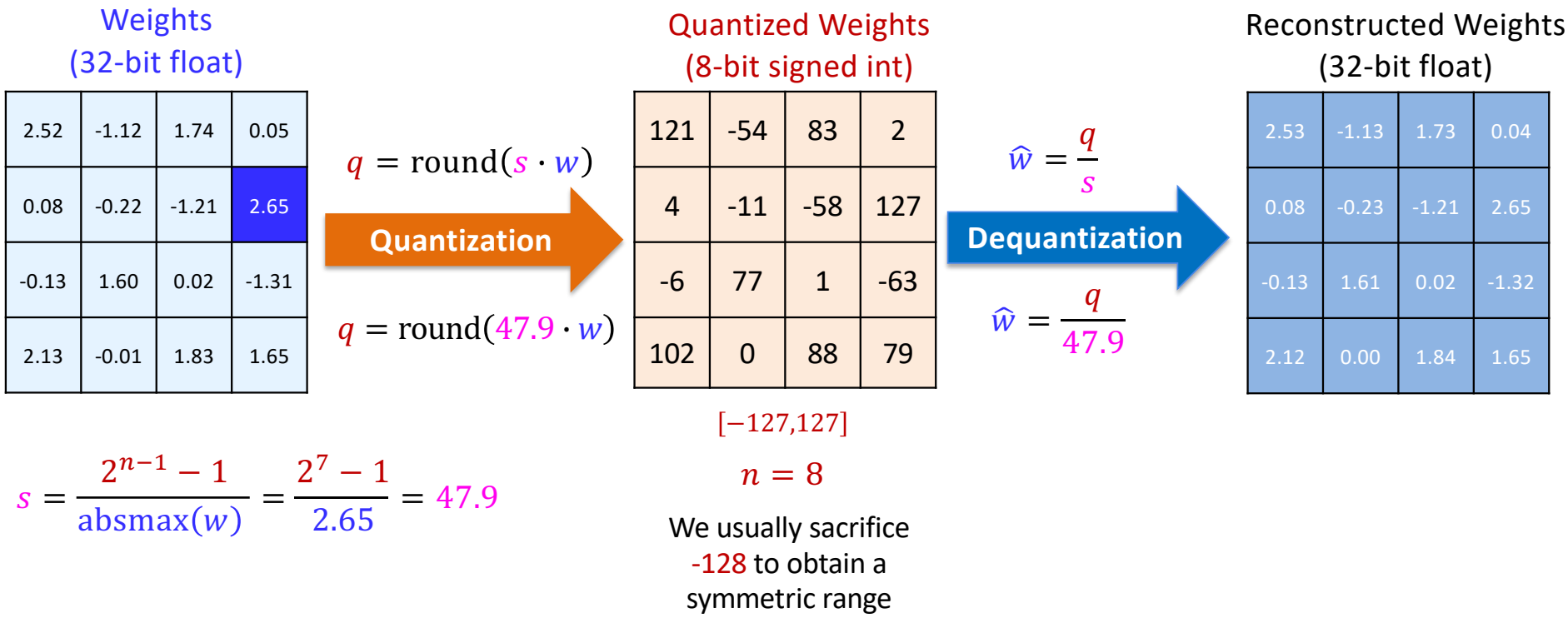
Dequantization

Reconstructed Weights
(32-bit float)

2.53	-1.13	1.73	0.04
0.08	-0.23	-1.21	2.65
-0.13	1.61	0.02	-1.32
2.12	0.00	1.84	1.65

$$s = \frac{2^{n-1} - 1}{\text{absmax}(w)} = \frac{2^7 - 1}{2.65} = 47.9$$

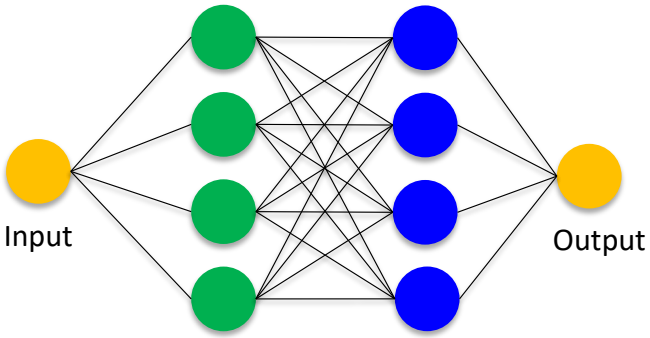
Example: 8-Bit Symmetric Linear Quantization



$$s = \frac{2^{n-1} - 1}{\text{absmax}(w)} = \frac{2^7 - 1}{2.65} = 47.9$$

Applying Linear Quantization

- Linear quantization assigns weights to a fixed set of quantized values, dependent on the range and number of bits, using an affine mapping of integers to real numbers.
- The mapping from integers to real numbers is linear, meaning it can be represented by a straight line.



Dequantization → **Output** → $\mathbf{y = Wx + b}$ ← **Perform all operations using integer arithmetic**

The main benefit is that integer operations are much faster in most hardware than floating point operation

Weight **Input** **Bias**

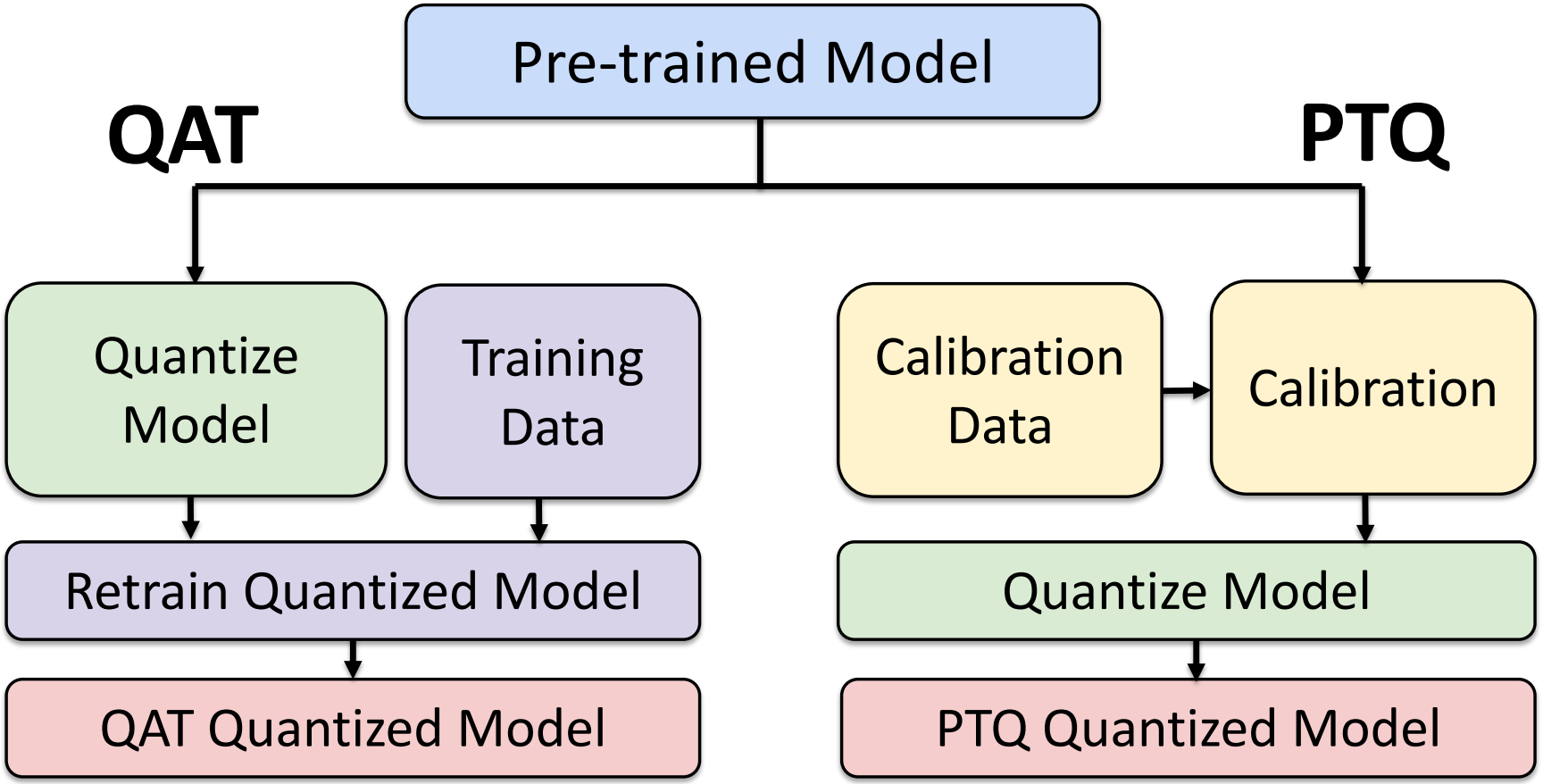
Usually quantized as int32

Quantize

Types of LLM Quantization

- It’s possible to divide the techniques of obtaining quantized models into two:
  1. **Post-Training Quantization (PTQ)**: converting the weights of an already trained model to a lower precision without any retraining. Though straightforward and easy to implement, PTQ might degrade the model’s performance slightly due to the loss of precision in the value of the weights.
  2. **Quantization-Aware Training (QAT)**: Unlike PTQ, QAT integrates the weight conversion process during the training stage. This often results in superior model performance, but it’s more computationally demanding. A highly used QAT technique is the QLoRA.

QAT vs PTQ


Pre-trained Model
QAT
PTQ
Quantize Model
Training Data
Calibration Data
Calibration
Retrain Quantized Model
Quantize Model
QAT Quantized Model
PTQ Quantized Model

Quantization-Aware Training (QAT)

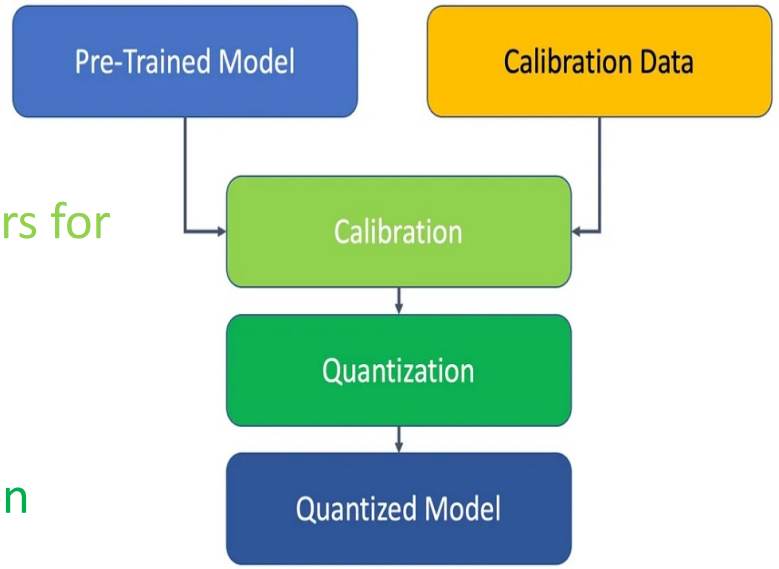
- QAT is a technique that incorporates weight conversion into the training process of LLMs.
- By simulating lower precision during training, models can learn to adapt to quantization noise, resulting in improved performance.
- Although QAT delivers superior results, especially at lower precision levels, its widespread adoption is hindered by **the need for retraining the model**.
- Furthermore, **QAT is computationally intensive and costly, particularly for large models.**
- Nevertheless, its benefits make it a crucial approach for optimizing LLMs and enabling efficient deployment on devices with limited resources.

PTQ (Post-Training Quantization)

- **PTQ don’t need to retrain the model** while quantization is directly applied to pre-trained models.
- **It’s simpler and faster to implement, making it ideal for rapid deployment.**
- PTQ analyzes weight distributions, determines quantization parameters, and converts weights to lower precision.
- While quick and resource-efficient, it may slightly degrade model performance, especially in smaller models or with outlier values.

Post-Training Quantization (PTQ)

- PTQ optimizes pre-trained deep models for deployment by reducing computational and memory requirements.
  - **Applied to pre-trained model without retraining.**
  - Uses representative calibration data.
  - Analyzes activation distributions to compute scale factors for each tensor.
  - Examines weight distributions to scale factors and zero-points as quantization parameters.
  - Typically converts 32-bit floating-point to lower precision formats (e.g. 8-bit integers)
  - Balances model size reduction with accuracy maintenance.
  - Requires post-quantization validation



Specific Quantization Techniques for LLMs

- LLMs have benefited from several specialized quantization techniques, each offering unique advantages for model compression and efficient deployment.
- Among these, **GPTQ**, **GGUF**, **NF4**, and **BitsandBytes** library stand out as particularly effective solutions.

GPTQ (Generalized Post-Training Quantization)

- **GPTQ is a Post-Training Quantization (PTQ) technique** for compressing LLMs by reducing model precision to 3–4 bits per weight while maintaining performance.
- GPTQ is **designed for GPU**, which is more efficient to deploy and run on GPU platforms.
- GPTQ achieves this through two main features:
  1. **Layerwise Quantization**: Quantizes each layer individually to minimize the difference between original and simplified weights, aiming for the lowest mean squared error to preserve accuracy.
  2. **Optimal Brain Quantization**: Adjusts remaining weights to compensate for errors introduced during quantization, enhancing model reliability and efficiency.

GPTQ

- The GPTQ technique efficiently quantizes large models like GPT-3 in hours on a GPU, achieving a 2x higher compression ratio than baseline methods.
- Key optimizations include reducing memory bandwidth and using quantization order heuristics to minimize errors.

- The open-source AutoGPTQ tool (MIT license) allows quantization to various bit widths (8-bit, 4-bit, 3-bit, 2-bit).
- For example, quantizing the LLaMA-3–8B model to 4-bit reduces its size from 16.07 GB to 5.74 GB, making it more efficient to deploy and run.

GPTQ Example

```
from transformers import AutoModelForCausalLM, AutoTokenizer
from optimum.gptq import GPTQQuantizer
import torch
model_path = 'meta-llama/Meta-Llama-3-8B'
b = 4
quant_path = 'Meta-Llama-3-8B-gptq-'+str(b)+'bit'

tokenizer = AutoTokenizer.from_pretrained(model_path, use_fast=True)
model = AutoModelForCausalLM.from_pretrained(model_path, torch_dtype=torch.float16, device_map="auto")

quantizer = GPTQQuantizer(bits=b, dataset="c4", model_seqlen = 2048)
quantized_model = quantizer.quantize_model(model, tokenizer)

quantized_model.save_pretrained("./"+quant_path, safetensors=True)
tokenizer.save_pretrained("./"+quant_path)
```

GGUF (GGML Unified Format)

- The GGUF release, led by Georgi Gerganov and the llama.cpp team, introduced a significant milestone in LLM development by enabling **efficient execution of LLMs on consumer-grade CPUs**.
- **Advantages:**
  - **CPU and Apple M series compatibility**: GGUF is the preferred method for running LLMs like LLaMA and Mistral on CPU-based systems and Apple devices with M series chips.
  - **Wide support**: The GGUF file format is supported by popular frameworks such as llama.cpp and HuggingFace.
  - **Improved performance**: GGUF models show lower perplexity scores, indicating better language understanding and generation capabilities.

GGUF Example

```
# pip install ctransformers[cuda]

from ctransformers import AutoModelForCausalLM
from transformers import AutoTokenizer, pipeline

# Load LLM and Tokenizer
# Use `gpu_layers` to specify how many layers will be offloaded to the GPU.
model = AutoModelForCausalLM.from_pretrained(
"TheBloke/zephyr-7B-beta-GGUF",
model_file="zephyr-7b-beta.Q4_K_M.gguf",
model_type="mistral", gpu_layers=50, hf=True
)
tokenizer = AutoTokenizer.from_pretrained(
"HuggingFaceH4/zephyr-7b-beta", use_fast=True
)

# Create a pipeline
pipe = pipeline(model=model, tokenizer=tokenizer, task='text-generation')
```

GGUF vs GPTQ

- Basically, GGUF and GPTQ are two prominent quantization techniques for LLMs.
- Many versions of LLMs already quantized using GPTQ or GGUF on the Hugging Face Hub.

Feature	GGUF	GPTQ
Inference platform	CPU	GPU
Inference speed	Faster on CPUs	Faster on GPUs
Inference quality	Similar	Similar
Model size	Larger	Small
Compatibly	HF Transformers	HF Transformers

NF4 (Normalized Float 4)

- NF4 is a 4-bit quantization data type that balances compression and accuracy.
- It normalizes neural network weights to the [-1, 1] range and outperforms 4-bit integers and floats.
- When combined with Double-Quantization (DQ), NF4 achieves higher compression ratios (around 0.37 bits per parameter) without sacrificing performance, enabling significant memory savings.
- For example, a 65B model sees a 3 GB memory reduction. NF4 with DQ is used in QLoRA for efficient fine-tuning of LLMs, offering compression similar to INT4 but with better accuracy preservation

Bitsandbytes

- **Bitsandbytes** is a **quantization library** for LLMs that:
  - Compresses models to 8-bit or 4-bit precision
  - Reduces memory usage while maintaining performance
  - Uses normalization, quantization, and dequantization steps
  - Employs NF4 (Normalized Float 4) for 4-bit quantization
  - Stores weights in lower precision but computes in 16-bit
  - Integrates easily with Hugging Face's Transformers library
  - Enables running large models on consumer hardware
  - Supports fine-tuning with parameter-efficient methods
- It offers an accessible approach to compress LLMs for deployment on resource-constrained devices.

NF4 and Double Quantization

- NF4 and Double Quantization can be leveraged using the bitsandbytes library which is integrated inside the transformers library. Here is an example of **how to easily load and quantize any Hugging Face model**:

```
!pip install bitsandbytes

from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig
import torch

nf4_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_use_double_quant=True,
        bnb_4bit_compute_dtype=torch.bfloat16 )

model_name = "PY007/TinyLlama-1.1B-step-50K-105b"

tokenizer_nf4 = AutoTokenizer.from_pretrained(model_name, quantization_config=nf4_config)

model_nf4 = AutoModelForCausalLM.from_pretrained(model_name, quantization_config=nf4_config)
```

NF4 and Double Quantization

- NF4 and Double Quantization can be leveraged using the bitsandbytes library which is integrated inside the transformers library. Here is an example of **how to easily load and quantize any Hugging Face model**:

```
!pip install bitsandbytes

from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig
import torch

nf4_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_use_double_quant=True,
        bnb_4bit_compute_dtype=torch.bfloat16 )

model_name = "PY007/TinyLlama-1.1B-step-50K-105b"

tokenizer_nf4 = AutoTokenizer.from_pretrained(model_name, quantization_config=nf4_config)

model_nf4 = AutoModelForCausalLM.from_pretrained(model_name, quantization_config=nf4_config)
```

GPTQ vs AWQ vs GGUF vs NF4 vs BitsandBytes

- **GPTQ** is ideal for GPU environments, offering efficient post-training quantization with 4-bit precision.
- **AWQ** focuses on salient weights for better performance and faster inference, making it suitable for resource-constrained devices.
- **GGUF** is designed for CPU inference, allowing flexible model loading but generally slower performance.
- **NF4** aims for efficient compression but is less established.
- **BitsandBytes** provides easy implementation with Hugging Face models but has slower performance and longer load times.
- Choose based on your specific needs: GPU optimization, CPU compatibility, or ease of use in prototyping.