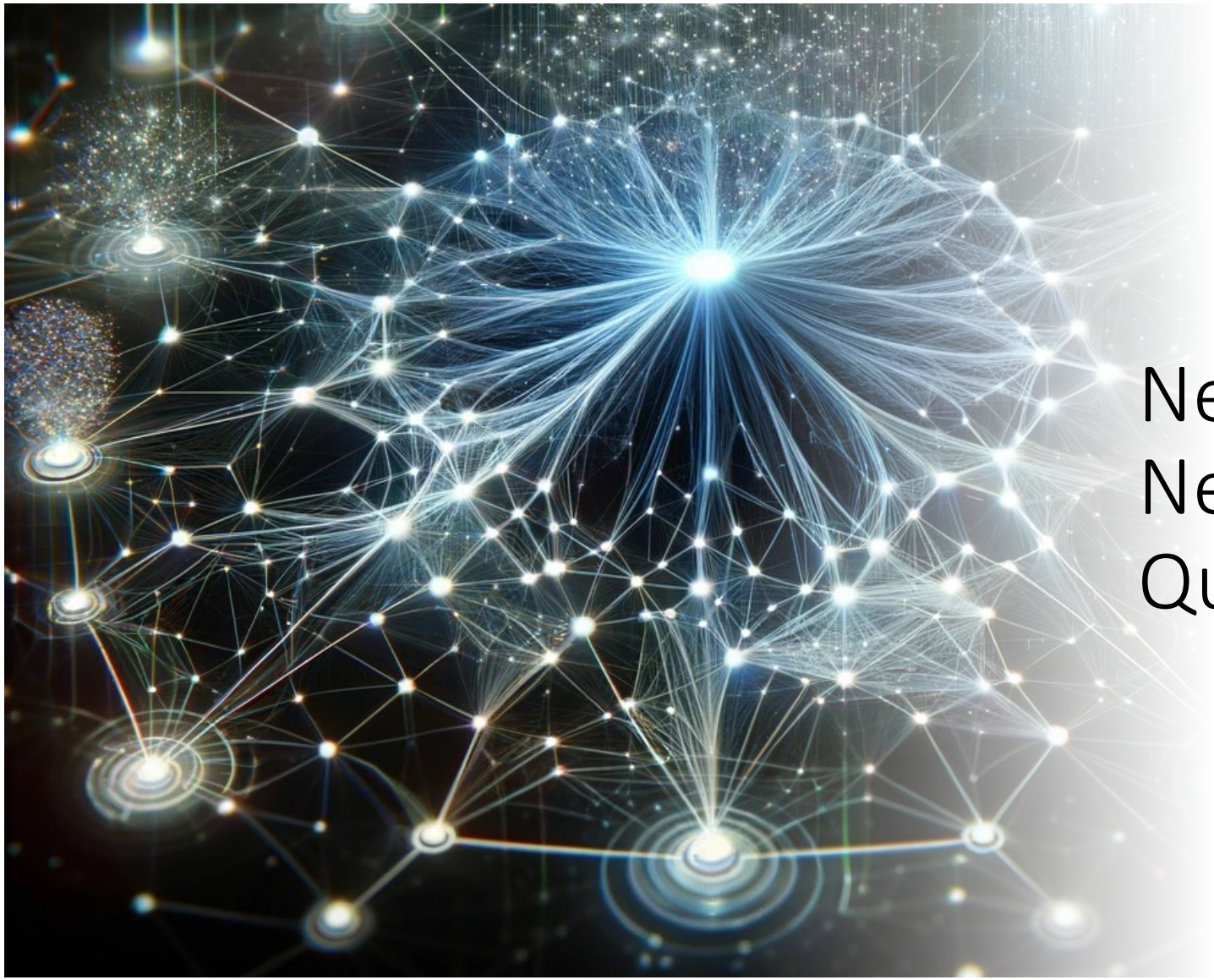
Introduction to High-Performance Machine Learning

Lecture 9 03/28/24

Dr. Kaoutar El Maghraoui

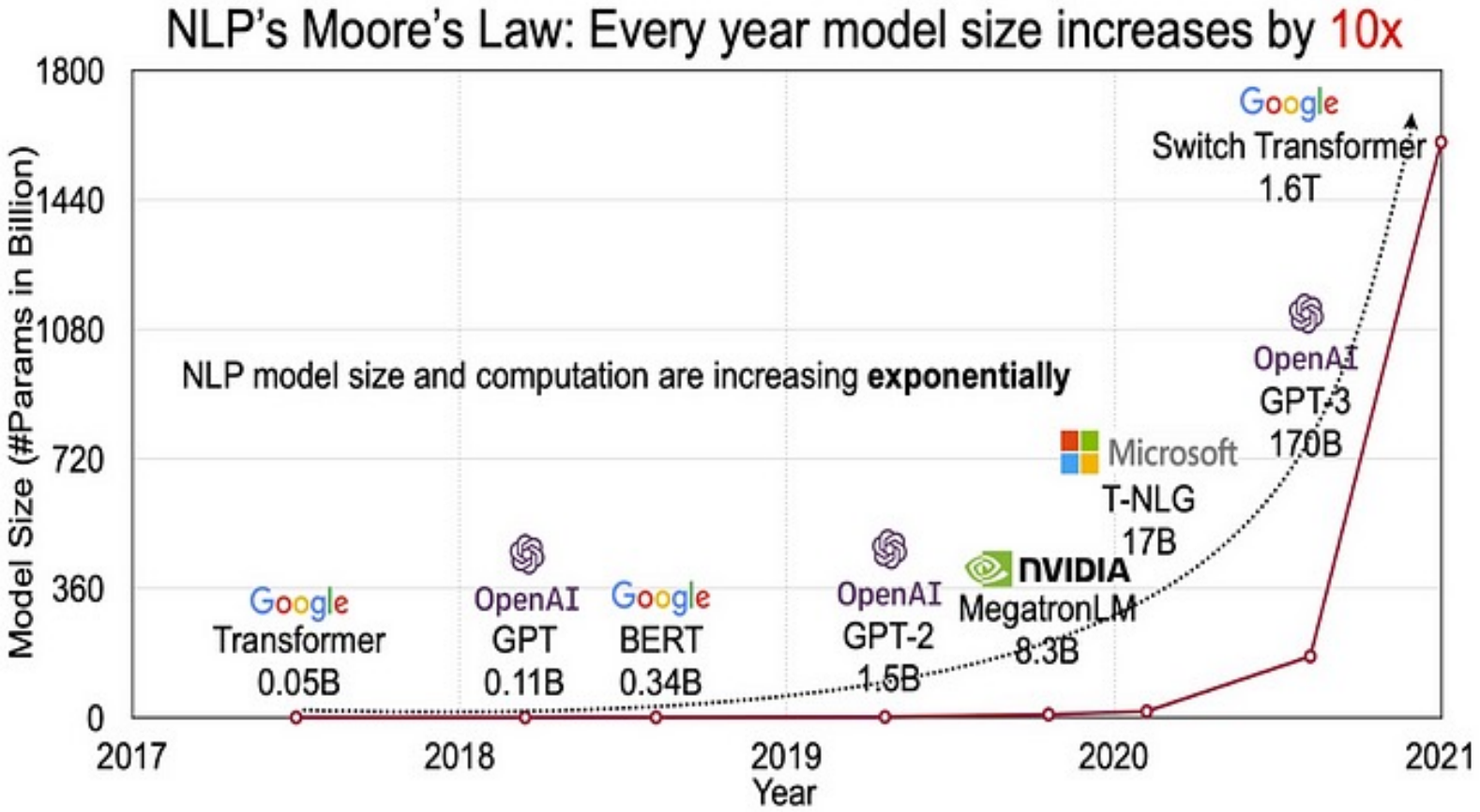
Dr. Parijat Dube

Final exam -
Determining bit-width; Mixed and varying precision; Quantization: post-training quantization, static vs dynamic quantization, quantization aware training, graph mode quantization, hardware aware quantization, calculating quantization error, quantization hyperparameters, symmetric vs non-symmetric quantization.



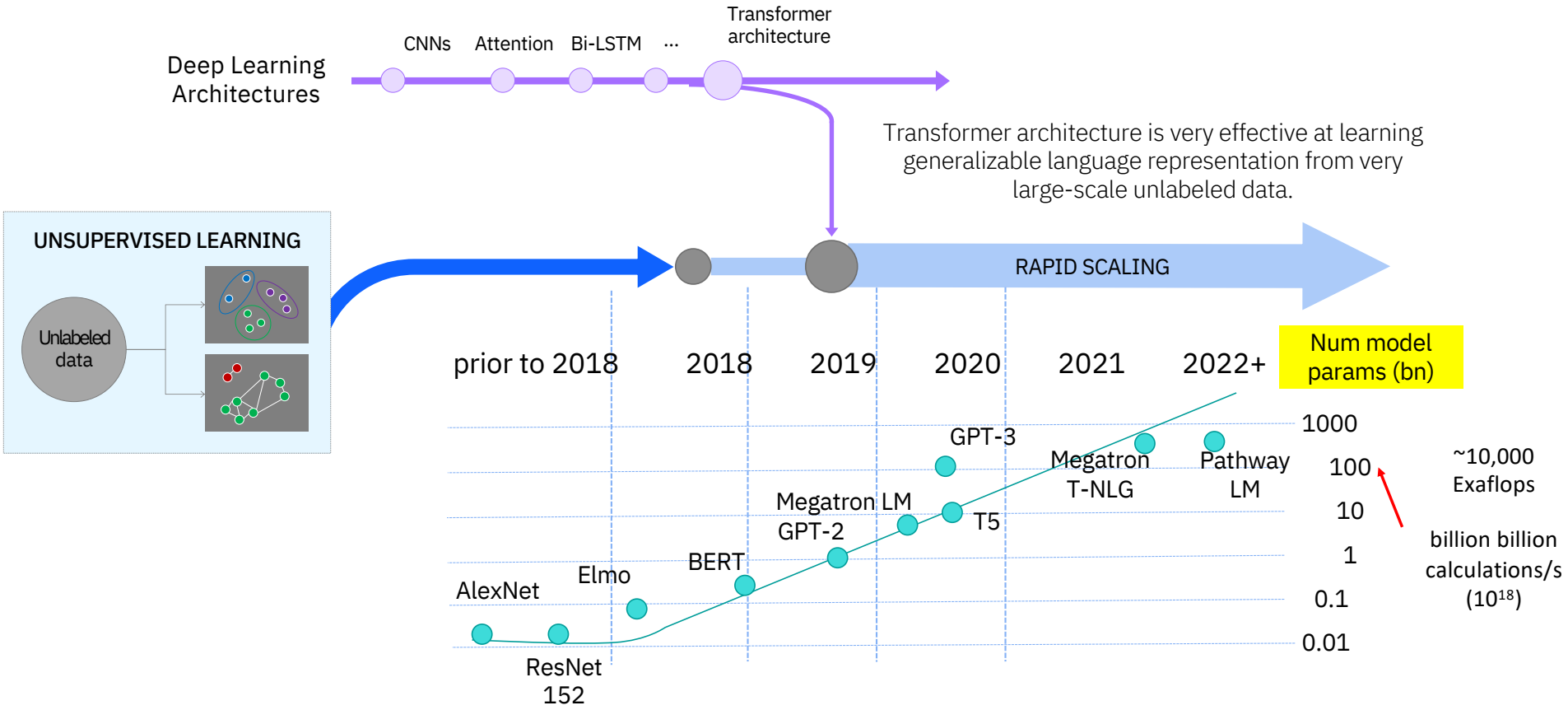
Neural Network Quantization

Today's AI is too BIG!



https://efficientml.ai

Foundation models are blowing up compute requirements



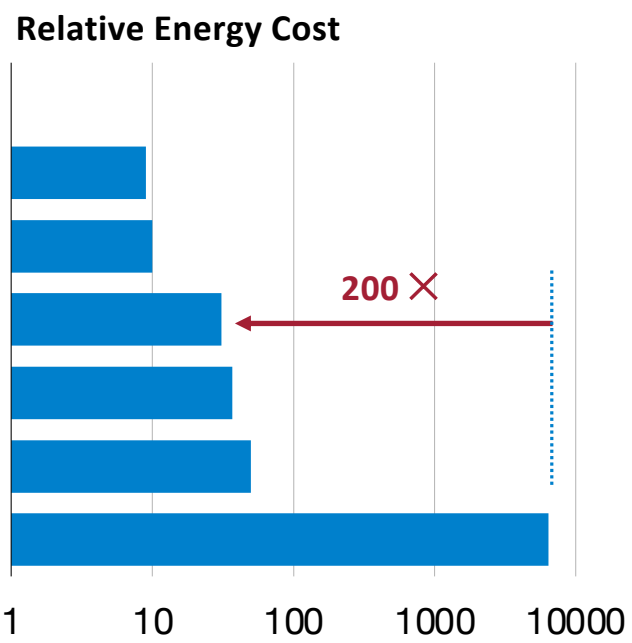
© 2023 IBM Corporation

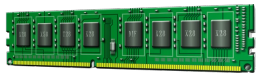
Memory is Expensive

Data Movement → More Memory Reference → More Energy

Operation	Energy [pJ]
32 bit int ADD	0.1
32 bit float ADD	0.9
32 bit Register File	1
32 bit int MULT	3.1
32 bit float MULT	3.7
32 bit SRAM Cache	5
32 bit DRAM Memory	640

Rough Energy Cost For Various Operations in 45nm 0.9V



1  = 200 ×+

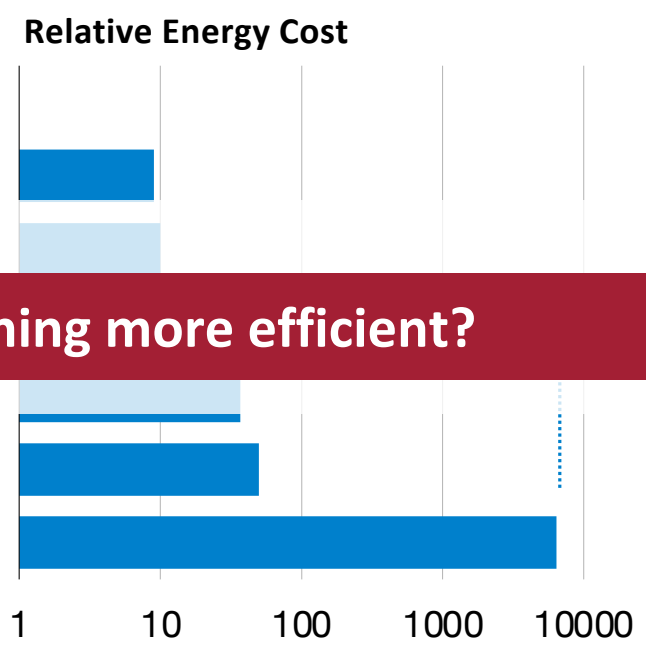
This image is in the public domain

Computing's Energy Problem (and What We Can Do About it) [Horowitz, M., IEEE ISSCC 2014]

Memory is Expensive

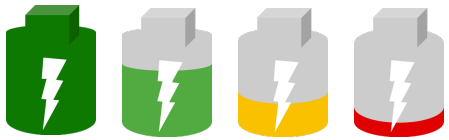
Data Movement → More Memory Reference → More Energy

Operation	Energy [pJ]
32 bit int ADD	0.1
32 bit float ADD	0.9
32 bit Register File	1
32 bit float MULT	3.7
32 bit SRAM Cache	5
32 bit DRAM Memory	640



How should we make deep learning more efficient?

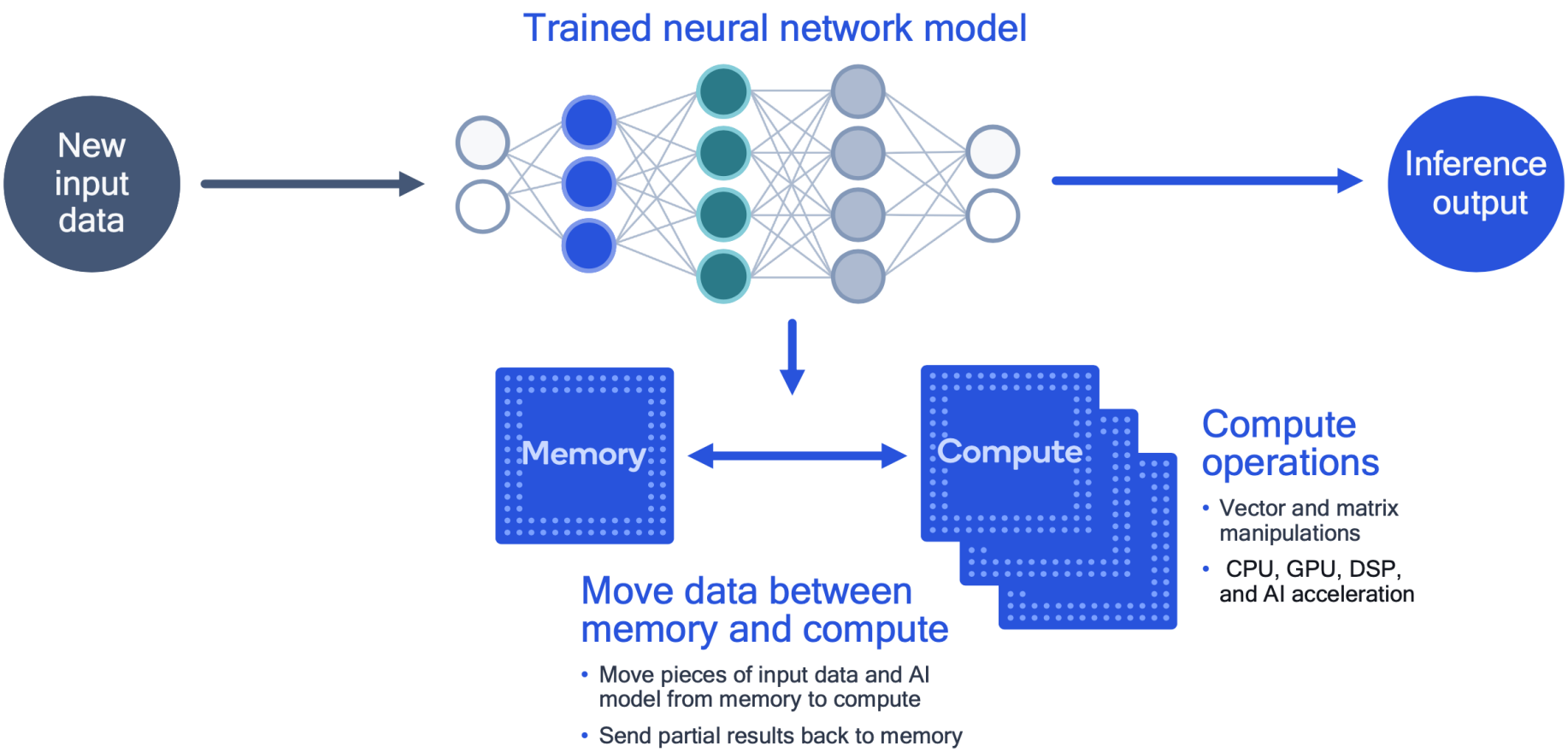
Rough Energy Cost For Various Operations in 45nm 0.9V

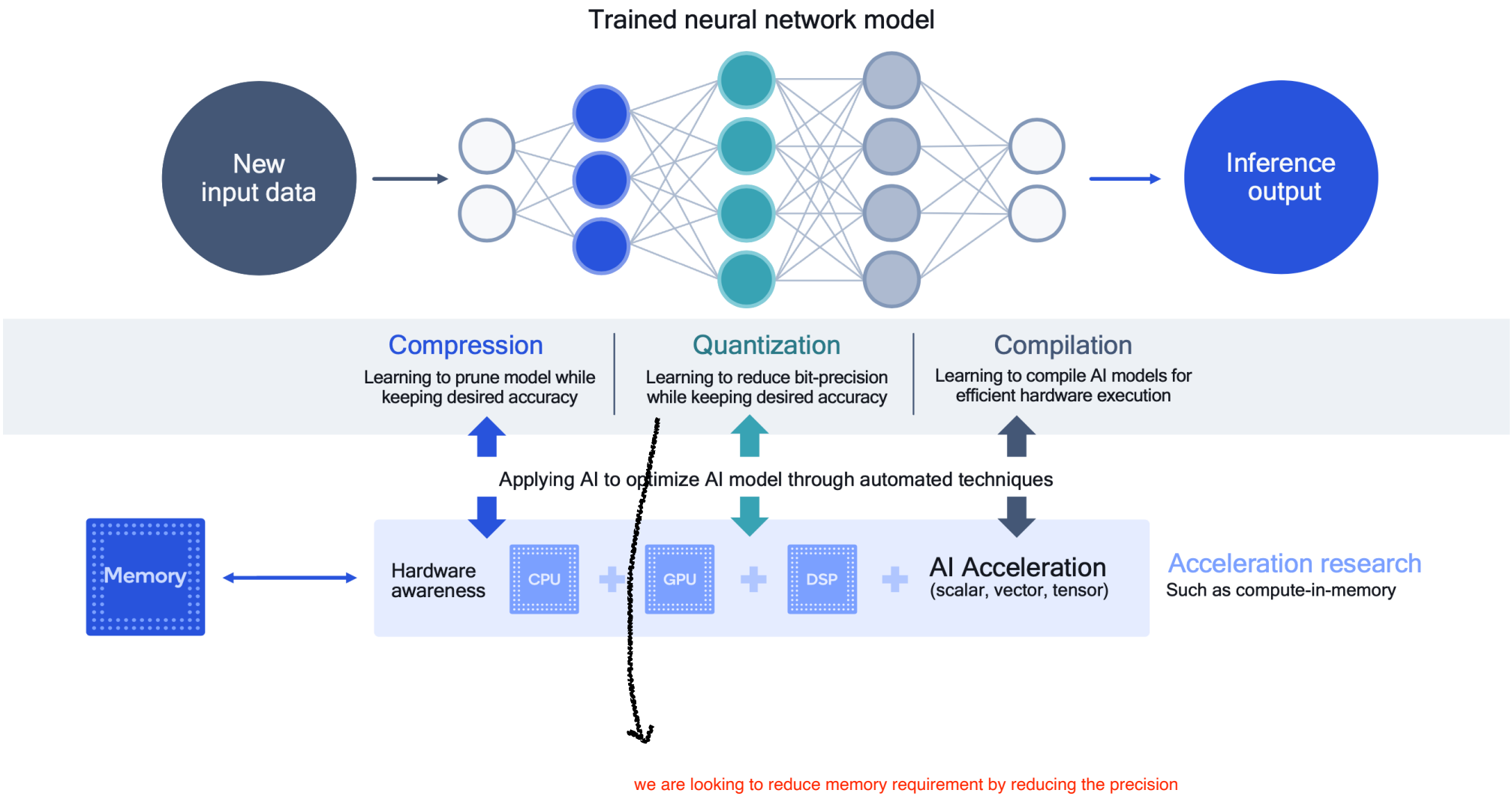


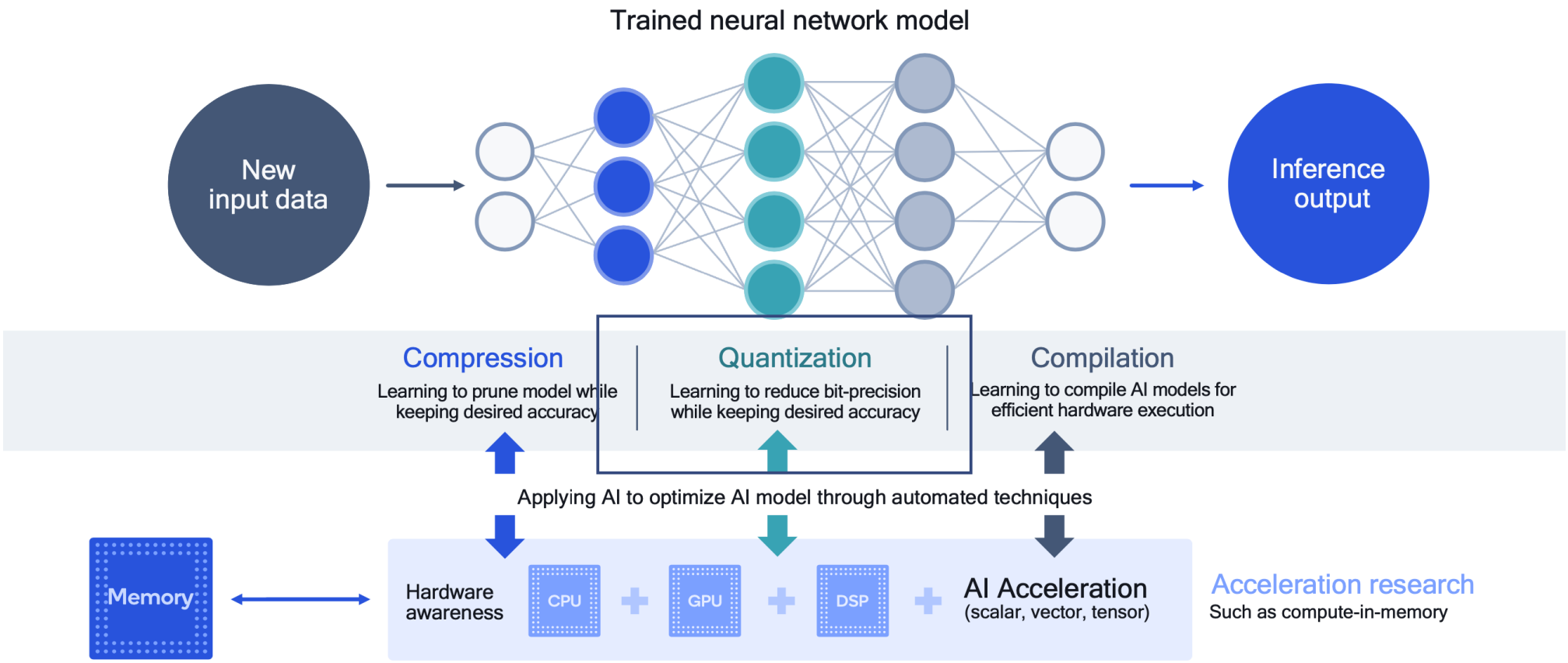
Battery images are in the public domain
Image 1, image 2, image 2, image 4

Computing's Energy Problem (and What We Can Do About it) [Horowitz, M., IEEE ISSCC 2014]

Advancing AI research to increase power efficiency

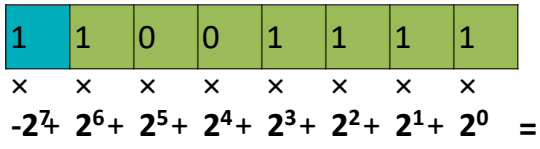


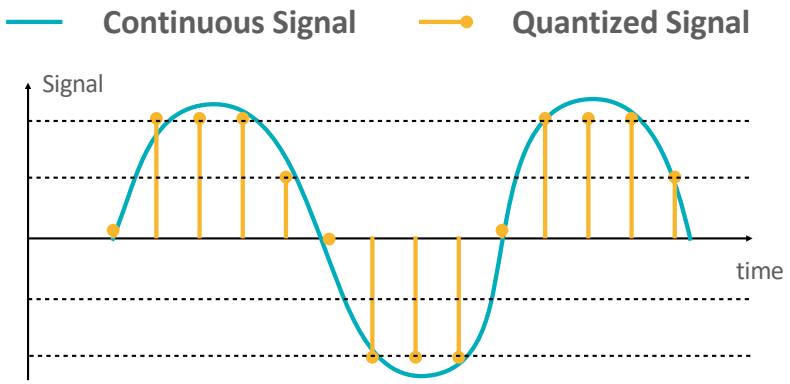

Trained neural network model
New input data
Inference output
Compression
Learning to prune model while keeping desired accuracy
Quantization
Learning to reduce bit-precision while keeping desired accuracy
Compilation
Learning to compile AI models for efficient hardware execution
Applying AI to optimize AI model through automated techniques
Memory
Hardware awareness
CPU
GPU
DSP
AI Acceleration
(scalar, vector, tensor)
Acceleration research
Such as compute-in-memory
we are looking to reduce memory requirement by reducing the precision


Trained neural network model
New input data
Inference output
Compression
Learning to prune model while keeping desired accuracy
Quantization
Learning to reduce bit-precision while keeping desired accuracy
Compilation
Learning to compile AI models for efficient hardware execution
Applying AI to optimize AI model through automated techniques
Memory
Hardware awareness
CPU
GPU
DSP
AI Acceleration
(scalar, vector, tensor)
Acceleration research
Such as compute-in-memory

Agenda

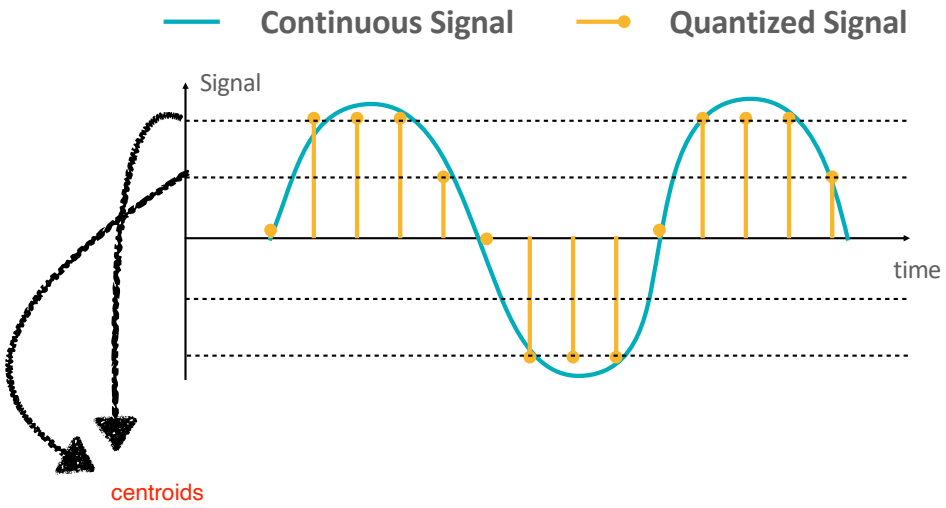
1. Numeric ***data types*** used in the modern computing systems, including integers and floating-point numbers.
2. Learn the basic concept of ***neural network quantization***
3. **Quantization approaches:**
    1. **Post-Training Quantization (PTQ)** that quantizes a floating-point neural network model, including weight quantization, activation quantization, and bias quantization.
    2. **Quantization-Aware Training (QAT)** that emulates inference-time quantization during the training/fine-tuning and recover the accuracy.

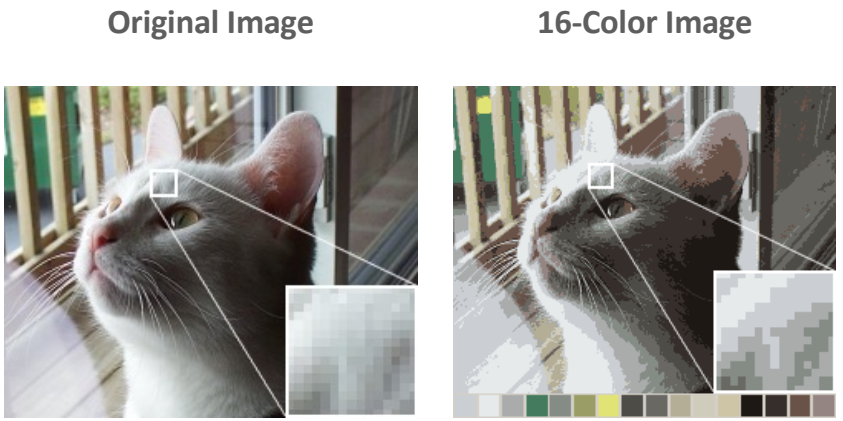




What is Quantization?

Quantization is the process of constraining an input from a continuous or otherwise large set of values to a discrete set.





Images are in the public domain.

- Wikipedia: Quantization

Quantizing an image - Example

8 bits/pixel

4 bits/pixel

2 bits/pixel

Numeric Data Types

How is numeric data represented in modern computing systems?

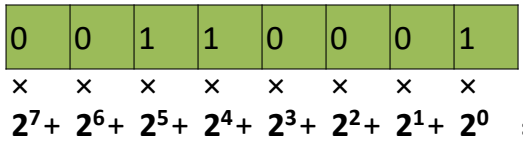
Integer

- Unsigned Integer
  - *n*-bit Range: $[0, 2^n - 1]$

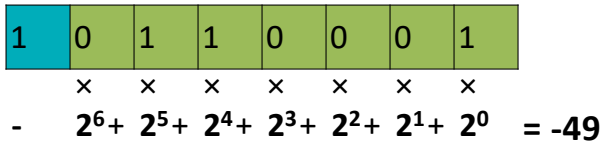
- Signed Integer
  - Sign-Magnitude Representation
    - *n*-bit Range: $[-2^{n-1} - 1, 2^{n-1} - 1]$
    - Both 000…00 and 100…00 represent 0
  - Two's Complement Representation
    - *n*-bit Range: $[-2^{n-1}, 2^{n-1} - 1]$
    - 000…00 represents 0
    - 100…00 represents -2^{n-1}

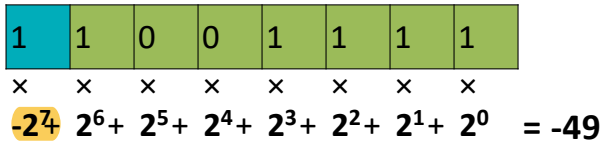
to overcome this

8 bit



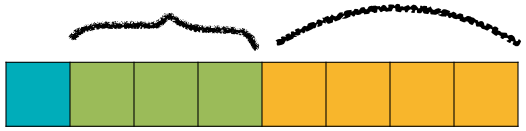
Sign Bit





Fixed-Point Number

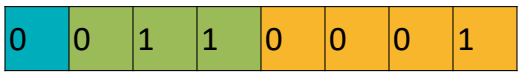
As we can use a fixed number of bits for the fraction part



Integer . Fraction

"Decimal" Point

$$-2^3 + 2^2 + 2^1 + 2^0 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} = 3.0625$$



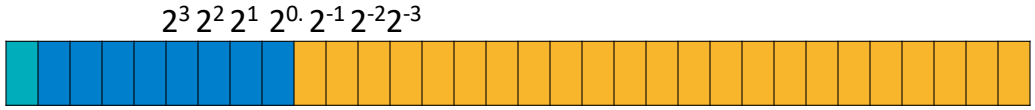
$$(-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0) \times 2^{-4} = 49 \times 0.0625 = 3.0625$$

(using 2's complement representation)

Floating-Point Number

Example: 32-bit floating-point number in IEEE 754

exponent bias = (#bits in exponent - 1) - all these bits set to 1

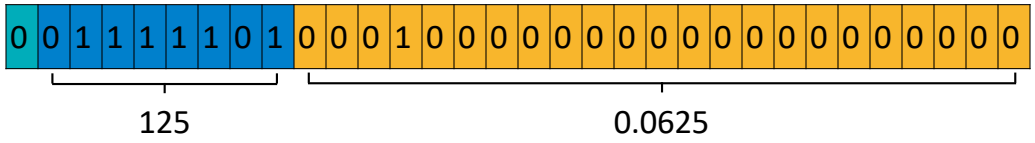


Sign **8-bit Exponent** (Integer) **23-bit Fraction**

$(-1)^{sign} \times (1 + \textbf{Fraction}) \times 2^{\textbf{Exponent}-127}$ ← **Exponent Bias = 127 = $2^{8-1}-1$**

(significant / mantissa)

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



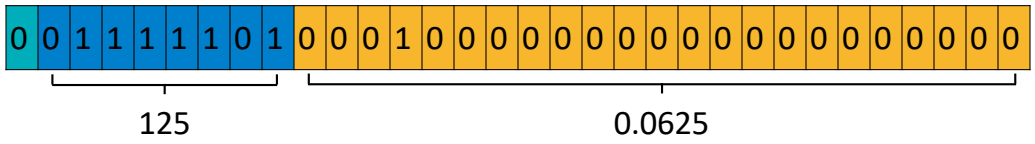
Floating-Point Number

Example: 32-bit floating-point number in IEEE 754

Sign **8-bit Exponent** **23-bit Fraction**

this is a shortcoming with this

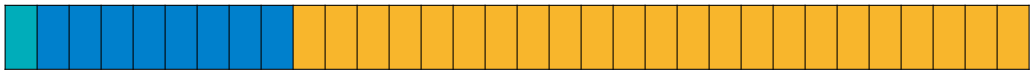
How to represent 0?



To include zero in this format, few changes are done -

Floating-Point Number

Example: 32-bit floating-point number in IEEE 754



Sign **8-bit Exponent** **23-bit Fraction**

Normal Numbers, Exponent ≠ 0

i.e. our number is non zero. then we can use this representation itself.

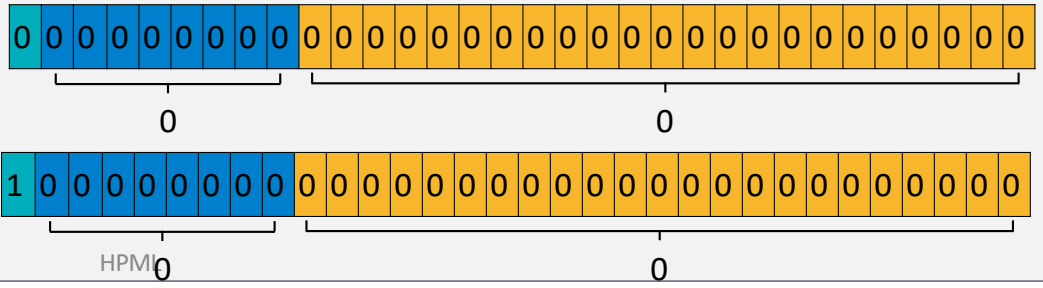
$(-1)^{sign} \times (1 + \textbf{Fraction}) \times 2^{\textbf{Exponent}-127}$

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

Subnormal Numbers, Exponent = 0

if, exponent = 0.
There are chances that our number is zero. so we need a different representation.

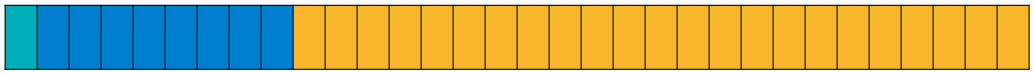
Instead of $(-1)^{sign} \times (1 + \textbf{Fraction}) \times 2^{\textbf{0}-127}$, we force it to be $(-1)^{sign} \times \textbf{Fraction} \times 2^{\textbf{1}-127}$



$0 = 0 \times 2^{-126}$

Floating-Point Number

Example: 32-bit floating-point number in IEEE 754

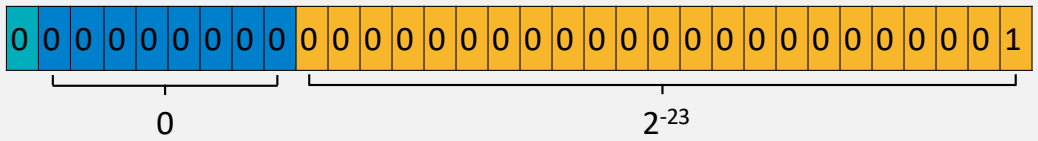


Sign **8-bit Exponent** **23-bit Fraction**

Subnormal Numbers, Exponent = 0

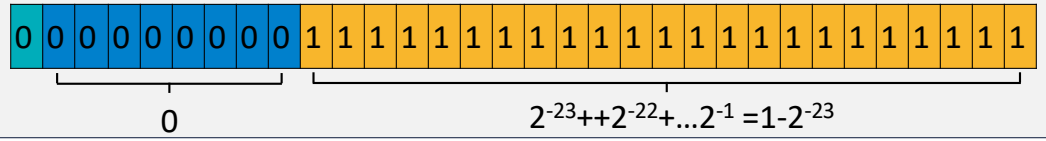
$(-1)^{sign} \times \mathbf{Fraction} \times 2^{\mathbf{1}-127}$

Smallest number that can be represented:



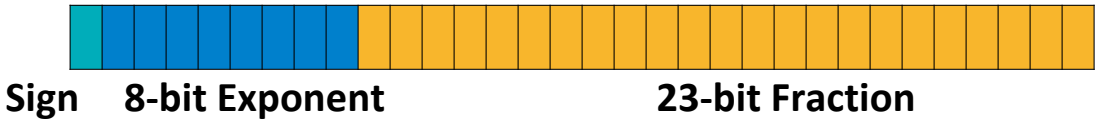
$2^{-149} = 2^{-23} \times 2^{-126}$

Largest subnormal number that can be represented:



Floating-Point Number

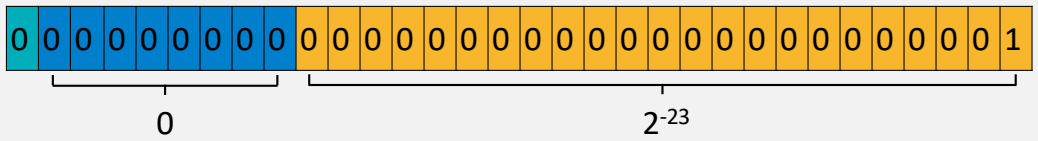
Example: 32-bit floating-point number in IEEE 754



Subnormal Numbers, Exponent = 0

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

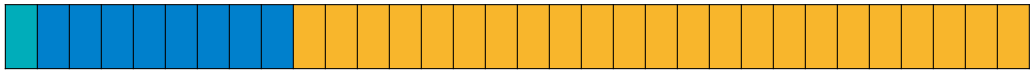
Smallest positive subnormal value that can be represented:



$2^{-149} = 2^{-23} \times 2^{-126}$

Floating-Point Number

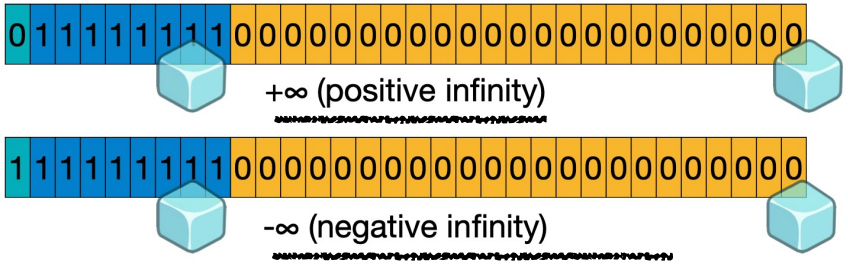
Example: 32-bit floating-point number in IEEE 754



Sign **8-bit Exponent** **23-bit Fraction**

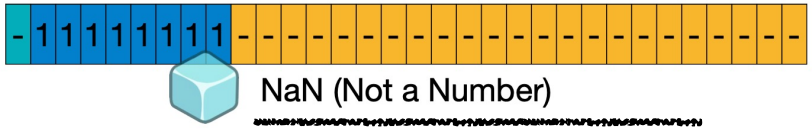
$(-1)^{sign} \times (1 + \textbf{Fraction}) \times 2^{\textbf{Exponent}-127}$

(Normal Numbers, Exponent≠0)



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

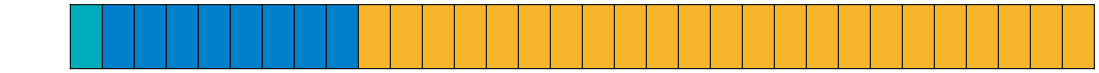
(Subnormal Numbers, Exponent=0)



much waste. Revisit in fp8.

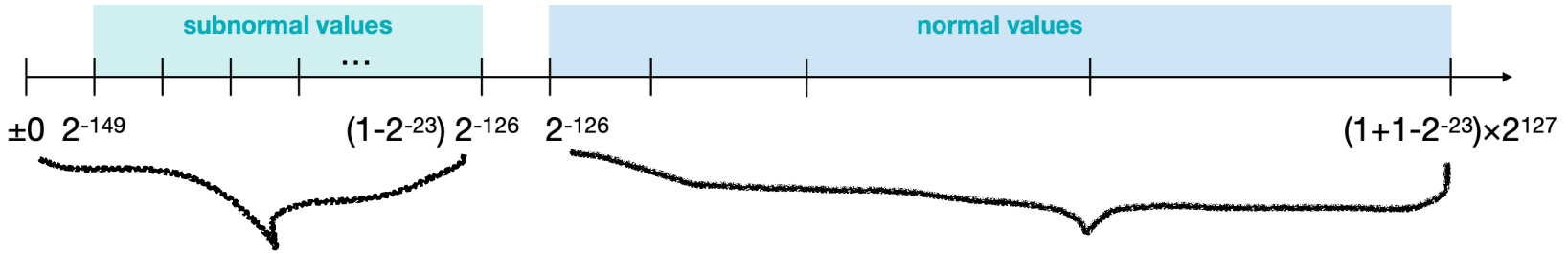
Floating-Point Number

Example: 32-bit floating-point number in IEEE 754



Sign **8-bit Exponent** **23-bit Fraction**

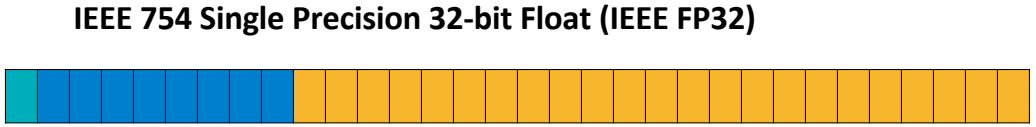
Exponent	Fraction=0	Fraction≠0	Equation
$00_H = 0$	±0	subnormal	$(-1)^{sign} \times \mathbf{Fraction} \times 2^{1-127}$
$01_H \ldots FE_H = 1 \ldots 254$	normal		$(-1)^{sign} \times (1 + \mathbf{Fraction}) \times 2^{\mathbf{Exponent}-127}$
$FF_H = 255$	±INF	NaN	

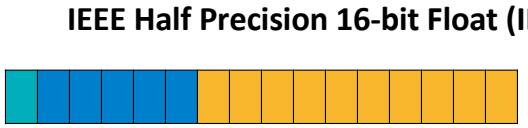


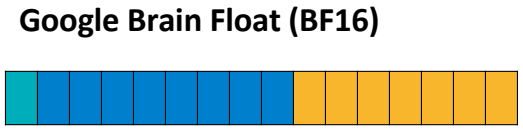
During training we need higher dynamic range.
But during inference, we need higher precision

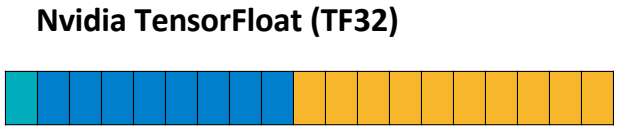
Floating-Point Number

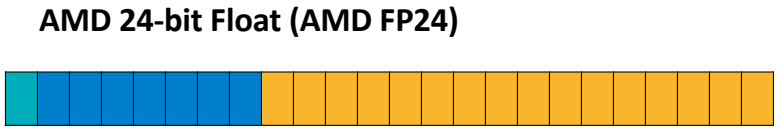
Exponent Width → Range; Fraction Width → Precision











Exponent (bits)	Fraction (bits)	Total (bits)
8	23	32
5	10	16
8	7	16
8	10	19
7	16	24

Floating-Point Number – NVIDIA FP8

Exponent Width → Range; Fraction Width → Precision

- **The FP8 datatype supported by H100 is actually 2 distinct datatypes, useful in different parts of the training of neural networks.**
- **The tradeoff of the increased dynamic range is the lower precision of the stored values.**

Nvidia FP8 (E4M3)

- FP8 E4M3 does not have INF, and S.1111.1112 is used for NaN.
- The largest FP8 E4M3 normal value is S.1111.1102 =448.
- **Forward activations and weights require more precision**, so the E4M3 datatype is best used during forward pass.

Nvidia FP8 (E5M2)

- FP8 E5M2 have INF (S.11111.002) and NaN (S.11111.XX2).
- The largest FP8 E5M2 normal value is S.11110.112 =57344.
- **In the backward pass, gradients** flowing through the network typically are less susceptible to the loss of precision but **require a higher dynamic range**. Therefore, they are best stored using the E5M2 data format.

Reference: https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8_primer.html

Floating-Point Number – NVIDIA FP8

Exponent Width → Range; Fraction Width → Precision

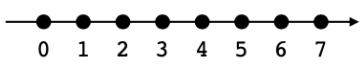
INT4

S			
0	0	0	1
0	1	1	1

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

=1

=7



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

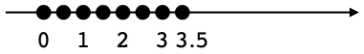
FP4 (E1M2)

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

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

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

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



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

FP4 (E2M1)

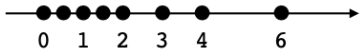
S	E	E	M
0	0	0	1
0	1	1	1

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

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

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

no inf, no NaN



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

FP4 (E3M0)

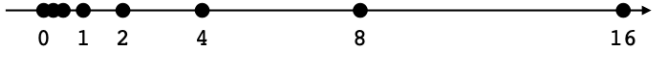
S	E	E	E
0	0	0	1
0	1	1	1

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

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

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

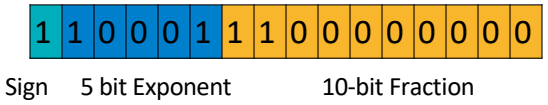
no inf, no NaN



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

Numeric Data Types

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



Exponent Bias = 15_{10}

$(-1)^{sign} \times (1 + \textbf{Fraction}) \times 2^{\textbf{Exponent}-15}$ ⟵ Exponent Bias = $15 = 2^{5-1}-1$

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

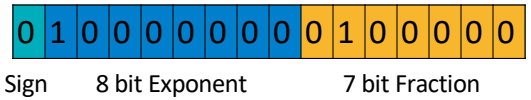
Numeric Data Types

- **Question**: What is the decimal 2.5 in Brain Float (BF16)?

$2.5_{10} = 1.\underline{25}_{10} \times 2^{\underline{1}}$

Exponent Bias = 15_{10}

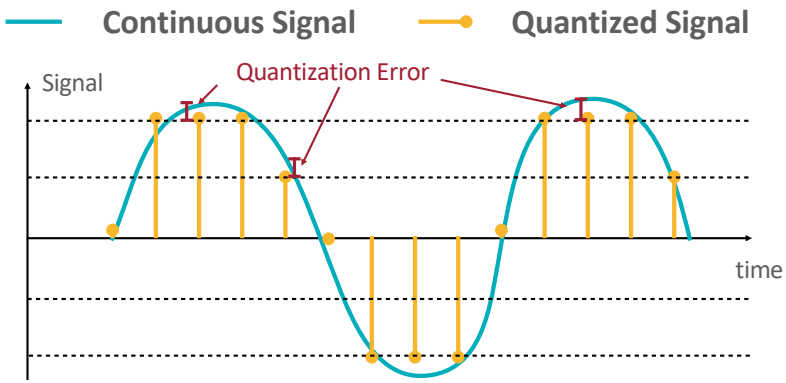
  - Sign: +
  - Exponent Binary: $1_{10} + 15_{10} = 16_{10} = 00010000_2$
  - Fraction Binary: $0.25_{10} = 0100000_2$
  - Binary Answer



What is Quantization?

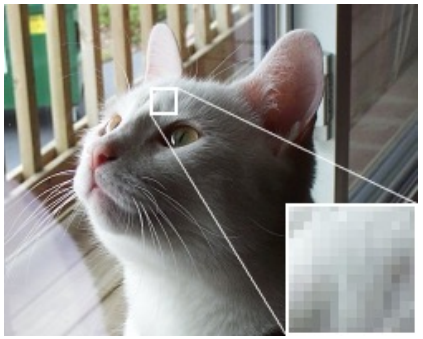
a floating point number is converted to discrete number

Quantization is the process of constraining an input from a continuous or otherwise large set of values to a discrete set.

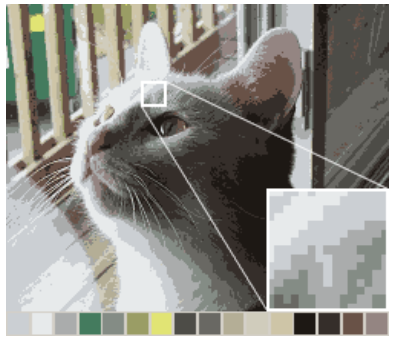


The difference between an input value and its quantized value is referred to as quantization error.

Original Image



16-Color Image



Images are in the public domain.

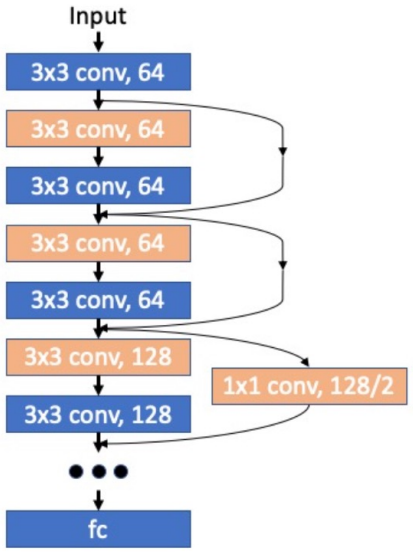
"Palettization"

- Quantization [Wikipedia]

Neural Network Quantization

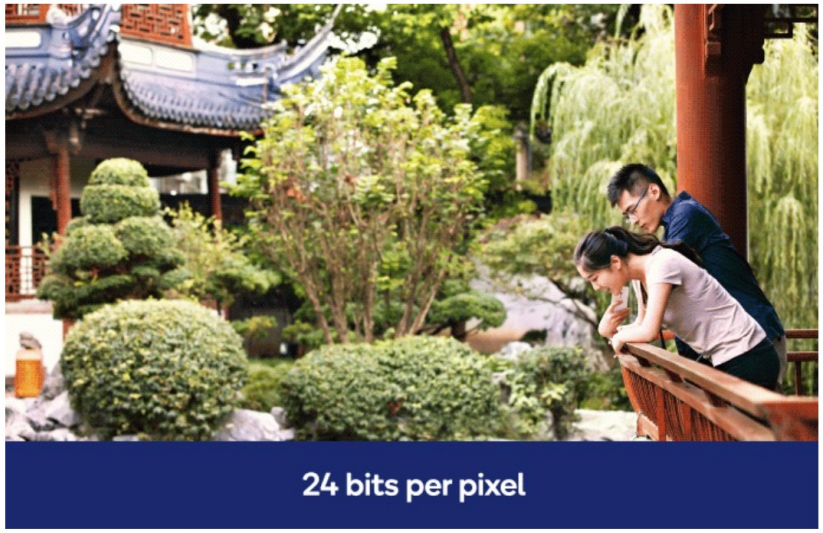
For any given trained neural network:

- Store weights in low bits (INT8)
- Compute calculations in low bits



Quantization Analogy

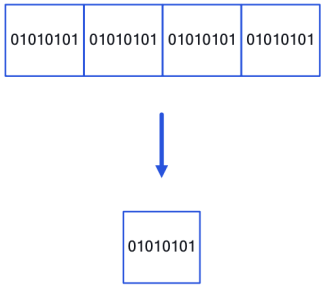
Use fewer bits to represent each pixel in an image



Significant Benefits with Quantization

Memory usage

8-bit versus 32-bit weights and activations stored in memory



Power consumption

Significant reduction in energy for both computations and memory access

Add energy (pJ)	
INT8	FP32
0.03	0.9
30X energy reduction	

Mult energy (pJ)	
INT8	FP32
0.2	3.7
18.5X energy reduction	

Mem access energy (pJ)	
Cache (64-bit)	
8KB	10
32KB	20
1MB	100
DRAM	1300-2600
Up to 4X energy reduction	

Latency

With less memory access and simpler computations, latency can be reduced

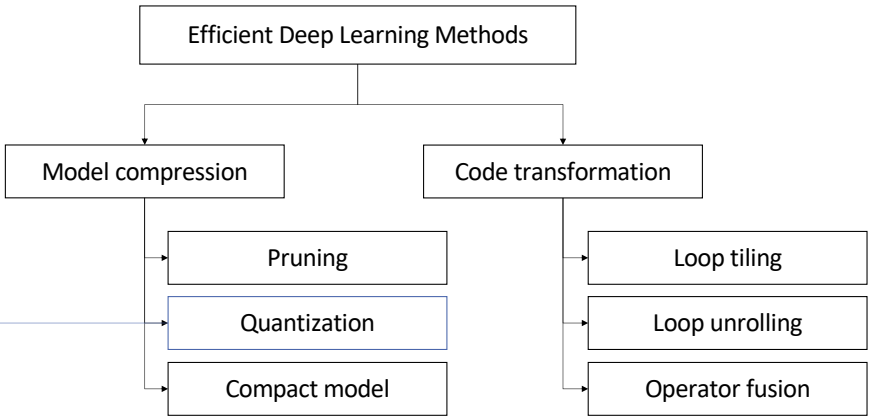
Silicon area

Integer math or less bits require less silicon area compared to floating point math and more bits

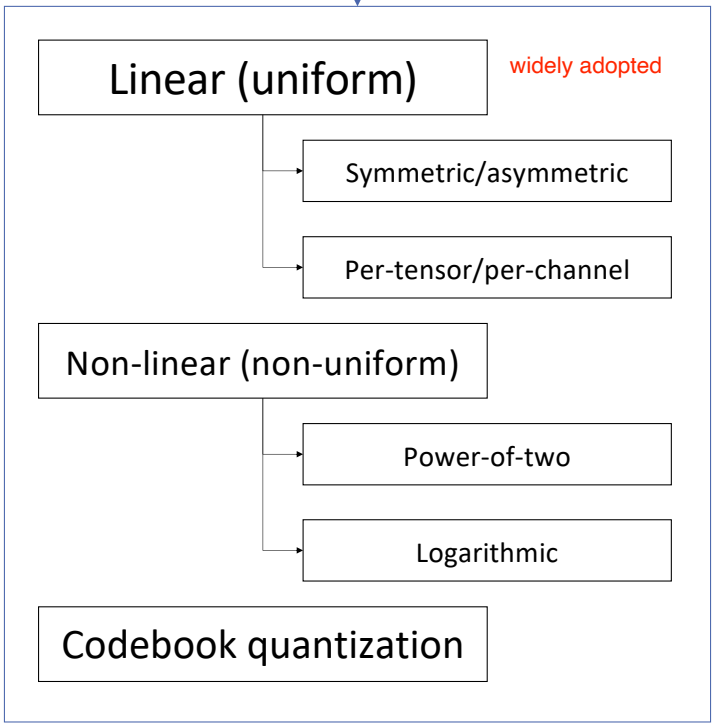
Add area (µm²)	
INT8	FP32
36	4184
116X area reduction	

Mult area (µm²)	
INT8	FP32
282	7700
27X area reduction	

How to quantize?



few approaches -



Quantization methods can be classified as Linear Quantization and Non-linear Quantization depending on whether the original data uniformly distributed or not. The weights and activation values of model are usually uneven, so the precession loss caused by Non-linear Quantization is smaller. On the other hand, the Linear Quantization, which is more commonly used, is more effective during inference than Non-linear Quantization.

why is it more effective during inference?

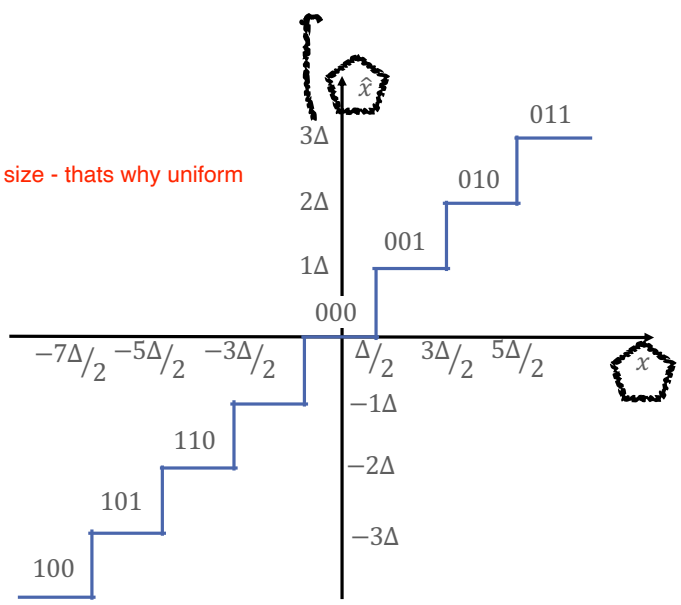
Linear (uniform) quantization

- Linear implies constant step size Δ
- Step size is determined by the range
  - Symmetric or asymmetric mode
  - Per-tensor or per-channel
- $\hat{x}$ the quantized version of x:

$$\hat{x} = \text{round}\left(x/\Delta\right) \cdot \Delta$$

centroid

constant step size - thats why uniform



3 bits quantization example

maximum 8 levels

delta -> scaling factor

range of x = -3.5delta to 3.5delta

range of x' = -3delta to 3delta

kind of, higer dimmension is mapped to lower dimmension

delta is determined by the range of the values. (min and max values)

Symmetric mode

symmetric around 0

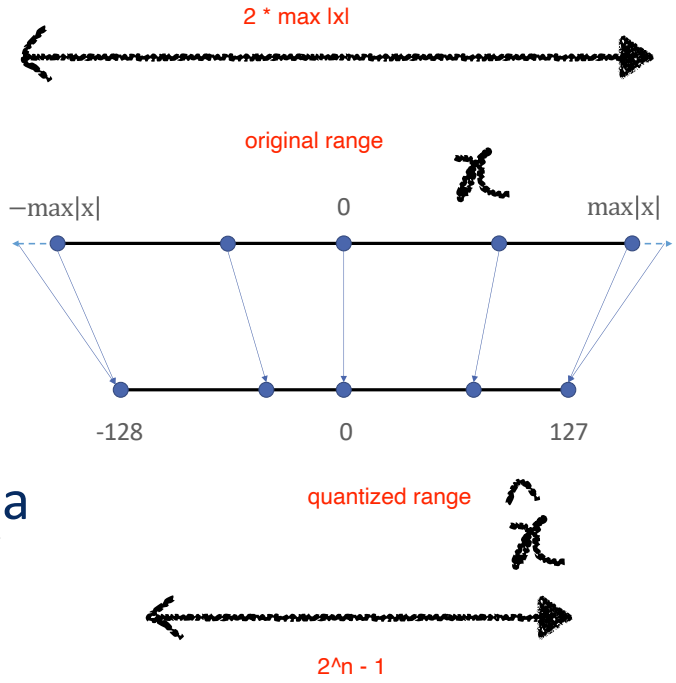
zero is mapped to zero

- Quantization range is symmetric around 0
- Step size given by:
- $\Delta = \frac{2 * \max|x|}{2^n - 1}$

range of x

max possible number with 'n' bits

- Conventional fixed-point format when step size is a power-of-two



Asymmetric mode

i.e. our inputs are not symmetric around zero

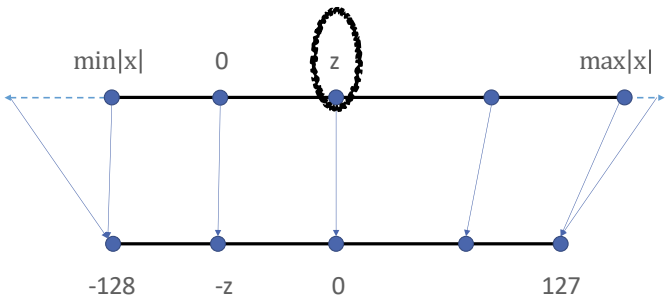
Useful when the distribution of values is **not** symmetric around 0 (e.g. after ReLU)

Step size given by:

$$\Delta = \frac{\max x - \min x}{2^n - 1}$$

this is same as the symmetric one

zero point



Use a zero-point z to shift the distribution:

$$\hat{x} = \text{round}\left(\frac{x}{\Delta} - z\right) \cdot \Delta + z$$

In practice the zero-point is chosen such that zero is exactly representable by an integer in the quantized domain

step size is still constant as this is uniform quantization

zero point is caliberated emperically

More details on asymmetric quantization matrix multiplication: https://github.com/google/gemmlowp/blob/master/doc/quantization.md#implementation-of-quantized-matrix-multiplication

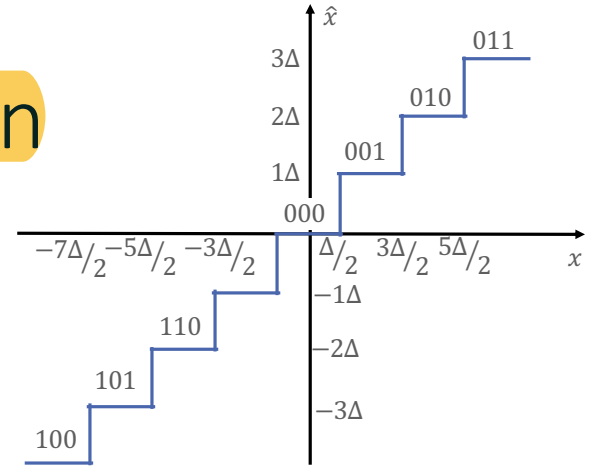
Unlike traditional linear quantization, which uniformly scales floating-point numbers to a smaller set of discrete values, non-linear quantization applies a more sophisticated mapping that preserves the distribution and characteristics of the original data more effectively.

Non-linear (non-uniform) quantization

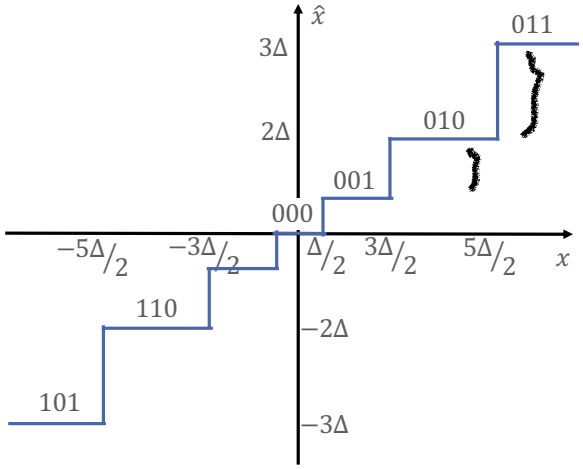
- Non-linear implies variable step size Δ
- Variability follows a regular pattern:
  - Power-of-two step
  - Logarithmic
- Concept:

  Assume weights/activations are non-uniformly distributed. More sampling points where many values reside increases the entropy of the quantized signal.

in linear, this would be be the case



Linear quantization



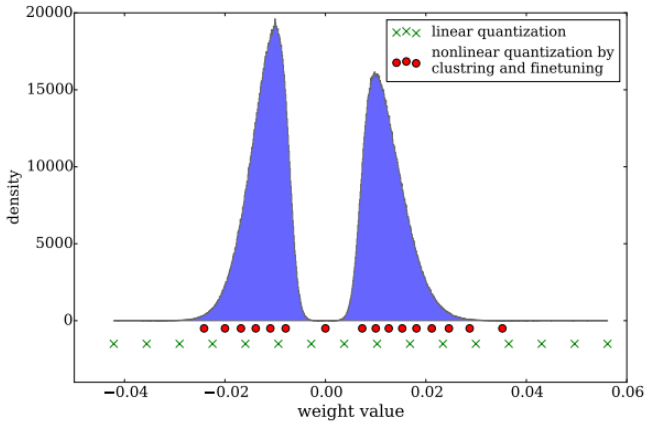
Non-linear quantization

Codebook quantization

- Useful when distribution of values is not normally distributed

why?

- Quantized values (codewords) are represented by a codebook
- Codewords are selected using a clustering algorithm or training process
- Codewords may be scalars or vectors
- Note: requires hardware support for lookup



Codebook example

Index	Value	Interval
0	-0.07	[-0.08, -0.04]
1	0	[-0.04, 0.05]
2	0.01	[0.05, 0.011]
3	0.012	[0.011, 0,013]

similar to K-Mean clusturing

each data point is mapped to its cluster mean (which is in low dimmension. i.e. uses less bits for representation)

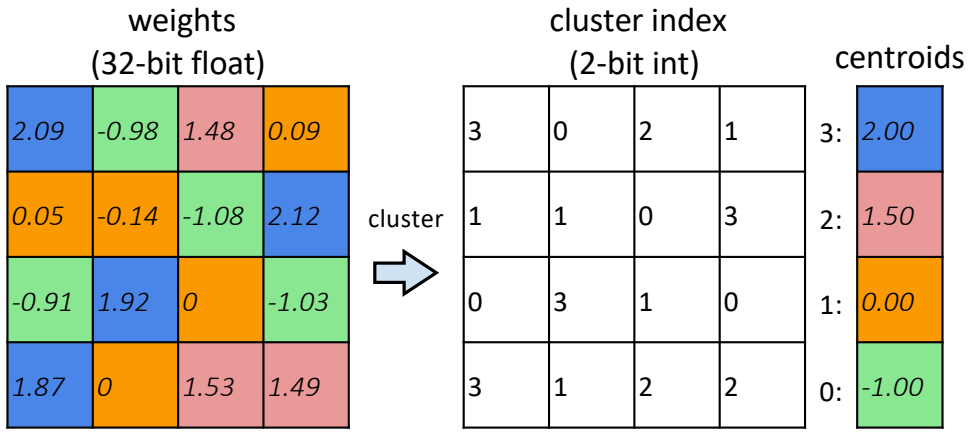
Example: K-Means-based Weight Quantization – Example of Codebook Quantization

weights
(32-bit float)

2.09	*-0.98*	*1.48*	*0.09*
0.05	*-0.14*	*-1.08*	*2.12*
-0.91	*1.92*	*0*	*-1.03*
1.87	*0*	*1.53*	*1.49*

weights with same colour are in same cluster

Example: K-Means-based Weight Quantization (Inference Pass)



reconstructed weights
(32-bit float)

2.00	-1.00	1.50	0.00
0.00	0.00	-1.00	2.00
-1.00	2.00	0.00	-1.00
2.00	0.00	1.50	1.50

storage

32 bit × 16
= 512 bit = 64 B

+

2 bit × 16
= 32 bit = 4 B

+

32 bit × 4
= 128 bit = 16 B

= 20 B

3.2 × smaller

Assume *N*-bit quantization, and #parameters = $M >> 2^N$.

32 bit × *M*
= *32M* bit

N bit × *M*
= *NM* bit

~~32 bit × 2^N~~
~~= 2^{N+5} bit~~

32/*N* × smaller

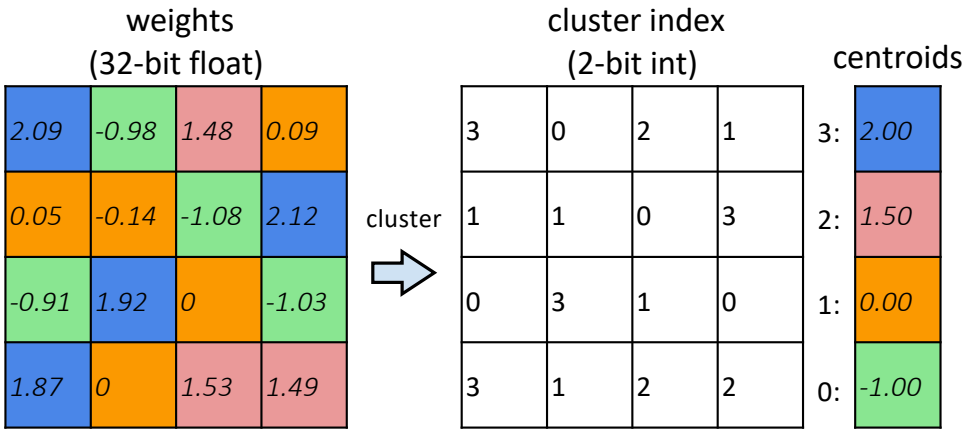
Deep Compression [Han *et al.*, ICLR 2016]

quantization error

0.09	0.02	-0.02	0.09
0.05	-0.14	-0.08	0.12
0.09	-0.08	0	-0.03
-0.13	0	0.03	-0.01

K-Means-based Weight Quantization

- Fine-tuning Quantized Weights



gradient

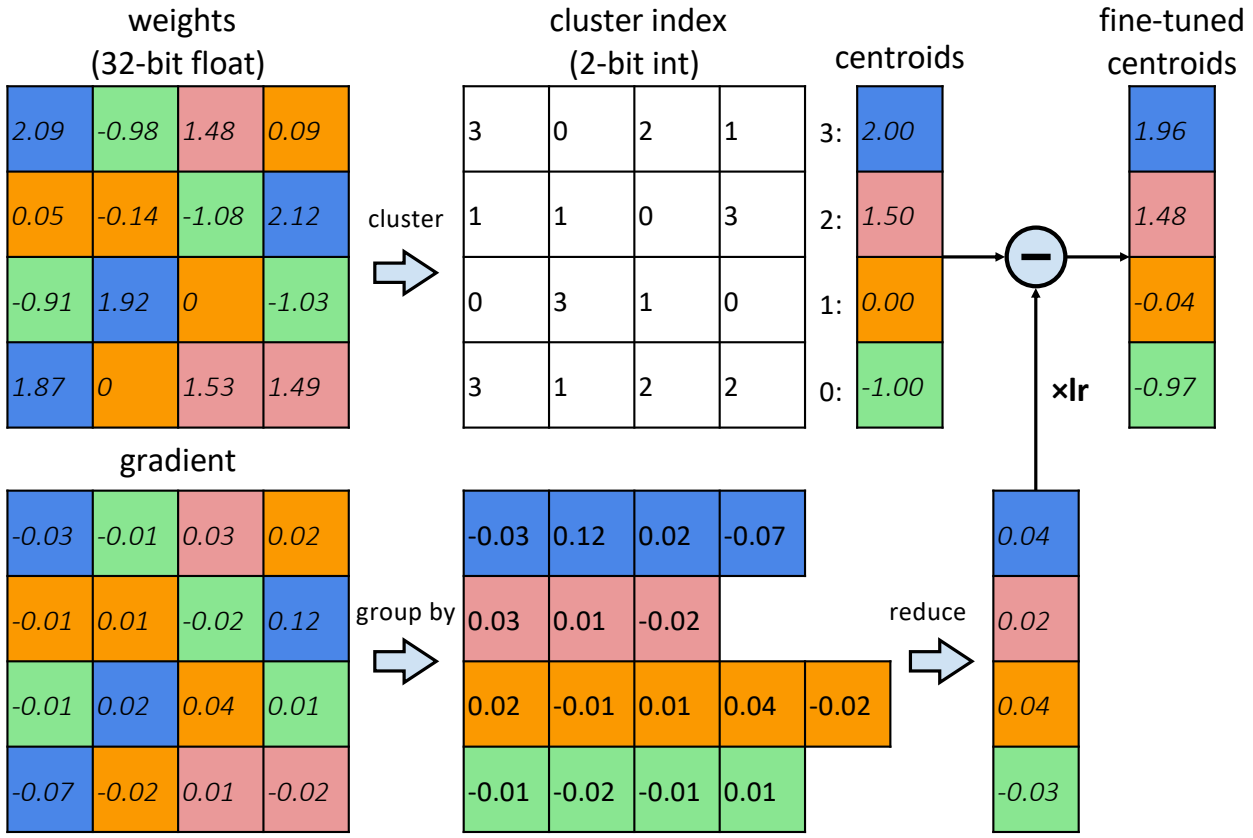
-0.03	-0.01	0.03	0.02
-0.01	0.01	-0.02	0.12
-0.01	0.02	0.04	0.01
-0.07	-0.02	0.01	-0.02

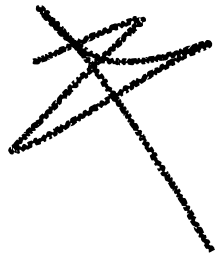
gradients are classified using same pattern as the weights.
i.e. assigned to same clusters.

Deep Compression [Han *et al.*, ICLR 2016]

Example: K-Means-based Weight Quantization

- Fine-tuning Quantized Weights

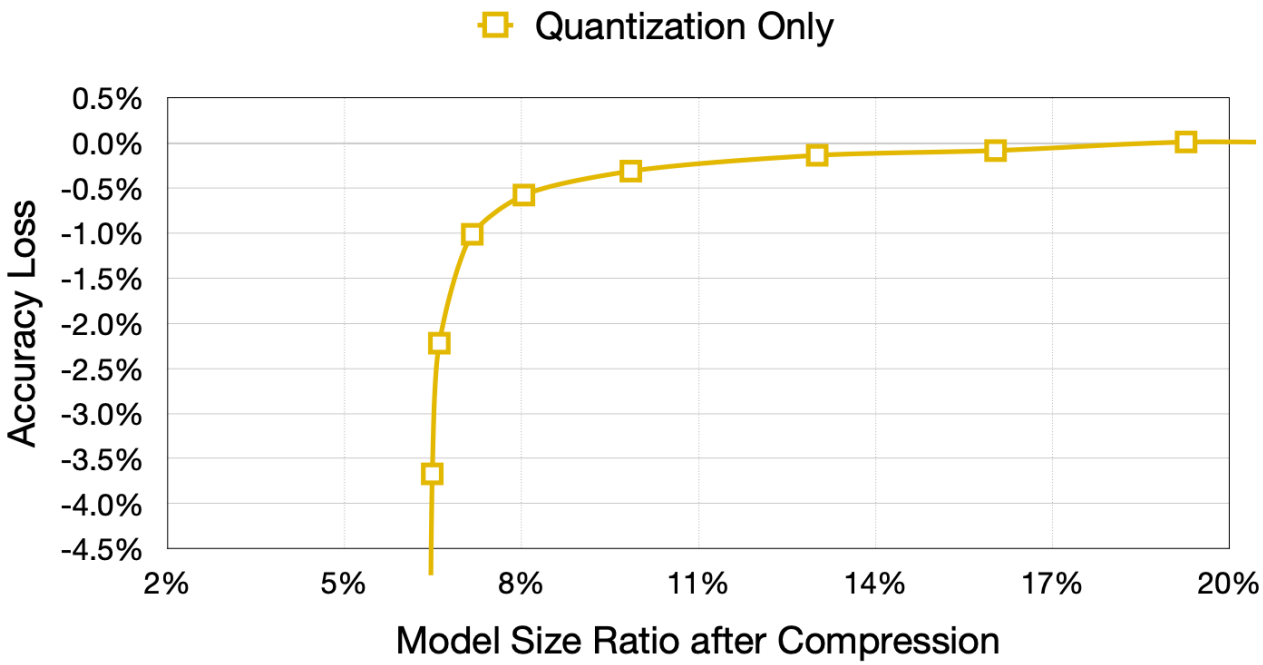




Deep Compression [Han *et al.*, ICLR 2016]

Example: K-Means-based Weight Quantization

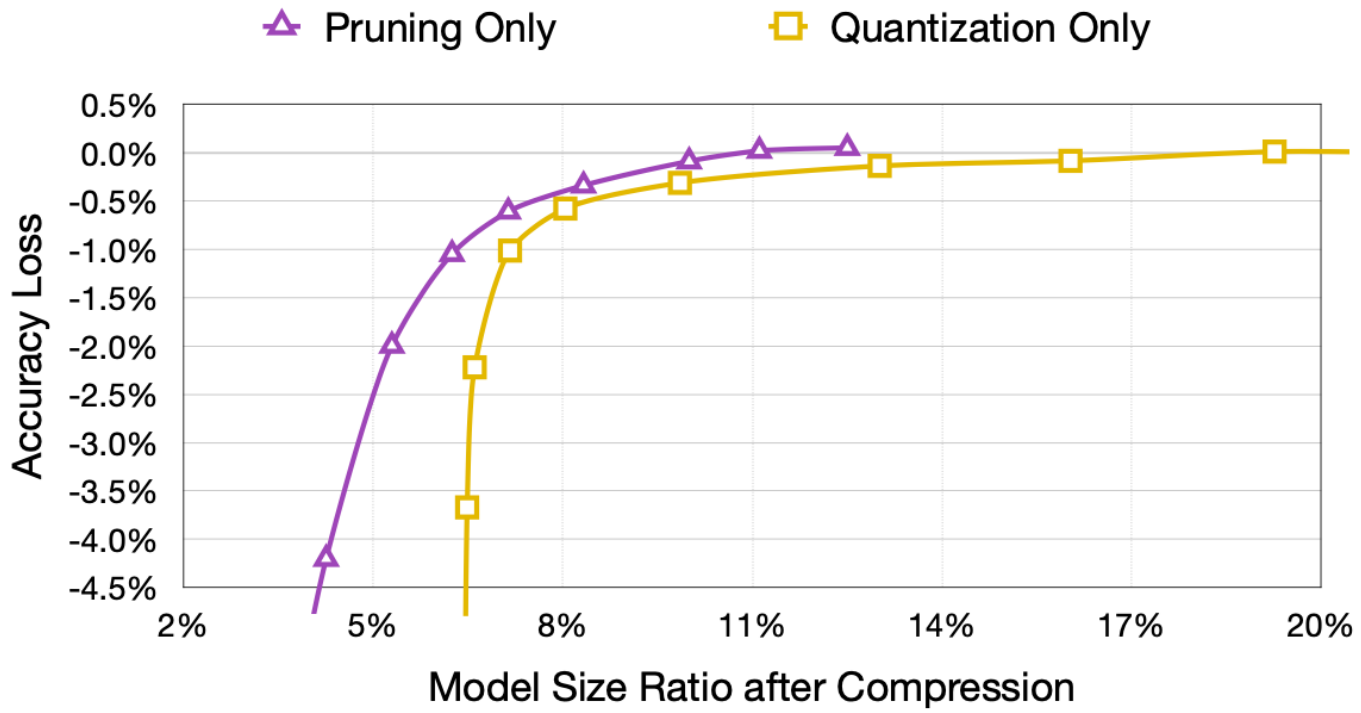
Accuracy vs. compression rate for AlexNet on ImageNet dataset



Deep Compression [Han *et al.*, ICLR 2016]

Example: K-Means-based Weight Quantization

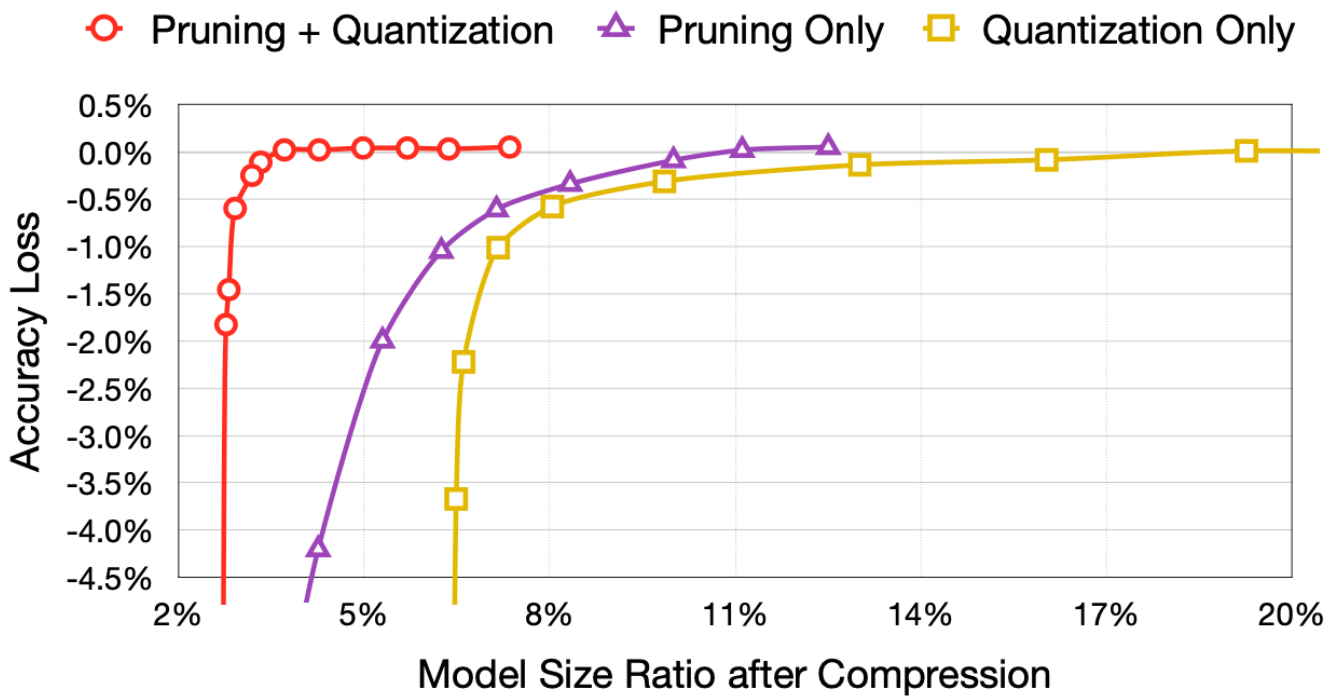
Accuracy vs. compression rate for AlexNet on ImageNet dataset



Deep Compression [Han *et al.*, ICLR 2016]

Example: K-Means-based Weight Quantization

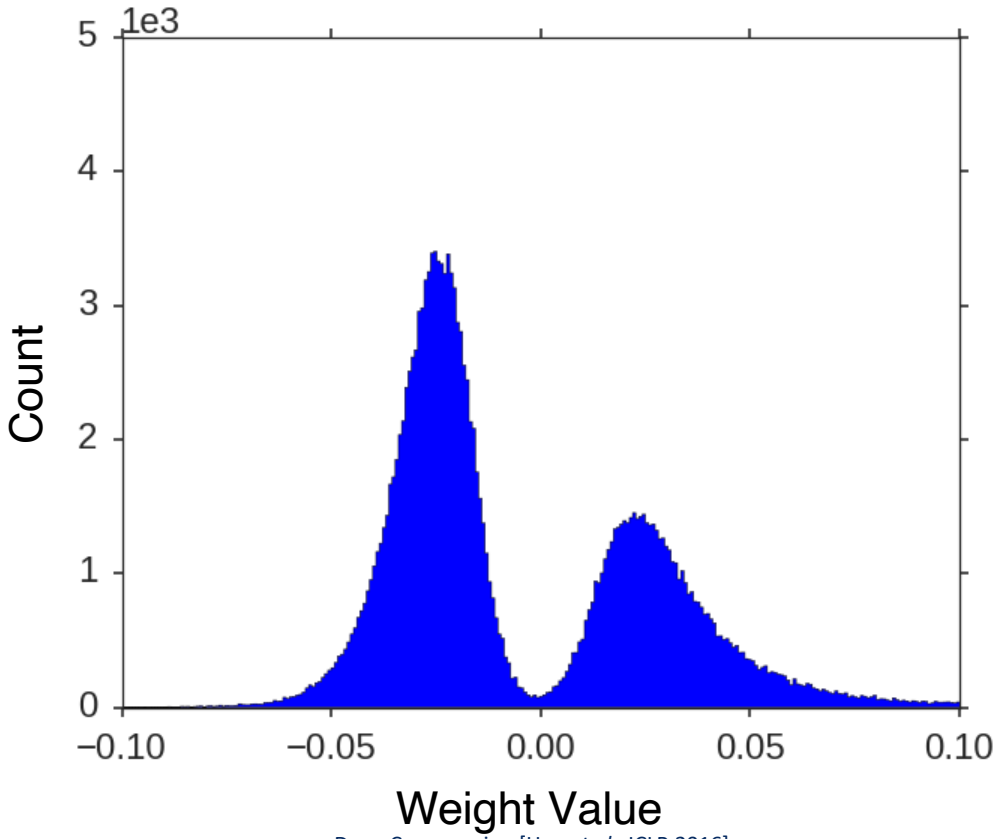
Accuracy vs. compression rate for AlexNet on ImageNet dataset



Deep Compression [Han *et al.*, ICLR 2016]

Before Quantization: Continuous Weight

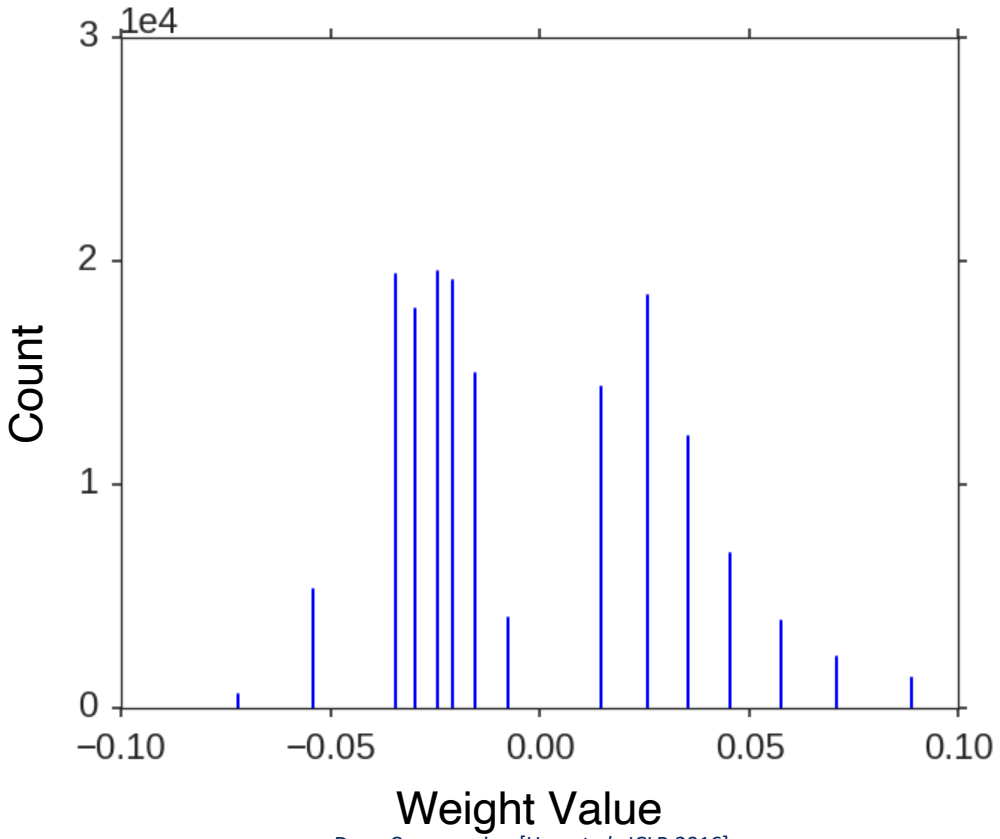
generally it works for non-normal distribution



- Deep Compression [Han *et al.*, ICLR 2016]

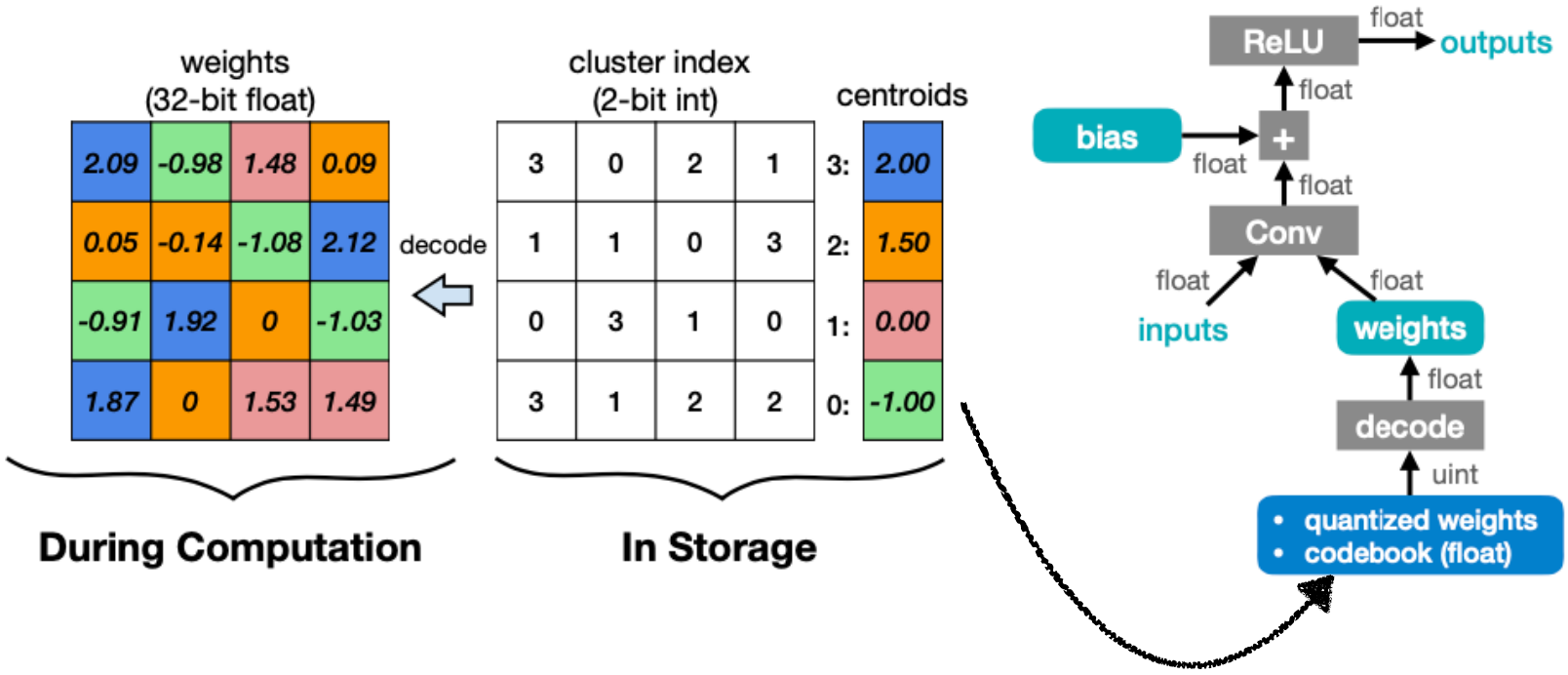
https://efficientml.ai

After Quantization: Discrete Weight

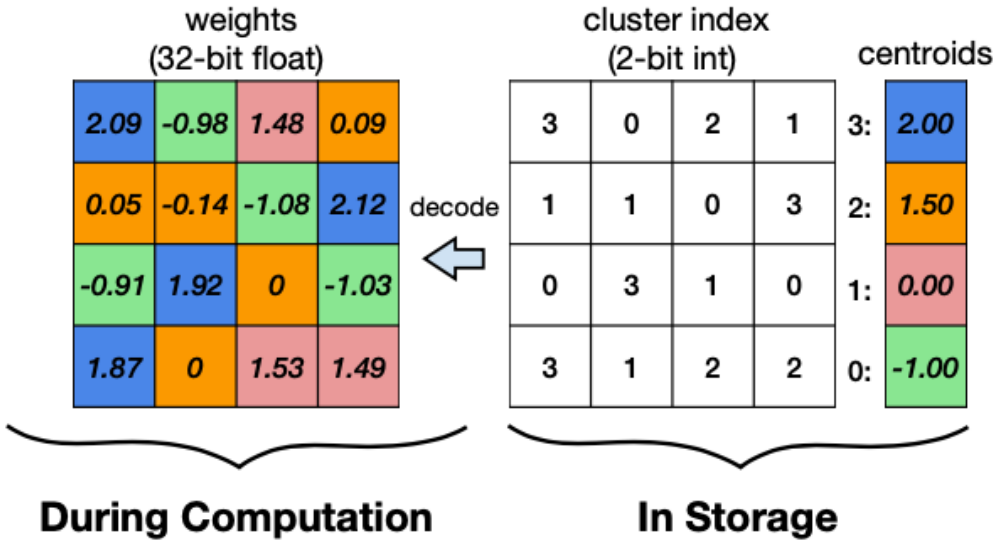


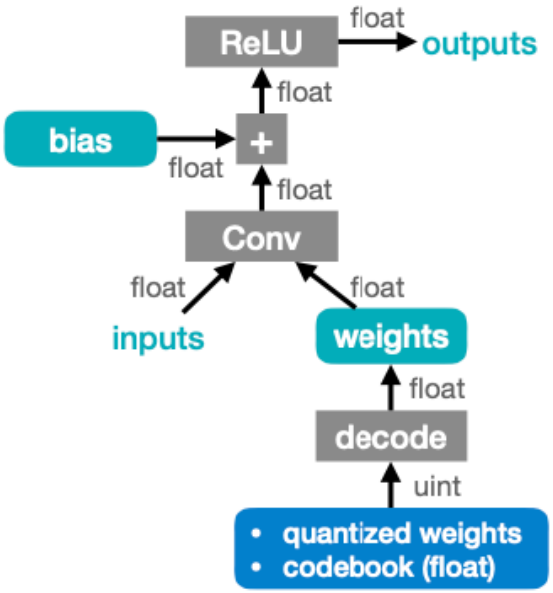
- Deep Compression [Han *et al.*, ICLR 2016]

K-Means-based Weight Quantization



K-Means-based Weight Quantization



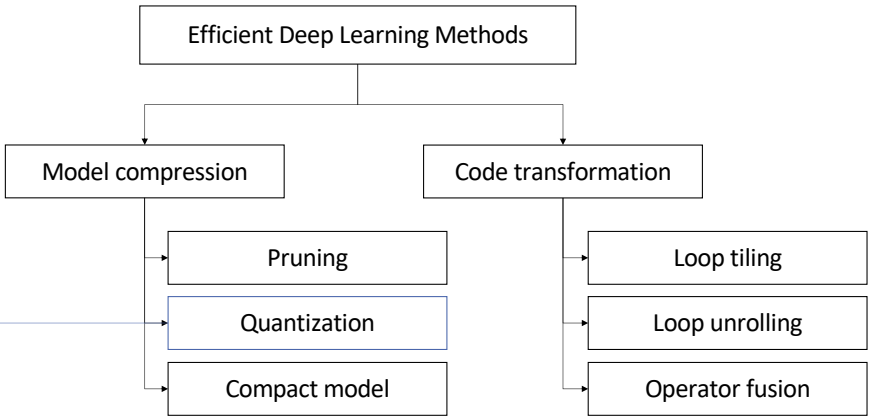


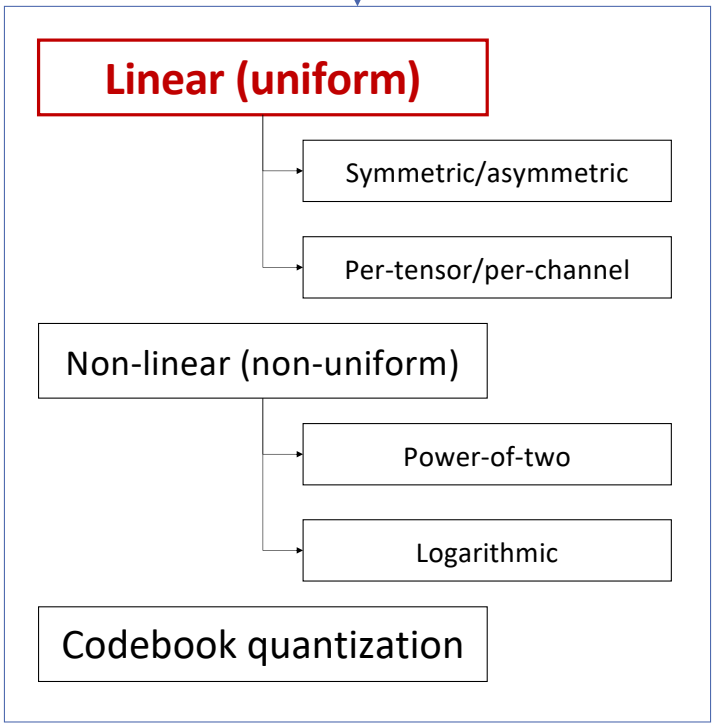
- The weights are decompressed using a lookup table (*i.e.*, codebook) during runtime inference.
- K-Means-based Weight Quantization only saves storage cost of a neural network model.
  - All the computation and memory access are still floating-point.

How to quantize? - Summary

Scheme	Step size	Step size pattern
Linear	Fixed	Regular
Non-linear	Variable	Regular
Codebook	Variable	Irregular

How to quantize?



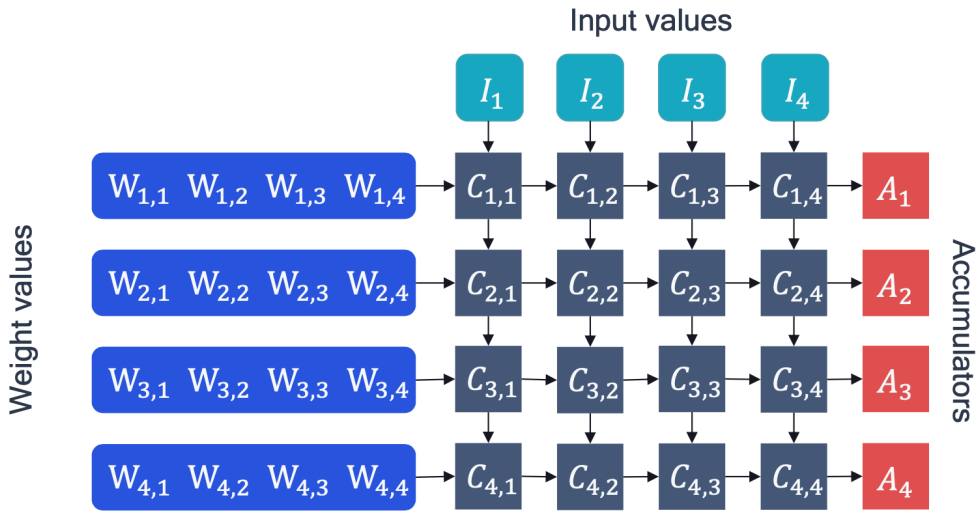


Matrix Operations: Backbone of neural networks

- How do we make these operations more efficient
- How to most efficiently calculate WX+b?

$$\boldsymbol{W} = \begin{pmatrix} 0.97 & 0.64 & 0.74 & 1.00 \\ 0.58 & 0.84 & 0.84 & 0.81 \\ 0.00 & 0.18 & 0.90 & 0.28 \\ 0.57 & 0.96 & 0.80 & 0.81 \end{pmatrix} \quad \boldsymbol{X} = \begin{pmatrix} 0.41 & 0.25 & 0.73 & 0.66 \\ 0.00 & 0.41 & 0.41 & 0.57 \\ 0.42 & 0.24 & 0.71 & 1.00 \\ 0.39 & 0.82 & 0.17 & 0.35 \end{pmatrix} \quad \boldsymbol{b} = \begin{pmatrix} 0.1 \\ 0.2 \\ 0.3 \\ 0.4 \end{pmatrix}$$

A schematic MAC array for efficient computation

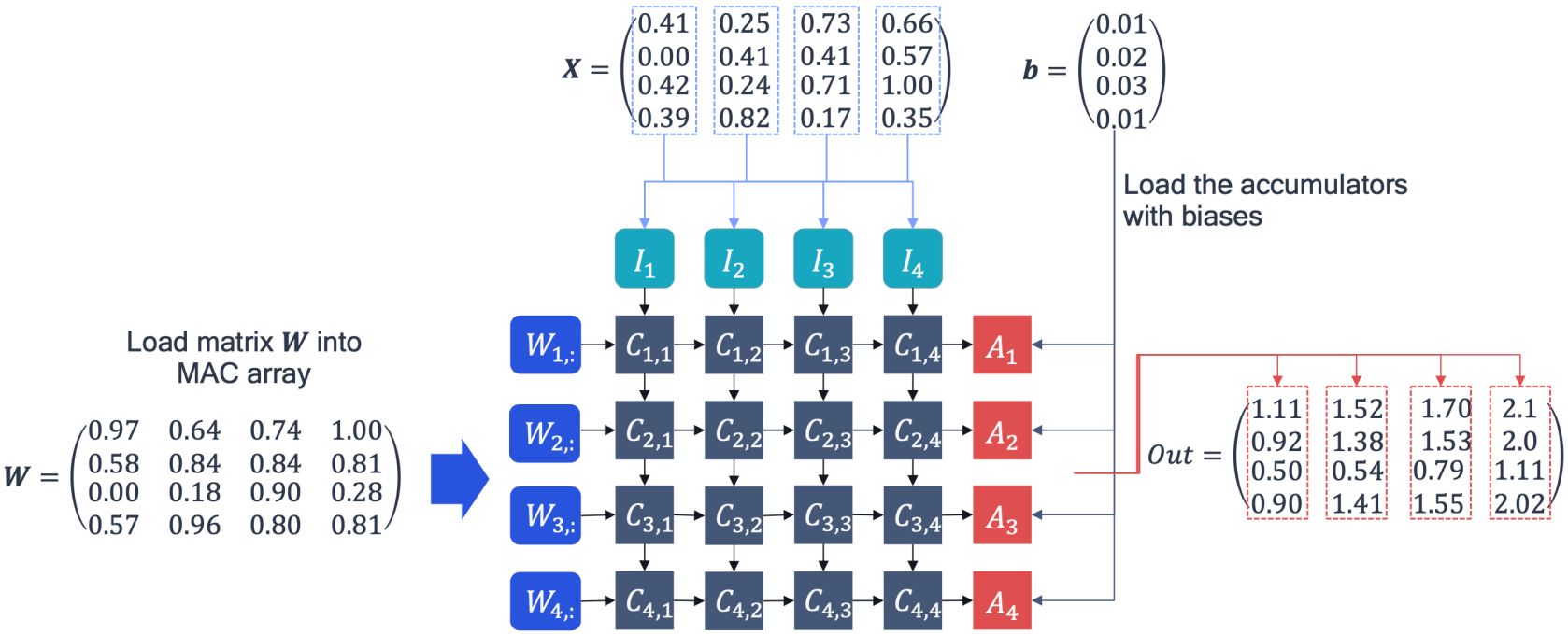


The array efficiently calculates the dot product between multiple vectors

$$A_i = \sum_j C_{i,j} + \boldsymbol{b}_i$$

$$A_i = W_i \cdot I_1 + W_i \cdot I_2 + W_i \cdot I_3 + W_i \cdot I_4$$

Step-by-step Matrix Multiplication in MAC array



Quantization comes at a cost of lost precision

- We can approximate an FP tensor with an integer tensor multiplied by a scale-factor S_X

$$\boldsymbol{X} \approx s_X \boldsymbol{X}_{\text{int}} = \widehat{\boldsymbol{X}}$$

FP32 tensor ← $\boldsymbol{X} \approx s_X \boldsymbol{X}_{\text{int}} = \widehat{\boldsymbol{X}}$ → scaled quantized tensor

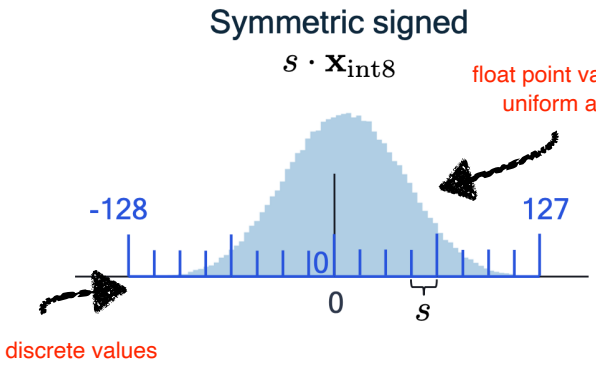
$$\boldsymbol{W} = \begin{pmatrix} 0.97 & 0.64 & 0.74 & 1.00 \\ 0.58 & 0.84 & 0.84 & 0.81 \\ 0.00 & 0.18 & 0.90 & 0.28 \\ 0.57 & 0.96 & 0.80 & 0.81 \end{pmatrix} \approx \frac{1}{255} \begin{pmatrix} 247 & 163 & 189 & 255 \\ 148 & 214 & 214 & 207 \\ 0 & 46 & 229 & 71 \\ 145 & 245 & 204 & 207 \end{pmatrix} = s_W \boldsymbol{W}_{\text{uint8}}$$

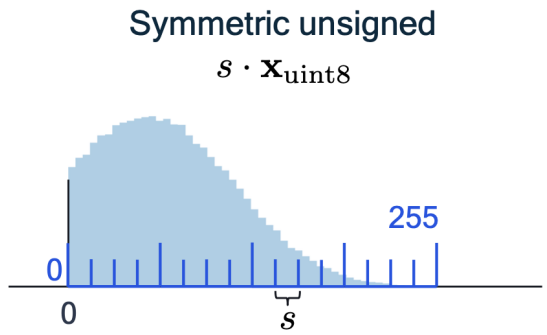
- Quantization is not free:

$$\boldsymbol{\epsilon} = \boldsymbol{W} - s_W \boldsymbol{W}_{\text{int}} = \frac{1}{255} \begin{pmatrix} 0.35 & 0.20 & -0.3 & 0 \\ -0.1 & 0.20 & 0.20 & -0.45 \\ 0.00 & -0.1 & -0.5 & 0.40 \\ 0.35 & -0.2 & 0 & -0.45 \end{pmatrix}$$

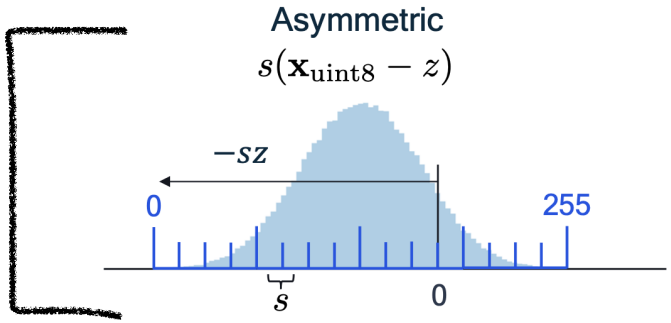
Different Types of quantization

- Symmetric, asymmetric, signed and unsigned quantization





Fixed Point Grid
Floating point Grid
S: scale factor
Z: zero point



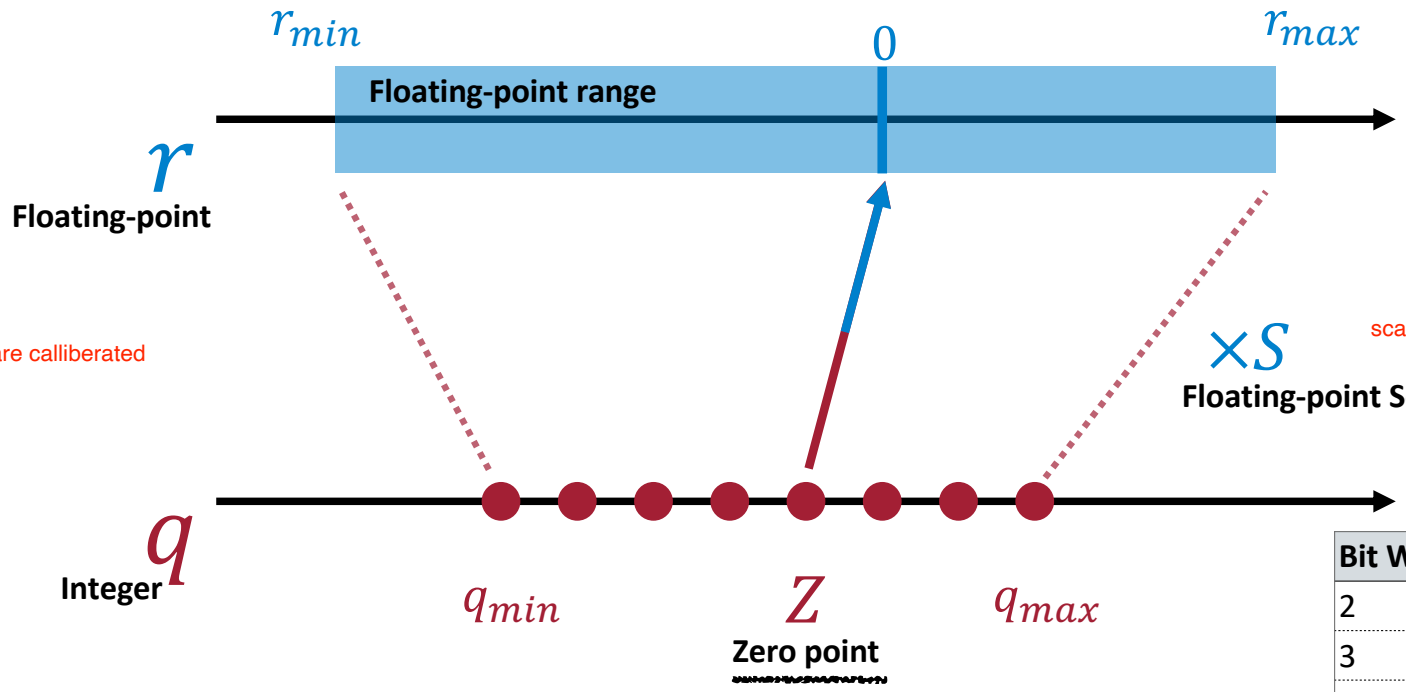
distribution of the original tensor AND
the grid it is mapped to (i.e. quantized values)

equation to get normal weights from quantised ones in the case of uniform linear mapping

Linear Quantization Definition

- An affine mapping of integers to real numbers $r = S(q - Z)$

linear



s and z are calliberated

scaling factor

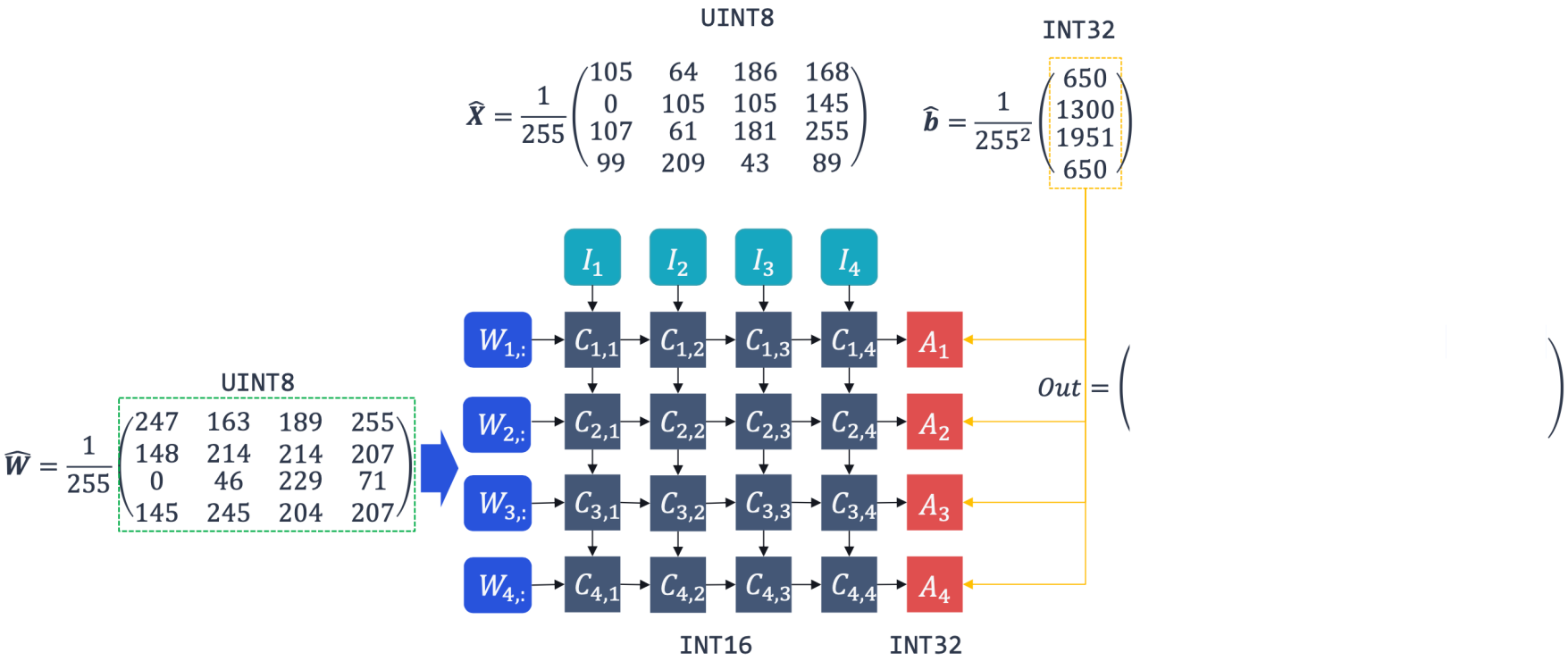
uniform trend

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

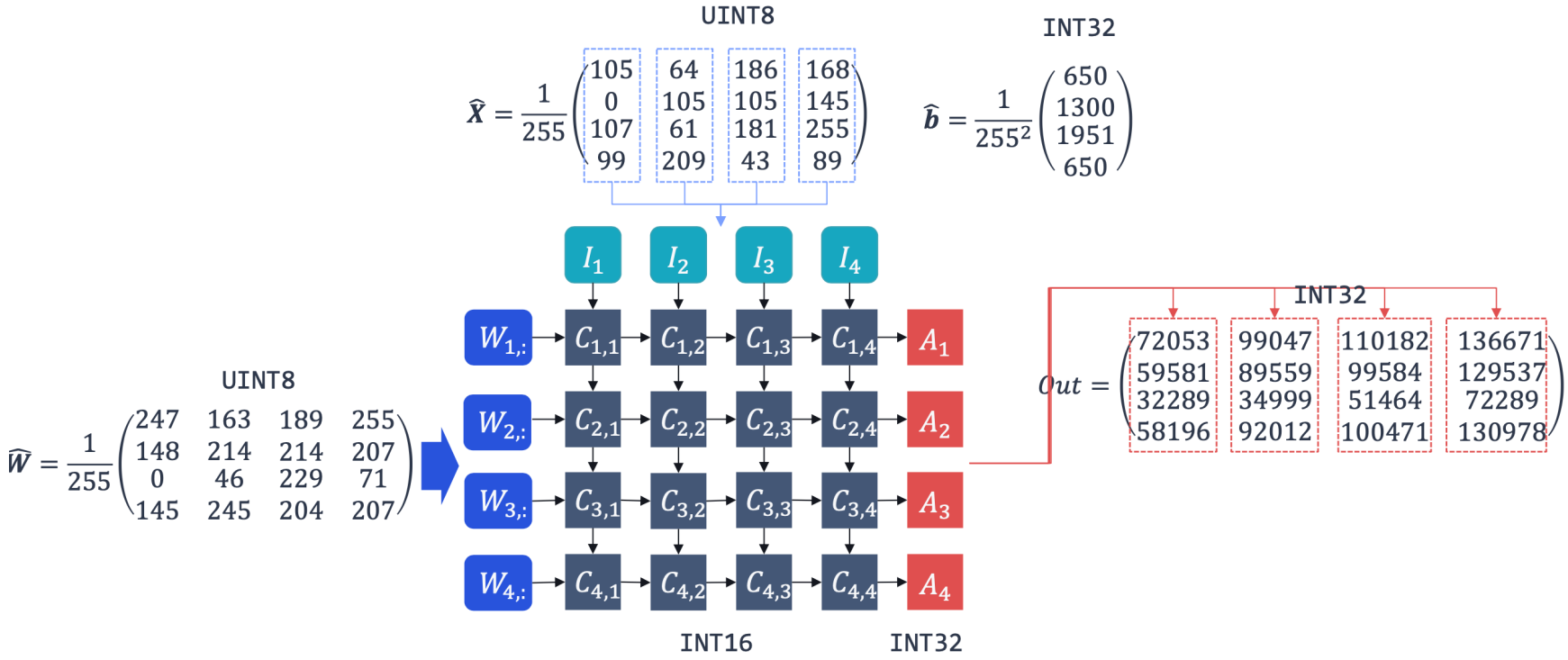
Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference [Jacob *et al.*, CVPR 2018]

$$S = \frac{r_m}{q_m}$$

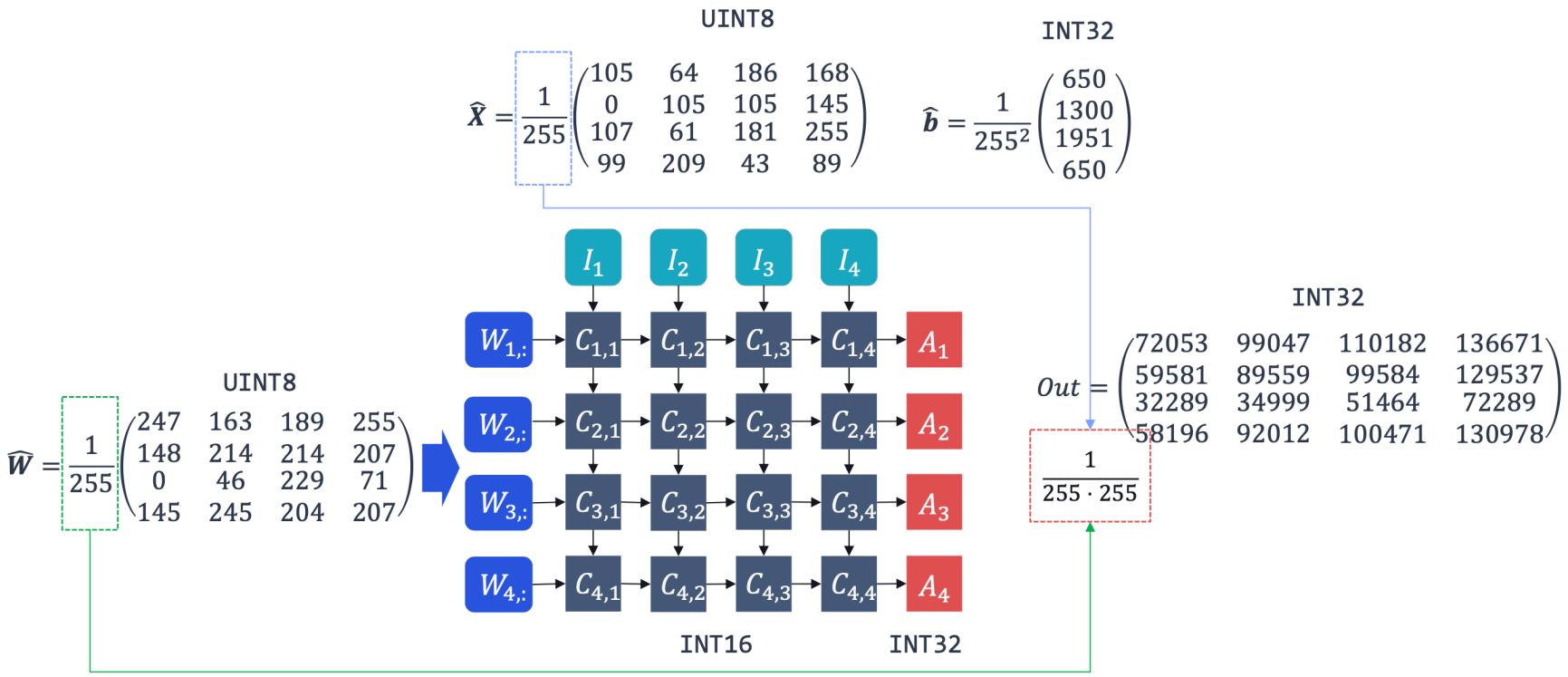
Quantized inference using symmetric quantization



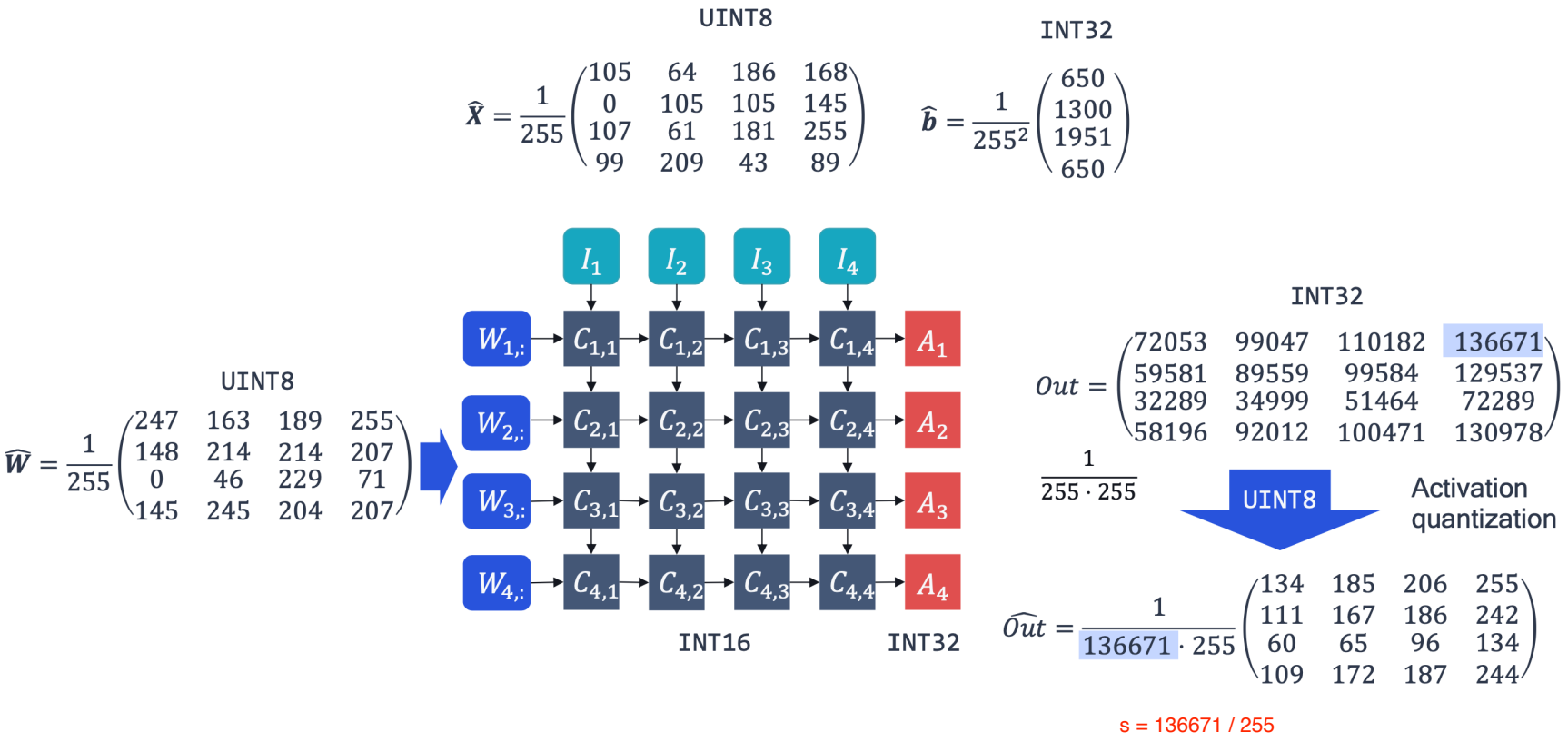
Quantized inference using symmetric quantization



Quantized inference using symmetric quantization



Quantized inference using symmetric quantization



What types of quantization to use?

$\boldsymbol{W}$: weight matrix

$\boldsymbol{X}$: input of a layer

Symmetric quantization

$$\boldsymbol{WX} \approx s_W(\boldsymbol{W}_{\text{int}})\, s_X(\boldsymbol{X}_{\text{int}})$$

$$= s_W s_X(\boldsymbol{W}_{\text{int}}\, \boldsymbol{X}_{\text{int}})$$

quantised

Same calculation

Asymmetric quantization

$$\boldsymbol{WX} \approx s_W(\boldsymbol{W}_{\text{int}} - z_W)\, s_X(X_{\text{int}} - z_X)$$

$$= s_W s_X(\boldsymbol{W}_{\text{int}}\, \boldsymbol{X}_{\text{int}}) + s_W s_X z_X \boldsymbol{W}_{\text{int}} + s_W z_W s_X z_X + s_W s_X z_W \boldsymbol{X}_{\text{int}}$$

Same calculation

Precompute, add to layer bias

Data-dependent overhead

Asymmetric weight quantization is equivalent to adding an input channel

this is because of the last term.

one more layer of computation during inference

What types of quantization to use?

$\boldsymbol{W}$: weight matrix

$\boldsymbol{X}$: input of a layer

Symmetric quantization

$$\boldsymbol{WX} \approx s_W(\boldsymbol{W}_{\text{int}})\, s_X(\boldsymbol{X}_{\text{int}})$$
$$= s_W s_X(\boldsymbol{W}_{\text{int}}\, \boldsymbol{X}_{\text{int}})$$

Same calculation

Asymmetric quantization

$$\boldsymbol{WX} \approx s_W(\boldsymbol{W}_{\text{int}} - z_W)\, s_X(X_{\text{int}} - z_X)$$
$$= s_W s_X(\boldsymbol{W}_{\text{int}}\, \boldsymbol{X}_{\text{int}}) + s_W s_X z_X \boldsymbol{W}_{\text{int}} + s_W z_W s_X z_X + s_W s_X z_W \boldsymbol{X}_{\text{int}}$$

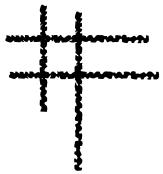
Precompute, add to layer bias

Data-dependent overhead

Asymmetric weight quantization is equivalent to adding an input channel

Symmetric weights and asymmetric activations are more hardware efficient

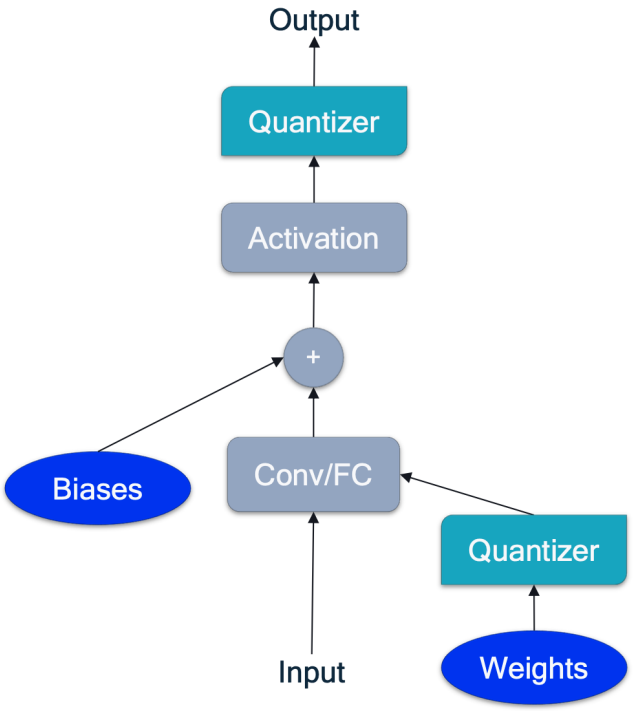
why?



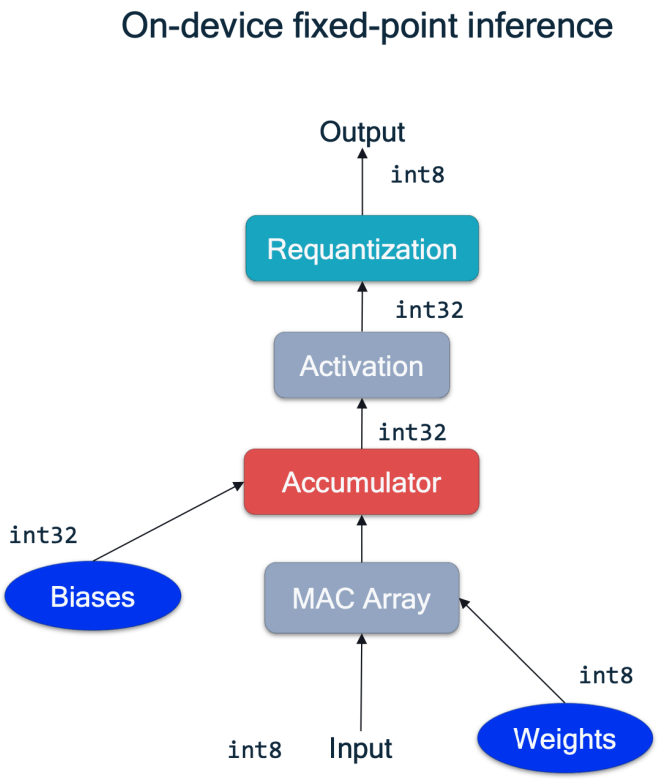
Why simulate quantization?

to find out the s and z beforhand. (before actual quantization on hardware)

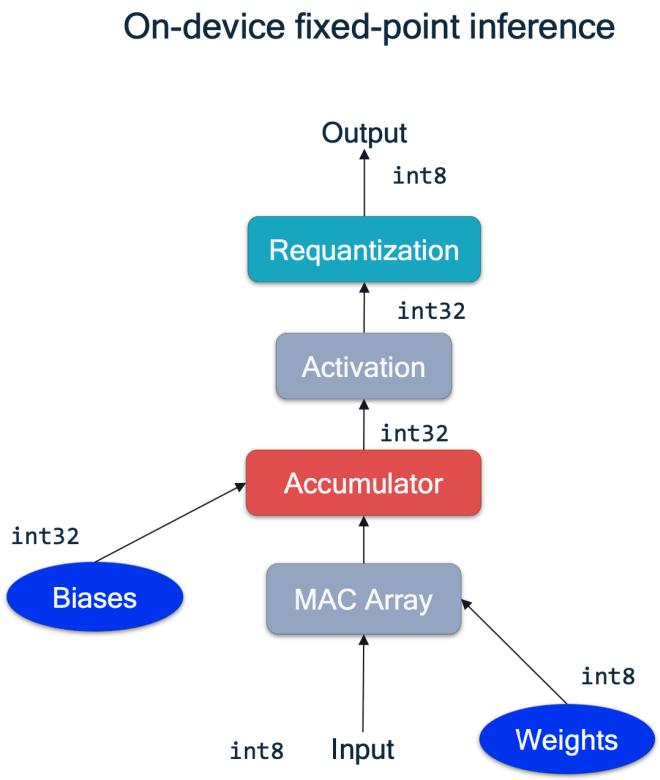
- We simulate fixed-point operations with floating-point numbers using general purpose hardware (e.g., CPU, GPU)
- This simulation is achieved by introducing simulated quantization operations (quantizers) to the compute graph.
- Quantization simulation benefits:
  - Enables GPUs acceleration
  - No need for dedicated kernel
  - Test various quantization option and bit-widths

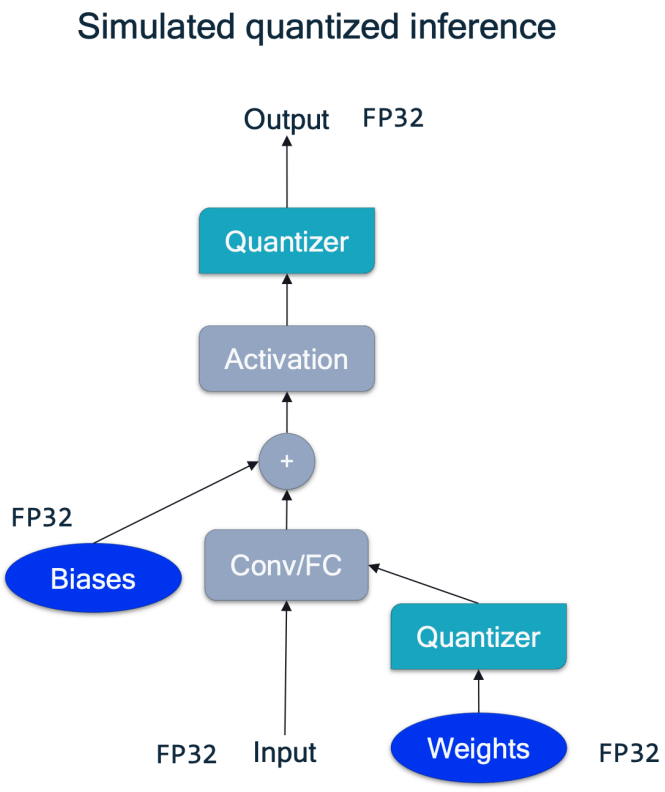


On-device vs. Simulated Quantization

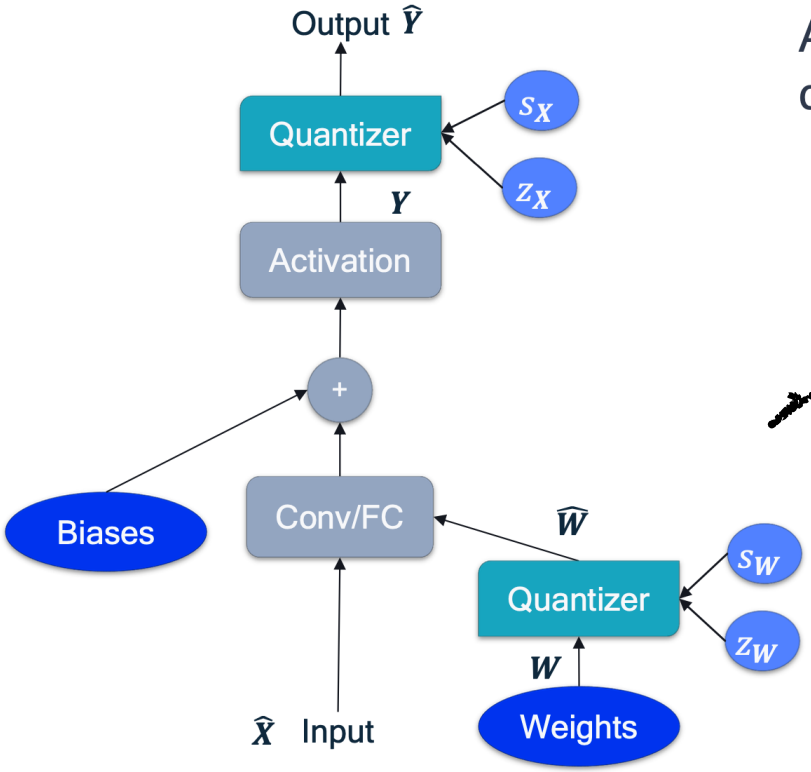


On-device vs. Simulated Quantization





Quantizers Operations



Assuming asymmetric quantization the quantization operation applied to input tensor $\boldsymbol{X}$:

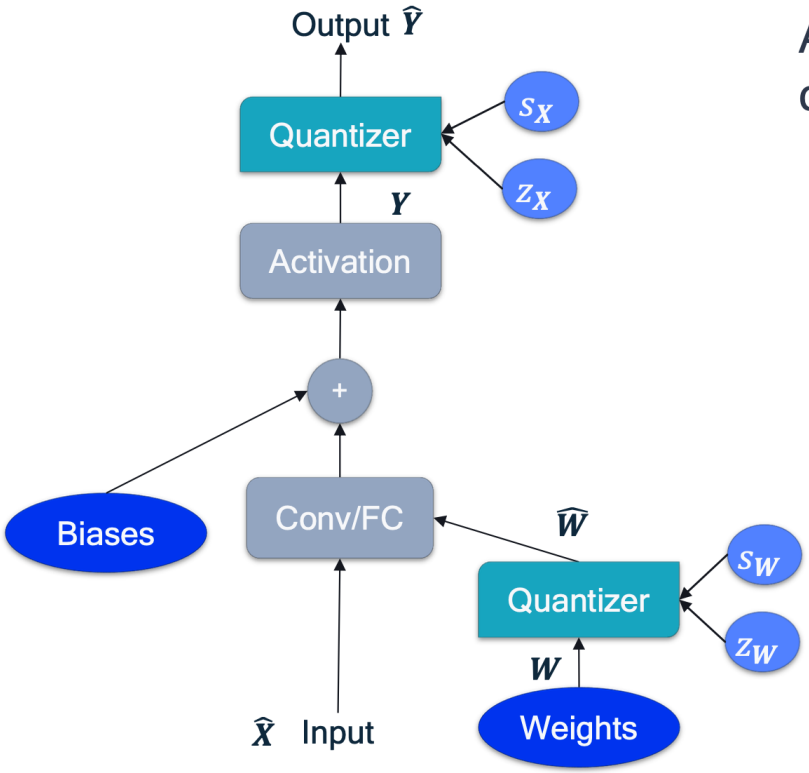
$$\boldsymbol{X}_{\text{int}} = \text{clip}\left(\text{round}\left(\frac{\boldsymbol{X}}{s}\right) + z, \min = 0, \max = 2^b - 1\right)$$

$$\hat{\boldsymbol{X}} = s\,(\boldsymbol{X}_{\text{int}} - z)$$

de-quantization

quantization equation

Quantizers Operations



Assuming asymmetric quantization the quantization operation applied to input tensor $\boldsymbol{X}$:

$$\boldsymbol{X}_{\text{int}} = \text{clip}\left(\text{round}\left(\frac{\boldsymbol{X}}{s}\right) + z, \text{min} = 0, \text{max} = 2^b - 1\right)$$

$$\hat{\boldsymbol{X}} = s\,(\boldsymbol{X}_{\text{int}} - z)$$

Example using $b = 4$: 4 bits

$$X = \begin{pmatrix} 0.41 & 0.0 \\ 0.8 & -0.5 \end{pmatrix}$$

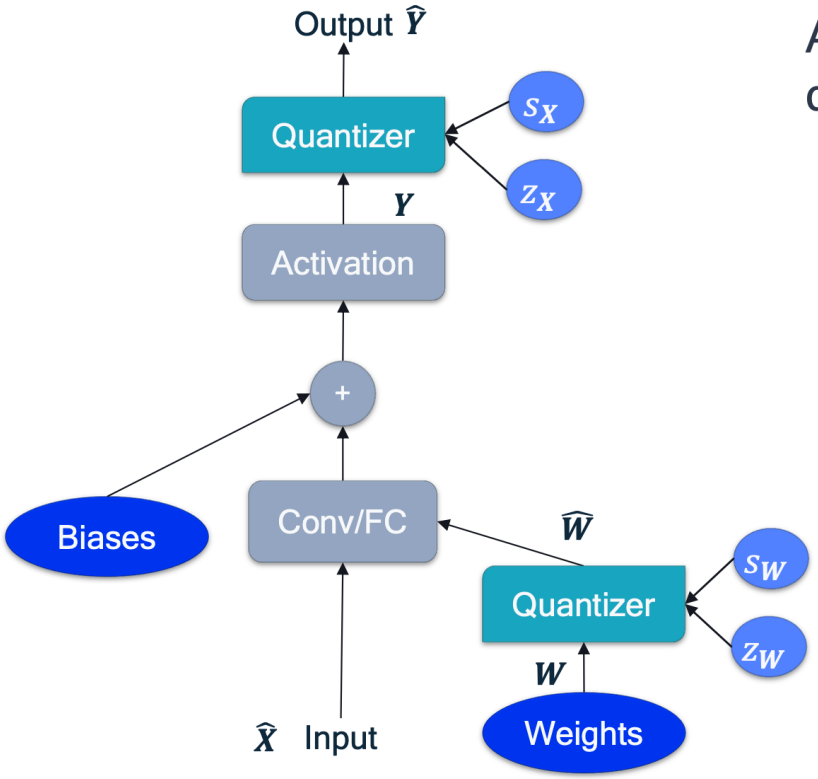
$$s = \frac{1}{15} = 0.067$$

because of 4 bits. (max possible value in 4 bits)

$$z = \text{round}\left(\frac{0.5}{0.067}\right) = 8$$

arbitrarily choosen

Quantizers Operations



Assuming asymmetric quantization the quantization operation applied to input tensor $\boldsymbol{X}$:

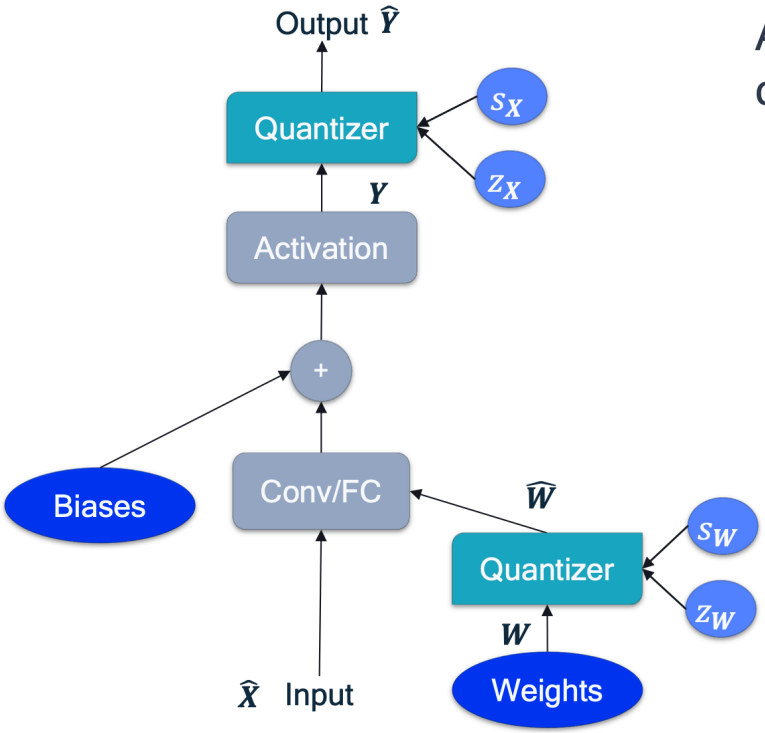
$$\boldsymbol{X}_{\text{int}} = \text{clip}\left(\text{round}\left(\frac{\boldsymbol{X}}{s}\right) + z, \text{min} = 0, \text{max} = 2^b - 1\right)$$

$$\hat{\boldsymbol{X}} = s\,(\boldsymbol{X}_{\text{int}} - z) \qquad \text{round}\left(\frac{X}{s}\right) + z = \begin{pmatrix} 14 & 8 \\ 20 & 0 \end{pmatrix}$$

Example using $b = 4$: $s = 0.067$ $z = 8$

$$\frac{X}{s} = \begin{pmatrix} 6.15 & 0.0 \\ 12 & -7.5 \end{pmatrix}$$

Quantizers Operations



Assuming asymmetric quantization the quantization operation applied to input tensor $\boldsymbol{X}$:

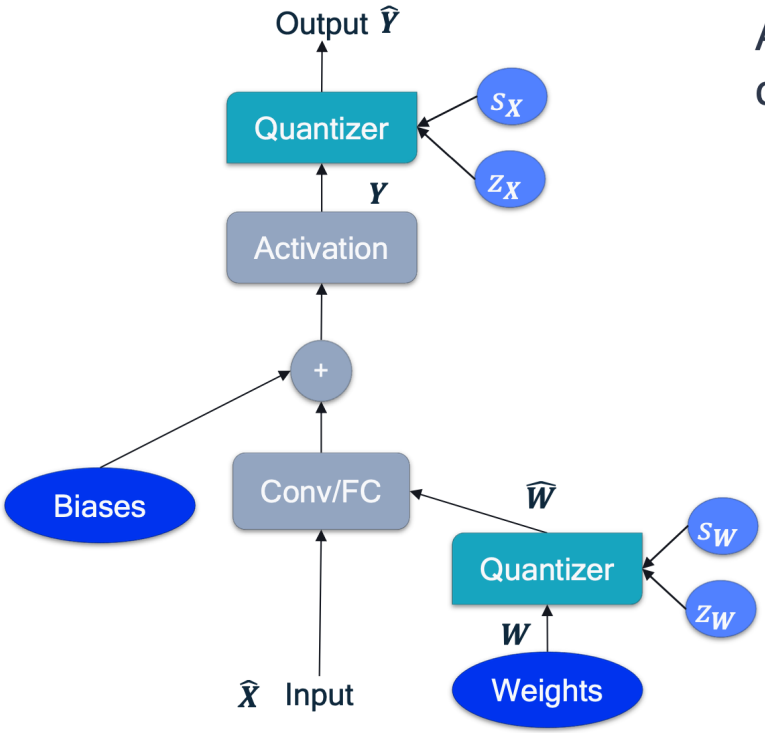
$$\boldsymbol{X}_{\text{int}} = \text{clip}\left(\text{round}\left(\frac{\boldsymbol{X}}{s}\right) + z, \text{min} = 0, \text{max} = 2^b - 1\right)$$

$$\hat{\boldsymbol{X}} = s\,(\boldsymbol{X}_{\text{int}} - z) \qquad \text{round}\left(\frac{X}{s}\right) + z = \begin{pmatrix} 14 & 8 \\ 20 & 0 \end{pmatrix}$$

Example using $b = 4$: $\quad s = 0.067 \quad z = 8$

$$\text{round}\left(\frac{X}{s}\right) + z = \begin{pmatrix} 14 & 8 \\ 20 & 0 \end{pmatrix}$$

Quantizers Operations



Assuming asymmetric quantization the quantization operation applied to input tensor $\boldsymbol{X}$:

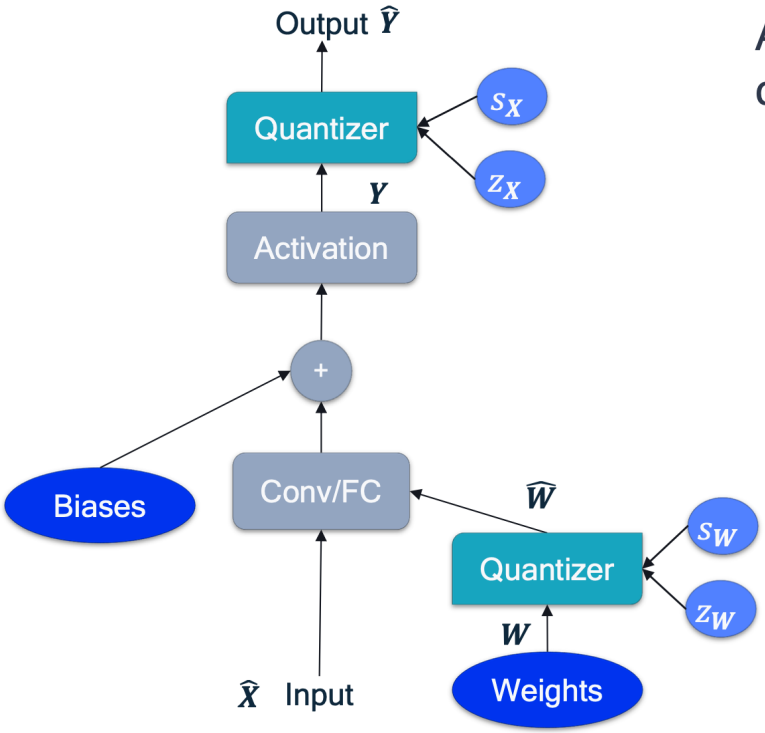
$$\boldsymbol{X}_{\text{int}} = \text{clip}\left(\text{round}\left(\frac{\boldsymbol{X}}{s}\right) + z, \text{min} = 0, \text{max} = 2^b - 1\right)$$

$$\hat{\boldsymbol{X}} = s\,(\boldsymbol{X}_{\text{int}} - z) \qquad \text{round}\left(\frac{X}{s}\right) + z = \begin{pmatrix} 14 & 8 \\ 20 & 0 \end{pmatrix}$$

Example using $b = 4$: $s = 0.067$ $z = 8$

$$\text{round}\left(\frac{X}{s}\right) + z = \begin{pmatrix} 14 & 8 \\ 20 & 0 \end{pmatrix} \xrightarrow{\text{clip}} \begin{pmatrix} 14 & 8 \\ \mathbf{15} & 0 \end{pmatrix}$$

Quantizers Operations



Assuming asymmetric quantization the quantization operation applied to input tensor $\boldsymbol{X}$:

$$\boldsymbol{X}_{\text{int}} = \text{clip}\left(\text{round}\left(\frac{\boldsymbol{X}}{s}\right) + z, \text{min} = 0, \text{max} = 2^b - 1\right)$$

$$\hat{\boldsymbol{X}} = s\,(\boldsymbol{X}_{\text{int}} - z) \qquad \text{round}\left(\frac{X}{s}\right) + z = \begin{pmatrix} 14 & 8 \\ 20 & 0 \end{pmatrix}$$

Example using $b = 4$: $s = 0.067$ $z = 8$

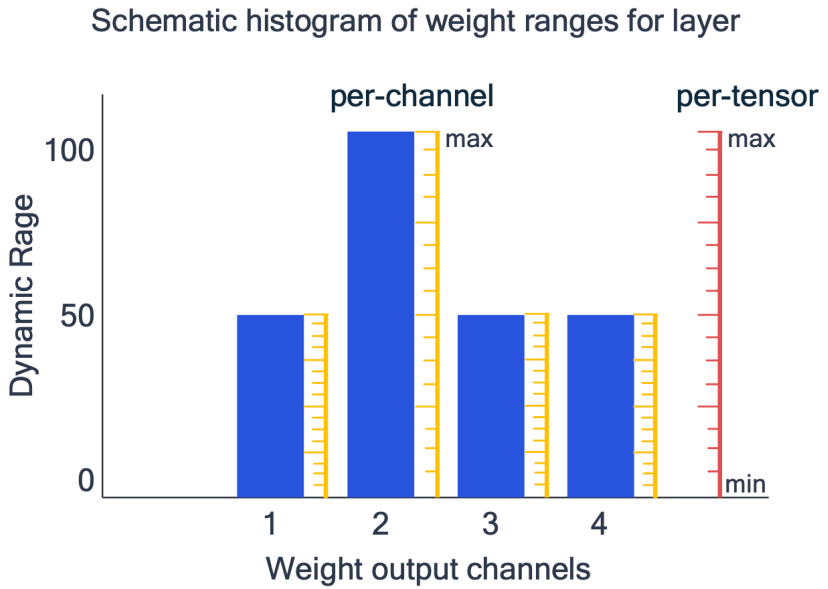
$$\text{round}\left(\frac{X}{s}\right) + z = \begin{pmatrix} 14 & 8 \\ 20 & 0 \end{pmatrix} \xrightarrow{\text{clip}} \begin{pmatrix} 14 & 8 \\ \mathbf{15} & 0 \end{pmatrix}$$

de-quantize

$$X = \begin{pmatrix} 0.41 & 0.0 \\ 0.8 & -0.5 \end{pmatrix} \qquad \hat{X} = \begin{pmatrix} 0.4 & 0.0 \\ 0.47 & -0.53 \end{pmatrix}$$

Per-channel vs Per-tensor Quantization of Weights

- **Per-tensor quantization** most supported by fixed-point accelerators
- **Per-channel quantization** better utilizes the quantization grid
- Per-channel quantization increasingly popular for weights
- Check for HW support

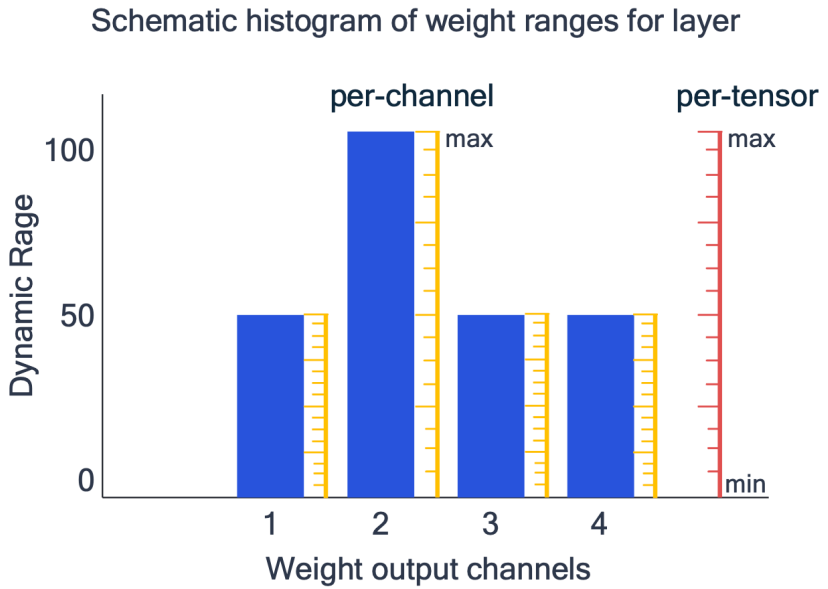


per-channel is more fine grained.

we have more pwer to control the quantization

Per-channel vs Per-tensor Quantization of Weights

- **Per-tensor quantization** most supported by fixed-point accelerators
- **Per-channel quantization** better utilizes the quantization grid
- Per-channel quantization increasingly popular for weights
- Check for HW support



Per-tensor vs per-channel

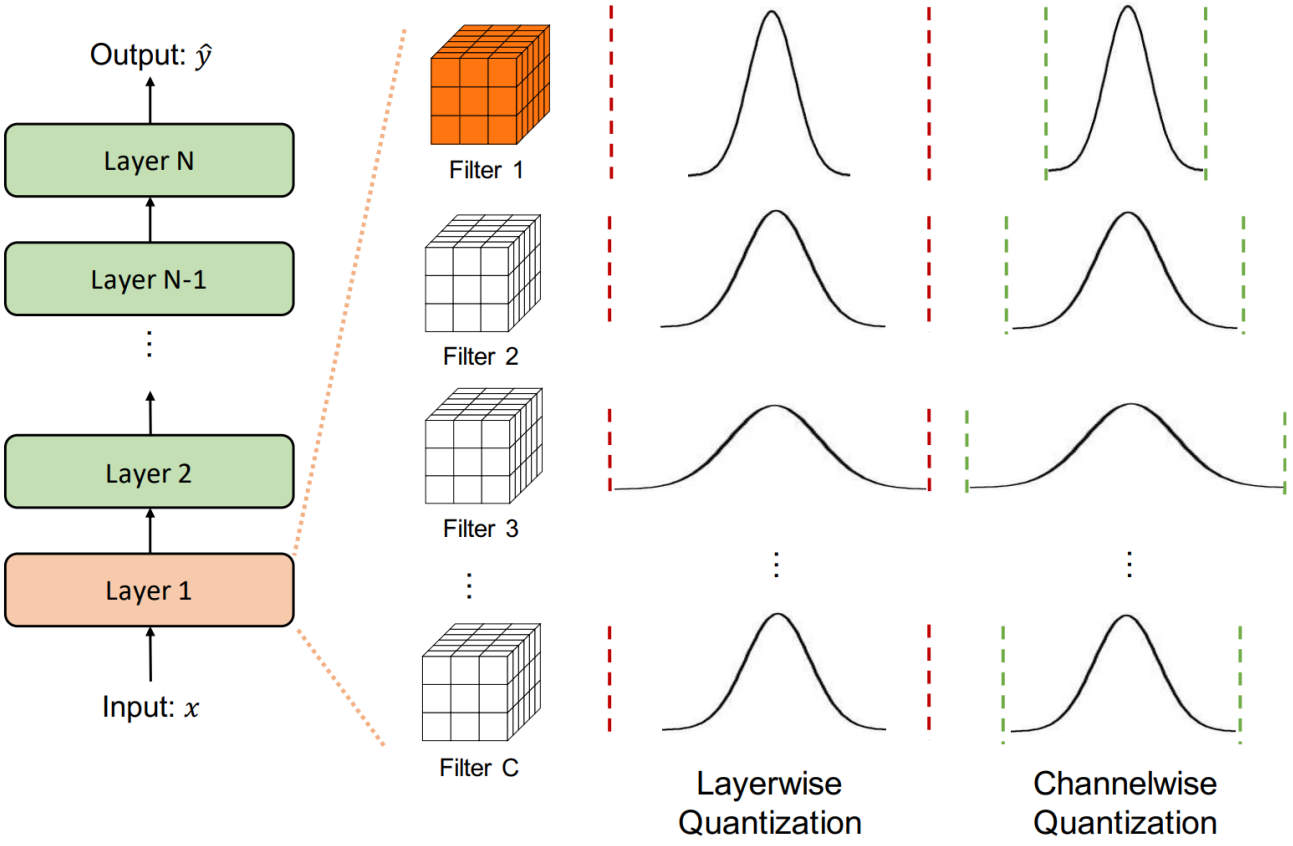


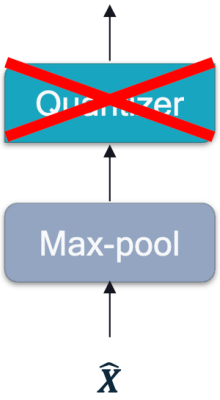
Image source: A Survey of Quantization Methods for Efficient Neural Network Inference – A. Gholami et al.

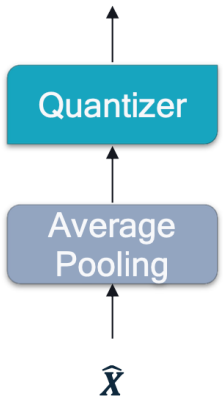
Per-channel is useful when channels have different ranges

Note: per-channel quantization does not require special hardware when applied to **weights**

How to simulate quantization in common DL Layers

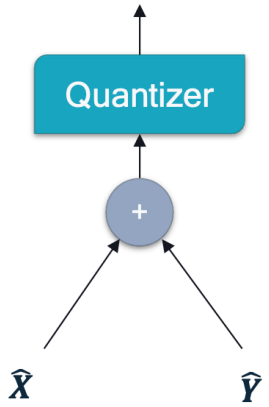
when should we add quantization step while simulating it.

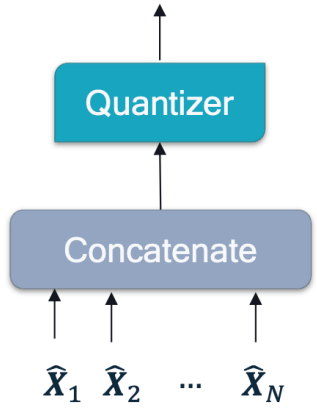




We can tie input and output quantizers

Elementwise Add

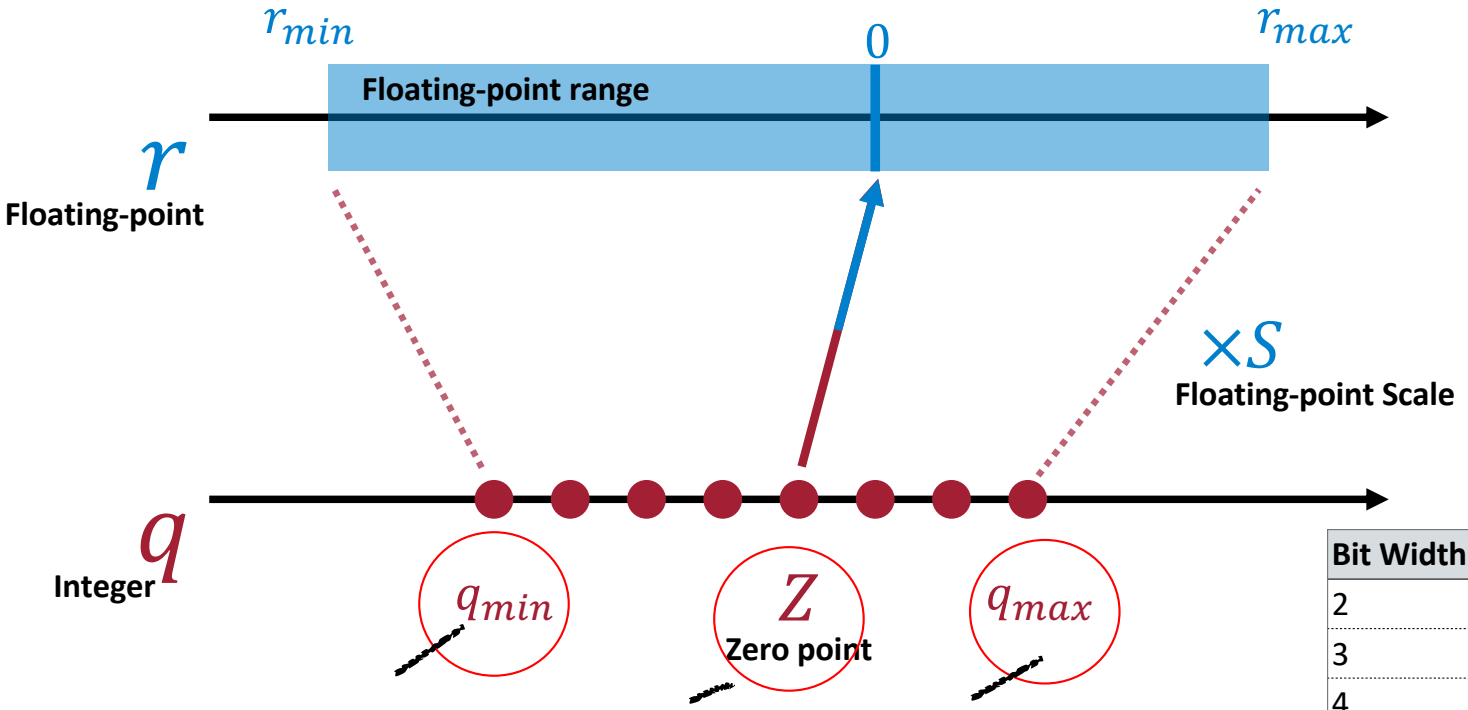




How to choose the right quantization Parameters?

Linear Quantization

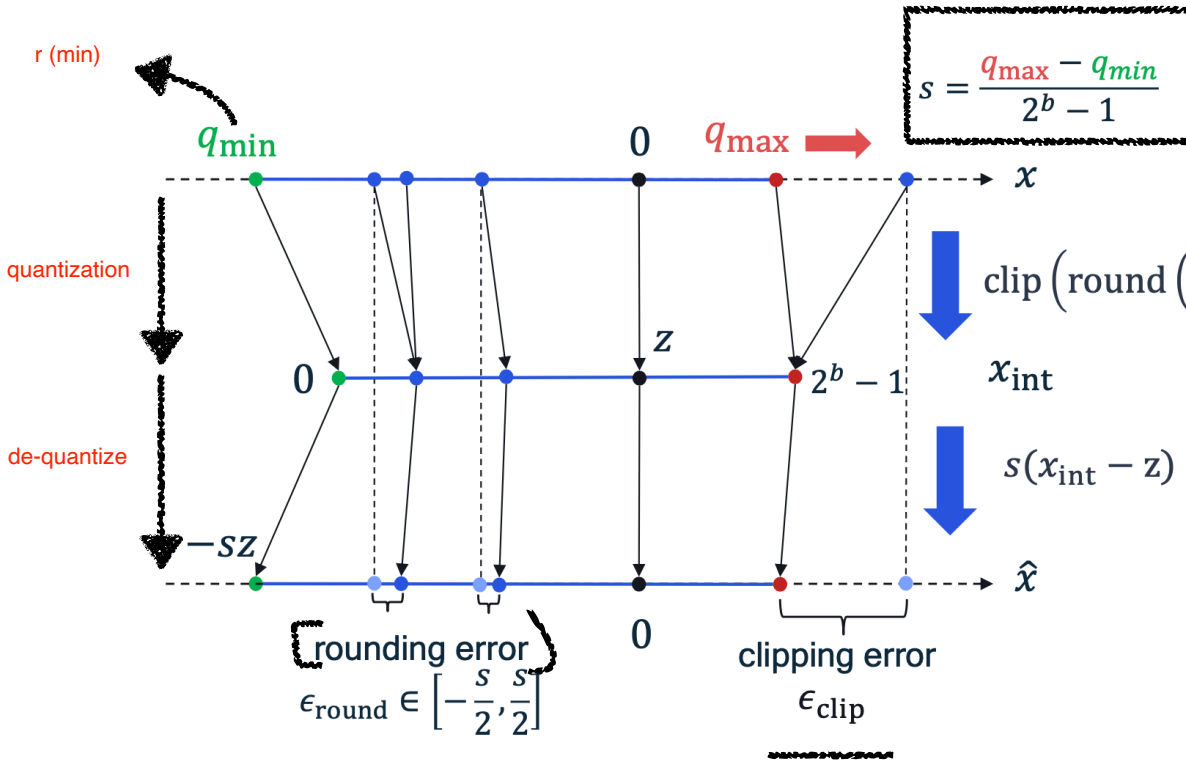
- An affine mapping of integers to real numbers $r = S(q - Z)$



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

Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference [Jacob *et al.*, CVPR 2018]

Sources of Quantization Error

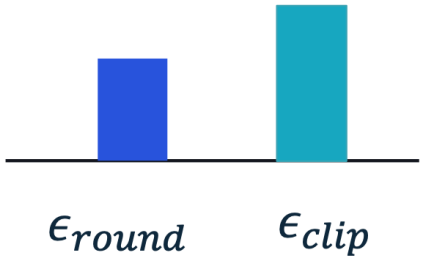


$$s = \frac{q_{max} - q_{min}}{2^b - 1}$$

$$\epsilon_{quant} = \sum_{data} \epsilon_{round} + \epsilon_{clip}$$

$$\text{clip}\left(\text{round}\left(\frac{x}{s}\right) + z, 0, 2^b - 1\right)$$

$$s(x_{\text{int}} - z)$$

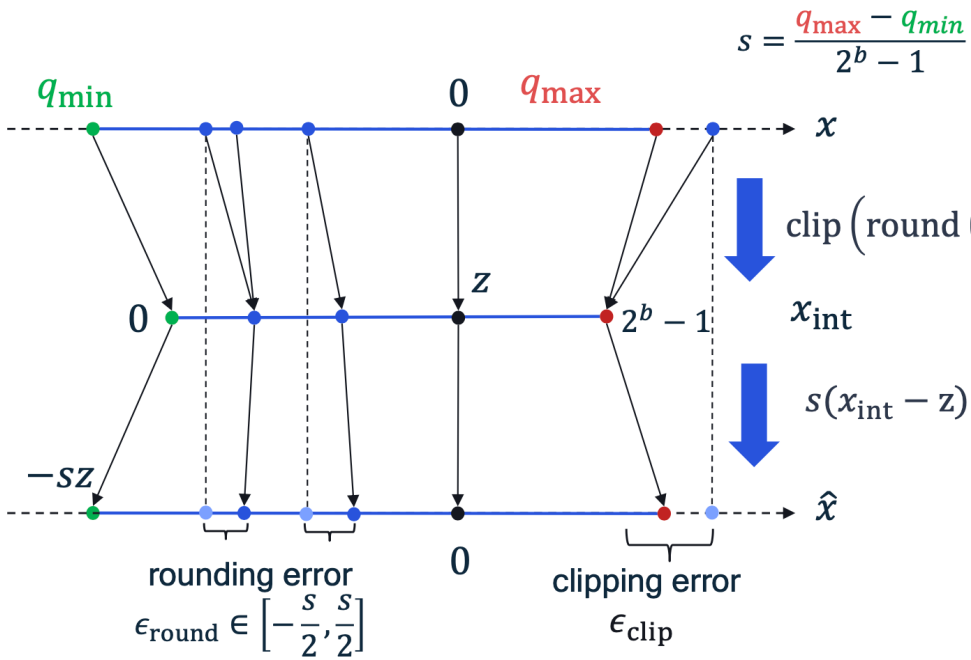


clipping error can be reduced by increasing the quantisation range

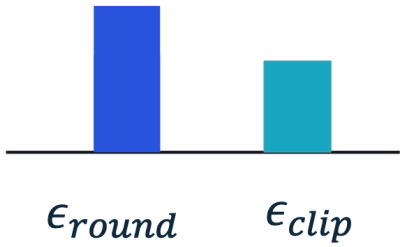
however by doing so the rounding error will increase why?

and if we decrease the rounding error, it will increase the clipping error

Sources of Quantization Error



$$\epsilon_{quant} = \sum_{data} \epsilon_{round} + \epsilon_{clip}$$



few methods to find out qmin and qmax

Quantization Parameters Setting Methods

- Min-max range:

$$q_{\min} = \min \boldsymbol{X}$$
$$q_{\max} = \max \boldsymbol{X}$$

- Optimization-based methods:

we want to find optimal qmin and qmax so as to minimise this error (loss) between actual values and quantized values.

$$\text{argmin}_{q_{\min}, q_{\max}} \ell\left(\boldsymbol{X}, \widehat{\boldsymbol{X}}(q_{\min}, q_{\max})\right)$$

MSE Cross-entropy

- Batch-Norm Based [1]:

$$q_{\min} = \min\left(\boldsymbol{\beta} - \alpha\boldsymbol{\gamma}\right)$$
$$q_{\max} = \max\left(\boldsymbol{\beta} + \alpha\boldsymbol{\gamma}\right)$$

$$\text{BatchNorm}\left(\boldsymbol{z}_k\right) = \boldsymbol{\gamma}_k \frac{\boldsymbol{z}_k - \boldsymbol{\mu}_k}{\sqrt{\boldsymbol{\sigma}_k + \epsilon}} + \boldsymbol{\beta}_k$$

[1] Nagel et al, 2019, Data-Free Quantization Through Weight Equalization and Bias Correction

Quantization Setting Methods Ablation Study

Model (FP32 Accuracy)	ResNet18 (69.68)		MobileNetV2 (71.72)	
Bit-width	A8	A6	A8	A6
Min-Max	69.60	68.19	70.96	64.58
MSE	69.59	67.84	71.35	67.55
MSE & X-entropy	**69.60**	**68.91**	**71.36**	**68.85**
BN ($\alpha = 6$)	69.54	68.73	71.32	71.32

Average ImageNet validation accuracy (%) over 5 seeds

Quantization Algorithms

What algorithm to choose to improve accuracy?

Two Approaches

Post-Training Quantization (PTQ)

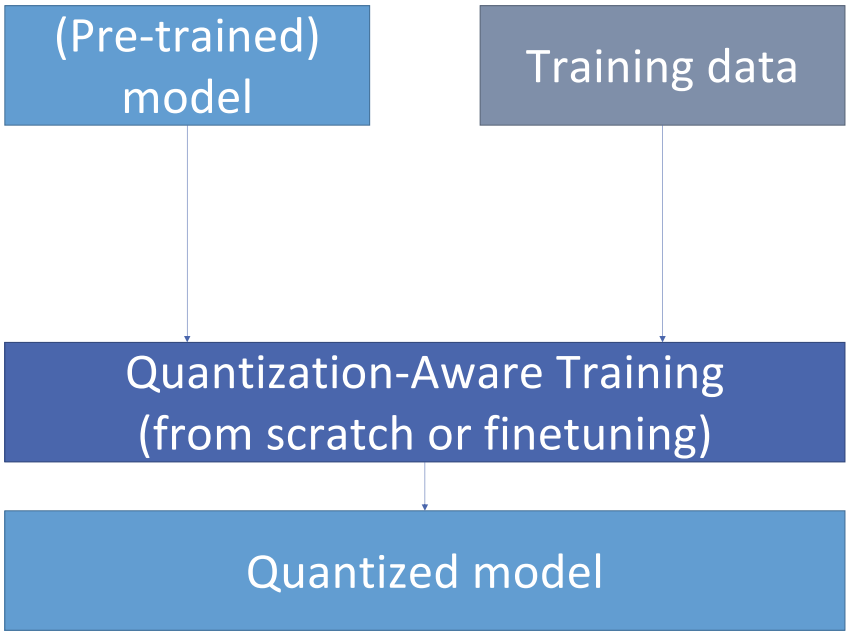
- ✓ Takes a pre-trained network and converts it to a fixed-point network without access to the training pipeline
- ✓ Data-free or small calibration set needed
- ✓ Use though single API call
- ✗ Lower accuracy at lower bit-widths

Quantization-Aware Training (QAT)

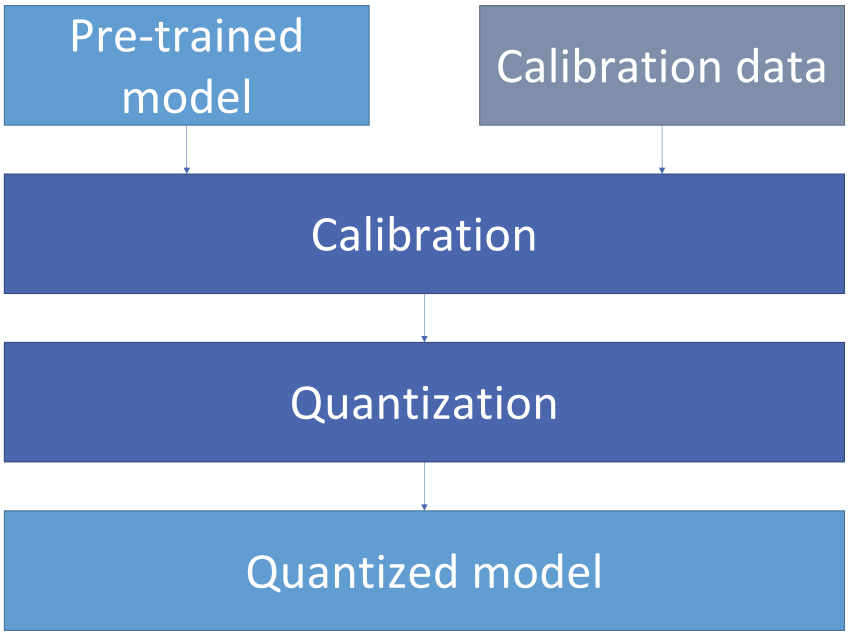
- ✗ Requires access to training pipeline and labelled data
- ✗ Longer training times
- ✗ Hyper-parameter tuning
- ✓ Achieves higher accuracy

When to quantize?

Quantization-Aware Training



Post-Training Quantization



Post-Training Quantization

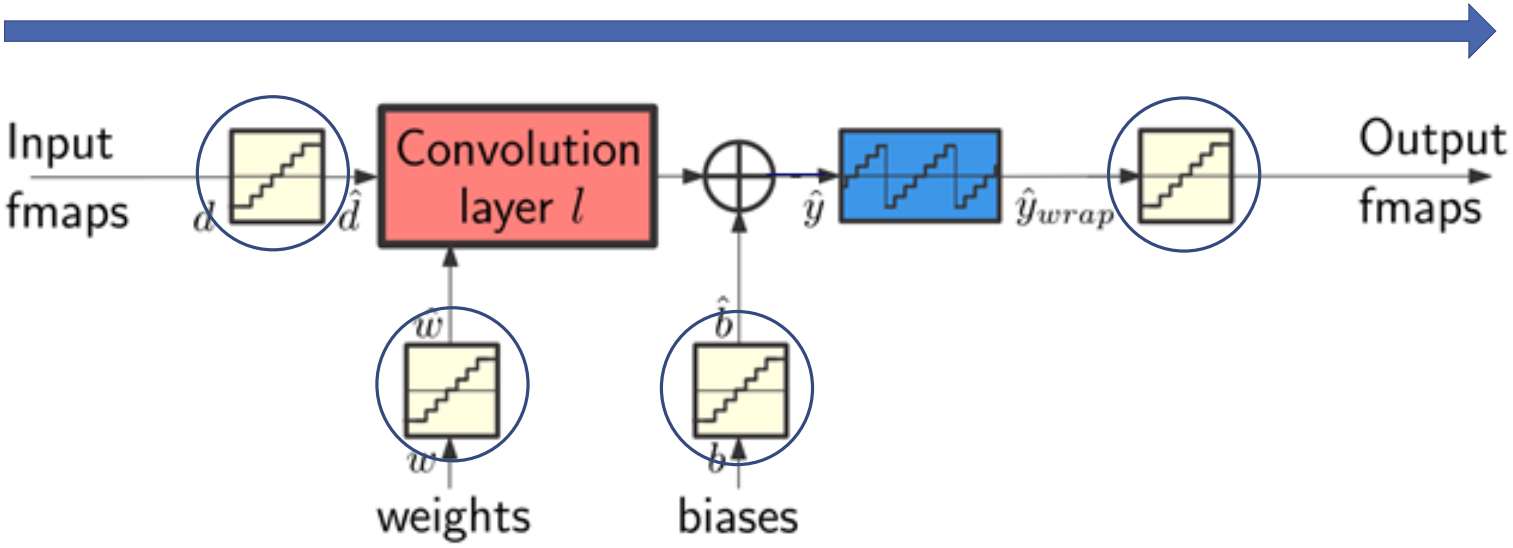
- Typical PTQ procedure:

1. Range analysis using calibration data (subset of training data)
   - To determine step size for activations
   - Step size for weights can be determined without any data

2. Check if quality of final solution is sufficient. If not, do some finetuning (using entire training data)

Quantization-Aware Training

Forward pass: simulated quantization to compute loss function

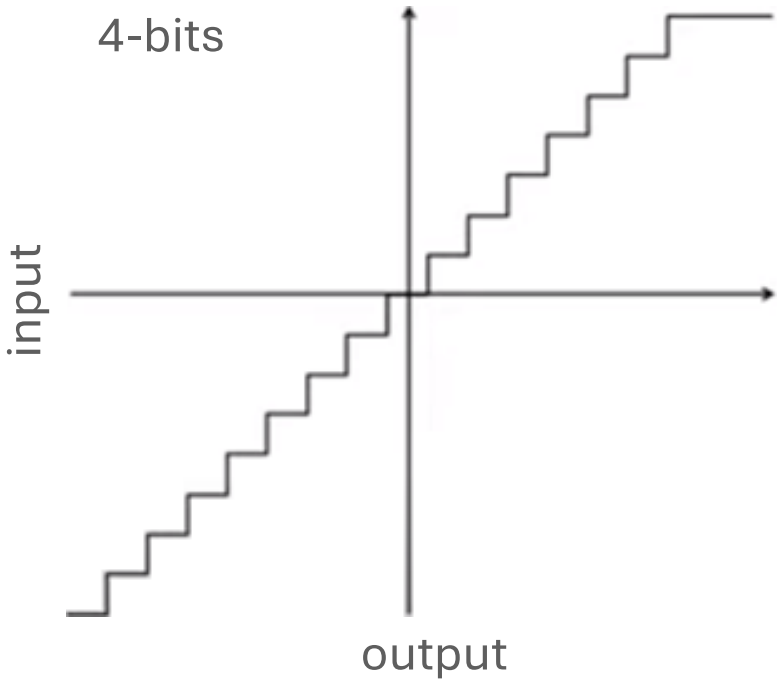


Backward pass: how to back propagate through the quantizers?

Image source: Quantization of deep neural networks for accumulator-constrained processors – B. de Bruin et al.

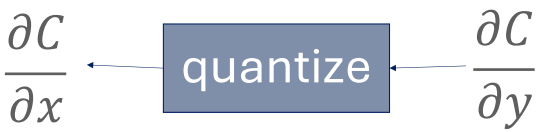
Quantizers: how to backpropagate?

$y = \text{quantize}(x)$



Forward pass

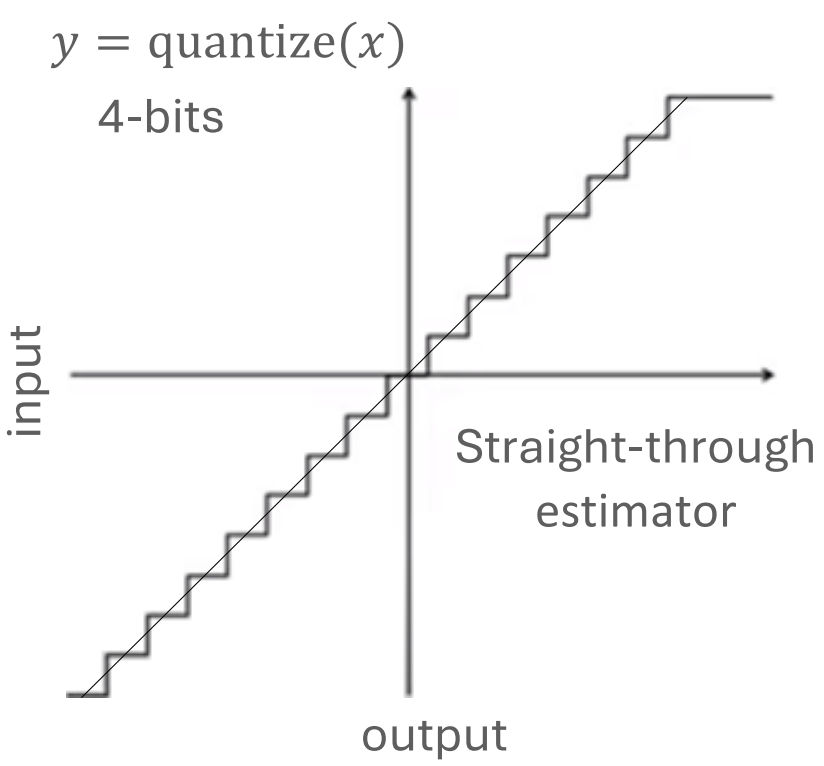
Backward pass



$$\frac{\partial C}{\partial x} = \frac{\partial C}{\partial y}\frac{\partial y}{\partial x}$$

?

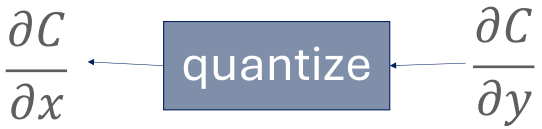
Quantizers: how to backpropagate?



(STE) (Hinton et al., 2012, Bengio, 2013

Forward pass

Backward pass



$$\frac{\partial C}{\partial x} = \frac{\partial C}{\partial y}\frac{\partial y}{\partial x}$$

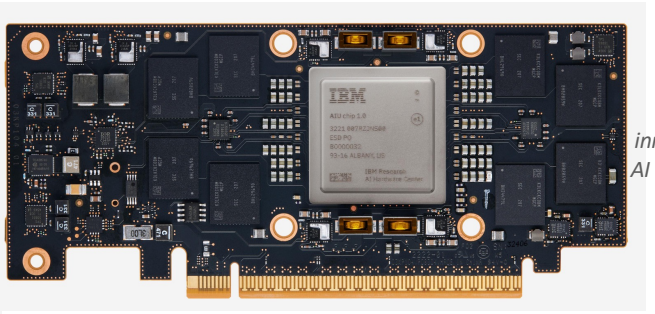
$$\Rightarrow \quad \frac{\partial C}{\partial y}\,1$$

Quantizatin Approaches - Summary

Approach	Training time	Quantization time	Quality of results
Post-Training Quantization	None	1 forward pass	Low
Post-Training Quantization with fine-tuning	Short	Multiple forward & backward passes	Medium
Quantization-Aware Training	Long	Entire training	High

IBM Reduced Precision AI hardware and model quantization

- Specialized AI HW has “low-precision” INT compute engine.
  - INT4 is ~2X-8X more power efficient than FP16. A 4-core Sentient chip* has a peak at 16.5 TOPS/W!



AIU

SoC implements IBM’s innovations in low-precision AI arithmetic and algorithms

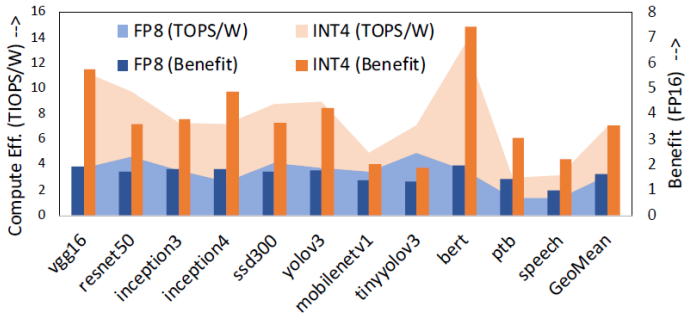
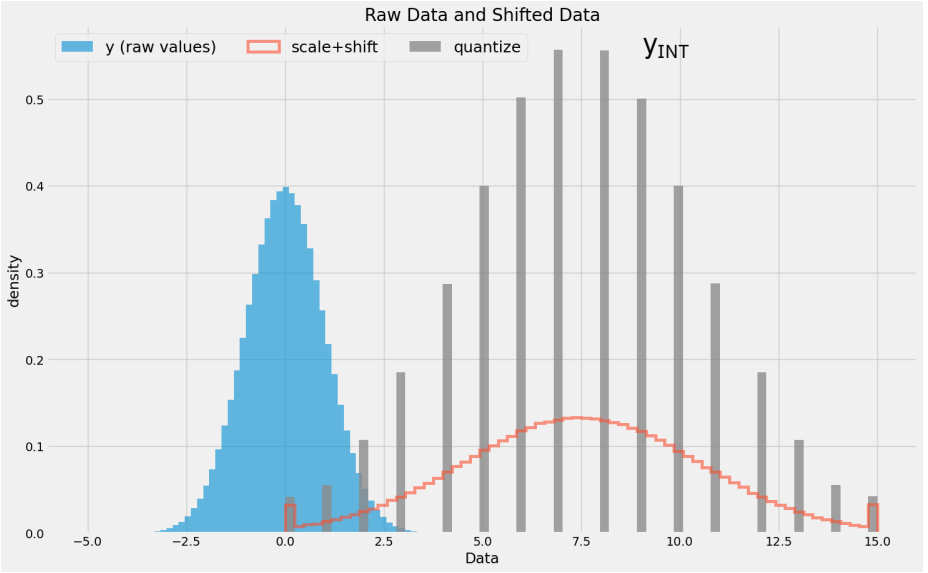


Figure 14: Sustained TOPS/W on 4-core RAPID chip

Venkataramani, Swagath, et al. "RaPiD: AI accelerator for ultra-low precision training and inference." *ISCA* 2021.

- Models needs to be “quantized” to fully utilize the INT4 engine.



$$y_{INT} = \left\lfloor \frac{clamp(y, \alpha_l, \alpha_u) - zp}{scale} \right\rceil$$

$$y_q = y_{INT} * scale + zp$$

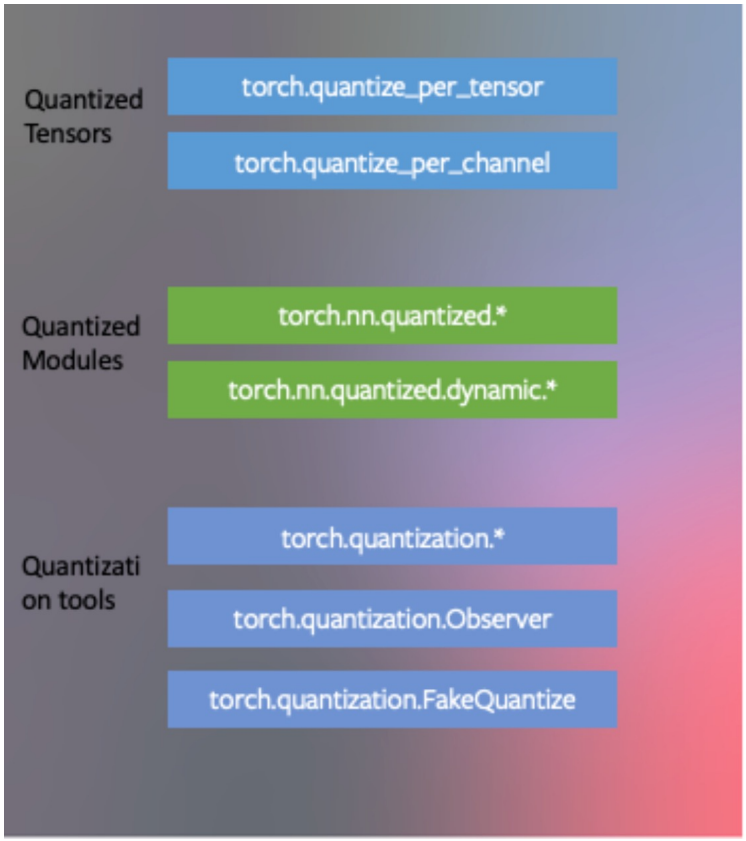
Matrix multiplication of quantized tensors (y_q) can be done in INT space (y_{int}) then scaled.

Quantization Approaches - Summary

Approach	Training time	Quantization time	Quality of results
Post-Training Quantization	None	1 forward pass	Low
Post-Training Quantization with fine-tuning	Short	Multiple forward & backward passes	Medium
Quantization-Aware Training	Long	Entire training	High

Quantization in PyTorch

- PyTorch has data types corresponding to quantized tensors, which share many of the features of tensors.
- One can write kernels with quantized tensors, much like kernels for floating point tensors to customize their implementation. PyTorch supports quantized modules for common operations as part of the torch.nn.quantized and torch.nn.quantized.dynamic na me-space.
- Quantization is compatible with the rest of PyTorch: quantized models are traceable and scriptable. The quantization method is virtually identical for both server and mobile backends. One can easily mix quantized and floating-point operations in a model.
- Mapping of floating-point tensors to quantized tensors is customizable with user defined observer/fake-quantization blocks. PyTorch provides default implementations that should work for most use cases.



https://pytorch.org/blog/introduction-to-quantization-on-pytorch/

PTQ in PyTorch

- **Post-Training Static Quantization in PyTorch**
  - Post-training static quantization involves the following steps:

1. **Model Preparation**: Modify the model to support quantization (insert quantization/dequantization layers).
2. **Calibration**: Run a calibration step with a representative dataset to gather statistics about the model (like the range of different tensors).
3. **Conversion**: Convert the model to use quantized weights and activations based on the calibration data.

Quantized dtypes and quantization schemes

- torch.qscheme — Type to describe the quantization scheme of a tensor. Supported types:
  - torch.per_tensor_affine — per tensor, asymmetric
  - torch.per_channel_affine — per channel, asymmetric
  - torch.per_tensor_symmetric — per tensor, symmetric
  - torch.per_channel_symmetric — per tensor, symmetric
- torch.dtype — Type to describe the data. Supported types:
  - torch.quint8 — 8-bit unsigned integer
  - torch.qint8 — 8-bit signed integer
  - torch.qint32 — 32-bit signed integer

https://glaringlee.github.io/quantization.html

PTQ Example in PyTorch (1/2)

- Step 1: Environment Setup
  - First, ensure you have PyTorch installed. You might need the torchvision package for dataset and model utilities.
  - Then, import the necessary libraries

```
!pip install torch torchvision
```

```
import torch
import torchvision.models as models
import torch.quantization
```

- Step 2: Load and Prepare a Model
  - For this example, let's use a pre-trained model like ResNet18
  - Next, we specify the quantization configuration for the model. fbgemm is one of the backends supported for x86. for ARM qnnpack is the backend to use.
  - Prepare the model for quantiztaion

```
model = models.resnet18(pretrained=True)
model.eval()  # Set the model to inference mode
```

```
model.qconfig = torch.quantization.get_default_qconfig('fbgemm')
print(model.qconfig)
```

```
torch.quantization.prepare(model, inplace=True)
```

PTQ Example in PyTorch (2/2)

- Step 3: Calibration
  - Now, calibrate the model with representative data. This step adjusts the internal quantization parameters (scale and zero-point) for optimal accuracy.
  - Assuming you have a DataLoader data_loader for your dataset

```
def calibrate_model(model, data_loader):
    model.eval()
    with torch.no_grad():
        for images, _ in data_loader:
            model(images)
```

```
# Call the calibrate_model function with your data_loader
calibrate_model(model, data_loader)
```

- Step 4: Convert to Quantized Model
  - After calibration, convert the model to a quantized version

```
quantized_model = torch.quantization.convert(model, inplace=True)
```

- Step 5: Save or Use the Quantized Model
  - The model is now quantized and can be saved or directly used for inference.

```
# Saving the model
torch.save(quantized_model.state_dict(), 'quantized_model.pth')

# Load and use it for inference
quantized_model.load_state_dict(torch.load('quantized_model.pth'))
quantized_model.eval()
```

QAT Example in PyTorch (1/2)

- Step 1: Environment Setup
  - Ensure PyTorch and torchvision are installed, then import the necessary libraries

```
import torch
import torchvision.models as models
import torch.quantization
```

- Step 2: Load and Modify the Model for QAT
  - fuse_model is a step specific to QAT, where we fuse certain layers (like Convolution + BatchNorm + ReLU) before quantization, as it can improve model performance.

```
model = models.resnet18(pretrained=True)
model.eval()  # Set the model to inference mode

# Modify the model for QAT
model.fuse_model()  # Fuses modules like Conv, BN, ReLU
model.qconfig = torch.quantization.get_default_qat_qconfig('fbgemm')

# Prepare the model for QAT
torch.quantization.prepare_qat(model, inplace=True)
```

QAT Example in PyTorch (2/2)

- Step 3: Fine-Tuning the Model with QAT
  - This involves training it for a few more epochs. During this process, the model weights and activations are simulated as quantized, which helps the model adapt to the quantization effects.

```
# Fine-tuning
# Note: You should adjust the learning rate, number of epochs, etc., based on your s
optimizer = torch.optim.SGD(model.parameters(), lr=0.0001)
criterion = torch.nn.CrossEntropyLoss()

model.train()
for epoch in range(num_epochs):
    for images, labels in data_loader:  # Assuming you have a data_loader
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

- Step 4: Convert to Quantized Model
  - After fine-tuning, convert the model to a fully quantized version for inference.

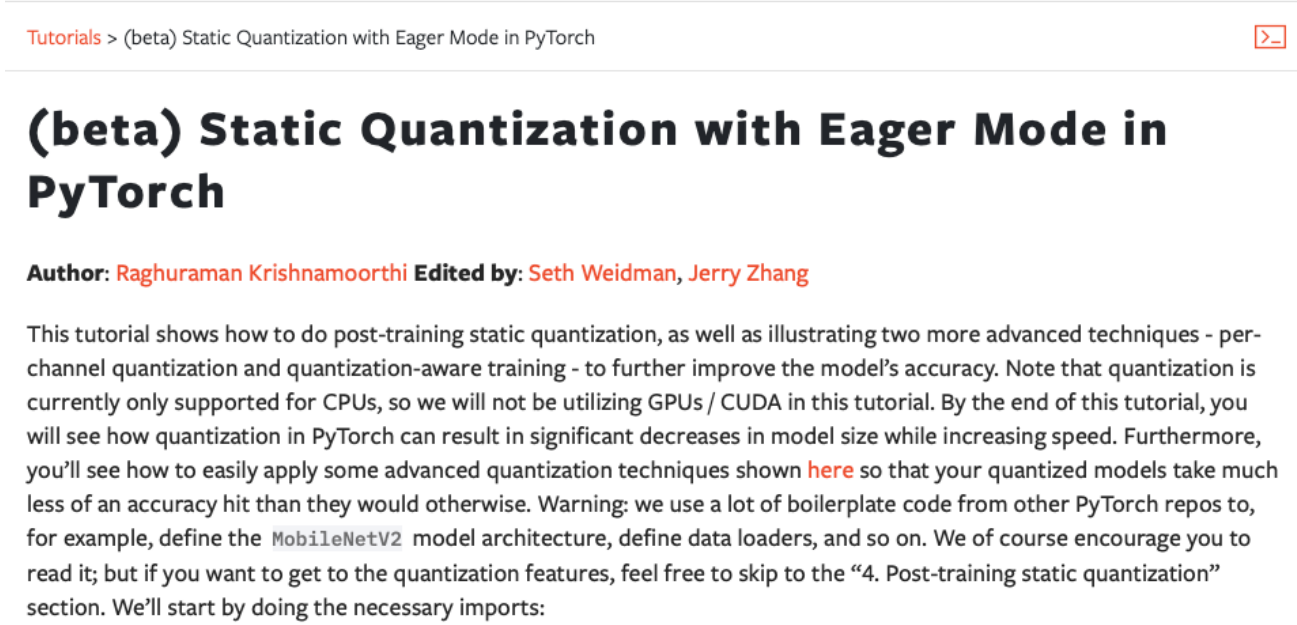
```
model.eval()
quantized_model = torch.quantization.convert(model, inplace=False)
```

- Step 5: Save or Use the Quantized Model
  - The model is now quantized and can be saved or directly used for inference.

```
# Saving the model
torch.save(quantized_model.state_dict(), 'qat_quantized_model.pth')

# Load and use it for inference
quantized_model.load_state_dict(torch.load('qat_quantized_model.pth'))
quantized_model.eval()
```

Static Quantization with Eager Mode in PyTorch Tutorial



Tutorials > (beta) Static Quantization with Eager Mode in PyTorch

(beta) Static Quantization with Eager Mode in PyTorch

Author: Raghuraman Krishnamoorthi **Edited by**: Seth Weidman, Jerry Zhang

This tutorial shows how to do post-training static quantization, as well as illustrating two more advanced techniques - per-channel quantization and quantization-aware training - to further improve the model's accuracy. Note that quantization is currently only supported for CPUs, so we will not be utilizing GPUs / CUDA in this tutorial. By the end of this tutorial, you will see how quantization in PyTorch can result in significant decreases in model size while increasing speed. Furthermore, you'll see how to easily apply some advanced quantization techniques shown here so that your quantized models take much less of an accuracy hit than they would otherwise. Warning: we use a lot of boilerplate code from other PyTorch repos to, for example, define the `MobileNetV2` model architecture, define data loaders, and so on. We of course encourage you to read it; but if you want to get to the quantization features, feel free to skip to the "4. Post-training static quantization" section. We'll start by doing the necessary imports:

https://pytorch.org/tutorials/advanced/static_quantization_tutorial.html

References

1. A White Paper on Neural Network Quantization (https://arxiv.org/pdf/2106.08295.pdf)
2. Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey [Deng et al., IEEE 2020]
3. Computing's Energy Problem (and What We Can Do About it) [Horowitz, M., IEEE ISSCC 2014]
4. Deep Compression [Han et al., ICLR 2016]
5. Neural Network Distiller: https://intellabs.github.io/distiller/algo_quantization.html
6. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference [Jacob et al., CVPR 2018]
7. BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations [Courbariaux et al., NeurIPS 2015]
8. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or −1. [Courbariaux et al., Arxiv 2016]
9. XNOR-Net: ImageNet Classification using Binary Convolutional Neural Networks [Rastegari et al., ECCV 2016]
10. Ternary Weight Networks [Li et al., Arxiv 2016]
11. Trained Ternary Quantization [Zhu et al., ICLR 2017]

Acknowledgments:

Part of this material has been adapted from the MIT efficient Deep Learning class by Prof. Song Han, Quantization Neural Network white paper (https://arxiv.org/pdf/2106.08295.pdf), and Floran de Putter lecture