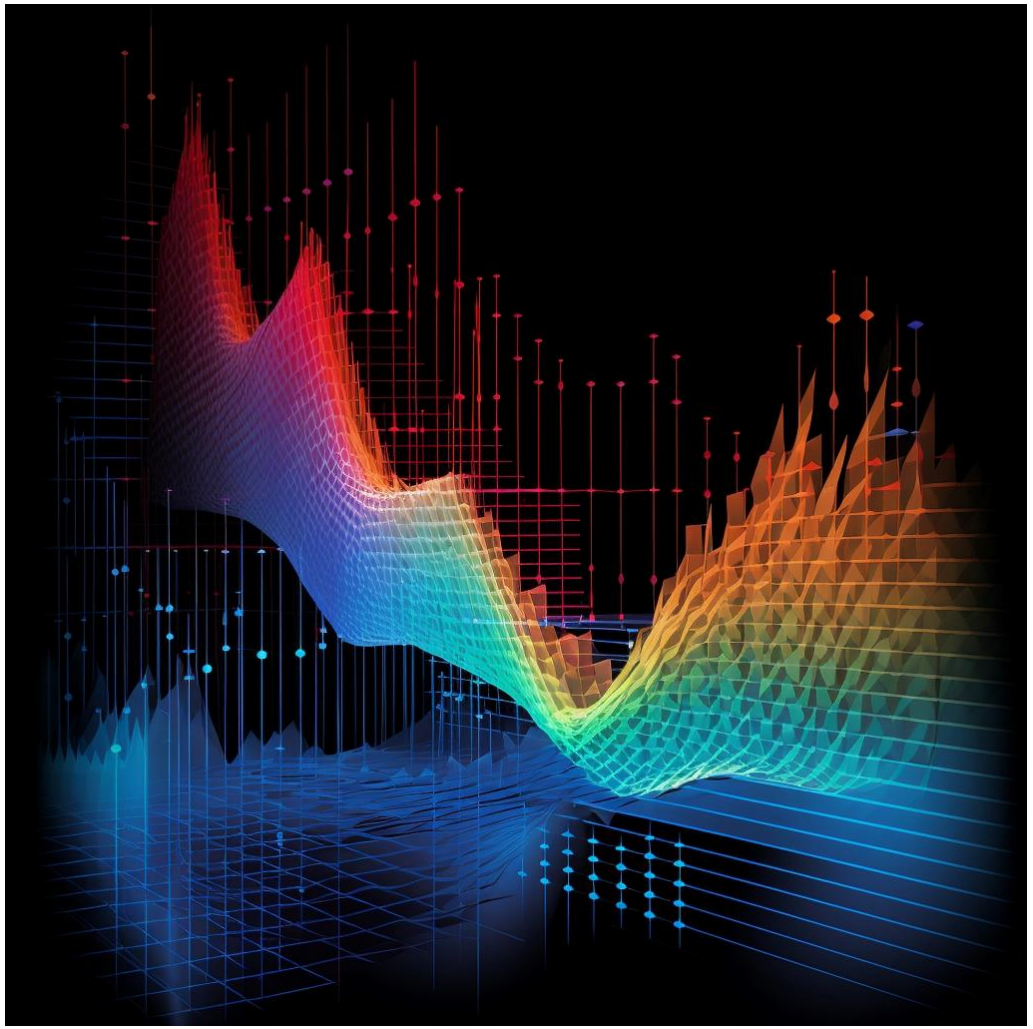


Generated by Midjourney

QUANTIZATION EXPLAINED

Umar Jamil

Outline

- What is quantization
  - Why we need quantization
- Numerical representation
  - Integers
  - Floating-point numbers
- Introduction to quantization
  - Review of how neural networks work
  - Quantization of NN layers
- Types of quantization
  - Asymmetric and symmetric quantization
  - Quantization range
  - Quantization granularity
  - Post-Training Quantization
  - Quantization-Aware Training

Prerequisites

- Basic understanding of neural networks
- High school mathematics

Outline

- What is quantization
  - Why we need quantization
- Numerical representation
  - Integers
  - Floating-point numbers
- Introduction to quantization
  - Review of how neural networks work
  - Quantization of NN layers
- Types of quantization
  - Asymmetric and symmetric quantization
  - Quantization range
  - Quantization granularity
  - Post-Training Quantization
  - Quantization-Aware Training

What is quantization?

The problem

- Most modern deep neural networks are made up of billions of parameters. For example, the smallest LLaMA 2 has 7 billion parameters. If every parameter is 32 bit, then we need $\frac{7\times10^9\times32}{8\times10^9} = 28\ \text{GB}$ just to store the parameters on disk.
- When we inference a model, we need to load all its parameters in the memory, this means big models cannot be loaded easily on a standard PC or a smart phone.
- Just like humans, computers are slow at computing floating-point operations compared to integer operations. Try to do 3×6 and compare it to 1.21×2.897, which one can you compute faster?

The solution

- Quantization aims to reduce the total amount of bits required to represent each parameter, usually by converting floating-point numbers into integers. This way, a model that normally occupies 10 GB can be "compressed" to less than 1 GB (depending on the type of quantization used). **Please note:** quantization doesn't mean truncating/rounding. We don't just round up or down all the floating-point numbers! We will see later how it works
- Quantization can also speed up computation, as working with smaller data types is faster (for example multiplying two integers is faster than multiplying two floating point numbers).

Advantages of quantization

- Less memory consumption when loading models (important for devices like smart phones)
- Less inference time due to simpler data types
- Less energy consumption, because inference takes less computation overall.

Outline

- What is quantization
  - Why we need quantization
- Numerical representation
  - Integers
  - Floating-point numbers
- Introduction to quantization
  - Review of how neural networks work
  - Quantization of NN layers
- Types of quantization
  - Asymmetric and symmetric quantization
  - Quantization range
  - Quantization granularity
  - Post-Training Quantization
  - Quantization-Aware Training

How are integers represented in the CPU (or GPU)?

- Computers use a fixed number of bits to represent any piece of data (a number, a character or a pixel's color). A bit string made up of *n* bits can represent up to 2^N distinct numbers. For example, with 3 bits, we can represent a total of 2^3 distinct numbers. We usually represent numbers in blocks of 8 bits (byte), 16 bits (short), 32 bits (int) or 64 bits (long).

$$110 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 6$$

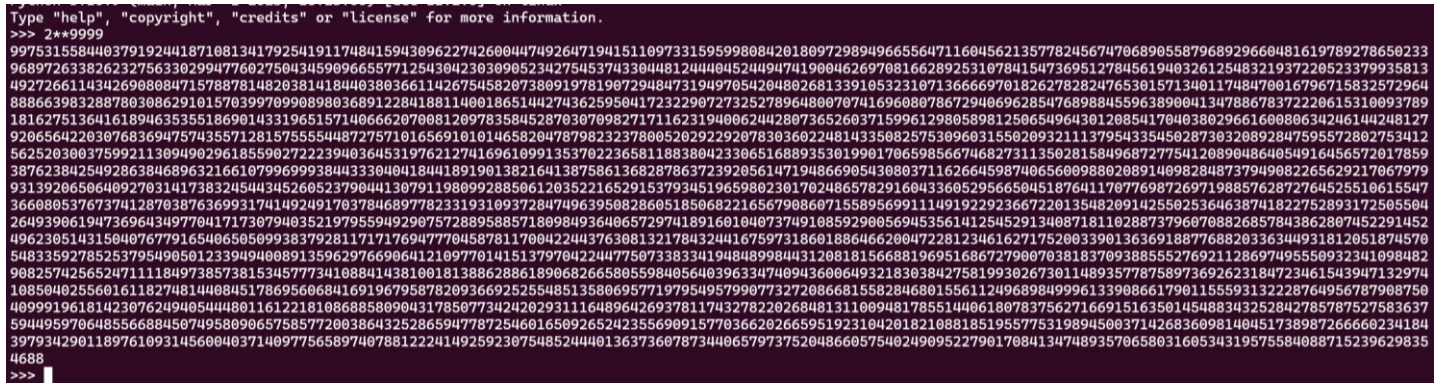
Binary	Number
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

With 3 bits we can represent 2^3 distinct numbers

- In most CPUs integers are represented using the 2's complement: the first bit indicates the sign, while the rest indicate the absolute value of the number (in case it's positive), or its complement in case it's negative. 2's complement also gives a unique representation to the number zero.

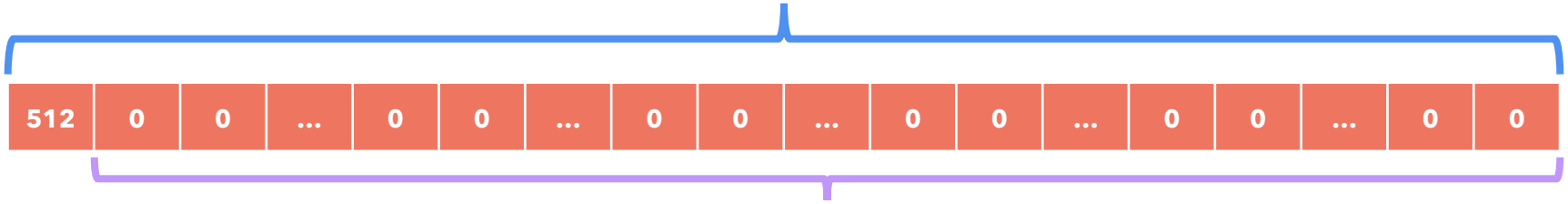
But, but… Python can handle anything!

- Python can represent arbitrary big numbers by using the so-called BigNum arithmetic: each number is stored as an array of digits in base 2^{30}.
  - This is a function of CPython (the Python interpreter), not something built-in in the CPU or the GPU. This means that if we want to perform operations fast, using hardware acceleration provided by CUDA (for example), we are forced to use numbers in fixed format (32 bits usually).



If stored in base 10 digits, it would require an array of 3010 digits

Array of 334 elements, each element is a 32-bit unsigned integer



All elements with index 0… 332 are equal to zero.

$$2^{9999} = 512 \times (2^{30})^{333}$$

```
>>> 512*((2**30)**333) == 2**9999
True
```

How are floating point numbers represented?

Decimal numbers are just numbers that also include negative powers of the base. For example:

$$\mathbf{85.612 = 8 \times 10^{1} + 5 \times 10^{0} + 6 \times 10^{-1} + 1 \times 10^{-2} + 2 \times 10^{-3}}$$

The IEEE-754 standard defines the representation format for floating point numbers in 32 bit.

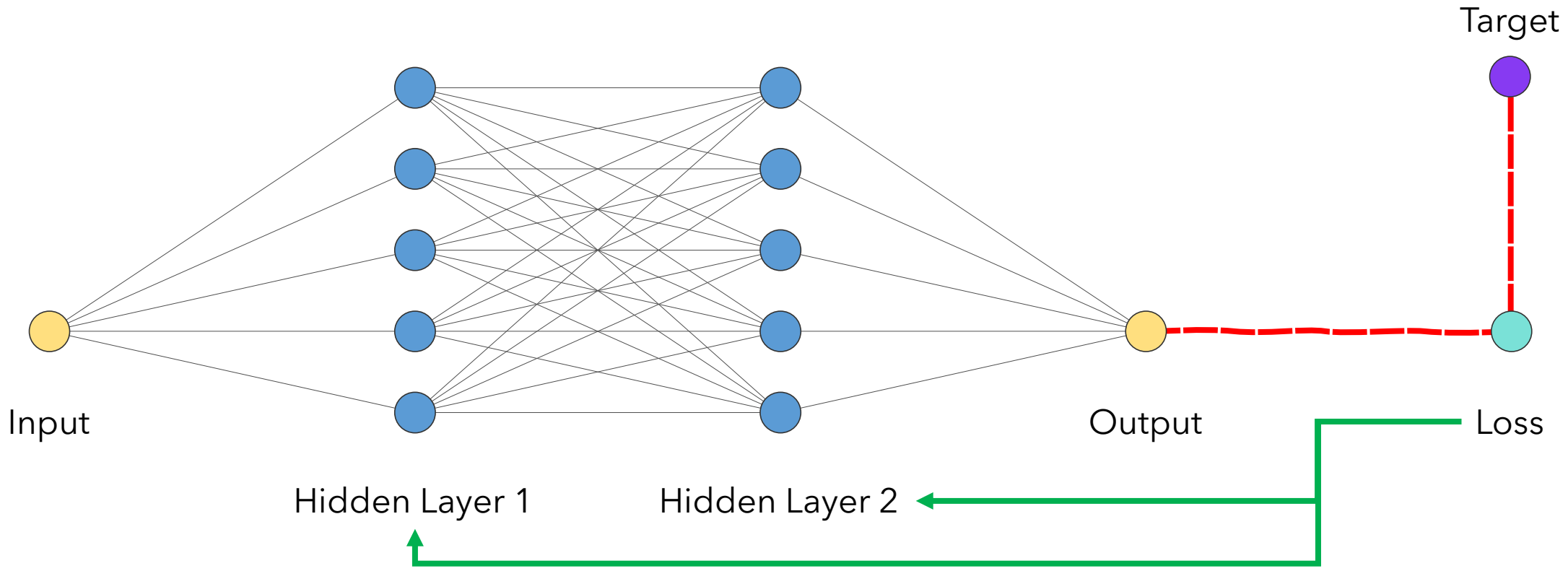
$$\mathbf{Value} = (-1)^{sign} \times 2^{(E-127)} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right) = (+1) \times 2^{-3} \times 1.25 = +0.15625$$

Modern GPUs also support a 16-bit floating point number, with less precision.

Outline

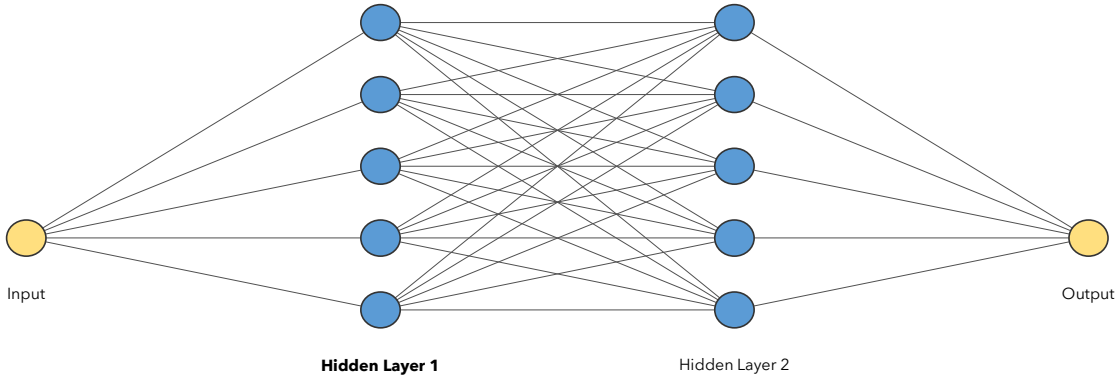
- What is quantization
  - Why we need quantization
- Numerical representation
  - Integers
  - Floating-point numbers
- Introduction to quantization
  - Review of how neural networks work
  - Quantization of NN layers
- Types of quantization
  - Asymmetric and symmetric quantization
  - Quantization range
  - Quantization granularity
  - Post-Training Quantization
  - Quantization-Aware Training

A review of neural networks



- The neural network can be made of up many different layers. For example, linear layers are made up of two matrices, called the **weight** and the **bias**, which are commonly represented using floating-point numbers. Quantization aims to use integer numbers to represent these two matrices, while maintaining the accuracy of the model.

Applying quantization



Dequantize → **Output** → $Y = XW + B$ ← **Perform all operations using integer arithmetic**

Input
Sometimes called "activation"

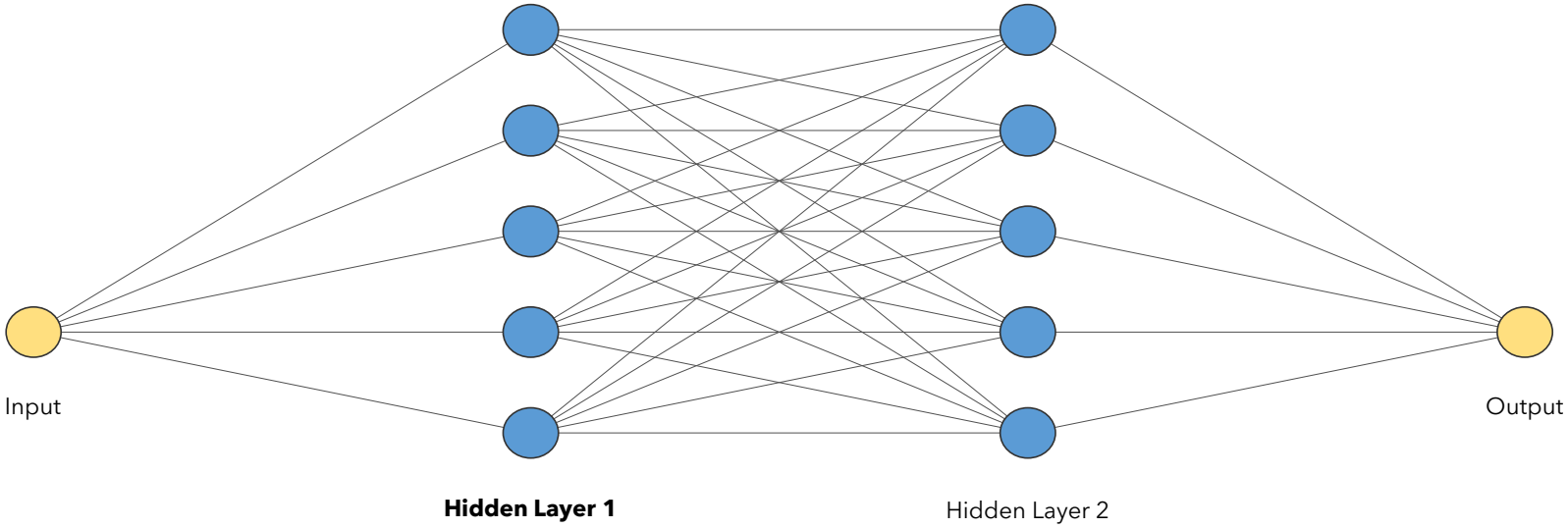
Weight

Bias
Usually quantized as int32

Quantize

The main benefit is that integer operations are much faster in most hardware (especially on embedded devices) than floating point operations

Applying quantization



2484.8	-323.89	0	1638.11	1999.61
-275.59	58.62	-85.98	1187.55	1022.13
2483.8	1055.71	59.72	338.39	1442.53
2021.12	-322.89	2459.82	900.31	1844.46
1963.27	829.25	1352.48	567.54	1588.66

Quantize

[-127,127]

We usually *sacrifice* -128 to obtain a symmetric range

127	-17	0	84	102
-14	3	-4	61	52
127	54	3	17	74
103	-17	126	46	94
100	42	69	29	81

Dequantize

2484.8	-332.61	0	1643.49	1995.67
-273.91	58.7	-78.26	1193.49	1017.4
2484.8	1056.53	58.7	332.61	1447.84
2015.23	-332.61	2465.23	900.01	1839.14
1956.54	821.74	1350.01	567.4	1584.79

Umar Jamil – https://github.com/hkproj/quantization-notes

Outline

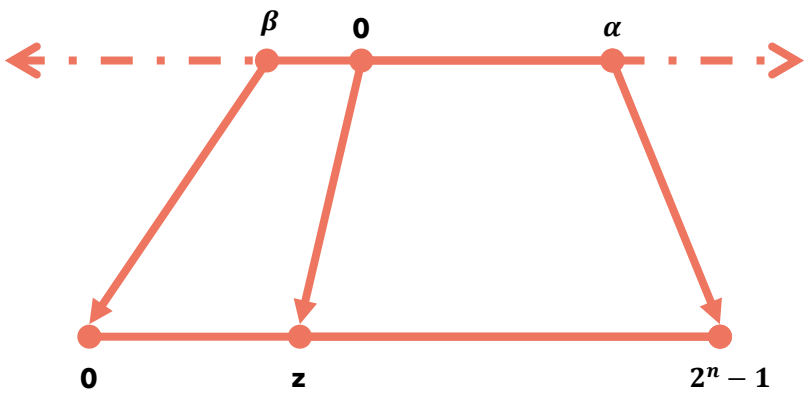
- What is quantization
  - Why we need quantization
- Numerical representation
  - Integers
  - Floating-point numbers
- Introduction to quantization
  - Review of how neural networks work
  - Quantization of NN layers
- Types of quantization
  - Asymmetric and symmetric quantization
  - Quantization range
  - Quantization granularity
  - Post-Training Quantization
  - Quantization-Aware Training

Asymmetric vs Symmetric quantization

43.31	-44.93	0	22.99	-43.93	-11.35	38.48	-20.49	-38.61	-28.02

Asymmetric

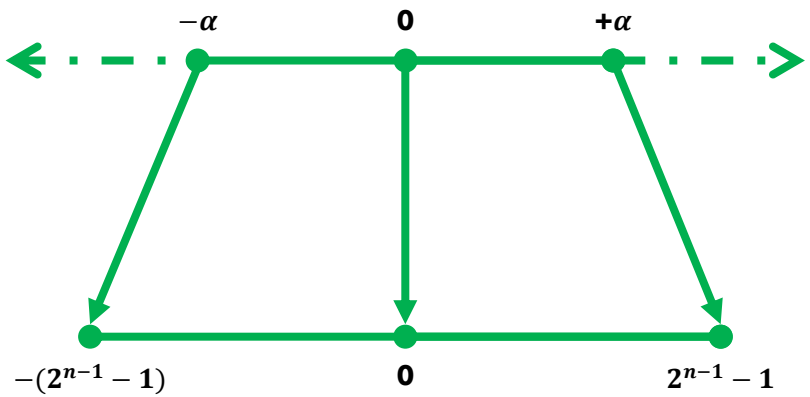
[-44.93, 43.31]



[0, 255]

Symmetric

[-44.93, +44.93]



[-127, 127]

255	0	130	196	3	97	241	71	18	49

122	-127	0	65	-124	-32	109	-58	-109	-79

Asymmetric quantization

- It allows to map a series of floating-point numbers in the range $[\beta, \alpha]$ into another in the range $[0, 2^n - 1]$. For example, by using 8 bits, we can represent floating-point numbers in the range [0, 255]

α β

43.31	-44.93	0	22.99	-43.93	-11.35	38.48	-20.49	-38.61	-28.02

$$x_q = clamp\left(\left\lfloor \frac{x_f}{s} \right\rfloor + z; 0; 2^n - 1\right) \qquad s = \frac{\alpha - \beta}{2^n - 1} \qquad z = \left\lfloor -1 \times \frac{\beta}{s} \right\rfloor$$

Quantize

255	0	130	196	3	97	241	71	18	49

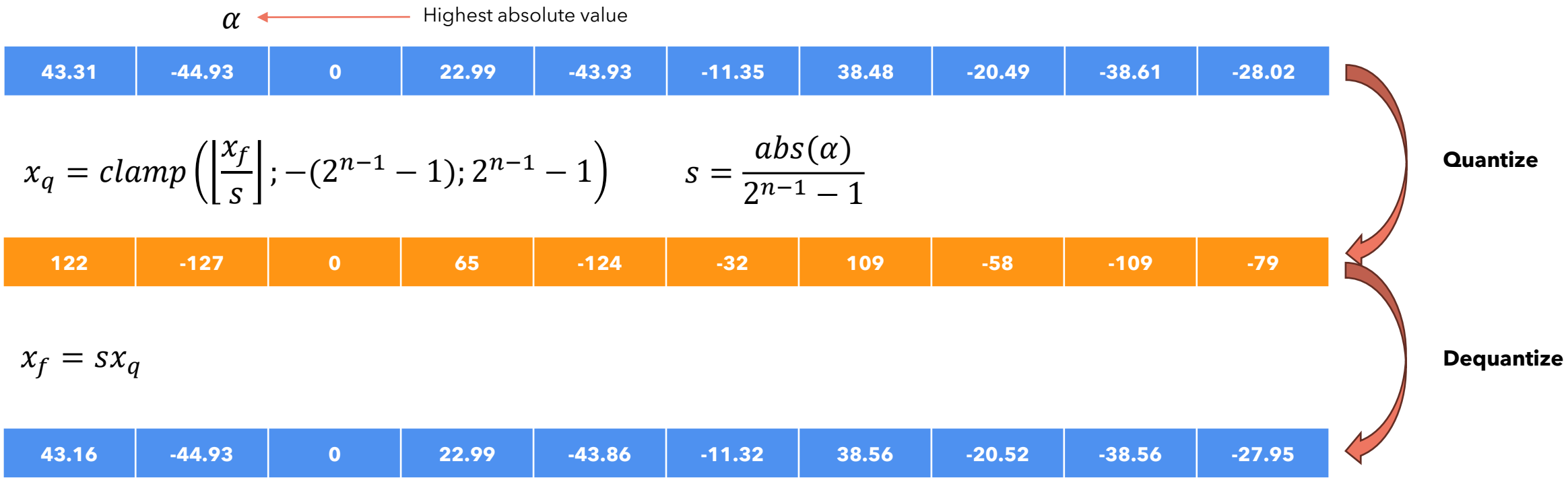
$$x_f = s(x_q - z)$$

Dequantize

43.25	-44.99	0	22.84	-43.95	-11.42	38.41	-20.42	-38.76	-28.03

Symmetric quantization

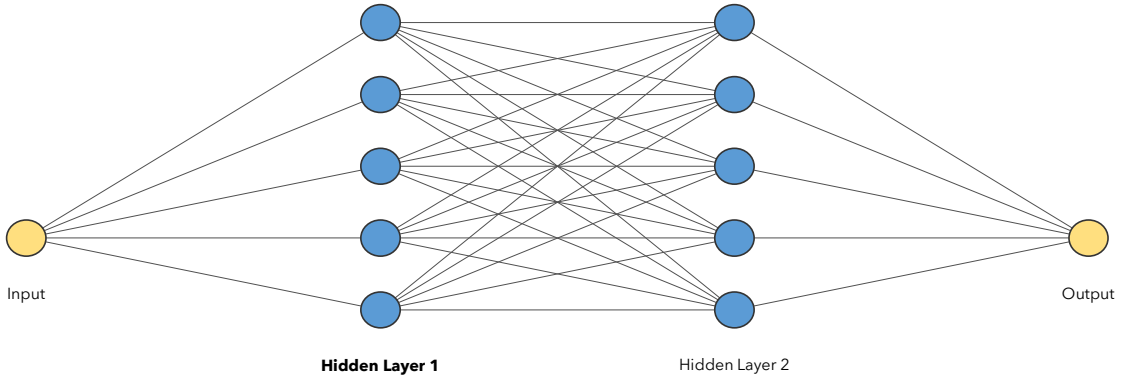
- It allows to map a series of floating-point numbers in the range $[-\alpha, \alpha]$ into another in the range $[-(2^{n-1}-1), 2^{n-1}-1]$. For example, by using 8 bits, we can represent floating-point numbers in the range [-127, 127]



Talk is cheap. Show me the code.

- Linus Torvalds

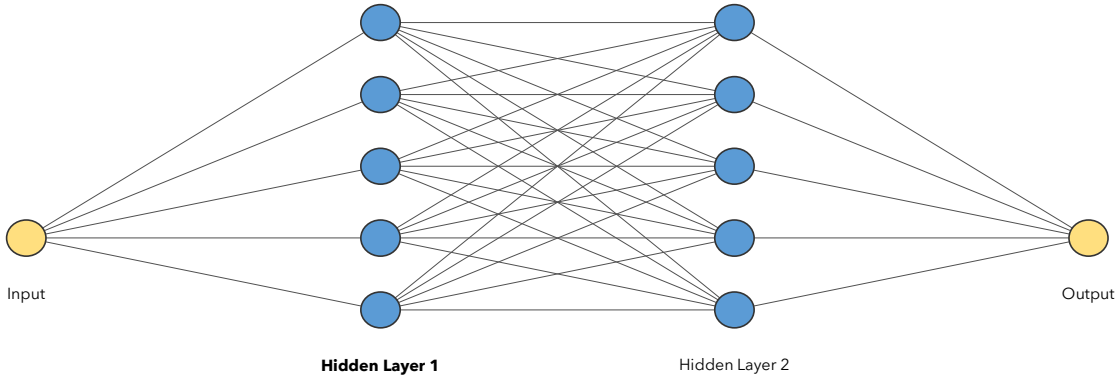
Applying quantization: floating point case



Output $\rightarrow$ $Y_f = XW + B$

Input **Weight** **Bias**

Applying quantization: integer case



Dequantize → **Output** → $Y_q = XW + B$ ← **Perform all operations using integer arithmetic**

How do we dequantize Y?
Y is the result of an operation, how can we dequantize it since we never calculated its scale (s) and zero (z) parameters?

We basically run inference on the model using a few inputs and "observe" the typical output to calculate the scale (s) and the zero (z). This process is called calibration. **We will see it later with Post-Training Quantization.**

Input
Sometimes called "activation".
Can be quantized "on the fly" using a process called (**dynamic quantization**) or with observers.

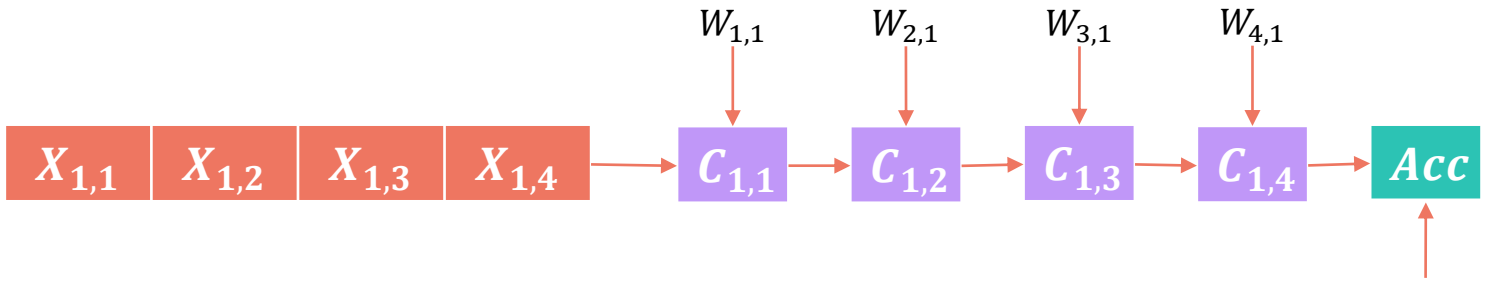
Weight

Bias
Usually quantized as int32

Quantize

Low-precision matrix multiplication

When we calculate the product $\boldsymbol{XW} + \boldsymbol{B}$ in a linear layer, this results in a list of dot products between each row of the matrix **X** with each column of the matrix **Y**, summing the corresponding element of the bias vector **B**.
The GPU can accelerate this operation by using a **Multiply-Accumulate (MAC)** block, which is a physical unit in the GPU that works as follows:



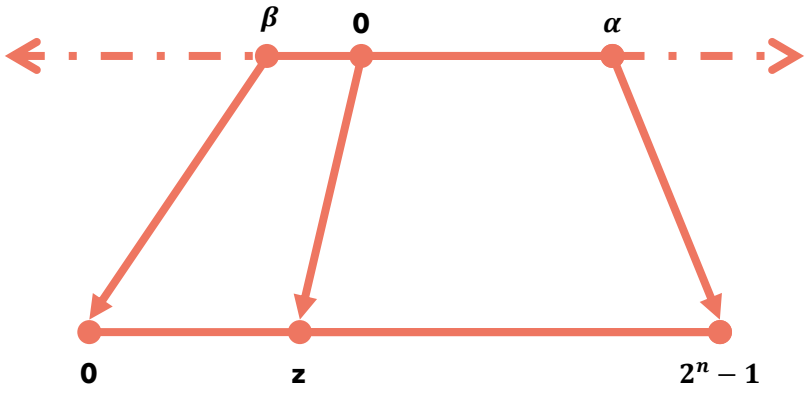
Initialized with the bias element.
This is also the reason we quantize the bias as 32 bit, because it has to accumulate the product of many 8 bit integers.

The GPU will perform this operation in parallel for every row and column of the initial matrices using many Multiply-Accumulate (MAC) blocks.

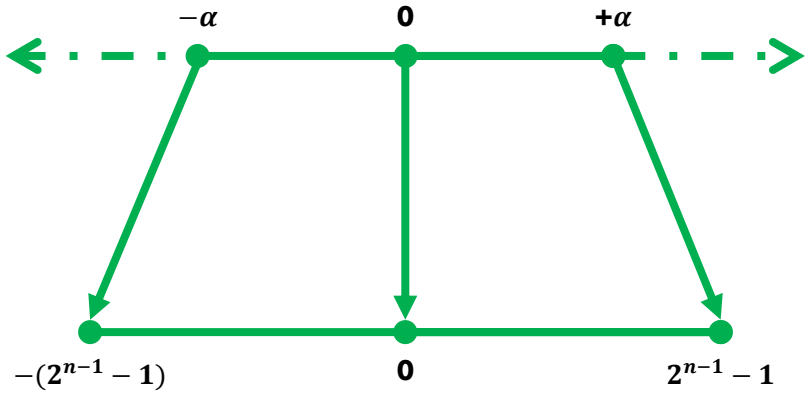
For the full derivation of the math behind low-precision matrix multiplication, Google's GEMM library provides a fantastic explanation: https://github.com/google/gemmlowp/blob/master/doc/quantization.md

Asymmetric vs Symmetric quantization

Asymmetric



Symmetric



How do we choose the $[\alpha, \beta]$ range? Any value outside this range will be clipped to the range. We have many strategies to decide it. Let's review them.

Quantization range: how to choose $[\alpha, \beta]$

Min-Max: To cover the whole range of values, we can set

- $\alpha = \max(V)$
- $\beta = min(V)$
- Sensitive to outliers.

Outlier

Original	43.31	-44.93	0	...	...	...	38.48	-20.49	1000.00	-28.02
Dequantized (Min-Max)	45.08	-45.08	0	24.59	-45.08	-12.29	36.88	-20.49	999.85	-28.68

Percentile: Set the range to the percentile of the distribution of V, to reduce sensitivity to outliers

Dequantized (Percentile)	43.38	-44.52	0	...	...	...	38.48	-20.49	50.00	-28.01

Only the outlier is quantized with a large error

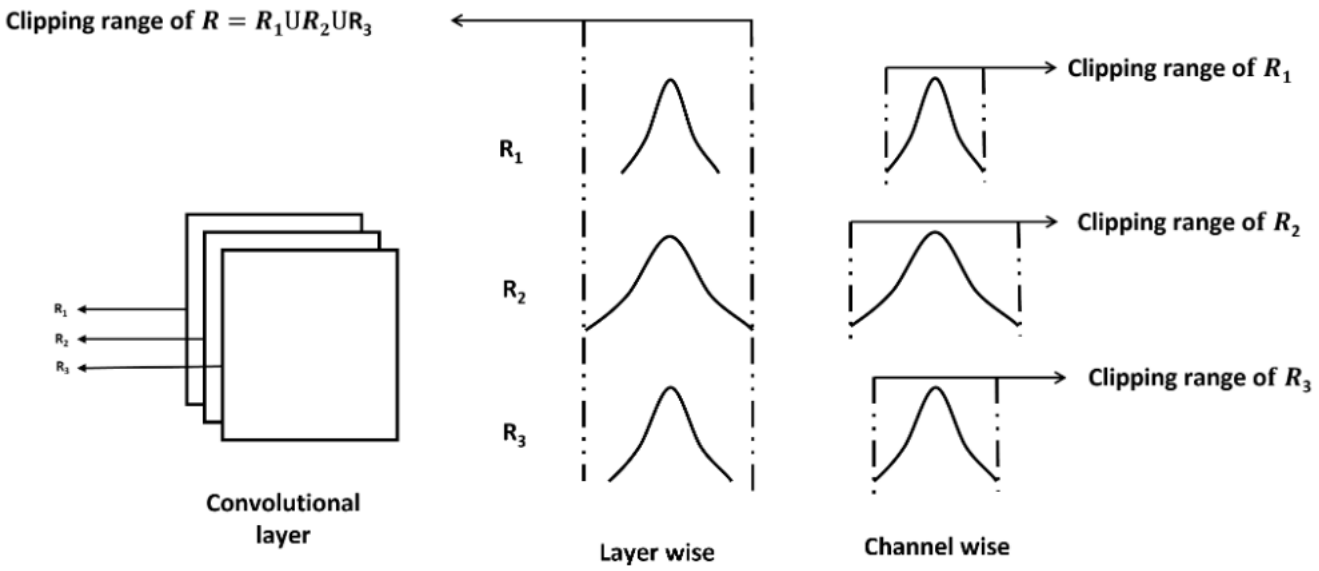
Let's have a look at the code!

Quantization range: how to choose $[\alpha, \beta]$

If the vector V represents the tensor to be quantized, we can choose the $[\alpha, \beta]$ range according to the following strategies:

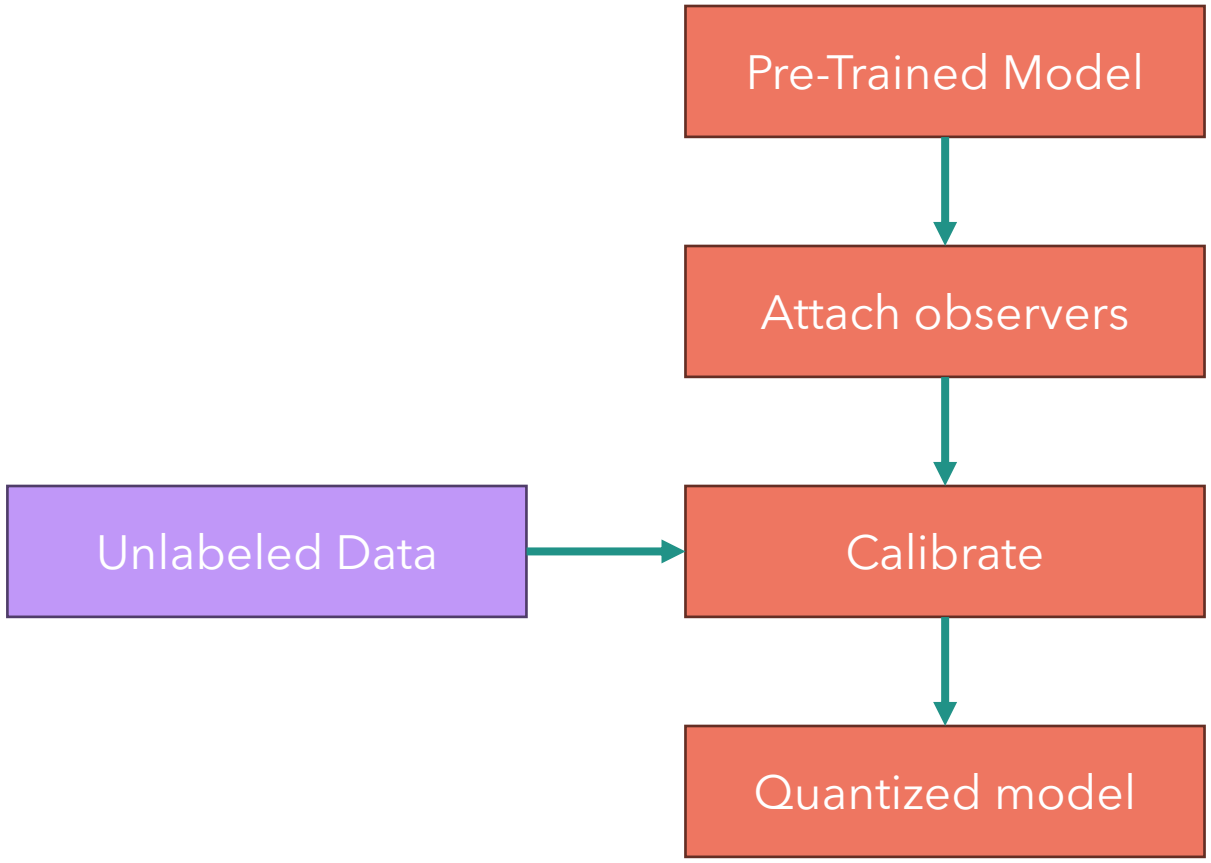
- **Mean-Squared-Error**: choose $[\alpha, \beta]$ such that the MSE error between the original values and the quantized values is minimized.
  - It is usually solved using Grid-Search
- **Cross-Entropy**: used when the values in the tensor being quantized are not equally important. This happens for example in the Softmax layer in Large Language Models. Since most of the inference strategies are Greedy, Top-P or Beam search, it is important to preserve the order of the largest values after quantization.
  - $\underset{\alpha,\beta}{\operatorname{argmin}} \operatorname{CrossEntropy}\left((\operatorname{softmax}(V), \operatorname{softmax}\left(\hat{V}\right)\right))$

Quantization granularity



Evaluation of Quantization Methods for Neural Networks, Junaid Mundichipparakkal

Post Training Quantization (PTQ)



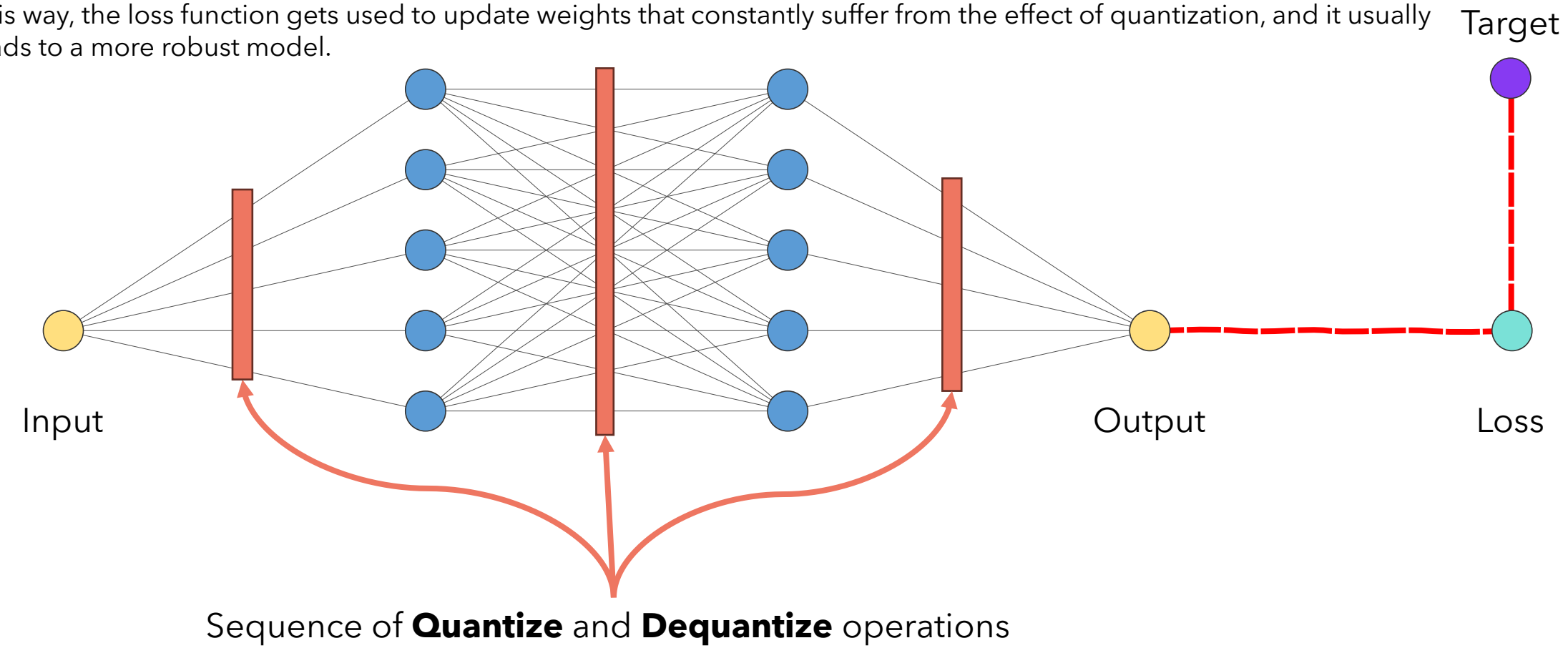
Calculate the s and z parameter using the observed data

Talk is cheap. Show me the code.

- Linus Torvalds

Quantization Aware Training (QAT)

We insert some fake modules in the computational graph of the model to simulate the effect of the quantization during training.
This way, the loss function gets used to update weights that constantly suffer from the effect of quantization, and it usually leads to a more robust model.

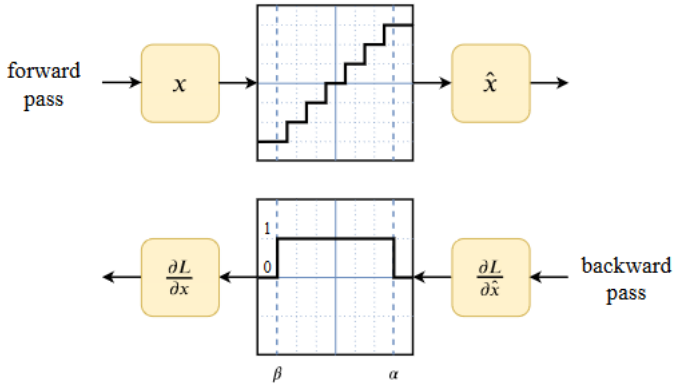


Talk is cheap. Show me the code.

- Linus Torvalds

Quantization Aware Training (QAT): gradient

During backpropagation, the model needs to evaluate the gradient of the loss function w.r.t every weight and input. A problem arises: what is the derivative of the quantization operation we defined before?
A typical solution is to approximate the gradient with the STE (Straight-through Estimator) approximation.

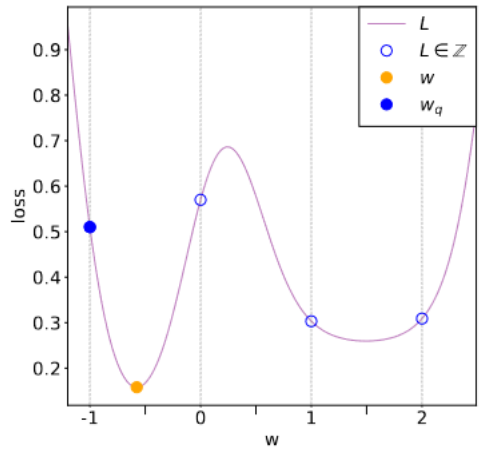


The STE approximation results in 1 if the value being quantized is in the range $[\alpha, \beta]$, otherwise it is 0.

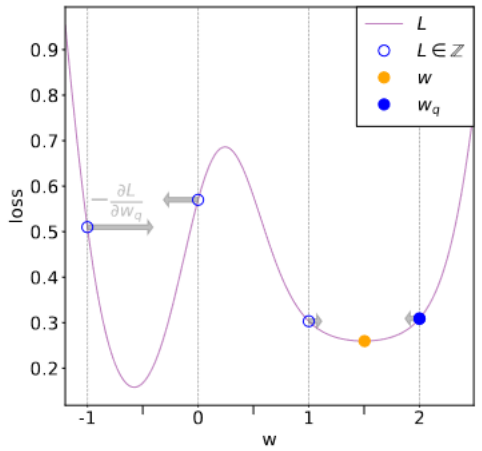
INTEGER QUANTIZATION FOR DEEP LEARNING INFERENCE: PRINCIPLES AND EMPIRICAL EVALUATION, Wu et al.

Quantization Aware Training (QAT): why it works

Why does QAT work? What is the effect of inserting fake quantization operations?



(a) Post training quantization



(b) After quantization aware fine-tuning

INTEGER QUANTIZATION FOR DEEP LEARNING INFERENCE: PRINCIPLES AND EMPIRICAL EVALUATION, Wu et al.

Thanks for watching!
Don’t forget to subscribe for more amazing content on AI and Machine Learning!