Quantization course by Deeplearning.ai

Author	naot97
Date	2025-08-31
Source code	https://github.com/naot97/quantization

1. Benefit of Quantization

The main benefit of quantization is memory saving. By representing model parameters with a lower-precision data type, the model requires significantly less memory while maintaining nearly the same quality.

For example, Image 1 shows how a tensor's values look when stored in different formats:

- FP64 (double precision, 64-bit) → very high precision but large memory usage
- FP32 (single precision, 32-bit) → standard baseline for training/inference
- FP16 / BF16 (half precision, 16-bit) → much smaller memory footprint, still good quality

Quantization goes even further by reducing to int8, int4, or even int2, where memory savings are huge, and inference can be accelerated on hardware that supports these formats.

```
In [12]: print(f"fp64 tensor: {format(tensor_fp64.item(), '.60f')}")
         print(f"fp32 tensor: {format(tensor_fp32.item(), '.60f')}")
         print(f"fp16 tensor: {format(tensor_fp16.item(), '.60f')}")
         print(f"bf16 tensor: {format(tensor_bf16.item(), '.60f')}")
fp64 tensor: 0.333333333333333314829616256247390992939472198486328
125000000
fp32 tensor: 0.333333343267440795898437500000000000000000000000000
000000000
fp16 tensor: 0.333251953125000000000000000000000000000000000000000
000000000
bf16 tensor: 0.333984375000000000000000000000000000000000000000000
000000000
```

Image 1: The value of the tensor in fp64, fp42, fp16 and bf16.

```
In [13]: # Information of `16-bit brain floating point`
         torch.finfo(torch.bfloat16)

finfo(resolution=0.01, min=-3.38953e+38, max=3.38953e+38, eps=0.00
78125, smallest_normal=1.17549e-38, tiny=1.17549e-38, dtype=bfloat
16)

In [14]: # Information of `32-bit floating point`
         torch.finfo(torch.float32)

finfo(resolution=1e-06, min=-3.40282e+38, max=3.40282e+38, eps=1.1
9209e-07, smallest_normal=1.17549e-38, tiny=1.17549e-38, dtype=flo
at32)
```

Image 2: Information of torch.bfloat16 and torch.float3

2. 8-bit linear quantization

2.1. Definition

8-bit linear quantization is a technique that converts floating-point tensor values (e.g., FP32) into 8-bit integers within the range [-128, 127]. The original values are scaled and shifted using a linear mapping so that they fit inside this fixed range.

In image 3, the FP32 tensor is transformed into INT8: values are rescaled proportionally, and numbers outside the range are clipped to the nearest limit (-128 or 127). This reduces memory usage from 32 bits to only 8 bits per value, while still keeping the overall distribution of the data.

2.2. Asymmetric Linear Quantization

Asymmetric linear quantization is a technique that maps floating-point values into a fixed integer range (e.g., INT8: -128 → 127) using a scale and a zero-point. Unlike symmetric quantization (where the range is centered around zero), asymmetric quantization introduces a zero-point offset (z) to ensure that the real value 0 is exactly representable in the quantized domain.

- Scale (s): defines how real values are compressed into the integer range.
- Zero-point (z): shifts the mapping so that zero in real values aligns with an integer value.
- Result: values are linearly transformed into integers, preserving relative differences, while those outside the range are clipped.

This method is especially useful when the original data range is not symmetric around zero.

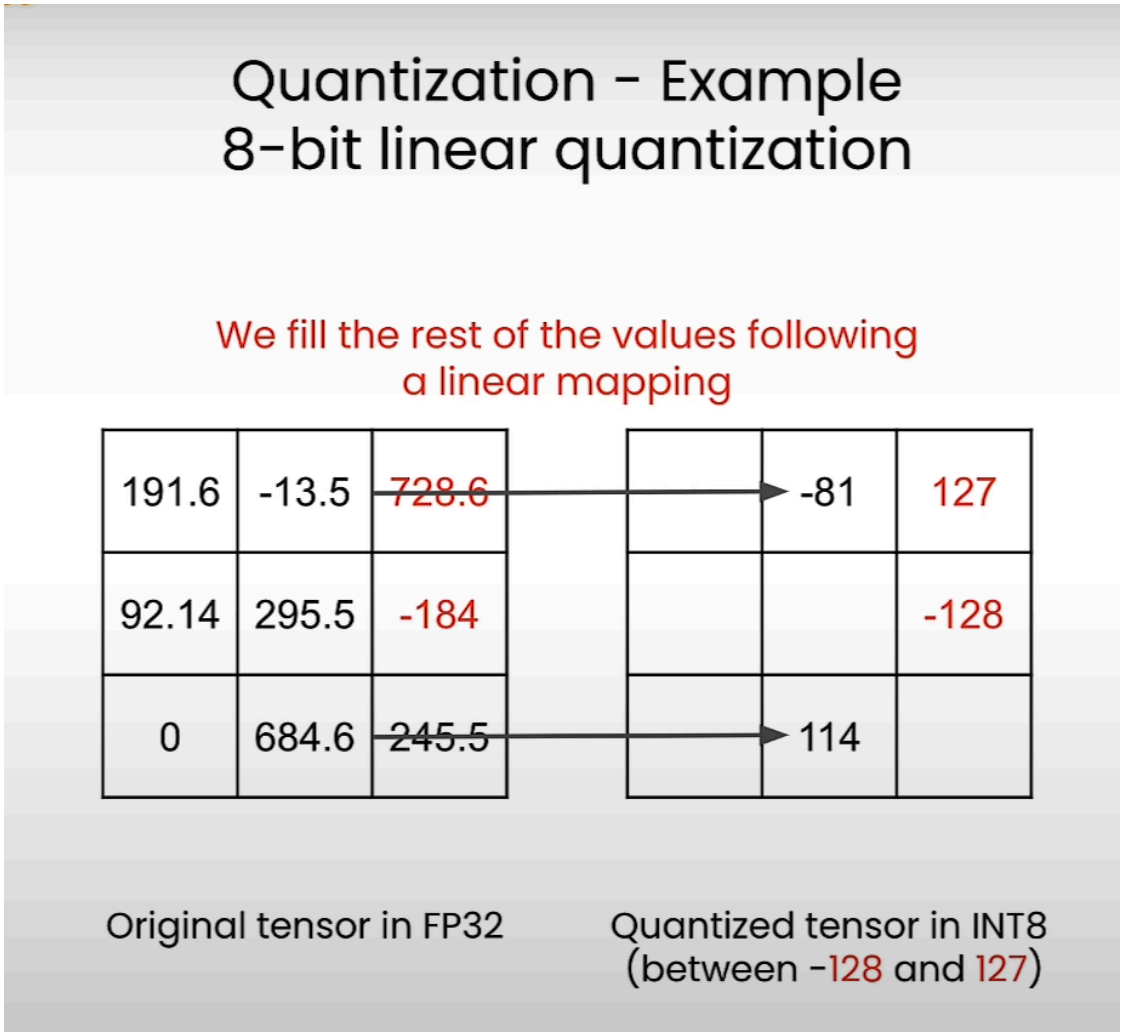

Image 3: Example for 8-bit linear quantization

Formula:

Formula: R = S * (Q - Z)
Where:
Q_min is the minimum value of the quantization type
Q_max is the maximum value of the quantization type
R_min is the minimum value of the real type
R_max is the maximum value of the real type
S is the scale factor
Z is the zero point
R is the real value
Q is the quantized value

From the formula:
R = S * (Q - Z)
=> Q = R / S + Z

Finding the optimal Scale (S) and Zero point (Z):

Scale (S): S = (R_max - R_min) / (Q_max - Q_min)
Because:
R_max - R_min = S * (Q_max - Z) - S * (Q_min - Z)
=> R_max - R_min = S * (Q_max - Q_min)
=> S = (R_max - R_min) / (Q_max - Q_min)

Zero point (Z): Z = Q_min - R_min / S
Because:
Q_min = R_min / S + Z
=> Z = Q_min - R_min / S

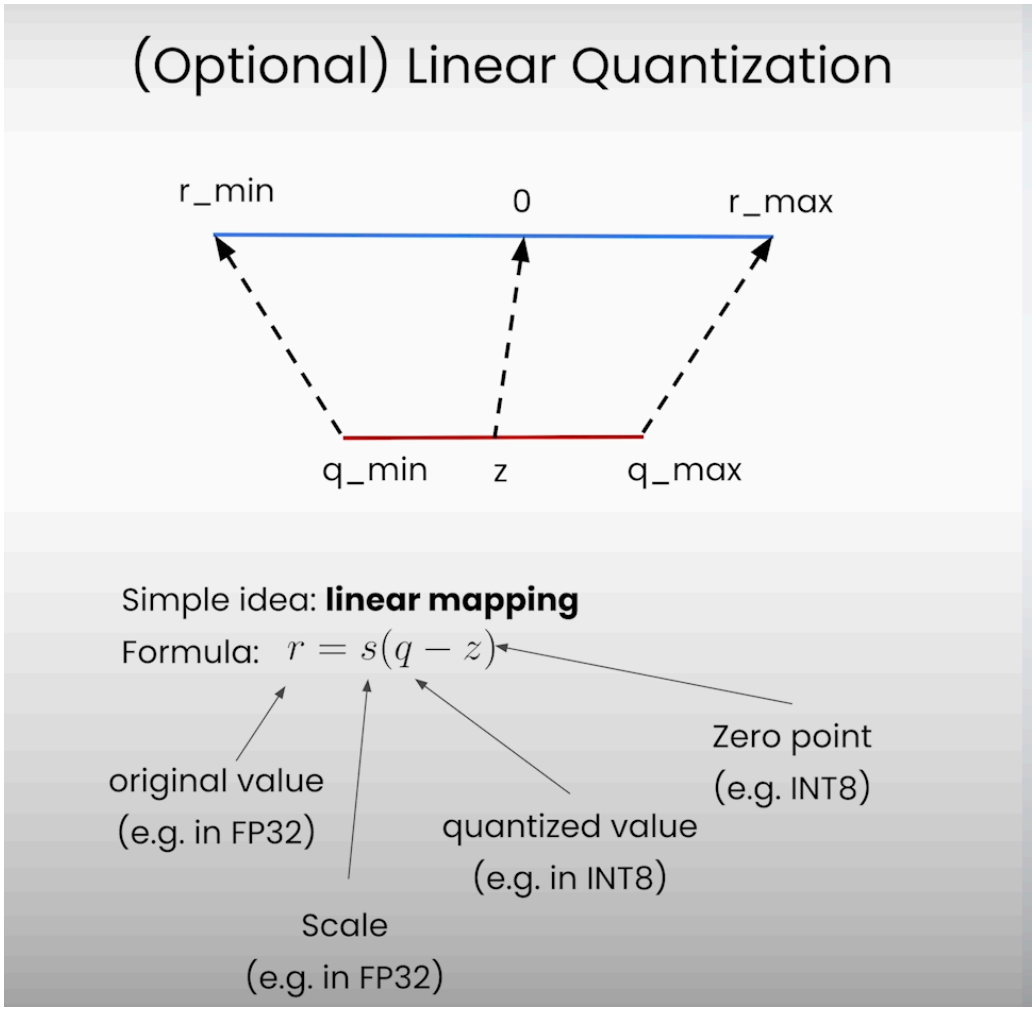

Image 4: Formula of Asymmetric Linear Quantization

2.3. Symmetric Linear Quantization

In symmetric quantization, the quantization range is centered around zero. There is no zero-point offset (Z = 0). The real values are simply scaled into the integer range.

Formula:

Formula:
R = S * Q
Q = R / S

Scale (S): S = max(|R_min|, |R_max|) / Q_max
Zero point (Z): Z = 0

Pros & Cons:

Pros:
• Simple implementation
• More memory-efficient

Cons:
• May introduce bias if data range is not symmetric
• Less optimal for distributions shifted away from zero

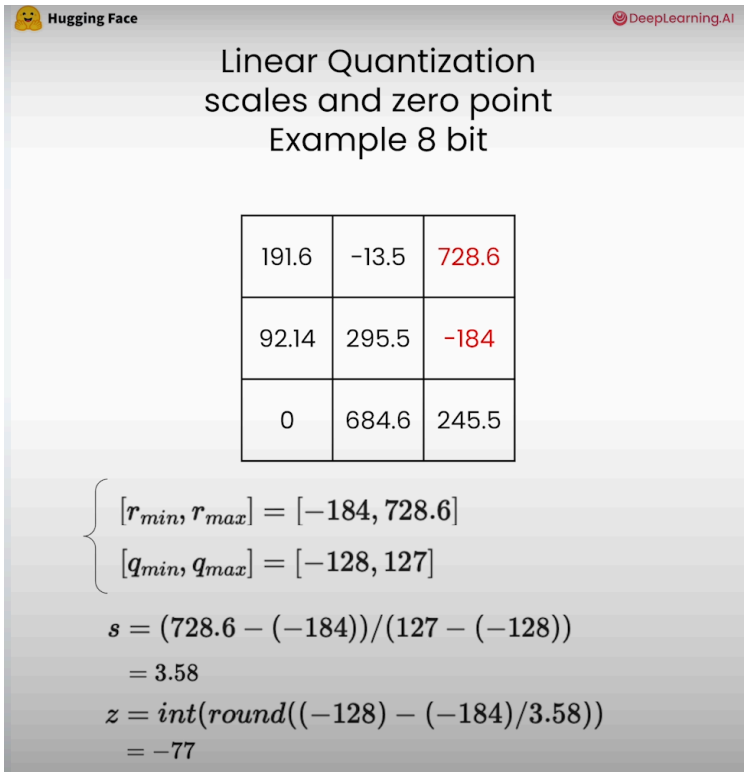

Image 5: Formula of Asymmetric Linear Quantization

3. Challenges

Quantization offers significant benefits in memory and computation efficiency, but it also introduces several challenges:

3.1. Outliers in Activations

Problem: Activations often contain rare but very large values (outliers). These outliers stretch the quantization range, which forces the scale factor to accommodate extreme values. As a result, most of the normal values get mapped into a small region of the quantized range, leaving them with fewer effective bits. This leads to precision loss and degraded model accuracy.

Example: In the original distribution, |X| (activations) shows strong spikes caused by outliers, making them hard to quantize. In contrast, |W| (weights) are more evenly distributed and easy to quantize.

Solution direction: Techniques like SmoothQuant redistribute this difficulty by scaling activations and migrating variance into weights. After smoothing, both activations and weights become easier to quantize with minimal accuracy drop.

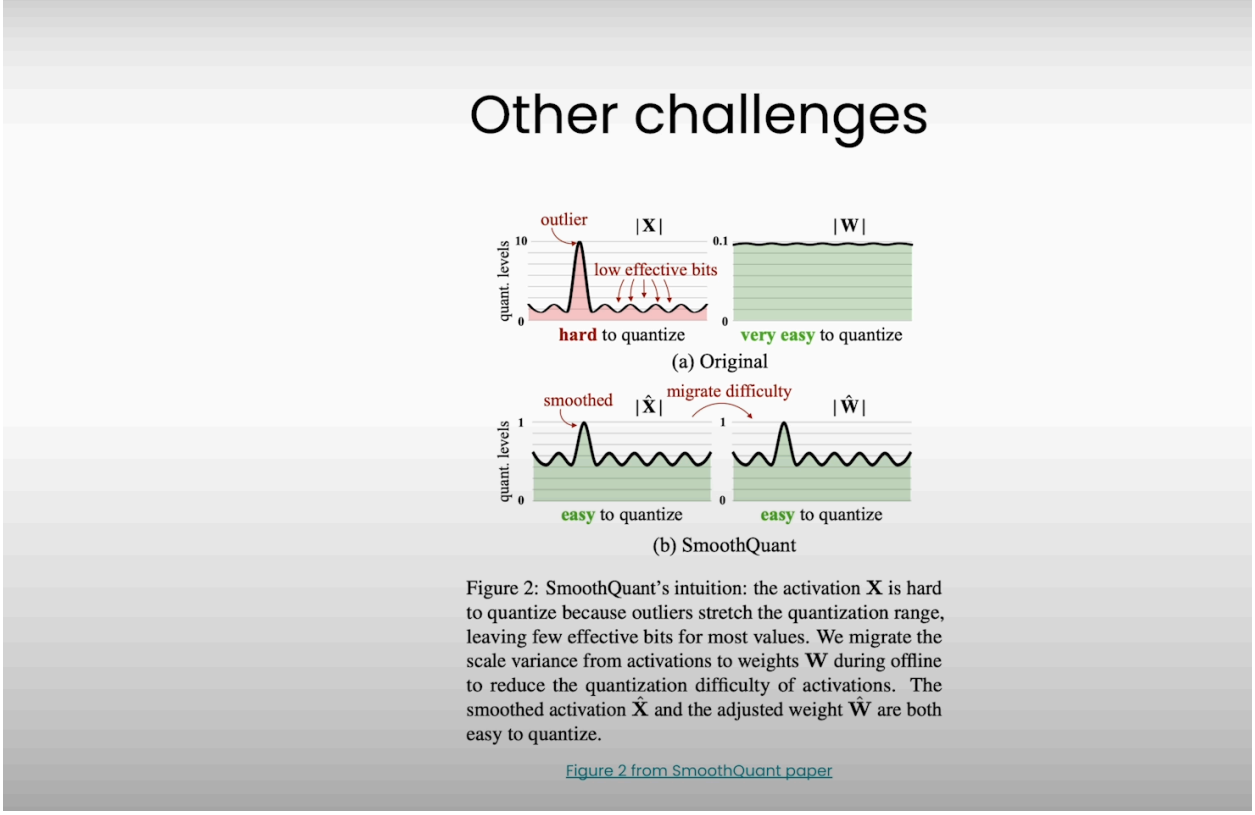

Image 6: Approach of SmoothQuant for outliers

3.2. Mixed Precision and Hardware Inefficiency

Problem: A naïve solution to handle outliers is to keep a small fraction of weights or activations in higher precision (e.g., FP16), while the rest are quantized (e.g., INT3). Although this improves accuracy, it breaks hardware efficiency because accelerators are optimized for uniform low-bit operations. Mixing FP16 and INT3 in the same computation creates inefficiency and under-utilization of hardware resources.

Example (AWQ):
(a) RTN quantization maps all weights to low-bit, leading to poor accuracy (PPL 43.2).
(b) Keeping 1% salient weights in FP16 improves accuracy (PPL 13.0) but sacrifices efficiency.
(c) AWQ scales weights before quantization so that all weights can stay in INT3, achieving both accuracy (PPL 13.0) and hardware efficiency.

Solution direction: Prefer methods like AWQ that preprocess weights before quantization, avoiding reliance on mixed precision.

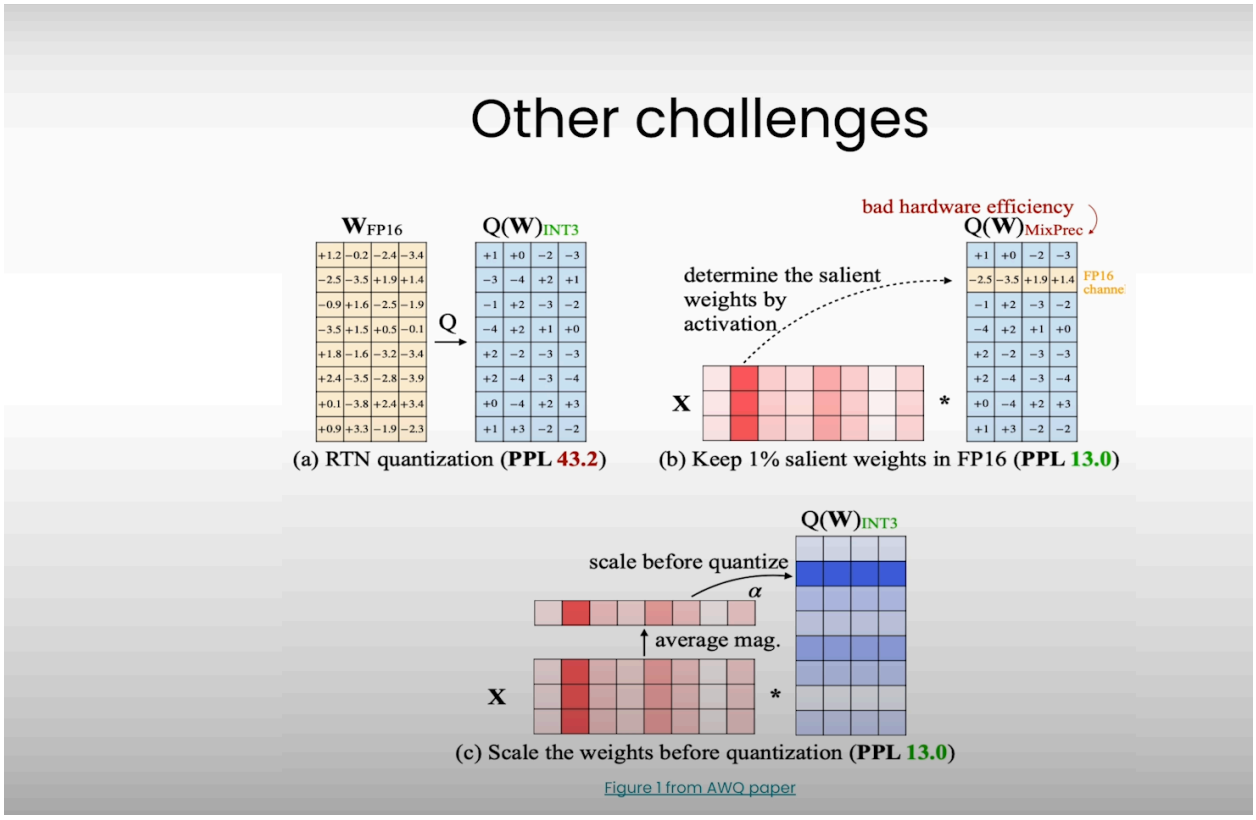

Image 7: Approach of AWQ

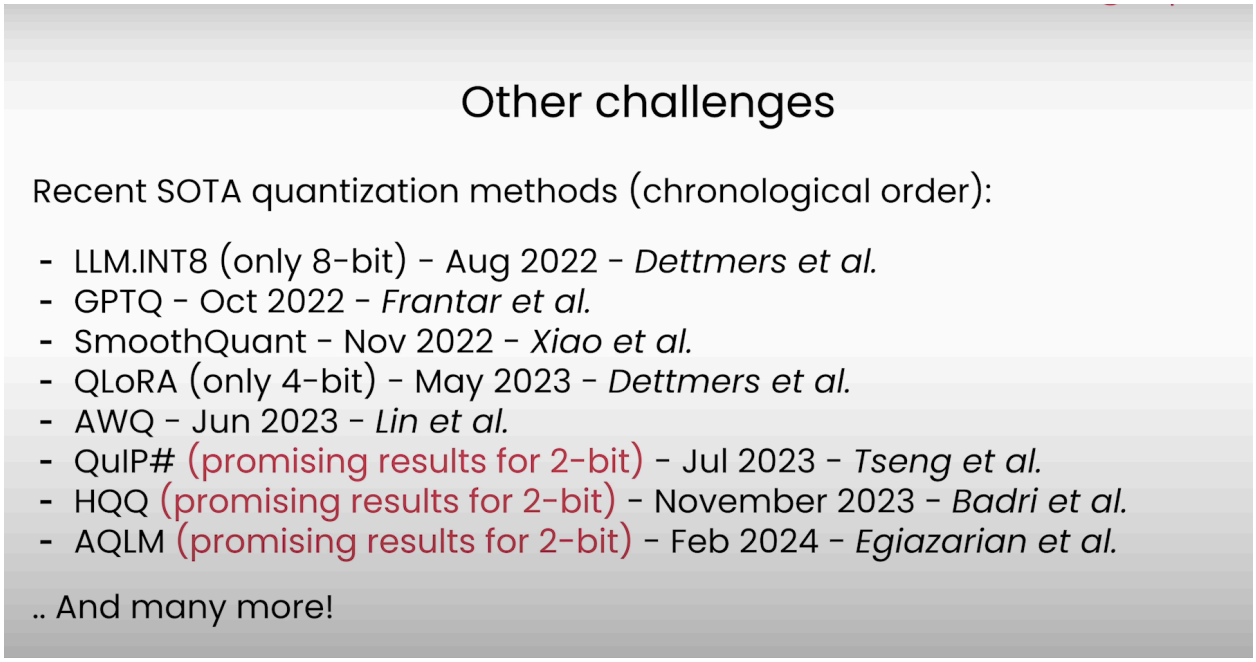

Image 8: Other Approaches

3.3. Matrix Multiplication in Low-Bit (INT8)

Problem: Directly performing large matrix multiplications in INT8 often introduces significant accuracy loss, especially when activations include outliers.

Why: When both weights and activations are quantized, the dot-product accumulates many small quantization errors, which can amplify across layers in large models.

Example(LLM.int8()):
Instead of a single INT8 matmul, the computation is separated into two stages:
Most of the input is quantized and computed in INT8 matmul.
Outliers are isolated and computed in higher precision (FP16).
The results are combined at the end, preserving model quality while keeping most of the efficiency benefits.

Solution direction: Hybrid approaches (INT8 + FP16 for outliers) or structured decompositions help balance efficiency and accuracy in matmul-heavy LLMs.

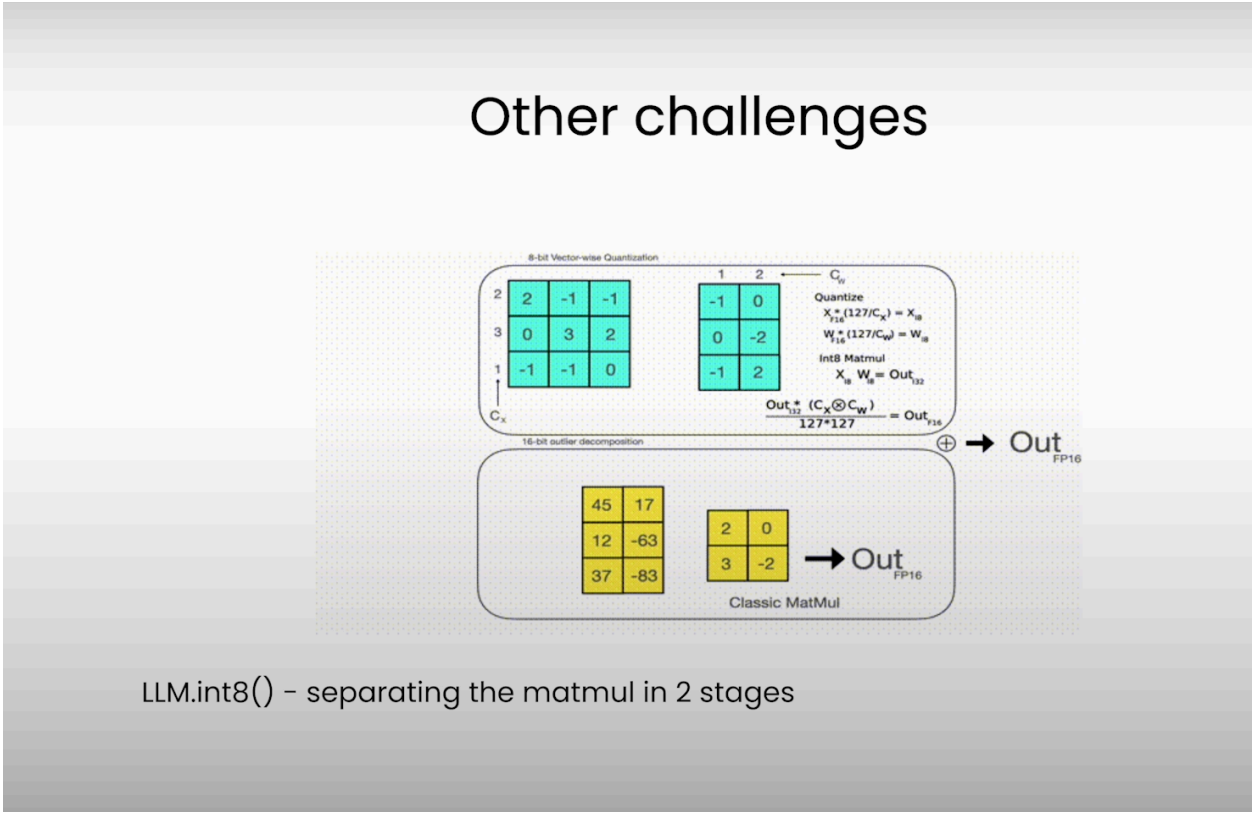

Image 9: Approach of LLM.int8()

3.4. Others

Retraining requirements
Some quantization methods require Quantization-Aware Training (QAT), which is expensive and time-consuming.

Hardware limitations
Not all hardware supports low-bit quantization efficiently (e.g., INT3 or INT2).
Efficient kernels and accelerators are still an active research area.

Calibration and data dependence
Post-training quantization often needs a representative calibration dataset.
Poor calibration can lead to significant accuracy loss.

Packing and unpacking overhead
Efficiently storing and retrieving quantized values adds complexity.
This can offset some of the theoretical memory/computation gains.

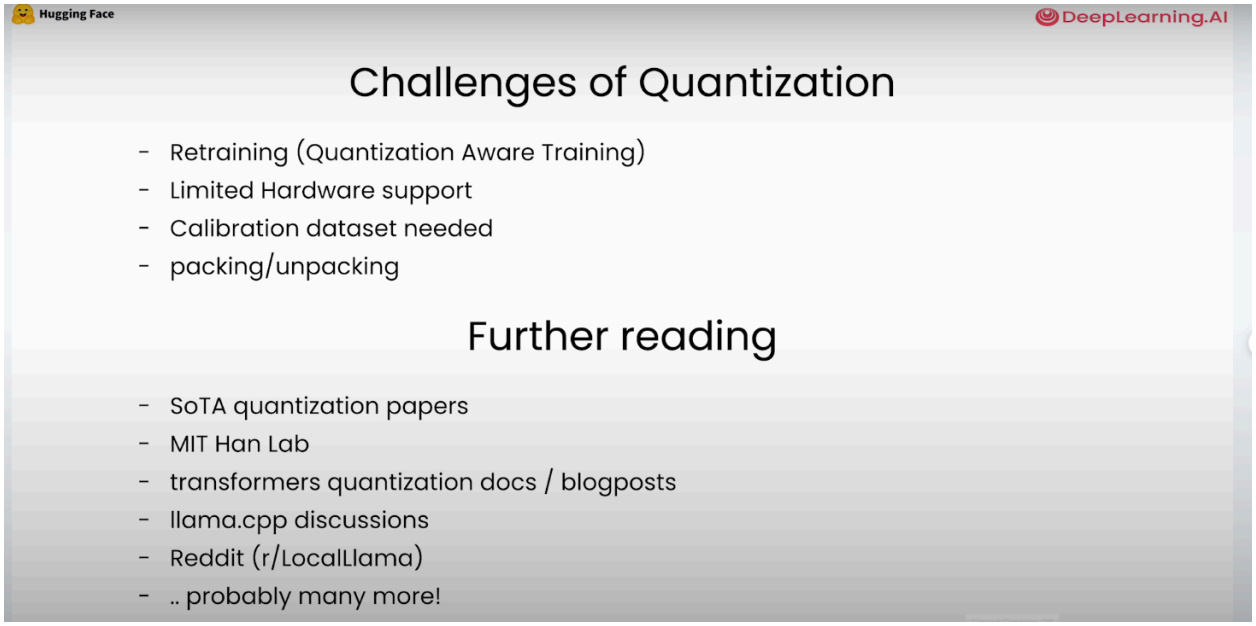

Image 10: Other challenges and Further reading

-