QUANTIZATION IN PRACTICE

Accelerating Vision Models for Production

LLM quantization zoo

LLMs / VLMs

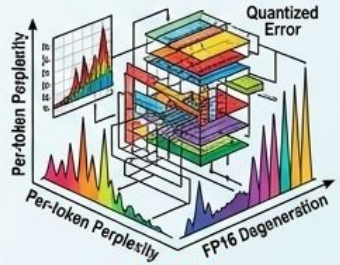
- LLM quants start at “Q8” and only get smaller
  - Q6_K, Q4_K_M, IQ4_XS, Q2_K, ...
  - Weighted / imatrix quants
  - Weight-only quantization
  - Dynamic activation quantization
  - AWQ

- Can we use all of these tricks for “Normal” Vision Models?

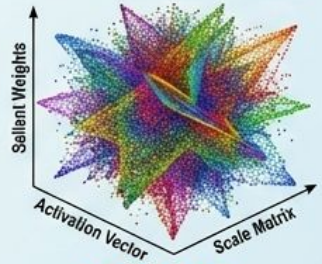
STOP QUANTIZING LLMs

- PARAMETERS WERE NOT SUPPOSED TO BE STORED IN INTEGERS
- TRILLIONS OF DOLLARS IN VENTURE CAPITAL yet NO* REAL WORLD HARDWARE SUPPORTS data types smaller than 8 BITS
- Wanted to get a faster model anyway for a laugh? We had a tool for that: It was called "DISTILLATION"
- "Yes please give me the IQ4_XS quants GGUF. Please give me the NF4 weights" - Statements dreamed up by the utterly Deranged

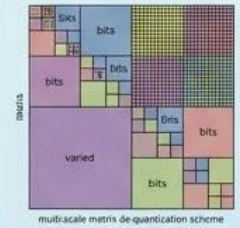
Look at what NLP Researchers have been publishing in Respectable peer reviewed journals, trained with all the GPU compute they stole from us **(These are REAL quantization techniques, used by REAL ML engineers)**:



?????



???????



??????????

"Hello I would like a 0.5 bit value please"

They have played us for absolute fools

*this is a lie

LLM quantization is different

Large LM/VLMs

- 7B+ parameters even for small models

- Models that don't fit into VRAM
- Models that are bottlenecked by
  CPU RAM ⇔ HBM ⇔ SRAM
  memory transfers, not compute

- Many of these methods choose to do a lot of extra compute in the hopes of reducing the memory pressure
- Many of these methods try to do something "smarter" to work around compounding quantization errors
  - Q6_K, Q4_K_M, IQ4_XS, Q2_K, ...
  - Weighted / imatrix quants
  - Weight-only quantization
  - Dynamic activation quantization
  - AWQ

"Normal" Vision Models

- ViT-Large is ~300M parameters

- Models normally fit into device memory
- Models that are compute-limited

- It actually sometimes makes sense to run these models on the CPU

- At least in my tests, many "advanced" quantization methods have computational overhead that ends up **slowing down** inference for vision models

Hardware Support Matters

Modern NVIDIA GPUs

- fp16 (or similar) inference 🟢
  - almost universally supported
  - almost always at least **2x faster** compared to fp32

Most Modern CPUs

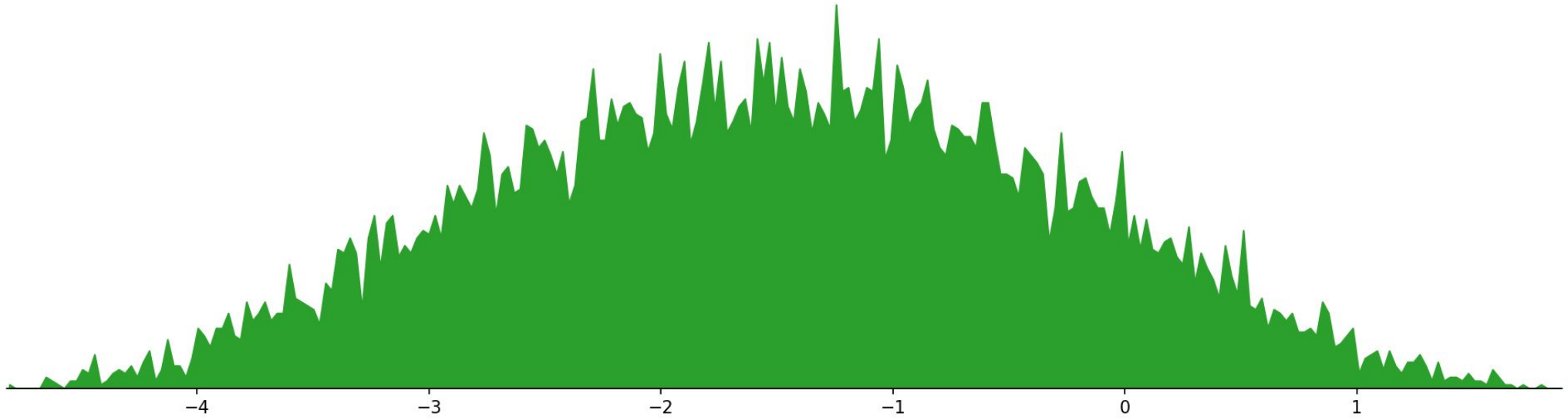
- fp16 (or similar) inference 🔴
  - not well supported on some devices
  - **similar or slower speed** compared to fp32

Both CPUs and GPUs

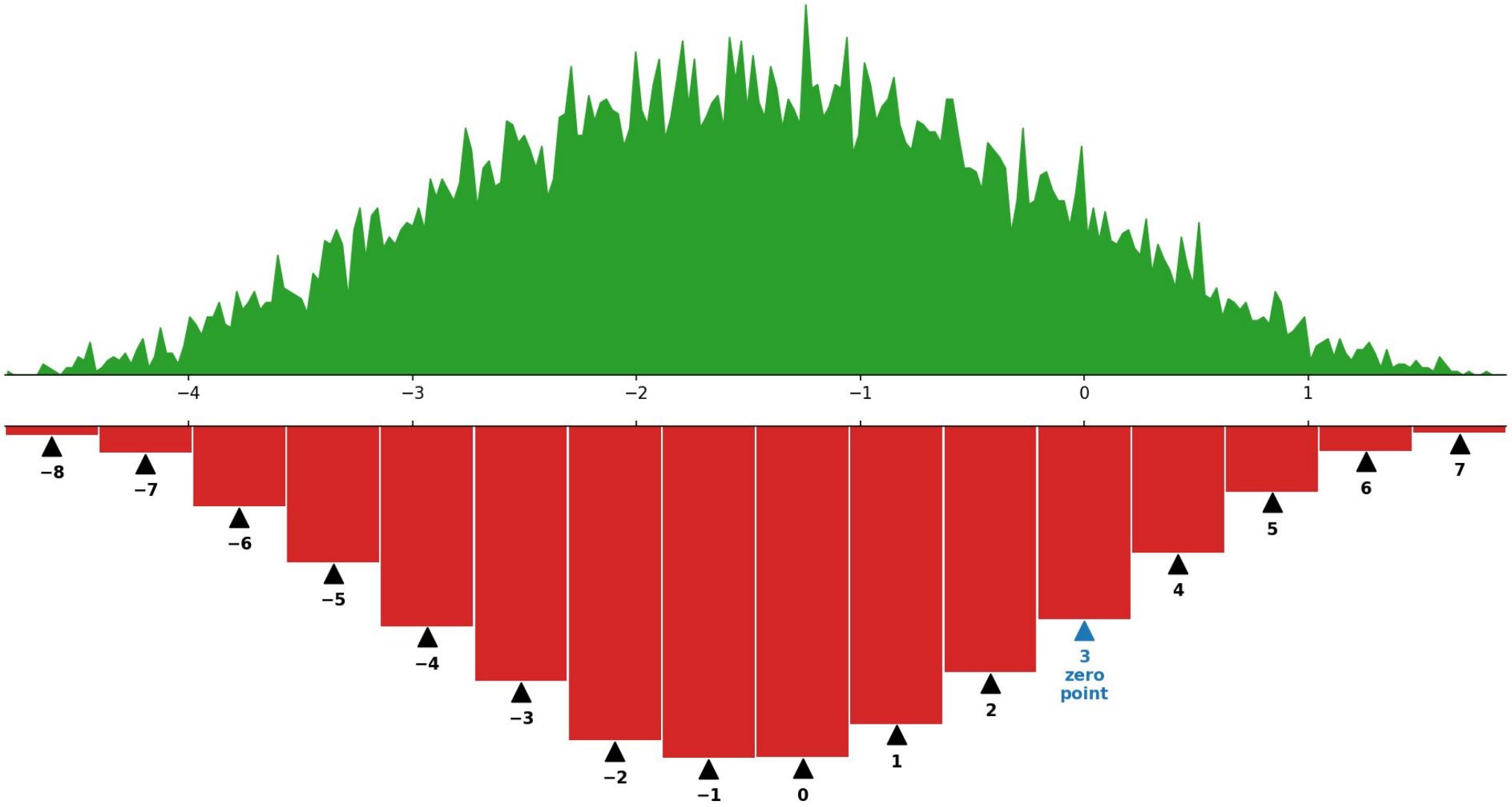
- int8 inference 🟠
  - well supported, but all integer quantization methods require at least some extra operations compared to floating point
  - static int8 quantization is the simplest, but it still has **non-zero overhead** even when the graph is fully evaluated using only integer operations (activation offsets and scales have to be adjusted between nodes)

Linear (Uniform) Static Quantization Basics

Uniform Value Quantization



Uniform Value Quantization



Quantize and Dequantize Operations

$$\text{Let} \quad Y_{\text{real}} \in \mathbb{R}, \quad S_Y \in \mathbb{R}, \quad Z_Y \in \mathbb{Z}$$

$$\text{Quantize:} \quad Y_{\text{int}} := \text{saturate}\left(\text{round}\left(Y_{\text{real}}/S_Y\right) + Z_Y\right)$$

$$\text{Dequantize:} \quad Y_{\text{real}} \approx \left(Y_{\text{int}} - Z_Y\right) \cdot S_Y$$

MatMul

$$Y_{\text{real}} = A_{\text{real}} \times B_{\text{real}}$$

MatMul → QLinearMatMul

$$\text{Dequantize:} \quad \begin{cases} A_{\text{real}} = (A - Z_A) \cdot S_A \\ B_{\text{real}} = (B - Z_B) \cdot S_B \end{cases}$$

$$\text{Compute:} \quad Y_{\text{real}} = A_{\text{real}} \times B_{\text{real}}$$

$$\text{Quantize:} \quad Y = \text{saturate}\left(\text{round}\left(Y_{\text{real}}/S_Y\right) + Z_Y\right)$$

QLinearMatMul

$$Y = \text{saturate}\left(\text{round}\left(\frac{(A - Z_A) \cdot S_A \times (B - Z_B) \cdot S_B}{S_Y}\right) + Z_Y\right)$$

QLinearMatMul

$$Y = \text{saturate}\left(\text{round}\left(\left[(A - Z_A) \times (B - Z_B)\right] \cdot \frac{S_A \cdot S_B}{S_Y}\right) + Z_Y\right)$$

QLinearMatMul

$$Y = \text{saturate}\left(\underbrace{\text{round}\left(\underbrace{\left[(\overset{\text{int}}{A} - \overset{\text{int}}{Z_A}) \times (\overset{\text{int}}{B} - \overset{\text{int}}{Z_B})\right]}_{\text{int}} \cdot \overbrace{\frac{S_A \cdot S_B}{S_Y}}^{\text{real}}\right)}_{\text{int}} + \overset{\text{int}}{Z_Y}\right)$$

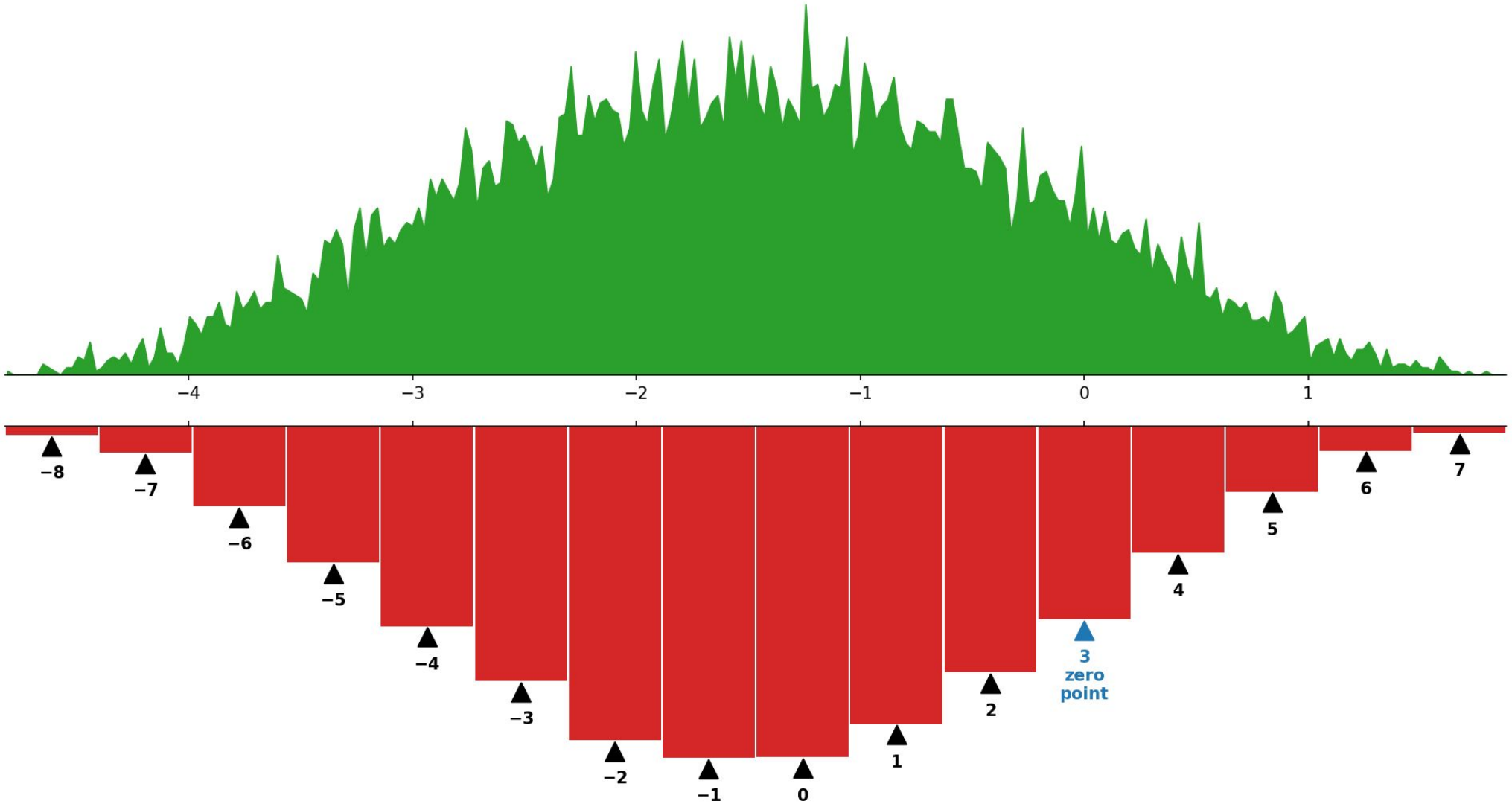
Overflow/Saturation Hazards

$$\underset{\text{(u)int8}}{Y} = \text{saturate}\left(\underbrace{\text{round}\left(\underbrace{\left[\left(\overset{\text{(u)int8}}{A} - \overset{\text{int32}}{Z_A}\right) \times \left(\overset{\text{(u)int8}}{B} - \overset{\text{int32}}{Z_B}\right)\right]}_{\text{int32}} \cdot \overbrace{\frac{S_A \cdot S_B}{S_Y}}^{\text{real}}\right)}_{\text{int32}} + \overset{\text{int32}}{Z_Y}\right)$$

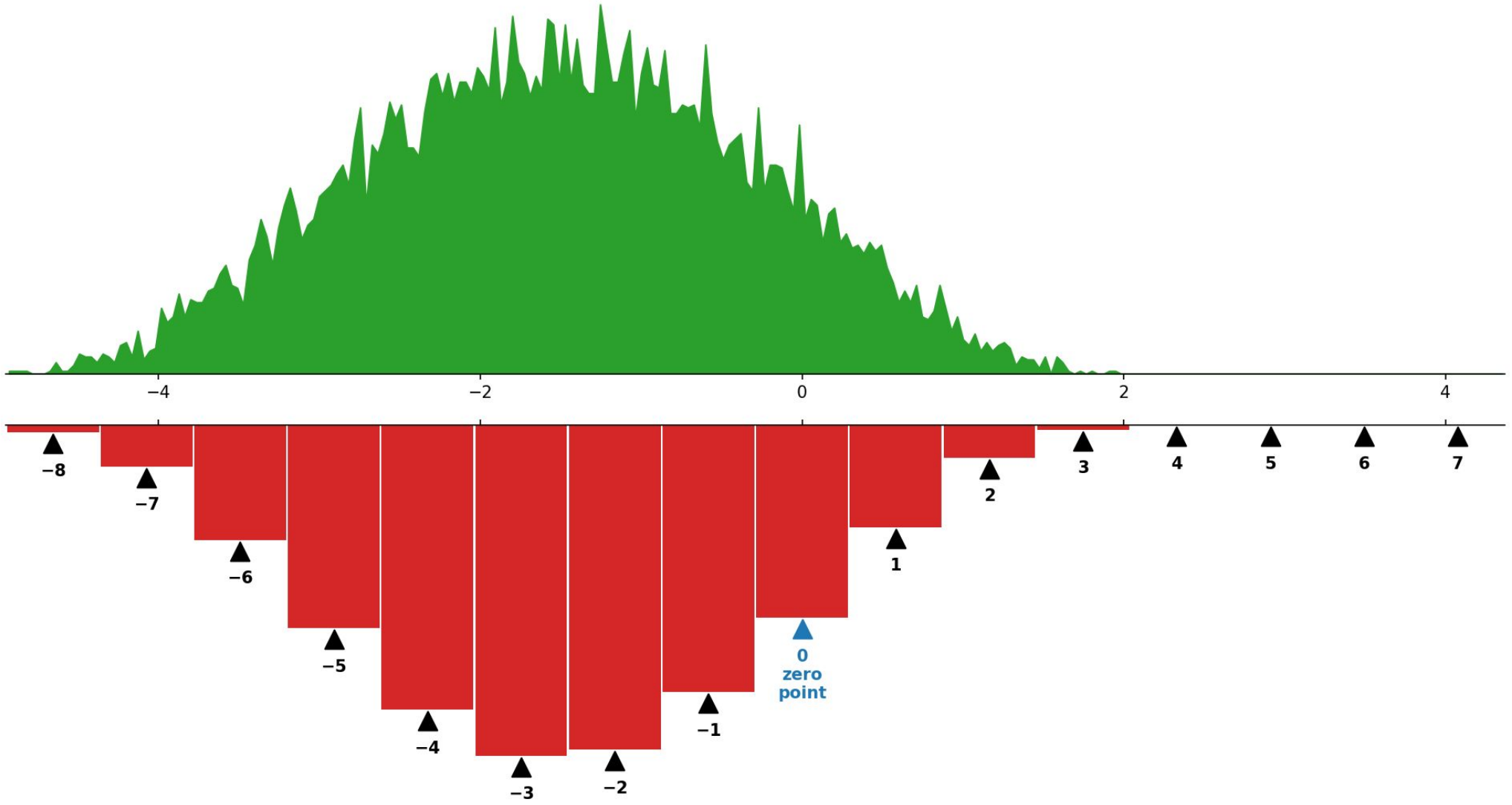
Optional Optimization: Rescaling Integers

$$\underbrace{\text{round}\left(\overbrace{[(A - Z_A) \times (B - Z_B)]}^{\text{int}} \cdot \frac{\overbrace{S_A \cdot S_B}^{\text{real}}}{S_Y}\right)}_{\text{int}} = \underbrace{\text{round}(\overset{\text{int}}{G} \cdot \overset{\text{real}}{S})}_{\text{int}}$$

Asymmetric Quantization



Symmetric Quantization



Optional Optimization:
Asymmetric Activations
Symmetric Weights

$$(A - Z_A) \times (B - Z_B)$$

$$\text{Let } B = \text{const and } Z_B = 0, \text{ then}$$

$$(A - Z_A) \times B =$$

$$A \times B - Z_A \times B$$

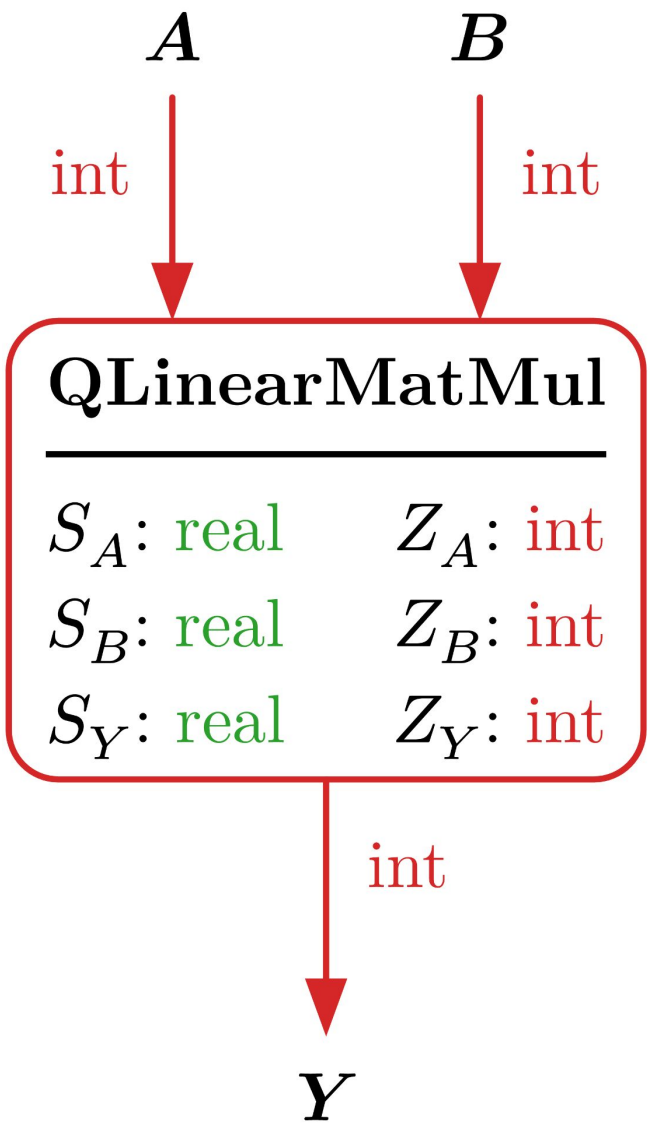
$$\text{Where } C = -Z_A \times B \text{ is also constant}$$

$$\text{and the whole calculation is a GEMM.}$$

$$\text{Bonus: you might also be able to fuse the bias into } C.$$

FakeQuantize
(aka QDQ)

QLinearX for each Node X



Problems:

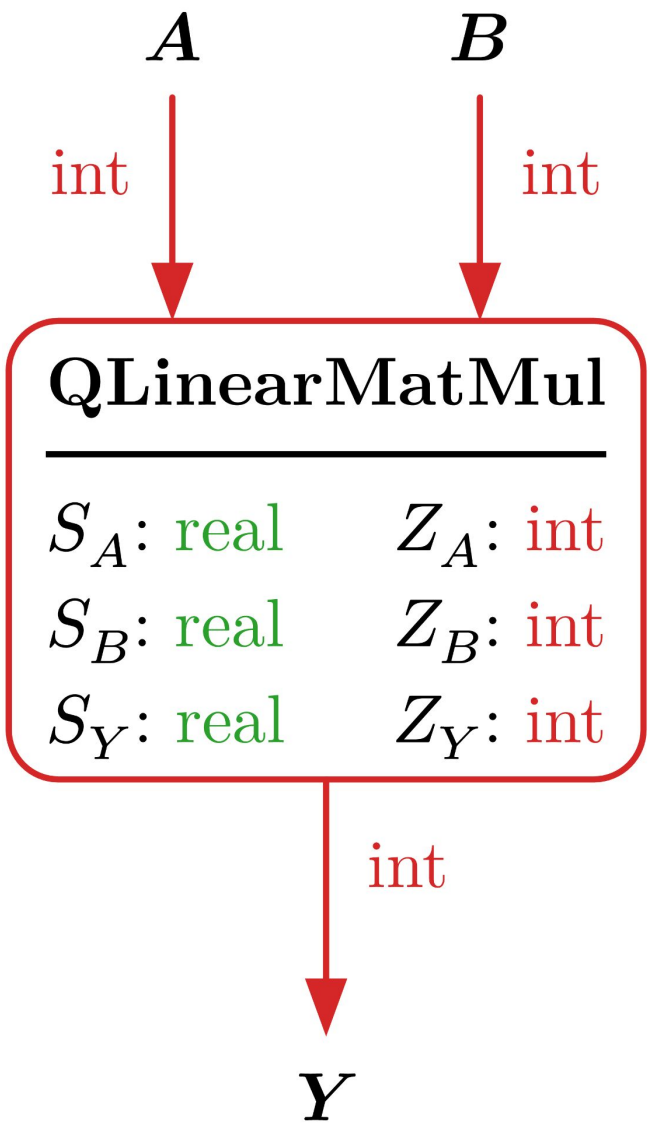
1. For each Node X we might want to use in a quantized model, we are forced to define a new custom Node QLinearX

2. IEEE 754 floating point numbers are nice to work with – they are *almost* associative / commutative / distributive / etc.

   Integers are **not**. Every Node that works with integers will need to carefully define, how it will handle overflows, what precision guarantees it provides, etc.

This is a massive pain.

QLinearX for each Node X

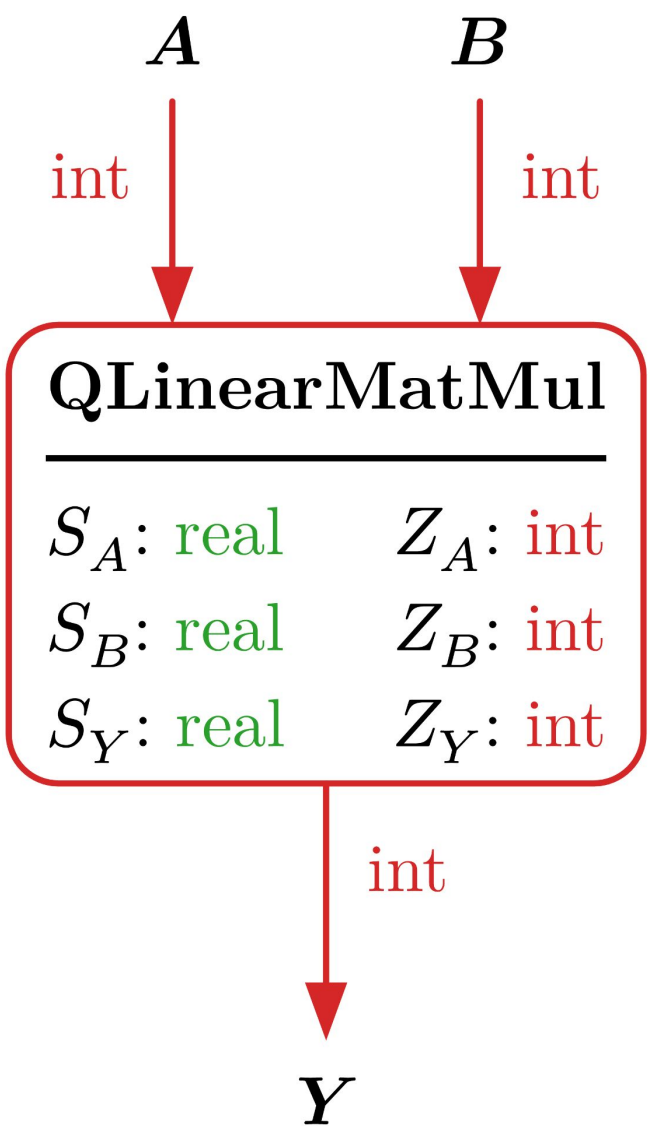


Dequantize:
$$\begin{cases} A_{\text{real}} = (A - Z_A) \cdot S_A \\ B_{\text{real}} = (B - Z_B) \cdot S_B \end{cases}$$

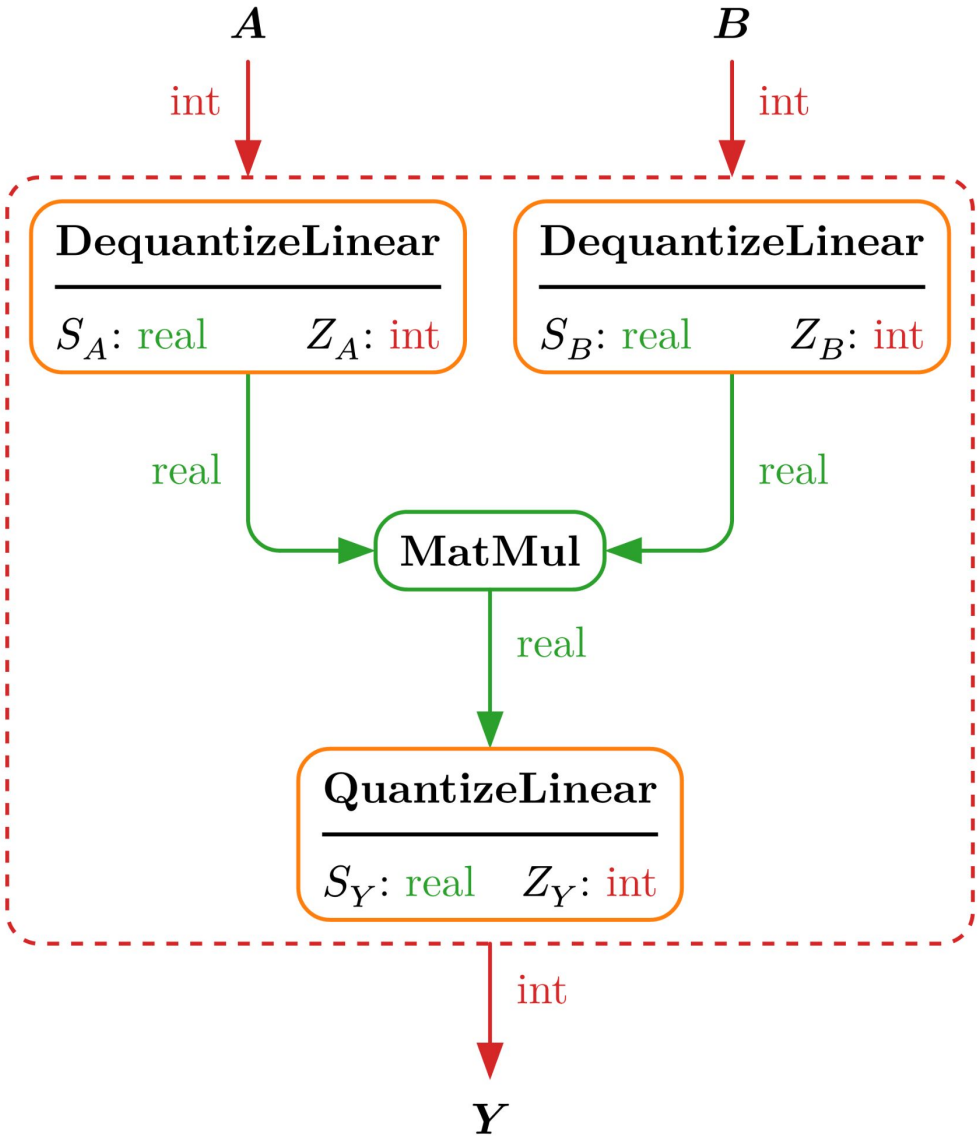
Compute:
$$Y_{\text{real}} = A_{\text{real}} \times B_{\text{real}}$$

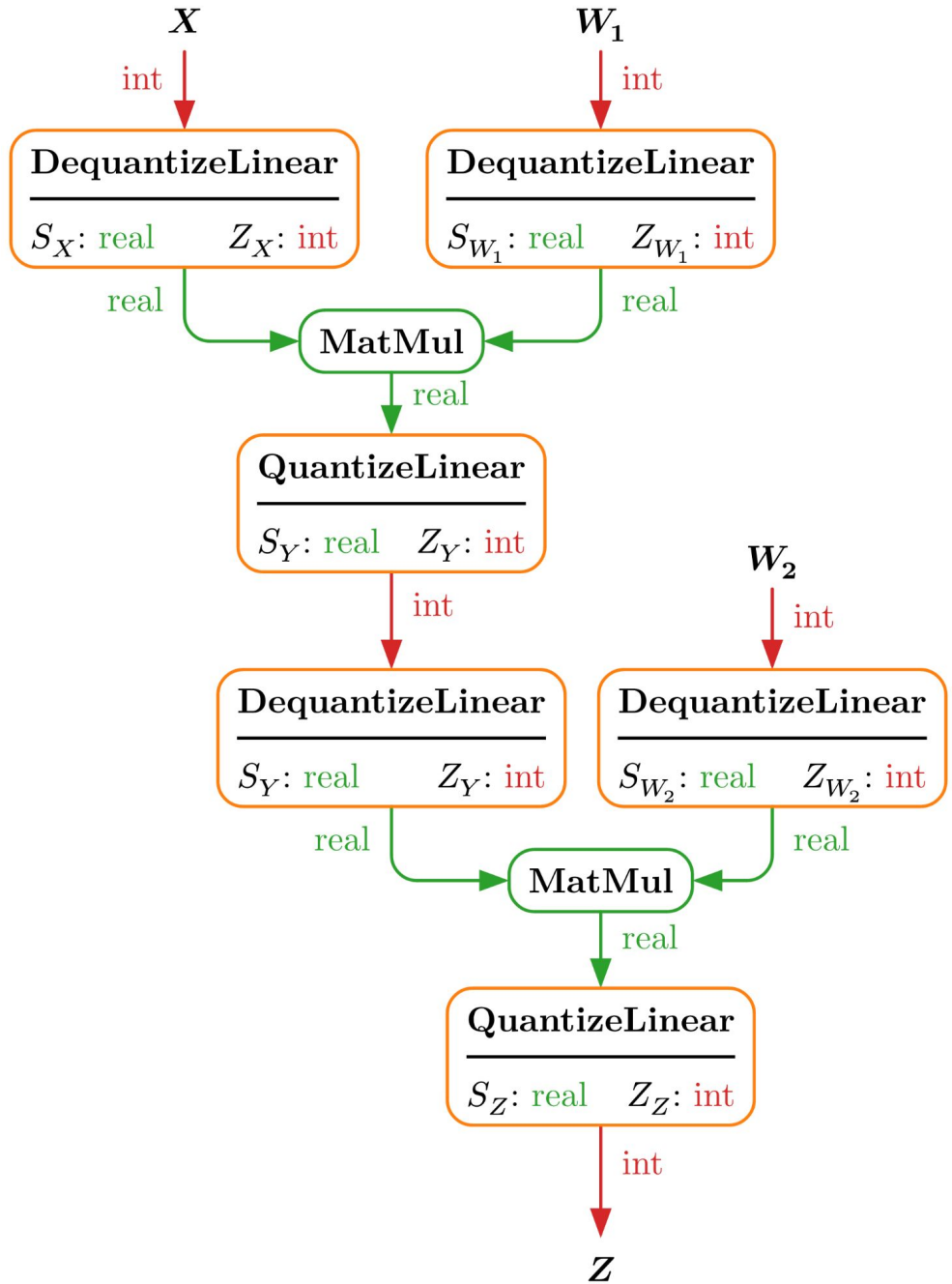
Quantize:
$$Y = \text{saturate}\left(\text{round}\left(Y_{\text{real}}/S_Y\right) + Z_Y\right)$$

1. Use Dedicated DequantizeLinear / QuantizeLinear Nodes

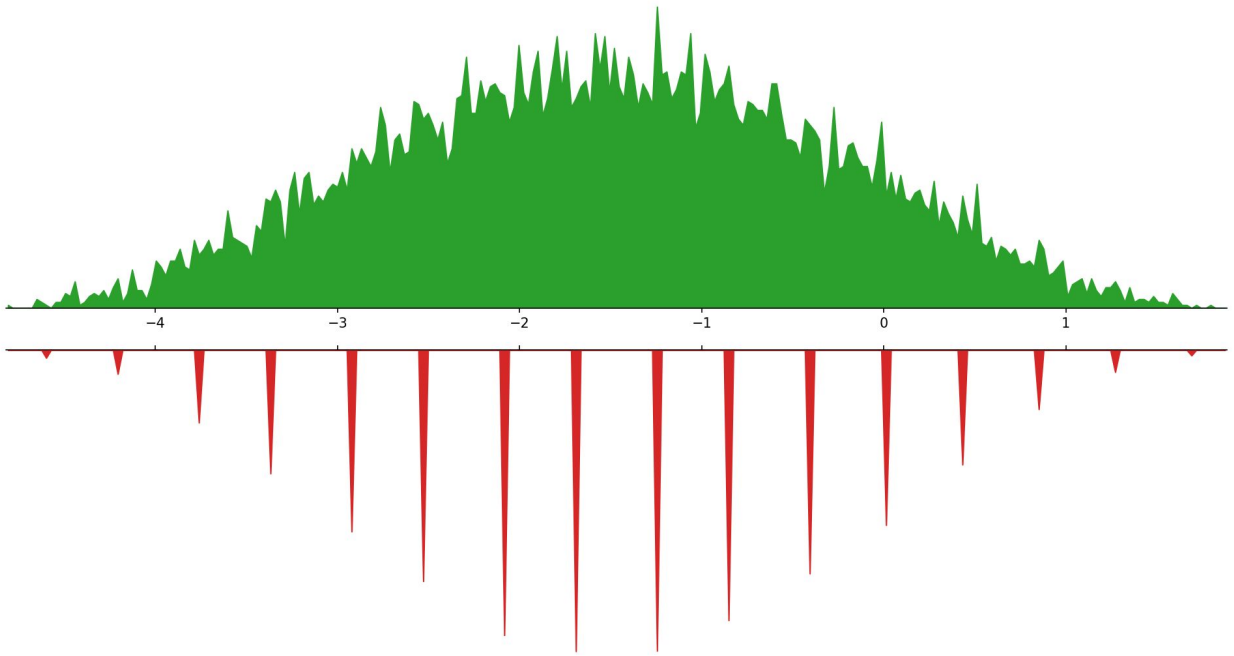







X
int
W1
int
DequantizeLinear
S_X: real Z_X: int
DequantizeLinear
S_{W_1}: real Z_{W_1}: int
real
real
MatMul
real
QuantizeLinear
S_Y: real Z_Y: int
int
W2
int
DequantizeLinear
S_Y: real Z_Y: int
DequantizeLinear
S_{W_2}: real Z_{W_2}: int
real
real
MatMul
real
QuantizeLinear
S_Z: real Z_Z: int
int
Z

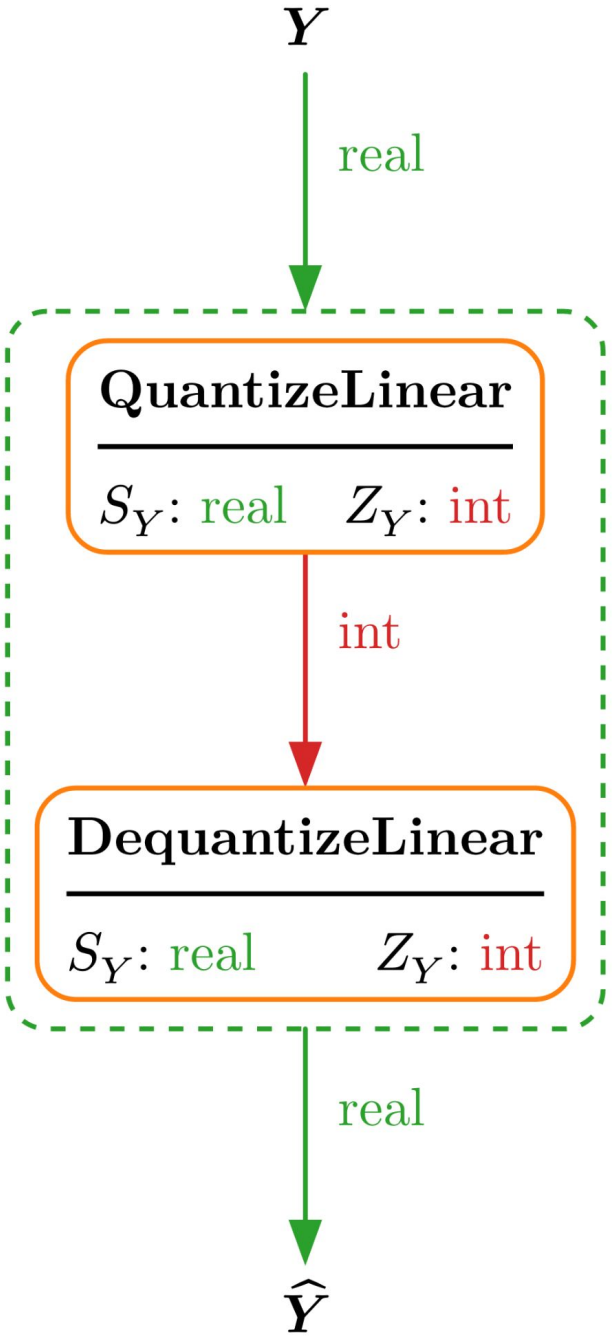
2. Hide ints inside QDQ pairs



Only sources of errors **should be**:

1. Rounding and saturation explicitly modeled by the FakeQuantize layer

2. Minor IEEE 754 floating point rounding errors (think -ffast-math, reduction order shenanigans, etc)



QDQ

QuantizeDequantize

FakeQuantize

Bonus: Mixed Input / Output Precision

Some layers may not support efficient quantized computation. We may want to mix "quantizable" and "not quantizable" layers in the same graph. For example, consider:

$$\text{QLinearMatMul} \rightarrow \text{Dequantize} \rightarrow \text{NodeThatOnlySupportsReals}$$

In this case, we are effectively computing

$$\underbrace{\text{Dequantize} \rightarrow \text{MatMul} \rightarrow \text{Quantize}}_{\text{fused integer kernel}} \rightarrow \text{Dequantize} \rightarrow \text{NodeThatOnlySupportsReals}$$

But the Quantize $\rightarrow$ Dequantize pair in the middle is basically a no-op, except worse, because we are forced to do extra computations **and** lose precision due to rounding.

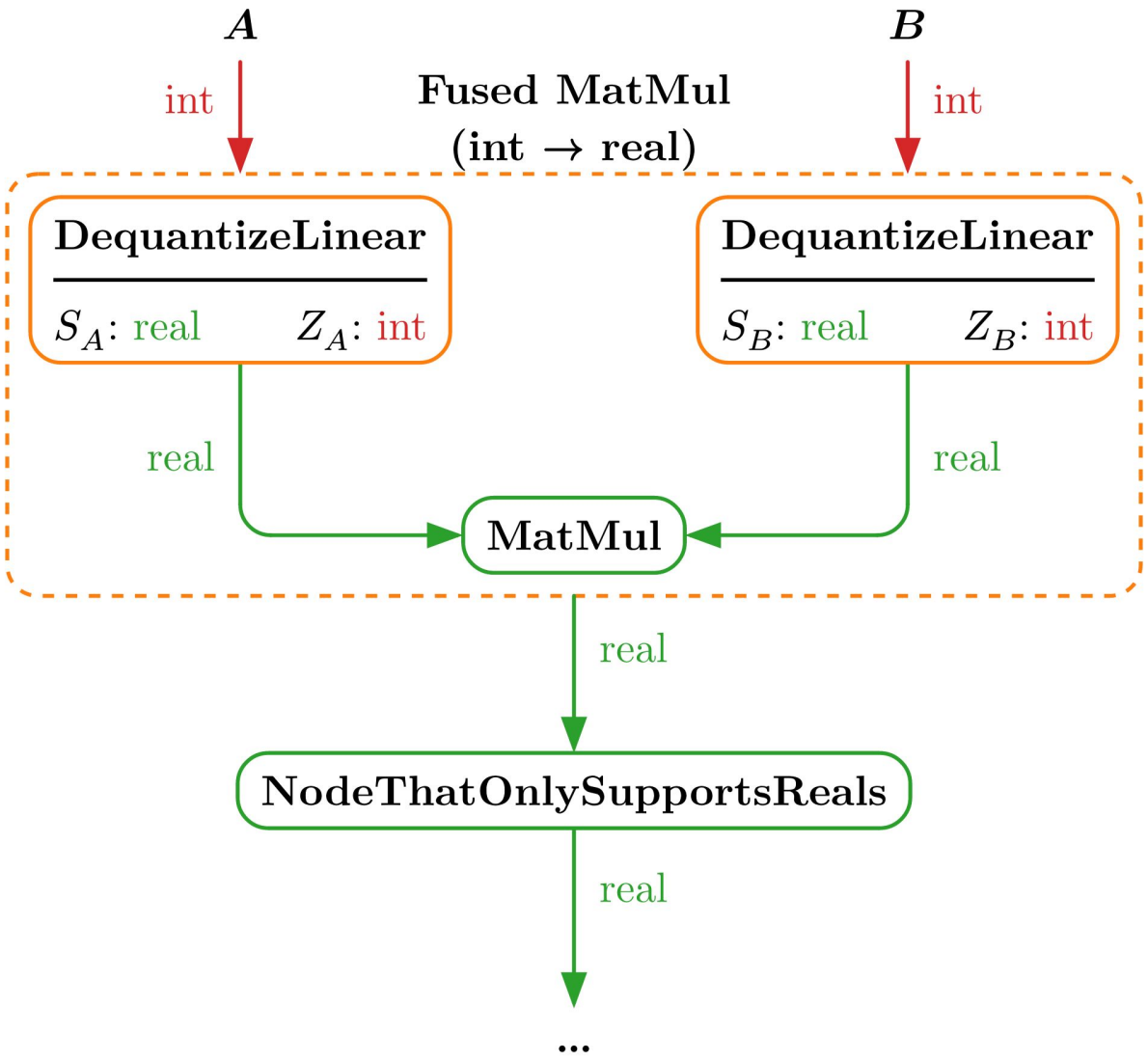
Bonus: Mixed Input / Output Precision

$$Y = \text{saturate}\left(\underbrace{\text{round}\left(\underbrace{\left[(\overset{\text{int}}{A} - \overset{\text{int}}{Z_A}) \times (\overset{\text{int}}{B} - \overset{\text{int}}{Z_B}) \right]}_{\text{int}} \cdot \overbrace{\frac{S_A \cdot S_B}{S_Y}}^{\text{real}} \right)}_{\text{int}} + \overset{\text{int}}{Z_Y} \right)$$

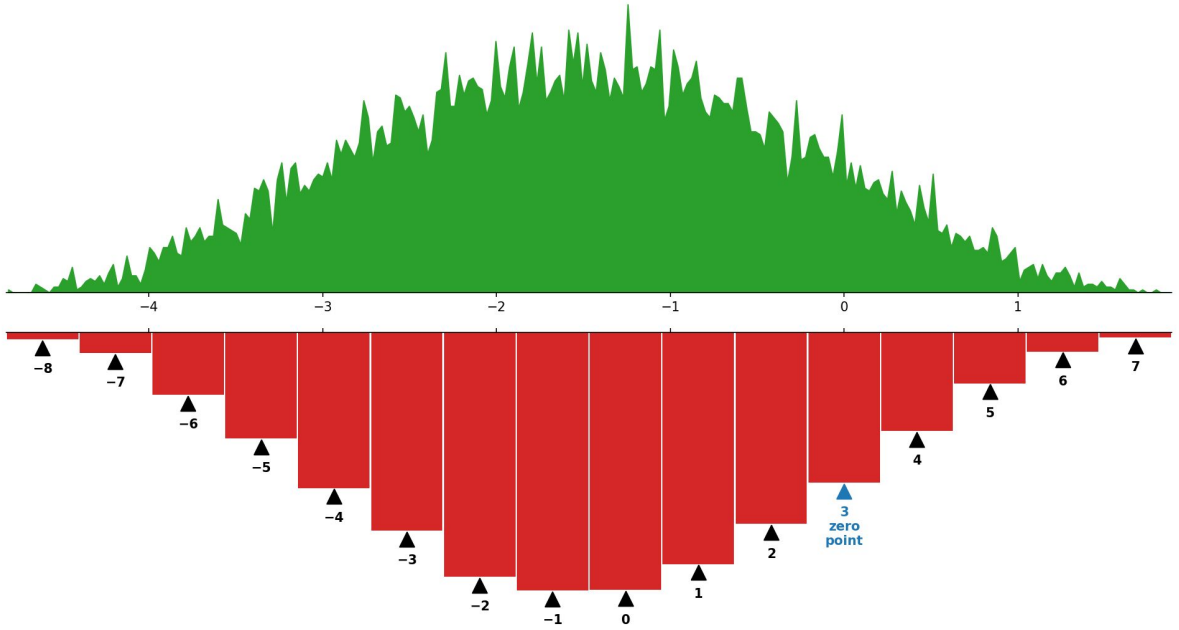
Bonus: Mixed Input / Output Precision

$$Y_{\text{real}} = \underbrace{\left[(\overset{\text{int}}{A} - \overset{\text{int}}{Z_A}) \times (\overset{\text{int}}{B} - \overset{\text{int}}{Z_B}) \right]}_{\text{int}} \cdot \underbrace{S_A \cdot S_B}_{\text{real}}$$

Bonus: Mixed Input / Output Precision



Post Training Quantization (PTQ)



$$Y_{\text{int}} := \text{saturate}\left(\text{round}\left(Y_{\text{real}}/S_Y\right) + Z_Y\right)$$

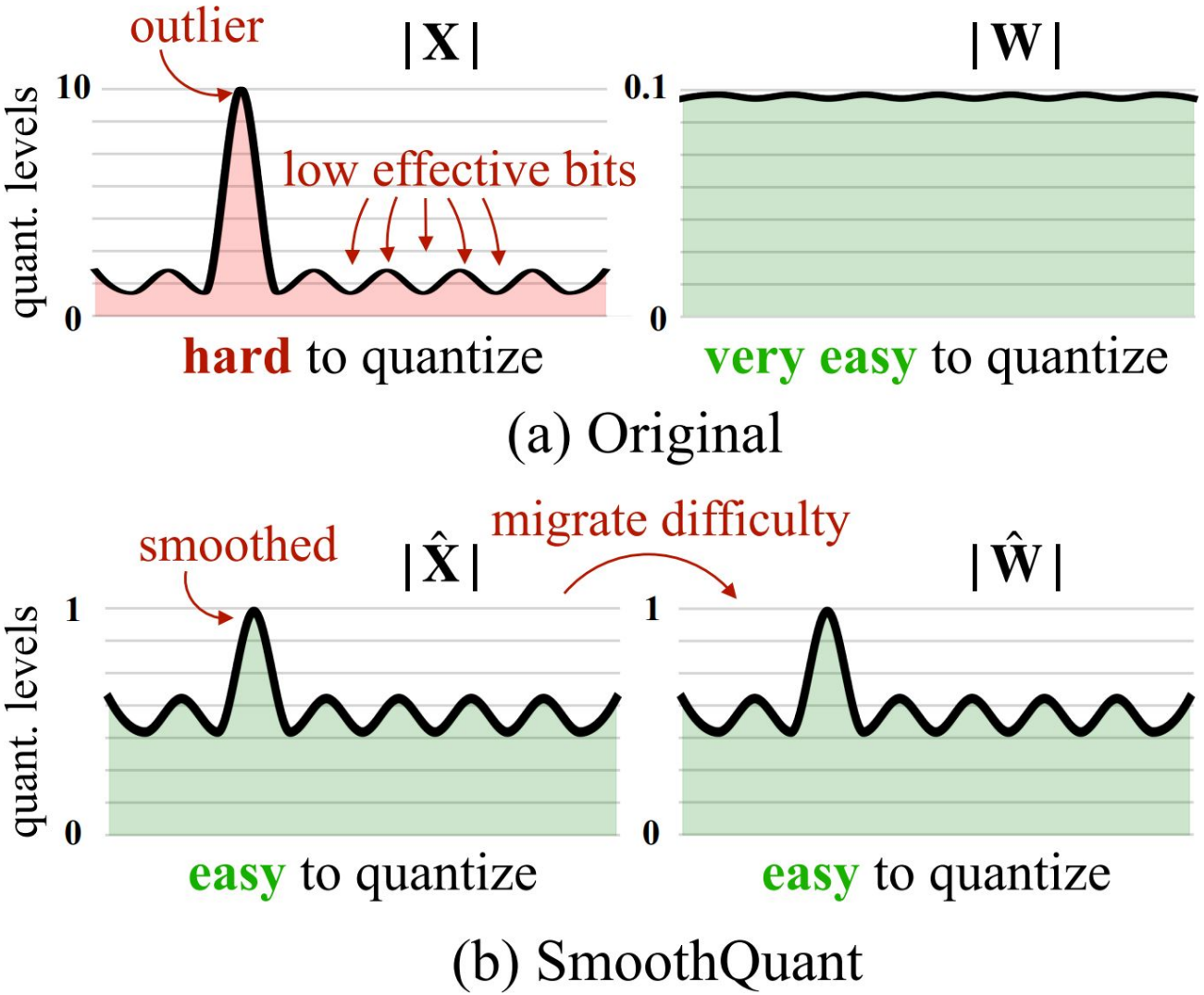
$$Y_{\text{real}} \approx \left(Y_{\text{int}} - Z_Y\right) \cdot S_Y$$

How to choose $S_Y \in \mathbb{R}$ and $Z_Y \in \mathbb{Z}$ to maximize precision?

SmoothQuant

$$Y_{\text{real}} = \underbrace{\left[(\overset{\text{int}}{A} - \overset{\text{int}}{Z_A}) \times \overset{\text{int}}{W} \right]}_{\text{int}} \cdot \underbrace{S_A \cdot S_W}_{\text{real}}$$

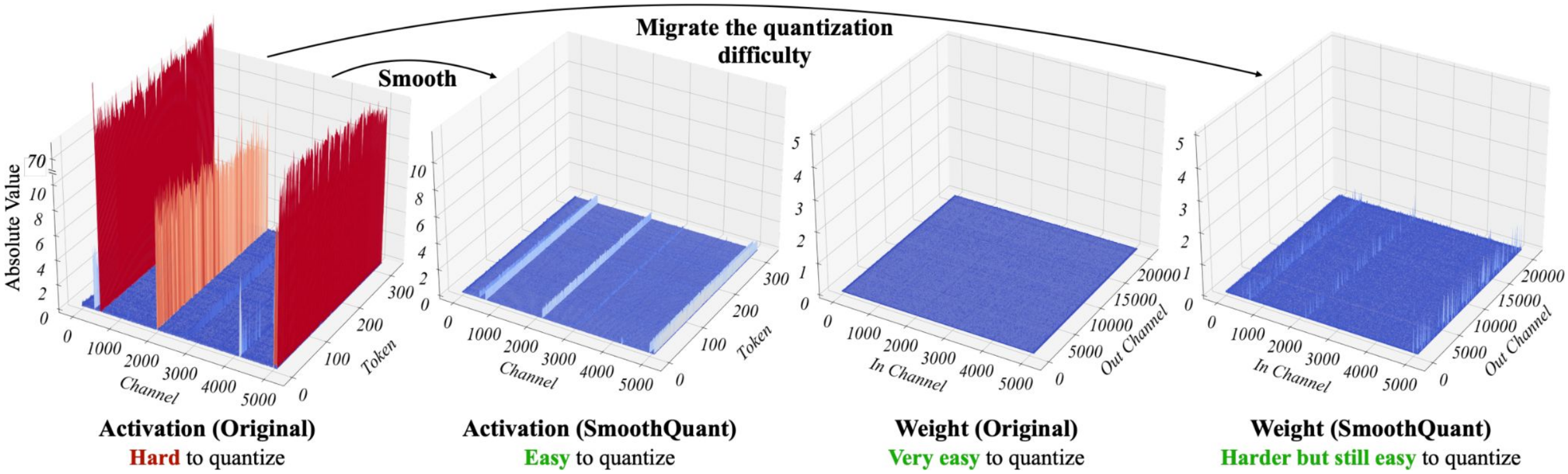
SmoothQuant



"SmoothQuant: Accurate and Efficient PTQ for LLMs"

ICML 2023. Xiao, et al.

SmoothQuant



"SmoothQuant: Accurate and Efficient PTQ for LLMs"

ICML 2023. Xiao, et al.

SmoothQuant

Before quantizing activations and weights, premultiply each activation row and corresponding weight column by a correction factor:

$$M_j = \frac{\max\left(|A_j|\right)^{\alpha}}{\max\left(|W_j|\right)^{1-\alpha}}, \text{ then replace}$$

$$A \times W = \left(A \times \text{diag}\left(M^{-1}\right)\right) \times \left(\text{diag}(M) \times W\right)$$

“SmoothQuant: Accurate and Efficient PTQ for LLMs”

ICML 2023. Xiao, et al.

In Practice

Requirements for fast quantization inference:

1. Good Hardware Support
2. Good Inference Framework
3. Good Quantization Framework

Inference Frameworks Matter

The QDQ format makes no guarantees about how each DQ → X → Q subgraph will be computed:

- If the inference framework knows how to fuse the DQ → X → Q subgraph into a single kernel, then it most likely **will be computed using integer arithmetic**

- Otherwise, the DQ, X and Q nodes will be computed separately:
  - The previous DQ node returns floating point values
  - The X node is computed **like in a normal non-quantized graph**
  - The next Q node receives floating point values

For each “unfused” QDQ node in the graph, you are **paying the overhead** of static quantization **without receiving the benefits** of using integer arithmetic for the “real” nodes / operators!

Introspection Tooling

- Netron
  - ONNX
  - OpenVino
    - imported graph (after load)
    - runtime graph (after compile)

- ONEDNN_JIT_DUMP=1 + binaryninja

Quantization Frameworks Matter

- in PyTorch
  - torch.ao (legacy)
    - quantization.prepare/convert
    - quantization.quantize_fx
    - quantization.quantize_pt2e
  - torchao (beta / unstable)
    - quantization.quantize_
- in ONNX / backend-agnostic
  - onnxruntime.quantization
  - optimum (by huggingface)
- in TensorRT
  - TensorRT “calibration cache” API (legacy)
  - TensorRT “pytorch_quantization” (legacy-ish)
  - TensorRT-Model-Optimizer (recently updated) ← 💡 might be worth revisiting

Quantization Frameworks Matter

What I **recommend**:

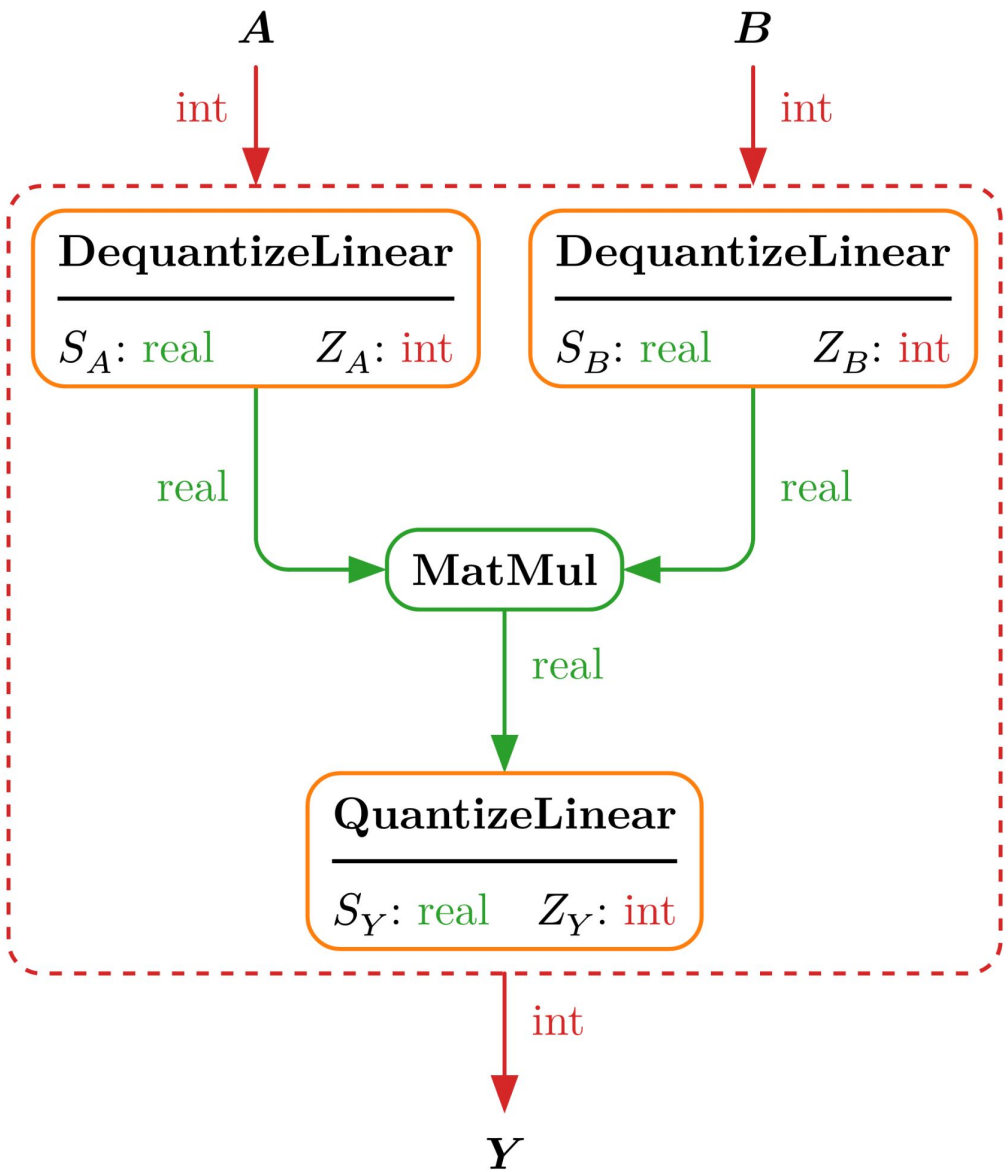
- Neural Network Compression Framework (NNCF)
  - from the authors of OpenVINO
  - QDQ node placement takes into account OpenVINO inference capabilities
  - the framework is “backend-agnostic”, it can insert QDQ nodes into:
    - OpenVINO graphs
    - ONNX graphs
    - Tensorflow models
    - PyTorch torch.fx graphs
    - PyTorch **torch.nn.Module** models

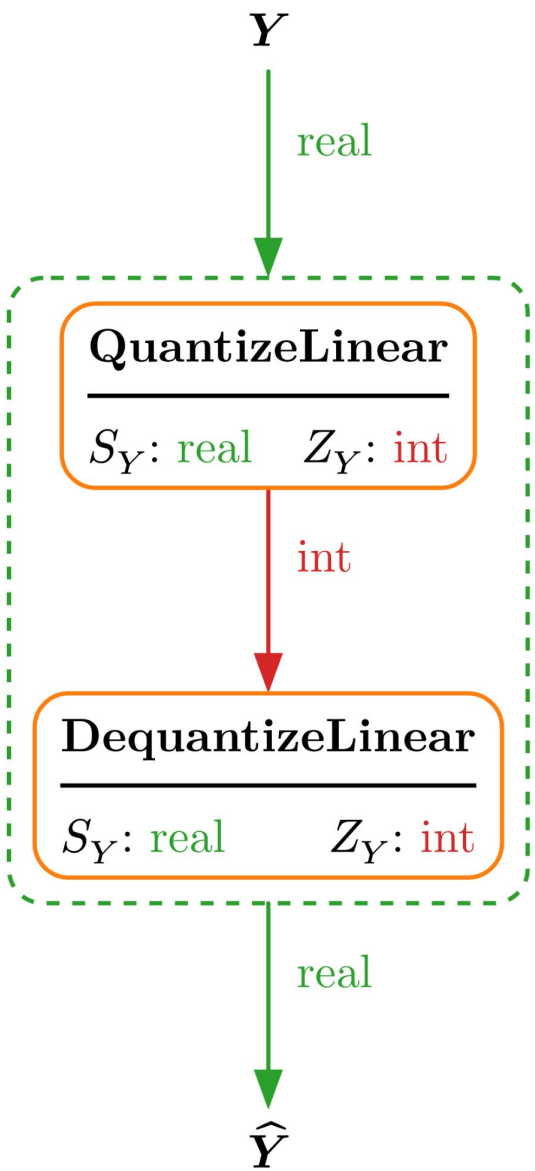
Quantization Frameworks Matter

What I **recommend**:

- Neural Network Compression Framework (NNCF)
  - **mainly targets modern CPUs**
    - optimized for Intel x86-64 CPUs
    - in practice seems to give reasonably good results for
      - Intel x86-64
      - AMD x86-64
      - ARM Cortex-A
  - supports “GPU” devices (this means integrated Intel Iris Xe or dedicated Intel ARC, not NVIDIA)
  - supports “NPU” devices (low power / mobile Intel NPUs, might be interesting to investigate further)

OverflowFix





QDQ

QuantizeDequantize

FakeQuantize

OverflowFix

Only sources of errors **should be**:

1. Rounding and saturation explicitly
   modeled by the FakeQuantize layer

2. Minor IEEE 754 floating point rounding errors
   (think -ffast-math, reduction order shenanigans, etc)

OverflowFix



Inference library authors be like

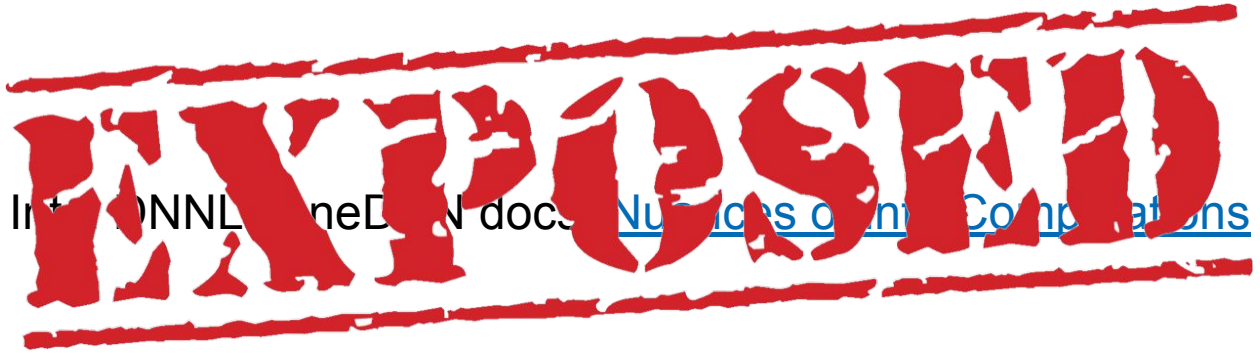
OverflowFix

Intel DNNL / oneDNN docs: [Nuances of int8 Computations]

oneDNN implements matrix multiplication such as operations with u8 and s8 operands on the Intel AVX2 and Intel AVX512 Instruction Set by using a sequence of `VPMADDUBSW, VPMADDWD, VPADDD` instructions [1] :

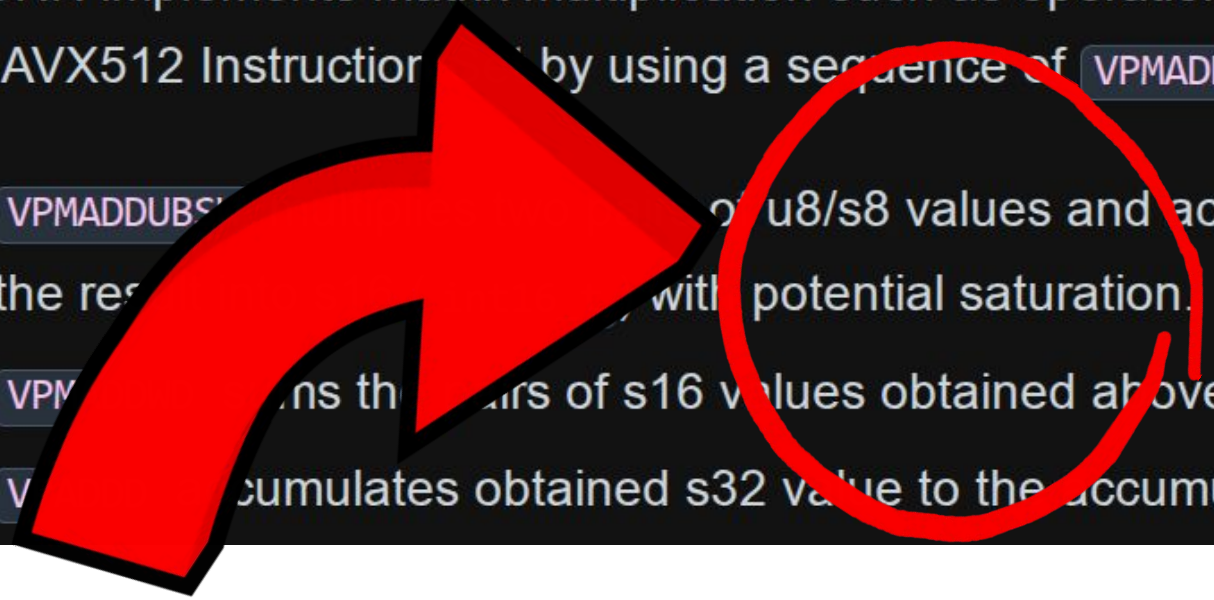
1. `VPMADDUBSW` multiplies two pairs of u8/s8 values and accumulates the result into s16 (`int16_t`) with potential saturation.
2. `VPMADDWD` sums the pairs of s16 values obtained above into s32. Computed sum is exact.
3. `VPADDD` accumulates obtained s32 value to the accumulator.

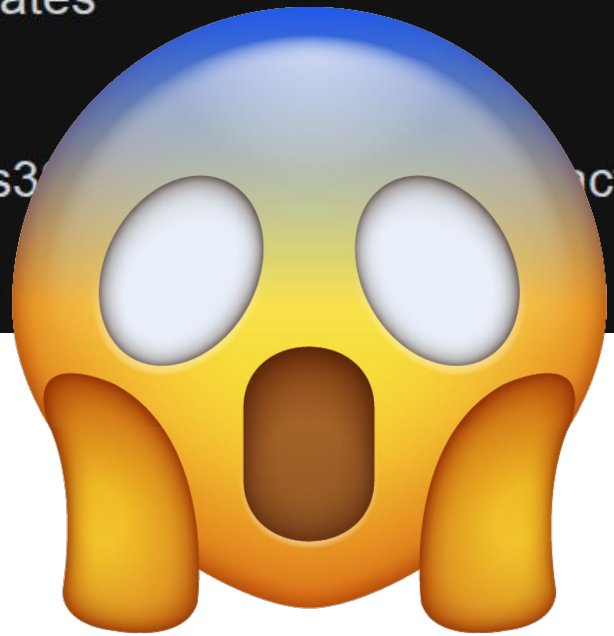
OverflowFix



oneDNN implements matrix multiplication such as operations with u8 and s8 operands on the Intel AVX2 and Intel AVX512 Instruction[illegible] by using a sequence of `VPMADDUBSW, VPMADDWD, VPADDD` instructions [1] :

1. `VPMADDUBS[illegible]` [illegible] u8/s8 values and accumulates the res[illegible] with potential saturation.
2. `VPM[illegible]` [illegible]ns th[illegible]irs of s16 values obtained above into s3[illegible]ct.
3. `V[illegible]` [illegible]cumulates obtained s32 value to the accumulator.





OverflowFix

Good news 1: NNCF has built in support for custom “int7” quantization mode that mitigates this issue

When one input is of type u8 and the other one is of type s8, oneDNN assumes that it is the user’s responsibility to choose the quantization parameters so that no overflow/saturation occurs. For instance, a user can use u7 `[0, 127]` instead of u8 for the unsigned input, or s7 `[-64, 63]` instead of the s8 one. It is worth mentioning that this is required only when the Intel AVX2 or Intel AVX512 Instruction Set is used.

`class nncf.OverflowFix` [source]

This option controls whether to apply the overflow issue fix for the 8-bit quantization.

If you are going to infer the quantized model on the architectures with AVX-2, and AVX-512 instruction sets, we recommend using FIRST_LAYER option as lower aggressive fix of the overflow issue. If you still face significant accuracy drop, try using ENABLE, but this may get worse the accuracy.

PARAMETERS:

- **ENABLE** – All weights of all types of Convolutions and MatMul operations are quantized using a half of the 8-bit quantization range.
- **FIRST_LAYER** – Weights of the first Convolutions of each model inputs are quantized using a half of the 8-bit quantization range.

OverflowFix

Good news 2: processors that AVX-VNNI or AVX512-VNNI don’t have this problem

1.2. Processors with the Intel DL Boost Support

System examples: Intel Xeon Scalable processor x2xx series (formerly Cascade Lake).

Intel DL Boost brings the `VPDPBUSD` instruction [2], which enables computing the sum of four products of s8 and u8 values. This instruction performs same computations as the sequence of `VPMADDUBSW, VPMADDWD, VPADDD` instructions shown above, but with the major difference that the intermediate overflow and saturation cannot occur.

Quantization Aware Training (QAT)

1. insert the QDQ/FakeQuantize layers and initialize zero_point/scale params by PTQ

2. fine-tune the model as normal
   - no special loss
   - no modifications to train loop

3. convert torch.quantization.FakeQuantize modules to an ONNX-compatible format

4. torch.onnx.export the model (set dynamo=True and use a reasonably modern opset_version)

Results

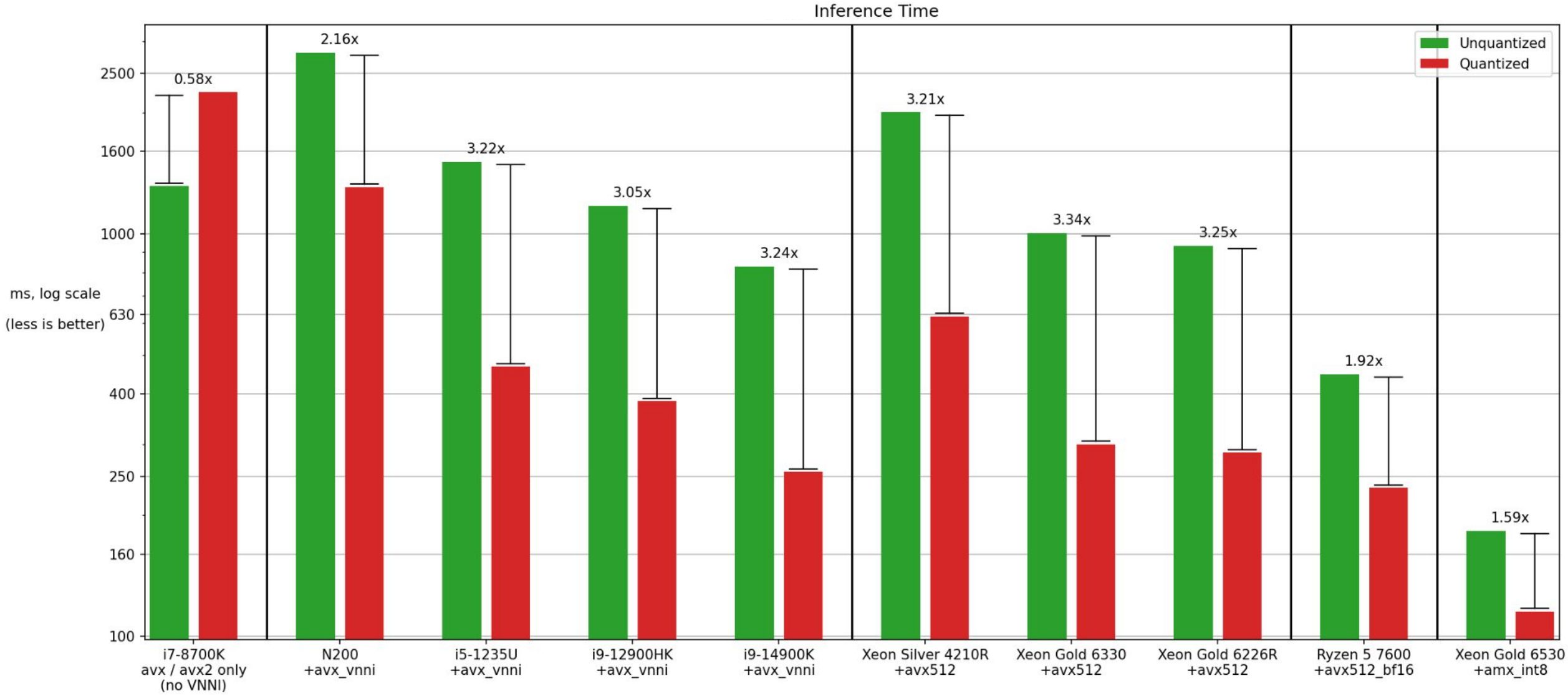
Quantization Results: Quality

It seems that quantization works **better** for larger models:

- “Large” models (~vit-large) give fairly good results even with simple PTQ
- “Medium” models (~vit-small) have a small drop in quality when using PTQ only, you can recover most of the degraded precision with QAT
- “Small” models (~small mobilenets) have a very big drop in quality when using PTQ only, you can recover **some of** the degraded precision with QAT

Recommendation: start quantization experiments with larger models

Quantization Results: Inference Speed



Quantization Results: Inference Speed

