TÉLÉCOM PARIS · SETI MASTER · LECTURE 02

Quantization

Fewer bits, same function: a numerical contract with the hardware

02

Number formats, scales and rounding, calibration, PTQ and QAT, the Vitis AI DPU flow

Wednesday 6 January 2027 · Session 2 of 8 · lecture + Lab 2

Where we are

S1 9 Dec	S2 6 Jan	S3 13 Jan	S4 27 Jan	S5 3 Feb	S6 10 Feb	S7 24 Feb	S8 3 Mar
L01 + Lab 1	**L02 + Lab 2**	L03 + Lab 3	L04 + Lab 4	Lab 5	Lab 6	Lab 7 (I)	Lab 7 (II)

What we keep from S1

Three currencies
operations (MACs), bytes moved, and time. Moving a number from DRAM costs far more than multiplying it.

The accelerator sets the rules
the KV260 DPU executes INT8 arithmetic on a fixed set of operators, from a batch-1 XMODEL.

One tag = one experiment
every number comes from a run folder: `out/runs/<TAG>/`, now with `quant/<QTAG>/`.

Today's question

Every weight and activation you trained in Lab 1 is a 32-bit float. The DPU only speaks 8-bit integers.

How many bits does each number really need – and who decides where they go?

S1 told us what to count. Today we change the size of every number.

THE THREAD – TODAY'S ANGLE

How much useful intelligence can we obtain from a limited budget of computation, memory and energy?

Precision is a budget like any other. A 32-bit float spends most of its bits on a dynamic range that a trained tensor never uses.

Today we give each tensor exactly the range it needs, store it in 8 bits – and learn to check that the network did not notice.

By the end of this session you can

Compute

by hand

a scale, an integer code, a reconstruction and its error; a fixed-point position f from a range.

Explain

with the right words

rounding vs clipping error, symmetric vs affine, per-tensor vs per-channel, calibration, CLE, the STE.

Choose

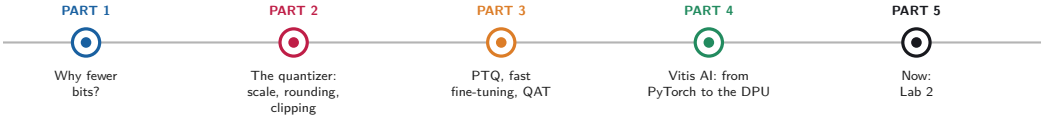
the cheapest method that works

PTQ first; better calibration, fast fine-tuning or QAT only when a stated tolerance is missed.

Deploy

on the KV260

inspect, calibrate, test, export a batch-1 XMODEL, compile, run with XIR/VART – and compare PTQ with QAT on the board.



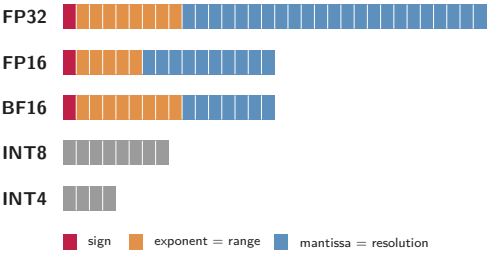
PART 1

Why fewer bits?

1

What a number format spends its bits on, what 8 bits buy in silicon – and why a trained network does not mind.

A number format spends its bits on range or on resolution



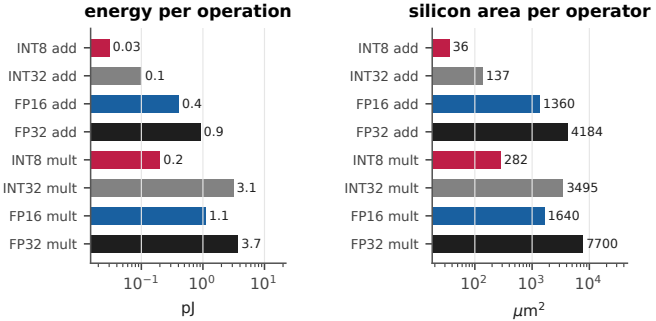
	largest value	relative error	step	typical use
FP32	3.4×10^{38}	6×10^{-8}	$\propto \lvert x \rvert$	training, reference
FP16	65 504	5×10^{-4}	$\propto \lvert x \rvert$	GPU inference
BF16	3.4×10^{38}	4×10^{-3}	$\propto \lvert x \rvert$	training on TPUs/GPUs
INT8	$127 \cdot s$	$s/(2\lvert x \rvert)$	s	DPU, NPUs, MCUs
INT4	$7 \cdot s$	$s/(2\lvert x \rvert)$	s	weights of LLMs

Floats keep the relative error bounded (unit roundoff 2^{-24}, 2^{-11}, 2^{-8} for normal values) and the step grows with $|x|$. Integers keep one step s: the relative error grows as $|x|$ shrinks. A float carries a scale in every number; an integer tensor shares one scale s.

A trained tensor often spans a limited range: a suitable INT8 scale can preserve accuracy. Check calibration and measured error.

Floating point buys dynamic range that a single trained tensor does not need. Integers spend every bit on resolution.

What 8 bits buy in silicon



Data: M. Horowitz, *Computing's Energy Problem*, ISSCC 2014 (45 nm). Absolute values shrink with newer nodes; the ratios survive.

4×

fewer bytes to store and to move than FP32: model size, DDR traffic, on-chip buffers

18×

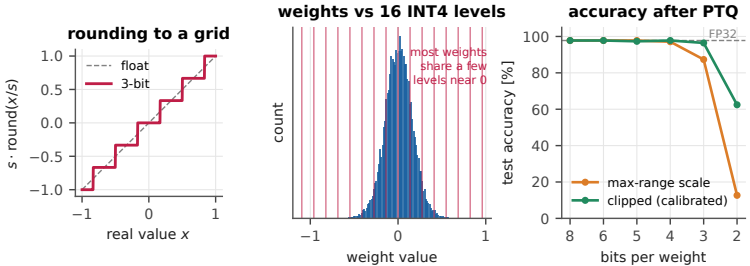
less energy per multiply; multiplier area grows roughly with the square of the bit width

2 per DSP

a DSP48E2 slice (27×18 multiplier) computes two INT8 products that share an operand – FP32 needs several DSPs plus LUTs (AMD WP486)

On an FPGA the same silicon gives you many more INT8 MACs per clock – the raw INT8 tensor payload is four times smaller than FP32; actual transfer speed depends on the memory system.

Recap from S1: quantization in one picture



Left: rounding to a grid of 2^b levels is a staircase. The error it adds is bounded by half a step.

Middle: trained weights are bell-shaped – most of them share the few levels near zero.

Right: why does the network not mind? Its decision only needs the right logit to stay on top; small, bounded noise rarely changes the argmax.

Computed for this course: 3-layer MLP (26 k parameters) on scikit-learn `digits`, per-tensor symmetric weight quantization, no retraining.

Quantization injects bounded noise. Today is about keeping that noise small where it matters and knowing when it is not.

PART 2

The quantizer: scale, rounding, clipping

2

One formula, two errors, one knob – and every design choice the DPU makes for you: symmetry, granularity, power-of-two scales, calibration.

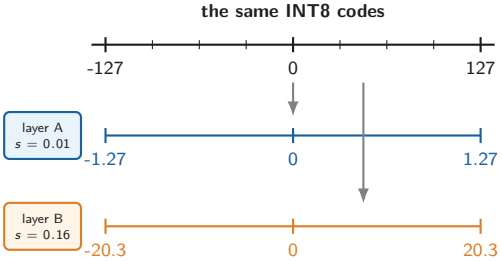
An integer code plus a real scale

$$x \approx \hat{x} = s \cdot q$$

q: the integer code
stored in 8 bits, signed: $q \in [-128, 127]$ on the DPU. Fixed by the hardware.

s: the scale
what one integer step means in real units. Chosen per tensor, from the weights or from calibration data.

Think of s as the spacing of the marks on a **ruler** with 256 marks. You cannot add marks – you can only choose how far apart they are.



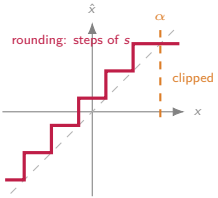
Layer A resolves steps of 0.01 but cannot represent 2. Layer B represents ± 20 but maps both 0.03 and 0.07 to zero.

The integer range is fixed by the hardware. Choosing the scale – the real range those integers cover – is the whole problem.

Uniform symmetric quantization, step by step

$$s = \alpha / Q$$
$$q = \text{clip}\big(\,\text{round}(x/s),\, -Q,\, Q\big)$$
$$\hat{x} = s\,q$$

α: the **threshold** you choose; $Q = 2^{b-1} - 1$ (7 for INT4, 127 for INT8); $e = x - \hat{x}$. The DPU also uses the code -128.



INT4, $\alpha = 1$ the first table of Lab 2, script 03c

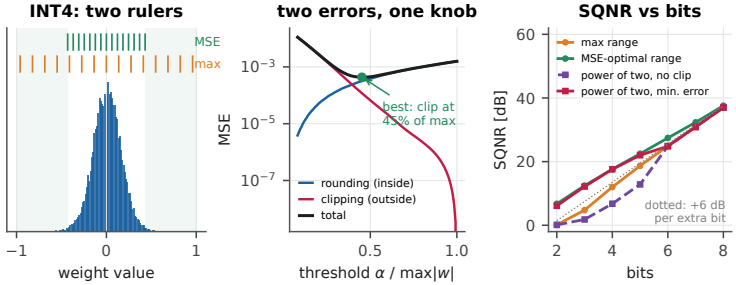
$s = 1/7 \approx 0.143$
$x = 0.55$: $0.55/0.143 = 3.85 \to q = 4 \to \hat{x} = 0.571$, $e = -0.021$ rounding
$x = 1.8$: $12.6 \to$ clip $\to q = 7 \to \hat{x} = 1.000$, $e = +0.80$ clipping

INT8, $\alpha = 2.54$

$s = 2.54/127 = 0.02$
$x = 1.20 \to q = 60 \to \hat{x} = 1.20$ exact
$x = 1.23 \to 61.5 \to q = 62 \to \hat{x} = 1.24$ $|e| \le s/2$
$x = 4.00 \to 200 \to q = 127 \to \hat{x} = 2.54$ $e = 1.46$!

Rounding error is small and everywhere; clipping error is rare but unbounded. Doubling α removes clipping – and doubles every step.

Two errors, one knob – measured on real weights



Left: the 15 INT4 levels placed by the max value (orange) waste most of them on empty tails. Clipping the tails (green) puts the levels where the weights are.

Middle: as α grows, rounding error rises ($\propto s^2/12$) and clipping error falls. Here the best INT4 threshold is **45% of the max**: 3.6× lower MSE.

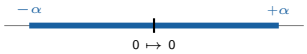
Right: each extra bit buys ≈6 dB. At 8 bits the range rule hardly matters; at 4 bits a good threshold is worth one bit. A power-of-two step that must not clip loses up to one more.

Computed for this course: layer-2 weights (128×128) of the `digits` MLP; symmetric, signed, narrow range.

Calibration is the choice of α. At INT8 it is forgiving; below 6 bits it decides whether the model survives.

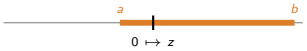
Symmetric or asymmetric: where does zero go?

Symmetric zero-point $z = 0$



$x \approx s\,q$. One multiplication per product; zero is exact (padding, ReLU).

Asymmetric (affine) zero-point $z \neq 0$



$x \approx s\,(q - z)$. Every code covers the real range $[a, b]$ – nothing is wasted.

What the hardware pays for z, in one dot product:

$$\sum_i (q_{w,i} - z_w)(q_{x,i} - z_x) = \sum_i q_{w,i} q_{x,i} - z_x \sum_i q_{w,i} - z_w \sum_i q_{x,i} + N z_w z_x$$

Weights: keep them symmetric

$z_w = 0$ removes the input-dependent term $z_w \sum q_x$; $z_x \sum q_w$ folds into the bias offline.

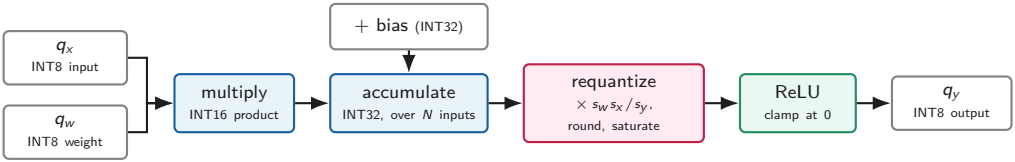
The price after ReLU

activations are ≥ 0: a signed symmetric tensor uses only codes $0 \ldots 127$ – **one bit lost**, for one simple MAC array.

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

Affine is more precise, symmetric cheaper; the DPU is symmetric throughout.

Inside the accelerator: the integer datapath of one layer



Why a 32-bit accumulator?

An INT8×INT8 product needs 16 bits; summing $N = 3 \cdot 3 \cdot 512 = 4608$ of them adds $\log_2 N \approx 13$ bits. Overflow would silently corrupt the output.

Why requantize?

The sum carries the scale $s_w s_x$; the next layer expects INT8 at its own scale s_y. One rescale per output value – not per MAC.

Where the floats went

Nowhere in the loop: scales are folded into one multiplier $M = s_w s_x / s_y$ per layer. If every scale is a power of two, M is a **bit shift**.

The expensive part – millions of MACs – is pure INT8. Real numbers survive only as one rescale per output, and on the DPU that rescale is a shift.

Power-of-two scales: what the DPU actually stores

Vitis AI records a bit width and a **fixed-point position** f:

$$q = \text{round}\left(x \cdot 2^{f}\right), \qquad \hat{x} = q \cdot 2^{-f}, \qquad s = 2^{-f}$$

Worked example

largest f that does not clip

weights, max $|w| = 0.77$: $\log_2(127/0.77) = 7.4 \Rightarrow f = 7$
step $2^{-7} = 0.0078$, range ± 0.99

activations, max $= 3.09$: $\log_2(127/3.09) = 5.4 \Rightarrow f = 5$
step $2^{-5} = 0.031$, range ± 3.97

requantize: $M = 2^{-(f_w + f_x - f_y)} \Rightarrow$ shift right by $f_w + f_x - f_y$

What you gain

no multiplier in the requantization, trivial to implement in the MAC array, identical results on every board.

What you pay

the step can only take values $\ldots, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \ldots$ – up to $2\times$ coarser than needed. Here the activation range is 3.97 for a max of 3.09: 22% of the codes unused.

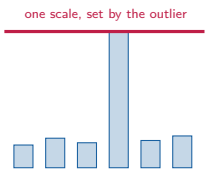
How Vitis AI limits the damage

its default method (`diffs`) may pick a larger f and *accept* some clipping when that lowers the error – the right-hand curve two slides ago.

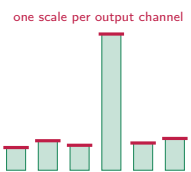
In Lab 2 you will read f for every tensor (script `04e`). A step of 2^{-f} is the same scale you computed by hand – restricted to powers of two.

One ruler per tensor, or one per channel?

per-tensor



per-channel



Why per-channel helps

output channel c of a conv layer only ever multiplies its own weights: giving it its own s_c costs one scale per channel in the requantization.

Where it matters

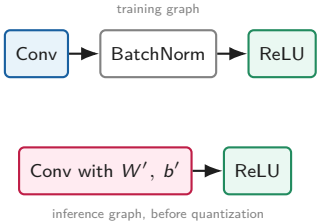
after BatchNorm folding and in depthwise convolutions, channel ranges can differ by 10–100×.

On the KV260 DPU

per-tensor only (`per_channel: false`). Lab 2 script `03d` lets you measure what that costs on a real `conv2`.

The best quantizer is not always the one your accelerator can run. When the hardware is per-tensor, the model must be made per-tensor friendly.

BatchNorm folding: one layer less, different weights to quantize



$$y=\gamma\,\frac{Wx+b-\mu}{\sqrt{\sigma^2+\epsilon}}+\beta \;\Rightarrow\; W'_c=\frac{\gamma_c}{\sqrt{\sigma_c^2+\epsilon}}\,W_c, \qquad b'_c=\frac{\gamma_c\,(b_c-\mu_c)}{\sqrt{\sigma_c^2+\epsilon}}+\beta_c$$

Why fold

at inference BN is a fixed per-channel affine map: merging it into the conv removes one layer and one memory round trip.

The catch

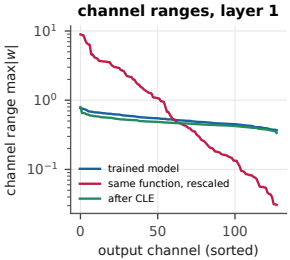
each output channel is multiplied by γ_c/σ_c. A channel with tiny variance gets a huge factor – the folded weights now have very unequal channel ranges.

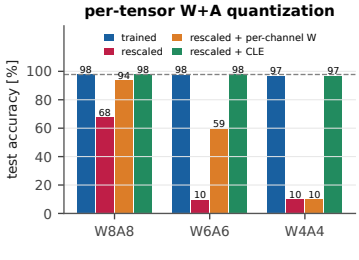
Who does it

the Vitis AI quantizer folds automatically; the ranges it calibrates are those of W', not the W you trained.

Folding is free in float. In a per-tensor integer world, it can create the outlier channels that ruin quantization.

Cross-layer equalization: rescale channels, keep the function





ReLU is positively homogeneous: $\mathrm{ReLU}(c\,x) = c\,\mathrm{ReLU}(x)$ for $c > 0$. Divide output channel k of layer i by s_k, multiply input channel k of layer $i{+}1$ by s_k: the float network computes *exactly* the same function. CLE picks $s_k = \sqrt{r_k^{(i)} r_k^{(i+1)}} \,/\, r_k^{(i+1)}$ so both channel ranges become equal.

We rescaled the neurons of a trained MLP by random factors in $[\frac{1}{16}, 16]$: **same function, same 97.8% in float**. Per-tensor W8A8 drops it to 68%, W6A6 to 10%. CLE restores 97.8%. Per-channel weights alone are not enough – the *activations* are imbalanced too.

Nagel et al., ICCV 2019. Computed for this course on the `digits` MLP; Lab 2 offers the same stress test (`03b_make_stress_model.sh`).

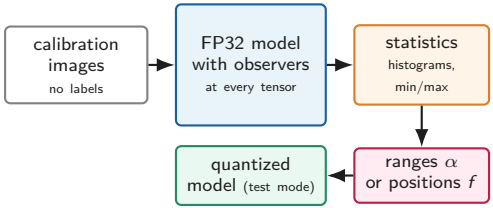
CLE is not calibration: it reshapes the weights *before* calibration, so that one ruler per tensor is enough.

Weights are known; activations must be observed

Weights
fixed after training: the whole distribution is in the checkpoint. Their ranges are computed directly.

Activations
exist only when an input flows through the network. Their range depends on the *data*.

Calibration
run a few hundred representative inputs through the float model, record statistics at every quantized tensor, fix the ranges. **No labels, no gradients, no training.**



The calibration images must go through the **same preprocessing** as in deployment: same resizing, same /255, same normalisation, same colour order. The observers cannot tell a preprocessing bug from a real distribution.

Calibration estimates ranges; it does not learn anything. Its only input is data – so its only failure mode is unrepresentative data.

Four calibration rules – four answers to "how much clipping?"

Max (min–max)

`maxmin` in Vitis AI

$\alpha = \max |x|$ over the calibration set. Never clips what it has seen; one outlier stretches the ruler for everyone.

layer-1 activations: $\alpha = 3.09$

Percentile

e.g. 99.99%

ignore the rarest values. Simple and robust to outliers; the percentile is one more hyper-parameter.

$\alpha = 2.95$

MSE

`diffs` in Vitis AI

choose α (or f) that minimises $\|x - \hat{x}\|^2$ on the observed values: the U-curve of the previous slides.

$\alpha = 2.92$

Entropy (KL)

TensorRT's calibrator

choose α so the quantized histogram stays closest (KL divergence) to the float one. Preserves the shape of the distribution.

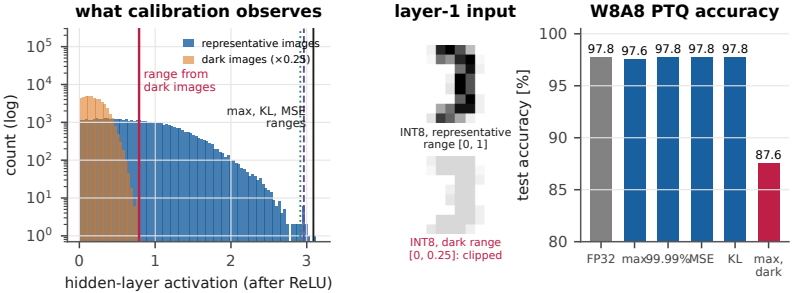
$\alpha = 2.96$

On our MLP all four give 97.6–97.8% at W8A8: when the calibration data are representative, the rule barely matters at 8 bits. The differences appear at 4 bits and below – and on tensors with heavy tails (attention, detection heads, image codecs).

Rules as implemented for this course on the `digits` MLP; S. Migacz, *8-bit Inference with TensorRT*, GTC 2017; AMD UG1414 (quantization strategy: `diffs`, `maxmin`).

Every rule decides how much to clip, each with its own measure of error.

Representative data matters more than the rule



Left: calibrate on the same images made 4× darker and every activation range shrinks ≈ 4× (red line vs black lines).

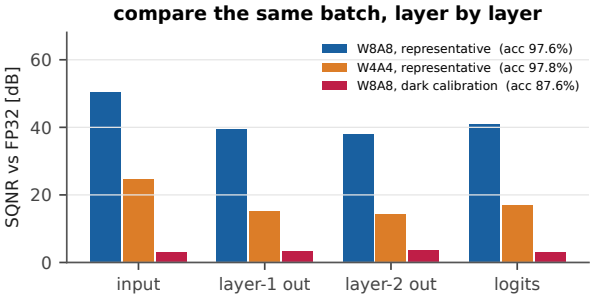
Middle: at test time normal images arrive – anything above the dark range is clipped. The network sees a flattened digit.

Right: the four rules agree within one image; the dark calibration costs **10 points** at W8A8 – with the same weights and the same bits.

Computed for this course (`digits` MLP, 512 calibration images, 450 test images: one image = 0.22 points). Lab 2 reproduces this on CIFAR-10: `QTAG=ptq_mismatch`, `CALIB_PROFILE=dark`, `MISMATCH_SCALE=0.25`.

A calibration set that does not look like deployment data is worse than no rule at all. Same preprocessing, same kind of images.

When accuracy drops: compare the same batch, layer by layer



SQNR of a tensor: $10 \log_{10} \frac{\|x\|^2}{\|x - \hat{x}\|^2}$, float vs quantized run, same inputs.

W8A8: ≈40 dB everywhere
noise $10^4\times$ weaker than the signal.

W4A4: ≈15 dB, still 97.8%
classification tolerates noise; regression does not.

Dark calibration: 3 dB
already at the input: the damage starts at the *first* quantizer. Check calibration and preprocessing, not the network.

Computed for this course on the digits MLP (450 test images).

Never debug a quantized model from its final accuracy alone. Find the first tensor where float and integer disagree.

PART 3

Who adapts to the rounding?

Post-training quantization, fast fine-tuning and quantization-aware training – three costs, three levels of recovery.

3

PTQ: quantize a trained model without training it



Why PTQ first

- minutes on a CPU; a few hundred unlabelled images;
- no optimiser, no hyper-parameter search, no risk of overfitting;
- at INT8, on most CNNs, the loss is a fraction of a point.

When PTQ is not enough

- fewer than 8 bits, or tensors with heavy tails;
- very compact models (depthwise convs, few channels): little redundancy to absorb noise;
- regression outputs – pixels, boxes, audio – where every dB counts.

PTQ never changes what the network learned; it only chooses rulers. That is both why it is cheap and why it can fail.

Between PTQ and QAT: correct the error you can measure

Bias correction

rounding errors do not average to zero per channel: $\mathbb{E}[\hat{W}x] \neq \mathbb{E}[Wx]$. Measure the shift of each output mean on calibration data and subtract it from the bias.

Adaptive rounding (AdaRound, AdaQuant)

rounding each weight to the *nearest* level is not optimal for the layer's output. Optimise, layer by layer, which way each weight rounds so that $\|Wx - \hat{W}x\|^2$ is minimal on unlabelled data.

Vitis AI fast fine-tuning

the AdaQuant-style option of `vai_q_pytorch`: a few hundred unlabelled images, layer-wise, no labels, no full training loop.

```
# calib mode, after the calibration pass
quantizer.fast_finetune(
    forward_only, # evaluation fn
    (quant_model, ft_loader, device))
quantizer.export_quant_config()

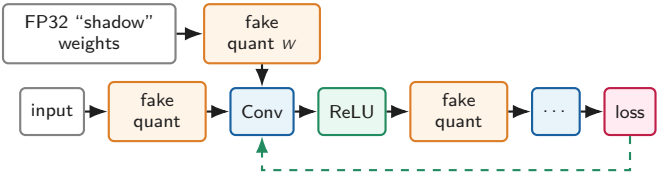
# test / export modes
quantizer.load_ft_param()
```

```
TAG=${TAG} QTAG=ptq_fast_ft \
CALIB_SAMPLES=512 CALIB_PROFILE=representative \
FAST_FINETUNE=1 FT_SAMPLES=512 \
bash scripts/04_ptq_export.sh
```

Nagel et al., ICCV 2019 and ICML 2020 (AdaRound); Hubara et al., ICML 2021 (AdaQuant); AMD UG1414, *Fast Finetuning*.

Fast fine-tuning: no labels, minutes. A negligible gain means PTQ sufficed.

QAT: let the network see the rounding while it trains



Fake quantization

quantize, then dequantize: values stay float, but only the representable ones survive.

The gradient problem

round is a staircase: its derivative is 0 almost everywhere – learning would stop.

Straight-through estimator

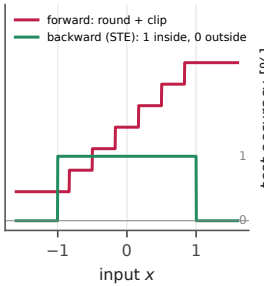
backward, pretend $\partial \hat{x} / \partial x = 1$ inside the range, 0 where clipped. A useful lie.

Bengio et al., arXiv:1308.3432, 2013; Jacob et al., CVPR 2018; Esser et al., *LSQ*, ICLR 2020.

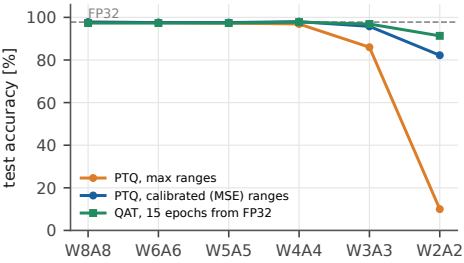
QAT does not remove the rounding. It moves the weights to where rounding hurts less.

PTQ or QAT? It depends on how many bits you ask for

the straight-through estimator



who adapts to the rounding?



Left: forward, the fake quantizer is a staircase; backward, the STE passes the gradient unchanged inside $[-\alpha, \alpha]$ and blocks it outside.

Right: at W8A8 there is nothing to recover – all three are within one or two test images of FP32. At W3A3 calibrated PTQ gives 95.8%, QAT 96.9%. At W2A2 (ternary!) max-range PTQ is at chance, calibrated PTQ reaches 82.2% and QAT 91.3%.

Computed for this course: `digits` MLP, W and A per-tensor symmetric; QAT = 15 epochs from the FP32 checkpoint with the STE.

QAT pays when PTQ fails: at low bit widths or on fragile models. At INT8, spend the effort on calibration data instead.

QAT in Vitis AI: mark the quantized region, train, convert

```python
from pytorch_nndct.nn import QuantStub, DeQuantStub
class Net(nn.Module):
    def __init__(self):
        ...
        self.quant, self.dequant = QuantStub(), DeQuantStub()
    def forward(self, x):
        x = self.quant(x) # INT8 region starts
        x = self.classifier(self.features(x))
        return self.dequant(x) # INT8 region ends
```

```python
from pytorch_nndct import QatProcessor
proc = QatProcessor(model, (torch.zeros(1, C, H, W),),
                    bitwidth=8, mix_bit=False)
trainable = proc.trainable_model() # fake-quantized
for x, y in train_loader: # ordinary loop
    opt.zero_grad(); crit(trainable(x), y).backward()
    opt.step()
deployable = proc.to_deployable(trainable, output_dir=qdir)
```

Model hygiene first

DPU-supported operators inside the stubs; module operations, not raw tensor arithmetic; preprocessing outside.

It is still PyTorch

same loss, same optimiser, a small learning rate (10^{-4}), a few epochs.

Same deployment path

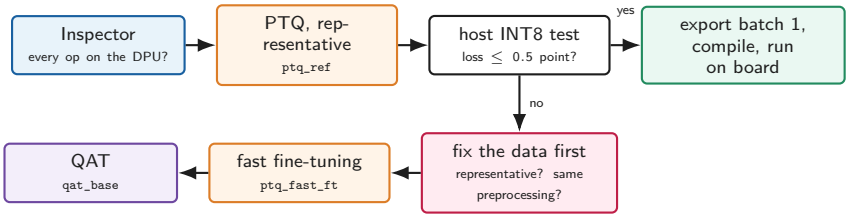
`deployable_model(...)`, one batch-1 forward, `export_xmodel`.

Read the gain honestly

part of a QAT gain may come from the extra training itself (`FLOAT_CONTROL=1`).

QAT changes how the quantized model is trained – not how it is deployed.

A decision procedure: escalate only when the tolerance is missed



A stated tolerance

Lab 2 accepts PTQ when $A_{\text{float}} - A_{\text{PTQ}} \leq 0.5$ point. Without a tolerance, "better" never ends.

Cheapest fix first

most PTQ failures at INT8 are data failures: dark, cropped or differently normalised calibration images.

A valid conclusion

"QAT gives the best INT8 accuracy, but representative PTQ already meets the tolerance and is the cheaper choice."

Choose methods, do not rank them. The right answer is the least expensive one that meets the requirement.

PART 4

Vitis AI: from PyTorch to the DPU

4

The contract the DPU enforces, the four objects of the quantizer, the batch-1 export, the compiler – and how the board runs and measures the result.

The DPU's numerical contract

Choice	DPU (KV260)	what it means
bit width	8	2^8 codes
signed	yes	$q \in [-128, 127]$
symmetric	yes	zero-point 0
granularity	per-tensor	one s per tensor
scale type	power of two	$s = 2^{-f}$
rounding	`std_round` (W, bias, input), `half_up` (activations)	ties away from 0 / up
range method	`diffs` (default) or `maxmin`	min. error / no clip

AMD UG1414 (Vitis AI User Guide); vai_q_pytorch 3.5, *Quant_Config.md* (DPU restrictions) and the checks in `quant_config.py`.

What the target gives you

`target="DPUCZDX8G_ISA1_B4096"` selects these defaults: the quantizer can only produce what the DPU can execute.

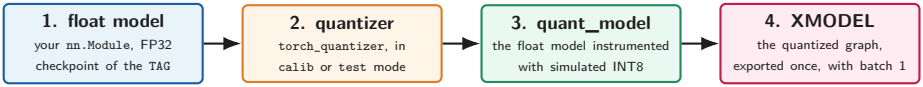
What not to promise

- INT4 or mixed precision on this DPU;
- asymmetric zero-points or per-channel activations;
- a graph that compiles just because PyTorch runs it.

A custom strategy file can *simulate* more than the DPU can *run*.

Lab 2 Part B plays with INT4 and per-channel to understand the numbers. The board path stays INT8, per-tensor, power of two.

Four objects to keep apart



Calibration writes a file, not a model

the ranges go to `quantize_result/` (`Quant_info.json`); test mode and export read them back.

`quant_model` is still PyTorch

you evaluate it with your usual loop – this is the "host INT8 accuracy".

Two XMODELs

the *quantizer* XMODEL (`quant/<QTAG>/`) is target-independent; the *compiled* XMODEL (`deploy/<QTAG>/`) is what VART runs.

Why "NNDCT" appears

`vai_q_pytorch` is implemented by `pytorch_nndct`: it traces the graph and inserts the quantizers.

At every step of Lab 2, name the active framework, the model state and the artifact being written.

`torch_quantizer`: inspect, calibrate, test, export

```python
from pytorch_nndct.apis import torch_quantizer, Inspector
example = torch.zeros(1, C, H, W) # shape template, batch 1
Inspector("DPUCZDX8G_ISA1_B4096").inspect(model, (example,))

# phase 1 -- calibration (04b): forward passes only
q = torch_quantizer("calib", model, (example,),
                    output_dir=qdir, target="DPUCZDX8G_ISA1_B4096")
for images, _ in calib_loader: # labels unused
    q.quant_model(images)
q.export_quant_config()
# phase 2 -- test (04c): reload ranges, INT8 accuracy
q = torch_quantizer("test", model, (example,),
                    output_dir=qdir, target="DPUCZDX8G_ISA1_B4096")
acc_int8 = evaluate(q.quant_model, test_loader)
# phase 3 -- export (04d): batch 1, ONE forward pass
images, _ = next(iter(export_loader)) # shape (1, C, H, W)
with torch.no_grad():
    q.quant_model(images)
q.export_xmodel(output_dir=qdir)
```

Inspector
predicts, per operator, whether the DPU can take it. It measures nothing; run it first, and record the *first* incompatibility.

Calibration
may use batches of 32; accuracy printed here is not the INT8 result.

Test
does not recalibrate – it reads the saved configuration.

Export
batch 1 and exactly one forward pass; `04_ptq_export.sh` chains the three phases.

Calibrate, test, export: three phases, one saved configuration.

The batch size is part of the exported graph

What the compiler says when the XMODEL was traced with the evaluation batch:

```
[UNILOG][WARNING] xir::Op{name = ..., type = conv2d-fix}
  DPU prefers ... input batch to be 1, but it is 128 now.
... const, reshape, bias ops assigned to CPU ...
```

What the Lab 2 export prints when it is right:

```
[DEPLOY] batch_size : 1
[DEPLOY] subset_len : 1
[DEPLOY] forward pass : exactly one
[DEPLOY] preflight : PASS - graph input batch is 1
```

One cause, many messages

`conv2d-fix` and `pool-fix` were recognised. The direct failure is the batch; the CPU assignments of constants and reshapes that follow are *cascading consequences*.

The fix

re-export with the dedicated batch-1 pass. Do not redesign the network from the cascade.

Why batch 1

the DPU processes one image per call; throughput comes from pipelining and threads, not from a batch dimension.

Train and calibrate with any batch; export with batch 1. Read the first compiler warning, not the fiftieth.

From the quantizer's XMODEL to the board's XMODEL

```
vai_c_xir \
  -x out/runs/${TAG}/quant/${QTAG}/model.xmodel \
  -a arch.json \
  -o out/runs/${TAG}/deploy/${QTAG} \
  -n model
```



arch.json
identifies the DPU configuration (B4096 on the KV260) by its fingerprint, which the board checks. Lab 2 supplies the one of the course boards, at the lab root.

Partitioning
supported operators are grouped into DPU instructions; anything else becomes a CPU subgraph.

Lab 2's rule
one batch-1 DPU computation partition, no CPU neural computation; only a terminal shape-preserving DPU-output `fix2float` passing the topology gate is allowed.

Never copy the quantizer's XMODEL to the board. The file VART runs is the one under `deploy/`.

On the board: XIR finds the DPU subgraph, VART runs it

```
graph = xir.Graph.deserialize("model.xmodel")
subs = graph.get_root_subgraph().toposort_child_subgraph()
dpu = [s for s in subs if s.has_attr("device")
       and s.get_attr("device").upper() == "DPU"][0]
runner = vart.Runner.create_runner(dpu, "run")
t_in = runner.get_input_tensors()[0]
t_out = runner.get_output_tensors()[0]

f_in = t_in.get_attr("fix_point") # e.g. 7
f_out = t_out.get_attr("fix_point")
x_q = np.clip(np.round(x * 2**f_in), -128, 127)
x_q = x_q.astype(np.int8) # quantize
job = runner.execute_async([x_q], [y_q])
runner.wait(job)
y = y_q.astype(np.float32) * 2**(-f_out) # dequantize
```



The application quantizes the input
with the input tensor's f – the same /255 and layout as in calibration, or the board accuracy drops.

The DPU is not the application
it does not open images, normalise pixels or compute accuracy: the Arm side does.

The f you read with `04e` on the host reappears on the board:
$q = \mathrm{round}(x \cdot 2^f)$ is now your code.

Four measurement levels – four different questions

Tool	What it measures	What it does *not* establish
VART application `vart_run_fpga.py`	task accuracy on the shared bundle; VART FPS (submit/wait) and application FPS	low-level DPU or memory bottlenecks
Raw DPU `xdputil benchmark`	throughput of the compiled DPU subgraph with N threads	accuracy; preprocessing; any application overhead
Trace `vai_trace_fpga.py` (`vaitrace`)	a timeline: where DPU, runtime and CPU time goes	correctness – it does not replace a labelled test

Power
`xmutil`
`xlnx_platformstats -p`
	SOM power (excluding carrier/camera) sampled around a controlled workload $\Rightarrow$ energy per input	DPU-only energy from a single sample

PTQ vs QAT: same architecture

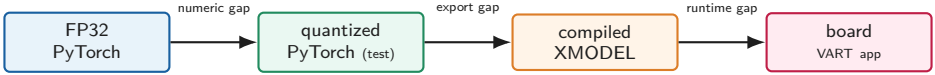
same INT8 operators $\Rightarrow$ same DPU throughput. Accuracy is the primary comparison; FPS a consistency check.

Energy is not power

$E = \bar{P} \cdot t$: same average power but a shorter run means less energy per image.

Every board number names its tool. Raw DPU FPS and application FPS answer different questions.

When the board disagrees with the host: three gaps



Symptom	First checks
host INT8 far below FP32	calibration profile and size; preprocessing identical to training; outlier channels (CLE); then fast fine-tuning or QAT
compiler moves layers to CPU	first direct warning: batch 128 $\Rightarrow$ re-export batch 1; otherwise an unsupported operator (Inspector report)
no DPU subgraph found	you copied the quantizer XMODEL instead of `deploy/<QTAG>/model.xmodel`
board accuracy far below host	/255 scaling, NCHW vs NHWC, the input `fix_point`, bundle checksum, model checksum
raw DPU FPS $\gg$ application FPS	preprocessing, buffer copies, Python overhead: not a DPU problem

Localise before you fix: save the outputs of the same batch at each stage and find the first stage where they diverge.

PART 5

Now: Lab 2

PTQ and QAT with Vitis AI on a CIFAR-10 CNN, compiled for the KV260 and compared on the board – one variable at a time.

5

Lab 2 in one picture

	Framework	What you do (scripts)	Artifact
A	ordinary PyTorch	supplied CIFAR-10 CNN (≈57%) or your own: `03_use_reference_model`, `03b_describe`	`float/model_float.pth`
B	PyTorch simulation	one tensor (03c), one weight channel per-tensor/per-channel (03d), one activation representative/dark (03e)	`concepts/` – never deployed
C	Vitis AI PTQ	inspect (04a), calibrate (04b), test (04c), export batch 1 (04d), read f (04e)	`quant/ptq_ref/`
D	Vitis AI QAT	`QuantStub`, `QatProcessor`: `04_qat_export.sh`	`quant/qat_base/`
E	compiler, XIR, VART	compile both (05), one shared bundle (06), board check and copy (07), paired table (09)	`deploy/`, `board_results/`
F	your choice	one controlled experiment, one new `QTAG`	`summary/results.csv`



Parts A–B teach the numbers; C–D produce two deployable variants; E runs both on the same board with the same data.

One variable at a time: the QTAGs you will create

QTAG	What changes (relative to `ptq_ref`)	The question it answers
`ptq_ref`	representative calibration, `CALIB_SAMPLES=512`	is PTQ within the 0.5-point tolerance?
`ptq_mismatch`	`CALIB_PROFILE=dark`, `MISMATCH_SCALE=0.25`	how much does unrepresentative calibration cost?
`ptq_fast_ft`	`FAST_FINETUNE=1`, `FT_SAMPLES=512`	does layer-wise recovery earn its cost?
`qat_base`	`QAT_EPOCHS=2`, `QAT_LR=0.0001`	does training with fake quantization help – and is it needed?

What stays fixed

the `TAG` (same float checkpoint), the test set, the preprocessing, INT8, the target DPU. Every row is a change from `float_source`.

Predict before you run

write down what you expect for `ptq_mismatch` using the calibration slide of this lecture – then check.

A QTAG is a controlled experiment: one change, one question, one row in `summary/results.csv`.

What your Lab 2 report must show

Model	Host accuracy	Host change vs FP32	KV260 accuracy	Interpretation
FP32 source	the checkpoint	+0.00 pt	–	visible reference; never deployed
PTQ `ptq_ref`	Vitis AI test mode	from the same source	board top-1	cheapest deployable reference
QAT `qat_base`	deployable model	from the same source	board top-1	training-based candidate

Same data

PTQ and QAT are run on the same bundle: same number of images, same SHA-256 – the script checks it.

Host vs board

each KV260 accuracy should agree with its host INT8 accuracy; a large gap is a runtime bug, not a quantization effect.

A decision

under the 0.5-point tolerance, which one would you deploy, and was the extra QAT training justified?

Accuracy first, FPS second, energy when measured – and a recommendation, not a ranking.

Before you start the lab

Checklist

1. One TAG for the whole lab
`export TAG=...`; every QTAG lives under it.

2. Part B before Part C
predict, run one command, read the table, change one parameter.

3. PTQ as three explicit phases
calibrate, test, export – then use the wrapper.

4. Board: learn XIR/VART once
with PTQ (`vart_walkthrough_fpga.py`), then run both variants with one command.

Part F – your own experiment

Pick one factor
`CALIB_SAMPLES`, `CALIB_SEED`, `CALIB_PROFILE`, `FT_SAMPLES`, `QAT_EPOCHS`, `QAT_LR`...

State a falsifiable prediction
then one new QTAG, one change, the unchanged controls listed.

Submit
the command, `metrics.json`, `summary/results.csv` and 5–8 sentences: prediction, result, recommendation, one limitation.

Keep your `TAG` folder: Lab 3 prunes, retrains and re-quantizes with the same scripts.

THE THREAD – WHAT TODAY ADDED

How much useful intelligence can we obtain from a limited budget of computation, memory and energy?

Precision is a budget: **8 bits** are enough for almost every weight and activation – *if* each tensor's range is chosen from representative data, and the model is made friendly to one ruler per tensor.

Quantization shrank every number but kept every computation. Next session removes computations altogether – pruning – and asks the uncomfortable question: does the hardware actually get faster?

Takeaways

$x \approx s \cdot q$	The integer range is fixed by the hardware; the scale – the real range it covers – is the design choice.
Two errors, one knob	Rounding is small and everywhere; clipping is rare and large. Calibration picks the threshold α.
The DPU contract	INT8, signed, symmetric, per-tensor, power of two. CLE makes a model friendly to one ruler per tensor.
Data before rules	Representative calibration data matter more than the rule. Debug layer by layer, from the first tensor.
PTQ first	Escalate to fast fine-tuning or QAT only when a stated tolerance is missed. QAT pays at low bit widths.

Quantization is not a conversion step at the end: it is a contract between the model's numbers and the hardware's arithmetic.

Sources and further reading

Foundations

- M. Nagel et al., *A White Paper on Neural Network Quantization*, arXiv:2106.08295, 2021.
- R. Krishnamoorthi, *Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper*, arXiv:1806.08342, 2018.
- B. Jacob et al., *Integer-Arithmetic-Only Inference*, CVPR 2018.
- M. Horowitz, *Computing's Energy Problem*, ISSCC 2014.
- MIT HAN Lab 6.5940, lectures on quantization: `https://hanlab.mit.edu/courses/2026-fall-65940`

Methods

- M. Nagel et al., *Data-Free Quantization* (CLE, bias correction), ICCV 2019; *AdaRound*, ICML 2020.
- I. Hubara et al., *Accurate Post Training Quantization With Small Calibration Sets* (AdaQuant), ICML 2021.
- Y. Bengio et al., STE, arXiv:1308.3432, 2013; S. Esser et al., *LSQ*, ICLR 2020.
- S. Migacz, *8-bit Inference with TensorRT*, GTC 2017.

Beyond INT8

- K. Wang et al., *HAQ*, CVPR 2019; Y. Umuroglu et al., *FINN*, FPGA 2017.
- E. Frantar et al., *GPTQ*, ICLR 2023; J. Lin et al., *AWQ*, MLSys 2024.
- OCP, *Microscaling (MX) Formats Specification v1.0*, 2023.

Vitis AI and the KV260

- AMD UG1414 (Vitis AI User Guide): `torch_quantizer`, quantization strategy, fast fine-tuning, QAT.
- Vitis AI MPSoC quick start and `resnet18_quant.py` example (calibration, test, batch-1 export).
- AMD WP486, *Deep Learning with INT8 Optimization on Xilinx Devices*, 2017.

Case study

- A. Mazouz et al., arXiv:2503.04832, 2025.

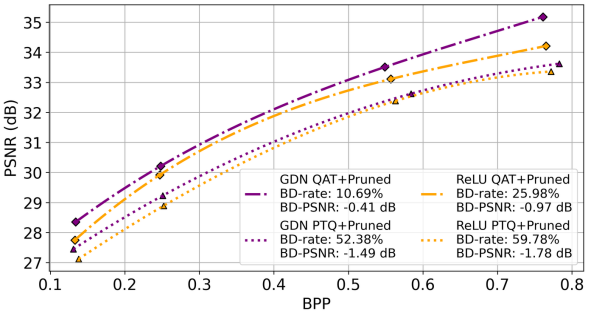
Figures marked "computed for this course" come from the course's figure scripts, which are not part of the student release.

FOR READING

Supplementary material

A case study on the same DPU, and what an FPGA allows beyond the DPU's INT8.

Case: PTQ versus QAT on a learned image codec, on the same DPU



A learned image codec (LIC) predicts *pixels*, not a class: every quantization error shows up as lost PSNR or extra bits.

PTQ: +52% bitrate
for the same quality (BD-rate, GDN model, pruned, INT8 on the DPU).

QAT: +11%
the same model, the same INT8 DPU – the network learned to tolerate its own rounding.

BD-rate: extra bitrate needed to reach the same PSNR as the paper's reference codec (lower is better).

A. Mazouz et al., *Lightweight FPGA Deployment of Learned Image Compression with Knowledge Distillation and Hybrid Quantization*, arXiv:2503.04832, 2025 (Kria KV260, DPUCZDX8G).

Classification forgives INT8 PTQ; regression does not. Know which kind of output your application needs before choosing the method.

Beyond INT8: what an FPGA lets you choose

Mixed precision

HAQ, CVPR 2019

a different bit width per layer, chosen with feedback from the hardware. First and last layers usually keep 8 bits.

Binary and ternary

FINN, FPGA 2017

multiplications become XNOR and popcount in LUTs: 12.3 M MNIST images/s at 95.8% – at a real accuracy cost on harder tasks.

Weight-only INT4

GPTQ, AWQ

when decoding is memory-bound (LLMs), the weights dominate traffic: 4-bit weights, 16-bit activations.

New float formats

FP8, microscaling (MX)

FP8 (E4M3/E5M2) and block formats with one shared exponent per 32 values: a float's range at an integer's cost.

Any width you like

HLS, S6–S8

`ap_int<W>` and `ap_fixed<W,I>` let you pick the width of every signal. DSPs, LUTs and BRAM follow.

The DPU

what Lab 2 uses

one contract, fixed in silicon: INT8, symmetric, per-tensor, power of two. Fast to use, impossible to change.

Wang et al., CVPR 2019; Umuroglu et al., FPGA 2017; Frantar et al., ICLR 2023; Lin et al., MLSys 2024; OCP *Microscaling Formats* specification, 2023.

A fixed accelerator gives you one numerical contract. A custom FPGA datapath lets you write your own – at the price of building it.