Next Generation Triggers ML Optimization: Quantization Techniques for Scalable Inference

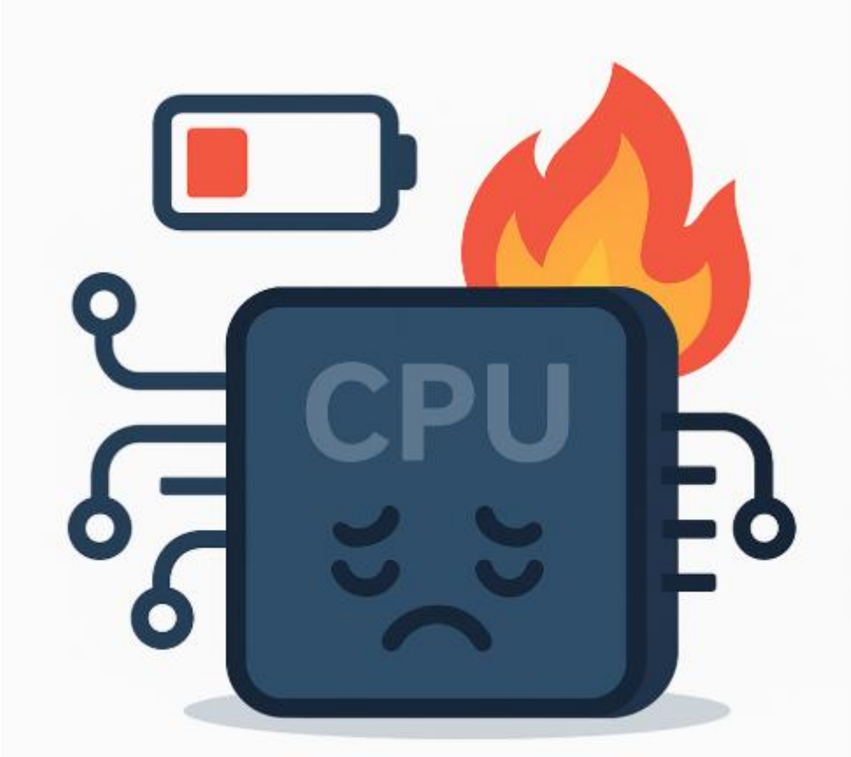
Najla Sadek, American University of Beirut
Supervisor: Amine Lahouel

Why Should We Even Quantize?

The Challenge

- Modern AI models are huge and slow.
- DeepSeek V2 has 236 billion parameters
- Requires 472 GB in FP32 → Not feasible for most devices

Why Should We Even Quantize?



Limitations

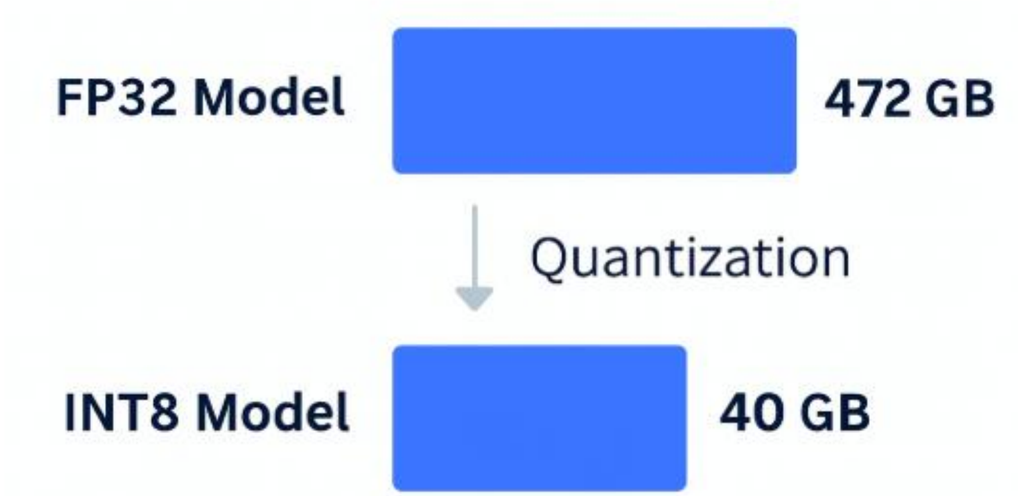
- Huge memory footprint
- Floating-point math = slow & energy-hungry
- Even computers prefer simple integer arithmetic

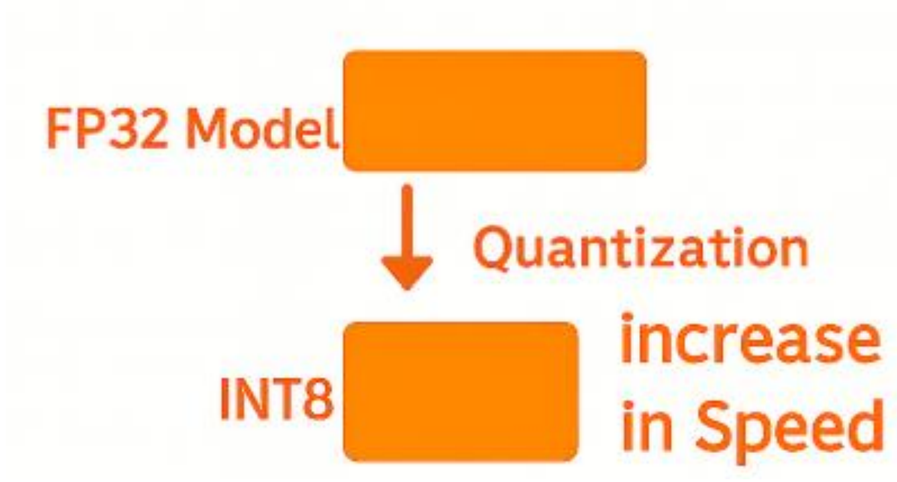
Why Should We Even Quantize?

Example

- Integer: 3 × 6 → quick
- Float: 1.21 × 2.897 → complex

Why Should We Even Quantize?

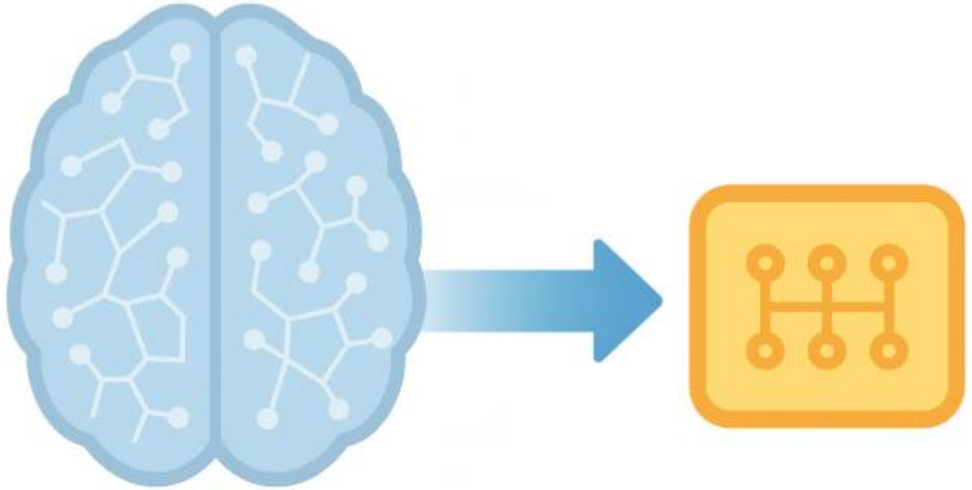




Solution:

- **Quantization** addresses both the storage and speed barriers.
- DeepSeek applied advanced quantization techniques to compress its massive model: 8× to 12x compression: From 472 GB (FB32) to 40-60 GB (INT8).

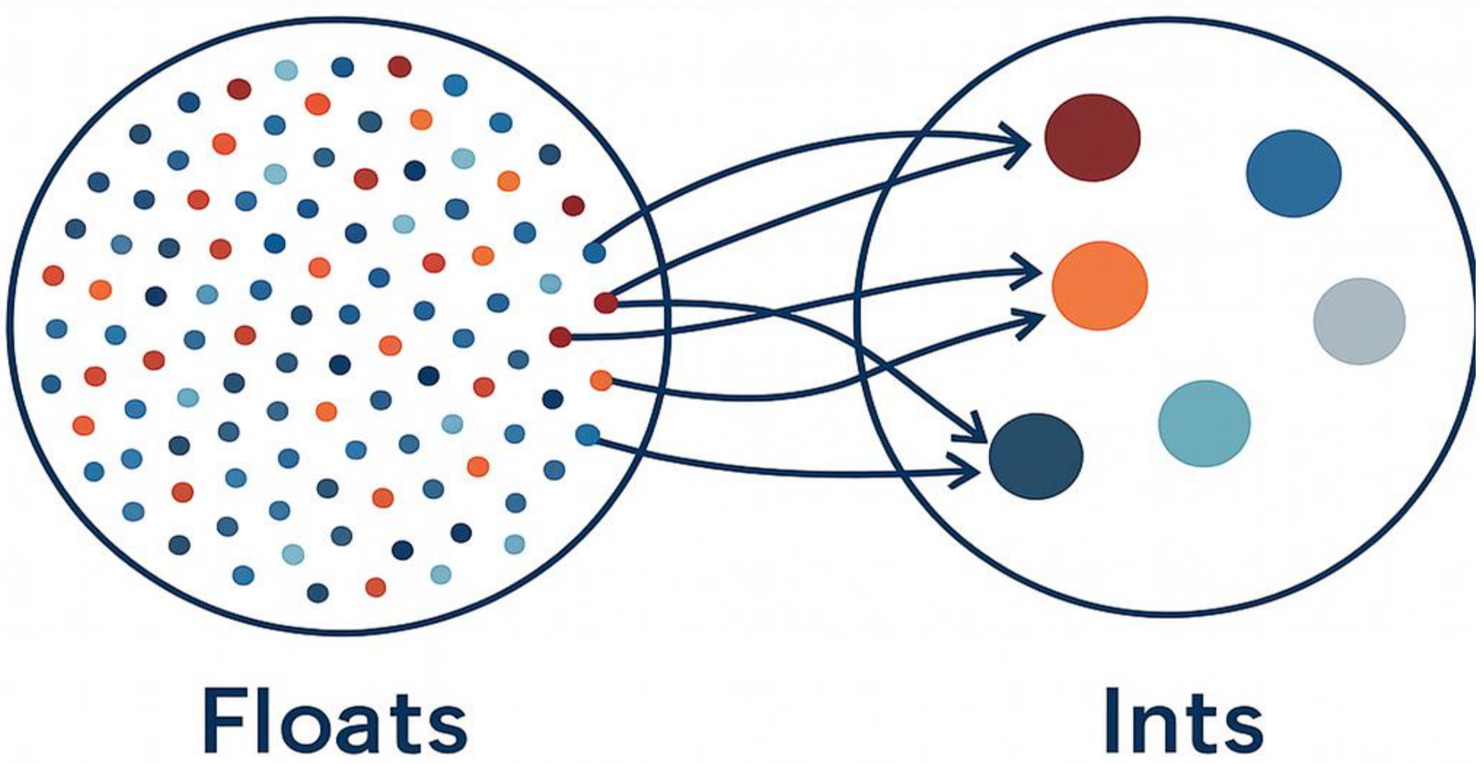
Why Should We Even Quantize?



Conclusion:

Quantization makes large models usable on smaller, faster hardware.

What is Quantization?



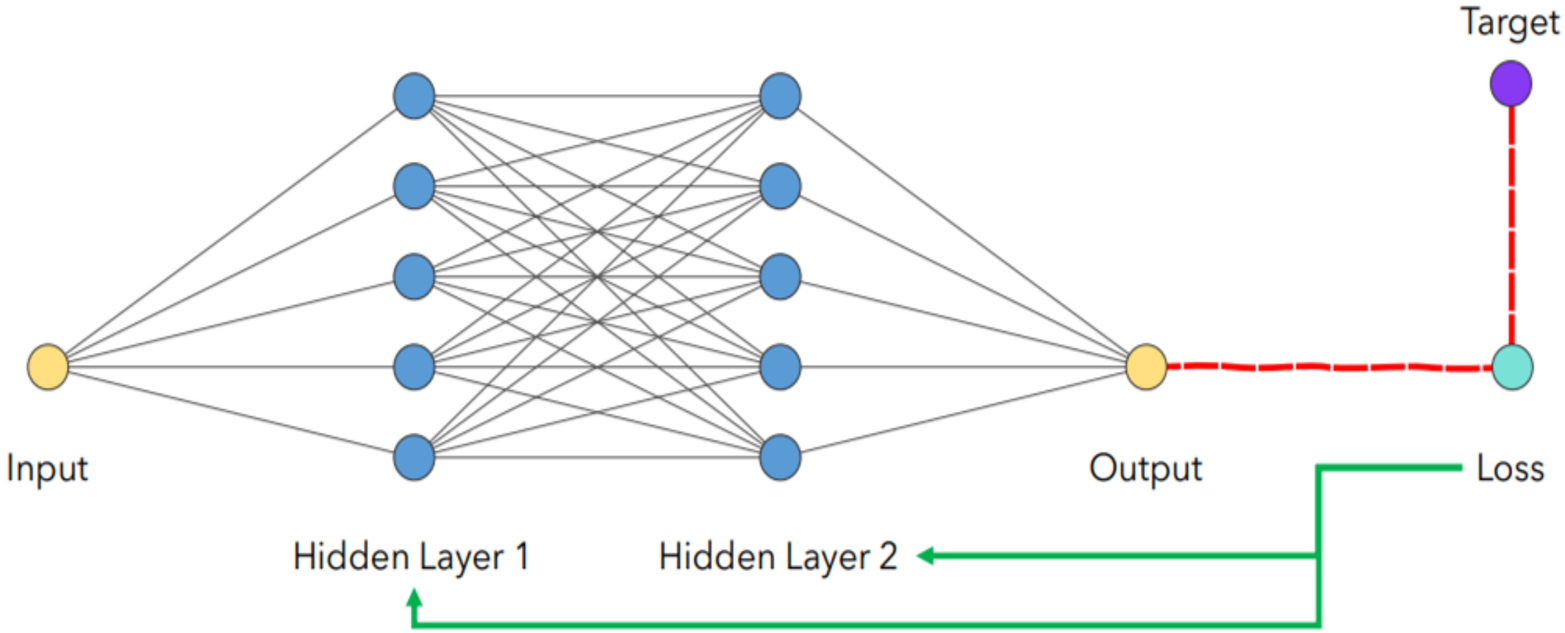
Quantization is mapping input values from a large set to output values in a smaller set.

Quantization is like when your teacher switches from grading out of 100 to grading out of 10 and she just chops off the decimals!

So your shining 97 turns into a humble 9, and your 91 also becomes an 9.

Of course, some teachers might round up—but you get the point. It's all about squeezing numbers into simpler buckets!

What is Quantization?



- Neural networks can have various types of layers.
- Linear layers contain two matrices: **weights** and **biases**.
- These matrices are usually stored as floating-point numbers.
- **Quantization** replaces floating-point values with integers.
- The goal is to reduce model size while keeping accuracy intact.

In NNets context: reducing the precision of weight values.

Types of Quantization (numerically)

Asymetric 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

Types of Quantization (numerically)

Symetric 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]

α ← Highest absolute value

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

$$x_q = clamp\left(\left\lfloor \frac{x_f}{s} \right\rceil; -(2^{n-1}-1); 2^{n-1}-1\right) \qquad s = \frac{abs(\alpha)}{2^{n-1}-1}$$

Quantize

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

$$x_f = sx_q$$

Dequantize

43.16	-44.93	0	22.99	-43.86	-11.32	38.56	-20.52	-38.56	-27.95

Types of Quantization (implementation)

PTQ – Post-Training Quantization

Quantizes fixed parameters (weights, biases) before inference.

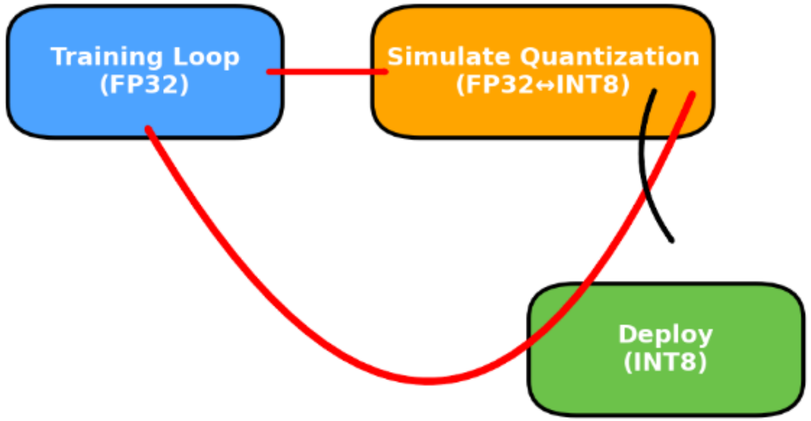
Supports symmetric and asymmetric quantization (usually linear scaling).

QAT – Quantization-Aware Training

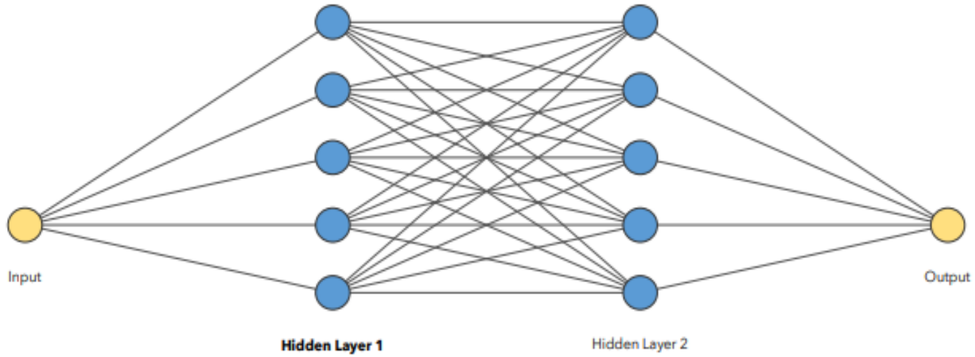
Simulates quantization effects during training by adding quantization/dequantization layers.

Helps the model reach wider local minima—making it robust to quantization-induced changes.

Typically delivers better accuracy than post-training quantization alone.



Applying Quantization



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

Input
Sometimes called "activation"

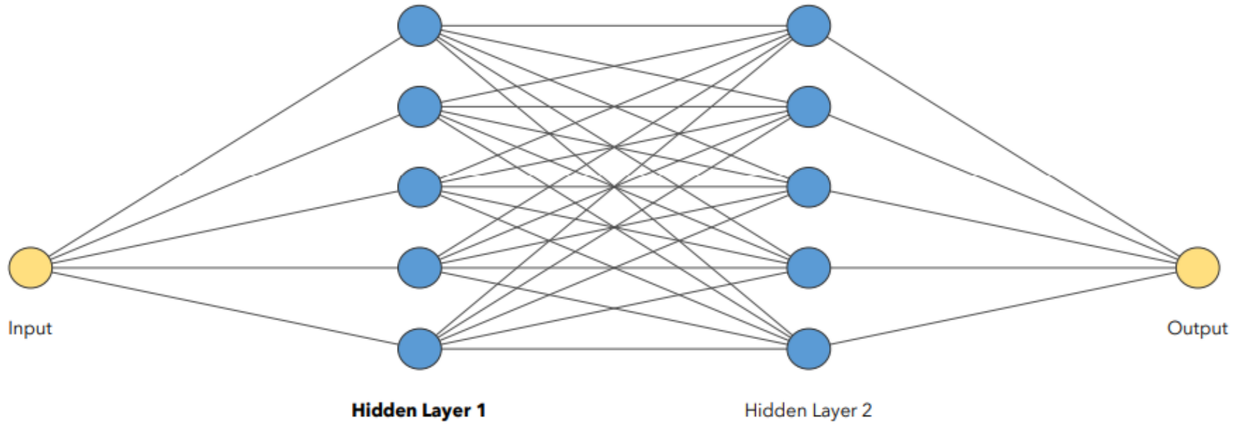
Weight

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]

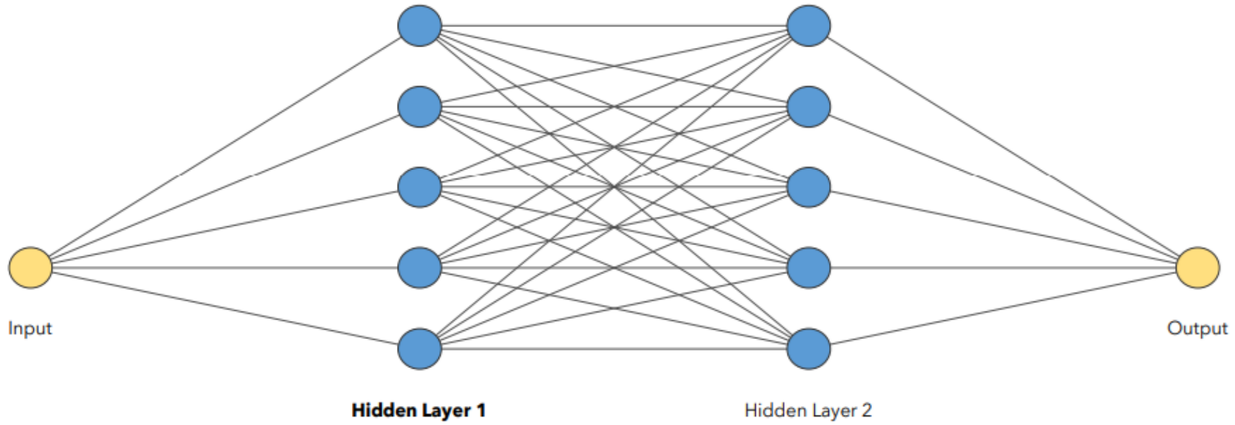
We usually *sacrifice* -128 to obtain a symmetric range

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

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]

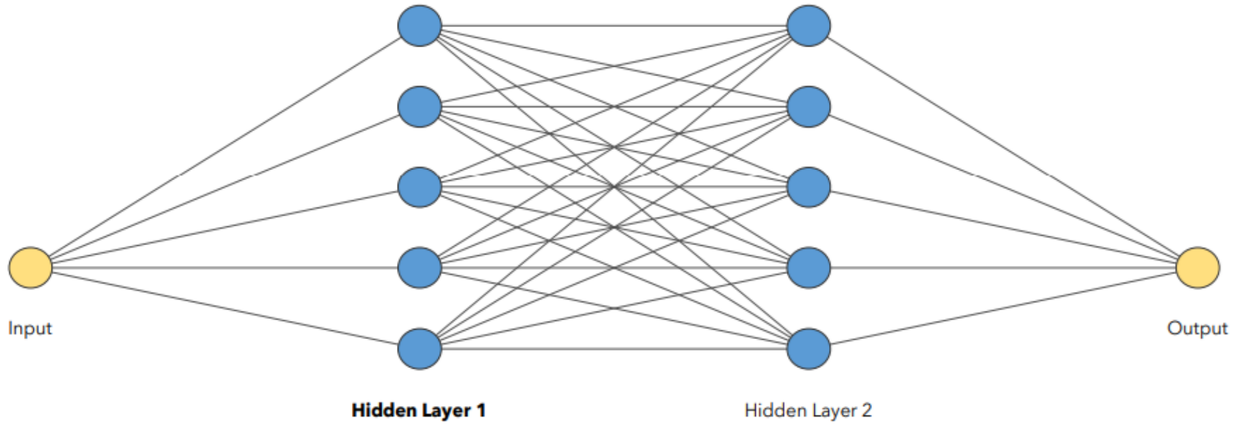
We usually *sacrifice* -128 to obtain a symmetric range

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

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

Applying Quantization

You might be asking yourself:

Aren't we losing something here?

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

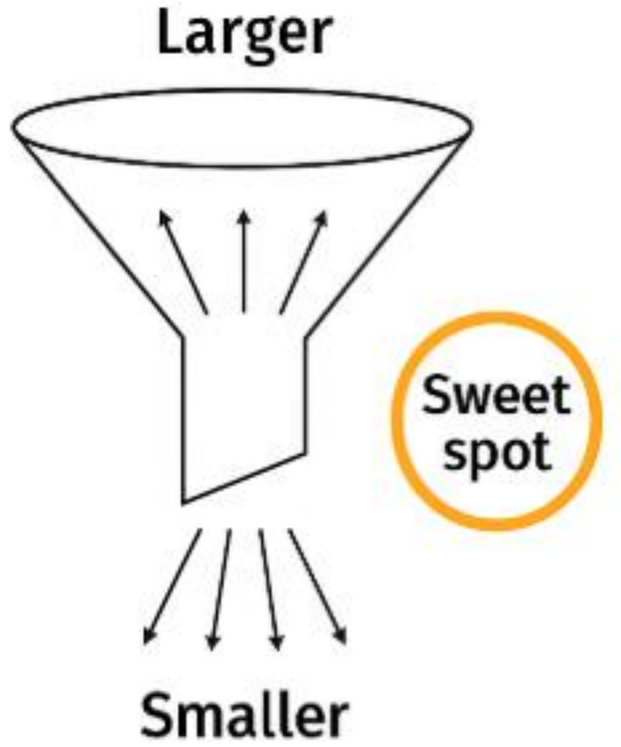
Dequantize

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

Quantization: Pros and Cons

Absolutely. That’s the **compression–performance tradeoff**.

According to the **pigeonhole principle**, many real values will be mapped to the same quantized value.
The challenge is to find a **sweet spot** where the model is **small enough** to be efficient,
but **accurate enough** to remain useful.



Frameworks

PyTorch
Used for model quantization (both PTQ & QAT), calibration, and inference testing.
Also helpful for exporting quantized models

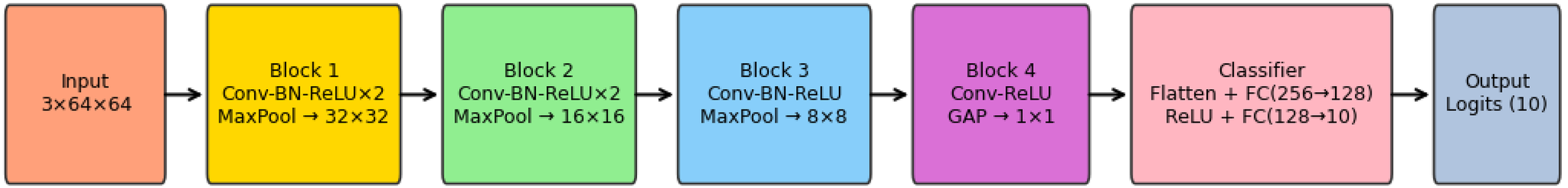
TensorFlow Quantization – TensorFlow supports post-training quantization and quantization-aware training through the tf.lite converter, enabling deployment of smaller, faster models on edge devices without significant accuracy loss.

ONNX Quantization – The ONNX Runtime provides tooling for dynamic, static, and QAT quantization, allowing cross-framework models to be optimized for int8 inference across diverse hardware backends.

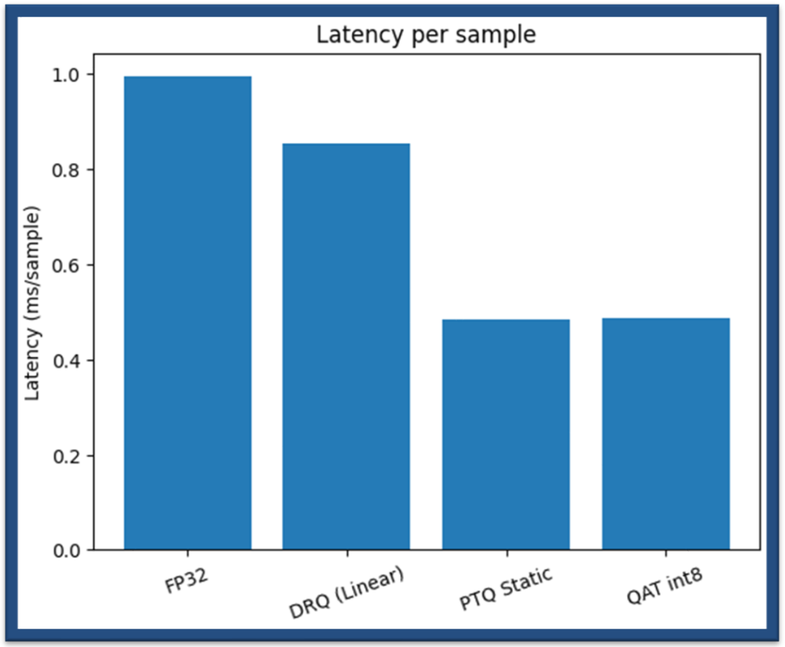
```
=== Dequantized Weight Tensor (default view) ===
tensor([[-0.0032,  0.2679, -0.4131, -0.3679],
        [-0.1937,  0.1356, -0.0097,  0.3970]], size=(2, 4), dtype=torch.qint8,
       quantization_scheme=torch.per_tensor_affine, scale=0.003227628068998456,
       zero_point=0)
Type: <class 'torch.Tensor'>
Dtype: torch.qint8
Quantization scheme: torch.per_tensor_affine
Scale: 0.003227628068998456
Zero-point: 0

=== Raw int8 Weight Values (via .int_repr()) ===
tensor([[  -1,   83, -128, -114],
        [ -60,   42,   -3,  123]], dtype=torch.int8)
Dtype: torch.int8

=== Quantized Bias Tensor (actually still float32) ===
Parameter containing:
tensor([-0.0444,  0.1323], requires_grad=True)
Type: <class 'torch.nn.parameter.Parameter'>
Dtype: torch.float32
```

Results

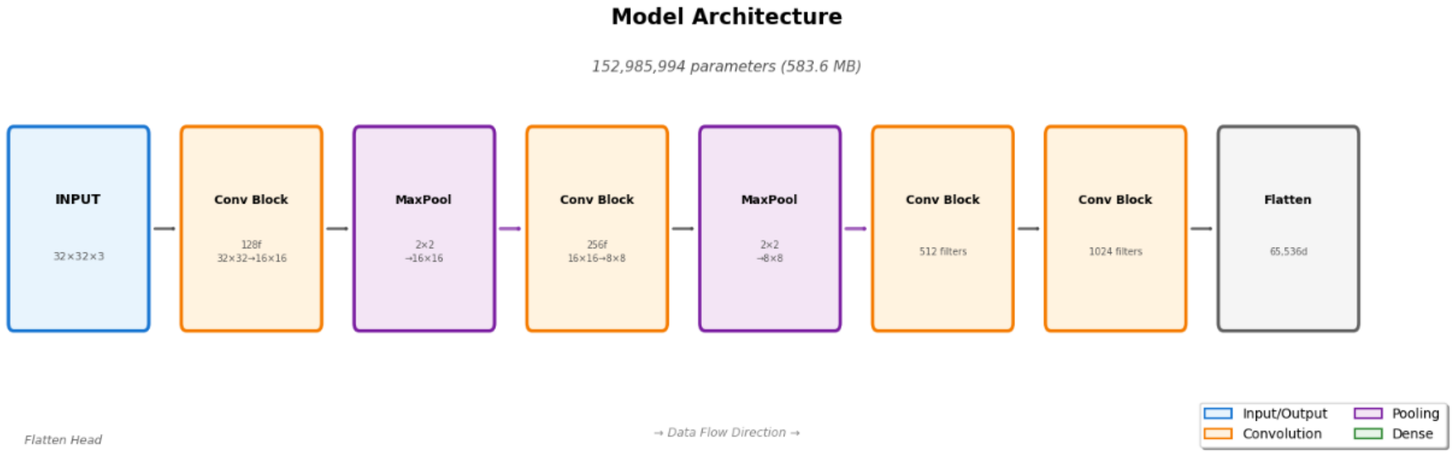
LargeCNN Architecture (Simplified Block Diagram)

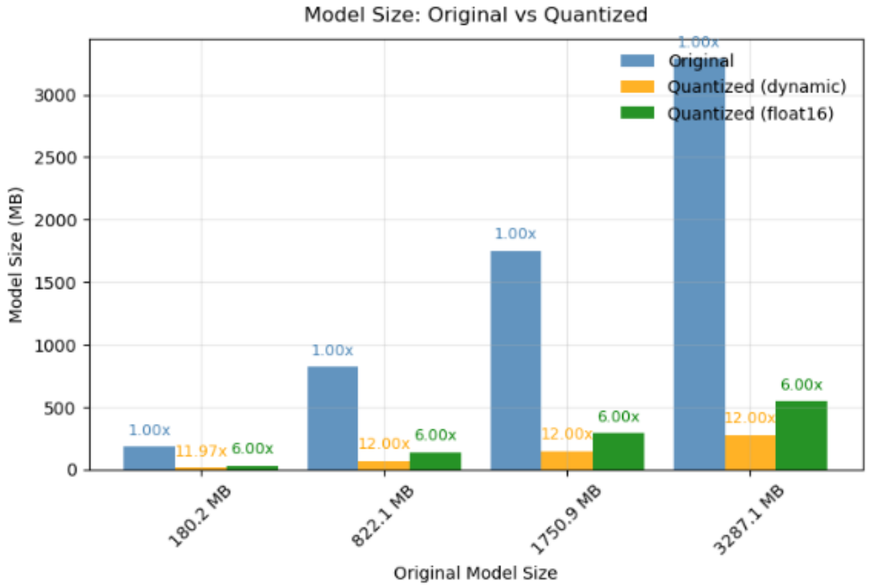


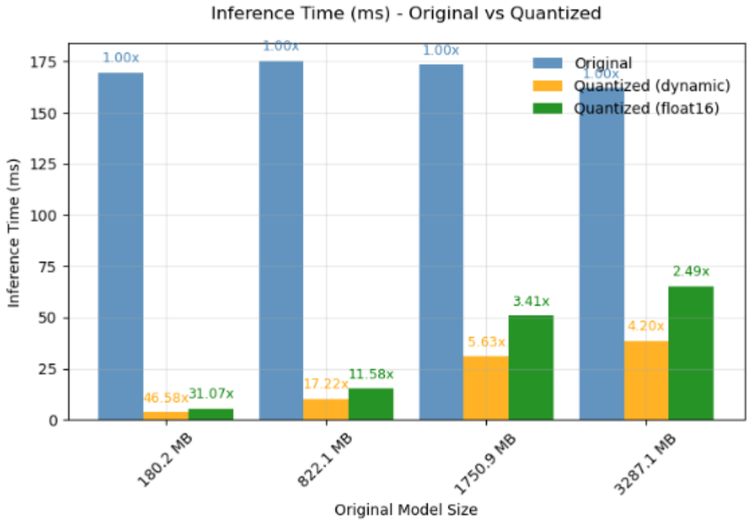
Method	Bits	Size(MB)	ms/batch	ms/sample	Size Red.(%)
FP32	32	1.09	248.61	0.994	0.0
DRQ (Linear)	8(int8)	0.34	213.86	0.855	68.5
PTQ Static	8(int8)	0.29	120.96	0.484	73.9
QAT int8	8(int8)	0.29	121.62	0.486	73.9

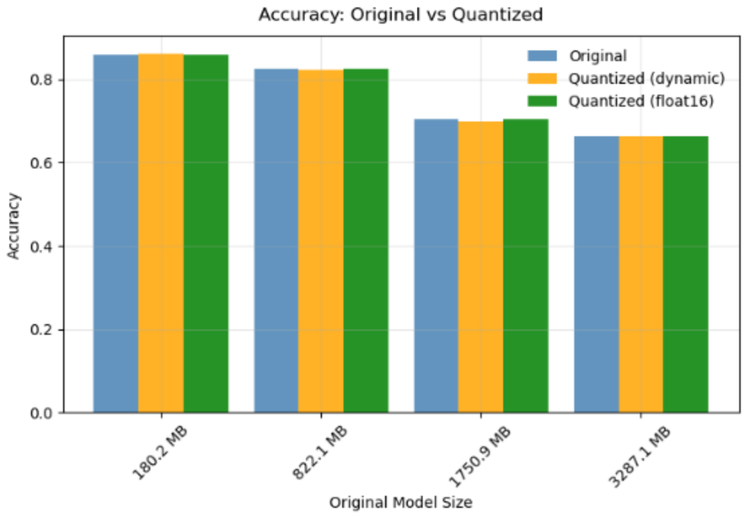


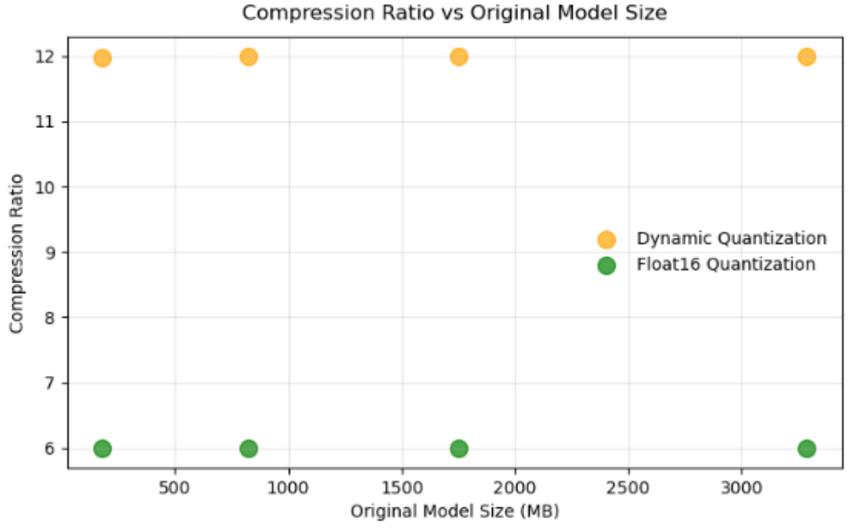
Results


Model Architecture
152,985,994 parameters (583.6 MB)
INPUT
32×32×3
Conv Block
128f
32×32→16×16
MaxPool
2×2
→16×16
Conv Block
256f
16×16→8×8
MaxPool
2×2
→8×8
Conv Block
512 filters
Conv Block
1024 filters
Flatten
65,536d
Flatten Head
→ Data Flow Direction →
Input/Output
Convolution
Pooling
Dense


Model Size: Original vs Quantized
Original
Quantized (dynamic)
Quantized (float16)
Model Size (MB)
1.00x
11.97x
6.00x
12.00x
Original Model Size
180.2 MB
822.1 MB
1750.9 MB
3287.1 MB


Inference Time (ms) - Original vs Quantized
Original
Quantized (dynamic)
Quantized (float16)
Inference Time (ms)
1.00x
46.58x
31.07x
17.22x
11.58x
5.63x
3.41x
4.20x
2.49x
Original Model Size
180.2 MB
822.1 MB
1750.9 MB
3287.1 MB


Accuracy: Original vs Quantized
Original
Quantized (dynamic)
Quantized (float16)
Accuracy
Original Model Size
180.2 MB
822.1 MB
1750.9 MB
3287.1 MB


Compression Ratio vs Original Model Size
Dynamic Quantization
Float16 Quantization
Compression Ratio
Original Model Size (MB)

Takeaways, Open Questions, and Future Works

- How do we balance **model accuracy** with **FPGA resource constraints** (LUTs, DSPs, BRAM)?
- What are the limits of compression before physics-relevant signals are lost?

- Integration within ml.cern.ch
- Explore mixed-precision quantization across layers
- Further Test different types of quantization with different kinds of models

Thank you for your attention!

Email | najla.samer.sadek@cern.ch

Websites | openlab.cern
| nextgentriggers.web.cern.ch

NexTGen
Next Generation Triggers

Quantization: Pros and Cons

Pros	Cons
• Reduces model size (e.g., 8x–12x smaller)	• Slight accuracy drop
• Speeds up inference	• Requires calibration/fine-tuning
• Enables deployment on edge devices	• May not work well on all architectures
• Saves energy and memory bandwidth	• Some quantization methods are complex

Appendix 1 – WOQ, DRQ & MPQ

WOQ – Weight-Only Quantization

•Only weights are quantized; activations remain in floating-point format.

•Compared to traditional INT8 quantization on both activations and weights, WOQ often offers a better trade-off between performance and accuracy.

DRQ – Dynamic Range Quantization

•Optimizes model size and speed by converting weights to lower-bit integers (e.g., FP32 → INT8).

•Activations remain in floating-point format during model conversion.

•During inference, activations are dynamically quantized as needed.

Note: Activations are quantized dynamically at inference, unlike WOQ where they remain floats.

Mixed Precision Quantization

•Uses different precisions for different parts of the model (e.g., INT8, INT4, FP16, FP32).

•Balances accuracy, speed, and memory across layers.

FIQ – Full Integer Quantization

•Both weights and activations are quantized to integers (typically INT8).

•Enables fully integer-based inference, improving speed and efficiency—ideal for edge devices (phones, microcontrollers, TPUs).

Appendix 2 – QAT & PTQ

QAT – Quantization-Aware Training

•Simulates quantization effects during training by adding quantization/dequantization layers.
•Helps the model reach wider local minima—making it robust to quantization-induced changes.
•Typically delivers better accuracy than post-training quantization alone.

PTQ – Post-Training Quantization

•Quantizes fixed parameters (weights, biases) before inference.
•Supports symmetric and asymmetric quantization (usually linear scaling).
•During inference:

- Inputs are quantized on the fly.
- Output dequantization uses **calibration** by collecting activation statistics during sample runs.
- Observers track statistics (e.g., scale s and zero-point z) for each layer to guide dequantization.

Note: In Post-Training Quantization (PTQ), we can use **percentile-based calibration** instead of Min-Max to reduce sensitivity to outliers.
This technique sacrifices outlier accuracy slightly but improves quantization for most activations.

Appendix 3

Per-Channel vs. Per-Tensor Quantization

Per-Tensor Quantization:
A **single scale and zero-point** is used for the **entire tensor** (e.g., the whole weight matrix or activation tensor).
Simple and efficient to implement.
Often less accurate, especially for convolutional layers with diverse ranges across channels.

Per-Channel Quantization:
Each **channel** (e.g., each output filter in a convolution layer) has its **own scale and zero-point**.
Provides better accuracy, especially for models with large variation between channels.
Slightly more complex to implement and requires more metadata, but widely supported by modern hardware.
Note: Per-channel quantization usually outperforms per-tensor quantization in terms of accuracy, particularly for convolutional layers.