Neural Network Quantization with Brevitas

Giuseppe Franco, Nicholas J. Fraser (RAD)

2026-01-12

AMD
together we advance_

Agenda

Quantization Landscape

Brevitas introduction

- The building blocks
- Creating a quantized model
- PTQ & QAT Algorithms

Research works

- Accumulator-Aware Quantization with Guaranteed Overflow Avoidance
- Accumulator-Aware Post-Training Quantization for Large Language Models
- Improving Quantization with Post-Training Model Expansion
- Qronos: Correcting the Past by Shaping the Future

Quantization Landscape

With the explosion of LLM, quantization has becoming fundamental to ensure functional and efficient deployment of neural networks.

Solutions like **llama.cpp**, **LLM-Compressor**, **Torch AO,** focus on pre-defined recipes for efficient training/inference.

Other solutions like **AMD's Quark** try to provide a balance between ready-to-use configurations and customization.

With **Brevitas**, our goal is to fully focus on flexibility, enabling full control over every single element of the process.

Stable Workflows

🟢 = stable, 🟡 = prototype, 🟠 = planned, ⚪ = not supported

recommended hardware	weight	activation	quantized training	QAT	PTQ data algorithms	quantized inference
H100, B200 GPUs	float8 rowwise	float8 rowwise	🟢 (link)	🟢 (link)	⚪	🟢 (link)
H100 GPUs	int4	float8 rowwise	⚪	🟢 (link)	🟠	🟢 (link)
A100 GPUs	int4	bfloat16	⚪	🟢 (link)	🟡: HQQ, AWQ, GPTQ	🟢 (link)
A100 GPUs	int8	bfloat16	⚪	🟢 (link)	⚪	🟢 (link)
A100 GPUs	int8	int8	🟡 (link)	🟢 (link)	⚪	🟢 (link)
edge	intx (1..7)	bfloat16	⚪	🟢 (link)	⚪	🟢 (link)
edge	intx (1..7)	bfloat16	⚪	🟢 (link)	⚪	🟢 (link)

TorchAO Quantization Table

AMD
together we advance_

Brevitas Rationale

Brevitas: Neural Network Quantization in PyTorch

- Brevitas is a 9-year-old research project with 1.5k github stars and more than 500k downloads
- First-class support for integer, minifloats, groupwise (e.g., MX) datatypes
- Supports PTQ & QAT
  - Emphasis on **composing multiple** PTQ, QAT algorithms
- A library of quantized layers (nn.Modules)
  - Match PyTorch nn.Modules interface, with extra quantization parameters
- Flexible quantizers (nn.Modules) many algorithmic options
  - Modules are designed with **composability** in mind
  - **"Preset" quantizers** for popular choices, with sensible defaults
  - **Flexible / extendable** (hackable) to support new quantizers & algorithms
- Out of the box support for distributed training – (e.g., DDP, interoperability with HuggingFace Accelerate (PP))

Which datatype(s) should we put into hardware?

How can we **maximise accuracy** at low bitwidths?

Table 3. We evaluate A2Q+ for W4A4 ImageNet models and compare against baseline QAT methods and standard A2Q, both with and without Euclidean projection-based initialization (EP-init.)

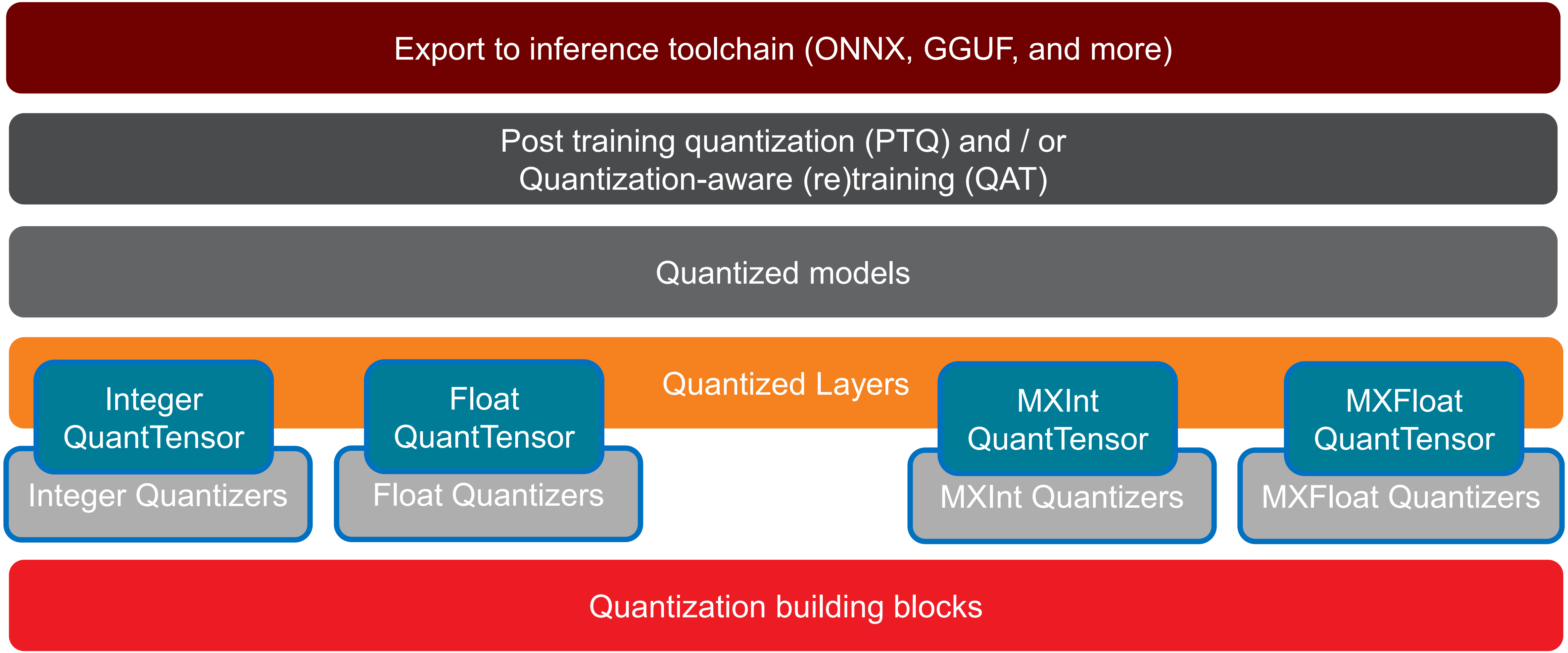
Network	Method	P	Top-1	Sparsity
ResNet18 (Float: 69.76%)	Base	32	70.2%	20.8%
	A2Q	16	69.2%	73.7%
		14	60.3%	91.7%
		12	35.5%	94.7%
	A2Q (w/ EP-init)	16	69.3%	73.7%
		14	62.5%	91.2%
		12	42.7%	94.6%
	A2Q+	**16**	**69.8%**	**50.4%**
		14	**67.1%**	**85.0%**
		12	**56.4%**	**93.4%**

[1] Su, Jiang, et al. "Accuracy to throughput trade-offs for reduced precision neural networks on reconfigurable logic." ARC 2018.

[2] Colbert, Ian, et al. "A2Q+: Improving Accumulator-Aware Weight Quantization." arXiv preprint arXiv:2401.10432 (2024).

together we advance_

Brevitas: A Modular Approach to Quantization



AMD
together we advance_

Quantization building blocks

Change the rounding with casting to support minifloat, etc.

Uniform Affine Quantization (Visualised)

According to NN literature:

Quantize + Dequantize (QDQ)

Quantization

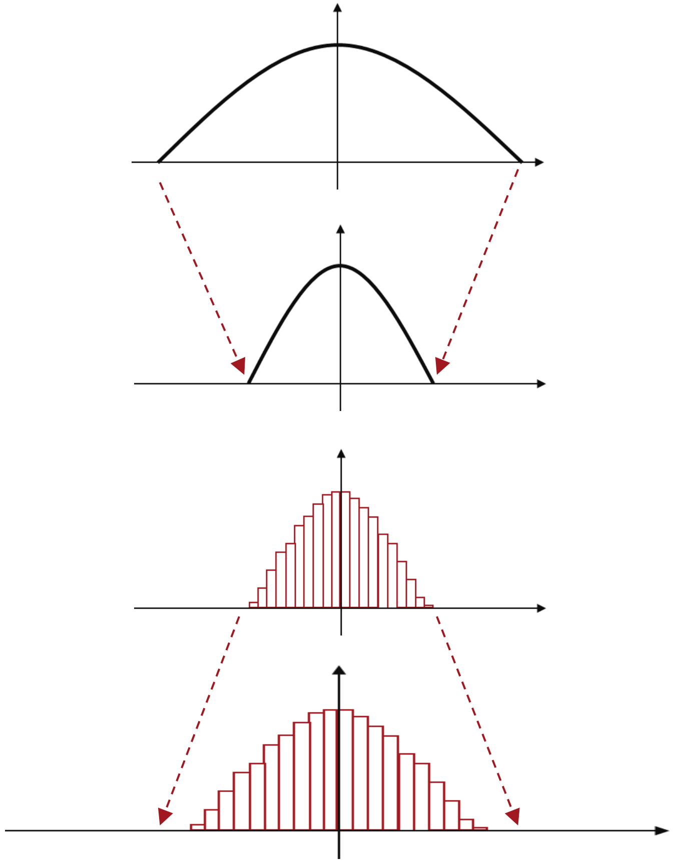
Original Distribution x

Scaling to suit range x/s

Rounding
(RNE, Stochastic) $\left\lceil \frac{x}{s} \right\rfloor$

Dequantization

Rescaling to match original distribution $s\left\lceil \frac{x}{s} \right\rfloor$



$$Q(x) = s \cdot round(clip(x/s, -2^{b-1}, 2^{b-1} - 1))$$

Note, clipping/zero-point omitted here

Quantize + Dequantize (QDQ) = “fake quantization”

Quantization Building Blocks & Algorithms

$$Q(x) = s \cdot round\left(clip\left(\frac{x}{s}, -2^{b-1}, 2^{b-1} - 1\right)\right)$$

- Precision
  - Binary
  - Ternary
  - Integer (constant or learned)
    - Narrow range or not (e.g. [-127, 127] at 8b) or not (e.g. [-128, 127] at 8b)
  - Minifloat
    - OCP, Custom formats
  - MX (i.e., groupwise quantization)
  - Options:
    - Signed or unsigned
    - Various types of rounding supported

- Scale, Zero-point
  - Restrictions:
    - Floating-point, fixed point, power-of-two, quantized, hierarchical
  - Granularities:
    - Per tensor/per channel/per group
    - Any group size, group dim
  - Algorithm:
    - Constant
    - Learned as a parameter (in log domain or not)
    - Calculated from statistics:
      - Absmax, percentile, MSE
    - Static/dynamic

- Maximise Accuracy (PTQ / QAT):
  - Adaptive rounding:
    - GPTQ/GPFQ/Learned Rounding
  - Equalization Methods:
    - Weight Equalization
    - SmoothQuant
    - Channel Splitting
  - Rotation:
    - Hadamard, Orthonormal, Cailey Opt
  - Bias Correction
  - Accumulation Aware Quantization

Instantiating a Quantizer (1)

- Example presets:
  - Various Int8 formats
    - Po2, float scaling
      - `Int8WeightPerTensorFixedPoint`, `Int8WeightPerTensorFloat`
    - Per-channel, per-tensor, per-group scaling
      - `Int8WeightPerChannelFloat`, `Int8WeightPerTensorFloat`, `IntWeightSymmetricGroupQuant`
    - Zero point
      - `ShiftedUint8WeightPerTensorFloat`
    - Unsigned / Signed
      - `Int8ActPerTensorFloat, Uint8ActPerTensorFloat`
  - FP8 OCP formats
    - E4M3, E5M2
    - `Fp8e4m3OCPWeightPerTensorFloat`, `Fp8e5m2OCPWeightPerTensorFloat`
  - MX Formats
    - MXFP, MXInt
    - `MXFloatWeight, MXIntWeight`

- Custom quantizer:

```
# imports ...

class MyCustomMX (MXFloatOCPWeight):
     group_size = 64
```

```
# imports ...

class MyInt8WeightPerTensorFloat (WeightQuantSolver):
    quant_type = "int"
    bit_width_impl_type = "const"
    float_to_int_impl_type = "round"
    signed = True
    zero_point_impl = ZeroZeroPoint
    scaling_impl_type = "stats"
    scaling_stats_op = "max"
    scaling_per_output_channel = False
    restrict_scaling_type = "fp"
    bit_width = 8
```

Assembling Quantizers to Make Quantized Layers

- Quantized layers: a PyTorch module which applies various quantizers in its forward function
  - Many common layers share common features – can be treated the same: Linear, ConvXd, ConvTransposeXd, etc…
- More complex nn.Modules need special treatment:
  - QuantLSTM, QuantMHA, QuantSDPA
- For MHA – 3 flows supported:
  - QuantLinear vs QuantMHA vs QuantSDPA

```python
class SimplifiedQuantLinear(nn.Linear):
    def __init__(
            self,
            weight_quant: Optional[WeightQuantType],
            bias_quant: Optional[BiasQuantType],
            input_quant: Optional[ActQuantType],
            output_quant: Optional[ActQuantType],
            **kwargs):
        nn.Linear.__init__(**kwargs)
        self.weight_quant = weight_quant
        # etc.

    def forward(self, x: Tensor):
        qx = self.input_quant(x)
        qw = self.weight_quant(self.weight)
        qb = self.bias_quant(self.bias)
        y = nn.functional.linear(qx, qw, qb)
        return self.output_quant(y)
```

Creating a quantized model

User Flow (1) - Manual

Vanilla PyTorch

```python
import torch.nn as nn


class FloatingPointNet(nn.Module):
  def __init__(self):
    super(FloatingPointNet,self).__init__()

    self.conv1=nn.Conv2d(1,20,5,1)
    self.conv2=nn.Conv2d(20,50,5,1)
    self.fc1=nn.Linear(4*4*50,500)
    self.fc2=nn.Linear(500,10)
```

PyTorch + Brevitas (Manual)

```python
import torch.nn as nn
import brevitas.nn as qnn
from brevitas.quant import Int8WeightPerTensorFloat

class QuantNet(nn.Module):
  def __init__(self):
    super(QuantNet, self).__init__()
    wq = Int8WeightPerTensorFloat
    self.conv1=qnn.QuantConv2d(1,20,5,1,weight_quant=wq)
    self.conv2=qnn.QuantConv2d(20,50,5,1,weight_quant=wq)
    self.fc1=qnn.QuantLinear(4*4*50,500,weight_quant=wq)
    self.fc2=qnn.QuantLinear(500,10,weight_quant=wq)
```

Many Brevitas users leverage the **manual flow**, when accuracy is paramount

User Flow (2) – Programmatic (Layerwise)

Vanilla PyTorch

```
import torch.nn as nn


class FloatingPointNet(nn.Module):
  def __init__(self):
    super(FloatingPointNet,self).__init__()

    self.conv1=nn.Conv2d(1,20,5,1)
    self.conv2=nn.Conv2d(20,50,5,1)
    self.fc1=nn.Linear(4*4*50,500)
    self.fc2=nn.Linear(500,10)
```

PyTorch + Brevitas (Layerwise)

```
import torch.nn as nn
import brevitas.nn as qnn
from brevitas.quant import Int8WeightPerTensorFloat
from brevitas.graph.quantize import layerwise_quantize

map = {
    nn.Conv2d: (qnn.QuantConv2d, {'weight_quant':
          Int8WeightPerTensorFloat,}),
    nn.Linear: (qnn.QuantLinear, {'weight_quant':
          Int8WeightPerTensorFloat,}),
}

net = FloatingPointNet()

qnet = layerwise_quantize(net, compute_layer_map=map)
```

`layerwise_quantize` fetches a nn.Module's attributes and uses them instantiate a new (quantized) module, using the map

Works at the module-level – torch.nn.functional's need to be "raised" to their Modular implementations

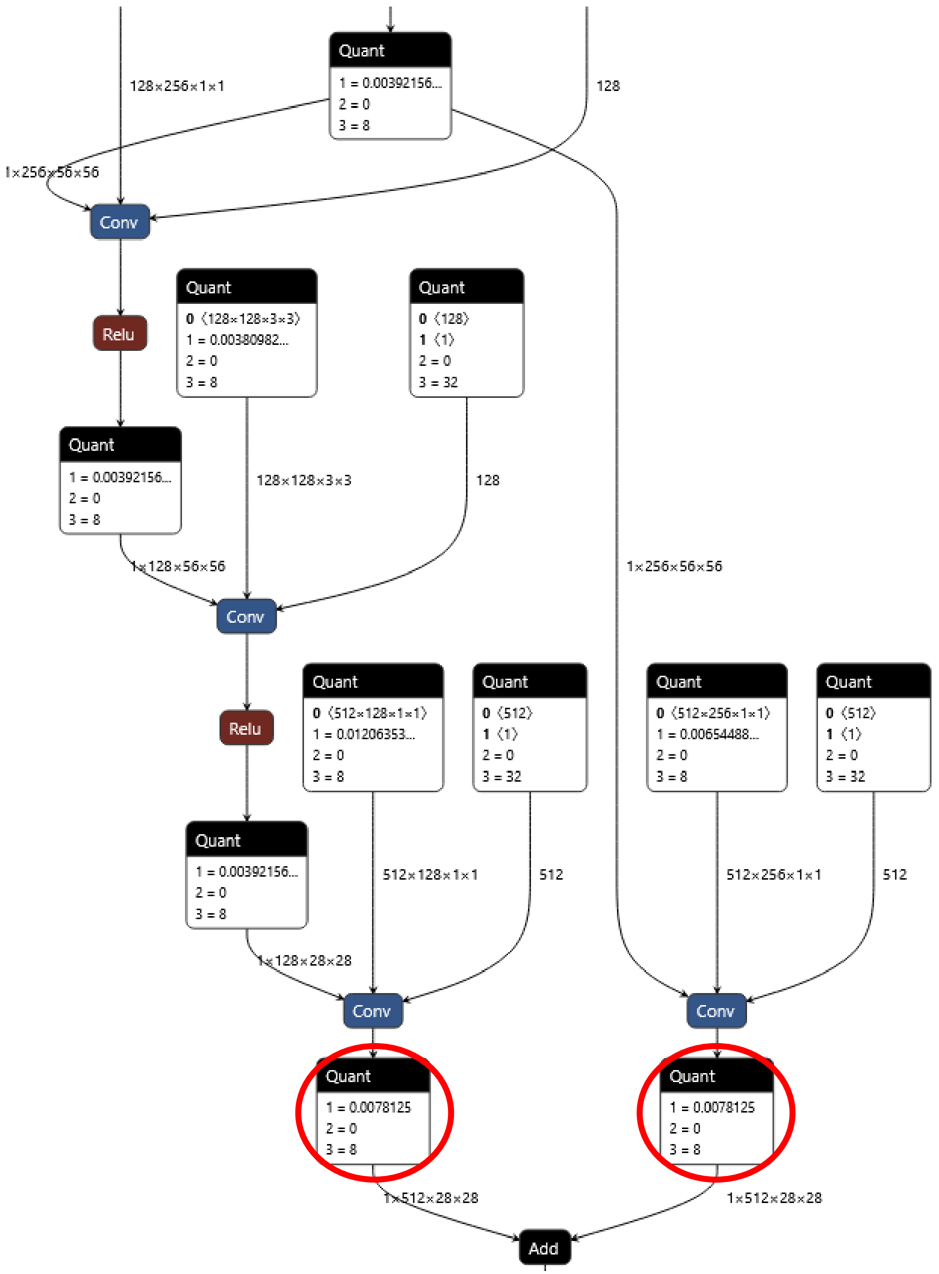
User Flow (3) – Programmatic (FX)

PyTorch + Brevitas (FX)

```python
import torch

from torchvision.models import resnet50

from brevitas.graph.quantize import preprocess_for_quantize
from brevitas.graph.quantize import quantize

net = resnet50(pretrained=True)
net = preprocess_for_quantize(net)
qnet = quantize(net)
```

FX-based graph quantization – useful for when seeing the ***entire computation graph*** is necessary:

- Alignments of scales / signedness before add/concat
    - Full integer quantization
- Avoid unnecessary (re-)quantization
- important operations like BN merging, data-free equalization



PTQ & QAT Algorithms

Achieving Good Accuracy

- Brevitas separates insertion of quantization nodes, calculation of quantization parameters and export
- Quantizers for PTQ and QAT are the same
  - Calibration performs forward passes to collect statistics (with quantization temporarily disabled).
  - QAT performs forward and backward pass

Quantization aware training:

- Brevitas-quantized nn.Modules are just PyTorch modules!
- QAT is the same as for any nn.Module
- Quantization parameters can be used for custom losses
- Compatibility with distribute training

```
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

Achieving Good Accuracy: PTQ

- Due to the massive size of LLMs, a lot of algorithms have been developed to efficiently quantize them without the need of re-training or large datasets
- In Brevitas, we natively support many of the most recent SOTA algorithms

PTQ Algorithms:

- Channel splitting
- Weight Equalization, based on FX
- Activation Equalization (i.e., SmoothQuant)
  - Layer-wise with no FX, Cross-layer with FX
- Learned Round (i.e., AdaRound, AutoRound)
  - Experimental support for BRECQ
- Error Correction algorithms
  - Qronos, GPTQ and its variants for low precision accumulation
- Bias Correction
- Rotation equalization (i.e., QuaRot, SpinQuant)
  - Layer-wise with no FX, Cross-layer with FX

How to apply the algorithms:

- Custom script

```
apply_<ptq_algorithm>(model[, calibration_dataset])
```

- Entry-points for specific model types
  - Transformers' Models: brevitas_llm_ptq
  - Torchvision: brevitas_ptq_imagenet_val
  - Diffusion: brevitas_diffusion_ptq (soon)

Example: how to use LLM entrypoint

```
brevitas_llm_ptq --config config.yaml
```

```yaml
model: HuggingFaceTB/SmolLM2-135M
dataset: wikitext2
dtype: float32
act_calibration: false
act_equalization: null
bias_corr: false
rotation: fused_no_fx
gptq: false
gpxq_block_name: model.layers
weight_equalization: false
input_bit_width: 8
input_group_size: 32
input_param_method: stats
input_quant_format: float_ocp_e4m3
input_quant_granularity: per_group
input_quant_type: sym
input_scale_precision: po2_scale
input_scale_type: dynamic
weight_bit_width: 8
weight_group_size: 32
weight_param_method: stats
weight_quant_format: float_ocp_e4m3
weight_quant_granularity: per_group
weight_quant_type: sym
weight_scale_precision: po2_scale
```

Model / Eval configuration:
- Support for several HF models
- Support for perplexity
- Support zero-shot evaluation through lm-eval or lighteval

PTQ Configuration:
- Support for all the latest SOTA algorithms
- Possibility to combine algorithms

Input Quantization Configuration:
- This is equivalent to MXFP8
- Possibility to fine-tune the configuration

Weight Quantization Configuration:
- This is equivalent to MXFP8
- Possibility to fine-tune the configuration
- Not tied to the input config

AMD together we advance_

Research works

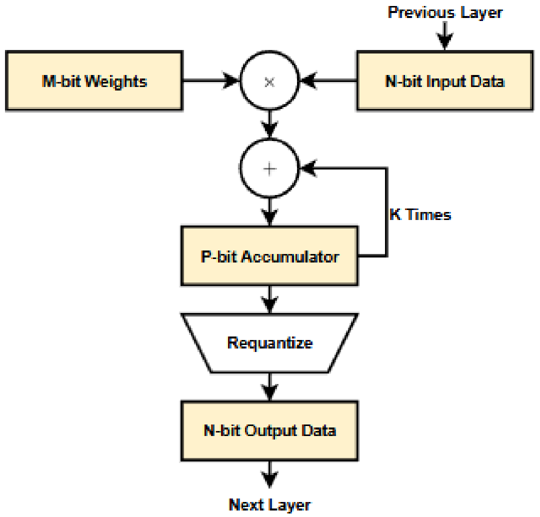
- Accumulator-Aware Quantization with Guaranteed Overflow Avoidance
- Accumulator-Aware Post-Training Quantization for Large Language Models
- Improving Quantization with Post-Training Model Expansion
- Qronos: Correcting the Past by Shaping the Future

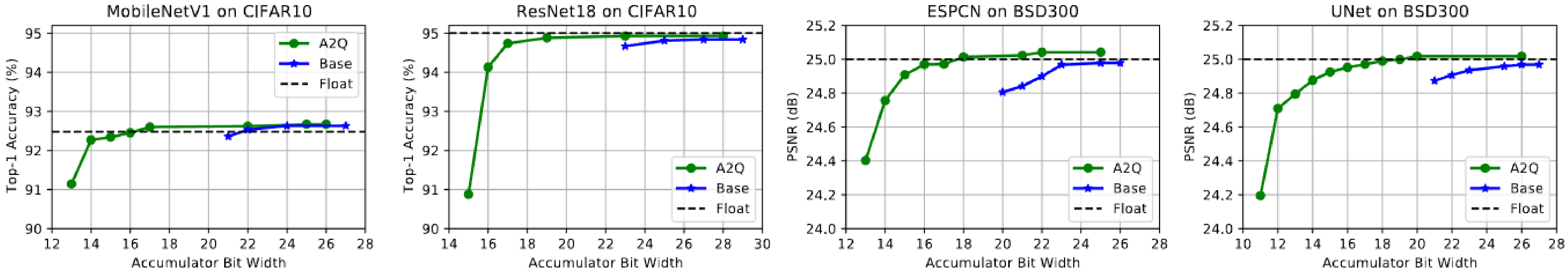
A2Q: Accumulator-Aware Quantization with Guaranteed Overflow Avoidance

The idea:

Given a quantization configuration for a layer (i.e., nn.Linear), A2Q makes sure that the dot-product can be stored in P bits without overflow.

This is done by enforcing normalization on the L1 Norm of the weight during QAT.





[2308.13504] A2Q: Accumulator-Aware Quantization with Guaranteed Overflow Avoidance

together we advance_

Accumulator-Aware Post-Training Quantization for Large Language Models

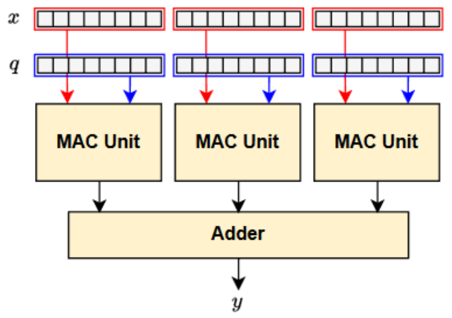
The idea:

Starting from the A2Q paradigm, AXE extends it to work with PTQ algorithms like GPTQ, by enforcing a layer-wise, tile-wise constraint during quantization.

This is more amenable to LLM given their size and the complexity of applying QAT to these models, while being the first algorithm designed to directly target a multi-stage accumulation datapath!

Table 1: We report the WikiText2 perplexity results when evaluating AXE on Pythia models quantized to W4A8 for 32-bit or 16-bit accumulation in tiles of 128 elements using either GPFQ or OPTQ with Hadamard-based incoherence processing. We use 128×16b to denote $P_I = 16$ and $T = 128$, from Eq. 20.

		70M	160M	410M	1.0B	1.4B	2.8B	6.9B	12B
	Float16	41.1	23.7	14.1	11.7	10.5	9.2	8.3	7.7
GPFQ	128×32b	56.8	35.2	19.4	12.3	11.2	9.5	8.6	7.9
	128×16b	76.4	55.5	23.2	12.7	11.8	9.8	8.7	8.0
OPTQ	128×32b	50.6	32.7	22.4	12.4	11.4	9.5	8.5	7.9
	128×16b	85.6	75.5	34.5	13.1	12.4	9.9	8.6	8.0



(b)

[2409.17092] Accumulator-Aware Post-Training Quantization for Large Language Models

AMD
together we advance_

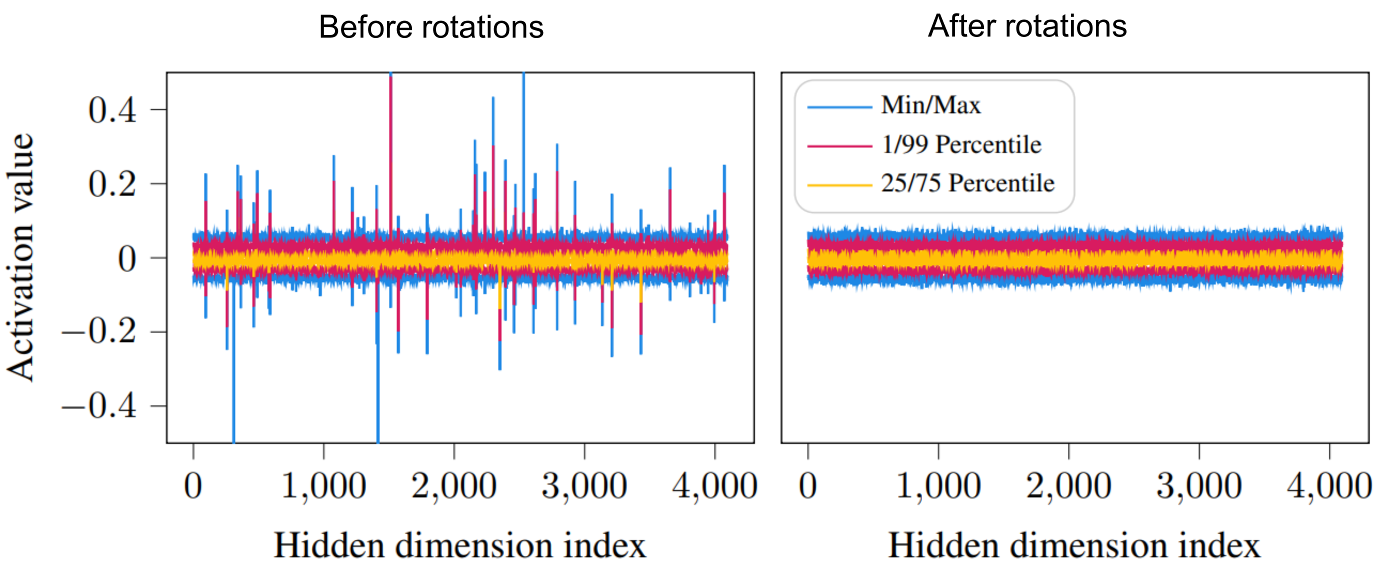
Improving Quantization with Post-Training Model Expansion

The idea:

This method offers the possibility of improving the model’s accuracy by simply adding extra parameters to a subset of layers, without retraining.

It builds on the Rotation-invariant transforms exploit, made popular by QuaRot and SpinQuant:

$X^T W = (RX)^T (RW) = X^T R^{-1} R W = X^T W$, where $R^T = R^{-1}$



[2503.17513] Improving Quantization with Post-Training Model Expansion
[2404.00456] QuaRot: Outlier-Free 4-Bit Inference in Rotated LLMs

Improving Quantization with Post-Training Model Expansion

The idea:

It builds on the Rotation-invariant transforms exploit:

$X^T W = (RX)^T (RW) = X^T R^{-1} R W = X^T W$, where $R^T = R^{-1}$

In QuaRot/SpinQuant, the rotation R is a square matrix; changing it to a rectangular shape, allows to perform a feature expansion, thus providing more channels along which the quantization error can be spread.

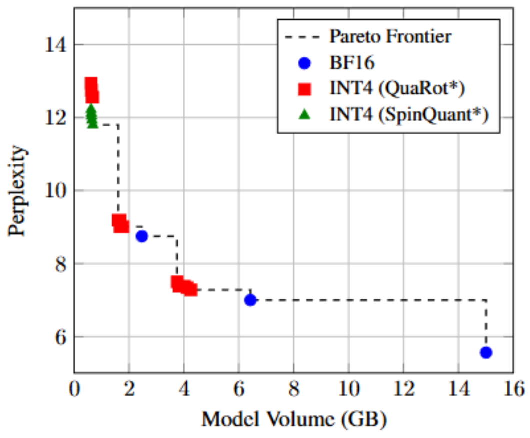


Figure 1: Comparison between Wikitext2 perplexity scores and model volume in GB.

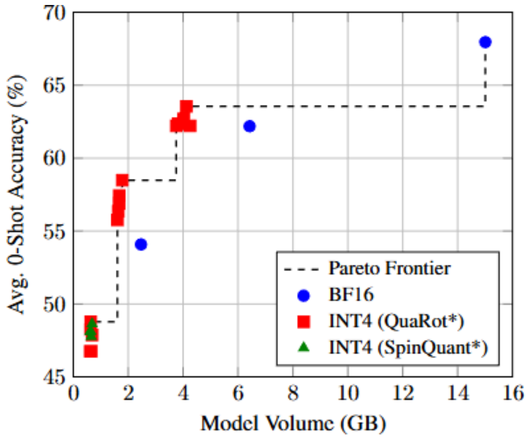


Figure 2: Comparison between geometric mean of 0-shot evaluation and model volume in GB.

These plots shows how adding more parameters to certain layers on Llama 1B, 3B and 8B can allow to improve accuracy without changing data-type.

[2503.17513] Improving Quantization with Post-Training Model Expansion

Qronos: Correcting the Past by Shaping the Future

The idea:

Qronos improves on the idea of GPTQ, by combing error diffusion with propagation of quantization error introduced by every previous layer in the network.

This allows to improve accuracy with only a small increase in quantization runtime.

Always reduce error w.r.t. original data and weights

Account for error from weights and activations already quantized

The weight we are trying to replace!

Account for error from future unquantized weights on quantized activations

$$\boxed{q_t} = \operatorname*{argmin}_{p \in \mathcal{A}} \frac{1}{2} \| \boxed{Xw} - \sum_{j=1}^{t-1} \boxed{q_j \widetilde{X}_j} - \boxed{p \widetilde{X}_t} - \sum_{j=t+1}^{N} \boxed{w_j^{(t-1)} \widetilde{X}_j} \|^2,$$

$$\boxed{w_{\geq t+1}^{(t)}} = \operatorname*{argmin}_{(v_{t+1}, \ldots, v_N) \in \mathbb{R}^{N-t}} \frac{1}{2} \| \boxed{Xw} - \sum_{j=1}^{t} \boxed{q_j \widetilde{X}_j} - \sum_{j=t+1}^{N} \boxed{v_j \widetilde{X}_j} \|^2.$$

Find new weights that preserve functionality

[2505.11695] Qronos: Correcting the Past by Shaping the Future... in Post-Training Quantization

Qronos: Correcting the Past by Shaping the Future

The idea:

Qronos improves on the idea of GPTQ, by combing error diffusion with propagation of quantization error introduced by every previous layer in the network.

This allows to improve accuracy with only a small increase in quantization runtime.

Algorithm 1 Qronos

$H^{-1} = (\widetilde{X}^\top \widetilde{X})^{-1} = LL^\top$
for every w in W (in parallel) **do**
  $q = \mathbf{0}^N$
  $w^{(0)} \leftarrow \mathrm{copy}(w)$
  $q_1 = \mathcal{Q}\left(\dfrac{\widetilde{X}_1^\top (Xw - \widetilde{X}_{\geq 2} w^{(0)}_{\geq 2})}{\|\widetilde{X}_1\|^2}\right)$ ▷ Calc. q_1 from (8), use (16) for a memory efficient ver.
  $w^{(1)}_{\geq 2} = \widetilde{X}^\dagger_{\geq 2}\left(Xw - \widetilde{X}_1 q_1\right)$ ▷ Calc. $w^{(1)}_{\geq 2}$ from (9), see (17) for a memory efficient ver.
  for $t = 2$ to N **do** ▷ Efficient implementation (Lemma 3.2)
    $q_t = \mathcal{Q}(w_t^{(t-1)})$
    $w^{(t)}_{\geq t+1} = w^{(t-1)}_{\geq t+1} - L_{\geq t+1,t} \cdot (w_t^{(t-1)} - q_t)/L_{tt}$
  end for
end for
return Q

Table 3: **Weight-activation quantization of Llama3 models.** We individually apply various quantization transforms (stage 1) to isolate the impact of different rounding functions (stage 2).

		W4A4KV16						W4A4KV4					
		WikiText2 (↓)			0-shot (↑)			WikiText2 (↓)			0-shot (↑)		
Stage 1	Stage 2	1B	3B	8B	1B	3B	8B	1B	3B	8B	1B	3B	8B
BF16	-	8.9	7.1	5.9	59.4	67.5	74.4	8.9	7.1	5.9	59.4	67.5	74.4
QuaRot	RTN	22.0	12.6	9.6	45.4	55.0	62.6	41.8	22.0	15.9	41.5	49.8	57.4
	OPTQ	14.3	9.8	8.0	50.4	59.9	66.7	19.8	14.3	10.3	45.8	56.2	64.1
	GPFQ	13.6	9.3	7.6	**50.9**	60.9	67.6	22.0	14.7	11.4	43.3	53.9	59.8
	Qronos	**13.2**	**9.1**	**7.4**	**50.9**	**61.5**	**68.9**	**17.8**	**11.6**	**9.3**	**47.8**	**57.3**	**64.8**
SpinQuant	RTN	20.1	12.8	9.3	47.4	57.5	63.7	34.7	20.3	13.3	42.6	52.5	60.4
	OPTQ	13.6	9.4	7.7	50.5	60.2	67.8	17.7	11.9	10.5	**49.4**	57.2	63.4
	GPFQ	13.5	9.2	7.6	50.8	61.0	66.6	21.5	12.3	10.8	44.1	53.6	60.7
	Qronos	**12.4**	**8.7**	**7.2**	**52.1**	**62.1**	**68.4**	**16.3**	**11.1**	**8.7**	48.2	**58.2**	**66.0**

[2505.11695] Qronos: Correcting the Past by Shaping the Future... in Post-Training Quantization

Conclusion

We plan to keep improving Brevitas with new algorithms, export flows, support for new models and more.

We focus on stability, composability and extendability, while trying to stay up-to-date with the latest and greatest pytorch features, e.g. torch.compile

We would love to collaborate!

If there is anything currently missing that you might need, reach us out and we’ll do our best to help.

If you want to learn more, check out our documentation here:

https://xilinx.github.io/brevitas/v0.12.1/

AMD