Brevitas: Neural Network Quantization in PyTorch

Pablo Monteagudo Lago

2026

PyTorch Conference 2026

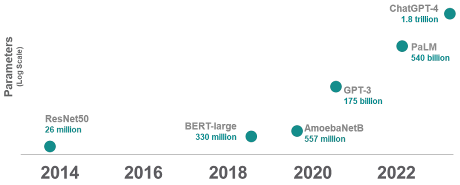
AMD
together we advance_

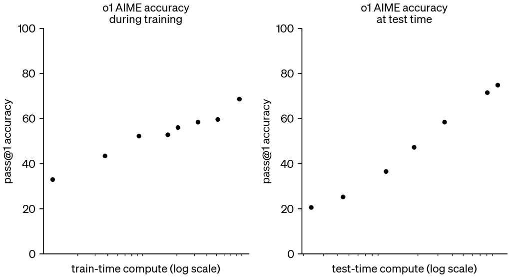
Outline

1. **Motivation** for quantization
2. **Brevitas overview** and the quantization landscape
3. **Flexibility** and **composability** in Brevitas
4. **Research** with Brevitas: **Qronos**
5. **End-to-end quantization** with Brevitas

The Cost of Deep Learning is Growing

- Neural network quality is tightly correlated with size
- Increased inference iterations correlate with higher reasoning capabilities for LLMs
- More than ever, we need to improve the trade-off between **inference cost** and **quality**

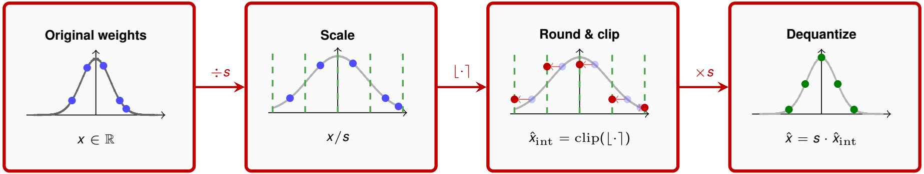




source: Learning to Reason with LLMs — OpenAI (accessed 11/20/2024)

Quantization Can Reduce Inference Costs

- **Quantization:** mapping weights/activations from a **high-precision** (e.g. `bfloat16`) representation to a **lower-precision** one (e.g. `int8`, `mxfp4`)



The benefits of using lower precision depend on the platform, but generally fall into:

Memory

16-bit $\rightarrow$ 4-bit:

4$\times$ less storage

Power

int8 vs int32 multiply:

15$\times$ less energy

Throughput

fp8 vs fp32 (MI355X):

8$\times$ more FLOP/s

Area

Lower-precision units

are **much smaller**

Brevitas: Neural Network Quantization in PyTorch

- Open-source PyTorch library from AMD for quantization research
  - ~8 years, >500k PyPI downloads, ~1.5k GitHub stars ★
  - Targets the **full AMD product range**: CPU, GPU, NPU and FPGA
- Supports both **Quantization-Aware Training** and **Post-Training Quantization**
  - Modular quantized layers matching PyTorch `nn.Module` interface
- Built for **composability and flexibility**, enabling SOTA quantization research
  - MixQuant (arXiv)
  - Qronos (ICLR 2026)
  - Accumulator-Aware Quantization (ICML 2024)
- **PyTorch ecosystem** integration
  - HuggingFace Transformers, vLLM, GGUF / llama.cpp

Brevitas

PyTorch

vLLM

LLaMA C++

AMD Open Source

Brevitas is for **researchers** and **practitioners** – from experimentation to deployment

Brevitas: **Quantization Landscape**

- Quantization involves many **interacting design choices**
  - **Datatypes:** Integer, Minifloat, MX, Binary, custom formats, . . .
  - **Scale** / **Zero-Point:** Per-tensor, per-channel, per-group; float, power-of-two, . . .
  - **Accuracy Preservation:** Rotations, equalization, GPTQ, learned rounding, . . .
- The number of valid combinations grows **combinatorially**
- Most tools focus on a few **fixed recipes** for specific use cases
  - **TorchAO**, **llama.cpp**, **LLM-Compressor**: performance-focused, pre-defined flows, e.g.
    - INT4 + GPTQ ✓
    - FP4 + GPTQ ✗

recommended hardware	weight	activation	quantized training	QAT	PTQ data algorithms	quantized inference
H100, B200	float8 rowwise	float8 rowwise	🟢	🟢	🟣	🟢
H100	int4	float8 rowwise	🟣	🟢	🟠	🟢
A100	int4	bfloat16	🟣	🟢	🟡: HQQ, AWQ	🟢
A100	int8	bfloat16	🟣	🟢	🟣	🟢
A100	int8	int8	🟡	🟢	🟣	🟢
B200	nvfp4	nvfp4	🟠	🟡	🟣	🟡
B200	mxfp8	mxfp8	🟡	🟣	🟣	🟡
B200	mxfp4	mxfp4	🟣	🟠	🟠	🟡

source: TorchAO Quantization Table

How do we navigate this space? We need **flexible, composable tooling**

The Need for Flexible Tooling

[Public]

The Need for Flexible Tooling: Quantization Building Blocks

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

Brevitas allows you to easily customize **every** aspect of the quantization pipeline:

Precision

- Integer, Minifloat, MX, Binary
- OCP formats, custom exp/mantissa
- Signed/unsigned, rounding modes

```
from brevitas.quant import (
  Int8WeightPerChannelFloat,
  Int8ActPerTensorFloat)

# Customize precision
weight_quant =
      Int8WeightPerChannelFloat.
      let(
  bit_width=4,
  quant_format="float_ocp_e2m1")

act_quant = Int8ActPerTensorFloat
      .let(
  bit_width=4,
  quant_format="float_ocp_e2m1")
```

Scale, Zero-point

- Float, power-of-two, quantized
- Per tensor / channel / group
- Static or dynamic

```
# Customize scale & granularity
weight_quant =
      Int8WeightPerChannelFloat.
      let(
  quant_granularity="per_group",
  group_size=32,
  scale_precision="po2_scale",
  param_method="mse")

act_quant = Int8ActPerTensorFloat
      .let(
  scale_type="dynamic")
```

Maximise Accuracy

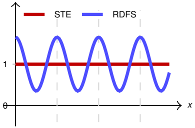
- **Stage 1** – Rotations, Equalization
- **Stage 2** – GPxQ, Qronos, AutoRound, QAT
- Bias correction, learned rounding

```
brevitas_ptq_llm
 --model meta-llama/Llama-3.2-1B
 # Stage 1: Rotations
 --rotation fused_no_fx
 --rotation-mode had
 # Stage 2: Error correction
 --gpxq-act-order
 --qronos-alpha 1e-3
```


AMD
together we advance_

The Need for Flexible Tooling: Working Example

- The gradient of $\lfloor \cdot \rceil$ vanishes: a **surrogate** is needed. The **STE** is flat: it misses the **periodic structure** of rounding.
- Researchers from Microsoft proposed a **Fourier-based surrogate** (RDFS)



- This innovation only affects the **rounding module**, while the rest of the pipeline remains untouched.

1. Define the rounding module:

```
class RoundRDFS(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x: torch.Tensor):
        ctx.save_for_backward(x)
        return torch.round(x)

    @staticmethod
    def backward(ctx, grad_output):
        x, = ctx.saved_tensors
        # Fourier surrogate: captures periodicity
        cos_term = torch.cos(
            (x + x.round()) * math.pi)
        grad = 2 / (1 + 0.93 * cos_term) - 1
        return grad * grad_output
```

2. Combine with *any* existing quantizer:

```
from brevitas.quant import Int8WeightPerTensorFloat

quantizer = Int8WeightPerTensorFloat.let(
    float_to_int_impl_type=ROUND_RDFS)
```

The researcher focuses only on the new rounding logic, while **Brevitas handles the integration** with **any** quantizer.

[2601.19320] StableQAT: Stable Quantization-Aware Training at Ultra-Low Bitwidths

AMD together we advance_

The Need for Composable Tooling

The Need for **Composable** Tooling: **PTQ Stages**

Naïvely mapping to a low-precision grid **degrades model quality**. Post-Training Quantization algorithms address this in two complementary stages:

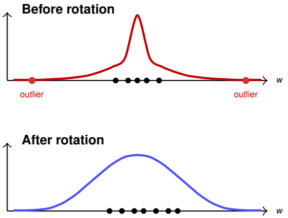
Pre-trained weights, Calibration data, Target format(s)

⇒

How can I transform my model in a way that ***mitigates*** quantization error?

Stage 1: Transform

- SpinQuant / QuaRot (rotations)
- SmoothQuant (equalization)
- MagR
- . . . and many more!



Rotations compact the distribution.

⇒

Quantized model, export for deployment

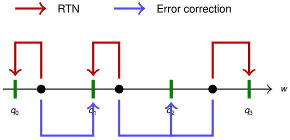
vLLM

The Need for **Composable** Tooling: **PTQ Stages**

Naïvely mapping to a low-precision grid **degrades model quality**. Post-Training Quantization algorithms address this in two complementary stages:

Pre-trained weights, Calibration data, Target format(s)

⇒



May assign non-nearest grid point.

How can I map to a discrete grid and ***recover*** from quantization error?

Stage 2: Round

- Qronos (ICLR 2026)
- AutoRound
- GPTQ
- QAT
- . . . and many more!

⇒

Quantized model, export for deployment

vLLM

The Need for Composable Tooling: PTQ Stages

Naïvely mapping to a low-precision grid **degrades model quality**. Post-Training Quantization algorithms address this in two complementary stages:

Pre-trained weights, Calibration data, Target format(s)

⇒

How can I transform my model in a way that ***mitigates*** quantization error?

Stage 1: Transform

- SpinQuant / QuaRot (rotations)
- SmoothQuant (equalization)
- MagR
- . . . and many more!

How can I map to a discrete grid and ***recover*** from quantization error?

Stage 2: Round

- Qronos (ICLR 2026)
- AutoRound
- GPTQ
- QAT
- . . . and many more!

⇒

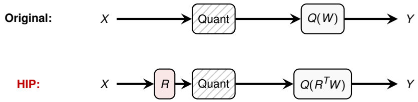
Quantized model, export for deployment

vLLM

The Need for **Composable** Tooling: **Algorithm Composition**

- MXINT2 + **RTN** on Llama 3.2 1B: **catastrophic** (PPL $\sim$1.7M).
- Finer group size (32 $\rightarrow$ 16) and scales (E8M23) does **not** help. The problem is **not** the datatype.
- We need algorithms to **mitigate the quantization error**.
- **HIP** suppresses outliers: **notable recovery** even with coarser E8M0 (PPL $\sim$1.4k).
- **HIP** + **AutoRound**: PPL **15.9**. **Composition** matters more than expressive datatypes.

Llama 3.2 1B	**Alg.**	**GS**	**Scale**	**PPL**
BF16	–	–	–	**8.9**
MXINT2	RTN	32	E8M0	$\sim$1.7M
MXINT2	RTN	16	E8M23	$\sim$1.8M
MXINT2	HIP	32	E8M0	$\sim$1.4k
MXINT2	HIP+AutoRound	32	E8M0	**15.9**



HIP introduces Hadamard rotations in the computational graph

Brevitas enables combining PTQ techniques in a **single**, **reproducible** pipeline

[2402.04396] QuIP#: Even Better LLM Quantization with Hadamard Incoherence and Lattice Codebooks

Research with Brevitas

Research with Brevitas: Qronos (ICLR 2026)

Idea 💡

- **Qronos** alternates between **greedy quantization** and **error diffusion** via optimal update rules, accounting for the quantization error in previous layers
- Improves accuracy with only a small increase in quantization runtime compared to **GPTQ**

	GPTQ	Qronos
Greedy quantization	$q_t = \underset{p \in \mathcal{A}}{\operatorname{argmin}} \ \frac{1}{2} \left\| \widetilde{X} w - \sum_{j=1}^{t-1} q_j \widetilde{X}_j - p \widetilde{X}_t - \sum_{j=t+1}^{N} w_j^{(t-1)} \widetilde{X}_j \right\|^2$	$q_t = \underset{p \in \mathcal{A}}{\operatorname{argmin}} \ \frac{1}{2} \left\| X w - \sum_{j=1}^{t-1} q_j \widetilde{X}_j - p \widetilde{X}_t - \sum_{j=t+1}^{N} w_j^{(t-1)} \widetilde{X}_j \right\|^2$
Error diffusion	$w_{\geq t+1}^{(t)} = \underset{v_{t+1}, \dots, v_N}{\operatorname{argmin}} \ \frac{1}{2} \left\| \widetilde{X} w - \sum_{j=1}^{t} q_j \widetilde{X}_j - \sum_{j=t+1}^{N} v_j \widetilde{X}_j \right\|^2$	$w_{\geq t+1}^{(t)} = \underset{v_{t+1}, \dots, v_N}{\operatorname{argmin}} \ \frac{1}{2} \left\| X w - \sum_{j=1}^{t} q_j \widetilde{X}_j - \sum_{j=t+1}^{N} v_j \widetilde{X}_j \right\|^2$

X: original activations $\widetilde{X}$: activations produced after quantizing the previous layers

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

Research with Brevitas: Qronos (ICLR 2026)

- **Qronos outperforms other greedy error-correction** algorithms (GPTQ, GPFQ)
- The accuracy gap **increases as bitwidth decreases**, where error correction matters most
- **Composing with Stage 1 is critical**
  - **HIP** suppresses outliers **before** quantization
  - **Qronos** + **HIP** consistently outperforms **Qronos** alone

Without Stage 1 (RTN baseline):

		W3						W4					
		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
	RTN	2e4	1e4	3e4	32.3	32.4	32.6	18.0	10.1	8.4	49.1	60.8	67.4
	OPTQ	42.5	13.8	11.4	37.5	48.1	53.8	10.4	7.8	6.5	54.3	63.4	71.0
None	GPFQ	35.3	13.4	11.1	35.7	49.9	53.5	10.4	7.8	6.5	56.0	**65.2**	71.2
	GPTAQ	28.4	12.6	10.3	39.3	49.6	**57.1**	10.3	7.8	6.5	**56.3**	63.3	71.0
	Qronos	**22.8**	**11.3**	**9.3**	**39.5**	**53.1**	56.7	**10.1**	**7.6**	**6.4**	56.2	64.5	**72.0**

With Stage 1 (**HIP** for outlier suppression):

	RTN	7e2	3e2	1e2	34.2	33.3	36.3	13.8	8.8	7.2	52.0	62.8	70.0
	OPTQ	16.1	10.3	8.6	44.1	56.6	58.8	9.9	7.6	6.3	56.8	**66.1**	72.1
HIP	GPFQ	16.6	10.4	8.6	44.9	54.8	58.9	9.9	7.6	6.3	56.5	65.7	72.0
	GPTAQ	14.7	9.9	8.3	46.5	56.9	59.3	9.8	**7.5**	6.3	**57.8**	66.0	**72.4**
	Qronos	**12.9**	**9.3**	**7.8**	**48.1**	**59.6**	**62.2**	**9.6**	**7.5**	**6.2**	57.1	65.9	71.0

PTQ **algorithm composition** is key for performance: **Brevitas** enables it transparently

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

AMD together we advance_

End-to-End Quantization with Brevitas

PTQ Flow: End-to-End Quantization with Brevitas

From pre-trained model to deployed quantized inference in a few steps:

```
from transformers import AutoModelForCausalLM
from brevitas.graph.quantize import layerwise_quantize
from brevitas.graph.equalize \
    import GraphRotationEqualization

model = AutoModelForCausalLM.from_pretrained(
    "meta-llama/Llama-3.2-1B")

model = apply_rotation_equalization(model)

model = layerwise_quantize(
    model, compute_layer_map=quant_config)

from brevitas.graph.qronos import apply_qronos
apply_qronos(model, calibration_loader)

from brevitas.export import export_for_vllm
export_for_vllm(model, "quantized_llama_vllm/")
```

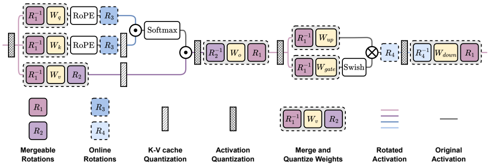
1. **Load** a HuggingFace model
2. **Stage 1:** Apply rotation equalization (via Dynamo graph tracing)
3. **Quantize** weights and activations
4. **Stage 2:** Apply **Qronos** for error correction
5. **Export** for deployment (**vLLM**, GGUF, ONNX, . . .)

PTQ Flow: Stage 1, Outlier Mitigation via Rotations

- **Rotation-based equalization** mitigates outliers, making distributions more amenable to quantization
- Brevitas uses **Dynamo** to trace the graph and identify **rotation-equivariant regions**

Rotation-Equivariance Property

$$g(f(X;W_2);W_1) = g(f(X;W_2R^T);RW_1)$$



Rotation-equivariant regions in a transformer block

Other libraries require **manually modifying** the model to insert rotations, e.g. adding `R1`, `R2` parameters to every attention forward pass:

```
def forward(
    self,
    hidden_states: torch.Tensor,
    attention_mask: Optional[torch.Tensor] = None,
    position_ids: Optional[torch.LongTensor] = None,
    past_key_value: Optional[Cache] = None,
    output_attentions: bool = False,
    use_cache: bool = False,
    cache_position: Optional[torch.LongTensor] = None,
    position_embeddings: Optional[
        Tuple[torch.Tensor, torch.Tensor]
    ] = None,
    R1=None,
    **kwargs,
) -> Tuple[...]:
    bsz, q_len, _ = hidden_states.size()

    query_states = self.q_proj(hidden_states, R1)
    key_states = self.k_proj(hidden_states, R1)
    value_states = self.v_proj(
        hidden_states, R1, R2=self.R2.weight)
```

Source: SpinQuant reference implementation. Must be repeated for **every** architecture.

AMD
together we advance_

[Public]

PTQ Flow: Inserting Quantization Nodes

Three ways to introduce quantization operators:

Manual

Define the model directly with quantized layers from `brevitas.nn`

```
import brevitas.nn as qnn
from brevitas.quant import (
    Int8WeightPerTensorFloat)

class QuantNet(nn.Module):
  def __init__(self):
    super().__init__()
    wq = Int8WeightPerTensorFloat
    self.fc1 = qnn.QuantLinear(
        64, 32, weight_quant=wq)
    self.fc2 = qnn.QuantLinear(
        32, 10, weight_quant=wq)
```

Programmatic

Replace modules in an existing model via `layerwise_quantize`

```
from brevitas.graph.quantize import (
    layerwise_quantize)

# Replace nn.Linear -> QuantLinear
# according to layer_map config
model = layerwise_quantize(
    model=model,
    compute_layer_map=layer_map,
    name_blacklist=name_blacklist)
```

FX Graph

Trace the computation graph and insert quantization nodes automatically

```
from brevitas.graph.quantize import (
    preprocess_for_quantize,
    quantize)

# Trace, merge BN, equalize
model = preprocess_for_quantize(
    model)
# Insert quant nodes in graph
quant_model = quantize(model)
```

From **full control** to **fully automatic** – choose the right level of abstraction for your use case

PTQ Flow: **Stage 2, Error Correction**

- After quantization nodes are inserted, **greedy error-correction** algorithms mitigate the quantization error
- Brevitas supports multiple **Stage 2** algorithms through a unified API

Applying Qronos

```python
from brevitas.graph.qronos import apply_qronos
apply_qronos(model, calibration_loader)
```

Applying GPTQ

```python
from brevitas.graph.gptq import apply_gptq
apply_gptq(model, calibration_loader)
```

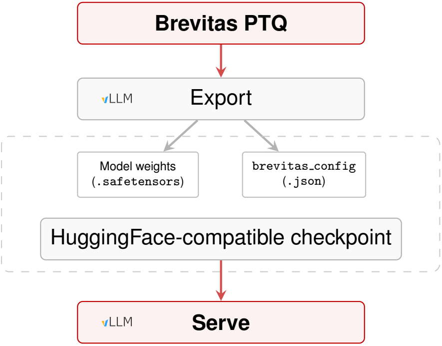
Also supports: **AutoRound**, **QAT**, **Bias Correction**...

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**

PTQ Flow: Export, From Research to Deployment

- Export to neural network inference frameworks
  - The path from research to **production**!
- Brevitas supports exporting to
  - **vLLM** – high-throughput LLM serving
  - **GGUF** (llama.cpp) – local/edge deployment
  - **ONNX** / **QONNX**
  - PyTorch (native ops)
  - SHARK / AITER
- Ensures a **smooth transition** from experimentation to deployment
- Want something else supported? Reach out!




```
vllm serve ./model --quantization quant_brevitas
```

Summary

- Brevitas is an open-source PyTorch quantization library from AMD
- **Unified environment** for modern PTQ: Qronos, SpinQuant, AutoRound
    - **Compose** complementary techniques
- Leverages latest **PyTorch features**
- Integrates with the **PyTorch ecosystem**
- Export to **vLLM**, **GGUF**, ONNX

https://github.com/Xilinx/brevitas

Reach out! We welcome contributors and collaborators

COPYRIGHT AND DISCLAIMER

AMD