Reducing the hardware gap for custom accelerators through quantization-aware training

Bastien Barbe , Romain Bouarah, Florent de Dinechin, Anastasia Volkova
INSA Lyon, Inria, CITI (UR3720), 69621 Villeurbanne, France
{firstname.lastname}@insa-lyon.fr

***Abstract*—Popular machine learning (ML) frameworks like PyTorch and TensorFlow now allow to train ML models for non-standard number representations thanks to quantization-aware training (QAT). However, these models tend to perform differently when deployed on their target hardware, due to improper modeling of the actual hardware arithmetic. This work introduces HATorch, a PyTorch-based training framework that supports arbitrary quantization schemes and transparent model-hardware co-design to reduce the hardware gap.**

***Index Terms*—Quantization, Hardware Aware Training, Convolutional Neural Networks, ML Accelerators**

I. Introduction

Quantization maps continuous numerical values to a finite set of discrete values [1]. In deep learning, quantization is widely used to reduce both the memory footprint and computational cost of Neural Networks (NN) at inference by replacing FP32 arithmetic with lower-precision formats (e.g., FP16, INT8, or INT4) for weights, activations and biases without significantly impacting accuracy [2]–[6].

Two main strategies are used in practice. Post-Training Quantization (PTQ) converts a trained full-precision model to low precision. Quantization-Aware Training (QAT) incorporates quantization into the training process itself. This work addresses QAT, which generally yields better accuracy [1] for low bit-width quantization, although it requires access to the training data and additional training time (QAT usually starts with a network trained in FP32).

As quantization is essential in providing efficient inference on hardware accelerators (GPUs, TPUs, FPGAs, ASICs), popular machine learning frameworks such as PyTorch [7] and TensorFlow [8] provide some QAT support.

One could assume that quantization to simpler formats should simplify the original network. However, the opposite is true. Indeed, FP hardware is popular because it simplifies numerics by automatically managing the scaling of the data. To lower an FP32 NN to an INT4 one, for example, QAT has to introduce such scaling explicitly. The main QAT approach, sometimes called *fake quantization* and detailed in [9] adds to the network new FP32 parameters, such as scaling factors and clipping boundaries. Actually most of QAT is about learning these parameters. Later steps of lowering to hardware have to manage this extra complexity, and this ironically includes quantizing some of the FP32 scaling factors left by QAT. There is therefore a difference between the model that has been trained and the model that is actually deployed on the hardware. This entails a difference in accuracy called the *hardware gap* [10] (Figure 1). Here "model" means the parameters of the network, but also the arithmetic that connects them. This gap is even more significant when the hardware target uses custom data formats and specialized arithmetic units [11].

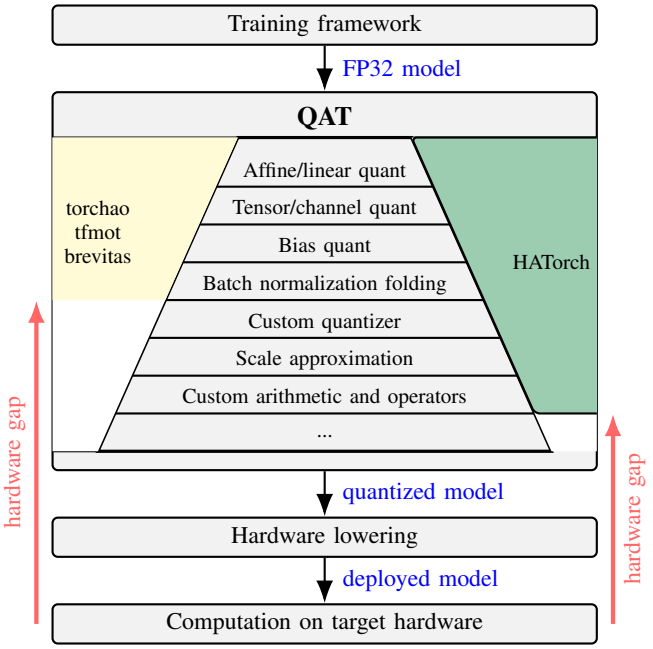


Fig. 1: Reducing the hardware gap on custom hardware requires hardware-aware quantization.

The objective of this paper is to reduce the hardware gap by improving the hardware-awareness of QAT tools. The ultimate goal is to train for what will be deployed, or conversely to deploy exactly the model that has been trained. As research is very active on hardware-friendly number systems, this work generalizes existing QAT techniques so that they become independent of the number system. A last objective is to minimize the additional model complexity introduced by QAT and remove irrelevant degrees of freedom.

These ideas motivate the development of HATorch, an Hardware-Aware Training framework based on PyTorch. It is demonstrated on standard fixed-point formats, and two non-standard formats (logarithmic, and shift-and-add friendly).

The code is available at https://github.com/GrannyDev/hatorch/tree/hipeac.

The paper is organized as follows: Section II reviews the foundations of quantization and the basic QAT technique. Section III presents the interface of the proposed HATorch framework, and Section IV details the transformations it performs to lower an FP32 model to a hardware-aware model. Section V reports empirical results obtained with HATorch on standard benchmarks using custom number formats. Finally, Section VI concludes and outlines future research directions.

II. Quantization of neural networks

Let Ω be a subset of $\mathbb{R}$ (quantization levels):

$$\Omega = \{q_i\} \quad \text{with } q_i < q_j \text{ for } i < j \quad . \tag{1}$$

An example is the set of 4-bit two's complement integers $\Omega = \{-8, -7, \ldots, 0, \ldots, 6, 7\}$ shown in Figure 2. This example is a subset of $\mathbb{Z}$, but all the approach presented here works for any subset of $\mathbb{R}$. For instance, a Logarithm Number System (LNS) typically uses a finite number of real values.

A quantization operator Q is a monotonic function that maps any real value to one of the discrete levels:

$$Q : \mathbb{R} \to \Omega, \qquad Q(x) = q_k \text{ with } q_k \in \Omega \quad . \tag{2}$$

For all practical purpose in this work, $\mathbb{R}$ is identified to the set of 32-bit floating-point (FP32) numbers[1].

Two canonical quantizations to $\Omega = \mathbb{Z}$ are defined by basic mathematics and available in any computing systems: the truncation function $\lfloor \cdot \rfloor$, and the rounding to nearest (with ties up) $\lfloor x \rceil = \lfloor x + \frac{1}{2} \rfloor$. These are actually rounding functions: a rounding function is a quantization with the additional property that $\forall q \in \Omega \quad \text{round}(q) = q$.

Rounding to integers cannot be used directly as a quantization to 4-bit integers since $\mathbb{Z}$ is an infinite set. Some transformation is required to align the range of values to be quantized (as shown at the top of Figure 2) with the range $\{q_{\min}, \ldots, q_{\max}\}$ of representable numbers (shown at the bottom). Usually, it is an affine transformation.

[1]Strictly speaking the set of FP32 number is iself finite and discrete. Indeed, the IEEE-754 standard for floating-point arithmetic also defines five quantification functions $\mathbb{R} \to$ FP32.

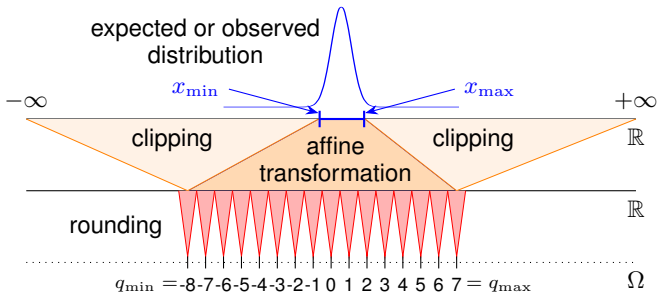


Fig. 2: Quantization involves clipping, scaling and rounding

A. Affine transformation

Given the finite quantization set $\Omega = \{q_{\min}, \ldots, q_{\max}\}$ and two real values $x_{\min}$ and $x_{\max}$, an affine transformation mapping a real number $x \in [x_{\min}, x_{\max}]$ into $[q_{\min}, q_{\max}]$ is defined as:

$$\begin{aligned} \text{scale}(x) &= (x - x_{\min}) \frac{q_{\max} - q_{\min}}{x_{\max} - x_{\min}} + q_{\min} \\ &= \frac{x - z}{s} \end{aligned} \tag{3}$$

where

$$s = \frac{x_{\max} - x_{\min}}{q_{\max} - q_{\min}} \tag{4}$$

$$z = x_{\min} - s \cdot q_{\min}. \tag{5}$$

In neural networks, the scaling parameters s and z can be defined either per-layer (using a single set of parameters for the entire layer) or per-channel (using different parameters for each output channel). This flexibility is particularly useful for achieving better accuracy, especially under extreme quantization (4 bits or less) [12].

B. Clipping (clamping, saturation)

For most activations, their range depends on the input, and their distribution can only be estimated (Figure 2). A worst-case bound would be too pessimistic to be practical, therefore in such cases the values of $x_{\min}$ and $x_{\max}$ that give the best scaling must be learnt [3]. Besides, it must be complemented at runtime with some clipping (or clamping, also long known as saturation in digital signal processing):

$$\text{clip}(x) = \min(\max(x, x_{\min}), x_{\max}) \quad . \tag{6}$$

Note that for the network parameters such as weights and biases, it is possible to use the actual $x_{\min}$ and $x_{\max}$ of the deployed weight tensor. Clipping is still necessary during training, due to the weight updates, but disappears from the deployed model.

C. Quantization and dequantization

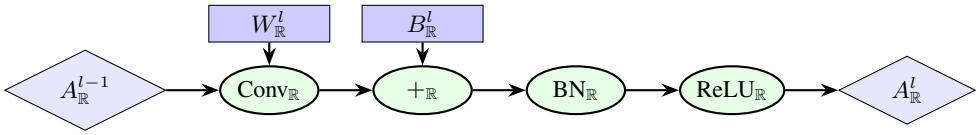
Finally, a quantization function is defined as:

$$Q(x) = \text{round}\left(\text{scale}\left(\text{clip}(x)\right)\right) \tag{7}$$

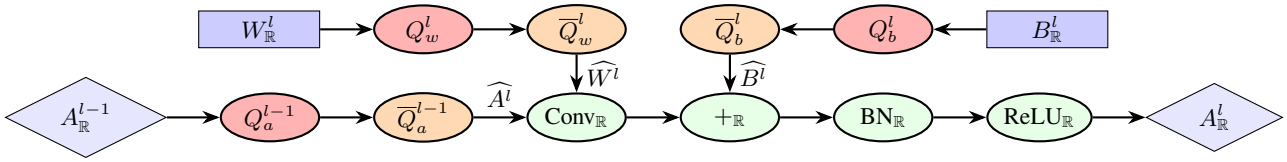
and goes top to bottom of Figure 2. A quantized number can be converted back to the original real domain by the dequantization function $\overline{Q}$, simply defined as $\overline{Q}(q) = \text{scale}^{-1}(q)$ since round is the identity on Ω and clip is the identity on $[x_{\min}, x_{\max}]$.

D. Linear (or symmetric) scaling

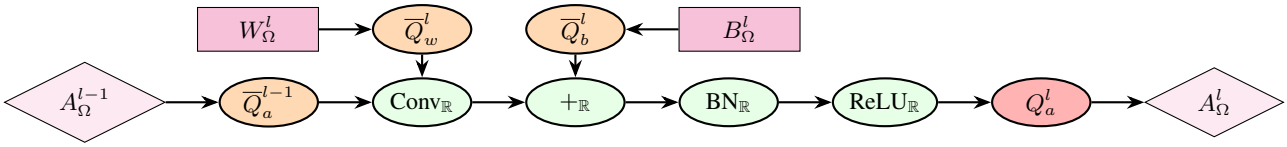
A useful special case of affine transformation is linear scaling where $x_{\min} = -x_{\max}$, $q_{\min} = -q_{\max}$, $z = 0$ and $s = x_{\max}/q_{\max}$. As the sequel will show, linear scaling of weights and biases saves computations in the deployed network: this simplification not only saves the addition of z, it also enables merging the scaling factor in the convolution. However, the set Ω may then be underused. Figure 4 shows



(a) Standard convolution layer computation graph (BN stands for Batch Normalization).



(b) Fake quantization consists in inserting quantize Q and dequantize $\overline{Q}$ operators, simulating rounding and clipping behavior.



(c) Quantized model that has to be deployed: the hardware gap is due to 1/ the remaining FP32 scaling factors in the $\overline{Q}$ operators, and 2/ the assumption that the hardware will perform an exact computation for convolution etc. (it has to requantize the activations).

Fig. 3: Computation graphs during QAT then lowering to hardware.

a source distribution living in $[-0.95, 0.7]$, mapping it to INT8 actually leads to the quantization interval $[-127; 94]$, thus not utilizing the rest of representable INT8 numbers in $[95; 127]$. Also, two's complement has one negative value without positive symmetric ($q_{\min} = -q_{\max} - 1$) and for symmetry this value is often ignored.

E. Non-uniform quantization

Quantization to a sub-range of $\mathbb{Z}$, using round to nearest as the rounding function R, is called uniform quantization. Non-uniform schemes can also be considered (an example is given in Figure 8a), where quantization levels are not evenly spaced, potentially better matching the source distribution [1], or targetting a hardware-friendly Ω such as powers of two [13], logarithm number systems [14] or shift-and-add-based multipliers [15]–[17].

F. QAT with fake quantization

QAT methods stem from pioneering work on binary neural networks [18], [19]. At its core, a QAT method (Figure 3) consists in simulating a quantized version of the network during training in the forward pass, while performing updates on the full-precision network parameters. This is called fake quantization because all the training is still performed in the original unquantized domain, for instance using the FP32 compute kernels found in GPUs.

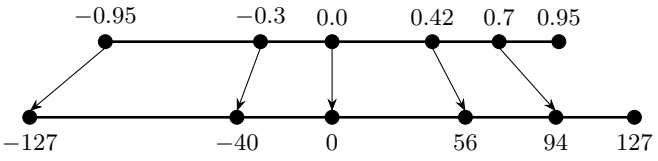


Fig. 4: Example linear quantization from [-0.95, 0.7] to INT8

Fake quantization essentially inserts quantization and dequantization operators in the computation graph (Figure 3b). In these new nodes, only the rounding function is not differentiable. It is therefore replaced with identity in the back propagation. This turns the Q function into a Straight-Through Estimator or (STE) [20], enabling schemes like Learned Step Size quantization (LSQ) [4] to learn the scaling factor in case of uniform quantization. LSQ+ [21] also learns the zero point whereas other methods like PACT [3] learn the clipping bounds for activations. The state ot the art is nuLSQ, a non-uniform variant [22], which learns the quantization set Ω (without favoring a hardware-friendly Ω).

This work identifies three main flaws with the existing QAT approaches. First, fake quantization leaves a trail of FP32 scaling factors in the computational graph (Figure 3c). This leaves hardware lowering backends to deal in a some undetermined way with those scalings, thus changing the arithmetic and potentially impacting the accuracy of the deployed inference. Second, the arithmetic of the deployed hardware is not accurately simulated by the QAT framework. This is not so much of a problem when deploying on standard GPUs, but it adds to the gap with custom hardware deployed on FPGAs using custom arithmetic. Finally, custom data formats such as logarithmic number system (LNS) [14] or shift-and-add friendly reconfigurable single constant multipliers (RSCM) [15] have their own rules of quantization that, so far, could not be captured during the QAT.

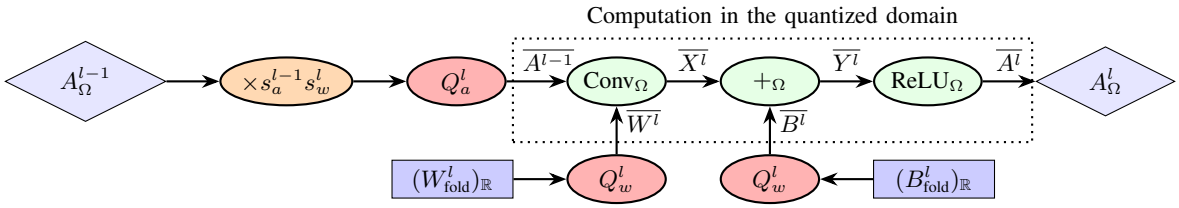


Fig. 5: HATorch quantized convolution computation graph. Note how the convolution can be performed purely in Ω arithmetic. The $\mathbb{R}$ arithmetic of the batch normalization is folded into the weights and biases as per Section IV-E.

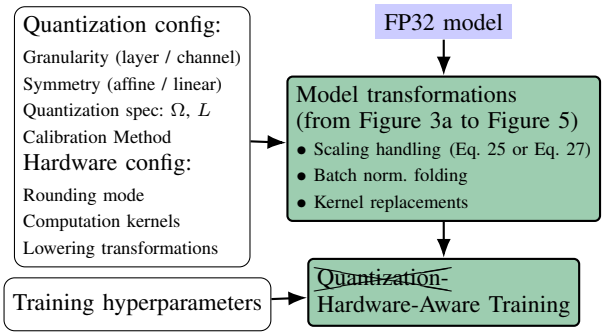


Fig. 6: HATorch training workflow

III. HATorch interface

A. Bird-eye view

Figure 6 illustrates the HATorch training workflow. The user specifies the quantization configuration, and HATorch adapts the model architecture accordingly ensuring the trained model matches the deployment constraints. Ultimately, the quantized model will be exported to ONNX format for hardware deployment, but this is still future work.

Similarly to Brevitas [23], HATorch implements quantized tensors that encapsulate both the real-valued tensor and its quantization parameters (scale, zero-point, bit-width). This abstraction greatly simplifies the management of quantization parameters throughout the model, most notably the sharing of scaling factors across layers.

A core idea in HATorch is to regroup **before training** all the scaling factors appearing in the standard QAT computational graph (Figure 3b). This is illustrated in Figure 5 and will be detailed in Section IV. Thus, the convolution and activation operations can be performed purely in the target arithmetic Ω, and the fused scaling factor can be constrained, during training, to match the target hardware.

B. A generic approach to low-bit rounding

Another core idea is to make quantization as generic as possible. In HATorch, a rounding function R is defined by

- an ordered list of N *quantization values*: $\Omega = \{q_i\}$ with $q_i \in \mathbb{R}$;
- another ordered list of $N+1$ *rounding limits*: $L = \{b_i\}$ such that $b_{\min} = -\infty$, $b_{\max} = +\infty$

such that

$$R(x) = q_i \quad \text{iff} \quad b_i \le x < b_{i+1} \quad . \tag{8}$$

Rounding to the nearest is defined by $b_i = \frac{1}{2}(q_{i-1} + q_i)$, but the list L allows the rounding itself to reflect the variation of the spacing of the q_i in case of non-uniform rounding, as illustrated for LNS by Figure 7.

HATorch implements this universal step-driven quantizer, with an interface that just consists of the two lists Ω and L.

This flexibility enables the implementation of various quantization methods, including uniform, non-uniform, logarithmic, and others.

The quantizer itself [15] is a non uniform derivation of the Learned Step Size Quantization (LSQ) method [4] and its affine counterpart LSQ+ [21]. The quantization function rounds according to (8).

Figure 8 illustrates the step-driven quantizer configured for the non-uniform quantization $\Omega = \{-20, -13, -8, -6, -5, -3, -2, -1, 0, 1, 2, 4, 5, 7, 12, 19\}$ and the corresponding rounding limits $L = \{-\infty, -17, -10, -7, -5.5, -4, -2.5, -1.5, -0.5, 0.5, 1.5, 3, 4.5, 6, 9, 15, 20, \infty\}$. This Ω corresponds to the values attainable by a shift-and-add circuit [15] shown in Figure 9. With 4 configuration bits, it provides more range than INT4 altogether with cheaper multiplier hardware. With HATorch, exploring the impact of such hardware-efficient quantization schemes is smooth and automated.

IV. HATorch details

A. Notations

Superscripts denote layer index. Subscripts in quantization/dequantization functions describe the tensor. The quantized version of a tensor T is written $\overline{T}$.

For any tensor T (e.g., activations A, weights W, bias B) in layer l let us denote

- $s_T^l \in \mathbb{R}_{>0}$ the positive real-valued scale;
- $z_T^l \in \mathbb{R}$ the zero-point.

When quantization is simulated with real-valued (FP32) arithmetic, the dequantized values are marked with a hat:

$$\widehat{T} = \overline{Q}(\overline{T}) = s \cdot \overline{T} + z, \tag{9}$$

with $z = 0$ for symmetric quantization.

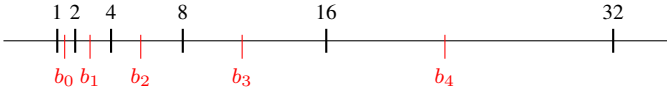
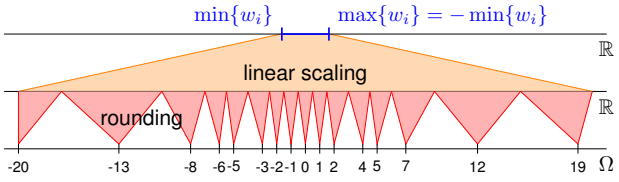
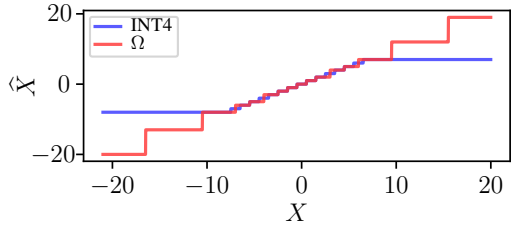


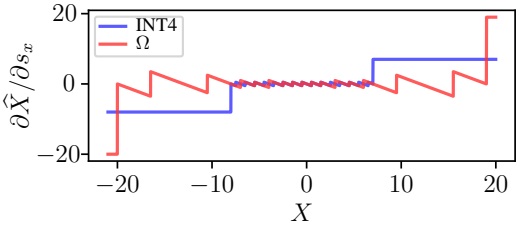
Fig. 7: Non-uniform rounding for $\Omega = \{2^i\}$, $L = \{2^{i+\frac{1}{2}}\}$



(a) Weight quantization to a hardware-friendly format Ω.



(b) Quantized weights as a function of the original weights



(c) Quantized gradients with respect to step size

Fig. 8: Comparison of INT4 and an shift-and-add friendly values quantized using the step-driven quantizer. The scale and zero-point have been set to 1 and 0 respectively for clarity.

B. From fake to hardware-aware quantization

Fake quantization leaves tensor storage in $\mathbb{R}$, but deployment requires expressing inference purely with Ω arithmetic. Consider a layer l computing

$$A^l = \text{ReLU}\left(W^l * A^{l-1} + B^l\right) \tag{10}$$

where $*$ represents a convolution or a fully-connected linear operation. The fake-quantized operation of the linear part is:

$$\widehat{X^l} = \widehat{W^l} * \widehat{A^{l-1}}. \tag{11}$$

Expanding the fake-quantized terms using (9) gives:

$$\widehat{X^l} = (s_w^l \cdot \overline{W^l} + z_w^l) * (s_a^{l-1} \cdot \overline{A^{l-1}} + z_a^{l-1}). \tag{12}$$

A common design choice is to use symmetric quantization for the weights (i.e., $z_w^l = 0$) as they usually have a distribution centered around zero [21]. Thanks to the linearity of $*$, it allows for the combination of the weight- and activation-related scaling factors into one:

$$\widehat{X^l} = (s_w^l \cdot \overline{W^l}) * (s_a^{l-1} \cdot \overline{A^{l-1}} + z_a^{l-1}) \tag{13}$$

$$= (s_w^l s_a^{l-1})(\overline{W^l} * \overline{A^{l-1}}) + (s_w^l \cdot \overline{W^l}) * (z_a^{l-1}\mathbb{1}). \tag{14}$$

This gives a scaled convolution of quantized terms on the left, and on the right a constant term. The latter may be added to the original bias tensor B^l (unquantized) to get a new bias:

$$C^l = B^l + (\overline{W^l} s_w^l) * (z_a^{l-1}\mathbb{1}). \tag{15}$$

Quantization of the original convolution + bias is therefore:

$$\widehat{Y^l} = (s_w^l s_a^{l-1})(\overline{W^l} * \overline{A^{l-1}}) + \widehat{C^l} \tag{16}$$

$$= (s_w^l s_a^{l-1})(\overline{W^l} * \overline{A^{l-1}}) + (s_c^l \cdot \overline{C^l} + z_c^l). \tag{17}$$

To obtain a pure machine convolution + bias, a second design choice is to use symmetric quantization for the new bias C ($z_c^l = 0$) while fixing $s_c^l = s_w^l s_a^{l-1}$. This yields:

$$\overline{Y^l} = \overline{W^l} * \overline{A^{l-1}} + \overline{C^l}, \tag{18}$$

with:

$$\widehat{Y^l} = (s_w^l s_a^{l-1})\overline{Y^l}. \tag{19}$$

Assuming a ReLU-like activation (positive-scale invariant), the next activation in Ω is:

$$\overline{A^l} = Q(\widehat{Y^l}) = \text{round}\left(\frac{1}{s_a^l} \cdot \text{ReLU}\left(\widehat{Y^l}\right) - \frac{z_a^l}{s_a^l}\right) \tag{20}$$

$$= \text{round}\left(m^l \cdot \text{ReLU}\left(\overline{Y^l}\right) - \frac{z_a^l}{s_a^l}\right) \tag{21}$$

where

$$m^l = \frac{s_w^l s_a^{l-1}}{s_a^l}. \tag{22}$$

and clipping is implicit for clarity.

Let us now review the two main strategies that can be used to remove $\mathbb{R}$ arithmetic from (21).

C. Deployment with Look-Up Tables (LUTs)

This technique, used by backends like FINN [24], builds hardware implementing the scaled and shifted ReLU described in (21). This constrains the deployment as this hardware will be specialized to a layer (or channel). It is for instance appealing if the network is small enough to be fully unrolled in hardware.

The idea is to tabulate the possible activation outputs $\overline{A^l}$ as a function of the accumulator result $\overline{Y^l}$ [25]. For a b-bit encoder there are 2^b output levels $\{q_i\}$; thresholds $\{\tau_i\}$ satisfy:

$$q_i = \text{round}(m^l \tau_i + z_a^l). \tag{23}$$

or

$$\tau_i = \frac{q_i - z_a^l}{m^l}. \tag{24}$$

Inference must perform a search over intervals:

$$\overline{A^l} = \begin{cases} q_0 & \overline{Y^l} < \tau_0, \\ q_i & \tau_{i-1} \leq \overline{Y^l} < \tau_i,\ i \in [1, 2^b - 2], \\ q_{2^b-1} & \overline{Y^l} \geq \tau_{2^b-1}. \end{cases} \tag{25}$$

The τ_i are actually rounded to the same format used in $\overline{Y^l}$ in the deployed network. With proper care, this method is arithmetically equivalent to the fake-quantized training graph, preserving the accuracy.

Note that there is an alternative method where some generic hardware implements (25). Its overhead is to replace, in the deployed network, the 3 parameters in (21) with the $|\Omega|$ parameters τ_i. For very low-bit quantization the overhead is minimal.

D. Deployment with a scaling multiplier

Another approach is to have, in the deployed network, a multiplier by the fused scaling factor m^l in (21). It must be approximated to remove $\mathbb{R}$ arithmetic: fixed-point approximation is a common choice [9], among others (power-of-two, log, ...). In fixed-point, m^l is a scaled n-bit integer: $m^l \approx m^l_{\mathbb{Z}} 2^{-n}$ with $n \in \mathbb{N}$. Then

$$\overline{A^l} \approx \left\lfloor m^l_{\mathbb{Z}} \cdot \text{ReLU}\left(\overline{Y^l}\right) \times 2^{-n} \right\rfloor - \frac{z^l_a}{s^l_a} \tag{26}$$

$$\approx \left\lfloor m^l_{\mathbb{Z}} \cdot \text{ReLU}\left(\overline{W^l} * \overline{A^{l-1}} + \overline{C^l}\right) \times 2^{-n} \right\rfloor - \left\lfloor \frac{z^l_a}{s^l_a} \right\rfloor \tag{27}$$

where the floor operation is actually a constant right shift. Here only Ω arithmetic remains.

Table II studies how the accuracy of the network depends on the bit-size n used for the scaling factor m^l.

E. Batchnorm folding

A batch normalization layer [26] is generally placed between the linear operation and the activation function during training:

$$A^l = \text{ReLU}(\text{BN}(W^l * A^{l-1} + B^l)) \tag{28}$$

$$= \text{ReLU}\Big(\gamma^l \cdot \frac{W^l * A^{l-1} + B^l - \mu^l}{\sqrt{(\sigma^l)^2 + \epsilon}} + \beta^l\Big) \tag{29}$$

where γ^l and β^l are learnable parameters, μ^l and $(\sigma^l)^2$ are the batch statistics (mean and variance respectively) and ϵ is a small constant added for numerical stability. This is problematic for quantization, as the batch normalization parameters are in $\mathbb{R}$ and cannot be directly mapped to Ω. A common practice is to fold the batch normalization into the weights and biases [9], [12], [27]:

$$A^l = \text{ReLU}(W^l_{\text{fold}} * A^{l-1} + B^l_{\text{fold}}) \tag{30}$$

$$W^l_{\text{fold}} = \frac{\gamma^l}{\sqrt{(\sigma^l)^2 + \epsilon}} \cdot W^l \tag{31}$$

$$B^l_{\text{fold}} = \frac{\gamma^l}{\sqrt{(\sigma^l)^2 + \epsilon}} \cdot (B^l - \mu^l) + \beta^l \tag{32}$$

Note that:

$$\frac{\gamma^l}{\sqrt{(\sigma^l)^2 + \epsilon}} \cdot \overline{W^l} \neq \left(W^l_{\text{fold}}\right)_\Omega \tag{33}$$

because of the quantization operation. The same applies to the biases. Instead of normalizing the quantized weights, the normalized weights must be quantized: this is not equivalent and requires the network to train with the folded weights and biases. This is an expensive operation, as it requires two linear operations to be computed during training in order to get the batch statistics [9], [12]. Note that the batch statistics σ^l (variance) and μ^l (mean) are replaced after a certain number of steps by running averages [12], which are the constants used when deploying the model.

F. Custom quantized layers

To allow the user to control the scaling transformations and accomodate the different lowering techniques, HATorch implements a dedicated quantization layer merged with the batch normalization and activation function, as shown in Figure 5. Thus, the scaling factors are fused together during training, this allows the user to train the network with the scaling approximation of its choice (fixed-point scaling with b bits, rounding to a power-of-two, LUTs, ...), reflecting the actual deployed behavior. A practical side-effect is that the convolution/linear operations are now computed directly in the target number format (Ω) at inference. This can ever be emulated by FP32 operations (which are exact if $\Omega \subset \mathbb{Z}$) or the user can provide a custom kernel computing directly in Ω domain [28].

G. Quantization of residual connections

Networks with residual connections like ResNet [29] and MobileNetv2 [30] use an addition operation between the layer convolution output and the previous layer's activation. When quantized, this addition requires both operands to use the same scale [9]. HATorch therefore rescales the skip-branch activation to match the scale of the convolutional branch before the addition. In order to preserve inference in the Ω domain, this rescaling uses the user-specified approximation method (typically fixed-point scaling).

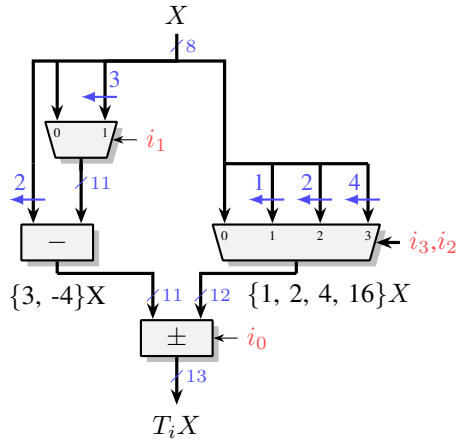
V. Experiments

Two simple experiments are conducted to validate the proposed approach using HATorch. The first one, in V-A, demonstrates that the step-driven quantizer can be used to train CNN models with hardware-friendly quantization levels. The second one, in V-B, shows the importance of matching training and deployment arithmetic to avoid accuracy degradation (the hardware gap).

A. Training with hardware-friendly quantization levels

To demonstrate HATorch's step-driven quantizer flexibility on arbitrary number formats, 3 networks are trained on 3 formats at very low bit widths. The networks are a VGG11 [31] with batch normalization, and ResNet-20 and ResNet-56 [29] (10 million and 300/900 thousand parameters respectively), trained on CIFAR-100. The formats are small integers, and two non uniform formats: the Logarithmic Number System (LNS) in base 2, and the format shown in Figure 8. These formats are chosen as attractive for custom hardware: LNS converts multiplications to additions ($\pm 2^x$ representation), and the Shift-and-Add [17] replaces multipliers entirely with the architecture of Figure 9.

All experiments are performed on a single NVIDIA RTX 4080 GPU with 16GB of GDDR6X. The first and last layers are quantized to 8-bit integers if the rest of the model is quantized to 4 bits or higher, and to 4 bits otherwise. Models were trained with a batch size of 128 using stochastic gradient descent with momentum 0.9 and weight decay 5×10^{-4}. The



i_3	i_2	i_1	i_0	T_i
0	0	0	0	4
0	0	0	1	2
0	0	1	0	-3
0	0	1	1	-5
0	1	0	0	5
0	1	0	1	1
0	1	1	0	-2
0	1	1	1	-6
1	0	0	0	7
1	0	0	1	-1
1	0	1	0	0
1	0	1	1	-8
1	1	0	0	19
1	1	0	1	-13
1	1	1	0	12
1	1	1	1	-20

$T_i \in$ {-20, -13, -8, -6, -5, -3, -2, -1, 0, 1, 2, 4, 5, 7, 12, 19}

Fig. 9: Example 2-adder RSCM for an 8-bit input, with a target set with 6-bit dynamic encoded in 4 bits [15].

first and last layers were quantized to 8 bits. Weights employed symmetric per-channel quantization, while activations used asymmetric per-tensor quantization. Data augmentation consisted of random 32×32 crops on 4 pixels padded images and horizontal flips. Batch-normalization statistics were frozen starting epoch 5. VGG11 training started from pretrained FP32 weights from our own training and used a cosine annealing learning rate schedule [32] over 100 epochs, with a starting learning rate of 5×10^{-3}. Calibration was performed over 1 epoch at a learning rate of 1×10^{-4}. For ResNets, training started from the weights of a pretrained model [33] and used a multi-stage learning rate schedule. A calibration phase of 2 epochs was performed with a learning rate of 1×10^{-4}, except for INT3 (1×10^{-5}), followed by 40 epochs with a cosine annealing schedule [32], decaying from 1×10^{-3} (or 5×10^{-3} for INT3) to 1×10^{-5}, and finally 10 epochs of fine-tuning with a linear decay to 1×10^{-6}.

Table I shows that the step-driven quantizer alows the Shift-and-Add format to achieve comparable accuracy to standard integer quantization at equivalent bit-widths, validating HATorch's flexibility in supporting custom number formats without accuracy penalty. The LNS formats used in this table are from [14], and are used as an example where the q_i are real numbers with a non-uniform distribution. Note that the LNS format was simulated using floating-point arithmetic as a specialized kernel is yet to be implemented.

TABLE I: HATorch 's accuracy on CIFAR-100 for standard integer quantization and custom hardware-friendly formats (S&A for Shift-and-Add). Best of 3 runs reported.

Model	FP32	INT4	INT3	S&A (W4A4)	LNS (W4A3)	LNS (W3A2)
VGG11	70.46	70.30	69.03	70.23	70.00	66.85
ResNet20	70.37	69.61	68.09	69.61	66.86	61.36
ResNet56	75.09	73.45	72.69	73.66	71.85	63.86

TABLE II: Comparison of accuracy when the FP32 scaling factors are rounded to FxP at deployment, versus training the model directly with fixed-point scales, or fine-tuning it on a few more epochs with fixed-point scales. Average best of 3 runs reported.

Model	Method	16-bit FxP	8-bit FxP	7-bit FxP	6-bit FxP
VGG11 W3A3 (69.03)	rounded	68.82	68.00	66.34	19.64
	trained	68.77	68.67	68.55	68.79
	fine-tuned	68.96	68.64	68.34	68.24
ResNet20 W4A4 (69.61)	rounded	69.42	1.62	1.09	1.01
	trained	69.50	68.62	67.47	64.48
	fine-tuned	69.22	68.15	66.27	62.93
ResNet56 W4A4 (73.45)	rounded	72.84	32.45	17.28	0.88
	trained	73.40	73.01	72.43	56.83
	fine-tuned	73.43	72.79	72.37	43.67

B. Reducing the hardware gap

To demonstrate the necessity of matching training and deployment arithmetic, Table II considers the same three models as in V-A with the same training recipe as in Table I. As it makes little sense to use 32-bit scaling factors m^l in the deployment of such low-precision networks, this table compares three alternatives for the quantization of these scaling factors in deployed networks. The first technique is to simply round them to fixed-point (16, 8, 7 and 6 bits), after training. The second technique is to train the model with fixed-point scaling factors using HATorch. The third technique is to first train the model with floating-point scaling factors, then fine-tune it for 15 additional epochs with fixed-point scaling factors. Table II reports the accuracy degradation between the trained and deployed models.

As expected, the first technique leads to accuracy degradation that increases with lower precision scaling factors, from a negligible 0.21% loss at 16-bit to catastrophic non-convergence at 8, 7 or 6-bit, depending on the model. Training with fixed-point m^l allowed to recover the catastrophic loss in every cases and significantly reduce the accuracy degradation at 6, 7 and 8-bit, but did not bring significant improvement at 16-bit. Fine-tuning is then explored, it is similar to how QAT starts from a pre-trained floating-point model. Fine-tuning the FxP16 converted model slightly outperforms both the converted and trained models, without bringing improvement at other bitwidths. This is likely because the trade-off between scale quantization noise and fixed-point conversion is less favorable at high precision.

These preliminary results validate that matching training and deployment arithmetic is important to close the hardware gap, here due to low-precision scales.

VI. Conclusion & Perspectives

This work presented HATorch, a novel framework for hardware-aware training of quantized CNNs, supporting arbitrary number formats and scaling methods. The framework addresses the hardware gap prevalent in simulated QAT approaches by transparently aligning training and deployment

arithmetic, ensuring reliable deployment on custom accelerators. Experimental results demonstrate HATorch's effectiveness in bridging this gap with fixed-point scales while maintaining flexibility for custom formats like LNS and Shift-and-Add without compromising accuracy. Future developments includes exploring additional number formats and interfacing with actual deployment frameworks through QONNX. Integrating more computation kernels for direct Ω arithmetic will improve efficiency in handling quantized models. This will enable testing on larger-scale datasets and architectures to validate the framework's broader applicability and scalability.

Acknowledgement

This work was partially supported by the PEPR IA HOLIGRAIL project of the Agence Nationale de la Recherche, ANR-23-PEIA-0010.

REFERENCES

[1] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," in *Low-power computer vision*. Chapman and Hall/CRC, 2022, pp. 291–326.

[2] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *1606.06160*, 2016.

[3] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized clipping activation for quantized neural networks," *arXiv:1805.06085*, 2018.

[4] S. Esser, J. McKinstry, D. Bablani, R. Appuswamy, and D. Modha, "Learned step size quantization," in *International Conference on Learning Representations*, 2020.

[5] Q. Jin, L. Yang, and Z. Liao, "Towards efficient training for neural network quantization," *arXiv:1912.10207*, 2019.

[6] M. Nagel, R. A. Amjad, M. Van Baalen, C. Louizos, and T. Blankevoort, "Up or down? adaptive rounding for post-training quantization," in *Proceedings of the 37th International Conference on Machine Learning*, ser. ICML'20, 2020.

[7] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: a system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, 2016, p. 265–283.

[9] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2704–2713.

[10] Y. Li, M. Shen, J. Ma, Y. Ren, M. Zhao, Q. Zhang, R. Gong, F. Yu, and J. Yan, "MQBench: Towards reproducible and deployable model quantization benchmark," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, J. Vanschoren and S. Yeung, Eds., vol. 1, 2021.

[11] G. Armeniakos, G. Zervakis, D. Soudris, and J. Henkel, "Hardware approximate techniques for deep neural network accelerators: A survey," *ACM Computing Surveys*, vol. 55, no. 4, pp. 1–36, 2022.

[12] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *arXiv:1806.08342*, 2018.

[13] Y. Li, X. Dong, and W. Wang, "Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks," in *International Conference on Learning Representations*, 2020.

[14] M. Christ, F. De Dinechin, and F. Pétrot, "Low-precision logarithmic arithmetic for neural network accelerators," in *IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2022, pp. 72–79.

[15] B. Barbe, X. Peng, A. Volkova, and F. de Dinechin, "Towards optimal reconfigurable constant multipliers," in *28th Euromicro Conference on Digital System Design (DSD)*, Italy, Sep. 2025.

[16] R. Garcia, L. Pradels, S.-I. Filip, and O. Sentieys, "Hardware-aware training for multiplierless convolutional neural networks," in *IEEE 32nd Symposium on Computer Arithmetic (ARITH)*, 2025, pp. 9–16.

[17] J. Faraone, M. Kumm, M. Hardieck, P. Zipf, X. Liu, D. Boland, and P. H. W. Leong, "AddNet: Deep Neural Networks Using FPGA-Optimized Multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 115–128, 2020.

[18] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," *Advances in neural information processing systems*, vol. 28, 2015.

[19] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," *Advances in neural information processing systems*, vol. 29, 2016.

[20] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv:1308.3432*, 2013.

[21] Y. Bhalgat, J. Lee, M. Nagel, T. Blankevoort, and N. Kwak, "LSQ+: Improving low-bit quantization through learnable offsets and better initialization," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, 2020, pp. 696–697.

[22] S. Gongyo, J. Liang, M. Ambai, R. Kawakami, and I. Sato, "Learning non-uniform step sizes for neural network quantization," in *Proceedings of the Asian Conference on Computer Vision*, 2024, pp. 4385–4402.

[23] G. Franco, A. Pappalardo, and N. J. Fraser, *Xilinx/brevitas*, 2025. [Online]. Available: https://doi.org/10.5281/zenodo.3333552

[24] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*, 2017, pp. 65–74.

[25] Y. Umuroglu and M. Jahre, "Streamlined deployment for quantized neural networks," *arXiv:1709.04060*, 2017.

[26] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. pmlr, 2015, pp. 448–456.

[27] F. de Dinechin and M. Kumm, *Application-Specific Arithmetic: Computing Just Right for the Reconfigurable Computer and the Dark Silicon Era*. Springer, 2024.

[28] S. B. Ali, S.-I. Filip, O. Sentieys, and G. Lemieux, "MPTorch-FPGA: a Custom Mixed-Precision Framework for FPGA-based DNN Training," in *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2025, pp. 1–7.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[30] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[31] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556*, 2014.

[32] I. Loshchilov and F. Hutter, "Stochastic gradient descent with warm restarts," in *Proceedings of the 5th Int. Conf. Learning Representations*, 2016, pp. 1–16.

[33] O. Sémery, "PyTorchCV: Computer vision models on pytorch," https://github.com/osmr/pytorchcv, 2024, accessed: 2025-04-14.