




# Single-Path Precision: Differentiable Bit-Width Numeric-Format Learning for FPGA-Efficient Neural Networks

Emanuel Trabes<sup>1,2</sup>, Kawthar Dellel<sup>1,3</sup>, Carlos Valderrama<sup>1</sup>, and Abdelmalik Taleb-Ahmed<sup>4</sup>

<sup>1</sup> Service d'électronique et de Microélectronique, University of Mons, Belgium

<sup>2</sup> Department of Electronics, Universidad Nacional de San Luis, Argentina

<sup>3</sup> Laboratory of Electronics and Microelectronics, Faculty of Science of Monastir, University of Monastir, Tunisia

<sup>4</sup> IEM UMR CNRS, Hauts-de-France Polytechnic University, France

`emanuel.trabes@umons.ac.be`

`kaouther.dellel@student.umons.ac.be`

`carlosalberto.valderramasukuyama@umons.ac.be`

`abdelmalik.taleb-ahmed@uphf.fr`

**Abstract.** Quantization in neural networks typically requires early commitment to a fixed numerical format—either fixed-point or floating-point—and a limited set of word sizes supported by the target hardware. This rigid choice often compromises either model accuracy or hardware efficiency. While FPGAs offer the flexibility to tailor precision per layer, hardware-aware differentiable architecture search (HW-DNAS) methods that explore this space suffer from high computational overhead due to layer duplication.

We propose a single-path, quantization-aware training framework integrated with a novel flexible numerical format. This approach allows each layer to learn, during training, whether to use fixed- or floating-point arithmetic and to determine the optimal bit-widths for exponent and mantissa, without duplicating layers. A multi-objective optimization strategy balances accuracy with hardware efficiency throughout training. Our method achieves up to 10x reductions in average bit-width on MNIST, CIFAR-10, and CIFAR-100, with 1% loss in accuracy. By deferring numerical format selection to training time and eliminating HW-DNAS-style overhead, we enable resource-efficient, precision-adaptive deep learning suitable for deployment on reconfigurable hardware.

**Keywords:** QAT · Bit-Width Learning · Low-Precision AI

## 1 Introduction

Deep neural networks have powered major breakthroughs in computer vision, speech recognition, and natural language processing. However, each gain in accuracy has come at the cost of increasingly large models and rising energy consumption. This growing computational demand threatens to concentrate progress in

the hands of a few organizations with access to industrial-scale hardware, sidelining researchers and businesses with more modest resources [17]. Thus, developing methods to reduce energy consumption and hardware cost has become critical.

Quantization offers a promising solution: reducing word sizes can significantly lower memory traffic and increase arithmetic throughput, all without requiring a complete architectural redesign [7, 9]. Yet, mainstream accelerators typically support only a limited set of hard-wired formats—such as 32- and 16-bit floating point and 8-bit integer—forcing developers to commit to specific precisions and arithmetic types early in the training process [11, 13]. Choosing the wrong format in advance can lead to either suboptimal accuracy or inefficient hardware usage. This challenge is compounded by the fact that optimal precision often varies from layer to layer.

ASIC-based AI accelerators attempt to address this by supporting a wider range of formats, but each additional datapath increases area and power consumption—even if the final model does not utilize it. This creates a difficult trade-off between flexibility and utilization. Field-programmable gate arrays (FPGAs), with their reconfigurable fabric, offer a compelling alternative. They can implement custom, layer-specific word sizes for both fixed- and floating-point arithmetic, enabling efficient, tailored inference.

However, most existing precision-selection methods suffer from significant drawbacks:

1. Pre-training searches that brute-force the combinatorial space of bit-widths [1, 19];
2. Post-training quantization (PTQ), which adjusts a completed model and often degrades accuracy [14]; and
3. Hardware-aware differentiable architecture search (HW-DAS), which incorporates precision into gradient descent by duplicating each layer for every candidate format, drastically increasing the computational burden.

The field also continues to grapple with the trade-off between fixed- and floating-point arithmetic. Integer formats are energy-efficient but suffer from limited dynamic range, while floating-point formats offer greater flexibility at higher computational cost. Studies show that some layers perform well with 8-bit integers [12], whereas others require floating-point representations to maintain accuracy [6]. Choosing this trade-off in advance is therefore doubly limiting, as it impacts both performance and model precision.

In this work, we propose a novel, single-path, fully differentiable quantization-aware training (QAT) framework that leverages a flexible numerical format. Our format comprises four fields: sign, exponent, mantissa, and scale. By setting the scale to zero, the format behaves as a floating-point representation; by setting the exponent to zero, it behaves as fixed-point. Crucially, the model learns during training whether to zero out the exponent or the scale, thus determining the appropriate arithmetic type on a per-layer basis. In addition, the bit-widths of the exponent and mantissa, along with the scale parameter, are learned dynamically—without duplicating layers, as required by traditional DNAS methods—thereby reducing computational overhead during training.

On MNIST, CIFAR-10, and CIFAR-100, our method reduces average bit-widths by up to  $10\times$ , all while maintaining accuracy within 1% of the baseline. By deferring numerical format selection to training time—without incurring the cost of HW-DAS—we advance practical, precision-adaptive deep learning for energy-constrained and resource-limited environments.

Our key contributions are:

1. A flexible numerical format capable of representing both floating- and fixed-point arithmetic;
2. A differentiable QAT framework that learns exponent/mantissa bit-widths and scale parameters without requiring layer duplication.

## 2 Proposed Approach

### 2.1 Overview

Quantization-aware training (QAT) approximates low-precision arithmetic by quantizing weights and activations during the forward pass while computing gradients in full precision during the backward pass. This enables the model to adapt to lower precision throughout training.

Our approach introduces a novel *bit penalty term* in the loss function, penalizing higher bit-widths to incentivize lower precision during quantization. By tuning this penalty, we balance memory efficiency and model performance without sacrificing accuracy. This method allows for more flexible and controlled experimentation, where the *bit penalty* acts as a tunable hyperparameter that influences the model’s *bit width’s precision* and *accuracy*. Through these experiments, we aim to determine the optimal balance between *memory* efficiency (bit-width reduction) and *accuracy* retention, which is the key challenge in deploying DNNs in real-world, resource-constrained environments.

### 2.2 Quantization Formulation

Let  $T$  denote a full-precision tensor over a field  $F$ . To be able to compute over  $F$  in a processor,  $F \notin R$ , but it is an approximation of a field using some numerical representation that the underlying hardware supports. The most widely used numerical format is the floating point  $FP = \{s, m, e\}$ , where  $s \in \{0, 1\}$  represents the sign,  $m \in \{0, \dots, 2^{b_m} - 1\}$  represents the mantissa and  $e \in \{0, \dots, 2^{b_e} - 1\}$  for the exponent. Both  $b_m$   $b_e$  represent the bit-width of the mantissa and the exponent, respectively. Common floating point representations use  $b_m = 23$  and  $b_e = 8$ , normally called single precision floating point, or  $b_m = 10$   $b_e = 5$ , know as half precision.

The underlying hardware can directly operate over  $FP$ , but to provide a connection between  $FP$  and  $F \in R$  one can use the following expressions:

$$F = (-1)^s \cdot (1, m) \cdot 2^{e-2^{(b_e-1)}} \quad (1)$$

where  $(-1)^s$  or  $(s \cdot 2 - 1)$  set the range of the sign into  $-1, 1$ ,  $(1.0 + m \cdot 2^{-m_b})$  adds the leading 1.0 into  $m$ ,  $2^{-m_b}$  sets the decimal point in the correct place.  $e - 2^{(b_e-1)}$  sets the range of the exponent to allow lower than one values.

Other numerical representations, such as the fixed point, are also being used successfully in deep learning [3]. Fixed point,  $FXP = m, d$  where  $m \in \{-2^{b_m-1}, \dots, 2^{b_m-1}\}$  represents the value of the integer, and  $d$  is a scaling factor representing the placement of the decimal place. The conversion to  $R$  can be expressed as follows:

$$F = m \cdot 2^d \quad (2)$$

Unlike  $FP$ ,  $FXP$  does not use a variable exponent  $e$  to set the decimal place. Instead, it relies on a fixed parameter  $d$ , which is determined during implementation.

Our goal is to learn the optimal numerical format—including bit-width and decimal point placement—during training. We define our numeric format  $\tilde{F}$  to be composed of 4 values:  $s \in \{0, 1\}$  to represent the sign,  $m \in \{0, \dots, 2^{b_m} - 1\}$  to represent the mantissa,  $e \in \{0, \dots, 2^{b_e} - 1\}$  to represent the exponent and  $d \in Z$  to represent a scale factor. To get the corresponding value  $F \in R$ , we follow:

$$\tilde{F} = (-1)^s \cdot (1, m) \cdot 2^{e-2^{(b_e-1)}+d} \quad (3)$$

This representation enables learning both floating-point and fixed-point formats. If we (i.e. the optimizer) set  $b_e = 0$ , we effectively have a fixed point with scale factor  $2^{b_s}$ . If  $b_e > 0$ , then  $d$  place the role as a scaler, setting the mid range of  $\tilde{F}$  so a value different than  $2^{(b_e-1)}$  to better represent the actual range of the variable.

### 2.3 Parametrization

Since  $b_e$  and  $b_m$  are strictly positive, a parameterization that naturally restricts values to the correct interval is preferable, avoiding the need for clamping during optimization.

A common approach is to use a logarithmic parameterization. Define  $p_e = \ln(b_e)$  and  $p_m = \ln(b_m)$ . The optimizer searches for optimal  $p_e$  and  $p_m$ , and the original variables are recovered by reversing the parameterization:  $b_e = e^{p_e}$  and  $b_m = e^{p_m}$ .

### 2.4 Gradient computation

For a given tensor defined in our representation, we need to compute:

$$\frac{\partial \tilde{T}(\tilde{F})}{\partial e, m, p_e, p_m, d} \quad (4)$$

For  $e, m$

$$\frac{\partial \tilde{T}(\tilde{F})}{\partial e, m} = \frac{\partial \tilde{T}(\tilde{F})}{\partial \tilde{F}} \frac{\partial \tilde{F}}{\partial e, m} \quad (5)$$

$\frac{\partial \tilde{T}(\tilde{F})}{\partial e, m}$  is computed by back-propagation by the system being used. For given  $p_e, p_m$   $\tilde{F}$  is the common floating point representation  $\tilde{F} \approx F$ , then  $\frac{\partial \tilde{F}}{\partial e, m} = 1$   
For  $p_e$

$$\frac{\partial \tilde{T}(\tilde{F})}{\partial p_e} = \frac{\partial \tilde{T}(\tilde{F})}{\partial \tilde{F}} \frac{\partial \tilde{F}}{\partial b_e} \frac{\partial b_e}{\partial p_e} \quad (6)$$

We have  $\frac{\partial b_e}{\partial p_e} = e^{p_e} = b_e$ .

Because  $\frac{\partial \tilde{F}}{\partial b_e}$  is highly nonlinear, we cannot compute the derivative analytically. Instead, we introduce a discrete approximation in the backward pass, using the five point midpoint approximation:

$$\begin{aligned} \frac{\partial \tilde{F}}{\partial b_e} \approx & (-\tilde{F}(e, m, b_e + 2, b_m, d) + 8 \cdot \tilde{F}(e, m, b_e + 1, b_m, d) \\ & - 8 \cdot \tilde{F}(e, m, b_e - 1, b_m, d) + \tilde{F}(e, m, b_e - 2, b_m, d))/12 \end{aligned} \quad (7)$$

The same is done for  $\frac{\partial \tilde{F}}{\partial b_m}$  and  $\frac{\partial \tilde{F}}{\partial d}$

## 2.5 Loss Function

Our total loss function,  $\mathcal{L}_{\text{total}}$ , is defined as:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{task}}(y, y_{\text{pred}}) + \lambda \sum_l^L (b_e(l) + b_m(l)), \quad (8)$$

where  $\mathcal{L}_{\text{task}}$  is the task-specific loss (e.g., cross-entropy). The second term in the equation enforces a penalty on larger bit-widths, for each layer  $l$  and  $\lambda$  is a hyperparameter that balances task accuracy and bit-width minimization.

## 2.6 Pytorch implementation

We built our system using Pytorch. We defined a function `fake_float`, which takes as input a tensor  $x$ , as well as the amount of bits  $e_b, e_m$  and the scale parameter  $d$ . This function first find  $e$  and  $m$  by:

$$\begin{aligned} e &= \text{floor}(\log_2(|\frac{F}{2^d}|)) \\ m &= \frac{|\frac{F}{2^d}|}{2^e} \end{aligned} \quad (9)$$

and then find the quantized  $e$  and  $m$  by

$$\begin{aligned} e_q &= \min(\max(e + 2^{b_e-1} - 1 - d, 0), 2^{b_e} - 1) \\ m_q &= \frac{\text{floor}(m \cdot 2^{b_m})}{2^{b_m}} - 1 \end{aligned} \tag{10}$$

Then the function reconstructs the tensor using equation 3. We built a *FakeFloat* class, derived from *torch.autograd.Function*, which takes as input a tensor,  $p_e$ ,  $p_m$  and  $s$ . In the forward pass it computes  $b_e$  and  $b_m$  from  $p_e$  and  $p_m$  respectively, and applies *fake\_float* to the input tensor. In the backward pass, it uses a straight-trough approximation for the gradient with respect to  $e$  and  $m$ , and uses equation 7 for the gradient with respect to  $b_e$ ,  $b_m$  and  $s$ . Finally, we built a class *QuantWrapper* derived from *torch.Module*, which takes a layer as input, defines the learnable quantization parameters  $p_e$ ,  $p_m$  and  $s$ . In the forward pass, *QuantWrapper* applies *FakeFloat* to the input tensor  $x$  the weights  $w$  and bias  $b$  of the layer.

### 3 Results

To use as baseline for comparison, we used as reference [18]. In this work, the authors employ a model inspired by VGG-net [16], training it with various carefully selected quantization configurations and reporting both precision and resource utilization.

The model contains a basic block consisting of 2D convolution, batch normalization, 2D convolution, batch normalization and a maxpooling layer. This basic block is repeated 3 times, skipping the last maxpooling. The output channels for the convolutions are 64, 128 and 256 for each basic block. This is followed by 2 sets of linear layer with 512 output channels followed by 1D batch normalization, and a final linear layer.

The authors report, for the CIFAR10 dataset, precision of 89.03% for a quantization of 2 bits weights and 2 bits activations, across all layers, 87.80% accuracy for 1 bits weights and 2 bits activations, and 84.22% accuracy for 1 bit weights and 1 bit activations.

#### 3.1 Training

Again, we followed the same training settings as in [18]. We used the adam optimizer with a learning rate of 0.01, a loss function consisting of the cross entropy for classification and sum of bit-widths for the bit loss. The lambda parameter of 8 was selected manually depending on the dataset used. In all experiments, we trained for 100 epochs using a batch size of 1024.

For the experiments, we set the initial bit configuration as follows: 5 bits for  $b_e$  10 bits for  $b_m$  and 0 for  $d$ . We first run a baseline training where we don't update any of the bit-widths. After the initial training, the model is retrained while allowing the bit-width for input quantization and weights to be updated. However, the initial quantization for the bias remains unchanged.

### 3.2 MNIST dataset

For the MNIST dataset, we set the lambda parameter of 8 was set to 0.5. The baseline model, without optimizing for the bit-widths, reached a 99.58% accuracy. While learning the bit-widths, the best accuracy reached was of 99.52%, with an average bit-width per layer of 1.44 bits. The corresponding bit widths were as following

**Table 1:** Bit-widths per layer for the MNIST dataset.

Layer input	$b_e$ $b_m$ $d$	weight $b_e$ $b_m$ , $d$
conv1	(1, 1, -2)	(0, 0, -1)
conv2	(1, 0, 1)	(0, 0, -1)
conv3	(0, 0, 3)	(0, 0, 0)
conv4	(0, 0, 2)	(0, 0, -1)
conv5	(0, 0, 2)	(0, 0, -1)
conv6	(0, 0, 2)	(0, 0, 0)
fc1	(0, 0, 2)	(0, 0, 0)
fc2	(0, 0, 2)	(0, 0, 0)
fc3	(0, 0, 0)	(0, 0, 0)

**Table 2:** Performance comparison of models on MNIST.

Model Type	Accuracy (%)	Average Loss	Bit Penalty
Baseline Model	99.58%	1.6340	16.0
Lower Bit Model	99.52%	0.2560	1.44

The results confirm that even aggressive bit-width reductions (down to  $\sim 1.44$  bits) can maintain near-perfect classification performance. This is due to the relatively low complexity of MNIST, where lower precision representations still capture enough discriminative information.

Our method found that the optimal exponent bit-width, for all the layer except the first two, are 0. Furthermore, our method found that the optimal bit-width for most of the mantissa are also 0, effectively setting a binarized neural network, as the only bit utilized is the one coming from the sign. This is no surprising, as the MNIST has been successfully binarized in works like [15] [10] [2]

### 3.3 CIFAR10 dataset

For the CIFAR10 dataset, the lambda parameter was also set to 0.5. The baseline model reached a 92.36% accuracy. The bit optimized model reached an accuracy of 91.97%, with an average bit-width per layer of 2.89 bits. The corresponding bit widths were as following:

**Table 3:** Bit-widths per layer for the CIFAR10 dataset.

Layer	input	$b_e$	$b_m$	$d$	weight	$b_e$	$b_m$	$d$
conv 1		2	3	-2		2	1	-1
conv 2		2	2	1		0	2	0
conv 3		1	2	1		1	1	1
conv 4		0	2	2		2	0	-1
conv 5		0	2	2		1	1	-1
conv 6		0	2	2		0	1	-1
linear 1		0	2	2		0	0	-1
linear 2		0	1	1		0	0	-1
linear 3		0	0	0		0	0	-1

**Table 4:** Performance comparison of models CIFAR10.

Model Type	Accuracy (%)	Average Loss	Bit Penalty
Baseline Model	92.36%	2.0990	16.0
Lower bit Model	91.97%	0.6540	2.89

CIFAR-10, a more complex dataset than MNIST, presents a trade-off between bit-width reduction and classification performance. While extreme quantization (1.44 bits) preserved a good accuracy in MNIST, CIFAR-10 requires higher precision. The optimized 2.89-bit model achieves 91.97% accuracy with reduced computational overhead, demonstrating that deep networks can operate efficiently at lower precision.

It is interesting to note that the method distributed more bits to the first convolution layers. For the linear layers, the weights as well as the last input quantization only utilize the sign of the tensor. Some of the input bit-widths of the convolutional layers required  $b_e > 0$ , suggesting that this layers benefit from the exponential precision given by floating point representations.

In [18], the authors used 2 bits per layer, 2 bits for activations and 2 bits for weights, reaching an accuracy of 89%. Our method found an bit-distribution slightly high, using 2.89 per layer in average, but with a higher accuracy.

### 3.4 CIFAR100 dataset

For this dataset, the lambda parameter was set to 0.01. The baseline model for the CIFAR100 dataset reached a 67.50% accuracy. The bit optimized model reached an accuracy of 67.33%, with an average bit-width per layer of 4.01 bits. The corresponding bit widths are illustrated in TABLE 5.

CIFAR-100 is more challenging than CIFAR-10 and MNIST, containing 100 different classes, making classification significantly harder. Unlike CIFAR-10, where we could push the bit-width down to  $\sim 2$  bits with minimal accuracy loss, CIFAR-100 requires at least 4-bit precision to maintain strong performance. The 4-bit optimized model is the best trade-off, achieving nearly the same accuracy

**Table 5:** Bit-widths per layer for the CIFAR100 dataset.

Layer	input	$b_e$	$b_m$	$d$	weight	$b_e$	$b_m$	$d$
conv 1		1	5	-1		0	4	-1
conv 2		3	3	1		1	3	0
conv 3		2	3	0		0	3	0
conv 4		1	3	1		1	2	-1
conv 5		2	3	1		1	3	0
conv 6		1	3	1		1	2	-1
linear 1		1	3	1		0	3	0
linear 2		1	3	1		2	3	-2
linear 3		1	3	0		2	2	-2

**Table 6:** Performance comparison of models on CIFAR100.

Model Type	Accuracy (%)	Average Loss	Bit Penalty
Baseline Model	67.50%	4.2123	16.0
Lower Bit Model	67.33%	3.2471	4.0

as the full-precision model (67.33% vs. 67.50%) while reducing the memory and computational requirements by 4x.

### 3.5 Comparison with previous methods

**Table 7:** State-of-the-art mixed-precision quantization results on **ResNet-20 / CIFAR-10**. “Avg. bit-width” lists the *average* weight / activation precision reached by each method.

Method	Avg. bit-width	Top-1 Acc. (%)
LQ-Nets [20]	2 / 2	90.2
HAWQ-V1 [4]	3.89 / 4	92.2
SDQ [8]	1.93 / 32	92.1
AdaQAT [5]	3 / 4	92.2
Ours	3.9 / 3.9	91.3

As can be seen in table 7, our method achieves similar results as [5] and [8], with slightly less accuracy. Regarding training performance, [20] reports a  $2.3\times$  increase in training time, with respect to a baseline floating point implementation. Our method converged to an accuracy of 91.3% in 53 minutes for the baseline model, and the quantized model took an additional 135 minutes, giving an increase in training time of  $2.54\times$ .

### 3.6 FPGA implementation

We implemented the resulting CIFAR10 model using FINN [18]. Our method selected to use fixed-point in most of the layers. Nevertheless, a few layers were set by our method as best implemented using floating-point. As FINN does not support floating point layer, we utilized an equivalent bit-width fixed-point representation. We implemented both the baseline 16-bit model and the optimized model.

**Table 8:** FPGA results for the CIFAR10 model

<b>Resources</b>	Baseline	16-bits	Optimized
LUT	4800000	34800	
FF	221000	58400	
DSP	31	31	
BRAM	6915	1220	
FMAX	103.82	102.34	
Throughput	11.1	10.4	

As can be seen in 8, the resources used in the quantized model are considerably reduced. The resource savings are not linear with the bit-width reduction, probably because the synthesis tool is able to more efficiently use the DSP with larger bit-widths.

## 4 Discussion

The results across MNIST, CIFAR-10, and CIFAR-100 confirm the effectiveness of bit-width optimization in reducing computational complexity while maintaining competitive accuracy. Quantization significantly lowers memory consumption and computational overhead, with optimal precision varying by dataset. For MNIST, extreme quantization (1.44-bit) retained 99.52% accuracy, proving its feasibility for simpler datasets. In CIFAR-10, the 2.89-bit model achieved 91.97% accuracy, closely matching the full-precision model (92.36%). CIFAR-100, being more complex, required at least 4-bit precision for stability, achieving 67.33% accuracy (vs. 67.50% at full precision).

The results underscore the potential for FPGAs to break the constraints of fixed bit-width general-purpose hardware. This has critical implications for democratizing AI, as lower-power and cheaper FPGA-based solutions can be deployed in areas lacking large data centers or advanced GPU installations. By reducing energy consumption, such systems may also mitigate environmental concerns associated with large-scale AI model training.

## 5 Conclusion

In this paper, we presented a novel quantization-aware training framework that simultaneously learns weights and bit-widths—covering mantissa, exponent and scale configurations for fixed floating-point arithmetic. Through extensive experimentation on MNIST, CIFAR-10, and CIFAR-100, we demonstrated that models can operate at significantly reduced precision— as low as 1.44-bit for MNIST and 4-bit for CIFAR-100 without severe accuracy degradation. By introducing a differentiable, learnable quantization strategy, our work bridges the gap between model compression and hardware-aware optimization, paving the way for more scalable, energy-efficient, and high-performance deep learning models.



This project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska Curie grant agreement No 101034383

## References

1. Cheng, Y., Wang, D., Zhou, P., Zhang, T.: A survey of model compression and acceleration for deep neural networks. *IEEE Signal Processing Magazine* **35**(1), 126–136 (2018)
2. Courbariaux, M., Bengio, Y.: Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR* **abs/1602.02830** (2016), <https://arxiv.org/abs/1602.02830>
3. Dai, D., Zhang, Y., Zhang, J., Hu, Z., Cai, Y., Sun, Q., Zhang, Z.: Trainable fixed-point quantization for deep learning acceleration on fpgas (jan 2024). <https://doi.org/10.48550/arXiv.2401.17544>, <https://arxiv.org/abs/2401.17544>
4. Dong, Z., Pan, Z., Zou, X., Cao, Y., Gu, J., Wang, Y.: Hawq: Hessian aware quantization of neural networks with mixed precision. In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. pp. 293–302 (2019)
5. Dupuis, V., Diann, A., Marchand, P., Ottavi, G.: Adaqat: Adaptive quantization-aware training for efficient low-bit neural networks. In: *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. pp. 1320–1325 (2024)
6. Gysel, P., Motamedi, M., Ghiasi, G.: Ristretto: Hardware-oriented approximation of convolutional neural networks. In: *Proceedings of the 33rd International Conference on Machine Learning (ICML) Workshop on Efficient Methods for Deep Learning* (2016)

7. Han, S., Mao, H., Dally, W.J.: Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In: Proceedings of the International Conference on Learning Representations (ICLR) (2016)
8. Huang, H., Zhang, Y., Guo, Y., Li, J.: Sdq: Stochastic differentiable quantization with mixed precision. In: Proceedings of the 39th International Conference on Machine Learning (ICML). pp. 9095–9110 (2022)
9. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Quantized neural networks: Training neural networks with low precision weights and activations. In: Proceedings of the International Conference on Learning Representations (ICLR) (2017)
10. Hubara, I., Soudry, D., Ran El Y.: Binarized neural networks. CoRR **abs/1602.02505** (2016), <https://arxiv.org/abs/1602.02505>, withdrawn—superseded by arXiv:1602.02830
11. Jouppi, N.P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Laudon, J.: In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA). pp. 1–12 (2017)
12. Krishnamoorthi, R.: Quantizing deep convolutional networks for efficient inference: A whitepaper. <https://arxiv.org/abs/1806.08342> (2018), arXiv:1806.08342
13. Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Zhu, H.: Mixed precision training. In: Proceedings of the International Conference on Learning Representations (ICLR) (2018)
14. Nagel, M., Pedersoli, M., Van Gool, L.: Data-free quantization through weight equalization and bias correction. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). pp. 7358–7367 (2019)
15. Rastegari, M., Ordonez, V., Redmon, J., Farhadi, A.: Xnor-net: Imagenet classification using binary convolutional neural networks. CoRR **abs/1603.05279** (2016), <https://arxiv.org/abs/1603.05279>
16. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. <https://arxiv.org/abs/1409.1556> (2015), arXiv:1409.1556
17. Strubell, E., Ganesh, A., McCallum, A.: Energy and policy considerations for deep learning in nlp. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL). pp. 3645–3650. Association for Computational Linguistics (2019)
18. Umuroglu, Y., Fraser, N.J., Gambardella, G., Blott, M., Leong, P., Jahre, M., Vissers, K.: 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)
19. Wang, K., Zhang, D., Zhou, A., Lin, Y., Xu, C., Jin, L., Li, Y.: Haq: Hardware-aware automated quantization. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). pp. 6801–6810 (2019)
20. Zhang, X., Fan, K., Li, X., et al.: Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In: Proceedings of the European Conference on Computer Vision (ECCV). pp. 365–382 (2018)