Octo: INT8 Training with Loss-aware Compensation and Backward Quantization for Tiny On-device Learning

Qihua Zhou†, Song Guo†, Zhihao Qu‡, Jingcai Guo†, Zhenda Xu†,
Jiewei Zhang†, Tao Guo†, Boyuan Luo†, Jingren Zhou*
†*Hong Kong Polytechnic University,* ‡*Hohai University,* **Alibaba Group*

Abstract

On-device learning is an emerging technique to pave the last mile of enabling edge intelligence, which eliminates the limitations of conventional in-cloud computing where dozens of computational capacities and memories are needed. A high-performance on-device learning system requires breaking the constraints of limited resources and alleviating computational overhead. In this paper, we show that employing the 8-bit fixed-point (INT8) quantization in both forward and backward passes over a deep model is a promising way to enable tiny on-device learning in practice. The key to an efficient quantization-aware training method is to exploit the hardware-level enabled acceleration while preserving the training quality in each layer. However, off-the-shelf quantization methods cannot handle the on-device learning paradigm of fixed-point processing. To overcome these challenges, we propose a novel INT8 training method, which optimizes the computation of forward and backward passes via the delicately designed *Loss-aware Compensation* (LAC) and *Parameterized Range Clipping* (PRC), respectively. Specifically, we build a new network component, the compensation layer, to automatically counteract the quantization error of tensor arithmetic. We implement our method in Octo, a lightweight cross-platform system for tiny on-device learning. Evaluation on commercial AI chips shows that Octo holds higher training efficiency over state-of-the-art quantization training methods, while achieving adequate processing speedup and memory reduction over the full-precision training.

1 Introduction

The unprecedented booming of Machine Learning (ML) techniques has achieved great success in the past decade [8, 35]. The magic of ML comes from model training on large-scale datasets, relying on the deployment in the cloud environment to meet the resource-hungry demands [44, 45, 60]. This kind of in-cloud computing paradigm can bring about the essential drawbacks that it is hard to provide personalized models [14], suffering from high latency [46] and privacy leakage [40]. These issues promote the rise of on-device learning [5, 39, 65], which eliminates in-cloud learning's limitations by handling the end-to-end learning process totally on user devices [4, 57], *e.g.,* mobile and IoT equipment [54]. This edge environment makes on-device learning often subject to limited computational capacity and memory [66]. Therefore, breaking resource constraints is the key step to implement on-device learning systems.

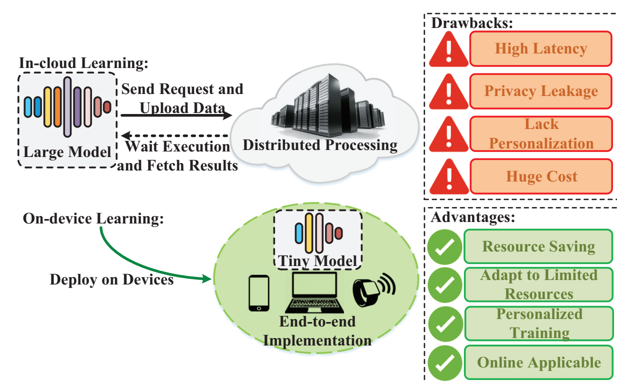


Figure 1: Conventional ML applications rely on the in-cloud learning paradigm, incurring essential drawbacks. A new trend is to use on-device learning to address these issues.

Conventional model compression methods, *e.g.,* Low-rank Decomposition [31, 61–63], Model Pruning [11, 16, 18, 20] and Network Sparsification [1, 19, 37], are insufficient as they are designed for large-scale training tasks and cannot well match the characteristic of tiny on-device learning. Fortunately, previous researches reveal that neural networks are often implemented in a 32-bit floating-point (FP32) format [17] and representing model parameters in such high precision is not always necessary [36]. It is feasible to represent parameters in lower bit precision while not downgrading the entire network quality. Based on our preliminary experiments (§2.4) that handle CNN training in the 8-bit fixed-point (INT8) format [2], we are inspired by the potential improvements and

thus intend to design a full-INT8 quantization-aware training system for on-device learning.

However, existing quantization methods can hardly be employed for on-device training due to the following limitations: (1) cannot apply to the training process [3, 5, 24, 36, 42], (2) cannot support generic networks without specific structure design [34, 47, 58], (3) cannot enable hardware-level INT8 acceleration during the training phase [7, 23, 49, 56], and (4) cannot make the gradient calibration fit on-device resource restrictions in backward pass [32, 67]. Therefore, we need a full-INT8 training method working on devices directly, covering both forward and backward passes.

Designing a desired INT8 on-device training is not easy, and we need to overcome the following challenges: (1) simplifying the computational procedure and truly accelerating processing speed on devices, (2) maintaining model quality when using INT8 quantization-aware training, (3) alleviating system overhead, *e.g.,* reducing memory footprint and I/O bandwidth utilization, and (4) making the system ease-of-use and compatible with multiple platforms. We achieve these objectives via a co-design of neural network constructor and 8-bit training engine.

More precisely, we give deep insights into the rationale of 8-bit training (§4.1) and present a theoretical analysis of layer-wise quantization error (§4.2). We point out the key to preserve model quality in forward pass is to fill the error gap in tensor dot products and thus propose the *Loss-aware Compensation* (LAC) method (§4.3) to approximately recover the quantized output to the full-precision domain. Specifically, we propose a new network component, the compensation layer (§4.3.1), to handle this adjustment with three learnable parameters, and design an L2-regularizer (§4.3.2) to bound the update rate of these parameters for convergence stability. Meanwhile, we optimize the gradient calculation in backward pass by abstracting the calculation of derivative flows (§4.4.1) as a series of primary instructions (*e.g.,* dot product, broadcasting and bit-wise shifting) and introducing symmetric quantization on them. We carefully profile the possible distribution of gradient tensors and propose the *Parameterized Range Clipping* (PRC) method (§4.4.2) to handle INT8 quantization of intermediate derivatives. We address the issue of zero point offset in backward pass by restricting clipping domain in the symmetric scheme, under 95% confidence interval of gradient distribution.

We implement our INT8 training method in Octo, a lightweight cross-platform system for on-device learning tasks. Octo's training engine and network constructor are built in pure Python without the dependency of other sophisticated third-party libraries, and thus it is easy to port Octo to embedded devices. We evaluate Octo in different operating systems and deploy it on commercial AI platforms, such as HUAWEI Atlas 200DK [26] and NVIDIA Jetson Xavier [27], which can utilize the dedicated INT8 neuron chips. The evaluation results show that Octo achieves higher training efficiency over state-of-the-art quantization training methods, with tiny system overhead. Specifically, Octo accelerates processing speed by up to 2.03× and saves memory footprint by up to 3.37×, over the full-precision training.

Overall, the key contributions of our work are as follows:

- We propose a lightweight INT8 training method, enabling data quantization for both forward and backward stages (§4.1), which is efficient for deploying on-device learning applications.
- We present the theoretical characterization of quantization errors and reveal that filling the error gap caused by inner product is the key to preserve model quality of quantization-aware training (§4.2). This analysis guides us to design the compensation term to adjust quantized convolutional output.
- To achieve stable training efficiency, we propose the *Loss-aware Compensation* (LAC) and *Parameterized Range Clipping* (PRC) methods to handle data quantization in forward pass (§4.3) and backward pass (§4.4), respectively. LAC introduces the novel compensation layers and updates the compensation term based on the feedback of network loss. Meanwhile, PRC maintains bit precision in gradient calculation and avoids offset error of the zero point via symmetric clipping.
- We build a cross-platform system named Octo, which is compatible with different operating systems and can be easily ported to embedded platforms (§5). Octo is specifically designed for on-device training by exploiting the fixed-point computational primitives of embedded processors, with no need of GPUs. Evaluation on HUAWEI Atlas 200DK [26] and NVIDIA Jetson Xavier [27] shows that Octo achieves higher system performance over state-of-the-art quantization training methods, thus verifying the effectiveness of our approach (§6).

To the best of our knowledge, Octo is the first general framework to implement INT8 training on devices, optimizing both forward and backward stages. The project of Octo is open-source[1] and will constantly contribute to the further development of on-device learning techniques in practice.

2 Motivation

We start by discussing the limitations of conventional in-cloud learning (§2.1), and introducing the background of on-device learning applications (§2.2). Then, we will point out quantization-aware training (§2.3) is the key to deploy on-device learning in real-world scenarios, holding significant advantages over conventional model compression methods. We also present a case study to demonstrate the performance improvements (§2.4) by leveraging INT8-based training on

[1] https://github.com/kimihe/Octo

devices, and analyze the limitations of existing quantization methods (§2.5).

2.1 Limitations of In-cloud Learning

Training ML models is a resource-hungry procedure, relying on a mass of learnable data and computational capacity. Thus, conventional ML applications are often deployed in the cloud environment, requiring expensive cost of machine clusters and distributed processing. However, such an in-cloud learning paradigm is vulnerable to privacy leakage as user information may be exposed to untrusted third parties. More seriously, the high latency for fetching inference results may become the bottleneck to slow down the learning performance. Also, as models are trained in a global scheme (*e.g.,* the Federated Learning [40]), it is not easy to provide customized models for different users to match personalized preferences [4].

2.2 Rise of On-device Learning

With the rapid growth of device processing capacity and memory volume, it comes the rise of on-device learning, which handles the end-to-end ML procedure on devices directly and breaks the limitations of in-cloud learning. A pertinent case in industry is the Apple Face ID [51], which enables personalized face recognition totally on user phones. Here, we briefly introduce the definition and key objectives of on-device learning.

Definition. In general, on-device learning refers to entirely transferring the model training and inference procedure on user devices, with no need of data exchanging to other machines [54]. Besides, on-device learning can apply to edge equipment (*e.g.,* IoT and mobile devices), thus it also corresponds to Edge Intelligence [66] in practice.

Objective #1: Resource Saving. The on-device hardware is often bounded by the resource-constrained environment, with limited computational capacity [30], memory volume [22, 33] and I/O bandwidth [38]. Therefore, saving the system overhead is the key to deploy on-device learning.

Objective #2: Model Quality. The learning algorithm should hold a stable convergence efficiency, with a comparable prediction accuracy and generalization ability as the in-cloud training schemes [39].

Objective #3: Personalized Training. As the training procedure is entirely based on the local data on user devices, it should provide customized models to meet user preference [14], instead of generating a global model for an ensemble of applications.

Objective #4: End-to-end Implementation. Considering the incremental user data, the trained models should keep up-to-date continuously. Thus, the entire learning process requires an end-to-end implementation on the devices [6].

Summary. These four objectives are the key metrics guiding the design of high-efficiency on-device learning systems.

2.3 Bit Precision and Data Quantization

Considering the resource-constrained environment on devices, compressing model size and alleviating computational pressure is the key to applying on-device learning in real-world scenarios. Although there are some common compression methods, *e.g.,* Low-rank Decomposition [31, 61–63], Model Pruning [11, 16, 18, 20] and Network Sparsification [1, 19, 37], they are designed for large-scale training tasks and cannot well match the pattern of tiny on-device learning. Fortunately, data quantization is a promising method to address these limitations.

Definition. The gist of quantization is to represent data via less bit precision, *e.g.,* converting a 32-bit floating-point (FP32) number to the 8-bit fixed-point (INT8) format. Here are two core operations of data quantization.

Operation #1: Number Discretization. The first operation is to map real numbers from a "continuous" domain to certain discrete values, *e.g.,* from floating-point numbers to integers. The number of discrete values is called the quantization level. A common way is to partition the original domain into several intervals and represent the numbers located in an interval by the central point [36]. The objective of number discretization is to reduce the value variety and represent the numbers by a few target points.

Operation #2: Domain Transformation. The second operation is to restrict the values from a wide representation range to a small range, *e.g.,* from 32 bits to 8 bits. The transformation procedure can be abstracted as a step function, where the "width" of the step can be uniformly or non-uniformly assigned [24]. Uniform transformation is hardware-friendly while non-uniform transformation can provide higher bit precision. Therefore, domain transformation aims at storing each real number in the fixed-point format with fewer bits.

Inspirations. Existing ML frameworks often implement the tensor arithmetic in FP32 or FP64 format to maintain high precision of numerical operations. However, previous work reveals that most neural networks are over-parameterized and representing model parameters in such high-precision format is not necessary [36]. It is feasible to maintain parameters in lower bit precision while not downgrading the entire network quality. Besides, numerical operations based on floating-point formats are much more expensive than fixed-point ones, especially for the tiny IoT equipment without floating-point processing units. Therefore, it comes to our motivation to compress model parameters from FP32 to INT8 format and handle the training procedure on devices.

Summary. Overall, we summarize the following properties of data quantization: First, it enables model in bit level, which can effectively reduce memory footprint and accelerate tensor arithmetic. This helps us implement on-device learning systems in resource-constrained cases. Second, quantization is more hardware-friendly for both generic hardware (*e.g.,* CPU/GPU) and specific chips (*e.g.,* FPGA), which can be

	Forward Pass (ms)	Backward Pass (ms)	Per-iteration Time (ms)	Parameter Memory (MB)	Model Accuracy
FP32	95.85	140.03	240.06	18.51	97.6%
INT8	54.57	67.66	126.41	9.42	95.2%
Comparison	1.86×	2.07×	1.89×	1.96×	−2.39%

Table 1: System performance using INT8 and FP32 training.

easily applied to edge intelligence applications.

2.4 Potential Gains

Employing data quantization into model training can save resource costs and accelerate processing speed. Here, we present an illustrative case study to show the potential gains. The experiment is the image classification task on a 3-layer CNN with MNIST dataset [9], running on the HUAWEI Atlas 200DK platform [26] with 10 epochs. We quantize the mini-batch input, weights and gradients of convolutional layers into INT8 format and inspect the system performance in terms of arithmetic efficiency, memory footprint and I/O bandwidth. From the comparison shown in Table 1, the INT8-based quantization-aware training can effectively alleviate system overhead while not downgrading the model quality. This raises an interesting question: can we achieve the same level of FP32 training performance with only INT8 operations for common on-device learning applications (*e.g.,* image classification)? Therefore, we are dedicated to building a lightweight INT8 training system to achieve this target.

2.5 Why not Existing Quantization Methods?

To implement on-device learning systems, existing quantization methods are insufficient due to the following limitations.
#1. Cannot apply to training process. Most quantization methods are designed for inference only, where quantization is used in the forward pass based on a pre-trained model for accelerating the prediction speed [3, 5, 24, 36, 42]. As these methods have not addressed the issues of calculating gradients on discretized parameters and eliminating error gap after convolutional operations, they cannot be used in training process.
#2. Cannot support generic networks without specific structure design. Some methods aim at quantizing parameters with extremely low bit precision, *e.g.,* the binary [58], ternary [34] and XNOR [47] networks. However, they are specifically designed and require fundamentally modifications of network structures. Thus, they are not suitable for the training of generic networks.
#3. Cannot enable hardware-level INT8 acceleration in training phase. Google has proposed a verifying quantization method, called Fake Quantization [23], which uses INT8-based numerical information for parameter representation, while still packaging values in FP32 format for tensor arithmetic. The subsequent methods [7, 49, 56] also follow this paradigm that tensors are quantized and dequantized before arithmetic operations. This paradigm cannot fundamentally accelerate processing speed or reduce memory footprint because it does not exploit the hardware-level power of fixed-point processing.
#4. Cannot make the gradient calibration in backward pass fit on-device resource restrictions. Recently, it is a trend to study the quantization-aware training with 8 bits. For example, Zhu *et al.* [67] proposed the unified INT8 training covering both forward and backward passes. However, current researches have not optimized the derivative calculation of model parameters and intermediate tensors during the backward pass, which still follows the fake quantization paradigm. As the backward pass often dominates the per-iteration time, it is of great potential to conduct backward quantization to further improve training efficiency.
Summary. Overall, we aim at designing an INT8 quantization method for training neural networks directly on devices.

3 Overview and Design Challenges

Enabling INT8 quantization on devices requires a co-design of neural network constructor and 8-bit training engine. We present Octo, an INT8 quantization-aware training system that addresses the following key challenges.
Challenge #1: How to fundamentally accelerate processing speed on devices? Existing quantization methods based on fake quantization cannot fully exploit the power of INT8 processing, thus the on-device training performance is still bounded by iterative tensor arithmetic. We need to simplify the computational procedure for more effective acceleration.
Our solution: We introduce the uniform 8-bit quantization into convolutional operations, affine blocks, activation functions and gradient calculation. The data quantization covers both forward and backward passes, thus the entire iteration can be accelerated. Also, the uniform range transformation is hardware-friendly and can be implemented in the layer wise.
Challenge #2: How to maintain model quality when using INT8 quantization-aware training? Introducing data quantization during model training will inevitably impact the numerical precision of parameters and incur accumulative errors cross layers. The final output will be significantly different from the full-precision training. Besides, the gradient calculation will face the same issue if we use quantization in the derivative calculation. These factors will degrade the final model accuracy and even make the training cannot converge.
Our solution: We maintain the model accuracy by adjusting the intermediate results of forward and backward passes together. In the forward pass, we propose the *Loss-aware Compensation* (LAC) method and design a new network component, called the compensation layer, to fill the error gap caused by quantized tensor arithmetic. The parameters inside the compensation layer will be optimized according to the network loss. For a higher updating efficiency, we introduce an L2-regularization term of compensation parameters

to modify the loss function. In the backward pass, we propose the *Parameterized Range Clipping* (PRC) to bound the transformation domain of quantized gradients, using a 95% confidence interval based on our theoretical analysis (§4.2).

Challenge #3: How to alleviate system overhead, especially reducing memory footprint? The training efficiency is often bounded by the limited memory volume and I/O bandwidth. This requires us to conduct memory and storage optimization in the training engine.

Our solution: We preserve all the parameters and intermediate derivatives in INT8 format. This effectively reduces the peak memory footprint and saves the storage cost for accessing parameter cache. Also, introducing LAC and PRC mentioned in challenge #2 may incur extra overhead. We transfer the compensation term from a higher-degree polynomial using FP32 tensors into an affine operation just relying on the output of convolutional layers. This can effectively restrict the computational cost of compensation and clipping.

Challenge #4: How to make the system ease-of-use and compatible with multiple platforms? In real-world cases, devices are often handled by tiny embedded systems that may not be well compatible with commodity learning frameworks (*e.g.,* PyTorch Mobile [41] and TensorFlow Lite [15]). This requires us to build the system based on the standard environment without the dependency of other sophisticated third-party libraries.

Our solution: We first abstract core computation of convolutional and fully-connected layers into the basic operation of tensor-wise dot product. Then, we optimize the dot products by using pure Python and C++. We use the light-weight Pybind tool [43] and C++ header-based Eigen [52] to embed the hardware-level matrix instructions in Python-level training. This kind of hybrid implementation makes our system compatible with most operating systems, including embedded Linux, macOS and Windows.

Summary: The above four challenges and solutions guide the design of our system.

4 Octo Design

We first discuss the rationale of 8-bit training, and highlight the key difference between Octo and the prior fake quantization method (§4.1). Then, we present a theoretical formulation of quantization error and analyze how to preserve training quality (§4.2). We propose the *Loss-aware Compensation* (LAC) and *Parameterized Range Clipping* (PRC) methods to conquer the aforementioned challenges, in forward (§4.3) and backward pass (§4.4), respectively.

4.1 Workflow of 8-bit Training

We describe the gist of 8-bit training by discussing the prior work of fake quantization (§4.1.1), which is helpful for understanding the next steps of Octo design. For a clear explanation, we take a 3-layer CNN (1 CONV + 2 FCs) as the example and list all the notations in Table 2.

Variable	Definition
n	The number of bits used for quantization.
W_f	The full-precision FP32 weights.
X_f	The full-precision FP32 input.
W_q	The quantized INT8 weights.
X_q	The quantized INT8 input.
Y_f	The FP32 output of dot product in fake quantization.
Y_q	The INT32 output of dot product in Octo.
s_w	The scaling factor when quantizing W_f.
s_x	The scaling factor when quantizing X_f.
z_w	The offset of zero point when quantizing W_f.
z_x	The offset of zero point when quantizing X_f.
$Q(X)$	The quantization function.
$Q^{-1}(X)$	The dequantization function.
$\texttt{scale}(M)$	The function for calculating scaling factor of tensor M.
$\texttt{round}(M)$	The discretization function using stochastic rounding on tensor M.
$\texttt{clip}(M)$	The clipping function to bound the range of tensor M.
$\texttt{dot}(X,W)$	The dot product of X and W.
r_x	The error gap incurred by discretizing X_q.
r_w	The error gap incurred by discretizing W_q.
δ	The error gap between FP32 and INT8 dot product.
$\hat{\delta}$	The compensation term to approximate δ .

Table 2: Notations used in INT8 quantization-aware training.

4.1.1 Fake Quantization Training

Compared with vanilla full-precision training (Figure 2(a)), fake quantization (Figure 2(b)) first quantizes the input and weight into INT8 format, then it dequantizes these variables back to the FP32 format before conducting tensor dot product. Note that the computational overhead of CONVs and FCs mainly comes from convolutional and affine operations, which can be unfolded as a series of dot products in low-level instructions. Therefore, we regard dot products as the key computation in forward pass. The rationale of fake quantization training can be described as following three steps:

Step #1: Quantization. This step transfers FP32 numbers to INT8 format.

$$X_q = \texttt{round}(\frac{X_f}{s_x} + z_x), \tag{1}$$

$$W_q = \texttt{round}(\frac{W_f}{s_w} + z_w), \tag{2}$$

$$s_x = \texttt{scale}(X,n) = \frac{\max(X_f) - \min(X_f)}{2^n - 1}, \tag{3}$$

$$z_x = \max(X_q) - \frac{\max(X_f)}{s_x}, \tag{4}$$

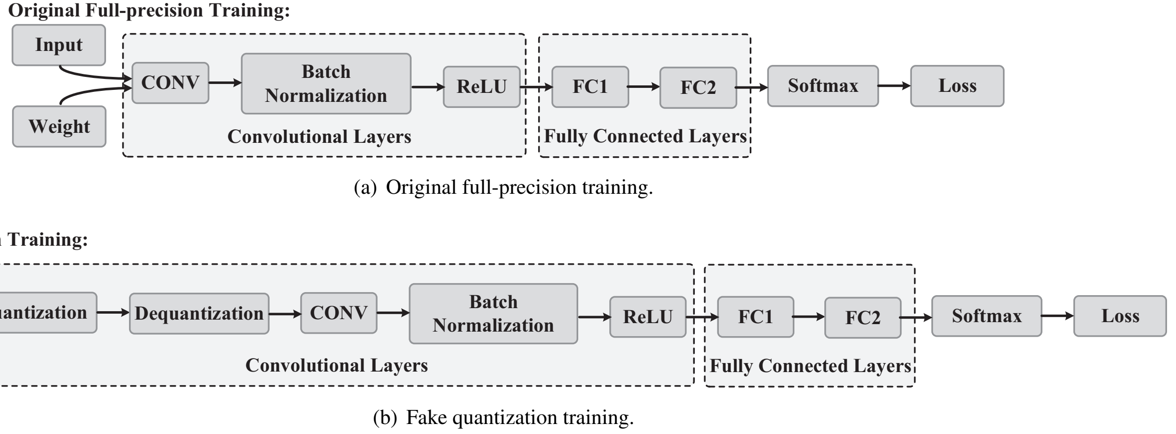
$$s_w = \texttt{scale}(W,n) = \frac{\max(W_f) - \min(W_f)}{2^n - 1}, \tag{5}$$

$$z_w = \max(W_q) - \frac{max(W_f)}{s_w}. \tag{6}$$

Step #2: Dequantization. This step recovers the INT8 numbers to FP32 format.

$$X_f = (X_q - z_x) \cdot s_x \tag{7}$$

$$W_f = (W_q - z_w) \cdot s_w \tag{8}$$



(a) Original full-precision training.

(b) Fake quantization training.

Figure 2: Workflow of previous training methods.

Step #3: Dot Products. This step conducts dot products based on the recovered FP32 tensors and yields FP32 results.

$$Y_f = \texttt{dot}(X_f, W_f) \tag{9}$$

Performance Analysis. Fake quantization captures the INT8 numerical information while still handling tensor arithmetic in FP32 format. Thus, *it cannot truly alleviate the computational overhead*. Also, the precision degradation after the first two steps mainly comes from the integer rounding, thus the dot products in the third step will not incur much error. However, a practical quantization-aware training method needs to move the dot products between quantization and dequantization, so as to utilize the power of fixed-point hardware. We follow this principle to design Octo's training.

4.1.2 Octo's Training

As shown in Figure 3, Octo holds the tensor arithmetic totally in INT8 format and contains the following three steps.

Step #1: Quantization. This step follows the same formulation as Eq. (2) and Eq. (1). We use the stochastic rounding [17] to build the discretization function $\texttt{round}(M)$, which maps floating-point numbers into integers and can be defined as:

$$\texttt{round}(M) = \begin{cases} \lfloor x \rfloor, & w.p. \quad 1 - \dfrac{x - \lfloor x \rfloor}{\varepsilon} \\ \lfloor x \rfloor + \varepsilon, & w.p. \quad \dfrac{x - \lfloor x \rfloor}{\varepsilon} \end{cases}, \tag{10}$$

where ε represents the smallest positive fixed-point value under given bits. For example, $\varepsilon = 2^{-8}$ in 8-bit quantization. This discretization function holds the unbiased rounding property, thus the expected rounding error is zero, *i.e.,* $\mathbb{E}[\texttt{round}(M)] = M$. This property alleviates the discretization error caused by integer rounding.

Step #2: Dot Product This step conducts dot products on INT8 tensors and returns the INT32 results to avoid overflow.

$$Y_q = \texttt{dot}(X_q, W_q) \tag{11}$$

Step #3: Dequantization with compensation This step converts the INT32 tensor to FP32 format and compensates the error of dot products caused by step #2.

Performance Analysis. If $z_x = 0$ and $z_w = 0$, we can simply use the following transformation to restore the INT32 output to FP32 format.

$$Y_f = Y_q \cdot (s_x \cdot s_w). \tag{12}$$

However, realistic INT8 processing requires `int` or `unsigned int` data type, which means the quantized values should be restricted within $[-128, 127]$ or $[0, 255]$, instead of using arbitrary 8-bit domains (*e.g.,* $[100, 355]$ is not feasible). Thus, we need non-zero z_x and z_w to serve as the offset of domain transformation. As z_x and z_w participate the dot product, we cannot recover Y_f by using Eq. (12), because the multiplication of zero point offset will incur a significant error gap. Compared with the vanilla dot product based on FP32 tensors, the dequantized output can be described as follows.

$$Y_f = Y_q \cdot (s_x \cdot s_w) + \delta, \tag{13}$$

where δ is the polynomial related to z_x and z_w. Therefore, step #3 is the key to Octo's forward pass, and the gist is to *find a proper compensation term to fill the error gap δ while not incurring too much computational overhead*. The details will be discussed in the next section (§4.2).

4.2 Analysis of Error Gap

In this section, we formulate the error gap δ and approximate it as an affine transformation. Due to the page limit, we omit proof details and present the major theorem.

Theorem 1. *The error gap δ between FP32 and INT8 dot product can be approximated as an affine transformation,*

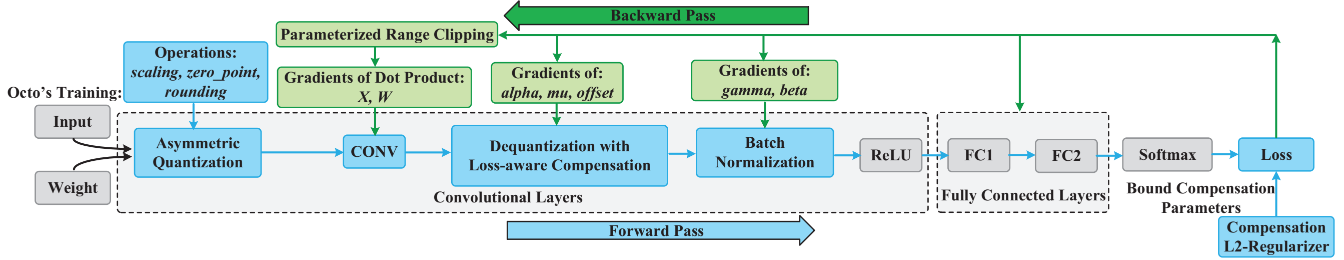


Figure 3: Workflow of Octo's training.

which is defined as:

$$\delta = s_x * \Delta X \cdot W_f + \gamma, \tag{14}$$

$$\Delta X = \frac{X_f}{s_x} - \mathtt{round}(\frac{X_f}{s_x} + z_x), \tag{15}$$

$$\gamma = X_f(r_w - z_w)s_w - (r_x - z_x)(r_w - z_w)s_x s_w, \tag{16}$$

where $*$ represents the broadcast operation on tensors. Note that $\Delta X \cdot W_f$ dominates the computational overhead of Eq. (14), with the same shape as the dot product. Also, s_x and γ can be regarded as the coefficient factor and bias, respectively. This formulation can be simplified as an affine transformation and we approximate δ as:

$$\hat{\delta} = \alpha * \mu + \beta. \tag{17}$$

We call $\hat{\delta}$ as the compensation term, where the three parameters α, μ and β can be optimized based on the loss function of the network. Specifically, we abstract these parameters as a new network component, *i.e.,* the compensation layer (§4.3), to better adjust the quantized dot product.

4.3 Loss-aware Compensation

We propose the *Loss-aware Compensation* (LAC) method to preserve the numerical precision of tensor arithmetic in forward pass, LAC holds two core modules: (1) compensation layer (§4.3.1) and (2) L2-regularization of compensation parameters (§4.3.2).

4.3.1 Compensation Layer

The compensation layer is a new component injected to the network structure. Our target of introducing the compensation layer is to fill the error gap mentioned in Eq. (13) by approximating $\hat{\delta}$ in an affine transformation. We intend to compensate the quantized output in each layer to avoid the large accumulative errors across layers. Thus, the compensation layer is designed as an independent component that can be added at the end of both convolutional (CONV) and fully-connected (FC) layers. As shown in Fig. 4, we conduct preliminary experiments to inspect the layer completion time on CONVs and FCs. We can observe that the time cost of FCs is far less than that of CONVs. Besides, FCs usually occupy a tiny part

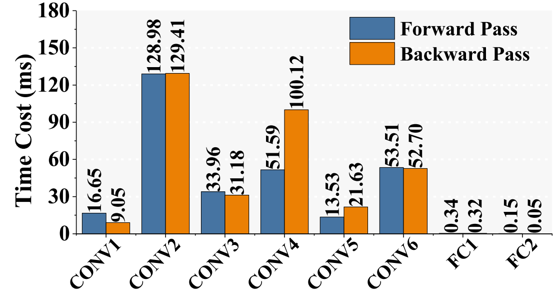


Figure 4: Lyer completion time on CONV and FC layers.

of the entire network, quantizing FCs will not bring essential improvements to alleviate computational overhead. Therefore, we focus on using LAC on CONVs (although LAC can also be applied to FCs) and insert a compensation layer at the end of each CONV, following batch normalization to adjust the input distribution before ReLu, as depicted in Figure 3. Indeed, the compensation layer is an optional component, thus we can add a compensation layer after multiple CONVs. Each compensation layer holds three learnable parameters and we summarize their properties as follows.

#1. α: This is a scalar controlling the scaling factor of compensation term.

#2. μ: This is a tensor with the same shape as the dot product, which represents the expectation of the distribution of δ.

#3. β: This is a tensor corresponding to the shape of μ, serving as the bias to adjust the compensation offset.

According to the expression of compensation term in Eq. (17), the calculation of $\alpha * \mu$ is handled by the broadcast operation, instead of the time-consuming tensor dot product. Also, adding β to $\alpha * \mu$ can be optimized as a shift operation, holding much less arithmetic complexity over multiplication. Therefore, calculating the compensation term will not incur much computational overhead.

4.3.2 L2-Regularization of Compensation Parameters

Although adding compensation layers can effectively fill the error gap caused by quantized dot product, the overall training efficiency may be impacted by the initialization of the compensation parameters. Actually, we can treat the approximated compensation term as special "noise" to counteract the

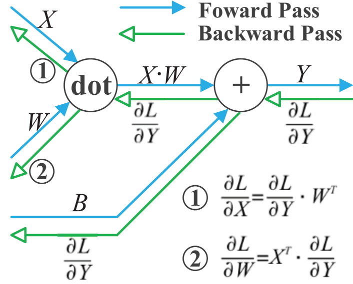


Figure 5: The derivative flows across layers in backward pass.

quantization error. Thus, the network robustness and convergence speed with compensation layers still rely on the update rule of the compensation layers. To improve the stability of compensation layers and accelerate the learning speed of compensation parameters, we design a dedicated L2-regularizer [55, 64] to better capture the significance of compensation layers. By adding the L2-regularization term to loss function, we modify the corresponding gradient calculation rule for updating the compensation parameters more efficiently. The modified loss function L is defined as follows:

$$L = \underbrace{-\frac{1}{N}\sum^{n}\sum^{k} t_{nk}\log y_{nk}}_{(1)} + \underbrace{\frac{1}{2}\lambda(\mu^2+\beta^2)}_{(2)}, \qquad (18)$$

where the first term represents the primary cross-entropy error measuring the difference between prediction y and ground truth t, based on mini-batch of size N. Meanwhile, the second term is the L2-regularizer reflecting the compensation performance from μ and β. As α is a scalar, we can omit its impact here. Actually, we can adjust the significance of these two terms by changing the value of λ. For example, we can set λ with a larger value to emphasize the effect of compensation layers. In practice, we empirically set λ as 0.1.

4.4 Backward Quantization

Apart from introducing LAC in forward pass, we also employ data quantization in backward pass as the calculation of gradients can also be abstracted as a series of dot product, following the chain rule of derivative flows (§4.4.1). However, we cannot simply use the INT8 quantization mentioned in forward pass because we cannot add a compensation-like component in backward pass to fill the error gap incurred by zero points. Here, we propose the *Parameterized Range Clipping* (PRC) method to address this issue by restricting the clipping domain in the symmetric scheme (§4.4.2).

4.4.1 Calculation of Derivative Flows

In practice, we usually calculate gradients based on chain rules instead of using numerical differential, for higher computational efficiency. As shown in Figure 5, the derivative flows will go through the entire network, from the last layer to the first one. Capturing the flows from the l+1-th layer, we can handily calculate the gradients of current l-th layer and push the flows to previous layers. Thus, the essential operations for calculating the parameter gradients ($\frac{\partial L}{\partial X}$ and $\frac{\partial L}{\partial W}$) of CONVs and FCs can also be abstracted as a serious of tensor dot products, which are described as follows:

$$\frac{\partial L}{\partial X} = \texttt{dot}(\frac{\partial L}{\partial Y}, W^\top), \qquad (19)$$

$$\frac{\partial L}{\partial W} = \texttt{dot}(X^\top, \frac{\partial L}{\partial Y}), \qquad (20)$$

where $W^\top$ and $X^\top$ represent the transpose of W and X, respectively. Following these equations, we can also quantize $\frac{\partial L}{\partial Y}$ into INT8 format, and conduct the quantized dot product by using W_q and X_q, which are obtained in forward pass. After that, we can dequantize $\frac{\partial L}{\partial X}$ and $\frac{\partial L}{\partial W}$ based on the corresponding scaling factors by using Eq. (12). Note that this dequantization relies on the prerequisite of using symmetric clipping, which will be discussed in the next section.

4.4.2 Parameterized Range Clipping

As the intermediate derivative flows are also quantized before the dot product for calculating gradients, we thus need to address the issue of zero point offset. However, adding a compensation-like component in backward pass is inefficient in filling the error gap. Instead, we propose the *Parameterized Range Clipping* (PRC) method that makes clipping in the symmetric scheme to avoid the involvement of zero point, such that the FP32 parameter gradients could be recovered after conducting the INT8 dot product. The clipping range will be specifically profiled according to the distribution of the FP32 tensors. More precisely, we will discuss the clipping strategy in the following two cases.

Case #1. The distribution of FP32 tensor locates on both sides of the original point. In this case, we will quantize the FP32 tensor within $[-127, 127]$, which can be covered by the `int` data type. The clipping function is described as:

$$\texttt{clip}(M) \in [-a, a], \qquad (21)$$

$$a = \min\{|\min(M)|, \max(M)\}. \qquad (22)$$

Case #2. The distribution of FP32 tensor locates on one side of the original point. In this case, we will quantize the FP32 tensor within $[0, 255]$ that can be covered by the `unsigned int` data type. The clipping function is defined as:

$$\texttt{clip}(M) \in [0, a], \qquad (23)$$

$$a = \max\{|\min(M)|, |\max(M)|\}. \qquad (24)$$

By profiling the clipping range in these two cases, we can assert the zero point offset as 0 and surpass the issue of error gap. Moreover, we further restrict the clipping range by applying

95% confidence interval on Eq. (21), as the gradient tensors usually follow the long-tailed but bell-shape distribution.

Gradient Recovery: Based on the above clipping, we can recover the FP32 parameter gradients ($\frac{\partial L}{\partial X_f}$ and $\frac{\partial L}{\partial W_f}$) from the intermediate INT8 quantized derivative flows (Y_q, W_q and X_q) as follows:

$$\frac{\partial L}{\partial X_f} = \texttt{dot}(\frac{\partial L}{\partial Y_q}, W_q^\top) \cdot (s_y s_w), \tag{25}$$

$$\frac{\partial L}{\partial W_f} = \texttt{dot}(X_q^\top, \frac{\partial L}{\partial Y_q}) \cdot (s_x s_y), \tag{26}$$

where s_y, s_x and s_w represent the scaling factors for quantizing Y_f, X_f and W_f, respectively.

5 Implementation

We implement Octo's training engine and network constructor in pure Python without the dependency of other sophisticated third-party libraries, making it a cross-platform system.

5.1 Network Construction

We abstract different layers for the construction of common CNNs, including CONV layer, FC layer, Batch Normalization, ReLu, Sigmoid, Pooling, Dropout, Softmax, and the proposed compensation layer. Each layer holds uniform APIs to handle the forward and backward passes. We can easily build CNNs by assembling different layers in a proper sequence. We provide a configuration file to set layer size (*e.g.,* neuron umber, filter number, filter size and padding) and initialize model parameters. We have embedded AlexNet, VGG11 and other deep CNNs in Octo, supporting the training on MNIST, Fashion MNIST and CIFAR-10/100 datasets. Moreover, we build the auto inspection module to monitor the training performance and record the key metrics, including model accuracy, training completion time, per-iteration cost, computational overhead, memory footprint and compensation performance.

5.2 Gradient Calculation

We implement the gradient calculation based on the chain rule of derivative flows. Each layer can automatically obtain its gradients by invoking the `backward` method. Here, we take the compensation layer as an example and calculate the gradients of α, μ and β by the following Python snippets:

```
def backward(self, dout)
    self.d_alpha = np.sum(dout*self.mu, axis=0)
    self.d_mu = self.alpha * dout
    self.d_beta = dout
    return dout
```

The `backward` method sequentially accepts the derivative of $\frac{\partial L}{\partial Y}$ (denoted as `dout`) from previous layers, calculates the gradients of three compensation parameters, stores them for model updating, and returns the latest derivative flows to the next layer.

5.3 Hardware Deployment

We mainly extract the computational operations of model training into three types: (1) tensor dot product, (2) tensor broadcast, and (3) tensor addition. We optimize these operations in low-level instructions that can exploit the power of fixed-point processing units. This requires us bridging the C++ level method calling with Python-level training engine. We use the light-weight Pybind tool [43] and C++ header-based Eigen [52] to embed the hardware-level matrix instructions in Python training. This kind of hybrid implementation makes our system compatible with most operating systems, including embedded Linux, macOS and Windows. Specifically, apart from the implementation of common PC and servers, we also deploy Octo on commercial AI devices, such as HUAWEI Atlas 200DK [26] and NVIDIA Jetson Xavier [27], which can utilize the dedicated INT8 neuron chips.

6 Evaluation

We evaluate Octo on real embedded platforms in the production environment. Our key insights are as follows:

#1. Does Octo preserve model quality and how is it compared to FP32 training? Octo achieves stable convergence efficiency in different benchmarks (§6.2) and holds comparable model accuracy as FP32 training (§6.3).

#2. Can Octo improve inference efficiency? Octo accelerates image processing speed, by up to 2.03× faster than FP32-based inference (§6.4).

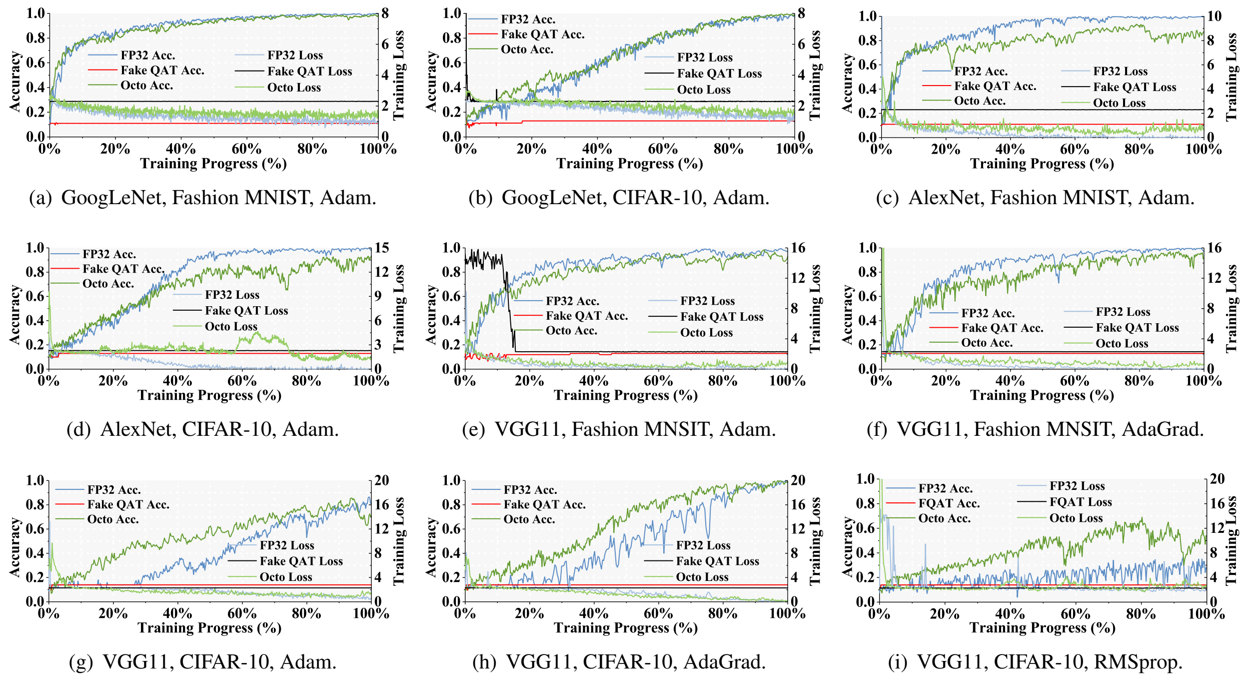
#3. How could Octo work? Octo can effectively fill the error gap caused by data quantization and maintain tensor distribution as FP32 does. Thus, Octo preserves model quality and makes training more stable (§6.5).

#4. What is the system overhead? Octo reduces the per-iteration time cost while introducing a tiny overhead of data quantization (§6.6.1). Meanwhile, Octo saves real-time memory footprint and decreases the peak memory usage, by up to 3.37× lower than FP32 training (§6.6.2).

6.1 Methodology

Testbed Setup. Considering the on-device learning characteristics, we focus on the experimental results on embedded devices. We deploy Octo on HUAWEI Atlas 200DK [26] based on Ascend 310 AI processors [13] and NVIDIA Jetson Xavier [27] equipped with dedicated INT8 neuron chips. All these devices are operated with the Ubuntu 18.04 LTS system with GNU/Linux 4.15.0-118-generic kernel.

Benchmarks. We use image classification tasks as our benchmark, based on the training of GoogLeNet [50], AlexNet



(a) GoogLeNet, Fashion MNIST, Adam. (b) GoogLeNet, CIFAR-10, Adam. (c) AlexNet, Fashion MNIST, Adam.

(d) AlexNet, CIFAR-10, Adam. (e) VGG11, Fashion MNSIT, Adam. (f) VGG11, Fashion MNSIT, AdaGrad.

(g) VGG11, CIFAR-10, Adam. (h) VGG11, CIFAR-10, AdaGrad. (i) VGG11, CIFAR-10, RMSprop.

Figure 6: Training convergence efficiency using different benchmarks and optimizers.

	FP32 Acc. (%)	Octo Acc. (%)	Acc. Degradation (%)
GoogLeNet, FM	99.1 – 99.5	97.9 – 98.6	0.9 – 1.2
GoogLeNet, CF	97.8 – 99.2	97.6 – 98.8	0.2 – 0.4
AlexNet, FM	95.6 – 98.4	92.8 – 94.3	2.8 – 4.1
AlexNet, CF	91.8 – 95.2	86.1 – 87.3	5.7 – 7.9
VGG11, FM	97.5 – 98.8	94.4 – 96.5	2.3 – 3.1
VGG11, CF	97.2 – 99.5	96.5 – 98.6	0.7 – 0.9

Table 3: Comparison of FP32 and Octo's model accuracy (from MIN to MAX) with Fashion MNIST (FM) and CIFAR-10 (CF) datasets, by using AdaGrad optimizer.

[29] and VGG11 [48]. Considering the resource constraints, large-scale datasets (*e.g.,* ImageNet [10]) are not suitable for training. We choose Fashion MNIST (FM) [59], CIFAR-10 (CF) [28] to fit the tiny on-device environment. We set the mini-batch size as 50 with 100 epochs and check Octo under Adam [25], AdaGrad [12] and RMSprop [21] optimizers.

Baselines. We build two pertinent baselines: the vanilla full-precision training (FP32) and fake quantization-aware training without error compensation (Fake QAT), using the performance metrics of training efficiency, model quality, image processing throughput and system overhead.

6.2 Training Efficiency and Quality

Maintaining convergence efficiency is one of the most crucial metrics in INT8 training design, we compare the model accuracy and training loss, by using FP32, Fake QAT and Octo training under different benchmarks in Figure 6. We can observe that Fake QAT fails to converge, where the curves of loss (in black) and accuracy (in red) almost remain unchanged. This phenomenon indicates that putting tensor arithmetic (*e.g.,* convolutional and affine operations) between quantization and dequantization will incur huge computational errors, thus Fake QAT is not suitable for hardware-level INT8 acceleration in practice. In contrast, Octo (in green) obtains comparable accuracy as the FP32 (in blue) training, holding a just slight degradation in most cases (*e.g.,* Figure 6(a)-6(f)). Specifically, as to the training of the deep VGG11 model, Octo even achieves faster convergence speed over FP32 with close final model accuracy (*e.g.,* Figure 6(g)). Training curves using other optimizers (*e.g.,* Figure 6(h) and 6(i)) also confirm this property because the compensation layers inside Octo can serve as specific 'noise' to help optimizers escape saddle points. Also, the PRC method bounds the gradients in a smoother distribution, making models update more stably. Overall, the accuracy comparison and degradation gap are summarized in Table 3, where Octo preserves model quality as FP32 does and is sufficient for INT8 on-device training.

6.3 Ablation Study

We conduct the ablation study to inspect how much improvement is achieved by LAC and PRC separately. The experiments are based on the training of AlexNet model on CIFAR-10 and Fashion MNIST datasets by using Adam optimizer. Note that the LAC and PRC operations are added in CONV2, CONV3 and CONV4. We compare the average model accuracy under different training configurations in Table 4. We can observe that simply adopting INT8 data representation

	Configuration	Acc. (%)	Gap over FP32 (%)
Fashion MNIST	FP32	97.1	0
	INT8	13	−84.1
	INT8 + LAC	90.4	−6.7
	INT8 + PRC	14.8	−82.3
	INT8 + LAC + PRC	93.6	−3.5
CIFAR-10	FP32	93.5	0
	INT8	11	−82.5
	INT8 + LAC	85.2	−8.3
	INT8 + PRC	12.1	−81.4
	INT8 + LAC + PRC	86.7	−6.8

Table 4: Comparison of average model accuracy under different training configurations.

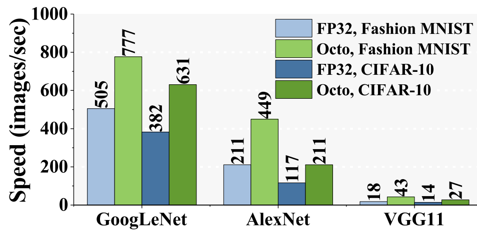


Figure 7: Impage processing throughput, *i.e.,* images per second, by using FP32 and Octo trained models.

into training will significantly deteriorate the model accuracy due to large errors of the quantized dot product. Enabling LAC operation can fill this error gap by learning parameters of an affine approximation. Note that directly using PRC on vanilla INT8 training without LAC cannot eliminate the deviation of intermediate results, thus still incurring a great degradation of accuracy. However, PRC can further improve the final accuracy when LAC has been enabled, where the clipped gradients can restrict the training process in a proper convergence boundary.

6.4 Image Processing Throughput

Apart from the training efficiency, we also inspect Octo's inference performance. We measure the image processing speed, *i.e.,* image count per second, by using the models trained by FP32 and Octo. As model parameters and tensor arithmetic are converted in INT8 format, Octo can effectively reduce I/O bandwidth and computational pressure. Therefore, Octo improves the image processing throughput, by up to 2.03×, on average, over FP32 model based inference. This property is significantly meaningful for on-device learning as we can reduce inference latency and improve user experience.

6.5 Octo Deep Dive

Observing the improvement of training and inference performance, we wonder how could Octo achieve this. Here, we give deep insights into how Octo compensates for quantization error and preserves model accuracy. We visualize the intermediate tensor distribution of a CONV layer's output by using FP32, Fake QAT and Octo. The data are collected from training an 8-layer deep CNN with CIFAR-10 after 80 iterations. Fake QAT holds a distinct distribution compared with FP32 because conducting dot products in the quantized domain will incur significant error to the final output. In contrast, the compensation layers inside Octo can fill the error gap and achieve similar distribution as FP32 does. Therefore, the model accuracy is maintained. Also, the PRC method in backward pass bounds derivative domains and smooths the tensor distribution, making the training more stable.

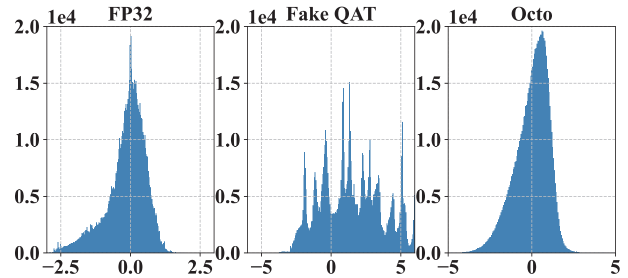
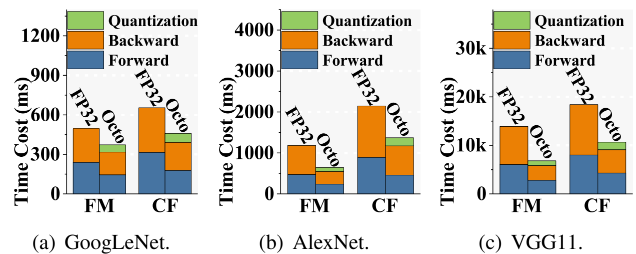


Figure 8: The visualization of tensor distribution of CONV2's output, under FP32, Fake QAT and Octo's training. Octo preserves similar distribution as FP32 while Fake QAT cannot.



(a) GoogLeNet. (b) AlexNet. (c) VGG11.

Figure 9: We inspect the details of computational time cost from 300 iterations and measure the overhead of Octo's data quantization on average, when training Fashion MNIST (FM) and CIFAR-10 (CF) dataset.

6.6 System Overhead

We compare Octo's system overhead with FP32 training, following two key metrics of embedded platforms: computational time cost (§6.6.1) and memory footprint (§6.6.2).

6.6.1 Computational Time Cost

We measure the average computational time cost of different stages in each iteration, as depicted in Figure 9. Although Octo introduces extra overhead of data quantization, about 17.21% increase on average, it reduces the completion time of both forward and backward passes by using INT8-based processing. Overall, Octo holds shorter per-iteration time, by up to 1.73× faster, on average, over vanilla FP32 training. Therefore, we believe introducing data quantization for on-device training is meaningful.

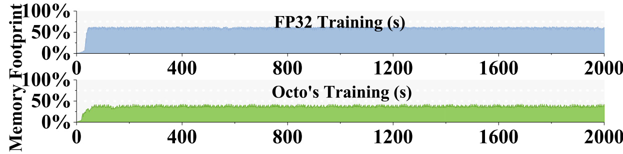


Figure 10: Octo effectively reduces real-time memory footprint over FP32 training.

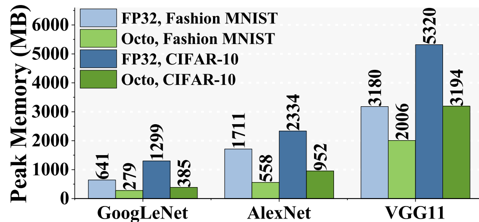


Figure 11: Peak memory usage of FP32 and Octo training.

6.6.2 Memory Footprint

We compare the memory usage under FP32 and Octo's training, respectively. As shown in Figure 10, we monitor the real-time memory footprint when training GoogLeNet with CIFAR-10 in 2000 seconds. The FP32 training requires about 61.29% memory usage on average, while Octo only requires 40.13%. More precisely, we compare their peak memory usage of different benchmarks in Figure 11. Due to the INT8-based parameters and gradient quantization, Octo can effectively reduce peak memory usage, by up to 3.37× lower than FP32 training. Such a reduction makes it possible to deploy VGG-like models.

7 Discussion

In this section, we will discuss the potential usages, extensions and limitations of Octo.

Deployment on edge devices. We mainly deploy Octo on two kinds of edge devices: (1) Atlas 200DK developer board integrating Ascend 310 AI processors and (2) NVIDIA Jetson Xavier equipped with INT8 neural chips. These devices support hardware-level INT8 operations, thus making Octo truly exploiting the power of quantization-aware training. Note that Octo is a cross-platform system and its INT8 quantization algorithm can apply to most existing ML frameworks. For example, it is possible to extend the TensorRT engine [53] to enable truly INT8 training on NVIDIA Pascal GPUs, rather than current post-quantization or inference-only INT8 usage.

Comparison with existing work. We have to build Octo from scratch due to the following limitations of existing quantization methods. PACT [7] is designed for quantized inference by optimizing the clipping range of activations in forward pass, without the consideration of gradient calculation in backward pass. Fake QAT [23] needs to pre-train a full-precision model and uses INT8 fine-tuning to preserve quantized model quality. Fake QAT only simulates INT8 calculations in forward pass and cannot bring actual acceleration. Directly using Fake QAT to INT8 training will cause large errors of dot product and cannot guarantee model convergence. Unified QAT [67] enables INT8 training by adjusting gradients in backward pass. However, it relies on the calculation of both quantized and full-precision gradients, as well as plenty of exponent arithmetic. This computational overhead requires the support of GPUs and is not feasible to the device's resource-constrained environment.

Extensions to other types of models and layers. Octo also supports other layers and models (*e.g.,* FC layers for saving more memory and RNNs for time-series prediction) because our basic optimization targets are tensor-level dot product and broadcast operations, which are prevalent in modern neural networks. Although Octo can reduce computational overhead and save memory footprint for most CNN models, we admit that Octo is just a first step to exploit the feasibility of deploying INT8 training on devices. Some complicated scenarios, such as conducting NLP on large-scale datasets or detecting real-time objects with high frame rates may still need supplementary methods, where the construction of compensation layers, clipping strategy of gradients, regularization terms of loss function should be carefully designed.

8 Conclusion

This work demonstrates that introducing INT8 quantization to training is a feasible way to implement on-device learning in practice. To truly enable hardware-level INT8 acceleration, the key of designing an efficient quantization-aware training method is to fill the error gap of dot products. This target is achieved by optimizing data quantization in both forward and backward passes, via the proposed *Loss-aware Compensation* (LAC) and *Parameterized Range Clipping* (PRC) methods, respectively. Specifically, we design a novel compensation layer to adjust the quantized output and smooth the model update procedure. Our method is implemented in Octo, a cross-platform system for tiny on-device learning. Evaluations show that Octo holds higher training efficiency over state-of-the-art quantization training methods and preserves comparable model quality as full-precision training.

Acknowledgements

This research was supported by the funding from Hong Kong RGC Research Impact Fund (RIF) with the Project No. R5060-19 and R5034-18, General Research Fund (GRF) with the Project No. 152221/19E and 15220320/20E, Collaborative Research Fund (CRF) with the Project No. C5026-18G, the National Natural Science Foundation of China (61872310), Shenzhen Science and Technology Innovation Commission (R2020A045), and Fundamental Research Funds for the Central Universities (B210202079).

References

[1] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *ACM J. Emerg. Technol. Comput. Syst.*, 13(3):32:1–32:18, 2017.

[2] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, pages 5151–5159, Montréal, Canada, 2018.

[3] Ron Banner, Yury Nahshan, Elad Hoffer, and Daniel Soudry. ACIQ: analytical clipping for integer quantization of neural networks. *arXiv preprint*, abs/1810.05723, 2018.

[4] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *Proceedings of the International Conference on Learning Representations (ICLR)*, Addis Ababa, Ethiopia, 2020.

[5] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. Tinytl: Reduce memory, not parameters for efficient on-device learning. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 578–594, Carlsbad, USA, 2018.

[7] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. PACT: parameterized clipping activation for quantized neural networks. *arXiv preprint*, abs/1805.06085, 2018.

[8] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel P. Kuksa. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537, 2011.

[9] MNIST Dataset. `http://yann.lecun.com/exdb/mnist/`, 2013.

[10] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. Imagenet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, Miami, USA, 2009.

[11] Xin Dong, Shangyu Chen, and Sinno Jialin Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, pages 4857–4867, Long Beach, USA, 2017.

[12] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, 2011.

[13] Ascend 310 AI Processor: Energy efficiency and high integration for edges. `https://e.huawei.com/se/products/cloud-computing-dc/atlas/ascend-310`, 2021.

[14] Biyi Fang, Xiao Zeng, and Mi Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 115–127, New Delhi, India, 2018.

[15] TensorFlow Lite: ML for Mobile and Edge Devices. `https://www.tensorflow.org/lite`, 2020.

[16] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, pages 1379–1387, Barcelona, Spain, 2016.

[17] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the International Conference on Machine Learning (ICML)*, volume 37, pages 1737–1746, Lille, France, 2015.

[18] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *Proceedings of the International Conference on Learning Representations (ICLR)*, San Juan, Puerto Rico, 2016.

[19] Song Han, Jeff Pool, Sharan Narang, Huizi Mao, Enhao Gong, Shijian Tang, Erich Elsen, Peter Vajda, Manohar Paluri, John Tran, Bryan Catanzaro, and William J. Dally. DSD: dense-sparse-dense training for deep neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, Toulon, France, 2017.

[20] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. *arXiv preprint*, abs/1506.02626, 2015.

[21] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Rmsprop: Divide the gradient by a running average of its recent magnitude. In *COURSERA: Neural Networks for Machine Learning, Lecture 6.5*, 2012.

[22] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint*, abs/1704.04861, 2017.

[23] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry 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 (CVPR)*, pages 2704–2713, Salt Lake City, USA, 2018.

[24] Sangil Jung, Changyong Son, Seohyung Lee, JinWoo Son, Jae-Joon Han, Youngjun Kwak, Sung Ju Hwang, and Changkyu Choi. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4350–4359, Long Beach, USA, 2019.

[25] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations (ICLR)*, San Diego, USA, 2015.

[26] Atlas 200DK AI Developer Kit. https://e.huawei.com/us/products/cloud-computing-dc/atlas/atlas-200, 2020.

[27] Jetson AGX Xavier Developer Kit. https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit, 2021.

[28] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 2012.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, pages 1106–1114, Lake Tahoe, USA, 2012.

[30] Liangzhen Lai, Naveen Suda, and Vikas Chandra. CMSIS-NN: efficient neural network kernels for arm cortex-m cpus. *arXiv preprint*, abs/1801.06601, 2018.

[31] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. A multilinear singular value decomposition. *SIAM J. Matrix Anal. Appl.*, 21(4):1253–1278, 2000.

[32] Yuhang Li, Xin Dong, and Wei Wang. Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, Addis Ababa, Ethiopia, 2020.

[33] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. Mcunet: Tiny deep learning on iot devices. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[34] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. In *Proceedings of the International Conference on Learning Representations (ICLR)*, San Juan, Puerto Rico, 2016.

[35] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. In *Proceedings of the European Conference on Computer Vision (ECCV)*, volume 9905, pages 21–37, Amsterdam, 2016.

[36] Christos Louizos, Matthias Reisser, Tijmen Blankevoort, Efstratios Gavves, and Max Welling. Relaxed quantization for discretized neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, New Orleans, USA, 2019.

[37] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J. Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint*, abs/1705.08922, 2017.

[38] Akhil Mathur, Nicholas D. Lane, Sourav Bhattacharya, Aidan Boran, Claudio Forlivesi, and Fahim Kawsar. Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 68–81, Niagara Falls, USA, 2017.

[39] Bradley McDanel, Sai Qian Zhang, H. T. Kung, and Xin Dong. Full-stack optimization for accelerating cnns using powers-of-two weights with FPGA validation. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 449–460, Phoenix, USA, 2019.

[40] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 54, pages 1273–1282, Fort Lauderdale, USA, 2017.

[41] PyTorch Mobile. https://pytorch.org/mobile/home/, 2020.

[42] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. In *Proceedings*

of the IEEE/CVF International Conference on Computer Vision (ICCV), pages 1325–1334, Seoul, Korea (South), 2019.

[43] Pybind11: Seamless operability between C++11 and Python. https://pybind11.readthedocs.io/en/stable/index.html, 2020.

[44] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young-ri Choi. Hetpipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 307–321, 2020.

[45] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 16–29, Huntsville, Canada, 2019.

[46] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: online learning of social representations. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 701–710, New York, USA, 2014.

[47] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, volume 9908, pages 525–542, Netherlands, 2016.

[48] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the International Conference on Learning Representations (ICLR)*, San Diego, USA, 2015.

[49] Pierre Stock, Armand Joulin, Rémi Gribonval, Benjamin Graham, and Hervé Jégou. And the bit goes down: Revisiting the quantization of neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, Addis Ababa, Ethiopia, 2020.

[50] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, Boston, USA, 2015.

[51] Apple Face ID Advanced Technology. https://support.apple.com/en-us/HT208108, 2020.

[52] Eigen: A C++ template library for linear algebra. https://eigen.tuxfamily.org/index.php, 2020.

[53] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt, 2021.

[54] Mineto Tsukada, Masaaki Kondo, and Hiroki Matsutani. A neural network-based on-device learning anomaly detector for edge devices. *IEEE Trans. Computers*, 69(7):1027–1044, 2020.

[55] Li Wan, Matthew D. Zeiler, Sixin Zhang, Yann LeCun, and Rob Fergus. Regularization of neural networks using dropconnect. In *Proceedings of the International Conference on Machine Learning (ICML)*, volume 28, pages 1058–1066, Atlanta, USA, 2013.

[56] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8612–8620, Long Beach, USA, 2019.

[57] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, volume 11217, pages 420–436, Munich, Germany, 2018.

[58] Xundong Wu, Yong Wu, and Yong Zhao. Binarized neural networks on the imagenet classification task. *arXiv preprint*, abs/1604.03058, 2016.

[59] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint*, abs/1708.07747, 2017.

[60] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 595–610, Carlsbad, USA, 2018.

[61] Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric P. Xing. Distributed machine learning via sufficient factor broadcasting. *arXiv preprint*, abs/1511.08486, 2015.

[62] Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric P. Xing. Lighter-communication distributed machine learning via sufficient factor broadcasting. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, New York, USA, 2016.

[63] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 181–193, Santa Clara, USA, 2017.

[64] Huaqing Zhang, Jian Wang, Zhanquan Sun, Jacek M. Zurada, and Nikhil R. Pal. Feature selection for neural networks using group lasso regularization. *IEEE Trans. Knowl. Data Eng.*, 32(4):659–673, 2020.

[65] Qihua Zhou, Zhihao Qu, Song Guo, Boyuan Luo, Jingcai Guo, Zhenda Xu, and R. Akerkar. On-device learning systems for edge intelligence: A software and hardware synergy perspective. *IEEE Internet of Things Journal*, pages 1–1, 2021.

[66] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proc. IEEE*, 107(8):1738–1762, 2019.

[67] Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. Towards unified INT8 training for convolutional neural network. *arXiv preprint*, abs/1912.12607, 2019.