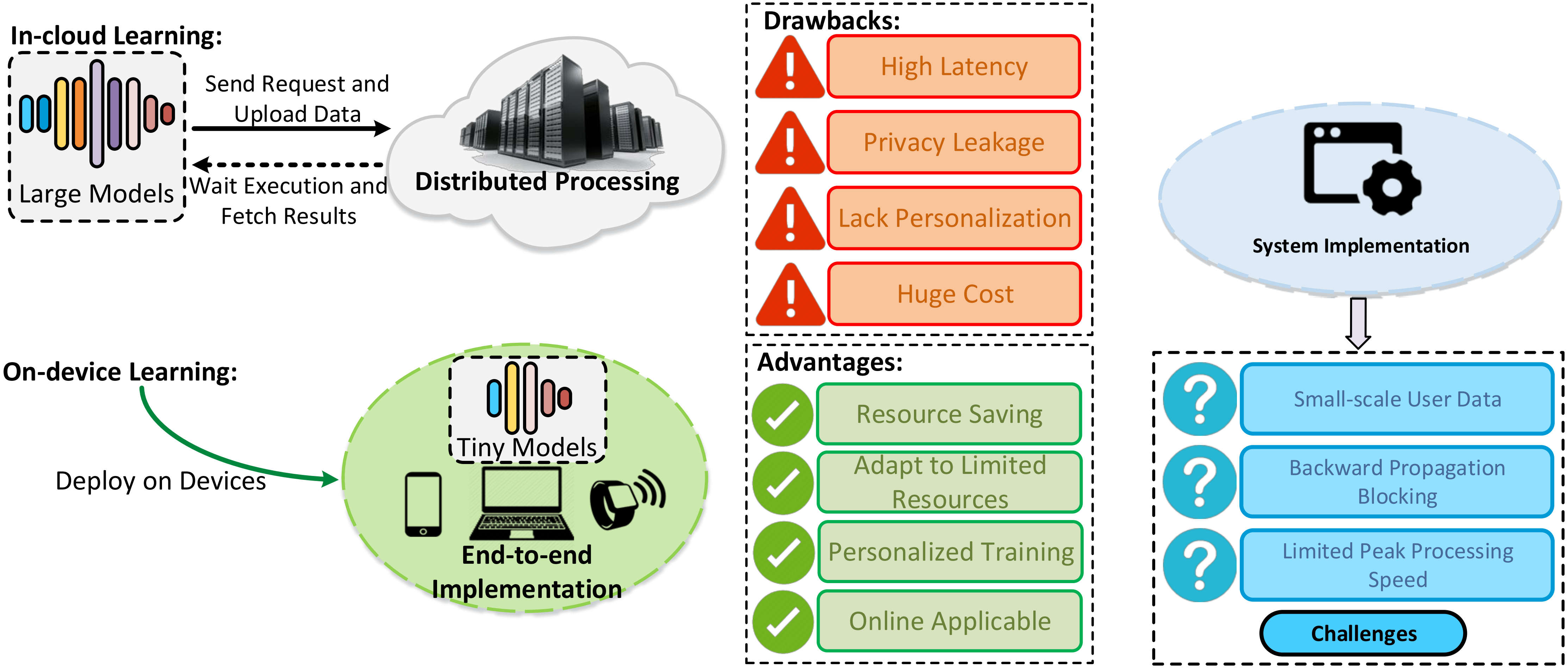
Octo: INT8 Training with Loss-aware Compensation and Backward Quantization for Tiny On-device Learning

Qihua Zhou[1], Song Guo[1], Zhihao Qu[2], Jingcai Guo[1], Zhenda Xu[1], Jiewei Zhang[1], Tao Guo[1], Boyuan Luo[1], Jingren Zhou[3]

[1]The Hong Kong Polytechnic University, [2]Hohai University, [3]Alibaba Group

Rise of On-device Learning



Question: how to overcome the challenges of resource constraints?

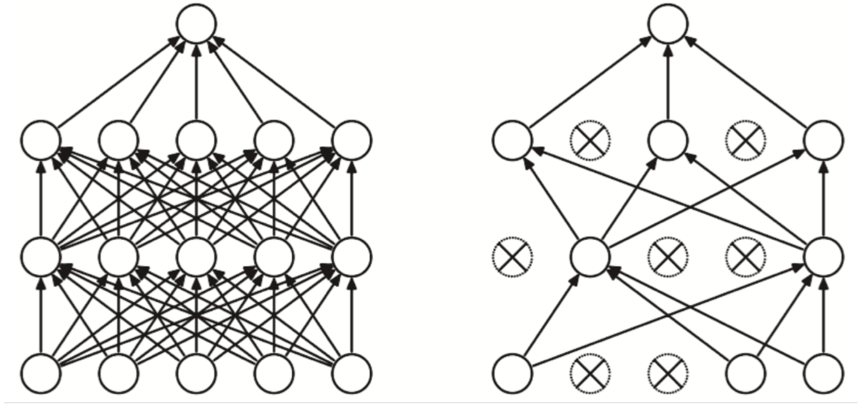
Solution: enable quantization-aware training.

Common Compression Methods

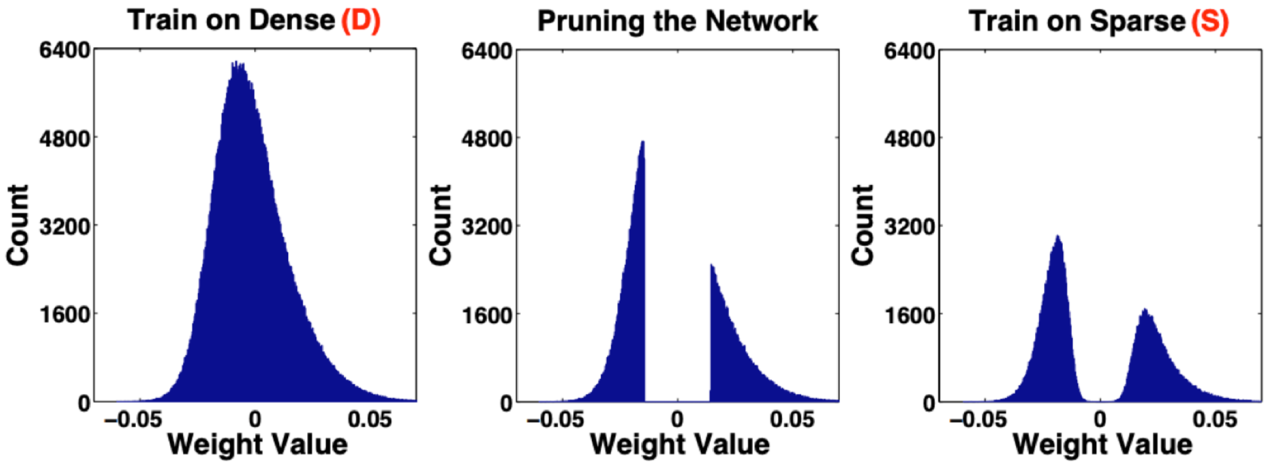
(1) Low Rank Factorization

$$m\left\{\overbrace{\begin{bmatrix}\quad A \quad\end{bmatrix}}^{n}\right. \approx m\left\{\overbrace{\begin{bmatrix} X \end{bmatrix}}^{k}\right.\left.\overbrace{\begin{bmatrix} \quad Y \quad \end{bmatrix}}^{n}\right\}k$$

(2) Model Pruning



(3) Network Sparsification

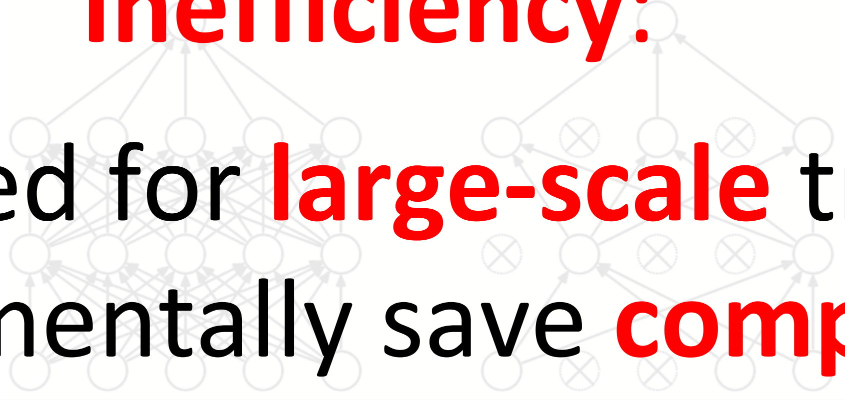


Common Compression Methods

(1) Low Rank Factorization

$$m\left\{\overbrace{\begin{bmatrix} & A & \end{bmatrix}}^{n}\right. \approx m\left\{\overbrace{\begin{bmatrix} X \end{bmatrix}}^{k}\right. \overbrace{\begin{bmatrix} & Y & \end{bmatrix}}^{n}\Big\}k$$

(2) Model Pruning

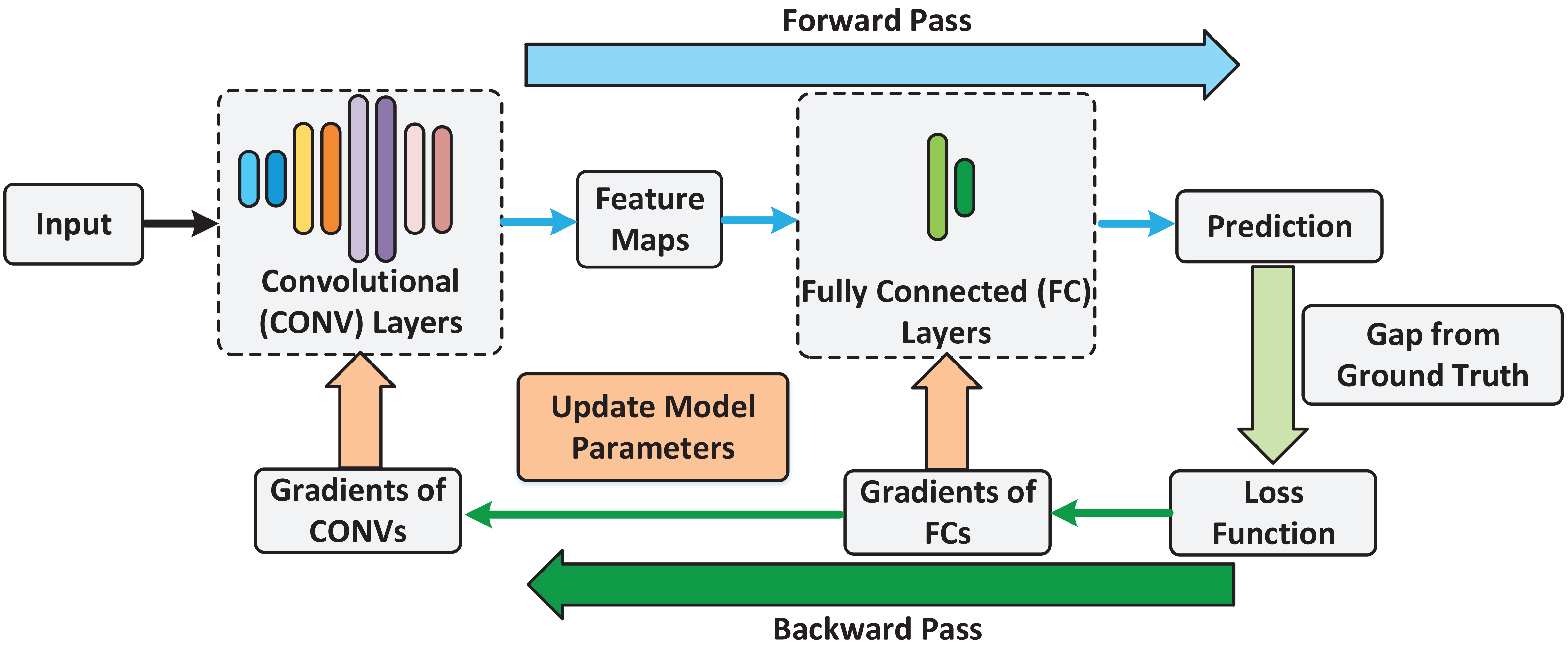


Inefficiency:

- Mainly designed for **large-scale** training tasks
- Cannot fundamentally save **computational cost**

(3) Network Sparsification

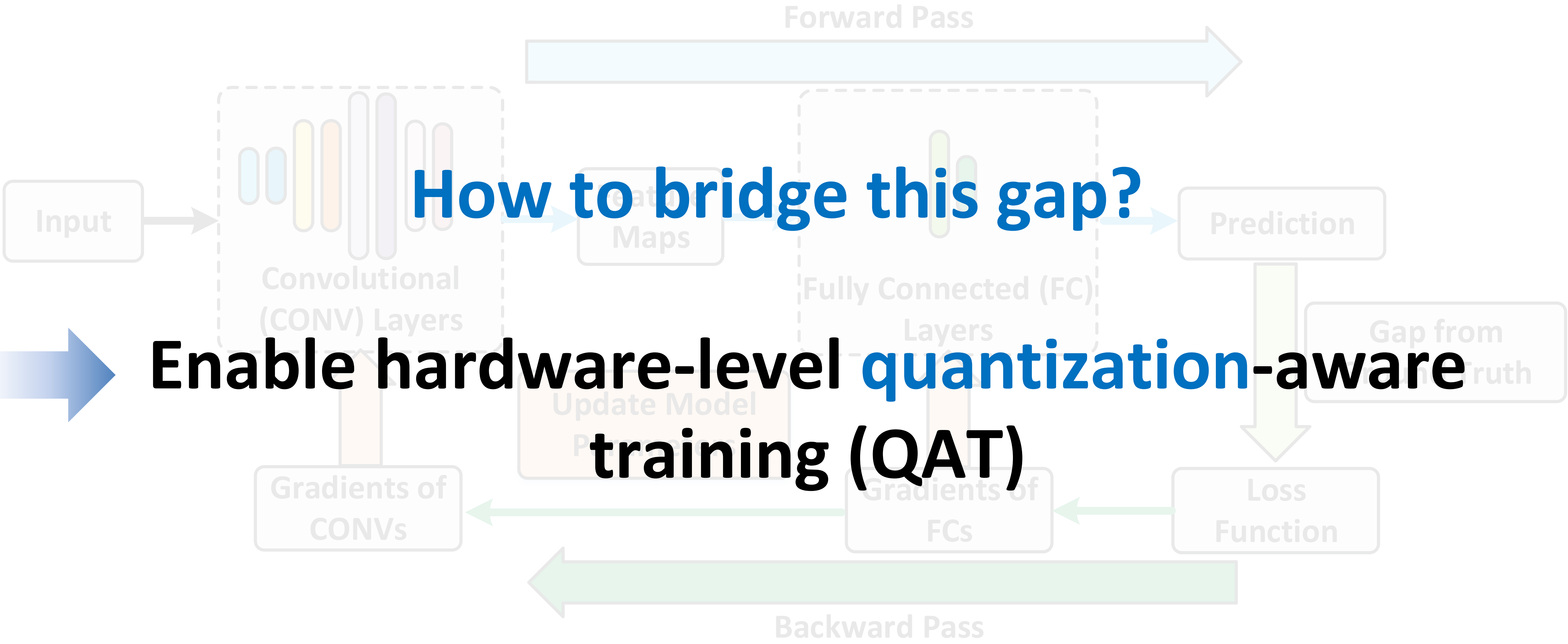
The Workflow of DNN Training



What are the **fundamental instructions** dominating the computational cost?

➡ **Tensor Dot Product** (e.g., FP: CONV, Affine, BP: Derivative) based on **FP32** format

The Workflow of DNN Training



How to bridge this gap?

Enable hardware-level quantization-aware training (QAT)

What are the fundamental instructions dominating the computational cost?

Tensor Dot Product (e.g., FP: CONV, Affine, BP: Derivative) based on FP32 format

Bridge the Gap: Data Quantization

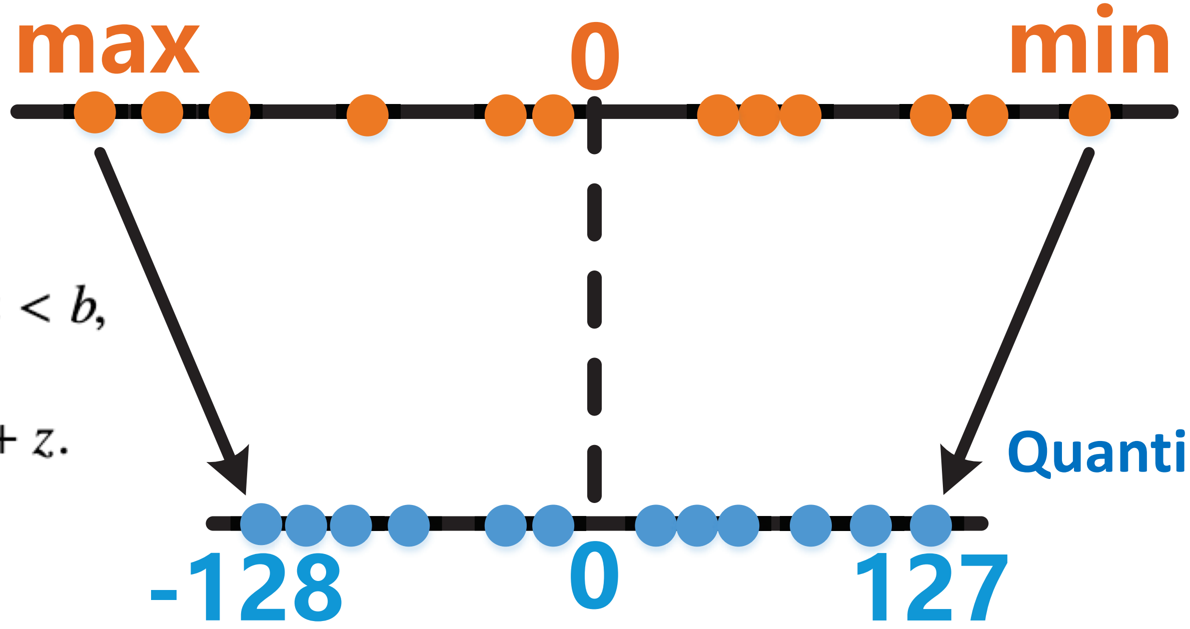
Represent a number via low bit width

➔ Example: from 32-bit floating-point (**FP32**) to 8-bit fixed-point numbers (**INT8**)

Formulation:

$$s = \texttt{scale}(a, b, n) = \frac{b-a}{2^n - 1}, a < b,$$

$$q = \texttt{round}\left(\frac{\texttt{clip}(r)}{\texttt{scale}(a, b, n)}\right) + z.$$



Quantization Level: 2^8=256

Core Operations: (1) Number Discretization and (2) Domain Transformation.

Why We Need Quantization?

The Property of Quantization

- Quantization enables compression (for memory footprint) and acceleration (for computation) in bit level

➔ enables on-device learning

- Quantization is more hardware friendly for both generic hardware (e.g., CPU/GPU) and specific chips (e.g., FPGA)

➔ suitable for the edge environment

Target

- A good quantization algorithm needs to consider model characteristics, training efficiency and hardware practicality

Potential Gains

Validation Experiments: System performance using INT8 and FP32 training

	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%

Inspiration: can we achieve the same level of FP32 training performance with only INT8 operations for common on-device learning applications (e.g., image classification)?

What about Existing Quantization Methods?

Limitations:

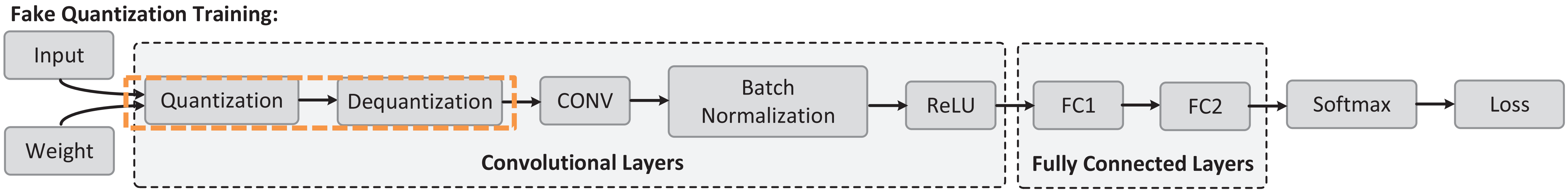
#1. Cannot apply to training process.

#2. Cannot support generic networks without specific structure design.

#3. Cannot enable hardware-level INT8 acceleration in training phase.

#4. Cannot make the gradient calibration fit on-device resource restrictions in backward pass.

Workflow of the pertinent Fake QAT:



Target: enable hardware-level INT8 training directly on devices.

Co-design of Network and Training Engine

Challenges:

#1. How to fundamentally **accelerate processing speed** on devices?
➔ **Uniform 8-bit quantization** for CONV, Affines, Activations and Gradients.

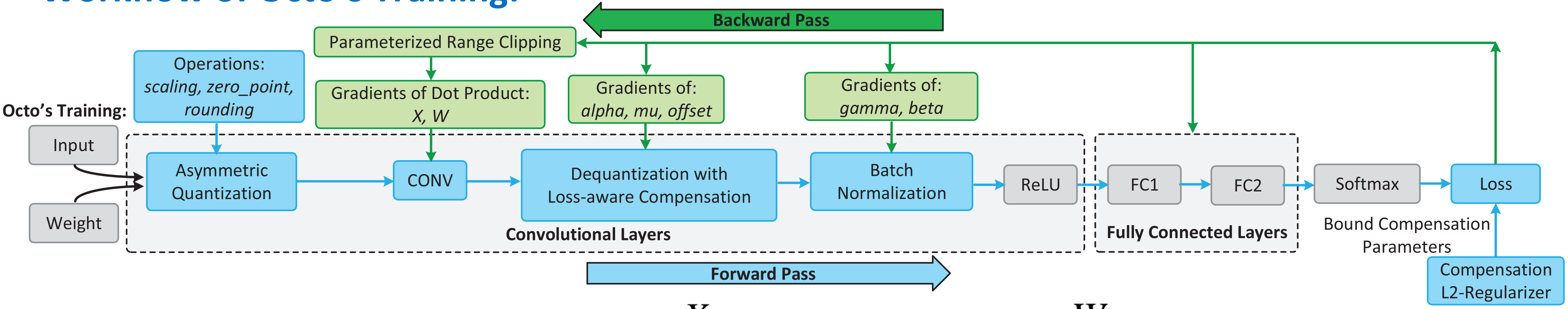
#2. How to **maintain model quality** when using INT8 quantization-aware training?
➔ **Loss-aware Compensation (LAC):** fill the error gap from quantized tensor arithmetic.
➔ **Parameterized Range Clipping (PRC):** bound the transformation domain of quantized gradients.

#3. How to **alleviate system overhead**, especially reducing memory footprint?
➔ Preserve all the **parameters and intermediate derivatives** in INT8 format with affine approximation.

#4. How to make the system **ease-of-use** and compatible with multiple platforms?
➔ Embed the **hardware-level matrix instructions** via C++ and Python **hybrid implementation**.

Our System: Octo

Workflow of Octo's Training:



Step #1 Quantization: $X_q = \mathrm{round}(\frac{X_f}{s_x} + z_x), \quad W_q = \mathrm{round}(\frac{W_f}{s_w} + z_w)$

Step #2 Dot Product: $Y_q = \mathrm{dot}(X_q, W_q)$

Step #3 Dequantization with Compensation: $Y_f = Y_q \cdot (s_x \cdot s_w) + \delta$

Analysis of Error Gap: $\delta = s_x * \Delta X \cdot W_f + \gamma$

Approximation via Affine Transformation: $\hat{\delta} = \alpha * \mu + \beta$

Handle this approximation: Loss-aware Compensation

Loss-aware Compensation

Compensation Layer: injected at the end of CONVs or FCs

Three Learnable Parameters:

Scalar: scaling factor

Tensor: compensation offset

$$\hat{\delta} = \boxed{\alpha * \mu} + \boxed{\beta}$$

Bit-wise Shifting

Approximated Compensation Term

Tensor: distribution expectation

Element Broadcasting

L2-Regularization of Compensation Parameters:

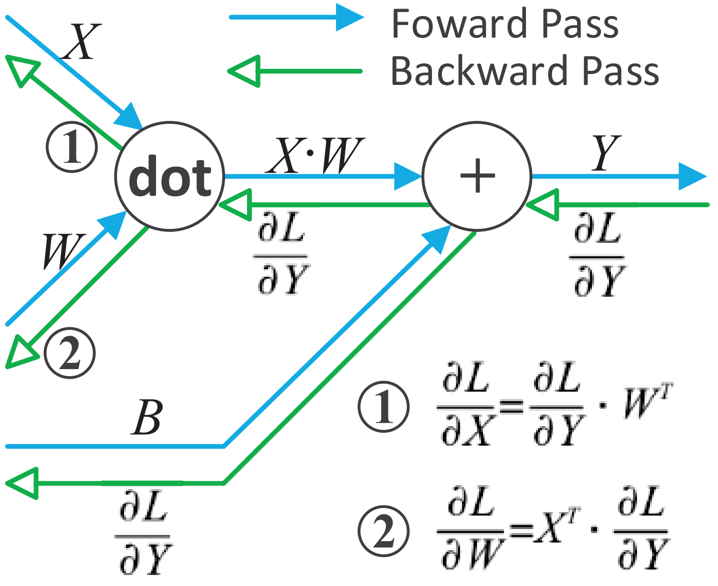
$$L = \underbrace{-\frac{1}{N}\sum^{n}\sum^{k} t_{nk}\log y_{nk}}_{(1)} + \underbrace{\frac{1}{2}\lambda(\mu^2 + \beta^2)}_{(2)},$$

Primary Cross-entropy Error:
Measure difference between prediction y and ground truth t

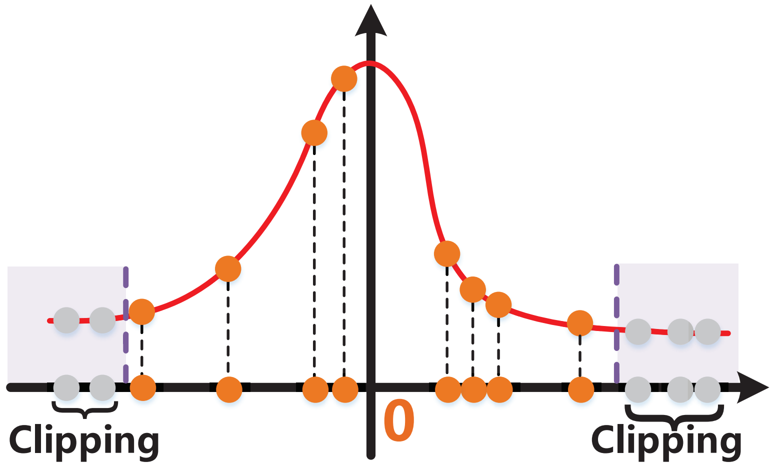
L2-regularizer:
reflect compensation performance based on μ and β

Backward Quantization

Calculation of Derivative Flows for weights W and features X :



Parameterized Range Clipping:



$$\texttt{clip}(M) \in [-a, a],$$
$$a = \min\{|\min(M)|, \max(M)\}.$$

Gradient Recovery:

$$\frac{\partial L}{\partial X_f} = \texttt{dot}\left(\frac{\partial L}{\partial Y_q}, W_q^\top\right) \cdot (s_y s_w),$$
$$\frac{\partial L}{\partial W_f} = \texttt{dot}\left(X_q^\top, \frac{\partial L}{\partial Y_q}\right) \cdot (s_x s_y),$$

Evaluation Setup

Platforms:

- HUAWEI Atlas 200 DK: Ascend 310 AI processor
- NVIDIA Jetson Xavier DK: 6-core Carmel ARM® CPU
- 8 GB RAM, 51.2 GB/s bandwidth

Benchmarks

- Model: GoogleNet, AlexNet, VGG11
- Dataset: CIFAR-10, Fashion MNIST
- Optimizer: Adam, Adagrad, RMSprop

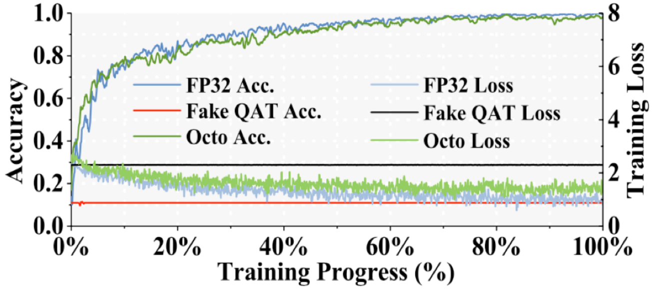
Baselines

- FP32
- Fake QAT

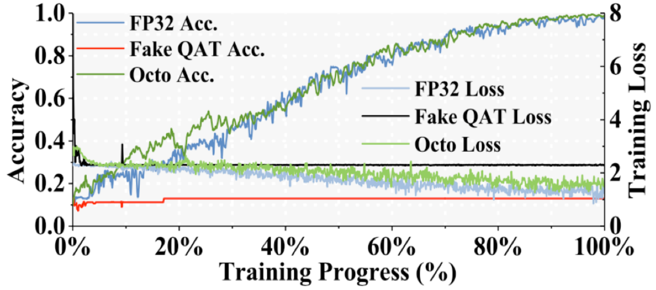
HUAWEI Atlas 200 DK


NVIDIA Jetson Xavier DK

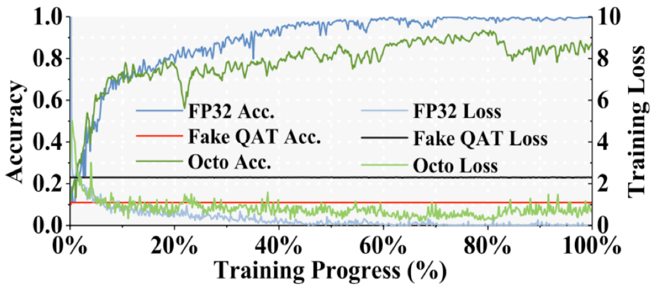
Convergence Results



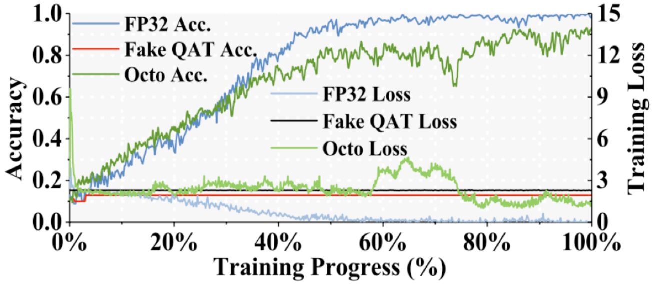
(a) GoogLeNet, Fashion MNIST, Adam.



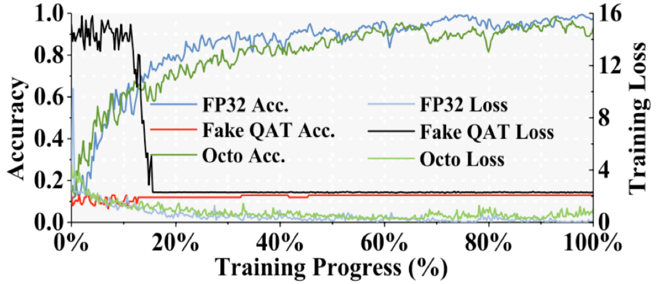
(b) GoogLeNet, CIFAR-10, Adam.



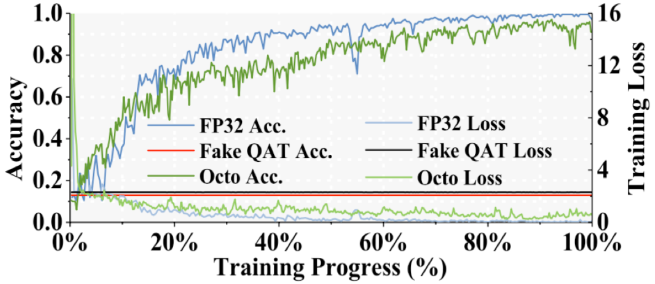
(c) AlexNet, Fashion MNIST, Adam.



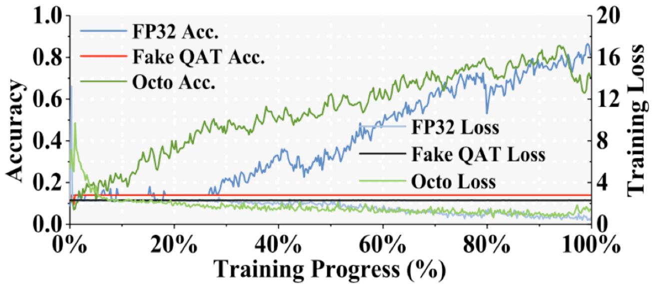
(d) AlexNet, CIFAR-10, Adam.



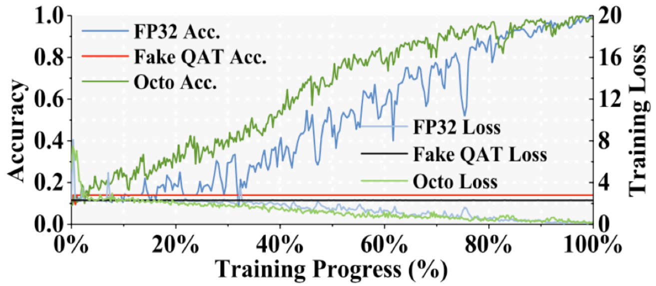
(e) VGG11, Fashion MNSIT, Adam.



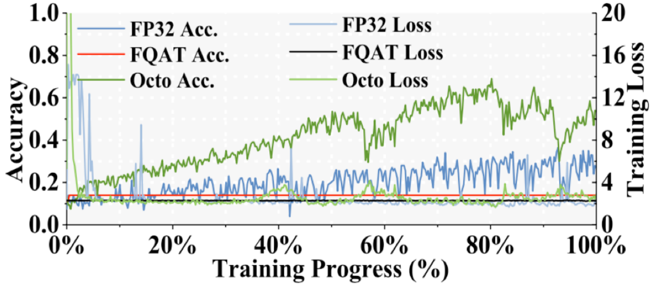
(f) VGG11, Fashion MNSIT, AdaGrad.



(g) VGG11, CIFAR-10, Adam.



(h) VGG11, CIFAR-10, AdaGrad.



(i) VGG11, CIFAR-10, RMSprop.

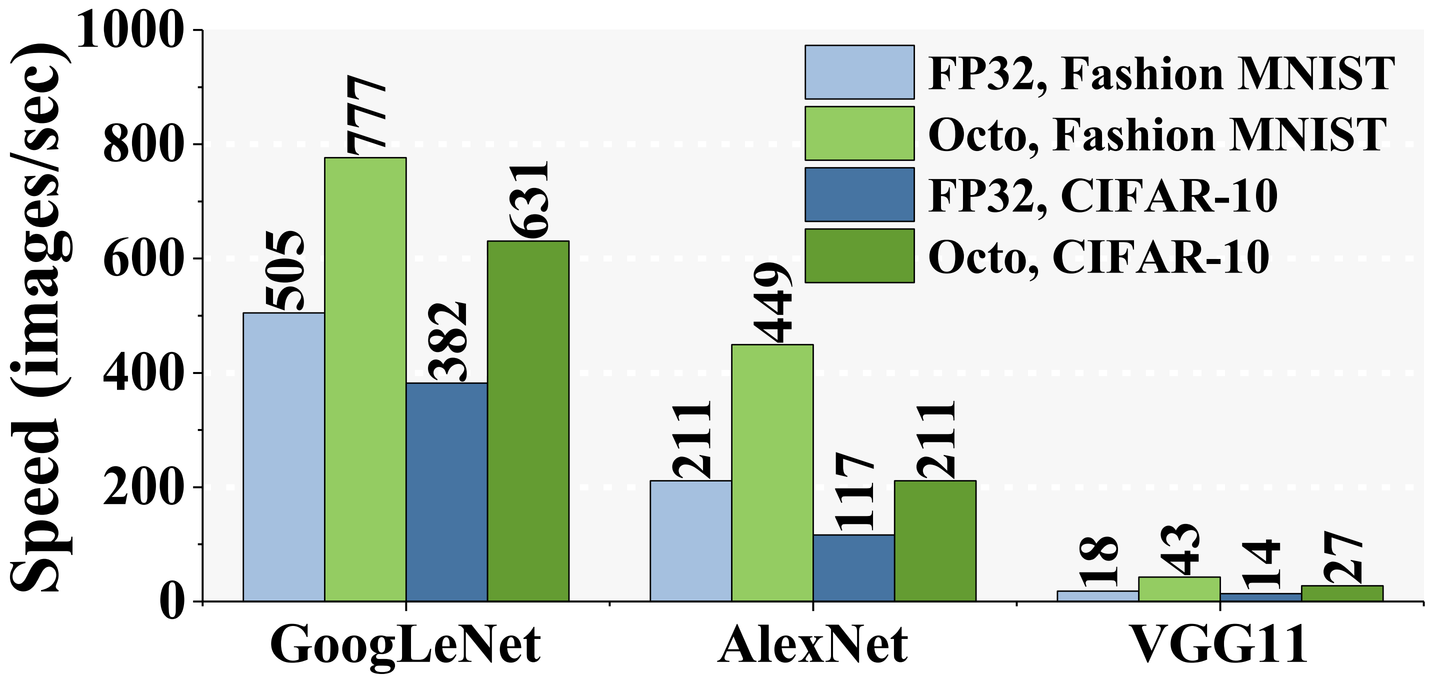
Octo preservers model accuracy as FP32 does while Fake QAT fails to converge

Ablation Study: Impact of LAC and PRC

	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

Image Processing Throughput

Images Counted Per Second:



Octo vs. FP32 Inference:

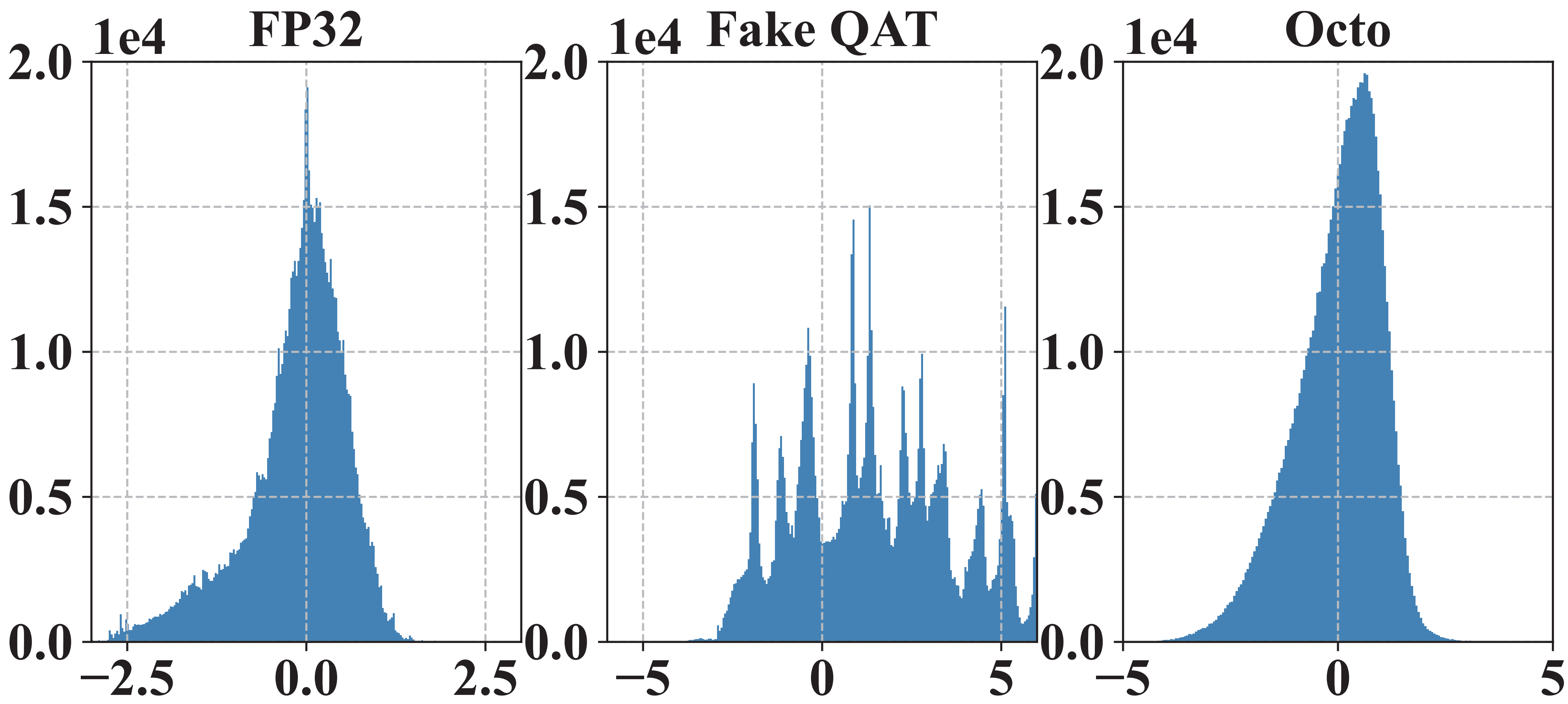
- 2.03× speedups on average

Meaningful to On-device Learning:

- Reduce inference latency
- Improve user experience

Deep Insight of Feature Distribution

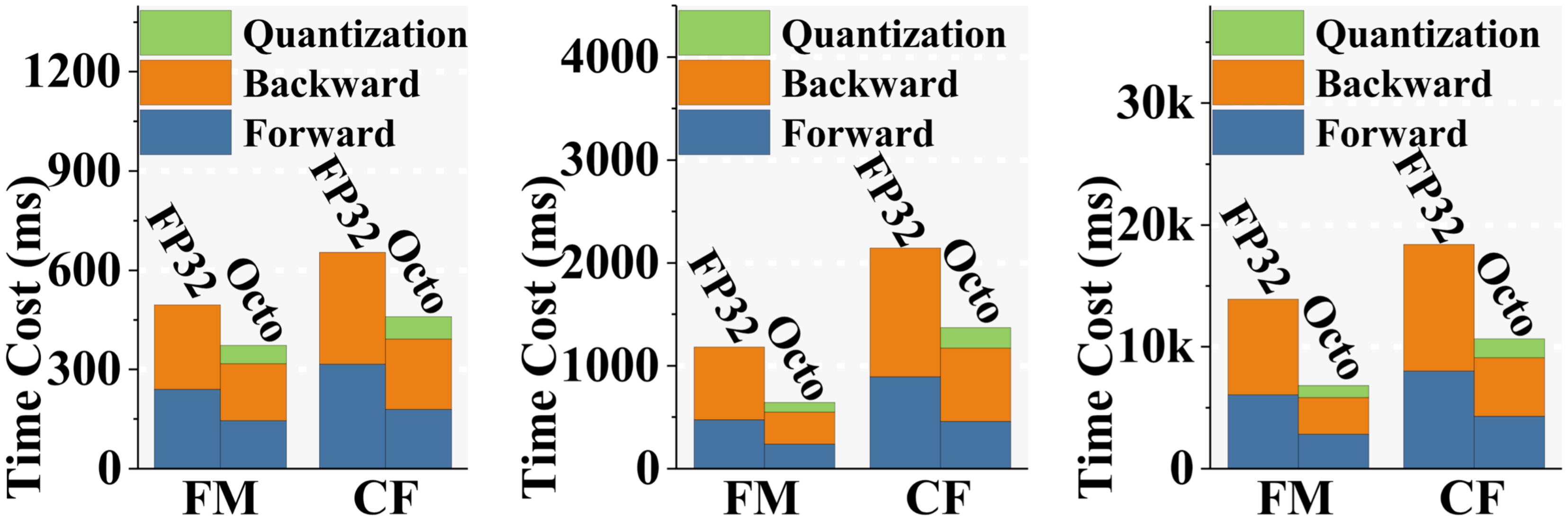
Visualization of Intermediate Feature Distribution:



Maintain Model Accuracy: Octo's compensation layers fills the error gap and achieves similar distribution as FP32 does, while Fake QAT cannot.

System Overhead

Computational Time Cost:



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

Quantization Overhead:

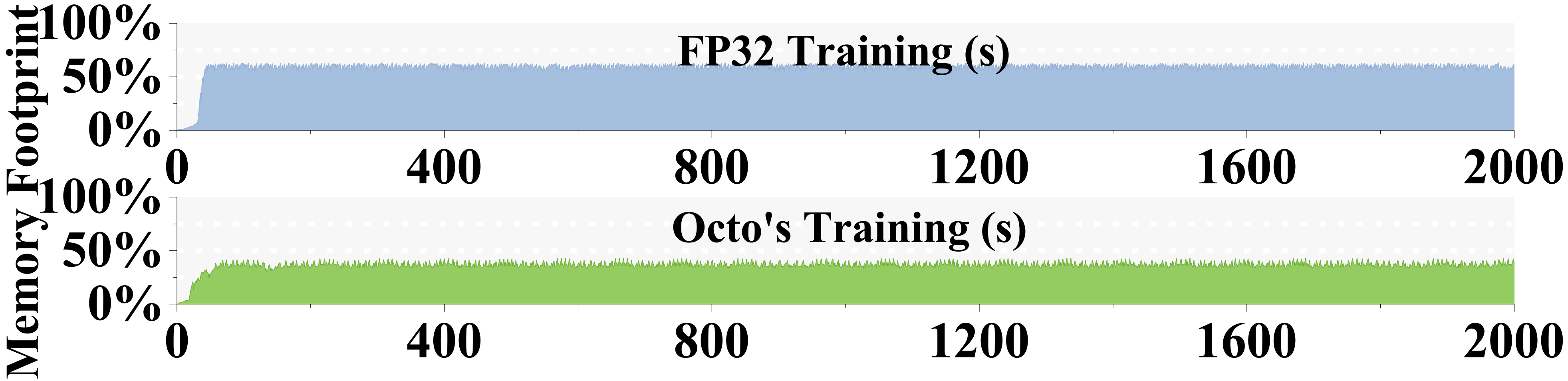
- Lower than 15%

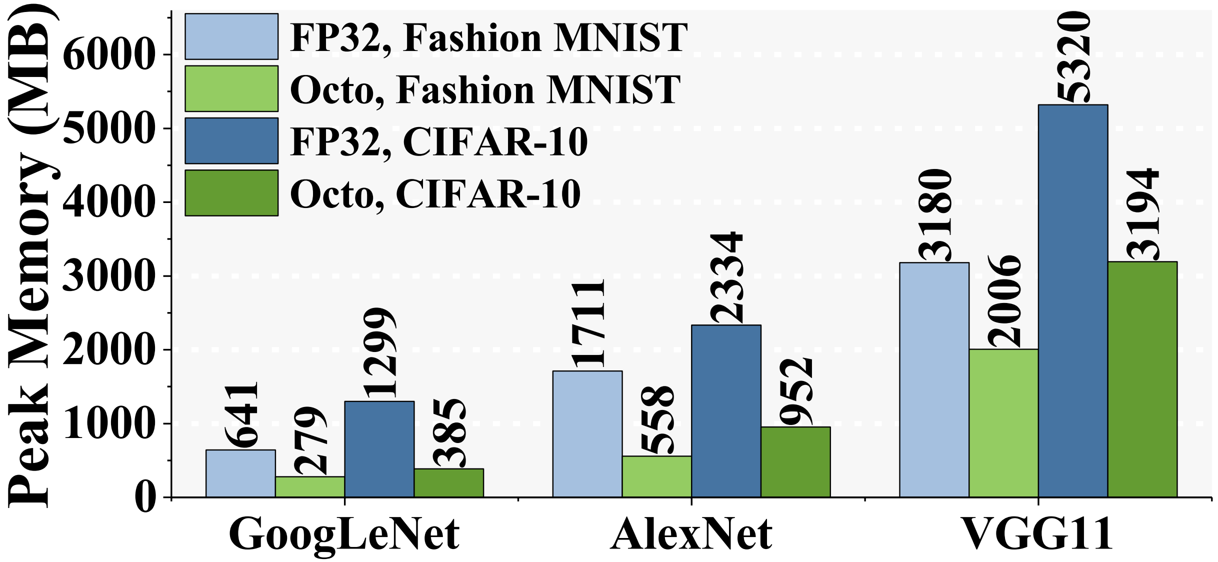
Per-iteration Time vs. FP32:

- 1.73× average speedups

System Overhead

Memory Footprint:





Average Memory vs. FP32:
- 21.19% reduction

Peak Memory vs. FP32:
- 3.37× reduction

Conclusion

Octo: a lightweight INT8 training framework for on-device learning

- **Hardware-level quantization**, which accelerate both forward and backward stages.
- **Loss-aware Compensation**, which fills the error gap of quantized dot product via an approximated affine transformation.
- **Parameterized Range Clipping**, which maintains bit precision in gradient calculation and avoids offset impact of the zero point via symmetric clipping.
- **Cross-platform prototype system**, which is compatible with different operating systems and can be easily ported to embedded platforms.
- Octo holds higher **training efficiency** over state-of-the-art quantization training methods, while achieving adequate **processing speedup (2.03×)** and **memory reduction (3.37×)** over the full-precision training.

Thank you!

csqzhou@comp.polyu.edu.hk

https://github.com/kimihe/Octo