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

Qihua Zhou\textsuperscript{1}, Song Guo\textsuperscript{1}, Zhihao Qu\textsuperscript{2}, Jingcai Guo\textsuperscript{1}, Zhenda Xu\textsuperscript{1}, Jiewei Zhang\textsuperscript{1}, Tao Guo\textsuperscript{1}, Boyuan Luo\textsuperscript{1}, Jingren Zhou\textsuperscript{3}

\textsuperscript{1}The Hong Kong Polytechnic University, \textsuperscript{2}Hohai University, \textsuperscript{3}Alibaba Group
Rise of On-device Learning

**In-cloud Learning:**
- Large Models
- Send Request and Upload Data
- Wait Execution and Fetch Results
- Distributed Processing

**On-device Learning:**
- Deploy on Devices
- Tiny Models
- End-to-end Implementation

**Drawbacks:**
- High Latency
- Privacy Leakage
- Lack Personalization
- Huge Cost

**Advantages:**
- Resource Saving
- Adapt to Limited Resources
- Personalized Training
- Online Applicable

**System Implementation**

**Challenges:**
- Small-scale User Data
- Backward Propagation Blocking
- Limited Peak Processing Speed

**Question:** how to overcome the challenges of resource constraints?
**Solution:** enable quantization-aware training.
Common Compression Methods

(1) Low Rank Factorization

\[ m \{ \begin{bmatrix} A \end{bmatrix} \} \approx m \{ \begin{bmatrix} X \end{bmatrix} \begin{bmatrix} Y \end{bmatrix} \} k \]

(2) Model Pruning

(3) Network Sparsification
Common Compression Methods

(1) Low Rank Factorization

\[ A \approx X Y \]

Inefficiency:

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

(2) Model Pruning

(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

What are the fundamental instructions dominating the computational cost?

Convolutional (CONV) Layers

Fully Connected (FC) Layers

Feature Maps

Prediction

Loss Function

Gap from Ground Truth

Update Model Parameters

Gradients of FCs

Gradients of CONVs

Forward Pass

Backward Pass

Input

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

How to bridge this gap?

Enable hardware-level quantization-aware training (QAT)
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:
\[
s = \text{scale}(a, b, n) = \frac{b - a}{2^n - 1}, a < b, \]
\[
q = \text{round}\left(\frac{\text{clip}(r)}{\text{scale}(a, b, n)}\right) + z.
\]

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

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

**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: \[ X_q = \text{round}\left(\frac{X_f}{s_x} + z_x\right), \quad W_q = \text{round}\left(\frac{W_f}{s_w} + z_w\right) \]

Step #2 Dot Product: \[ Y_q = \text{dot}(X_q, W_q) \]

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

Analysis of Error Gap: \[ \delta = s_x \cdot \Delta X \cdot W_f + \gamma \]

Approximation via Affine Transformation: \[ \hat{\delta} = \alpha \cdot \mu + \beta \]

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
- **Tensor:** distribution expectation

**Approximated Compensation Term:**

\[
\hat{\delta} = \alpha \times \mu + \beta
\]

**Element Broadcasting**

**Bit-wise Shifting**

**L2-Regularization of Compensation Parameters:**

\[
L = -\frac{1}{N} \sum_{n}^{N} \sum_{k}^{k} t_{nk} \log y_{nk} + \frac{1}{2} \lambda (\mu^2 + \beta^2),
\]

**Primary Cross-entropy Error:**
Measure difference between prediction \(y\) and ground truth \(t\)

**L2-regularizer:**
reflect compensation performance based on \(\mu\) and \(\beta\)
Backward Quantization

Calculation of Derivative Flows for weights $W$ and features $X$:

$$\frac{\partial L}{\partial Y} = \frac{\partial L}{\partial Y} \cdot W$$

Parameterized Range Clipping:

$$\text{clip}(M) \in [-a, a],$$

$$a = \min\{|\min(M)|, \max(M)|\}.$$
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
Octo preservers model accuracy as FP32 does while Fake QAT fails to converge.
## Ablation Study: Impact of LAC and PRC

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Acc. (%)</th>
<th>Gap over FP32 (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fashion MNIST</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP32</td>
<td>97.1</td>
<td>0</td>
</tr>
<tr>
<td>INT8</td>
<td>13</td>
<td>-84.1</td>
</tr>
<tr>
<td>INT8 + LAC</td>
<td>90.4</td>
<td>-6.7</td>
</tr>
<tr>
<td>INT8 + PRC</td>
<td>14.8</td>
<td>-82.3</td>
</tr>
<tr>
<td>INT8 + LAC + PRC</td>
<td>93.6</td>
<td>-3.5</td>
</tr>
<tr>
<td><strong>CIFAR-10</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP32</td>
<td>93.5</td>
<td>0</td>
</tr>
<tr>
<td>INT8</td>
<td>11</td>
<td>-82.5</td>
</tr>
<tr>
<td>INT8 + LAC</td>
<td>85.2</td>
<td>-8.3</td>
</tr>
<tr>
<td>INT8 + PRC</td>
<td>12.1</td>
<td>-81.4</td>
</tr>
<tr>
<td>INT8 + LAC + PRC</td>
<td>86.7</td>
<td>-6.8</td>
</tr>
</tbody>
</table>
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:

Quantization Overhead:
- Lower than 15%

Per-iteration Time vs. FP32:
- $1.73 \times$ 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!

csqzhuo@comp.polyu.edu.hk

https://github.com/kimihe/Octo