On-Device Training Under 256KB Memory

MIT, MIT-IBM Watson AI Lab

NeurIPS 2022
Background

TinyML is challenging due to limited memory

<table>
<thead>
<tr>
<th></th>
<th>Cloud AI</th>
<th>Mobile AI</th>
<th>Tiny AI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory (Activation)</td>
<td>32GB</td>
<td>4GB</td>
<td>320kB</td>
</tr>
<tr>
<td>Storage (Weights)</td>
<td>~TB/PB</td>
<td>256GB</td>
<td>1MB</td>
</tr>
</tbody>
</table>

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]

On-Device Training Under 256KB Memory

https://tinytraining.mit.edu
Background

TinyML is challenging due to limited memory

<table>
<thead>
<tr>
<th></th>
<th>Cloud AI</th>
<th>Mobile AI</th>
<th>Tiny AI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory (Activation)</td>
<td>32GB</td>
<td>4GB</td>
<td>320kB</td>
</tr>
<tr>
<td>Storage (Weights)</td>
<td>~TB/PB</td>
<td>256GB</td>
<td>~1MB</td>
</tr>
</tbody>
</table>

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
Background

MCUNet enables tinyML inference

Toy applications

Real-life applications

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
Can We Learn on the Edge?

From tinyML inference to training

- On-device learning:
  - **customization** by adapting to user data / **life-long** learning
  - better **privacy**, lower **cost**

Cloud-based Learning

On-device Learning

data cannot be sent to the cloud for privacy reason

User ➔ New and Sensitive Data ➔ Intelligent Edge Devices ➔ Cloud Server

MCUNet: System-Algorithm Co-Design for TinyML

https://mcunet.mit.edu
Can We Learn on the Edge?
From tinyML inference to training

- On-device learning:
  - customization by adapting to user data / life-long learning
  - better privacy, lower cost
  - Training is more expensive than inference
    - For example, store intermediate activation, extra back-propagation, etc.
On-Device Training Under 256KB Memory

- **Training** is more expensive than **inference** due to back-propagation, making it hard to fit IoT devices (e.g., MCU only has 256KB SRAM).

<table>
<thead>
<tr>
<th>Model</th>
<th>Cloud Memory</th>
<th>Edge Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow</td>
<td>652 MB</td>
<td></td>
</tr>
<tr>
<td>PyTorch</td>
<td>303 MB</td>
<td></td>
</tr>
<tr>
<td>MNN (edge)</td>
<td>41.5 MB</td>
<td>1 MB</td>
</tr>
</tbody>
</table>

- **Quantization-aware scaling**
- **Sparse layer/tensor update**

256KB constraint:
- TensorFlow: 141 KB
- PyTorch: 0.1 MB
- MNN (edge): 1 MB

2.9x, 2.0x, 8.8x, 2.4x, 2300x
On-Device Training Under 256KB Memory

- **Training** is more expensive than **inference** due to back-propagation, making it hard to fit IoT devices (e.g., MCU only has 256KB SRAM).

![Graph showing memory usage comparison between different frameworks and techniques.

- TensorFlow (cloud) uses 652 MB.
- PyTorch (cloud) uses 303 MB.
- MNN (edge) uses 41.5 MB.
- Tiny Training Engine uses 5.7 MB.
- With quantization-aware scaling, the Tiny Training Engine uses 2.9 MB, an improvement of 7.3x.
- With sparse layer/tensor update, it uses 355 KB, an improvement of 8.8x.
- With operator reordering, it uses 141 KB, an improvement of 2300x.]


On-Device Training Under 256KB Memory

1. Quantization-aware scaling
2. Sparse layer/tensor update
3. Tiny Training Engine
On-Device Training Under 256KB Memory

1. Quantization-aware scaling
2. Sparse layer/tensor update
3. Tiny Training Engine
1. Quantization-Aware Scaling (QAS)

Real quantized graphs save memory...

(a) Fake Quantization (quantization aware training)

- **output**: fp32
- **fake quantize**: fp32
- **ReLU6**: fp32
- **BatchNorm**: fp32
- **conv**: fp32

(b) Real Quantization (inference/on-device training)

- **output**: int8
- **scale & cast**: fp32
- **bias**: int32
- **conv**: int8
- **weights**: int8

<table>
<thead>
<tr>
<th></th>
<th>Fake</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Weight</strong></td>
<td>FP32</td>
<td>INT8</td>
</tr>
<tr>
<td><strong>Activation</strong></td>
<td>FP32</td>
<td>INT8</td>
</tr>
<tr>
<td><strong>Batch Norm</strong></td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
1. Quantization-Aware Scaling (QAS)

But are hard to quantize

Making training difficult:
- Mixed precisions: int8/int32/fp32…
- Lack BatchNorm

Performance Comparison (average on 10 datasets)

<table>
<thead>
<tr>
<th></th>
<th>Top-1 Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP32 SGD</td>
<td>86.0</td>
</tr>
<tr>
<td>Int8 SGD</td>
<td>75.4</td>
</tr>
</tbody>
</table>

(a) Real Quantization
1. Quantization-Aware Scaling (QAS)

Quantization leads to distorted gradient magnitudes

- Why is the training convergence worse?
- The scale of weight and gradients does not match in real quantized training!

\[
\log_{10}\left(\frac{\|W\|}{\|G\|}\right)
\]

(fp32, int8)
1. Quantization-Aware Scaling (QAS)

QAS addresses the optimization difficulty of quantized graphs

\[ \tilde{G}_W = G_W \cdot s_{W}^{-2}, \quad \tilde{G}_b = G_b \cdot s_{W}^{-2} \cdot s_{x}^{-2} = G_b \cdot s^{-2} \]

\[ \log_{10}(\frac{\|W\|}{\|G\|}) \]

Tensor Index

QAS aligns the W/G ratio with fp32
1. Quantization-Aware Scaling (QAS)

QAS addresses the optimization difficulty of quantized graphs.
On-Device Training Under 256KB Memory

1. Quantization-aware scaling
2. Sparse layer/tensor update
3. Tiny Training Engine
2. Sparse Layer/Tensor Update

Full update is too expensive

Updating the whole model is **too expensive**:
- Need to save all intermediate activation (quite large)
- Need to store the updated weights in SRAM (Flash is read-only)
2. Sparse Layer/Tensor Update

More efficient variants

- (a) full update
- (b) bias-only update
- (c) sparse layer update
- (d) sparse tensor update
2. Sparse Layer/Tensor Update

More efficient variants

- **(a) full update**
  - Activation to store: \((H, M)\)
  - Weight in SRAM: \((M, H)\)

- **(b) bias-only update**
  - Activation to store: \((H, 0.25M)\)
  - Weight in SRAM: \((0.25M, N)\)

- **(c) sparse layer update**
  - Activation to store: \((H, M)\)
  - Weight in SRAM: \((M, H)\)

- **(d) sparse tensor update**

Reduce weight and activation buffer
2. Sparse Layer/Tensor Update

Find layers to update by sensitivity analysis

(a) Investigate the contribution of last $k$ biases $\Delta \text{acc}_{b[l:k]}$

For bias update

* Accuracy goes higher as more layers are updated, but plateaus soon.

(b) Investigate the contribution of a certain weight $\Delta \text{acc}_{w_l,r}$

For weight update

* later layers are more important

* The first point-wise conv contributes more

\[
 k^*, i^*, r^* = \max_{k, i, r} (\Delta \text{acc}_{b[l:k]} + \sum_{i \in i, r \in r} \Delta \text{acc}_{w_{l,r}}) \quad \text{s.t.} \quad \text{Memory}(k, i, r) \leq \text{constraint},
\]
2. Sparse Layer/Tensor Update

Lower memory, higher accuracy

(a) MCUNet-5FPS

(b) MbV2-w0.35

(c) Proxyless-w0.3

Sparse update can achieve higher transfer learning accuracy using 4.5-7.5x smaller extra memory.
On-Device Training Under 256KB Memory

1. Quantization-aware scaling
2. Sparse layer/tensor update
3. Tiny Training Engine
3. Tiny Training Engine (TTE)

Existing frameworks cannot fit

- **Runtime** is heavy
  - Heavy dependencies and large binary size (>100MB static memory)
  - Auto-diff at runtime; low edge efficiency
- **Memory** is heavy
  - A lot of intermediate (and unused) buffers
  - Has to compute full gradients

![Chart comparing memory usage of TensorFlow (cloud), PyTorch (cloud), and MNN (edge) with a 256KB constraint. The memory usage for TensorFlow (cloud) is 652 MB, PyTorch (cloud) is 303 MB, and MNN (edge) is 41.5 MB.]
3. Tiny Training Engine (TTE)

Move workload from runtime to compile time

Forward graph

Feed data and execute
X → Pred

Autodiff and run

dX ← Grad

Apply gradient step

\[ g_1, \ldots, g_n = \text{backward}(L, W) \]
\[ w_i = w_i - \eta g_i \]
\[ w_n = w_n - \eta g_n \]

Conventional training framework performs most tasks at runtime.
3. Tiny Training Engine (TTE)

Move workload from runtime to compile time

Conventional training framework performs most tasks at runtime.

Tiny Training Engine (ours) separate the environment of runtime and compile time.
3. Tiny Training Engine (TTE)

Re-ordering reduces memory footprint

**Operator life-cycle analysis** shows memory footprint can be greatly reduced by operator re-ordering.
3. Tiny Training Engine (TTE)

Smaller memory usage, faster training speed

20x smaller memory

23x faster speed
2. On-device training

Train done

Prediction:
class 1
Ground-Truth
class 0

fps: 1.798

Prediction:
green: correct
red: incorrect
Thanks for Listening!
https://tinytraining.mit.edu

1. Quantization-aware scaling
2. Sparse layer/tensor update
3. Tiny Training Engine