On-Device Training Under 256KB Memory

Song Han
MIT
songhan.mit.edu
mcunet.mit.edu
Can we Learn on the Edge?

AI systems need to continually adapt to new data collected from the sensors. Not only inference, but also re-training the model on edge devices.

- **On-device learning**: better privacy, lower cost, customization, life-long learning
- But edge devices have limited memory
- Training is more expensive than inference, making it hard to fit tiny hardware
MCUNet: Bring AI to IoT Devices

Unlock ultra low-power AoIoT Applications

- TinyML: design light-weighted neural networks and deploy on cheap edge devices that has low power, computing, and memory.
- Low-cost ($1-2), low-power, small, everywhere in our lives.
- AI on MCU is **hard**: No DRAM. No OS. Extreme memory constraint.
- Existing work optimize for `#parameters`, but `#activation` is the real bottleneck.
- MCUNet: first to achieve >70% ImageNet top1 accuracy on a microcontroller.
- **Cloud AI**: ResNet; **Mobile AI**: MobileNet; **Tiny AI**: MCUNet.

![MCUNet Graph](image-url)
MCUNet-v2: Patch-Based Inference

Detect person using only 30KB of memory!

- MCUNet V2
- MbV2+TF-Lite
- MCUNet
- Proxyless+TF-Lite

Measured Peak SRAM (kB)

Peak SRAM (kB) @ 90% VWW accuracy

MCUNet V2, NeurIPS'21
Training on MCU is Difficult: 100MB vs. 100KB

- TensorFlow (cloud): 652 MB
- PyTorch (cloud): 303 MB
- MNN (edge): 41.5 MB

256KB constraint
Our Solution: System+Algorithm Optimization

• Reducing memory usage by >2000x
1. Address Optimization Difficulty of Quantized Graphs

- **Fake** quantized graph vs. **Real** quantized graph

<table>
<thead>
<tr>
<th></th>
<th>Fake</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>FP32</td>
<td>INT8</td>
</tr>
<tr>
<td>Activation</td>
<td>FP32</td>
<td>INT8</td>
</tr>
<tr>
<td>Batch Norm</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
1. Address Optimization Difficulty of Quantized Graphs

- **Real** quantized graphs vs. **fake** quantized graphs

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

(a) Real Quantization.

Performance Comparison (average on 10 datasets)

<table>
<thead>
<tr>
<th>Precision</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>

10.6% top-1↓
1. Address Optimization Difficulty of Quantized Graphs

- Why is the training convergence worse?
1. Address Optimization Difficulty of Quantized Graphs

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

![Graph showing log₁₀(∥W∥/∥G∥) vs Tensor Index with fp32 and int8 plots]
QAS: Quantization-Aware Scaling

QAS addresses the optimization difficulty of quantized graphs

Quantization overview
\[ \tilde{y}_{\text{int8}} = \text{cast2int8}[s_{\text{fp32}} \cdot (\bar{W}_{\text{int8}} \bar{x}_{\text{int8}} + \bar{b}_{\text{int32}})], \]

Per Channel scaling
\[ W = s_W \cdot (W / s_W) \quad \text{quantize} \quad s_W \cdot \bar{W}, \quad \bar{G}_W \approx s_W \cdot G_W, \]

Weight and gradient ratios are off by Sw
\[ \|\bar{W}\| / \|G_{\bar{W}}\| \approx \|W / s_W\| / \|s_W \cdot G_W\| = \boxed{s_W^{-2}} \cdot \|W\| / \|G\|. \]

Thus, re-scale the gradients
\[ \tilde{G}_{\bar{W}} = G_{\bar{W}} \cdot s_W^{-2}, \quad \tilde{G}_b = G_b \cdot s_W^{-2} \cdot s_x^{-2} = G_b \cdot s^{-2} \]
QAS: Quantization-Aware Scaling

QAS addresses the optimization difficulty of quantized graphs

\[ \tilde{G}_W = G_W \cdot s_W^{-2}, \quad \tilde{G}_B = G_B \cdot s_W^{-2} \cdot s_x^{-2} = G_B \cdot s^{-2} \]

QAS aligns the W/G ratio with fp32
QAS: Quantization-Aware Scaling

QAS addresses the optimization difficulty of quantized graphs

Performance Comparison (average on 10 datasets)

<table>
<thead>
<tr>
<th>Method</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>
<tr>
<td>Int8 LARS</td>
<td>64.8</td>
</tr>
<tr>
<td>Int8 Adam</td>
<td>84.5</td>
</tr>
<tr>
<td>Int8 QAS (ours)</td>
<td>86.9</td>
</tr>
</tbody>
</table>

Extra memory (3x) Improve convergence
2. Sparse Layer/Tensor Update

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

(a) full update  (b) bias-only update  (c) sparse layer update  (d) sparse tensor update

Dense Backward

\[
\frac{dy}{dw} : \begin{align*}
(A \cdot T) \\
x
\end{align*} = \begin{align*}
(A \cdot T) \\
(dW) \cdot T
\end{align*}
\]

Activation to store: (H, M)
Weight in SRAM: (M, H)

Sparse Tensor Backward

\[
\frac{dy}{dw} : \begin{align*}
(A \cdot T) \\
x
\end{align*} = \begin{align*}
(A \cdot T) \\
(dW) \cdot T
\end{align*}
\]

Activation to store: (H, 0.25*M)
Weight in SRAM: (0.25*M, N)

1/4
Find Layers to Update by Contribution Analysis

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

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

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

For weight update
* later layers are more important
* The first point-wise conv contributes more

$$k^*, i^*, r^* = \max_{k,i,r} (\Delta\text{acc}_{b_{[k]}} + \sum_{i \in I, r \in R} \Delta\text{acc}_{w_{i,r}}) \quad \text{s.t. Memory}(k, i, r) \leq \text{constraint},$$
Find Layers to Update by Contribution Analysis

(a) per-layer memory usage

(b) sparse update scheme

(high activation cost) not update bias/forward only

update bias (low activation cost)

sparse layer update (low memory cost) sparse tensor update (high acc)
Sparse update: Lower Memory, Higher Accuracy

Sparse update can achieve higher transfer learning accuracy using 4.5-7.5x smaller extra memory.
3. Tiny Training Engine (TTE)

(a) input model → (b) compile-time autodiff → (c) graph pruning → (d) op reordering → (e) on-device training

Sensor data → runtime update
Previous DL Training

1. Computation Graph (forward)
Previous DL Training

1. Computation Graph (forward)

Data → Matmul → Weight → Output

2. Autograd Engine

\[ f(x) \rightarrow f'(x) \]
Previous DL Training

1. Computation Graph (forward)

2. Autograd Engine

\[ f(x) \rightarrow f'(x) \]

3. Computation Graph (backward)
Previous DL Training

1. Computation Graph (forward)
2. Autograd Engine
   \[ f(x) \rightarrow f'(x) \]
3. Computation Graph (backward)
4. Execution Engine
   Detailed execution schedules.
1. Computation Graph (forward)

Data → Matmul → Weight → Output

2. Autograd Engine

\[ f(x) \rightarrow f'(x) \]

3. Computation Graph (backward)

Data → Matmul → Weight → Output

4. Execution Engine

Detailed execution schedules.

Conventional training framework focus on **flexibility**, and the auto-diff is performed at **runtime**.
Limitations with Previous Training Infra

• Runtime is heavy
  • Autodiff at runtime
  • Heavy dependencies and large binary size
  • Operators optimized for the cloud, not for edge

• Memory is heavy
  • A lot of intermediate (and unused) buffers
  • Has to compute full gradients
Tiny Training Engine (TTE) **separates** the runtime and compile-time.

**TTE offloads most workloads** like autodiff / graph optimization / perform tuning **into compile-time.**

Thus, the overhead of runtime is **minimized.**
Tiny Training Engine Workflow

PyTorch

```python
net = nn.Sequential(
    nn.Conv2d(3, 3, kernel=3, padding=1),
    nn.ReLU()
)

data = torch.randn(1, 3, 28, 28)
out = net(data)
```

Forward IR

```python
fn (%input0: Tensor[(1, 3, 28, 28), float32],
    %v0.weight: Tensor[(3, 3, 3, 3), float32]) {  
    %0 = nn.conv2d(%input0, %v0.weight, padding=[1, 1, 1, 1], channels=3, kernel_size=[3, 3]);  
    nn.relu(%0)
}
```
Tiny Training Engine Workflow

Forward IR

```python
fn (%input0: Tensor[(1, 3, 28, 28), float32], %v0.weight: Tensor[(3, 3, 3, 3), float32])
    %0 = nn.conv2d(%input0, %v0.weight, padding=[1, 1, 1, 1], channels=3, kernel_size=[3, 3]);
    %1 = nn.relu(%0);
    # grad_input
    %2 = padding(%grad_output);
    %3 = nn.conv2d_transpose(%grad_output, %v0.weight, %2, padding=[1, 1, 1, 1], channels=3, kernel_size=[3, 3]);
    # grad_weight
    %4 = reshape_padding(%grad_output);
    %5 = nn.conv2d(%input0, %grad_output, padding=[1, 1, 1, 1], channels=3, kernel_size=[3, 3]);
    % grad_bias
    %6 = sum(%grad_output, axis=[-1, -2]);
    (%3, %5, %6)
```
Tiny Training Engine Workflow

- Graph-level optimizations:
  - Sparse layer / sparse tensor update
  - Operator reordering and in-place update
  - Constant folding
  - Dead-code elimination
Sparse Layer / Sparse Tensor Update

(a) full update
(b) bias-only update
(c) sparse layer update
(d) sparse tensor update

\[
\begin{align*}
W_1 & \quad b_1 & \quad W_{j+1} & \quad b_{j+1} \\
\text{(a) full update} & & & & \\
\text{(b) bias-only update} & & & & \\
\text{(c) sparse layer update} & & & & \\
\text{(d) sparse tensor update} & & & & \\
\end{align*}
\]

\[
\text{Example from a matrix multiplication with full update}
\]

\[
\text{fn (} x : \text{Tensor}[(10, 10), \text{float32}],
\text{\%weight : Tensor}[(10, 10), \text{float32}],
\text{\%bias : Tensor}[(10), \text{float32}],
\text{\%grad : Tensor}[(10), \text{float32}],
\text{)}
\]

\[
\{
\text{# forward}
\text{\%0 = multiply(} x, \%weight); \\
\text{\%1 = add(} \text{\%0, \%bias); \\
\text{# backward}
\text{\%3 = multiply(} \%grad, \%weight); \text{====> } dy / dx \\
\text{\%4 = transpose(} \%grad \\
\text{\%5 = multiply(} \%4, \%x); \text{====> } dy / dw \\
\text{\%6 = sum(} \%grad, \text{axis=-1}); \text{====> } dy / db
\}
\]

\[
(\%3, \%5, \%6)
\]
fn (%x: Tensor[(10, 10), float32, needs_grad=True],
  %weight: Tensor[(10, 10), float32, needs_grad=False],
  %bias: Tensor[(10), float32, needs_grad=True],
  %grad: Tensor[(10), float32]),
{
  # forward
  %0 = multiply(%x, %weight);
  %1 = add(%0, %bias);
  # backward
  %3 = multiply(%grad, %weight);
  %4 = transpose(%grad);
  %5 = multiply(%4, %x);
  %6 = sum(%grad, axis=-1);
  (%3, %5, %6)
}
Sparse Layer / Sparse Tensor Update

\[ W_i \quad b_i \quad W_{i+1} \quad b_{i+1} \]

(a) full update

(b) bias-only update

(c) sparse layer update

(d) sparse tensor update

\[
\text{fn} \ (\%x: \text{Tensor}[\text{10, 10}, \text{float32}, \text{needs_grad}=\text{True}], \\
\%weight: \text{Tensor}[\text{10, 10}, \text{float32}, \text{needs_grad}=\text{False}], \\
\%bias: \text{Tensor}[\text{10}, \text{float32}, \text{needs_grad}=\text{True}], \\
\%grad: \text{Tensor}[\text{10}, \text{float32}]),
\]

\[
\{ \\
\quad \# \text{forward} \\
\quad \%0 = \text{multiply}(\%x, \%weight); \\
\quad \%1 = \text{add}(\%0, \%bias); \\
\quad \# \text{backward} \\
\quad \%3 = \text{multiply}(\%grad, \%weight); \\
\quad \%4 = \text{transpose}(\%grad); \\
\quad \%5 = \text{multiply}(\%4, \%x); \\
\quad \%6 = \text{sum}(\%grad, \text{axis}=\text{-1}); \\
\quad (\%3, \%5, \%6)
\}
\]

Remove unnecessary computations from DAG via dependency analysis and dead-code elimination.
Freely annotate any parameters and TTE will trim the computation accordingly.
Sparse Layer / Sparse Tensor Update

(a) full update
(b) bias-only update
(c) sparse layer update
(d) sparse tensor update

Automatically remove the buffers of pruned gradients from the computation graph.

\[
fn (\%x: \text{Tensor}[{(10, 10), \text{float32}]}, \\
\%weight: \text{Tensor}[{(10, 10), \text{float32}]}, \\
\%bias: \text{Tensor}[{(10), \text{float32}]}, \\
\%grad: \text{Tensor}[{(10), \text{float32}]})
\]

\[
\{ \\
\text{# forward} \\
\%0 = \text{multiply}(\%x, \%weight); \\
\%1 = \text{add}(\%0, \%bias); \\
\text{# backward} \\
\%3 = \text{multiply}(\%grad, \%weight); \\
\%4 = \text{transpose}(\%grad) \\
\%5 = \text{multiply}(\%4, \%x); \\
\%6 = \text{sum}(\%grad, \text{axis=-1}) \\
(\%3, \%5, \%6)
\}
\]

\[
fn (\%x: \text{Tensor}[{(10, 10), \text{float32}, \text{needs_grad=True}]}, \\
\%weight: \text{Tensor}[{(20, 10), \text{float32}, \text{needs_grad=0.5}]}, \\
\%bias: \text{Tensor}[{(20), \text{float32}, \text{needs_grad=True}]}, \\
\%grad: \text{Tensor}[{(10, 20), \text{float32}]})
\]

\[
\{ \\
\text{# forward} \\
\%0 = \text{multiply}(\%x, \%weight); \\
\%0.1 = \text{slice}(\%x, \text{begin=[0, 0]}, \text{ends=[10, 10]}); \\
\%1 = \text{add}(\%0, \%bias); \\
\text{# backward} \\
\%3 = \text{multiply}(\%grad, \%weight); \\
\%4 = \text{transpose}(\%grad) \\
\%5 = \text{multiply}(\%4, \%0.1); \\
\%6 = \text{sum}(\%grad, \text{axis=-1}) \\
(\%3, \%5, \%6)
\}
\]
Tiny Training Engine supports backward graph pruning and sparse update at IR-level. After pruning, un-used weights and sub-tensors are pruned from DAG => 8-10x memory saving. Combined with operator reorder => 22-28x memory saving.
Graph-level optimizations:
- Sparse layer / sparse tensor update
- **Operator reordering and in-place update**
- Constant folding
- Dead-code elimination
Operator Reordering and Inplace Update

out = model(data)
loss = criterion(out, label)
gradients = loss.backward()
optim.update(model, gradients)

Calculate **all gradients** first, then **apply one-by-one**. Intermediate buffers consume a lot of spaces.

---

out = model(data)
loss = criterion(out, label)
gradients = loss.backward()
optim.update(model, gradients)
for layers in model:
dydx, grad = layers.backward(loss)
optim.update(layers, grad)
loss = dydx

Gradient updates are **immediately applied** once calculated. Intermediate buffers can be released.
Operator Reordering and Inplace Update

By reordering, the gradient update can be immediately applied. Gradients buffer can be released earlier before back-propagating to earlier layers, leading to $2.7x \sim 3.1x$ peak memory reduction.
Life Cycle Analysis

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

Python Defined Models → Traced Static Graph → Backward Graph → Forward Graph → IR → IR → Tuned Schedules → CodeGen → Executable Binaries for Training

- Calculate derivatives at compilation time
- IR
- IR
- Graph Opt.

Latency (ms):

<table>
<thead>
<tr>
<th>Model</th>
<th>TF-Lite, full (projected, OOM)</th>
<th>TF-Lite, sparse</th>
<th>TTE, sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>MbV2</td>
<td>8,501</td>
<td>3,448</td>
<td>403</td>
</tr>
<tr>
<td>Proxyless</td>
<td>10,523</td>
<td>4,111</td>
<td>457</td>
</tr>
<tr>
<td>MCUNet</td>
<td>13,998</td>
<td>5,607</td>
<td>583</td>
</tr>
</tbody>
</table>

Our optimized operators demonstrate 21x ~ 23x speedup over TensorFlow-Lite.
Tiny Training Engine

Our codegen only generate binaries for used operators
TTE finally deliver a light-weight, portable, and efficient binary.
Comparison of Previous Infra and TTE

Conventional training framework performs most tasks at runtime.

Tiny Training Engine (ours) separate the environment of runtime and compile time.
2. On-device training

Train done

Prediction: class 1
Ground True: class 1
fps: 3.267

Prediction:
green: correct
red: incorrect
TinyML and Efficient Deep Learning

https://hanlab.mit.edu/

1. Learning both Weights and Connections for Efficient Neural Network, NeurIPS’15
2. Deep Compression, ICLR’16
3. AMC, ECCV’18
4. ProxylessNAS, ICLR’19
5. Once For All, ICLR’20
6. HAT, ACL’20
7. Anycost GAN, CVPR’21
8. SPVNAS, ECCV’21
9. Lite Pose, CVPR’22
10. NAAS, DAC’21
11. QuantumNAS, HPCA’22
12. QuantumNAT, DAC’22
13. QOC, DAC’22
14. MCUNet, NeurIPS’20
15. MCUNet-V2, NeurIPS’21
16. TinyTL, NeurIPS’20
17. MCUNet-V3, Arxiv’22
18. DGC, ICLR’18
19. DGA, NeurIPS’21
20. PVCNN, NeurIPS’19
21. Fast-LiDARNet, ICRA’21
22. BEVFusion, Arxiv’22
23. TSM, ICCV’19
24. GAN Compression, CVPR’20
25. SpAtten, HPCA’21
26. SpArch, HPCA’20
27. PointAcc, Micro’20
28. TorchSparse, SysML’22
HAN Lab Students: Yujun Lin (Arch PhD), Hanrui Wang (Arch PhD), Zhijian Liu (ML PhD)

Potential product impact for NVIDIA: future TensorRT and cuDNN libraries.

This is a large design space that's hard to be explored by human. It should be explored

training, inference x channel number x layer number x bit width x decimal point

linear quantization, log quantization, kmeans quantization

for deep learning. The design space include:

careful scaling factor tuning to avoid underflow or overflow

Given the large extremes of quantization. The former has easy hw implementation but poor

Project 2: "Optimal Number Representation for E

Potential product impact for NVIDIA: future DLA architectures in Xavier, Orin, etc.

speech recognition, image classification, and Progressive GAN, which makes real-time

learning applications accelerated with such sparse primitives: machine translation,

then integrate the HW primitive into TACO. Then, I want to co-design the machine

complexity arrays. After that, I'd like to implement this architecture in FPGA or ASIC,

complexity, O(n) area complexity arrays; or O(1) time complexity, O(n^2) space

(AND). Software implementation need O(n) cycles. I plan to work on O(log(n)) time

linear algebra. There are two basic operations to be accelerated: union (OR) and join

accelerator support. Therefore, I plan to work on an specialized accelerator for sparse

the first hardware accelerator for sparse DNN, it's e

Challenging for general purpose hardware to take advantage of sparsity. EIE [Han'16] is


Project 1: "E

fficient Hardware Primitives for Sparse Linear Algebra"
This presentation in this publication was presented at the tinyML® On Device Learning Forum 2022. The content reflects the opinion of the author(s) and their respective companies. The inclusion of presentations in this publication does not constitute an endorsement by tinyML Foundation or the sponsors.

There is no copyright protection claimed by this publication. However, each presentation is the work of the authors and their respective companies and may contain copyrighted material. As such, it is strongly encouraged that any use reflect proper acknowledgement to the appropriate source. Any questions regarding the use of any materials presented should be directed to the author(s) or their companies.

tinyML is a registered trademark of the tinyML Foundation.