On-Device Training Under 256KB Memory

Ji Lin*, Ligeng Zhu*, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, Song Han

Massachusetts Institute of Technology
Deep Learning Going “Tiny”

Cloud AI
Data centers

Mobile AI
Smartphones

?
Deep Learning Going “Tiny”

Cloud AI
Data centers

Mobile AI
Smartphones

Tiny AI
IoT Devices/
Microcontrollers
TinyML Faces Tight Memory Constraints

<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>

13,000x smaller

100,000x smaller
MCUNet: Bring AI to IoT Devices

![Graph showing the relationship between VWW Accuracy (%) and Measured Peak SRAM (kB) for different models: MCUNetV2, MbV2+TF-Lite, MCUNet, Proxyless+TF-Lite. The graph indicates that MCUNetV2 has a higher accuracy with a smaller measured peak SRAM, while Proxyless+TF-Lite has a lower accuracy with a higher measured peak SRAM.]
On-device Training is Essential

- AI systems need to adapt to new sensor data for **customization** and **continual learning**
- Cloud-based training leads to **privacy** issue and **high cost**
- However, training is more **expensive** than inference, making it hard to fit tiny hardware (MCU only has 256KB SRAM)

Users → New and sensitive data → Intelligent Edge Devices → Cloud Server
Existing solution: high memory cost

- 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

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

<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>

(a) Real Quantization.

(b) Fake Quantization.
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.
1. Address Optimization Difficulty of Quantized Graphs

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

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

Performance Comparison (average on 10 datasets)

(a) Real Quantization.
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 for fp32 and int8]
1. Address Optimization Difficulty of Quantized Graphs

• Quantization-Aware Scaling (QAS)

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

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

Weight and gradient ratios are off by \(Sw\)
\[
\|\bar{W}\|/\|G_W\| \approx \|W/s_W\|/\|s_W \cdot G_W\| = \left(\frac{s_W}{s_W}\right)^2 = \|W\|/\|G\|.
\]

Thus, re-scale the gradients
\[
\tilde{G_W} = G_W \cdot s_W^{-2}, \quad \tilde{G_b} = G_b \cdot s_W^{-2} \cdot s_x^{-2} = G_b \cdot s^{-2}
\]
1. Address Optimization Difficulty of Quantized Graphs

Quantization-Aware Scaling (QAS)

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

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

Tensor Index

QAS aligns the W/G ratio with fp32
1. Address Optimization Difficulty of Quantized Graphs

Quantization-Aware Scaling (QAS) addresses the optimization difficulty.

Performance Comparison (average on 10 datasets)

<table>
<thead>
<tr>
<th>Top-1 Accuracy (%)</th>
<th>FP32 SGD</th>
<th>Int8 SGD</th>
<th>Int8 LARS</th>
<th>Int8 Adam</th>
<th>Int8 QAS (ours)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>86.0</td>
<td>75.4</td>
<td>64.8</td>
<td>84.5</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

Dense Backward

\[
\frac{dy}{dx} : (N, H) \rightarrow (H, M) \rightarrow (N, M)
\]

\[
\frac{dy}{dw} : (H, N) \rightarrow (N, M) \rightarrow (H, M)
\]

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

Sparse Tensor Backward

\[
\frac{dy}{dx} : (N, H) \rightarrow (H, M) \rightarrow (N, M)
\]

\[
\frac{dy}{dw} : (H, N) \rightarrow (N, M) \rightarrow (H, M)
\]

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

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

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

(a) per-layer memory usage

Memory (KB)

(b) sparse update scheme

Ratio

Layer Index

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

1. Computation Graph (forward)
1. Computation Graph (forward)

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

3. Computation Graph (backward)

4. Execution Engine

Detailed execution schedules.

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

1. Computation Graph (forward)
   - Data
   - Weight
   - Matmul
   - Output

2. Autograd Engine
   \[ f(x) \rightarrow f'(x) \]

3. Computation Graph (backward)
   - Data
   - Weight
   - Matmul
   - Output

4. Execution Engine

Detailed execution schedules.

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

- Runtime is heavy
  - Autodiff at runtime
  - Heavy dependencies and large binary size
  - Operators optimized for 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. **offloads most workloads** like autodiff / graph optimization / perform tuning into compile-time. Thus, the overhead of runtime is minimized.
Tiny Training Engine Workflow

Python Defined Models

Traced Static Graph

Backward Graph

Calculate derivatives at compilation time

Forward Graph

Graph Opt.

IR

Tune Schedules

IR

CodeGen

Executable Binaries for Training

Python Defined Models

Traced Static Graph

Backward Graph

Calculate derivatives at compilation time

Forward Graph

Graph Opt.

IR

Tune Schedules

IR

CodeGen

Executable Binaries for Training

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);
  %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, %4, padding=[1, 1, 1, 1], channels=3, kernel_size=[3, 3]);
  % grad_bias
  %6 = sum(%grad_output, axis=[-1, -2]);
  (%3, %5, %6)
}
```

Backward IR

```python
fn (%input0: Tensor[(1, 3, 28, 28), float32], %v0.weight: Tensor[(3, 3, 3, 3), float32], %grad_output: Tensor[(1, 3, 28, 28), float32]) {
  # forward
  %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, %4, 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

- There are bunch of graph-level optimizations can be applied here.
  - Fold Constant
  - Dead-code Elimination
  - Sparsify gradient pass
  - Inline / Fusing

- Python Defined Models
- Traced Static Graph
- Backward Graph
- Forward Graph
- IR
- Graph Opt.
- IR
- Tune Schedules
- CodeGen
- Executable Binaries for Training

: Compile-Time
: Runtime
Sparse Update Schemes

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

Example from a matrix multiplication with full update

```
fn (%x: Tensor[(10, 10), float32],
    %weight: Tensor[(10, 10), float32],
    %bias: Tensor[(10), float32]),
    %grad: Tensor[(10), float32]),
{
    # forward
    %0 = multiply(%x, %weight);
    %1 = add(%0, %bias);
    # backward
    %3 = multiply(%grad, %weight);  ===> dy / dx
    %4 = transpose(%grad)
    %5 = multiply(%4, %x);  ===> dy / dw
    %6 = sum(%grad, axis=-1);  ===> dy / db
    (%3, %5, %6)
}
```
Sparse Update Schemes

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

\[
\begin{align*}
\text{(a) full update} & : W_i, b_i, W_{i+1}, b_{i+1} \\
\text{(b) bias-only update} & : W_i, W_{i+1}, b_{i+1} \\
\text{(c) sparse layer update} & : W_i, W_{i+1}, b_{i+1} \\
\text{(d) sparse tensor update} & : W_i, W_{i+1}, b_{i+1}
\end{align*}
\]

Annotate whether a param requires gradient or not

```python
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 Update Schemes

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

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

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

Automatically remove the buffers related with pruned gradients from computation graph.

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

Activation to store: (H, 0.5*M)
Weight in SRAM: (0.5*M, N)
Sparse Update System Support

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 thus achieve real saving \((8-10\times)\).
Our optimized operators demonstrate consistent speedup \((22x \sim 28x)\) over TensorFlow-Lite implementations.

### Peak Memory vs. Models

<table>
<thead>
<tr>
<th>Model</th>
<th>Peak Mem (KB)</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>
<td>28x smaller</td>
</tr>
<tr>
<td>Proxyless</td>
<td>10,523</td>
<td>4,111</td>
<td>457</td>
<td>24x smaller</td>
</tr>
<tr>
<td>MCUNet</td>
<td>13,398</td>
<td>5,607</td>
<td>583</td>
<td>22x smaller</td>
</tr>
</tbody>
</table>

---

**Optimized Implementation**
Tiny Training Engine Workflow

- There are bunch of **graph-level optimizations** can be applied here.
  - Fold Constant
  - Dead-code Elimination
  - Sparsify gradient pass
  - Reorder / Fusing
Re-order 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.

Gradient updates are immediately applied once calculated.
Intermediate buffers can be released.
Re-order 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.
Operator Cycle Analysis

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

Python Defined Models → Traced Static Graph → Forward Graph → IR → Graph Opt. → IR → Tune Schedules → CodeGen → Executable Binaries for Training

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

```
runtime::Module fwd_mod = runtime::Module::LoadFromFile("fwd.so");
runtime::Module bwd_mod = runtime::Module::LoadFromPathFile("bwd.so");

auto data = tensor::randn(1, 3, 128, 128);
auto out = fwd_mod(data);
auto gradients = bwd_mod(data);
```
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