Algorithm-System Co-Design for TinyML

Ligeng Zhu
ligeng@mit.edu
MIT
Today’s AI is too BIG
Better model always comes with higher computational cost (vision)

<table>
<thead>
<tr>
<th>Model</th>
<th>ImageNet Top-1 accuracy (%)</th>
<th>MACs (Billion)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBNetV2</td>
<td>73</td>
<td>2M</td>
</tr>
<tr>
<td>ShuffleNet InceptionV</td>
<td>75</td>
<td>4M</td>
</tr>
<tr>
<td>DenseNet-121</td>
<td>77</td>
<td>8M</td>
</tr>
<tr>
<td>DenseNet-169</td>
<td>79</td>
<td>16M</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>79</td>
<td>32M</td>
</tr>
<tr>
<td>ResNetXt-5</td>
<td>81</td>
<td>64M</td>
</tr>
<tr>
<td>InceptionV</td>
<td>79</td>
<td>2M</td>
</tr>
<tr>
<td>DPN-92</td>
<td>79</td>
<td>4M</td>
</tr>
<tr>
<td>DenseNet-26</td>
<td>79</td>
<td>8M</td>
</tr>
<tr>
<td>DenseNet-169</td>
<td>79</td>
<td>16M</td>
</tr>
<tr>
<td>ResNet-101</td>
<td>79</td>
<td>32M</td>
</tr>
<tr>
<td>ResNetXt-101</td>
<td>79</td>
<td>64M</td>
</tr>
<tr>
<td>X-ception</td>
<td>79</td>
<td>2M</td>
</tr>
<tr>
<td>MBNetV2</td>
<td>73</td>
<td>4M</td>
</tr>
<tr>
<td>ShuffleNet InceptionV</td>
<td>75</td>
<td>8M</td>
</tr>
<tr>
<td>DenseNet-121</td>
<td>77</td>
<td>16M</td>
</tr>
<tr>
<td>DenseNet-169</td>
<td>79</td>
<td>32M</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>79</td>
<td>64M</td>
</tr>
<tr>
<td>ResNetXt-5</td>
<td>81</td>
<td>2M</td>
</tr>
<tr>
<td>InceptionV</td>
<td>79</td>
<td>4M</td>
</tr>
<tr>
<td>DPN-92</td>
<td>79</td>
<td>8M</td>
</tr>
<tr>
<td>DenseNet-26</td>
<td>79</td>
<td>16M</td>
</tr>
<tr>
<td>DenseNet-169</td>
<td>79</td>
<td>32M</td>
</tr>
<tr>
<td>ResNet-101</td>
<td>79</td>
<td>64M</td>
</tr>
<tr>
<td>ResNetXt-101</td>
<td>79</td>
<td>2M</td>
</tr>
<tr>
<td>X-ception</td>
<td>79</td>
<td>4M</td>
</tr>
<tr>
<td>MBNetV2</td>
<td>73</td>
<td>8M</td>
</tr>
<tr>
<td>ShuffleNet InceptionV</td>
<td>75</td>
<td>16M</td>
</tr>
<tr>
<td>DenseNet-121</td>
<td>77</td>
<td>32M</td>
</tr>
<tr>
<td>DenseNet-169</td>
<td>79</td>
<td>64M</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>79</td>
<td>2M</td>
</tr>
<tr>
<td>ResNetXt-5</td>
<td>81</td>
<td>4M</td>
</tr>
<tr>
<td>InceptionV</td>
<td>79</td>
<td>8M</td>
</tr>
<tr>
<td>DPN-92</td>
<td>79</td>
<td>16M</td>
</tr>
<tr>
<td>DenseNet-26</td>
<td>79</td>
<td>32M</td>
</tr>
<tr>
<td>DenseNet-169</td>
<td>79</td>
<td>64M</td>
</tr>
<tr>
<td>ResNet-101</td>
<td>79</td>
<td>2M</td>
</tr>
<tr>
<td>ResNetXt-101</td>
<td>79</td>
<td>4M</td>
</tr>
<tr>
<td>X-ception</td>
<td>79</td>
<td>8M</td>
</tr>
<tr>
<td>MBNetV2</td>
<td>73</td>
<td>16M</td>
</tr>
<tr>
<td>ShuffleNet InceptionV</td>
<td>75</td>
<td>32M</td>
</tr>
<tr>
<td>DenseNet-121</td>
<td>77</td>
<td>64M</td>
</tr>
<tr>
<td>DenseNet-169</td>
<td>79</td>
<td>2M</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>79</td>
<td>4M</td>
</tr>
<tr>
<td>ResNetXt-5</td>
<td>81</td>
<td>8M</td>
</tr>
<tr>
<td>InceptionV</td>
<td>79</td>
<td>16M</td>
</tr>
<tr>
<td>DPN-92</td>
<td>79</td>
<td>32M</td>
</tr>
<tr>
<td>DenseNet-26</td>
<td>79</td>
<td>64M</td>
</tr>
<tr>
<td>DenseNet-169</td>
<td>79</td>
<td>2M</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>79</td>
<td>4M</td>
</tr>
</tbody>
</table>

Figures from Once-for-all project page.
Today’s AI is too BIG
Better model always comes with higher computational cost (NLP)

NLP model size is increasing exponentially

175 Billion model parameters
8 Million web pages
3 Million GPU hours*

*Measured on Nvidia A100
Figures from Microsoft Turing Project
Deep Learning Going “Tiny”

Cloud → Mobile → Tiny

Cloud AI
GPUs/TPUs
ResNet

- Data uploaded to the cloud for inference/training
Deep Learning Going “Tiny”

Cloud → Mobile → Tiny

Cloud AI
- GPUs/TPUs
- ResNet

Mobile AI
- Smartphones
- MobileNet

Tiny AI
- IoT/Microcontrollers
- MCUNet
Deep Learning Going “Tiny”

Squeezing deep learning into IoT devices

- Billions of IoT devices around the world based on **microcontrollers**
- **Low-cost**: low-income people can afford access. Democratize AI.
- **Low-power**: green AI, reduce carbon
Deep Learning Going “Tiny”

Squeezing deep learning into IoT devices

• Billions of IoT devices around the world based on microcontrollers
• Low-cost: low-income people can afford access. Democratize AI.
• Low-power: green AI, reduce carbon
• Various applications
TinyML is Challenging
Memory size is too small to hold DNNs

<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>
TinyML is Challenging

Memory size is too small to hold DNNs

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

| Storage (Weights)  | ~TB/PB   | 256GB     | 1MB     |

- 100,000x smaller
- 13,000x smaller
Overview

MCUNetV1
Tiny Image Recognition

MCUNetV2
Higher Resolution for Object detection, etc.

MCUNetV3
Tiny On-Device Training

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning [Lin et al., NeurIPS 2021]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]
MCUNetV1 - Classification

Tiny vision application: visual wake words

Visual wake words dataset. [Chowdhery et al., arXiv 2019]
MCUNetV2: Detection

Advancing object detection by allowing a larger resolution

Face/mask detection

Person detection
Overview

MCUNetV1
Tiny Image Recognition

MCUNetV2
Higher Resolution for Object detection, etc.

MCUNetV3
Tiny On-Device Training

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning [Lin et al., NeurIPS 2021]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]
Overview

Co-design

MCUNetV1
Tiny Image Recognition

MCUNetV2
Higher Resolution for Object detection, etc.

MCUNetV3
Tiny On-Device Training

System

Algorithm

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning [Lin et al., NeurIPS 2021]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]
Tiny On-Device Training

- Sparse Update
- Tiny Training Engine (TTE)
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**, empower **AIoT with limited connectivity**

data cannot be sent to the cloud for privacy reason
Can We Learn on the Edge?

From tinyML inference to training

A virtuous cycle:

- On-device learning:
  - customization by adapting to user data / life-long learning
  - better privacy, lower cost, empower AIoT with limited connectivity
- Training is more expensive than inference
  - For example, store intermediate activation, extra back-propagation, etc.
Training Memory is the Key Bottleneck

- Edge devices have tight memory constraints. The training memory footprint of neural networks can easily exceed the limit.

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai et al., NeurIPS 2020]
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>Platform</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow (cloud)</td>
<td>652 MB</td>
</tr>
<tr>
<td>PyTorch (cloud)</td>
<td>303 MB</td>
</tr>
<tr>
<td>MNN (edge)</td>
<td>41.5 MB</td>
</tr>
</tbody>
</table>

- Quantization-aware scaling
- Sparse layer/tensor update

256KB constraint

2.9 MB

355 KB

7.3x

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).

<table>
<thead>
<tr>
<th>Model</th>
<th>Cloud Memory</th>
<th>Edge Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow (cloud)</td>
<td>652 MB</td>
<td>303 MB</td>
</tr>
<tr>
<td>PyTorch (cloud)</td>
<td>303 MB</td>
<td>41.5 MB</td>
</tr>
<tr>
<td>MNN (edge)</td>
<td>256KB constraint</td>
<td>5.7 MB</td>
</tr>
<tr>
<td>Tiny Training Engine</td>
<td>355 KB</td>
<td>2.9 MB</td>
</tr>
<tr>
<td>+ Quantization-aware scaling</td>
<td>141 KB</td>
<td>2.0x</td>
</tr>
<tr>
<td>+ Sparse layer/tensor update</td>
<td>2.4x</td>
<td>8.8x</td>
</tr>
<tr>
<td>+ Operator reordering</td>
<td>7.3x</td>
<td>2300x</td>
</tr>
</tbody>
</table>

https://tinytraining.mit.edu
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, but are hard to quantize

(a) Fake Quantization (quantization aware training)

Most intermediate tensors are still in FP32 format in fake quantization, thus cannot save memory footprint
1. Quantization-Aware Scaling (QAS)

Real quantized graphs save memory, but are hard to quantize

(a) Fake Quantization (quantization aware training)

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

All tensors are in **int8/int32 format** for real quantization, thus save memory footprint, but leading to optimization difficulty
1. Quantization-Aware Scaling (QAS)

Quantized graphs save memory, but are hard to quantize

Difficult optimize:
- 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!
1. Quantization-Aware Scaling (QAS)

QAS addresses the optimization difficulty of quantized graphs

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

Per Channel scaling

\[
W = s_W \cdot (W/s_W) \overset{\text{quantize}}{\approx} s_W \cdot \tilde{W}, \quad G_W \approx s_W \cdot G_W,
\]

Weight and gradient ratios are off by \(s_W^{-2}\)

\[
\frac{||\tilde{W}||}{||G_W||} \approx \frac{||W/s_W||}{||s_W \cdot G_W||} = s_W^{-2} \cdot \frac{||W||}{||G||}.
\]

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. Quantization-Aware Scaling (QAS)

QAS addresses the optimization difficulty of quantized graphs

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

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

QAS addresses the optimization difficulty of quantized graphs.

Without QAS, poor convergence.

With QAS, better convergence.

After applying QAS, the convergence of real quantized is stable.
1. Quantization-Aware Scaling (QAS)

QAS addresses the optimization difficulty of quantized graphs

QAS improves the accuracy over naive int8 training, and shows no inferior performance than fp32 results.
On-Device Training Under 256KB Memory

1. Quantization-aware scaling
2. Sparse layer/tensor update
3. Tiny Training Engine
Training Memory is the Key Bottleneck

Question: Why training memory is much larger than inference?

Answer: Because of intermediate activations

Forward: \[ a_{i+1} = a_i W_i + b_i \]

Backward: \[ \frac{\partial L}{\partial W_i} = a_i^T \frac{\partial L}{\partial a_{i+1}} \]

- Inference does not need to store activations, training does.
- Activations grows linearly with batch size, which is always 1 for inference.
- Even with bs=1, activations are usually larger than model weights.

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai et al., NeurIPS 2020]
Training Memory is the Key Bottleneck

- Activation is the main bottleneck for on-device learning, not parameters.

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai et al., NeurIPS 2020]
• Activation is the main bottleneck for on-device learning, not parameters.
• Previous methods focus on reducing the number of parameters or FLOPs, while the main bottleneck does not improve much.

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai et al., NeurIPS 2020]
2. Sparse Layer/Tensor Update

**Full update**

<table>
<thead>
<tr>
<th>Layer</th>
<th>MB1 3x3</th>
<th>MB3 5x3</th>
<th>MB3 5x3</th>
<th>MB3 5x3</th>
<th>MB6 7x7</th>
<th>MB3 5x3</th>
<th>MB3 5x3</th>
<th>MB3 5x3</th>
<th>MB3 5x3</th>
<th>MB6 7x7</th>
<th>MB3 5x3</th>
<th>MB3 5x3</th>
<th>MB6 7x7</th>
<th>MB3 5x3</th>
<th>MB3 5x3</th>
<th>MB6 7x7</th>
<th>MB3 5x3</th>
<th>MB3 7x7</th>
<th>MB3 5x5</th>
</tr>
</thead>
</table>

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)

<table>
<thead>
<tr>
<th>Memory Cost (MB)</th>
<th>0</th>
<th>80</th>
<th>160</th>
<th>240</th>
<th>320</th>
<th>400</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cars Top1 (%)</td>
<td>50</td>
<td>61</td>
<td>72</td>
<td>83</td>
<td>94</td>
<td></td>
</tr>
</tbody>
</table>

Model: ProxylessNAS-Mobile

Far beyond the on-device learning capacity
2. Sparse Layer/Tensor Update

**Last layer update**

- Updating only the last cheap
  - No need to back propagating to previous layers
  - But the accuracy is low and not ideal.

### Memory Cost (MB)

- **Full**
- **Last**
- **Bias+Last**

### Cars Top1 (%)

- **Full**
- **Last**
- **Bias+Last**

**Significant accuracy degradation!**
2. Sparse Layer/Tensor Update

Bias-only update

Updating the only the bias part
- No need to store the activations
- Back propagating to the first layer.

Forward: \( a_{i+1} = a_i W_i + b_i \)

Backward:
\[
\begin{align*}
\frac{\partial L}{\partial W_i} &= a_i^T \frac{\partial L}{\partial a_{i+1}}, \\
\frac{\partial L}{\partial b_i} &= \frac{\partial L}{\partial a_{i+1}} = \frac{\partial L}{\partial a_{i+2}} W_{i+1}^T
\end{align*}
\]

Model: ProxylessNAS-Mobile

Still a performance gap
2. Sparse Layer/Tensor Update

Updated synapses are sparse

- 2500 synapses per neuron
- 15000 synapses per neuron
- 7000 synapses per neuron

Synapses are getting "sparse"

Newborn | 2-4 years old | Adolescence | Adult
---|---|---|---
2500 synapses per neuron | 15000 synapses per neuron | 7000 synapses per neuron | K-12 education

Data Source:
1. Do We Have Brain to Spare? [Drachman DA, Neurology 2004]

Time

Data Source: 1, 2
Slide Inspiration: Alila Medical Media
2. Sparse Layer/Tensor Update

Some layers are more important than others

![Graph showing relative accuracy gain for different channel updates](image-url)
2. Sparse Layer/Tensor Update

Some layers are more important than others

1. Later layers contribute more to the accuracy.
2. Sparse Layer/Tensor Update

Some layers are more important than others

1. Later layers contribute more to the accuracy.
2. First point-wise conv are more important to accuracy.
2. Sparse Layer/Tensor Update

Some layers are more important than others

1. Later layers contribute more to the accuracy.
2. First point-wise conv are more important to accuracy.
3. The more channels being updated, the higher the accuracy.
2. Sparse Layer/Tensor Update

Sparse Layer/Tensor Update

Model: ProxylessNAS-Mobile
2. Sparse Layer/Tensor Update

Sparse layer backpropagation

Model: ProxylessNAS-Mobile

- Sparse layer update: no need to store activation
2. Sparse Layer/Tensor Update

Sparse Layer/Tensor Update

- **Sparse layer update**: no need to store activation
- **Sparse tensor update**: only store a subset of the activations.

\[
\frac{dy}{dw} : \begin{align*}
& (H, N) \\
& (N, M) \\
& \text{G.T} \\
& \times \\
\end{align*} = \begin{align*}
& (H, M) \\
& (dW).T \\
\end{align*}

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

\[
\frac{dy}{dw} : \begin{align*}
& (H, N) \\
& (N, 0.25*M) \\
& \text{G.T} \\
& \times \\
\end{align*} = \begin{align*}
& (H, M) \\
& (dw).T \\
\end{align*}

Reduce by 4x

Activation to store: \((N, 0.25*M)\)
Weight in SRAM: \((0.25*M, H)\)
2. Sparse Layer/Tensor Update

- **Sparse layer update**: no need to store activation
- **Sparse tensor update**: only store a subset of the activations.
- **Sparse update**: no need to back propagate the early layers

Model: ProxylessNAS-Mobile
2. Sparse Layer/Tensor Update

Sparse Layer/Tensor Update

Model: ProxylessNAS-Mobile

Backpropagation stops here

Sparse tensor backpropagation

Sparse layer backpropagation

- update last k biases
- update last k layers
- sparse update (ours)

Average Acc (%) vs. Extra Memory (KB)

(a) MCUNet-5FPS

4.5× smaller higher acc

≤ 50kB

≤ 75kB

≤ 100kB

(b) MbV2-w0.35

7.5× smaller

69 71 73 75 77

40 155 270 385 500

(c) Proxyless-w0.3

7.1× smaller

69 71 73 75 77

40 110 180 250 320 524

≤ 50kB

≤ 75kB

≤ 100kB

≤ 150kB

≤ 250kB

≤ 320kB

≤ 524kB
Update Paradigms Comparison

(a) Full update

(b) Last-only update

(c) Bias-only update

(d) Sparse layer/Sparse tensor update
On-Device Training Under 256KB Memory

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

https://tinytraining.mit.edu
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

### Memory Usage Comparison

<table>
<thead>
<tr>
<th>Framework</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow (cloud)</td>
<td>652 MB</td>
</tr>
<tr>
<td>PyTorch (cloud)</td>
<td>303 MB</td>
</tr>
<tr>
<td>MNN (edge)</td>
<td>41.5 MB</td>
</tr>
</tbody>
</table>

256KB constraint
3. Tiny Training Engine (TTE)

Workflow of conventional training engine

1. Computation Graph (forward)

   Data → MatMul → Out

   Weight → MatMul → Out
3. Tiny Training Engine (TTE)

Workflow of conventional training engine

1. Computation Graph (forward)
   - Data
   - Weight
   - MatMul
   - Out

2. AutoDiff
   - $f(x) \rightarrow f'(x)$

3. Computation Graph (backward)
   - dy/dx
   - grad
   - dy/dw

3. Tiny Training Engine (TTE)

Workflow of conventional training engine

1. Computation Graph (forward)
   - Data
   - Weight
   - MatMul → Out

2. AutoDiff
   - \( f(x) \rightarrow f'(x) \)

3. Computation Graph (backward)
   - dy/dx
   - dy/dw
   - MatMul’ → grad

4. Execution Engine
3. Tiny Training Engine (TTE)

Workflow of conventional training engine

Conventional training framework focus on **flexibility**, and the auto-diff is performed at **runtime**.

Thus, any optimizations will lead to runtime overhead.
3. Tiny Training Engine (TTE)

TTE: Move workload from runtime to compile time

TTE moves most workload from runtime to **compile-time**, thus minimizes the **runtime overhead**, also enables opportunities for **extensive graph optimizations**.
3. Tiny Training Engine (TTE)

```plaintext
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)
}
```

Example from a matrix multiplication with full update

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

Example from a matrix

\[
\begin{align*}
W_i W_i + 1 & & b_i b_i + 1 \\
n & & n
\end{align*}
\]

Updated

Fixed

\[
y = mul(x, w) + b
\]

\[
\frac{dy}{dx} = mul(G, w)
\]

\[
\frac{dy}{dw} = mul(G^T, X)
\]

\[
\frac{dy}{db} = sum(G)
\]
3. Tiny Training Engine (TTE)

Annotate whether a tensor requires gradient or not

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

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

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

\{
    # forward
    y = \text{mul}(x, w) + b
    %0 = \text{multiply}(%x, %weight);
    %1 = \text{add}(%0, %bias);

    # backward
    \frac{dy}{dx} = \text{mul}(G, w)
    %3 = \text{multiply}(%\text{grad}, %weight); \quad \Rightarrow \quad \frac{dy}{dx}
    %4 = \text{transpose}(%\text{grad});
    \frac{dy}{dw} = \text{mul}(G^T, X)
    %5 = \text{multiply}(%4, %x); \quad \Rightarrow \quad \frac{dy}{dw}
    \frac{dy}{db} = \text{sum}(G)
    %6 = \text{sum}(%\text{grad}, \text{axis}=-1); \quad \Rightarrow \quad \frac{dy}{db}
    (%3, %5, %6)
\}

Updated
Fixed

(a) full update
(b) bias-only update
(c) sparse layer update
(d) sparse tensor update
3. Tiny Training Engine (TTE)

Remove unnecessary computations from DAG via dependency analysis and dead-code elimination.

```plaintext
fn (%x: Tensor[(10, 10), float32, needs_grad=True],
    %weight: Tensor[(10, 10), float32, needs_grad=False],
    %bias: Tensor[(10), float32, needs_grad=False],
    %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)
}
```
3. Tiny Training Engine (TTE)

Freely annotate ANY parameters
TTE will trim the computation accordingly.
3. Tiny Training Engine (TTE)

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

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

\[
\begin{align*}
\text{fn} (\%x: \text{Tensor}[\langle 10, 10 \rangle, \text{float32}], \\
\text{weight}: \text{Tensor}[\langle 10, 10 \rangle, \text{float32}], \\
\text{bias}: \text{Tensor}[\langle 10 \rangle, \text{float32}], \\
\text{grad}: \text{Tensor}[\langle 10 \rangle, \text{float32}]), \\
\{
\# \text{ forward}
\%0 = \text{multiply}(\%x, \text{weight}); \\
\%1 = \text{add}(\%0, \text{bias}); \\
\# \text{ backward}
\%3 = \text{multiply}(\text{grad}, \text{weight}); \\
\%4 = \text{transpose}(\text{grad}) \\
\%5 = \text{multiply}(\%4, \%x); \\
\%6 = \text{sum}(\text{grad}, \text{axis}=-1); \\
(\%3, \%5, \%6)
\}
\end{align*}
\]

\[
\begin{align*}
\text{fn} (\%x: \text{Tensor}[\langle 10, 10 \rangle, \text{float32}], \\
\text{weight}: \text{Tensor}[\langle 10, 10 \rangle, \text{float32}], \\
\text{bias}: \text{Tensor}[\langle 10 \rangle, \text{float32}], \\
\text{grad}: \text{Tensor}[\langle 10 \rangle, \text{float32}]), \\
\{
\# \text{ forward}
\%0 = \text{multiply}(\%x, \text{weight}); \\
\%0.1 = \text{slice}(\%x, \text{begin}=[0, 0], \text{ends}=[10, 10]); \\
\%1 = \text{add}(\%0, \text{bias}); \\
\# \text{ backward}
\%3 = \text{multiply}(\text{grad}, \text{weight}); \\
\%4 = \text{transpose}(\text{grad}) \\
\%5 = \text{multiply}(\%4, \%0.1); \\
\%6 = \text{sum}(\text{grad}, \text{axis}=-1); \\
(\%3, \%5, \%6)
\}
\end{align*}
\]
3. Tiny Training Engine (TTE)

Sparse update results

- Tiny Training Engine supports backward graph pruning and sparse update at IR-level.
- After graph pruning, un-used weights and sub-tensors are pruned from DAG => 6.5-8.7x memory saving
3. Tiny Training Engine (TTE)

Re-ordering reduces memory footprint

- Tiny Training Engine supports backward graph pruning and sparse update at IR-level.
- After graph pruning, un-used weights and sub-tensors are pruned from DAG => 6.5-8.7x memory saving

(a) Conventional way to update parameters

F: Forward, B: Backward, U: Update
3. Tiny Training Engine (TTE)

Re-ordering reduces memory footprint

- Tiny Training Engine supports backward graph pruning and sparse update at IR-level.
- After graph pruning, un-used weights and sub-tensors are pruned from DAG => 6.5-8.7x memory saving

Operator life-cycle analysis reveals the memory redundancy in the optimization step.

After re-ordering, the redundant memory usage is eliminated from training.

F: Forward, B: Backward, U: Update
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

(a) Peak memory vs. models

(b) Training latency vs. models

20x smaller memory

23x faster speed
Co-design reduces the training memory by 2300x times with the same transfer accuracy.
The numbers are measured with MobilenetV2-w0.35, batch size 1 and resolution 128x128.
2. On-device training

https://www.bilibili.com/video/BV1qv4y1d7MV/

https://youtu.be/XaDCO8YtmBw
Extending TTE to More Platforms

Accelerate on-device training on diverse edge hardware

- We extend TTE to support:
  - Diverse models (CNN + Transformers)
  - Diverse frontends
    - PyTorch
    - TensorFlow
    - Jax
  - Diverse hardware backends
    - Apple M1
    - Raspberry Pi
    - Smartphones
    - …
Extending TTE to More Platforms

Consistently speed up training on diverse platforms

- TTE provides a systematic support for sparse update schemes for vision and NLP models, leading to consistent memory saving at the same training accuracy

![Graph showing data/sec for different platforms]

- Results measured on Raspberry Pi 4B+.
System brings deep learning to “internet of things” devices
Advances could enable artificial intelligence on household appliances while enhancing data security and energy efficiency.

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning [Lin et al., NeurIPS 2021]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]
MCUNet: Tiny Deep Learning on Microcontrollers

This is the official implementation of MCUNet, a system-algorithm co-design framework for tiny deep learning on microcontrollers. TinyEngine is a part of MCUNet, an open-source system-algorithm co-design framework for tiny deep learning on microcontrollers.

The MCUNet and TinyNAS repos are [here](https://forms.gle/UW1uUmnfk1k6UJPPA).

Sign up here to get updates!

https://forms.gle/UW1uUmnfk1k6UJPPA
Future Work

• Scale up to **LLM/foundation models**
  • LLM models are hard to serve/fine-tune due to the huge model size
  • GPU memories are not enough to serve 100 billion-parameter models
  • Our techniques help democratize LLMs (e.g., quantization, sparse update, system support)

• **Collaboration welcome!**