MIT HAN LAB

Lecture 15
On-Device Training and Transfer Learning

Part I

Song Han
songhan@mit.edu

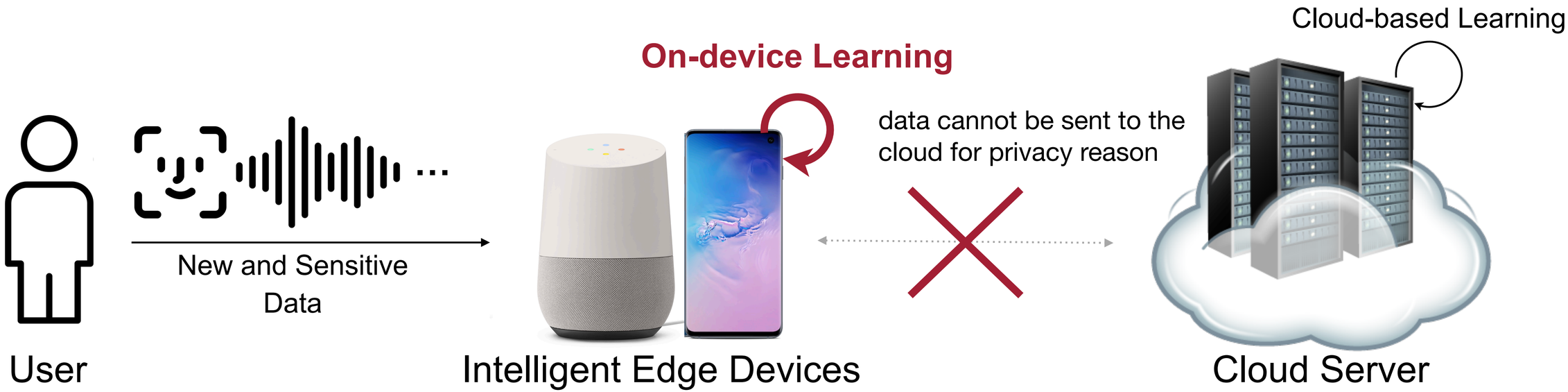
Adapt to New Data on Edge

- Customization: AI systems need to continually adapt to new data collected from the sensors.

Cloud-Based Learning

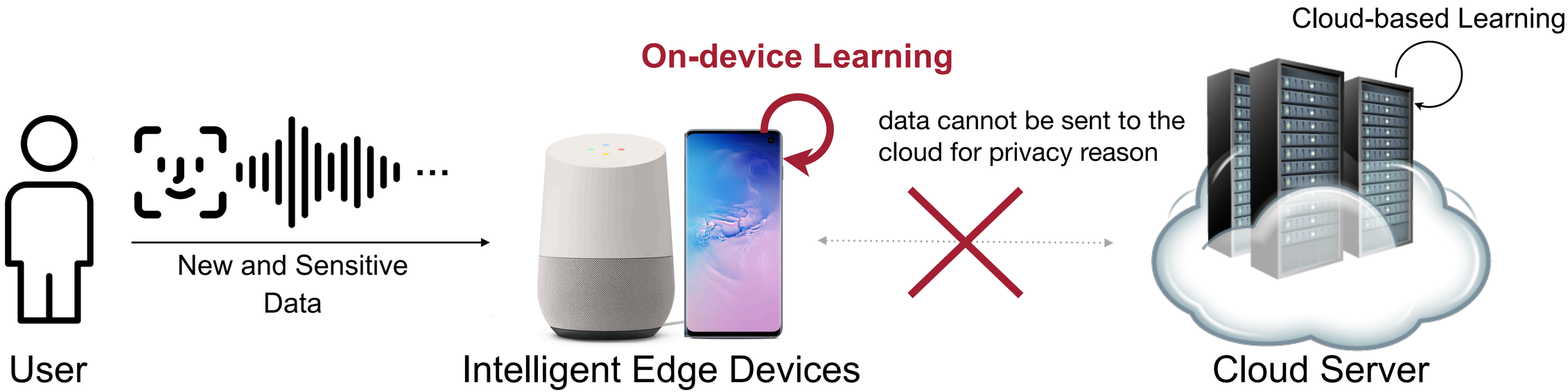
- Customization: AI systems need to continually adapt to new data collected from the sensors.

On-Device Learning



- Customization: AI systems need to continually adapt to new data collected from the sensors.
- Security: Data cannot leave devices because of security and regularization.

On-Device Learning



- Customization: AI systems need to continually adapt to new data collected from the sensors.
- Security: Data cannot leave devices because of security and regularization.

On-device learning: **better privacy, lower cost, customization, life-long learning**

On-Device Learning Applications

Squeezing training into IoT devices

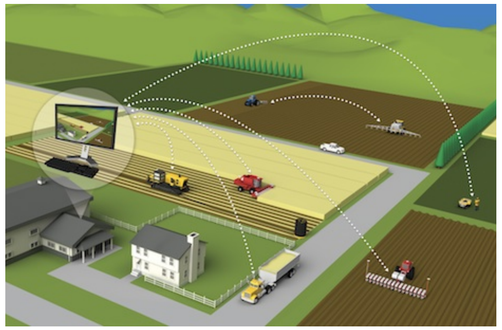
- Billions of IoT devices around the world based on **microcontrollers**
- **Low-cost**: low-income people can afford access. Democratize AI.
- **Low-power**: **green AI**, reduce carbon

Smart Home

Smart Manufacturing

Personalized Healthcare

Precision Agriculture



Lecture Plan

Today we will:

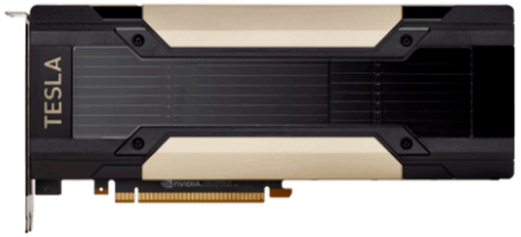
1. Analyze the memory bottleneck of on-device training.
2. Introduce efficient algorithms for on-device transfer learning.
3. Study the system support for efficient on-device training.

Section 1: Efficient On-device Learning Algorithms

Understand the training bottleneck and improve training efficiency.

On-Device Training is Challenging

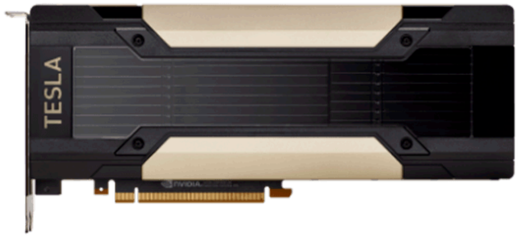
Memory size is too small to hold DNNs



	Cloud AI	Mobile AI
Memory (Activation)	32GB	4GB
Storage (Weights)	~TB/PB	256GB

On-Device Training is Challenging

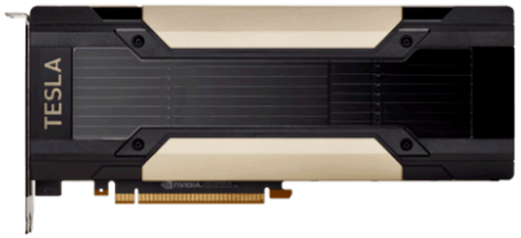
Memory size is too small to hold DNNs



	Cloud AI	Mobile AI	Tiny AI
Memory (Activation)	32GB	4GB	320kB
Storage (Weights)	~TB/PB	256GB	1MB

On-Device Training is Challenging

Memory size is too small to hold DNNs



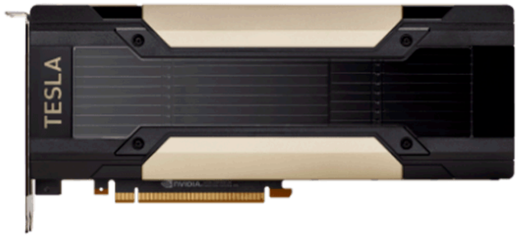
	Cloud AI	Mobile AI	Tiny AI
Memory (Activation)	32GB	4GB	320kB
Storage (Weights)	~TB/PB	256GB	1MB

13,000x smaller

100,000x smaller

On-Device Training is Challenging

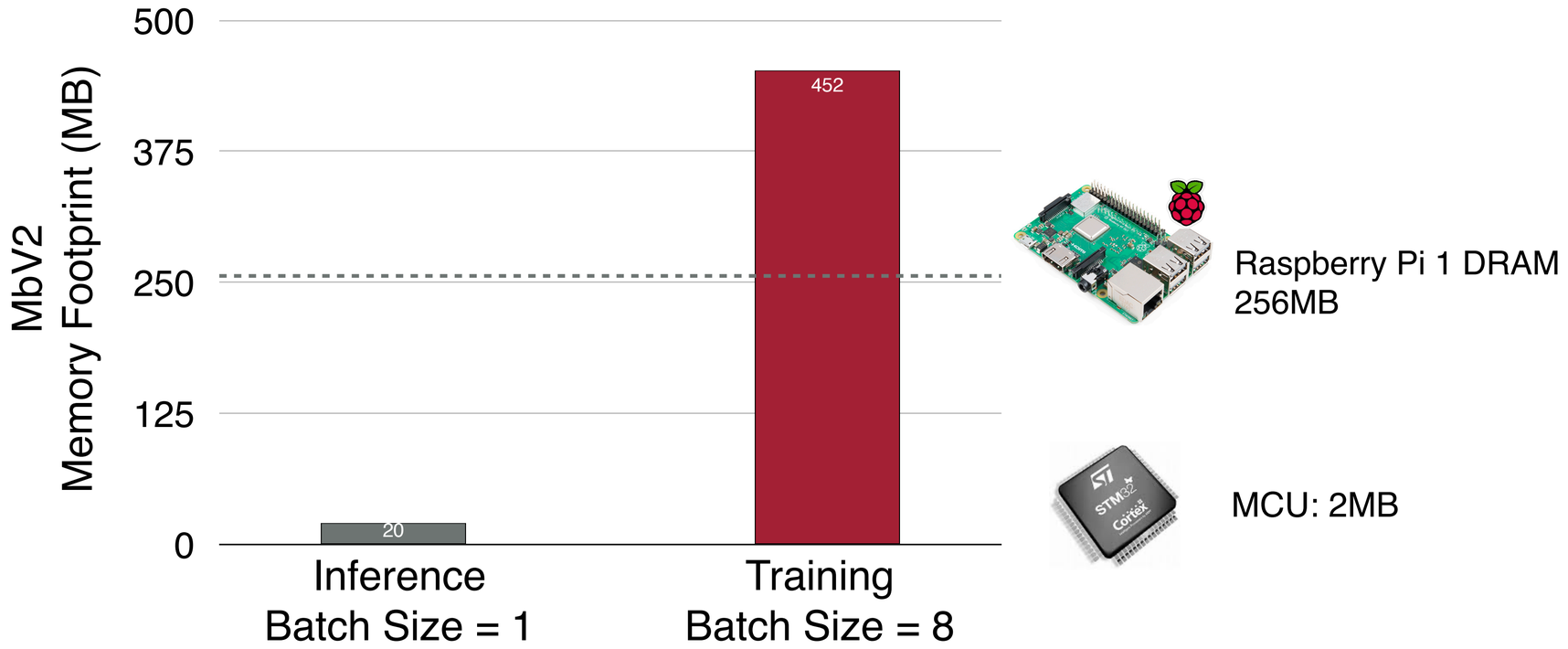
Memory size is too small to hold DNNs



	Cloud AI	Mobile AI	Tiny AI
Memory (Activation)	32GB	4GB	320kB
Storage (Weights)	~TB/PB	256GB	1MB

- We need to reduce both **weights** and **activation** to fit DNNs for On-Device Training

Training Memory is the Key Bottleneck



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

Question: Why training memory is much larger than inference?

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai *et al.*, NeurIPS 2020]

Training Memory is the Key Bottleneck

Question: Why training memory is much larger than inference?

Answer: Because of intermediate **activations**

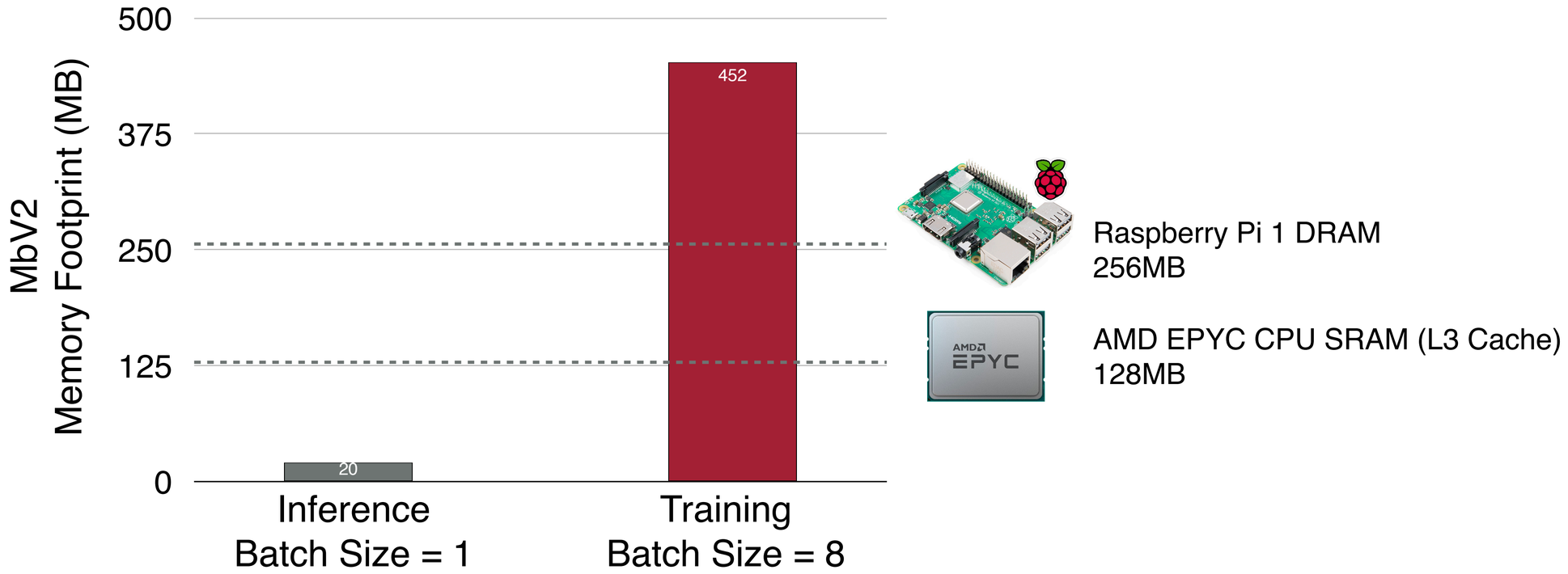
Forward: $\mathbf{a}_{i+1} = \mathbf{a}_i \mathbf{W}_i + \mathbf{b}_i$

Backward: $\frac{\partial L}{\partial \mathbf{W}_i} = \mathbf{a}_i^T \frac{\partial L}{\partial \mathbf{a}_{i+1}}$

- 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]

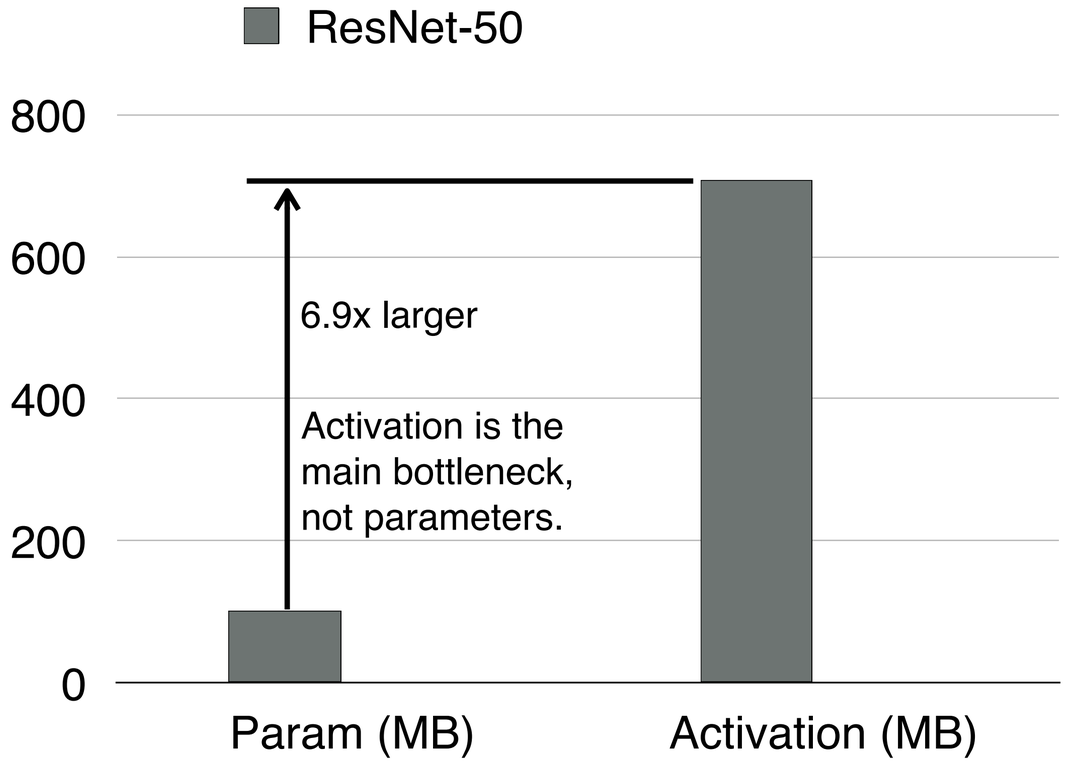
Training Memory is the Key Bottleneck



- Edge devices have tight memory constraints. The training memory footprint of neural networks can easily exceed the limit.
- Edge devices are energy-constrained. Failing to fit the training process into the energy-efficient cache will significantly increase the energy cost.

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai *et al.*, NeurIPS 2020]

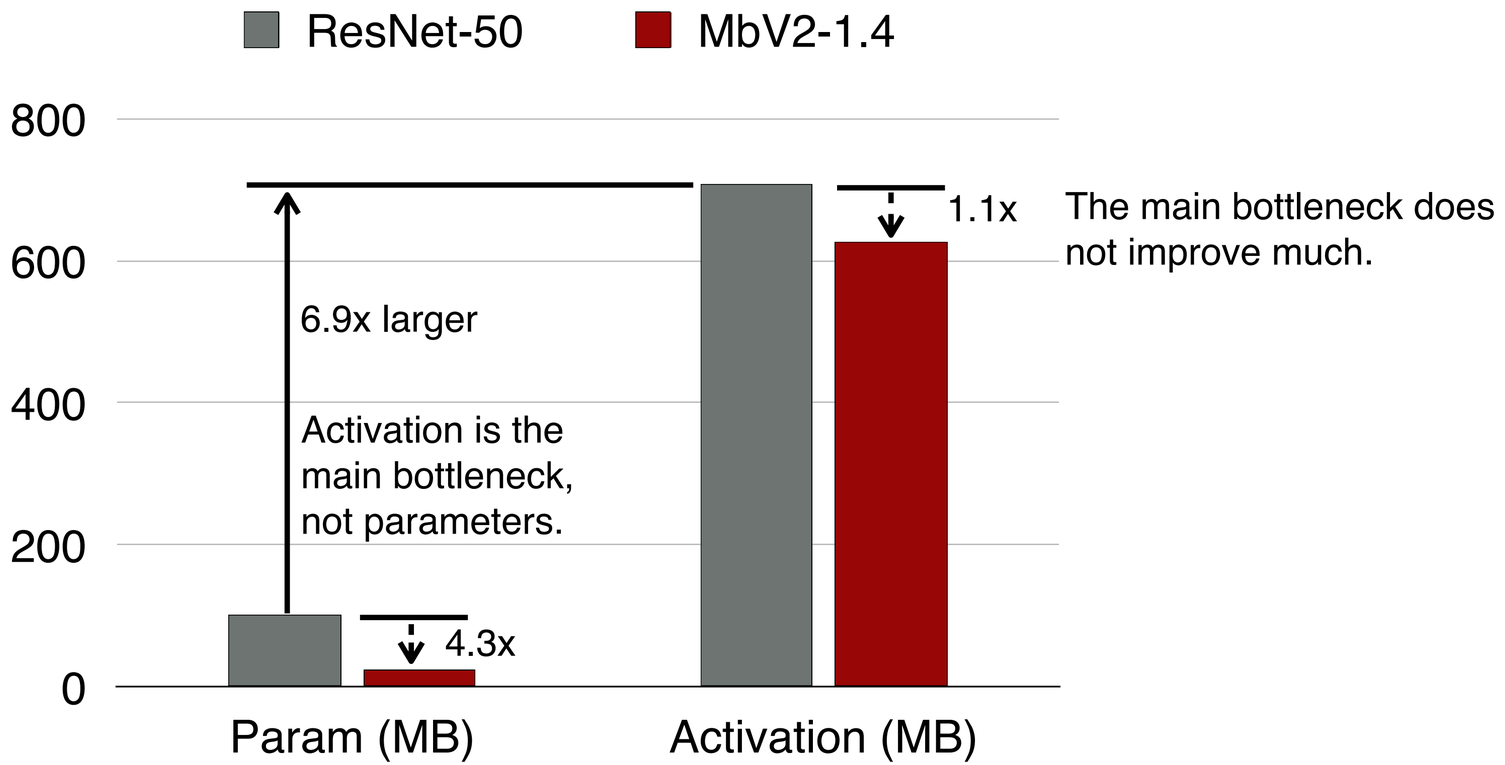
Activation is the Memory Bottleneck



- 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]

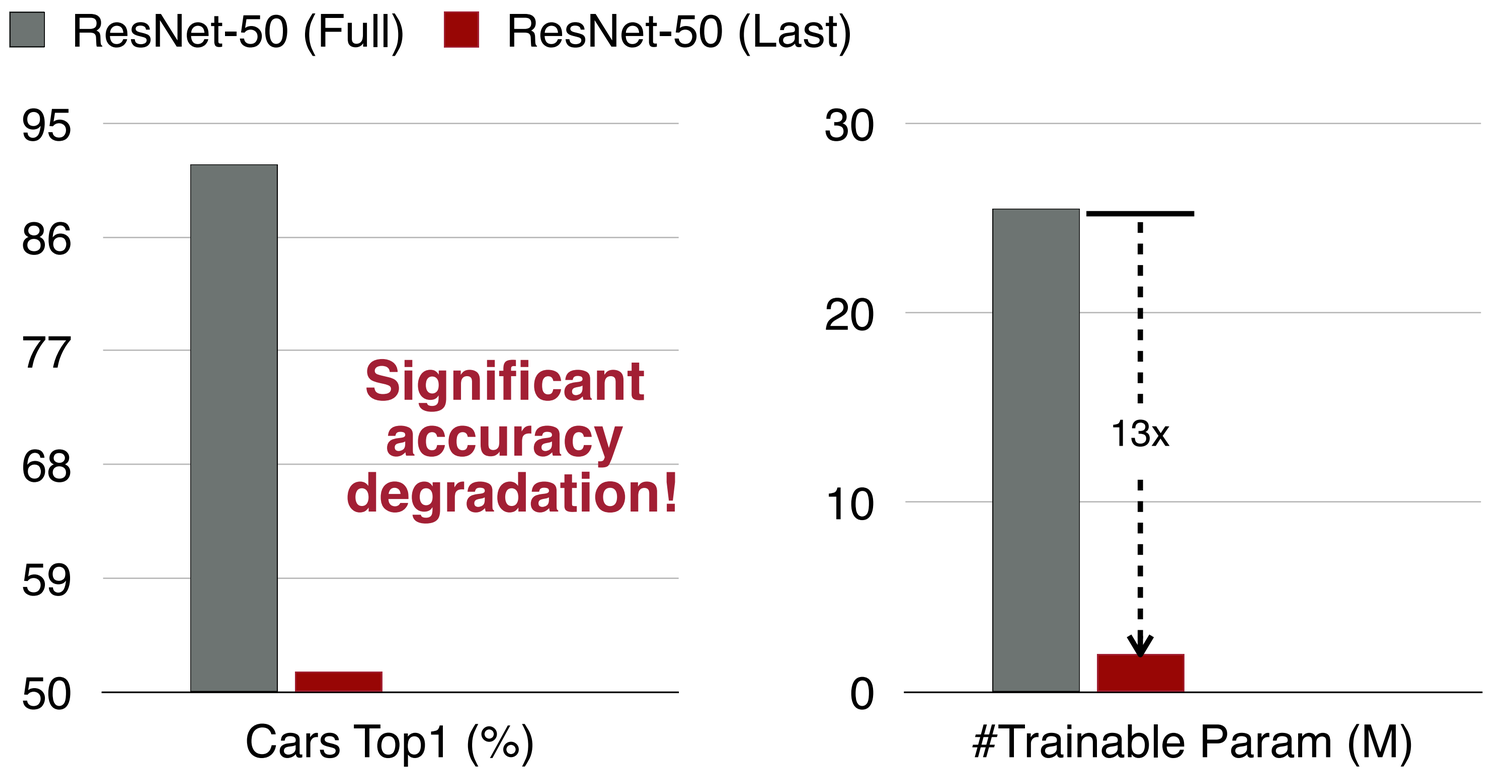
Activation is the Memory Bottleneck



- 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.

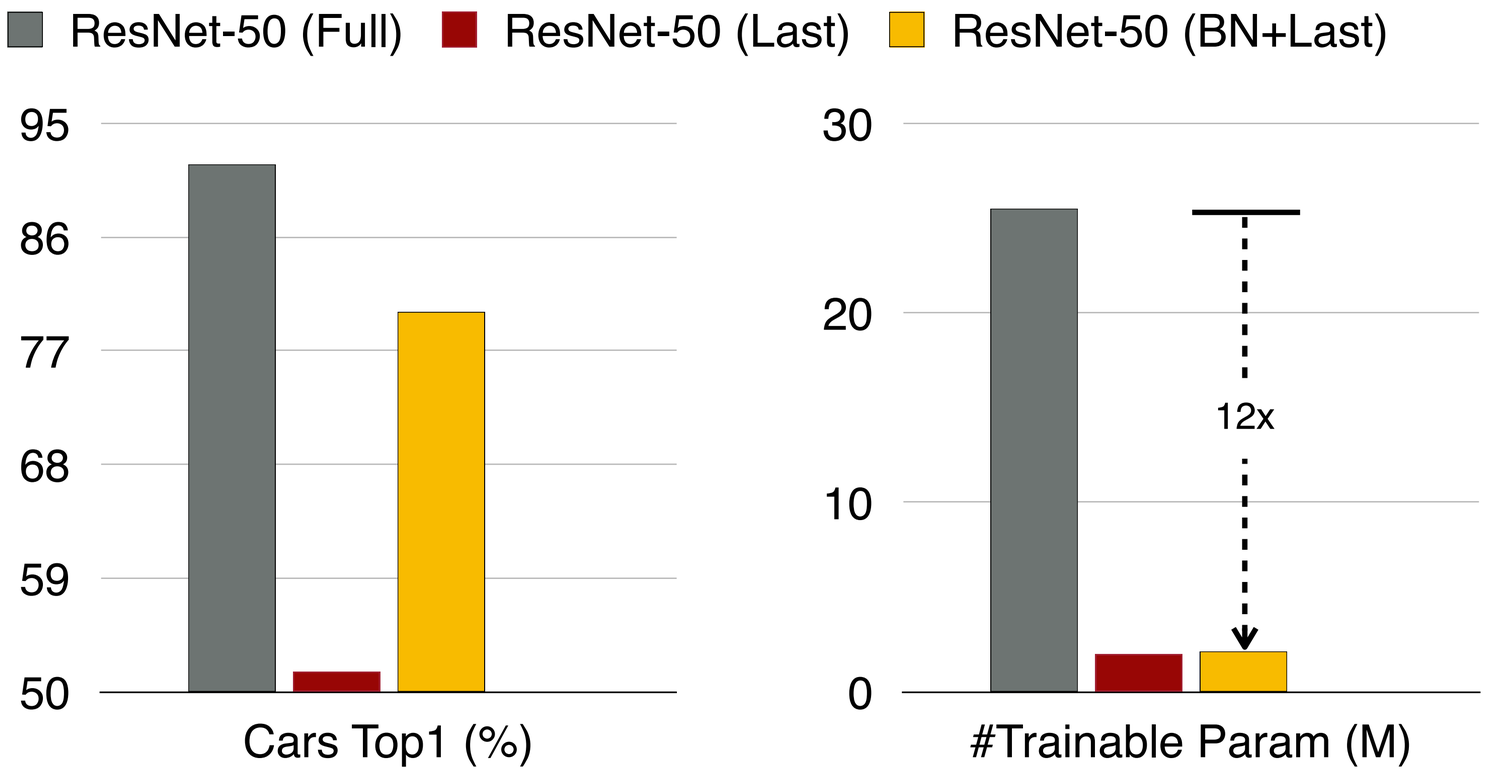
TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai *et al.*, NeurIPS 2020]

Parameter-Efficient Transfer Learning



- Full: Fine-tune the full network. Better accuracy but highly inefficient.
- Last: Only fine-tune the last classifier head. Efficient but the capacity is limited.

Parameter-Efficient Transfer Learning

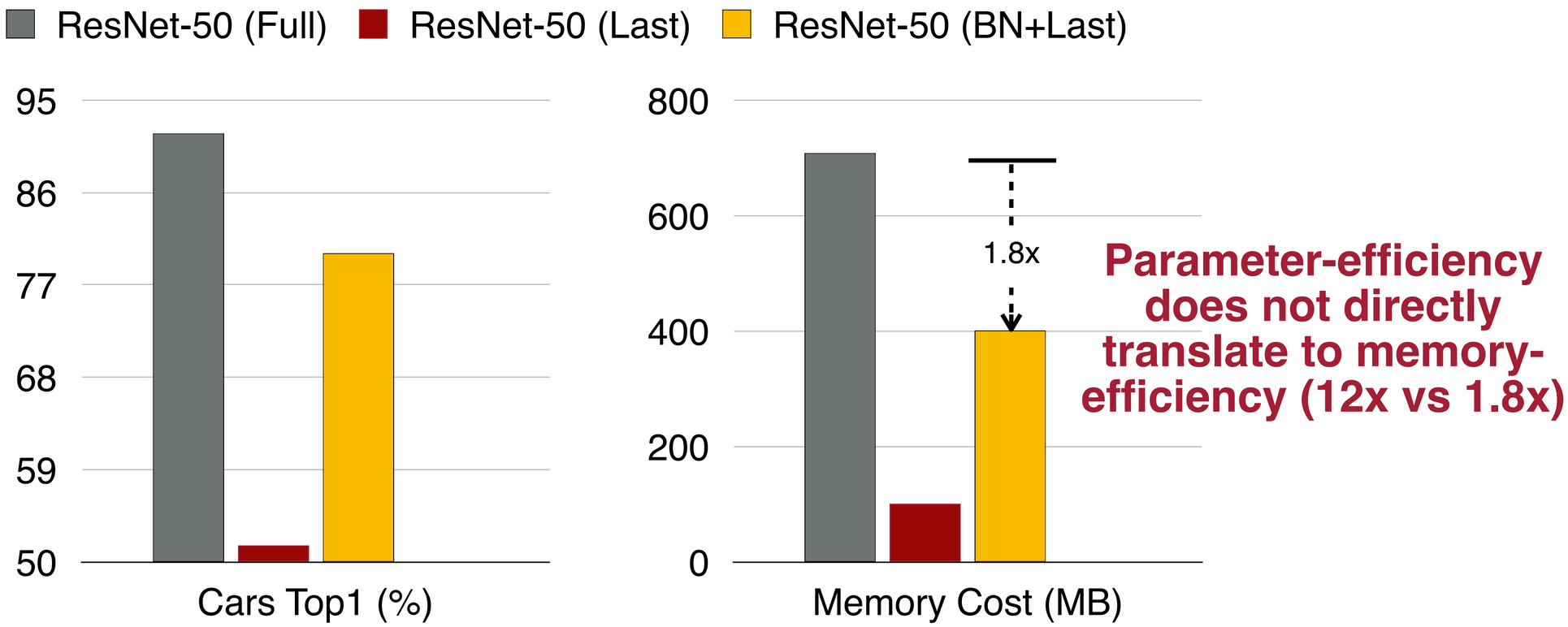


- Full: Fine-tune the full network. Better accuracy but highly inefficient.
- Last: Only fine-tune the last classifier head. Efficient but the capacity is limited.
- BN+Last: Fine-tune the BN layers and the last layer. Parameter-efficient.

Question: Is BN+Last update or Last-only update enough for on-device transfer learning?

K for the Price of 1: Parameter-efficient Multi-task and Transfer Learning [Mudrarkarta *et al.*, ICLR 2019]

Parameter-Efficient Transfer Learning

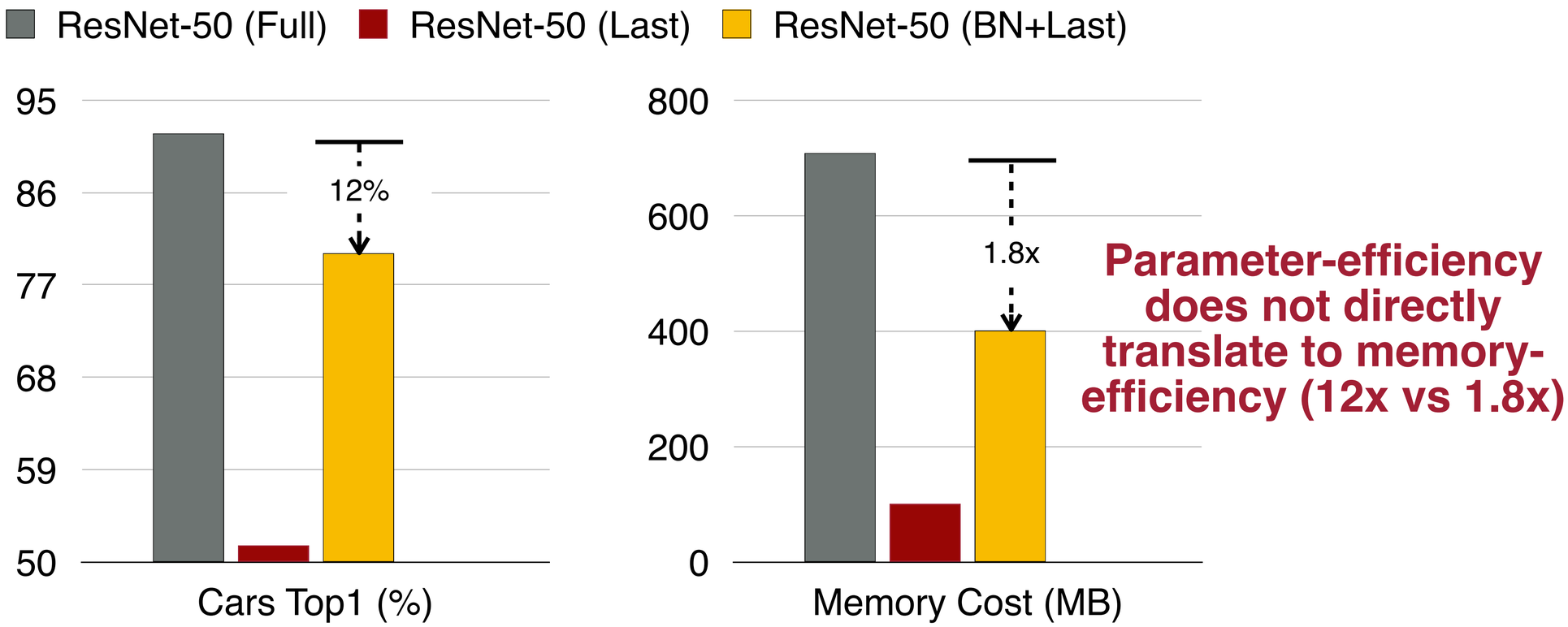


Parameter-efficiency does not directly translate to memory-efficiency (12x vs 1.8x)

- Full: Fine-tune the full network. Better accuracy but highly inefficient.
- Last: Only fine-tune the last classifier head. Efficient but the capacity is limited.
- BN+Last: Fine-tune the BN layers and the last layer. Parameter-efficient, **but the memory saving is limited.**

K for the Price of 1: Parameter-efficient Multi-task and Transfer Learning [Mudrarkarta *et al.*, ICLR 2019]

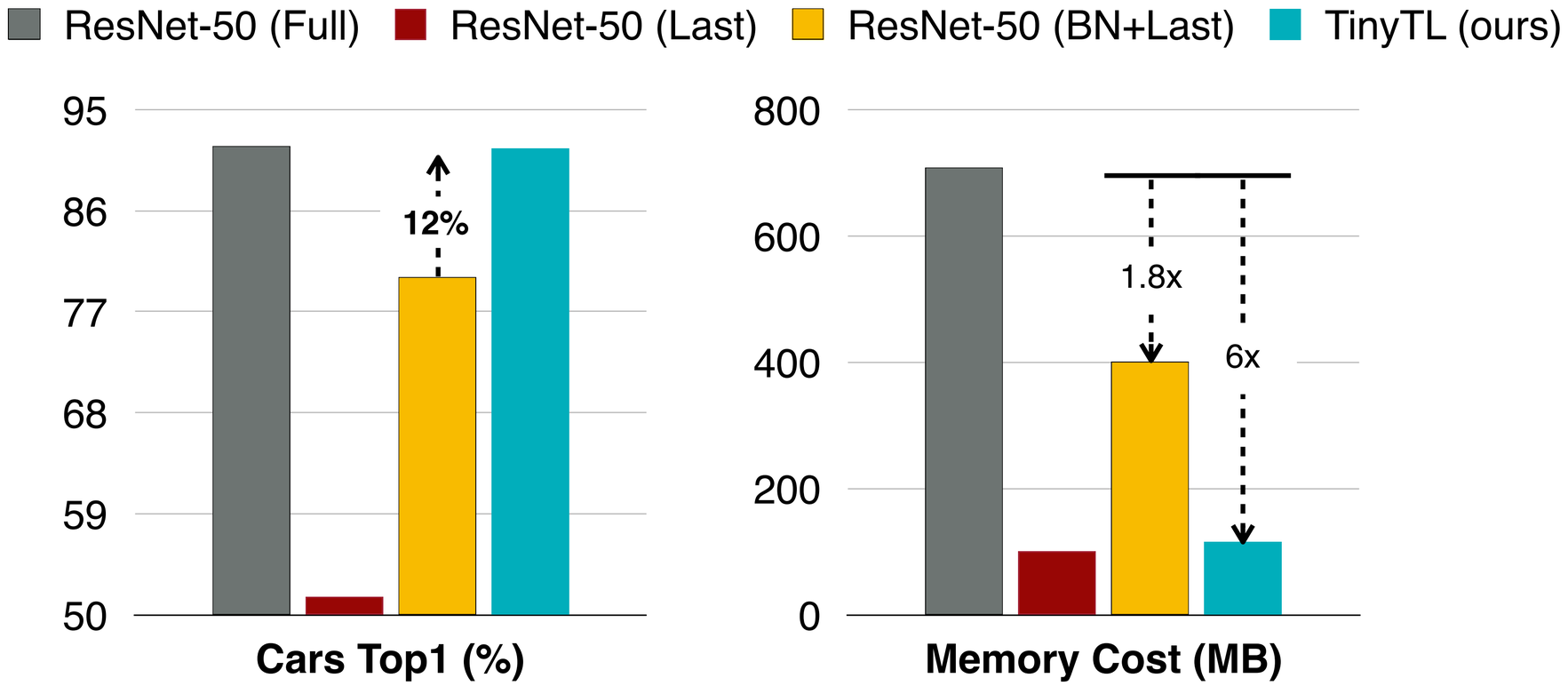
Parameter-Efficient Transfer Learning



Parameter-efficiency does not directly translate to memory-efficiency (12x vs 1.8x)

- Full: Fine-tune the full network. Better accuracy but highly inefficient.
- Last: Only fine-tune the last classifier head. Efficient but the capacity is limited.
- BN+Last: Fine-tune the BN layers and the last layer. Parameter-efficient, **but the memory saving is limited. Significant accuracy loss.**

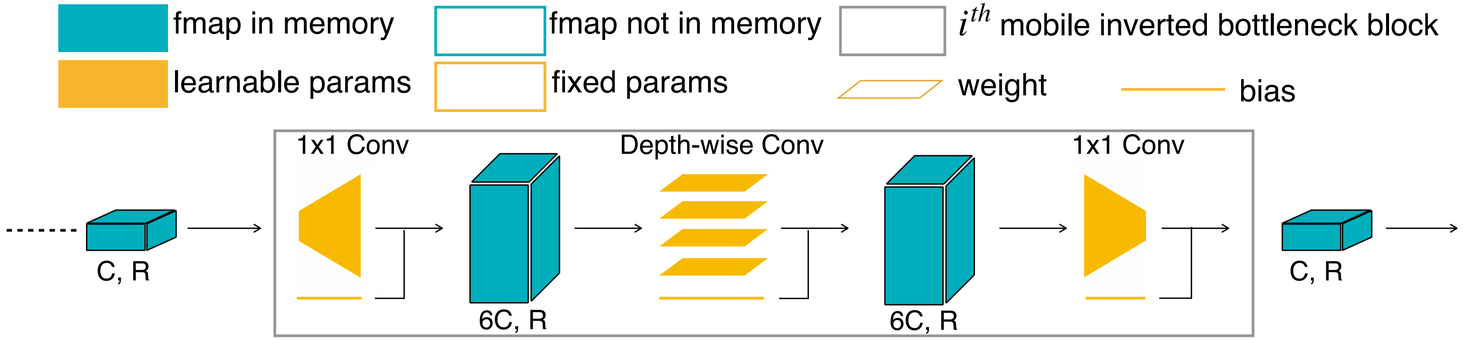
TinyTL: Memory-Efficient Transfer Learning



- Full: Fine-tune the full network. Better accuracy but highly inefficient.
- Last: Only fine-tune the last classifier head. Efficient but the capacity is limited.
- BN+Last: Fine-tune the BN layers and the last layer. Parameter-efficient, **but the memory saving is limited. Significant accuracy loss.**
- TinyTL: fine-tune bias only + lite residual learning: high accuracy, large memory saving

Updating Weights is Memory-Expensive

Updating the bias is memory-efficient



Fine-tune the full network (Conventional)

Forward: $\mathbf{a}_{i+1} = \mathbf{a}_i \mathbf{W}_i + \mathbf{b}_i$

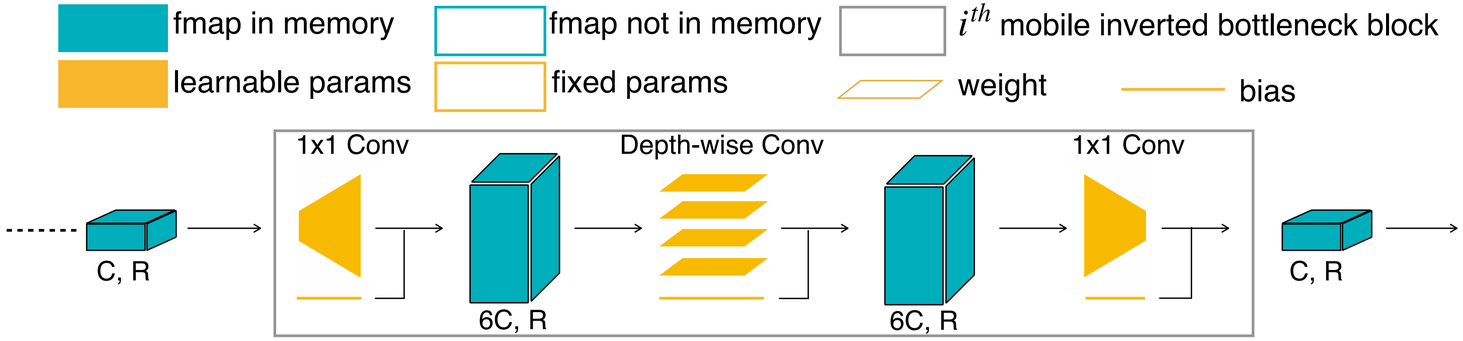
Backward: $\frac{\partial L}{\partial \mathbf{W}_i} = \mathbf{a}_i^T \frac{\partial L}{\partial \mathbf{a}_{i+1}}, \quad \frac{\partial L}{\partial \mathbf{b}_i} = \frac{\partial L}{\partial \mathbf{a}_{i+1}} = \frac{\partial L}{\partial \mathbf{a}_{i+2}} \mathbf{W}_{i+1}^T$

- Updating weights requires storing intermediate activations
- Updating biases does not

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai *et al.*, NeurIPS 2020]

Updating the Weight is Memory-Expensive

Updating the bias is memory-efficient

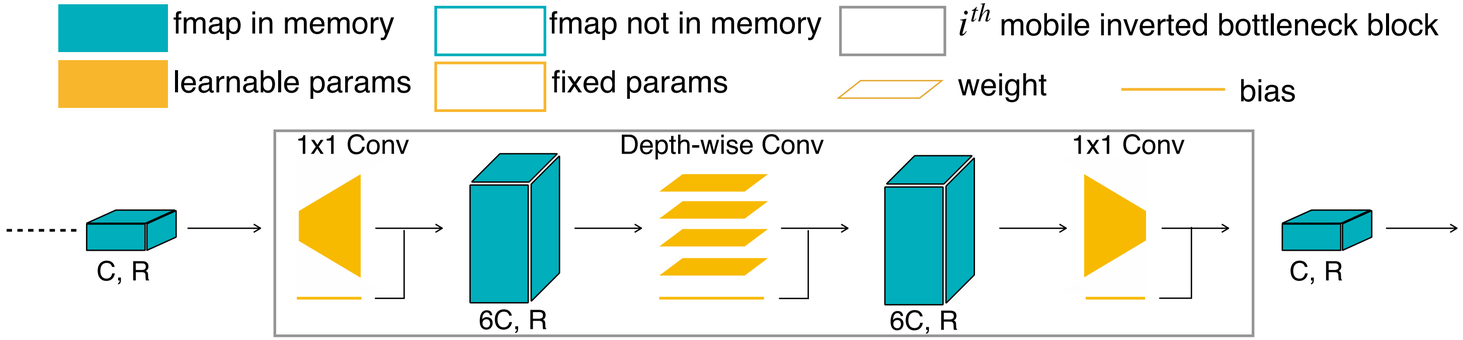


Fine-tune the full network (Conventional)

Forward: $\mathbf{a}_{i+1} = \mathbf{a}_i \mathbf{W}_i + \mathbf{b}_i$

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai *et al.*, NeurIPS 2020]

ReLU is Memory-Efficient

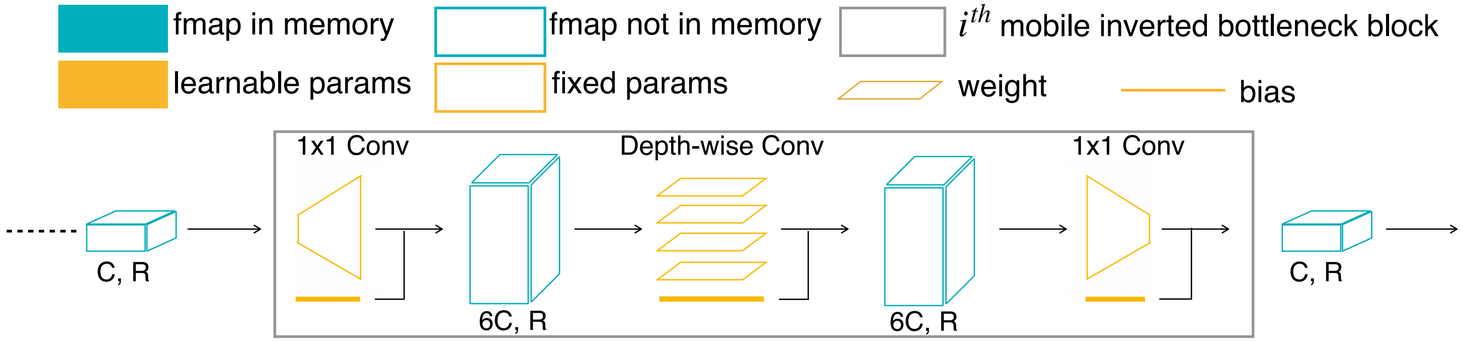


Fine-tune the full network (Conventional)

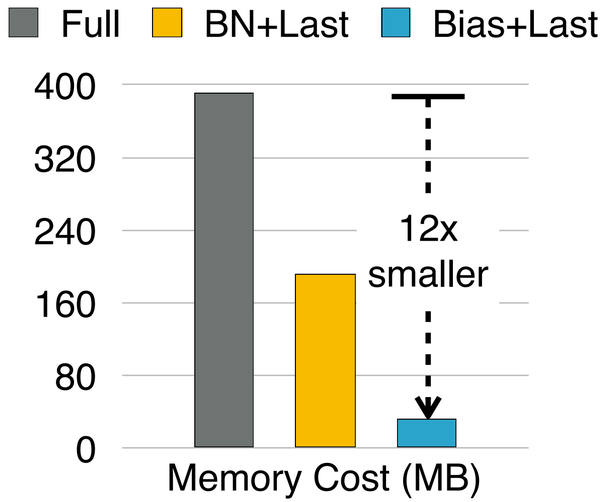
Layer Type	Forward	Backward	Memory Cost
ReLU	$\mathbf{a}_{i+1} = \max(0, \mathbf{a}_i)$	$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}} \circ \mathbf{1}_{\mathbf{a}_i \geq 0}$	$\lvert\mathbf{a}_i\rvert$ bits
sigmoid	$\mathbf{a}_{i+1} = \sigma(\mathbf{a}_i) = \frac{1}{1+\exp(-\mathbf{a}_i)}$	$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}} \circ \sigma(\mathbf{a}_i) \circ (1 - \sigma(\mathbf{a}_i))$	32 $\lvert\mathbf{a}_i\rvert$ bits
h-swish [8]	$\mathbf{a}_{i+1} = \mathbf{a}_i \circ \frac{\text{ReLU6}(\mathbf{a}_i+3)}{6}$	$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}} \circ \left(\frac{\text{ReLU6}(\mathbf{a}_i+3)}{6} + \mathbf{a}_i \circ \frac{\mathbf{1}_{-3 \leq \mathbf{a}_i \leq 3}}{6}\right)$	32 $\lvert\mathbf{a}_i\rvert$ bits

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai *et al.*, NeurIPS 2020]

TinyTL: Fine-tune Bias Only



Fine-tune bias only

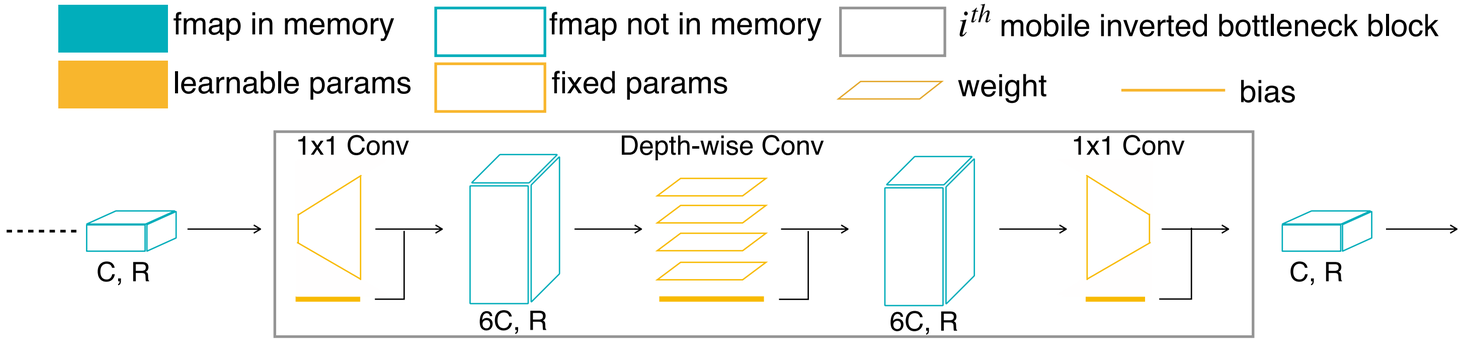


Freeze weights, only fine-tune biases

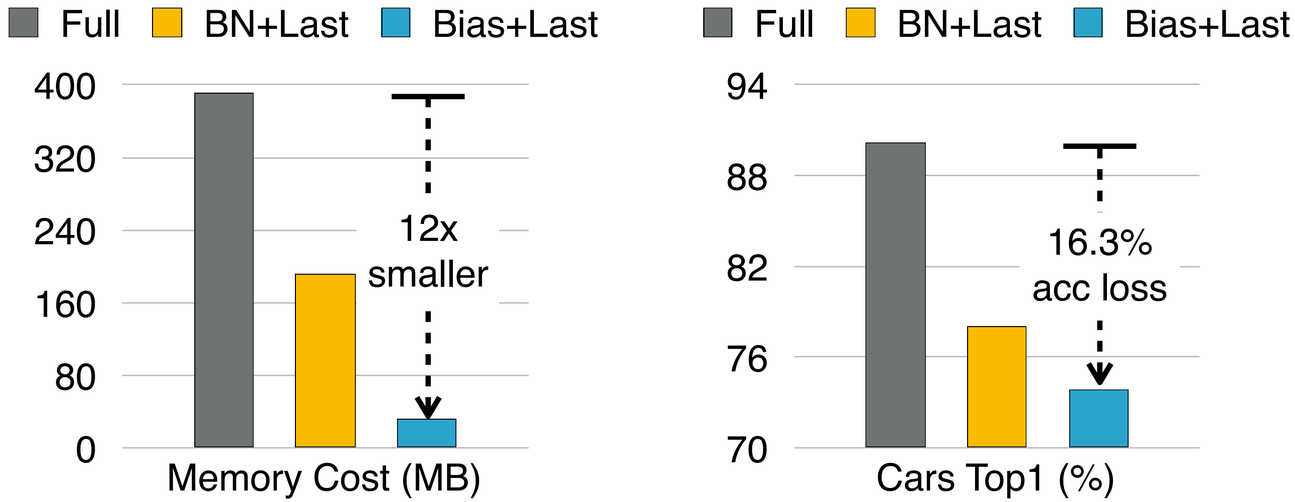
=> save 12x memory

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai *et al.*, NeurIPS 2020]

TinyTL: Fine-tune Bias Only



Fine-tune bias only

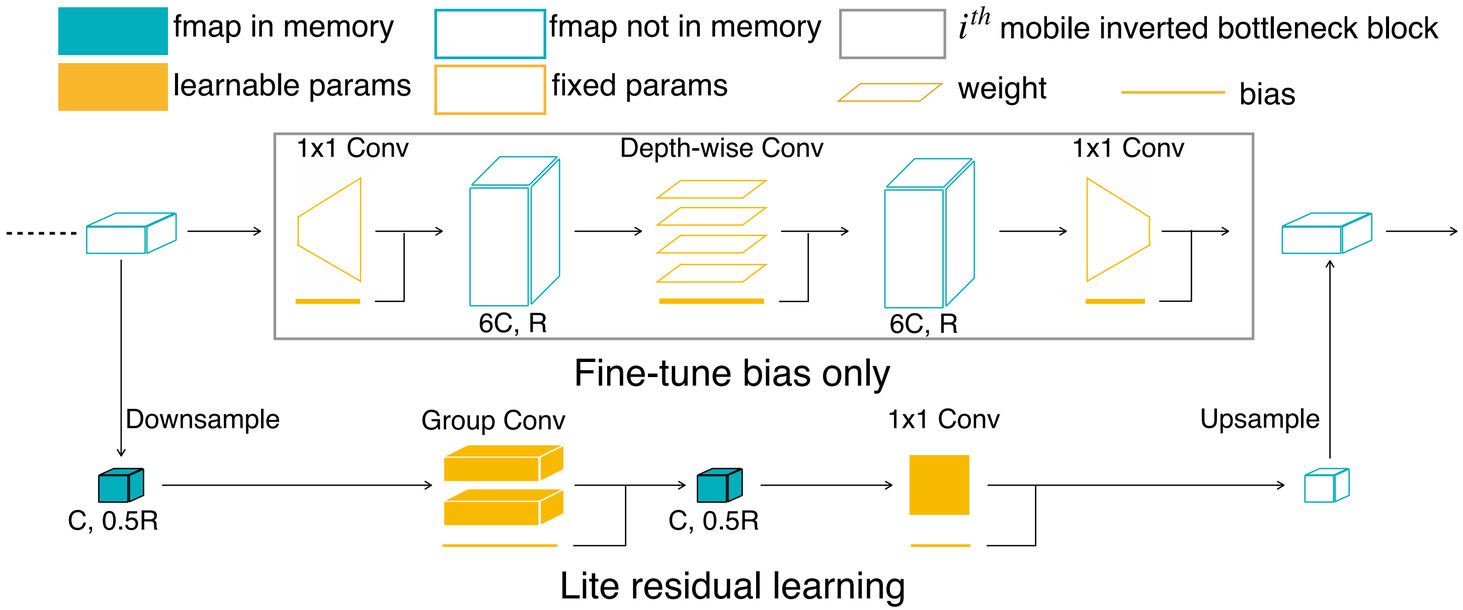


Freeze weights, only fine-tune biases

=> save 12x memory, but also hurt the accuracy

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai *et al.*, NeurIPS 2020]

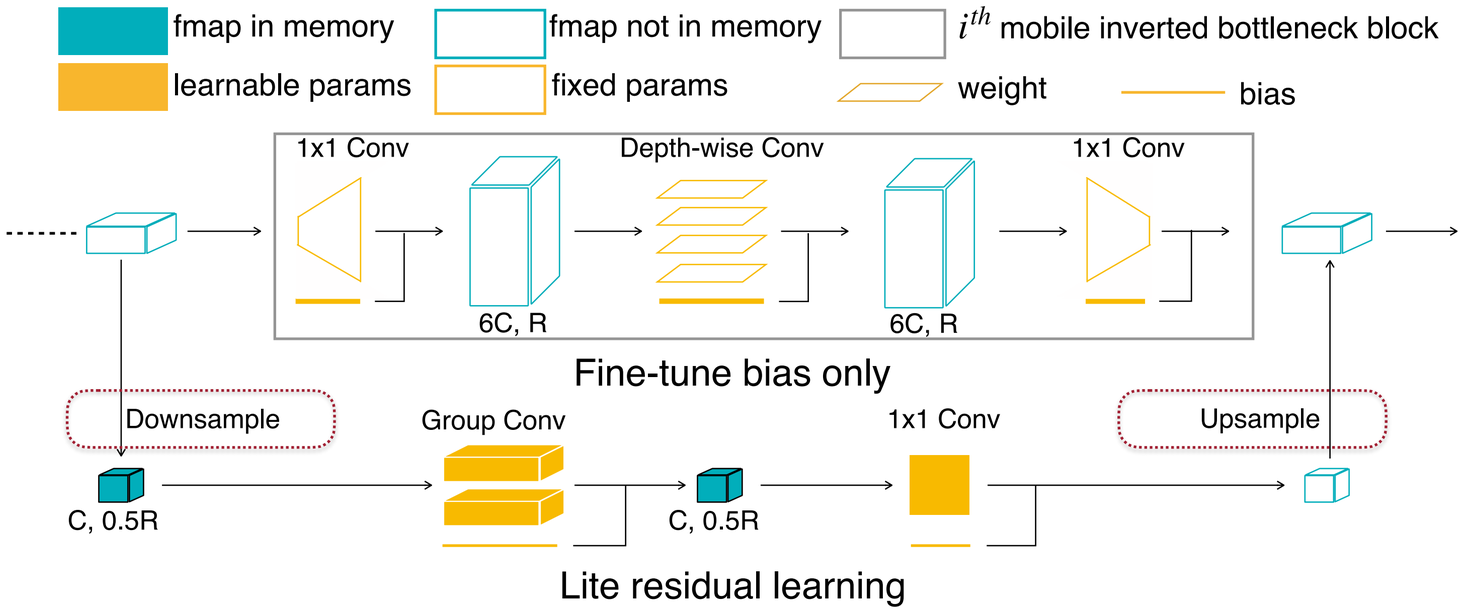
TinyTL: Lite Residual Learning



- Add lite residual modules to increase model capacity
- Key principle - keep activation size small

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai *et al.*, NeurIPS 2020]

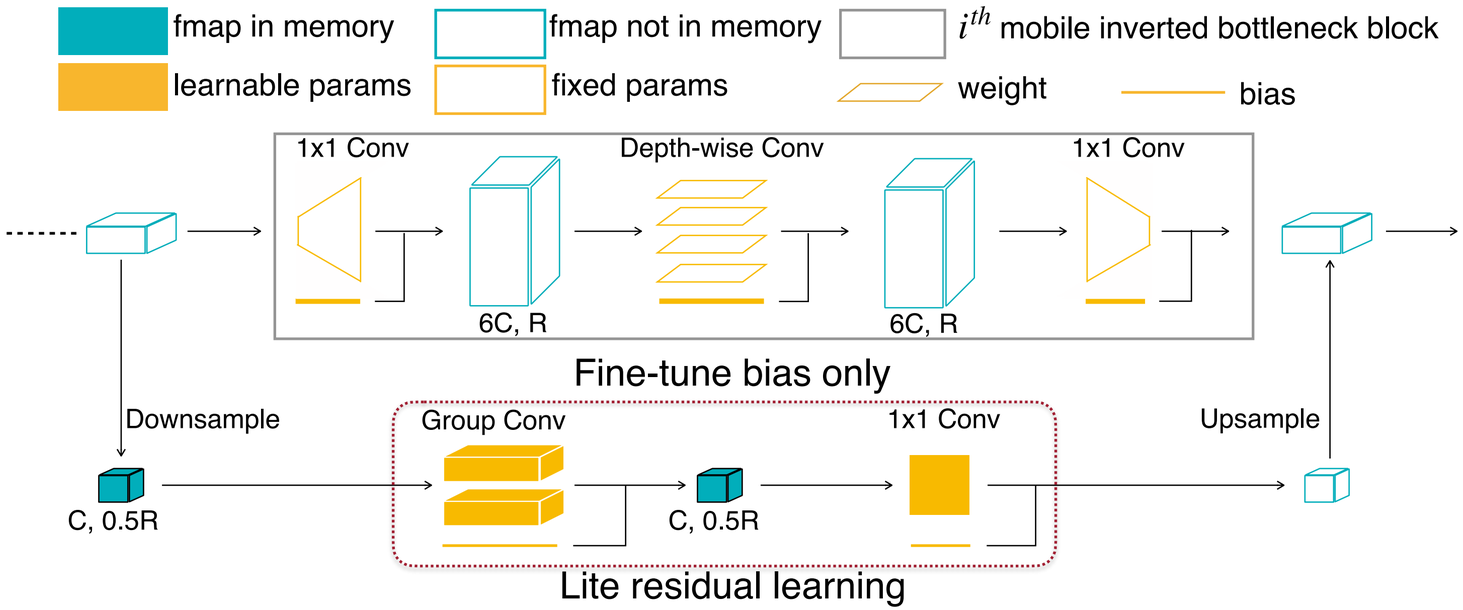
TinyTL: Lite Residual Learning



- Add lite residual modules to increase model capacity
- Key principle - keep activation size small
  1. Reduce the resolution

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai *et al.*, NeurIPS 2020]

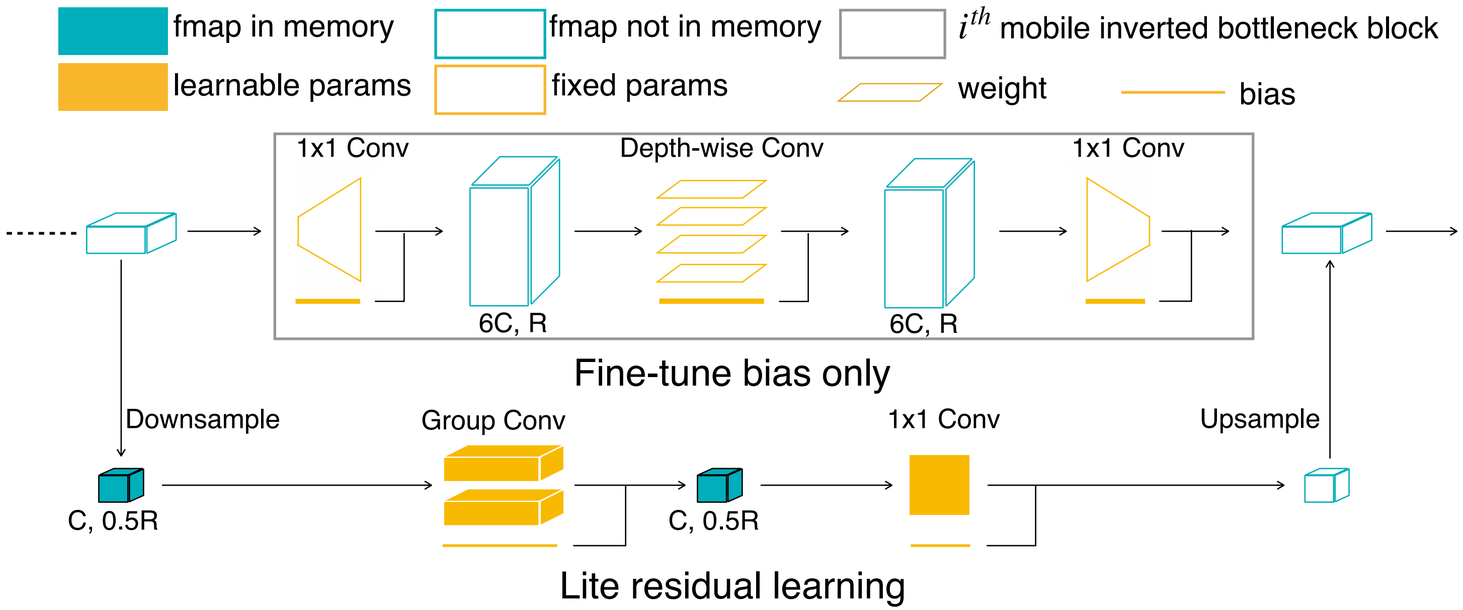
TinyTL: Lite Residual Learning



- Add lite residual modules to increase model capacity
- Key principle - keep activation size small
  1. Reduce the resolution
  2. Avoid inverted bottleneck

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai *et al.*, NeurIPS 2020]

TinyTL: Lite Residual Learning

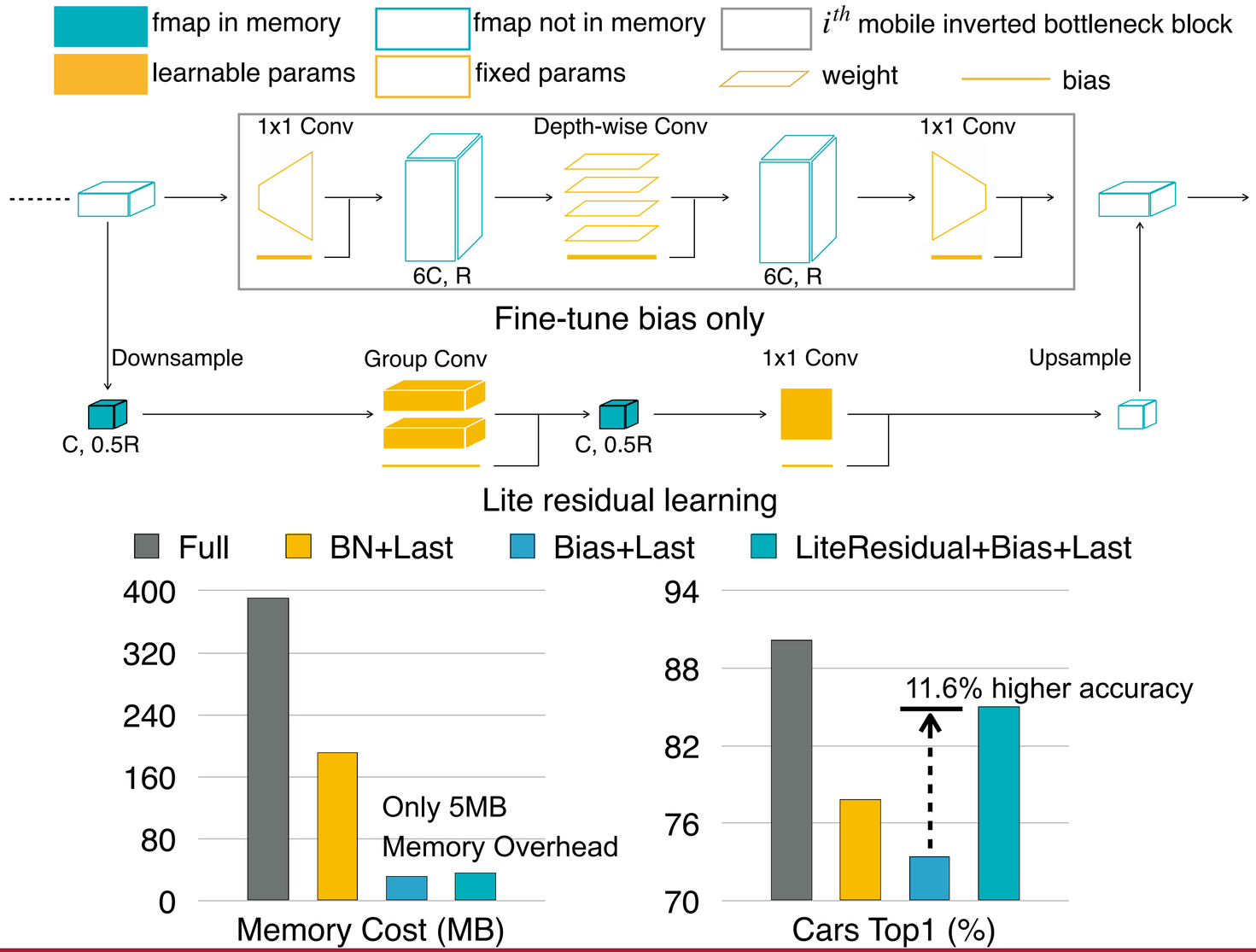


- Add lite residual modules to increase model capacity
- Key principle - keep activation size small
  1. Reduce the resolution
  2. Avoid inverted bottleneck

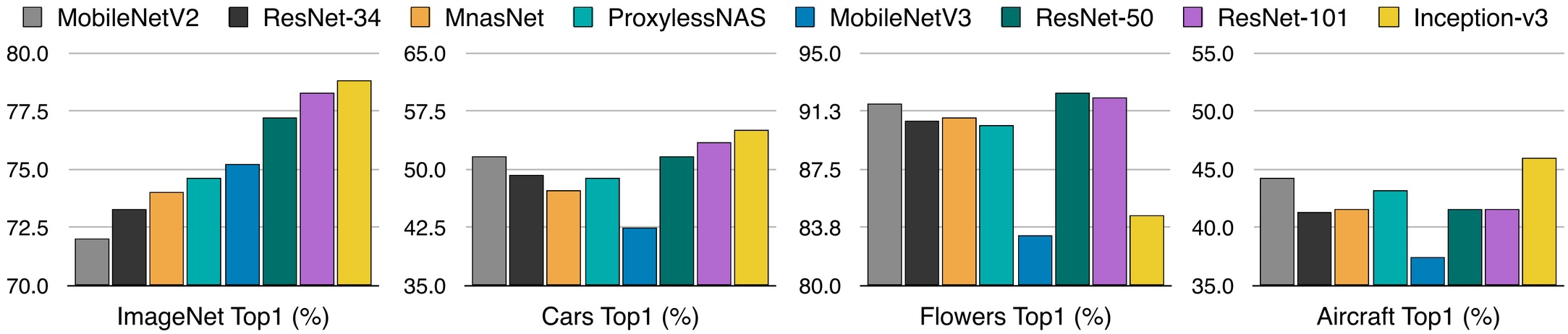
(1/6 channel, 1/2 resolution, 2/3 depth => ~4% activation size)

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai *et al.*, NeurIPS 2020]

TinyTL: Lite Residual Learning

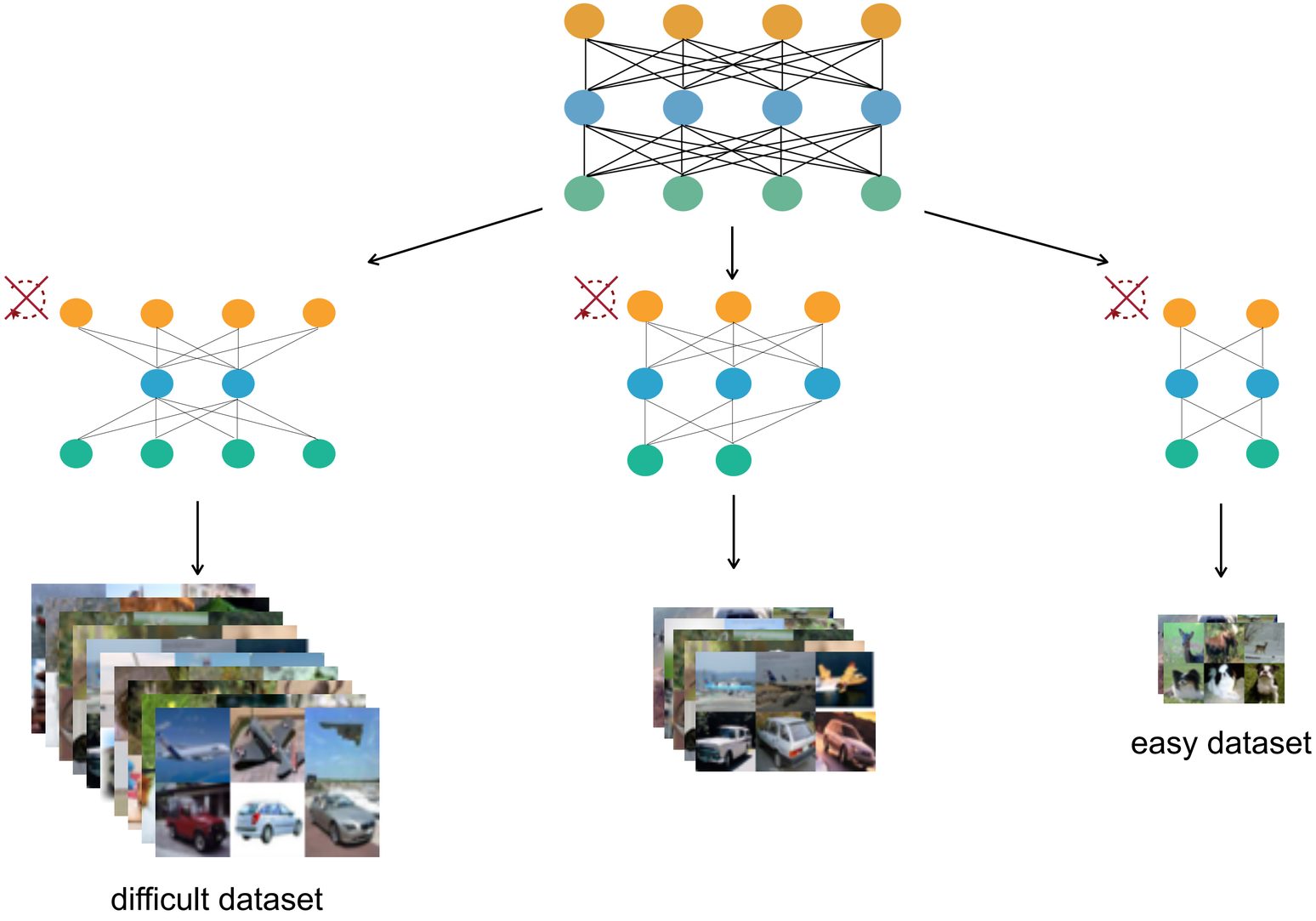


TinyTL: Specialized Models for Different Tasks

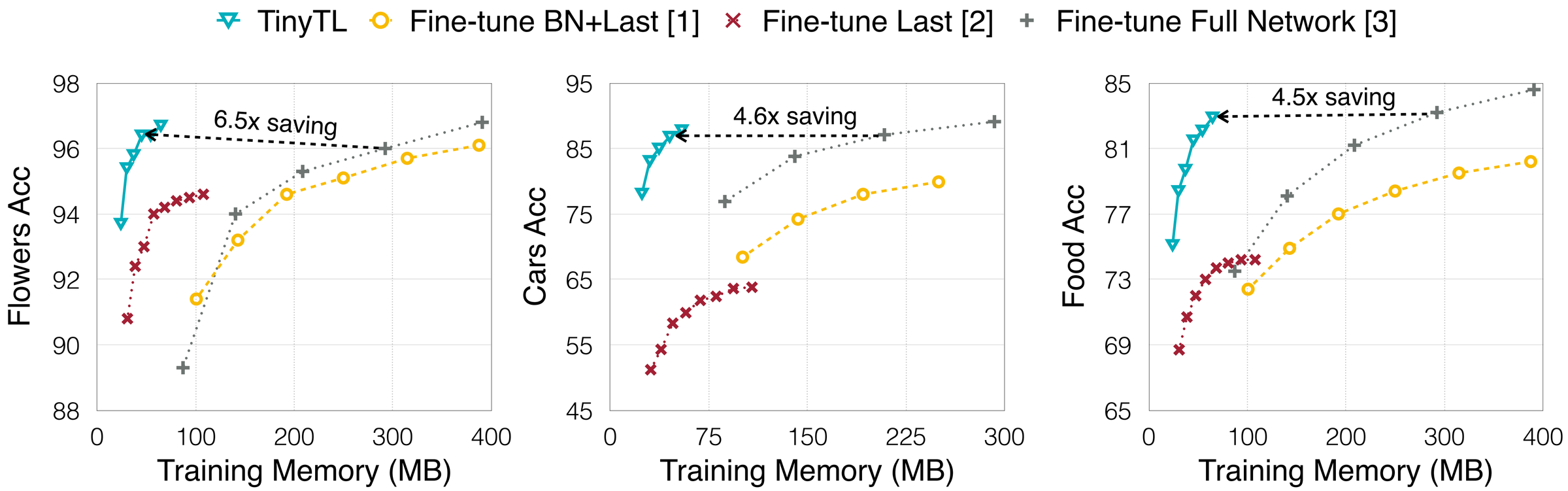


The relative accuracy order between different pre-trained models changes **significantly** among ImageNet and the transfer learning datasets, which motivates **personalized and specialized NN architecture for different downstream tasks**.

TinyTL + Once-for-All Network



TinyTL: Up to 6.5x Memory Saving

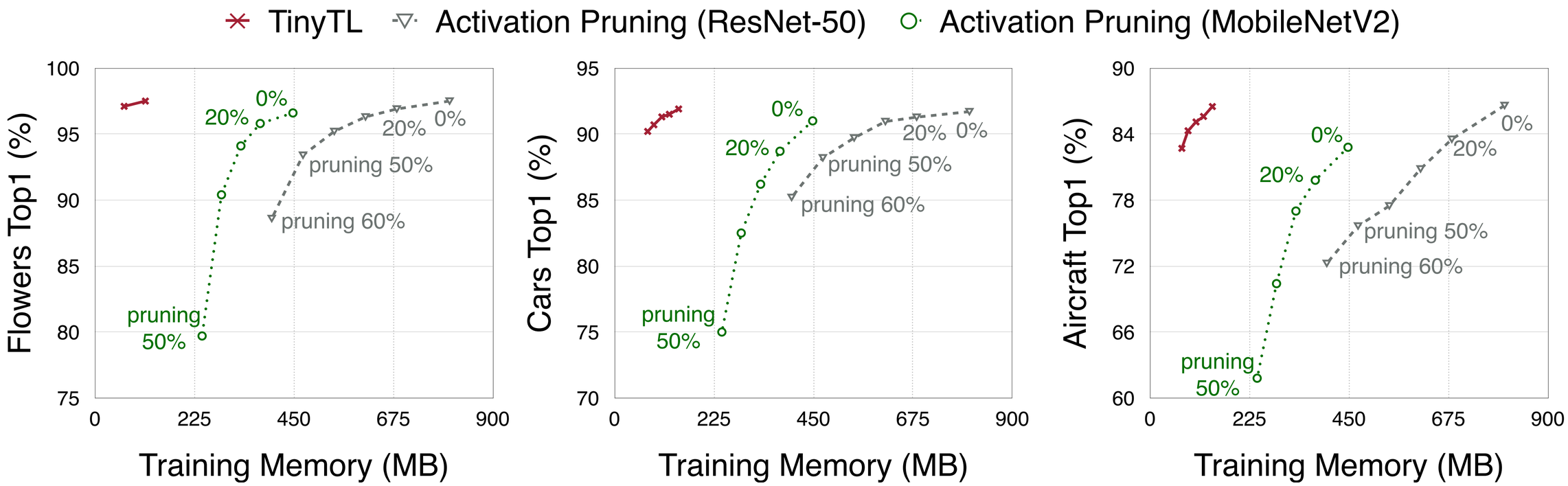


Backbone: ProxylessNAS-Mobile, Scanning over different resolutions

- TinyTL provides up to **6.5x** memory saving **without accuracy loss**.

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai *et al.*, NeurIPS 2020]

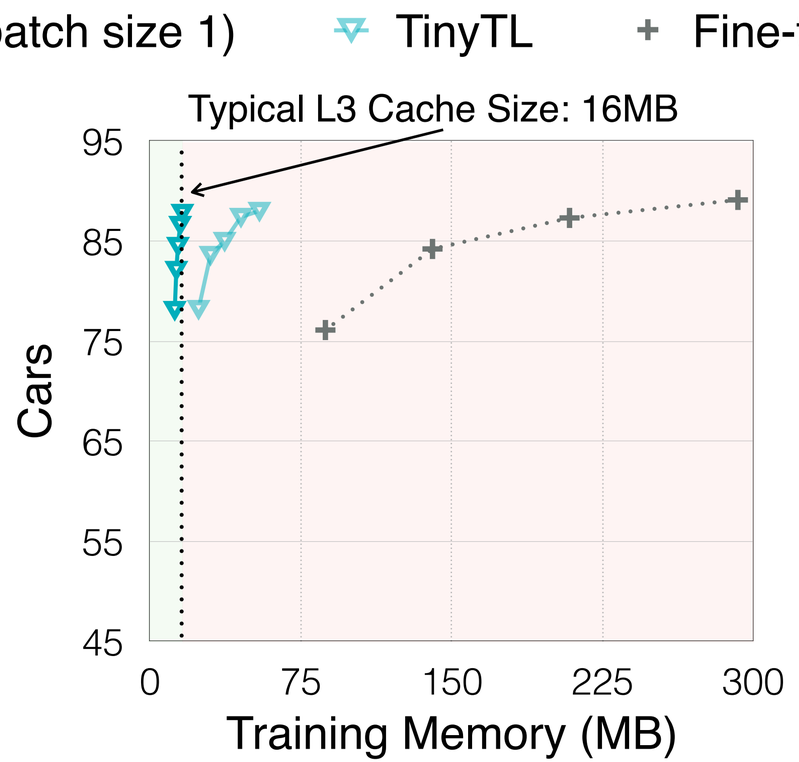
Comparison with Dynamic Activation Pruning



- Compared with dynamic activation pruning, TinyTL saves the memory more effectively.

TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai *et al.*, NeurIPS 2020]

TinyTL enables in-memory training



- TinyTL (tiny transfer learning) supports batch 1 training by **group normalization**.
- Together with the lite residual model, it further reduces the training memory cost to 16MB (fits L3 cache), enabling fitting the training process into cache, which is much more energy-efficient than training on DRAM.

On-Device Learning Design Principles

- Parameter-efficiency does not translate to memory-efficiency.
- The main memory bottleneck of the training is the activations, not #trainable parameters.
- TinyTL saves the activation memory by fine-tuning bias only, and lite residual learning.

How can we translate the algorithmic improvement

into actual savings?

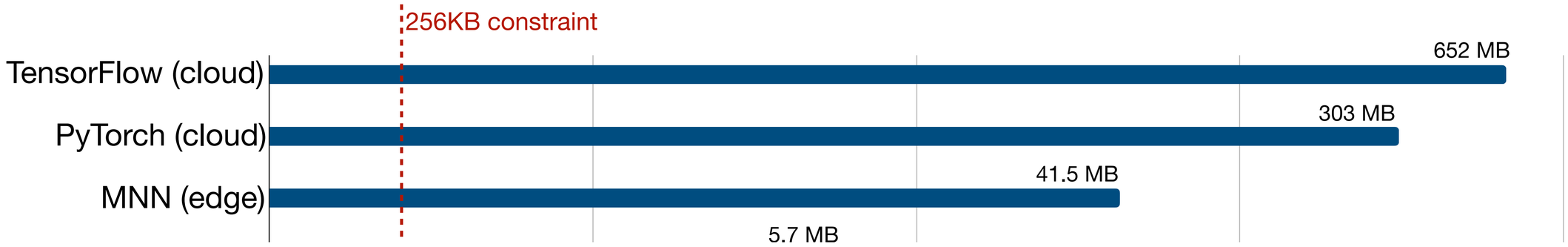
Algorithm System Co-Design.

Section 2: Co-Designs for On-Device Training

The difficulty of optimizing quantized model and system-level support.

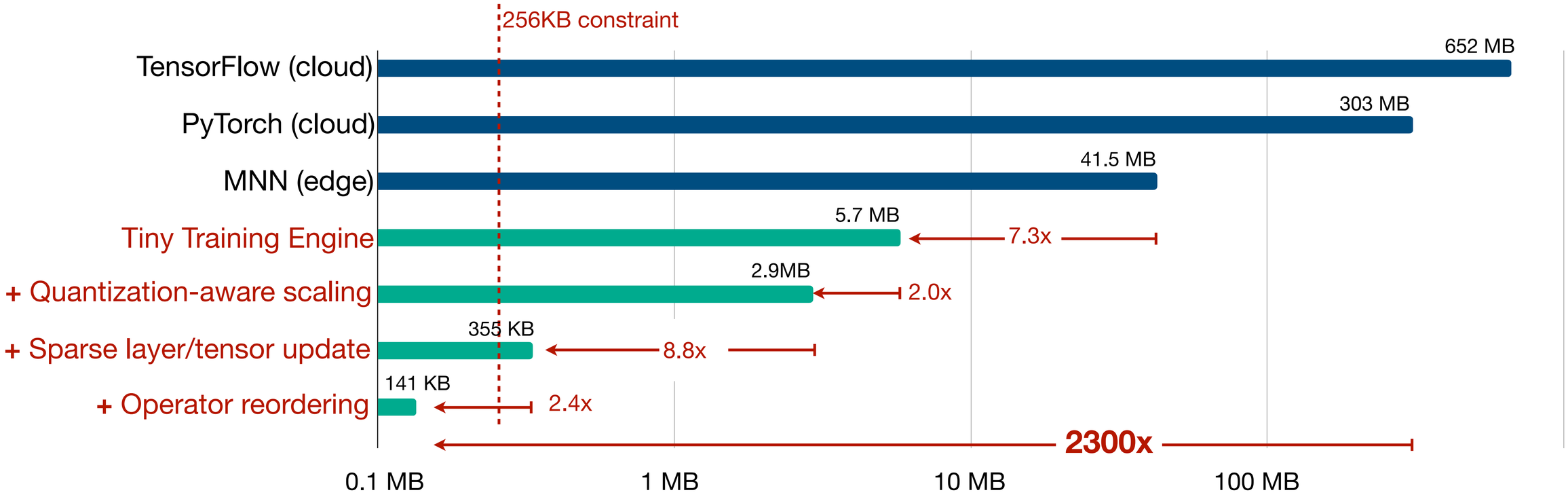
On-Device Training Under 256KB Memory

- **Training** is more expensive than **inference** due to back-propagation, making it hard to fit IoT devices. Can we go even smaller? e.g., on-device training on MCU that has only 256KB SRAM

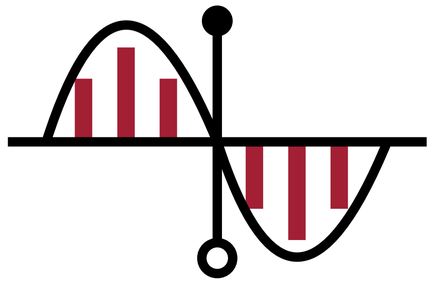


On-Device Training Under 256KB Memory

- **Training** is more expensive than **inference** due to back-propagation, making it hard to fit IoT devices. Can we go even smaller? e.g., on-device training on MCU that has only 256KB SRAM

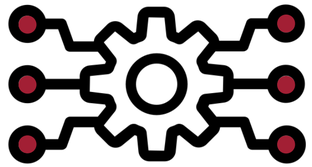


On-Device Training Under 256KB Memory



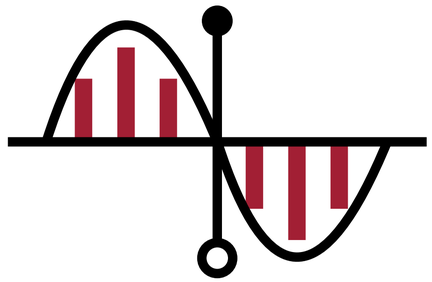
1. Quantization-aware scaling

2. Sparse layer/tensor update



3. Tiny Training Engine

On-Device Training Under 256KB Memory



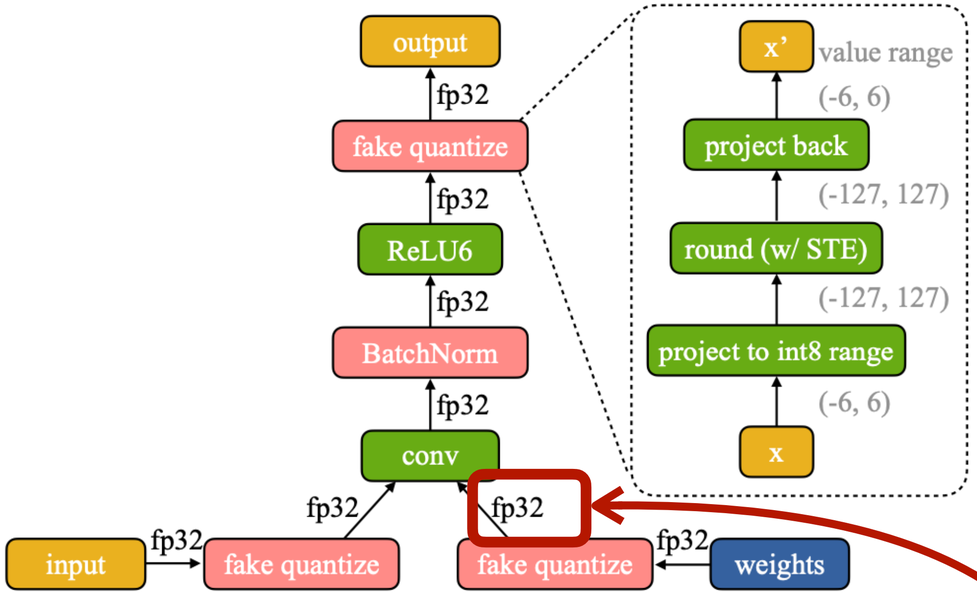
1. Quantization-aware scaling

2. Sparse layer/tensor update

3. Tiny Training Engine

Difficulty of Quantized Graphs

Fake quantized graphs for QAT, but does not save memory

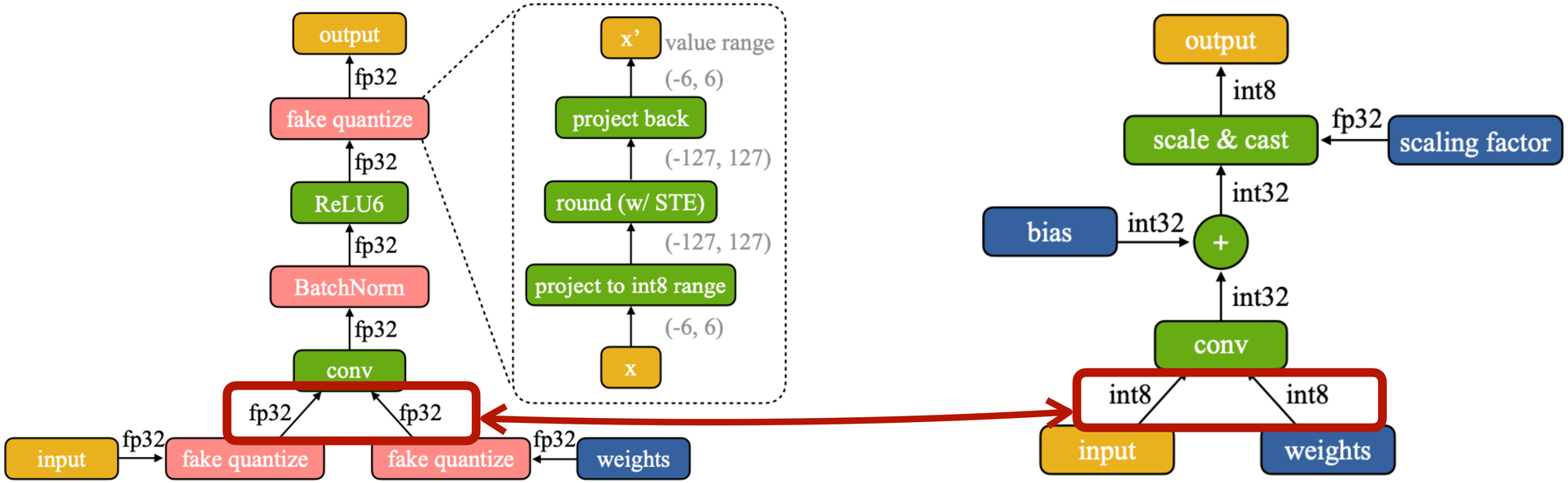


(a) Fake Quantization
(quantization aware training)

Most intermediate tensors are still in FP32 format in fake quantization, thus cannot save memory footprint

Difficulty of Quantized Graphs

Real quantized graphs save memory, but hard to quantize



(a) Fake Quantization
(quantization aware training)

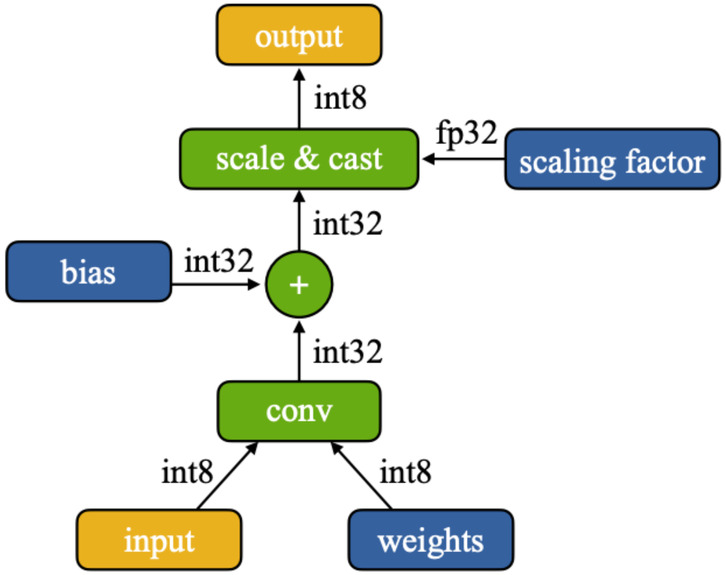
(b) Real Quantization
(inference/on-device training)

All tensors are in int8/int32 format for real quantization,
thus save memory footprint, but leading to optimization difficulty

	Fake	Real
Weight	FP32	INT8
Activation	FP32	INT8
Batch Norm	Yes	No

Difficulty of Quantized Graphs

But are hard to quantize

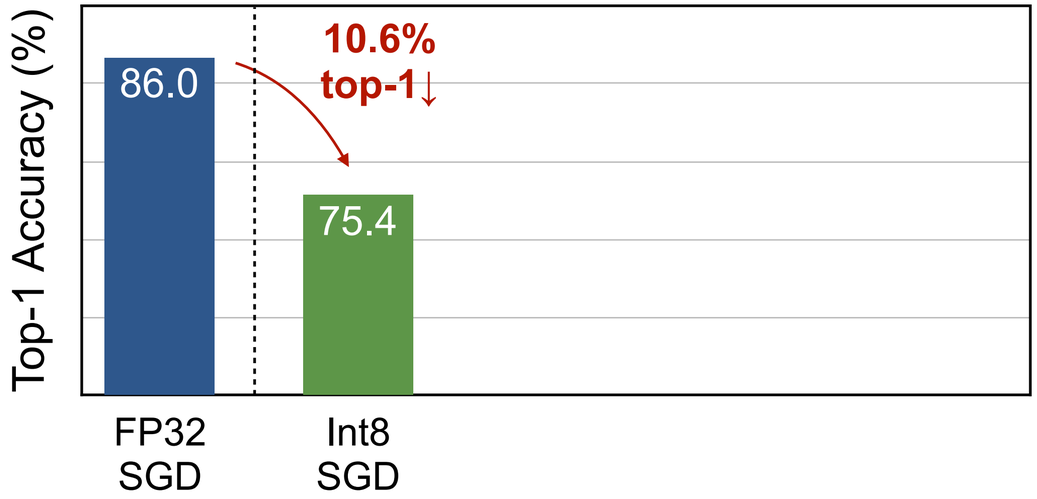


(a) Real Quantization

Making training difficult:

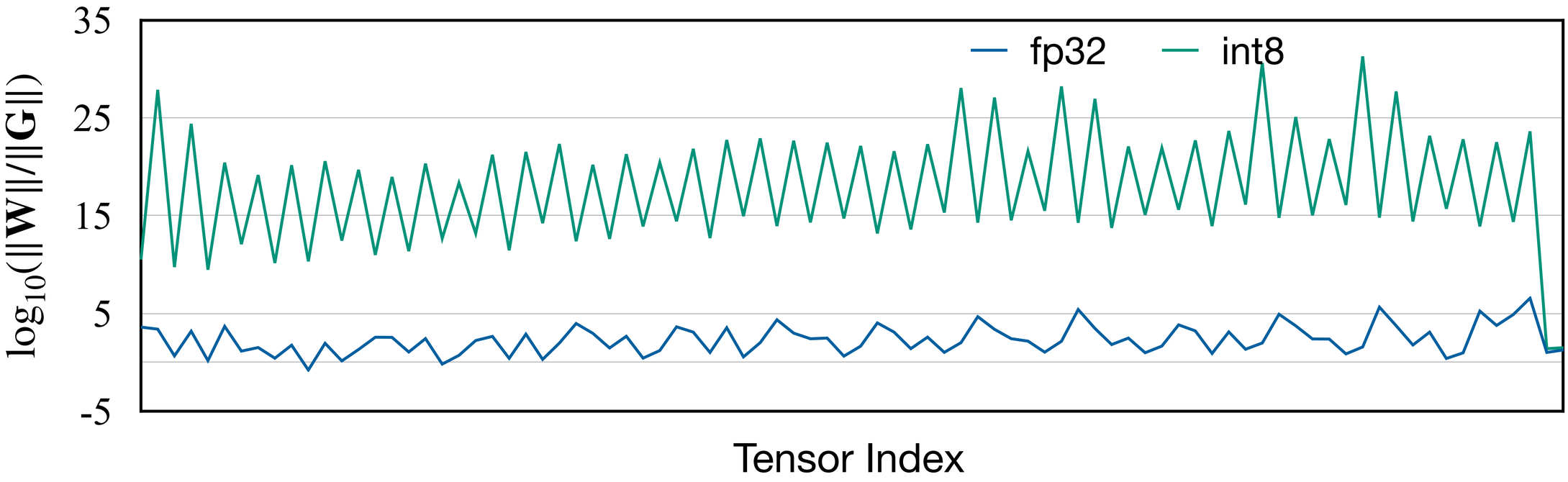
- Mixed precisions: int8/int32/fp32…
- Lack BatchNorm

Performance Comparison (average on 10 datasets)



Difficulty of Quantized Graphs

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



QAS: Quantization-Aware Scaling

QAS addresses the optimization difficulty of quantized graphs

Quantization overview

$$\bar{\mathbf{y}}_{\texttt{int8}} = \texttt{cast2int8}[s_{\texttt{fp32}} \cdot (\bar{\mathbf{W}}_{\texttt{int8}} \bar{\mathbf{x}}_{\texttt{int8}} + \bar{\mathbf{b}}_{\texttt{int32}})],$$

Per Channel scaling

$$\mathbf{W} = s_{\mathbf{W}} \cdot (\mathbf{W}/s_{\mathbf{W}}) \overset{\text{quantize}}{\approx} s_{\mathbf{W}} \cdot \bar{\mathbf{W}}, \quad \mathbf{G}_{\bar{\mathbf{W}}} \approx s_{\mathbf{W}} \cdot \mathbf{G}_{\mathbf{W}},$$

Question: is $S_{\mathbf{W}}$ >1 or <1?

Weight and gradient ratios are off by $S_{\mathbf{W}}^{-2}$

$$\|\bar{\mathbf{W}}\|/\|\mathbf{G}_{\bar{\mathbf{W}}}\| \approx \|\mathbf{W}/s_{\mathbf{W}}\|/\|s_{\mathbf{W}} \cdot \mathbf{G}_{\mathbf{W}}\| = \boxed{s_{\mathbf{W}}^{-2}} \cdot \|\mathbf{W}\|/\|\mathbf{G}\|.$$

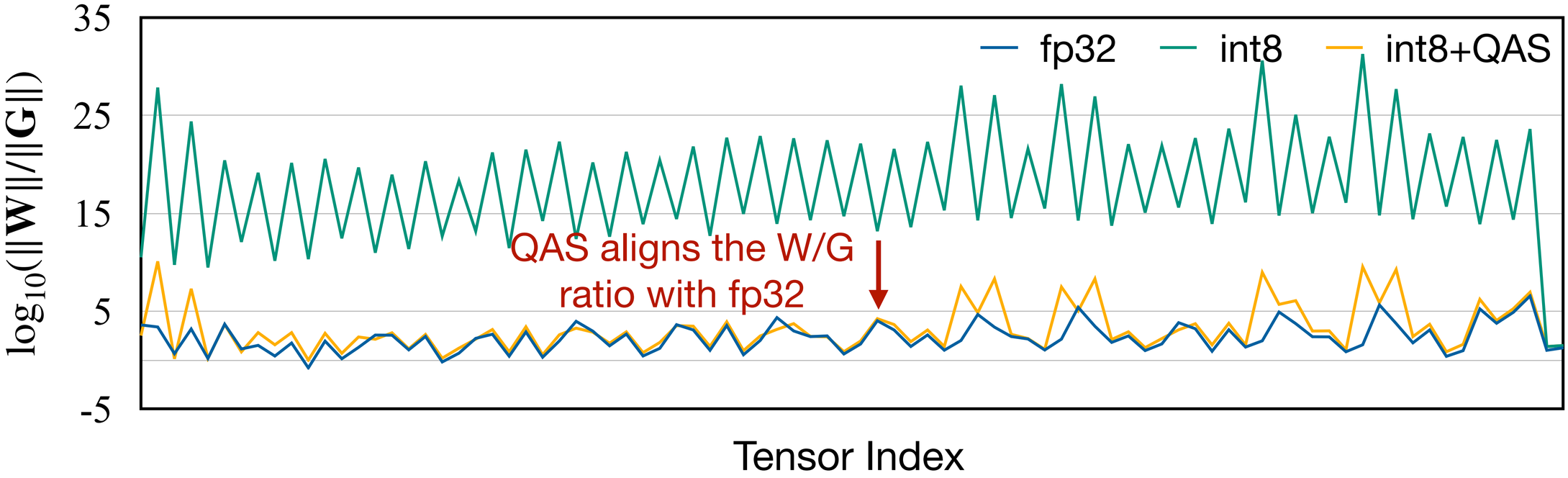
Thus, re-scale the gradients

$$\tilde{\mathbf{G}}_{\bar{\mathbf{W}}} = \mathbf{G}_{\bar{\mathbf{W}}} \cdot s_{\mathbf{W}}^{-2}, \quad \tilde{\mathbf{G}}_{\bar{\mathbf{b}}} = \mathbf{G}_{\bar{\mathbf{b}}} \cdot s_{\mathbf{W}}^{-2} \cdot s_{\mathbf{x}}^{-2} = \mathbf{G}_{\bar{\mathbf{b}}} \cdot s^{-2}$$

QAS: Quantization-Aware Scaling

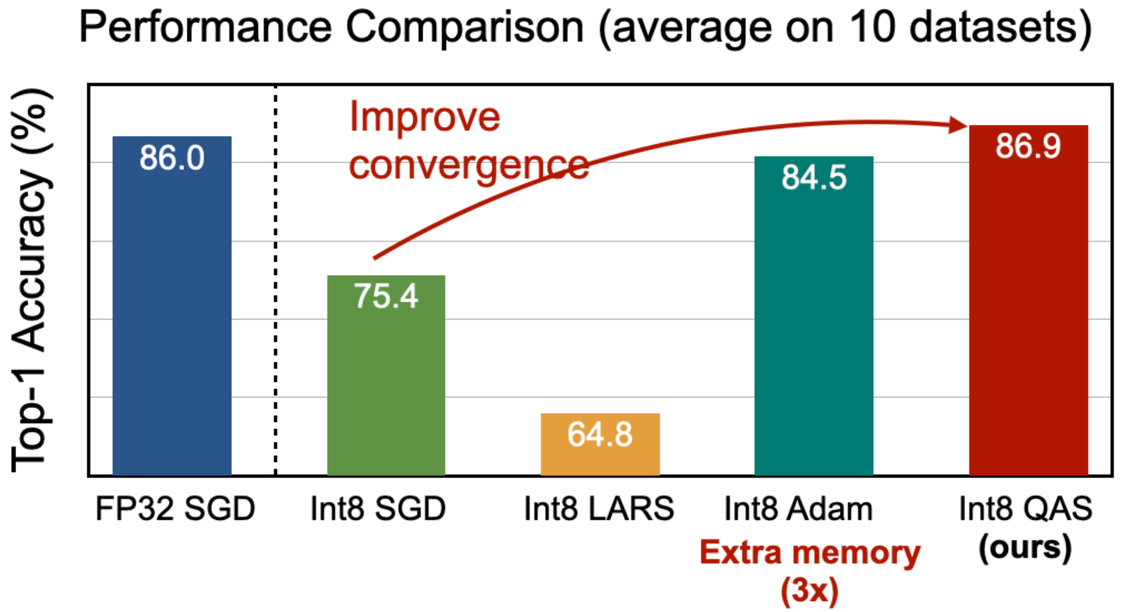
QAS addresses the optimization difficulty of quantized graphs

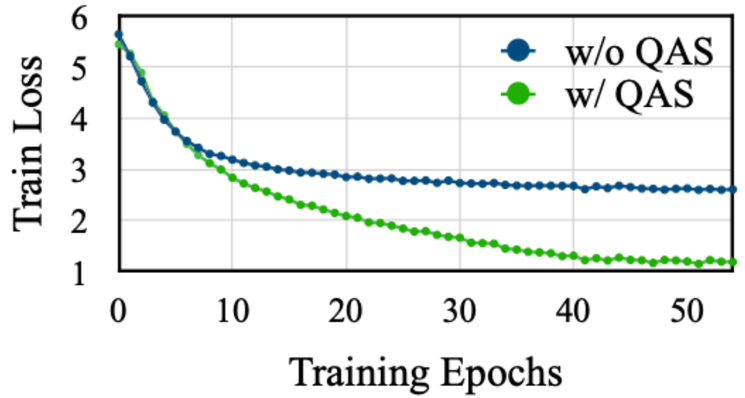
$$\tilde{\mathbf{G}}_{\bar{\mathbf{W}}} = \mathbf{G}_{\bar{\mathbf{W}}} \cdot s_{\mathbf{W}}^{-2}, \quad \tilde{\mathbf{G}}_{\bar{\mathbf{b}}} = \mathbf{G}_{\bar{\mathbf{b}}} \cdot s_{\mathbf{W}}^{-2} \cdot s_{\mathbf{x}}^{-2} = \mathbf{G}_{\bar{\mathbf{b}}} \cdot s^{-2}$$

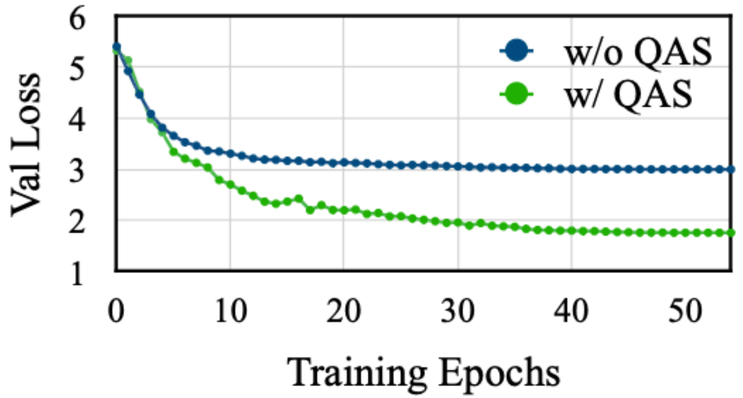


QAS: Quantization-Aware Scaling

QAS addresses the optimization difficulty of quantized graphs







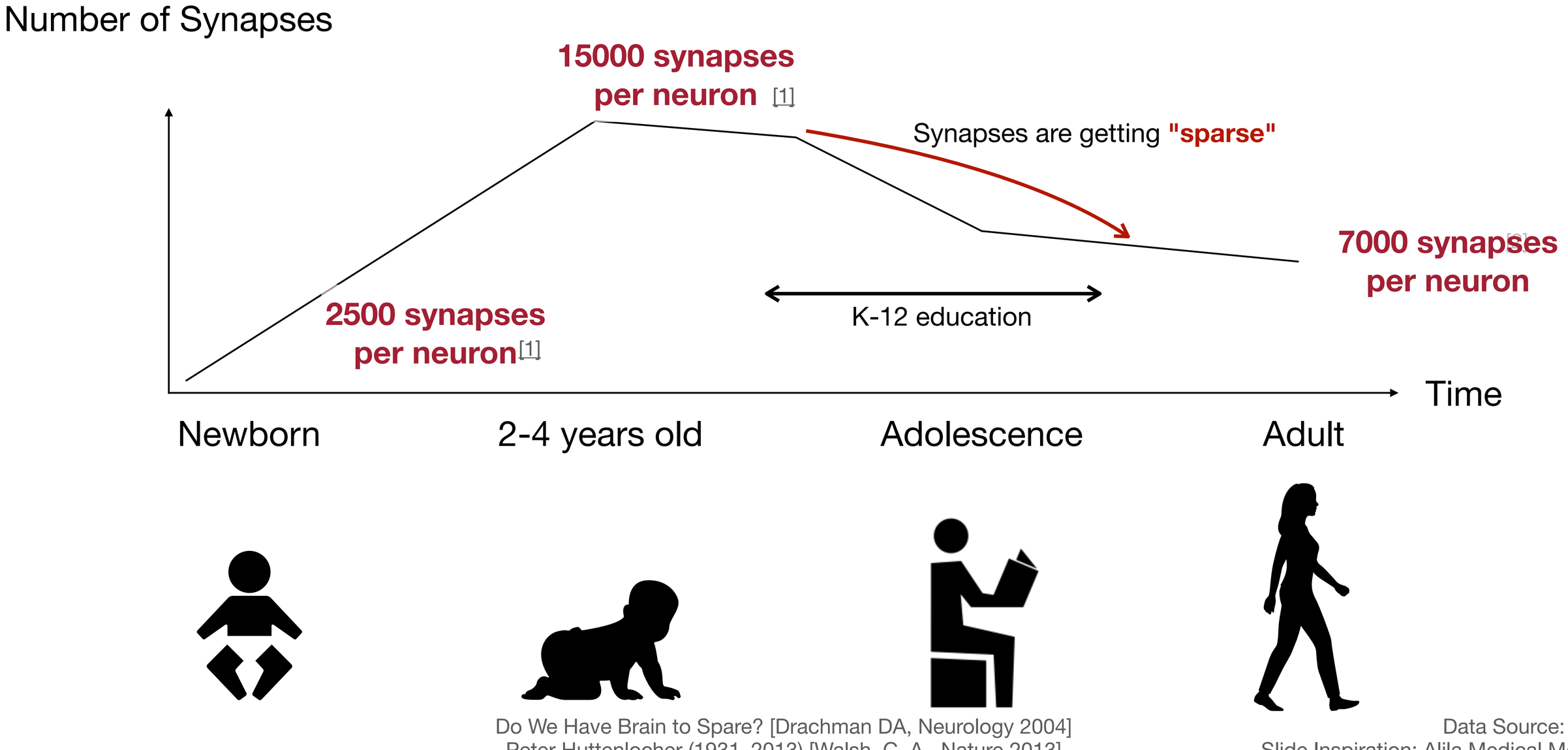
On-Device Training Under 256KB Memory

1. Quantization-aware scaling

2. Sparse layer/tensor update

3. Tiny Training Engine

Sparse Learning

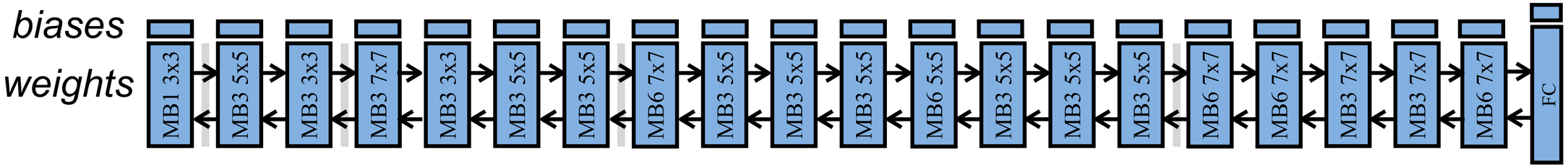


Do We Have Brain to Spare? [Drachman DA, Neurology 2004]
Peter Huttenlocher (1931–2013) [Walsh, C. A., Nature 2013]

Data Source: 1, 2
Slide Inspiration: Alila Medical Media

Sparse Layer/Tensor Update

Full update



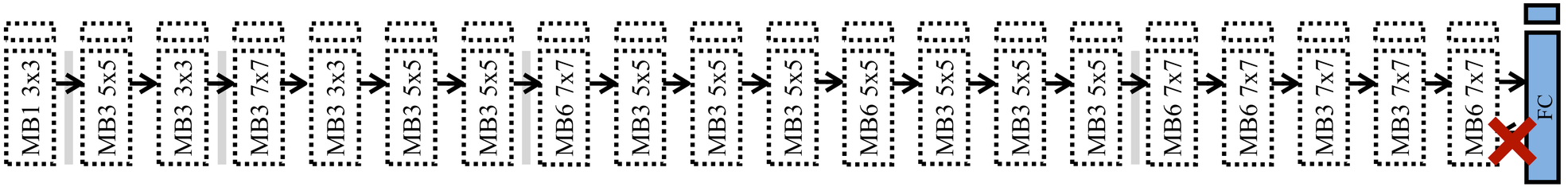
Model: ProxylessNAS-Mobile

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)

Sparse Layer/Tensor Update

Last layer update



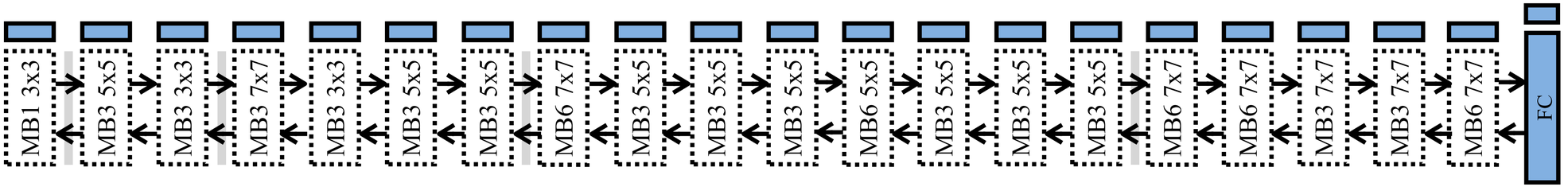
Model: ProxylessNAS-Mobile

Updating only the last cheap

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

Sparse Layer/Tensor Update

Bias-only update



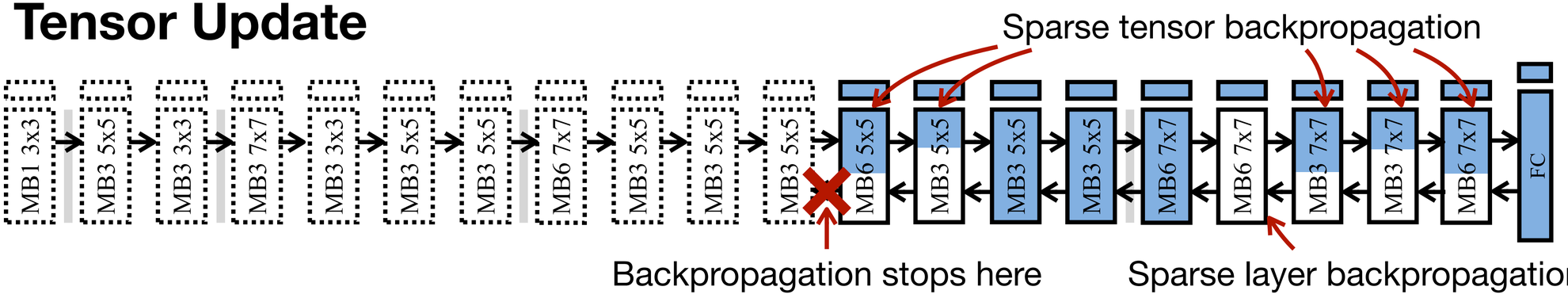
Model: ProxylessNAS-Mobile

Updating the only the bias part

- No need to store the activations
- Back propagating to the first layer.

Sparse Layer/Tensor Update

Sparse Tensor Update



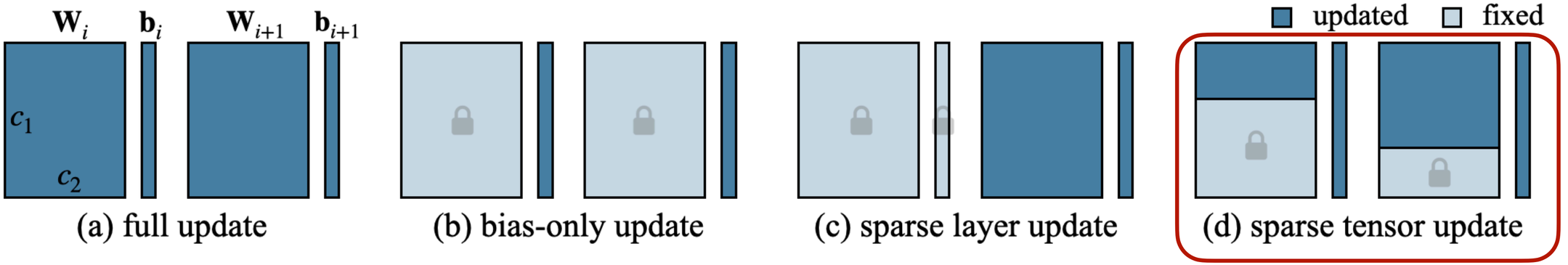
Model: ProxylessNAS-Mobile

Updating the sparse tensors / layers

- No need to back propagate the first layers
- Only need to store a subset of the activations.

Sparse Layer/Tensor Update

More efficient variants



Activation to store: (N, M)

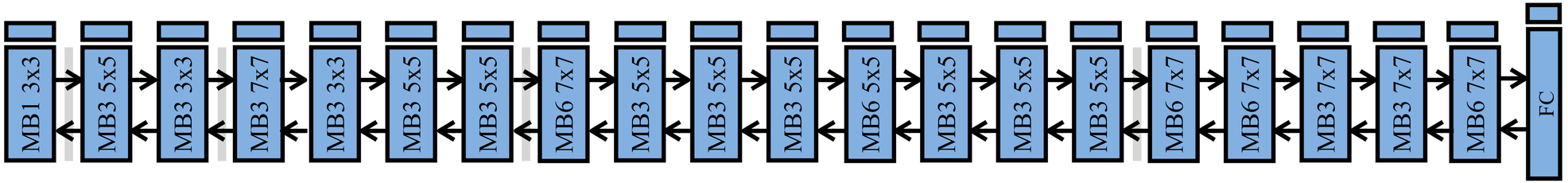
Weight in SRAM: (M, H)

Reduce by 4x

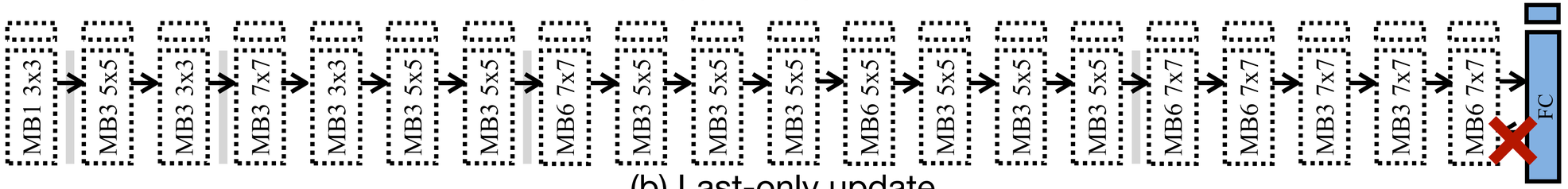
Activation to store: (N, 0.25*M)

Weight in SRAM: (0.25*M, H)

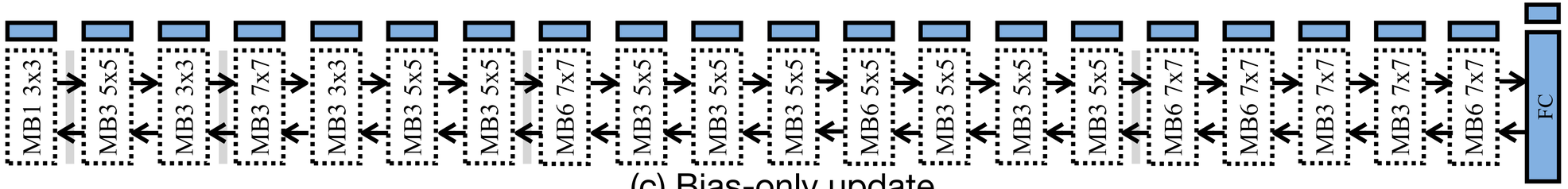
Update Paradigms Comparison



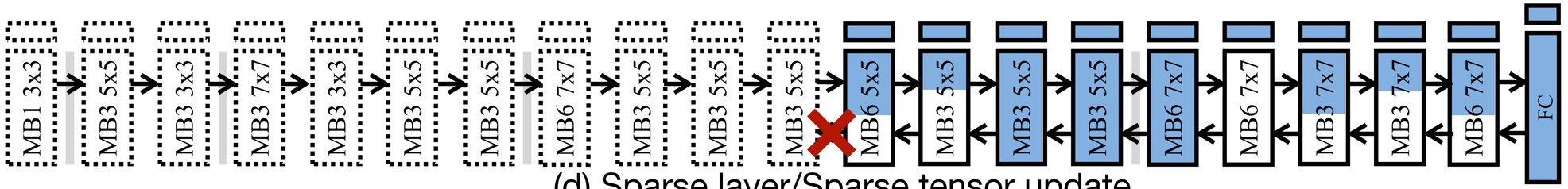
(a) Full update



(b) Last-only update



(c) Bias-only update

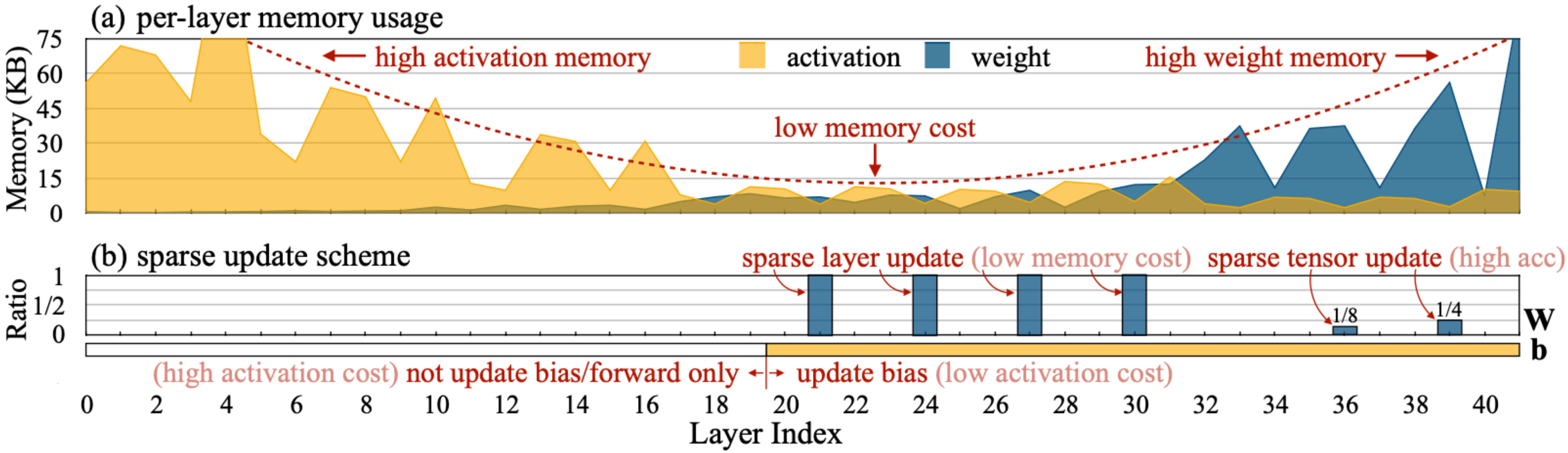


(d) Sparse layer/Sparse tensor update

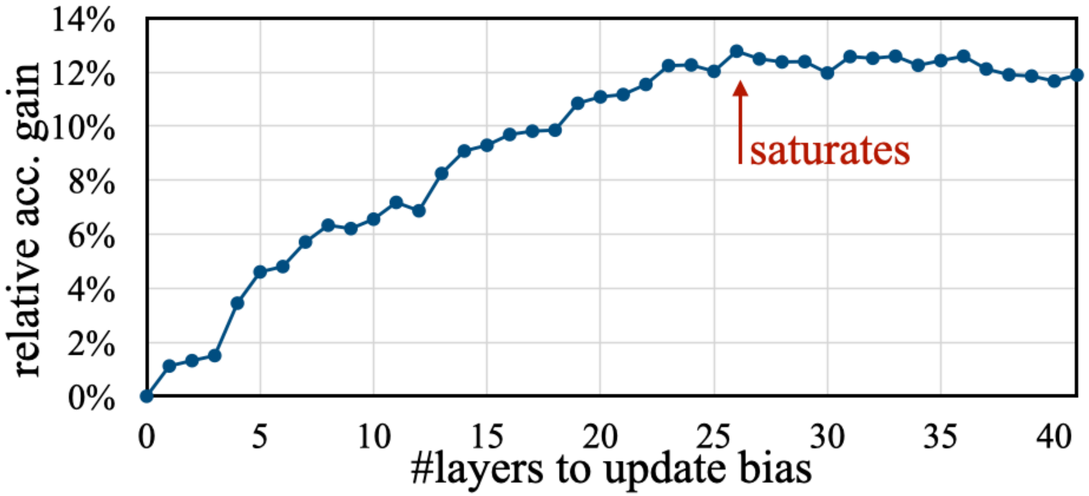
Find Layers to Update by Contribution Analysis

Which layer to update?

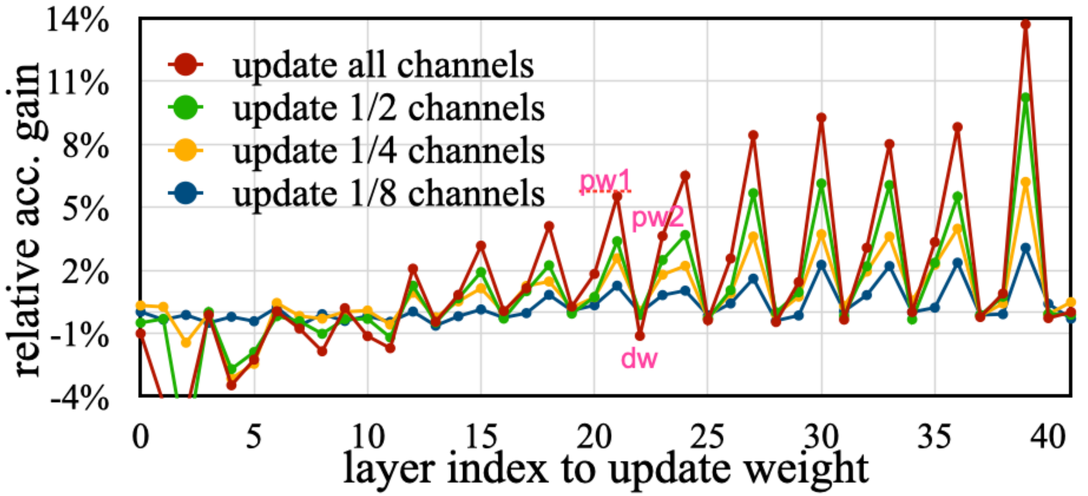
- The **activation cost** is high for the starting layers; the **weight cost** is high for the later layers; the **overall memory** cost is low for the middle layers.
- We update biases for the later layers (related to activation only), and weights for the intermediate layers (related to activation and weights)



Find Layers to Update by Contribution Analysis



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



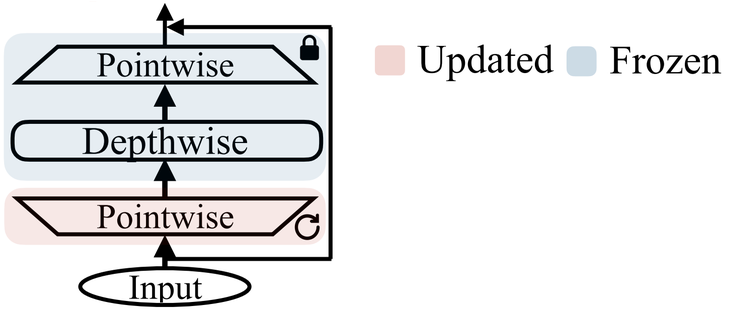
(b) Investigate the contribution of a certain weight $\Delta\text{acc}_{\mathbf{W}i,r}$

For bias update

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

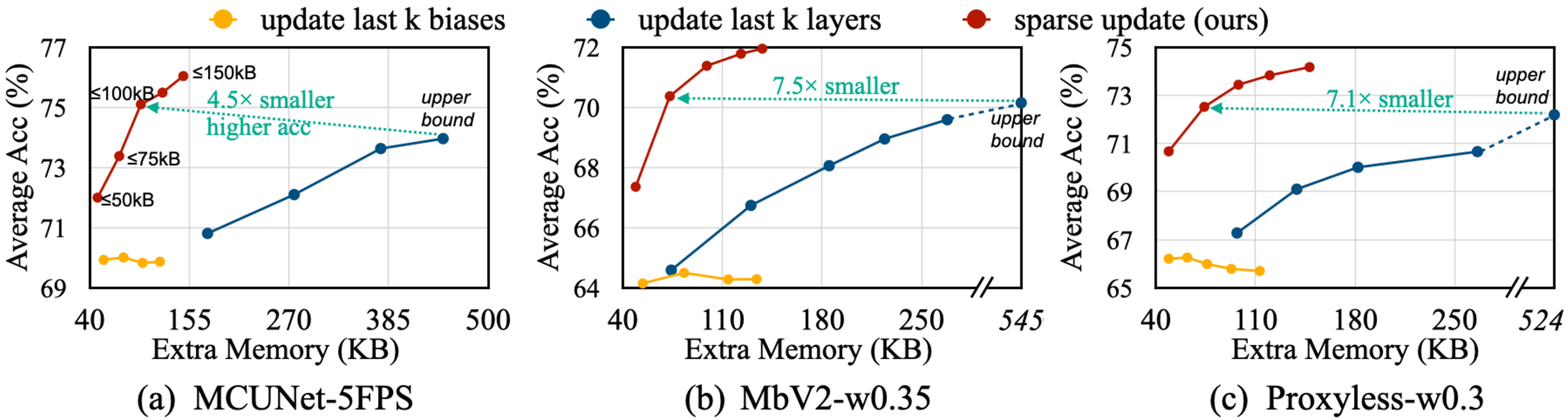
For weight update

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



$$k^*, \mathbf{i}^*, \mathbf{r}^* = \max_{k,\mathbf{i},\mathbf{r}}(\Delta\text{acc}_{\mathbf{b}[:k]} + \sum_{i\in\mathbf{i},r\in\mathbf{r}} \Delta\text{acc}_{\mathbf{W}i,r})$$

Sparse Update: Lower Memory, Higher Accuracy

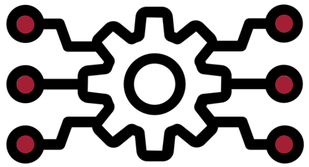


Sparse update can achieve higher transfer learning accuracy using **4.5-7.5x** smaller extra memory.

On-Device Training Under 256KB Memory

1. Quantization-aware scaling

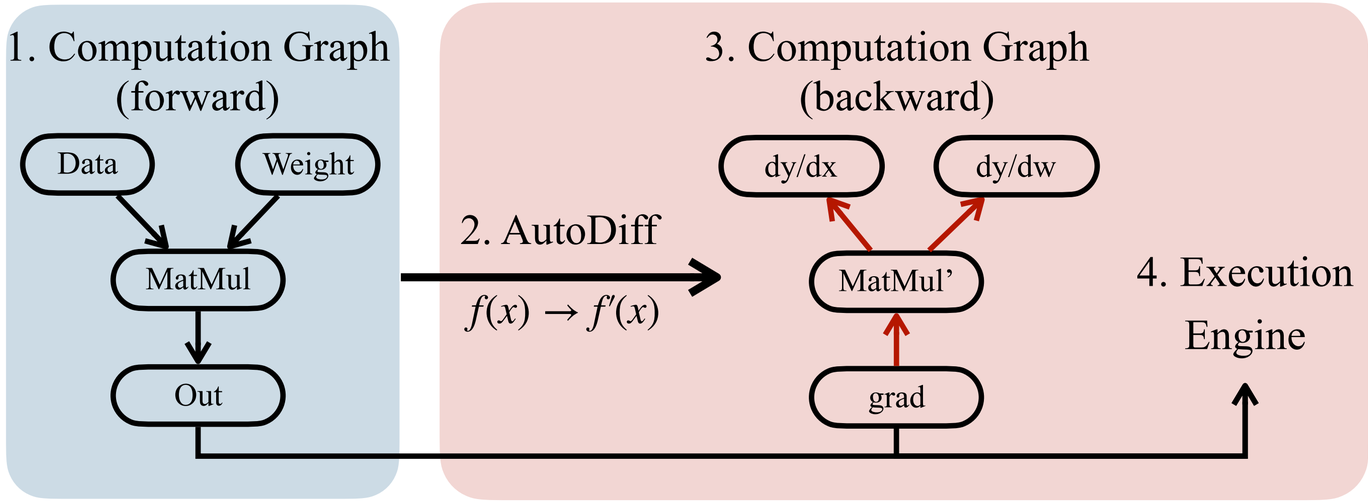
2. Sparse layer/tensor update



3. Tiny Training Engine

Tiny Training Engine (TTE)

Previous DL Training



Cannot apply many graph optimizations during runtime

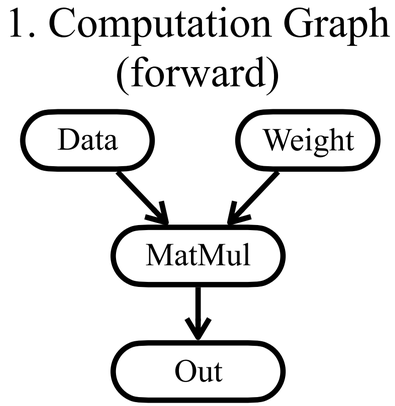
: Compile-Time

: Runtime

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

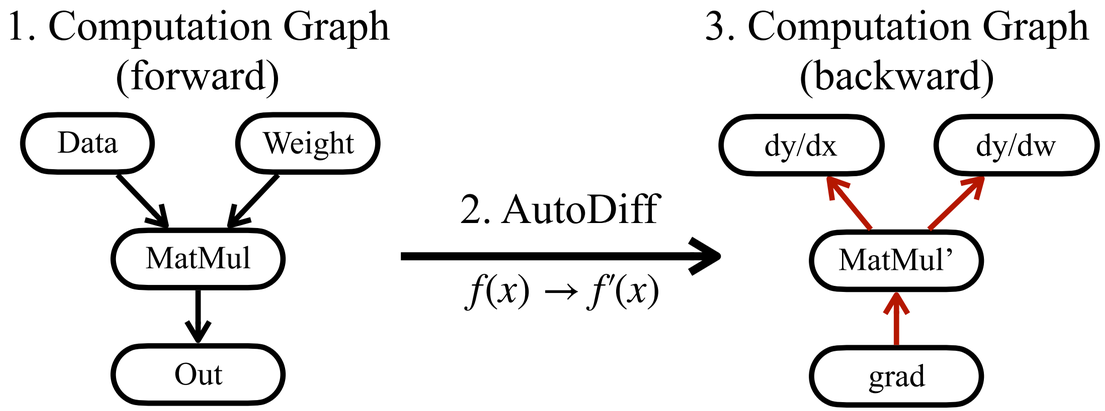
Tiny Training Engine (TTE)

Forward and Backward Computation Graph



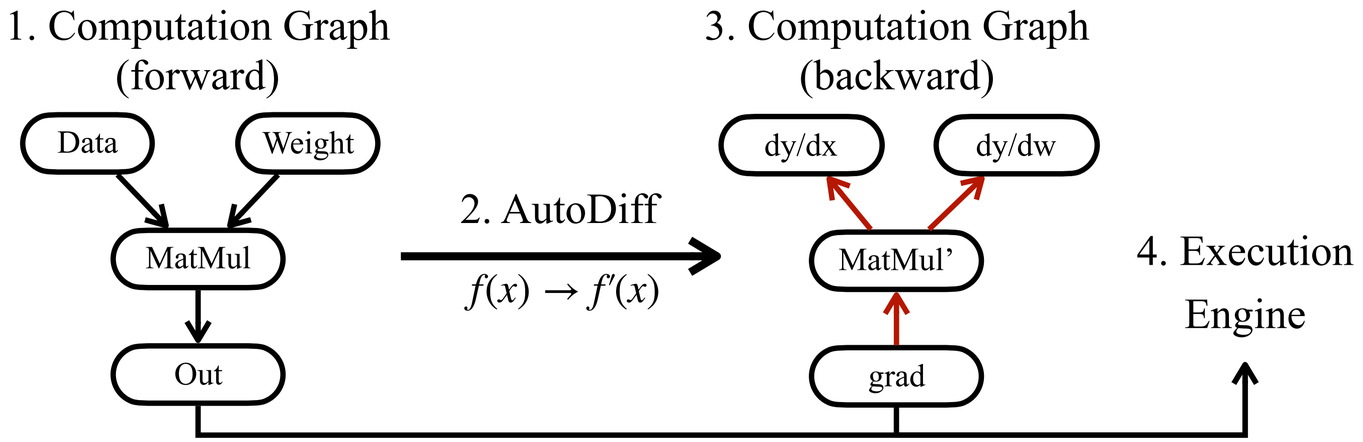
Tiny Training Engine (TTE)

Forward and Backward Computation Graph



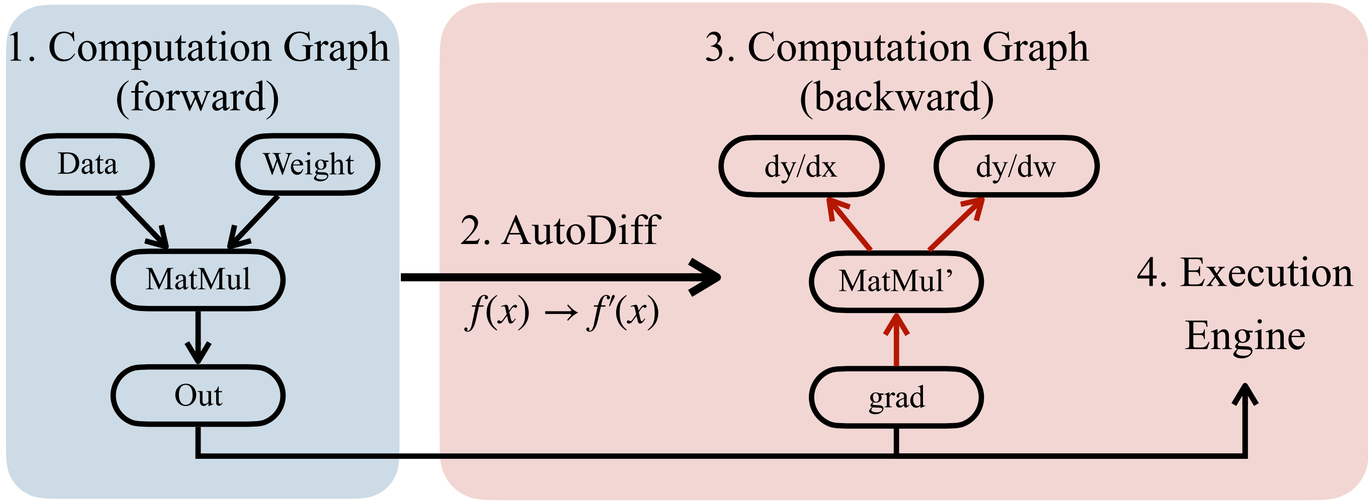
Tiny Training Engine (TTE)

Forward and Backward Computation Graph



Tiny Training Engine (TTE)

Forward and Backward Computation Graph



: Compile-Time

: Runtime

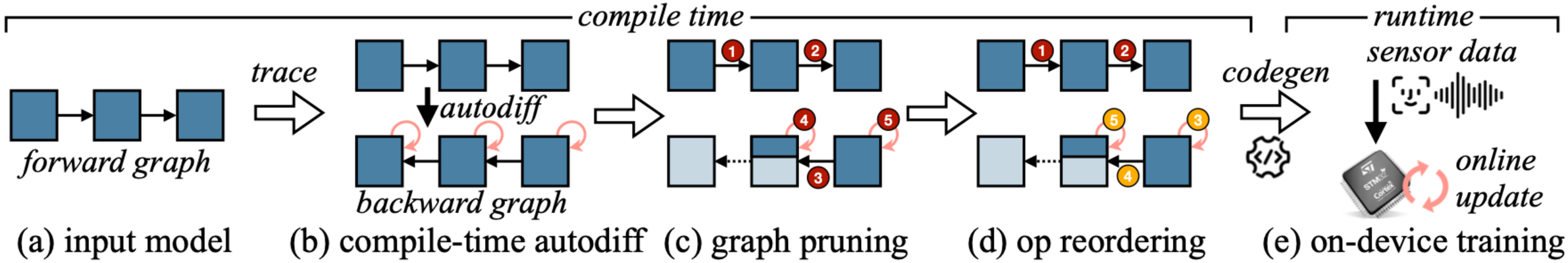
Conventional training framework focus on **flexibility**,
and the auto-diff is performed at **runtime**.
Thus, any optimizations will lead to runtime overhead.

Tiny Training Engine (TTE)

Existing frameworks cannot fit training into tiny devices due to:

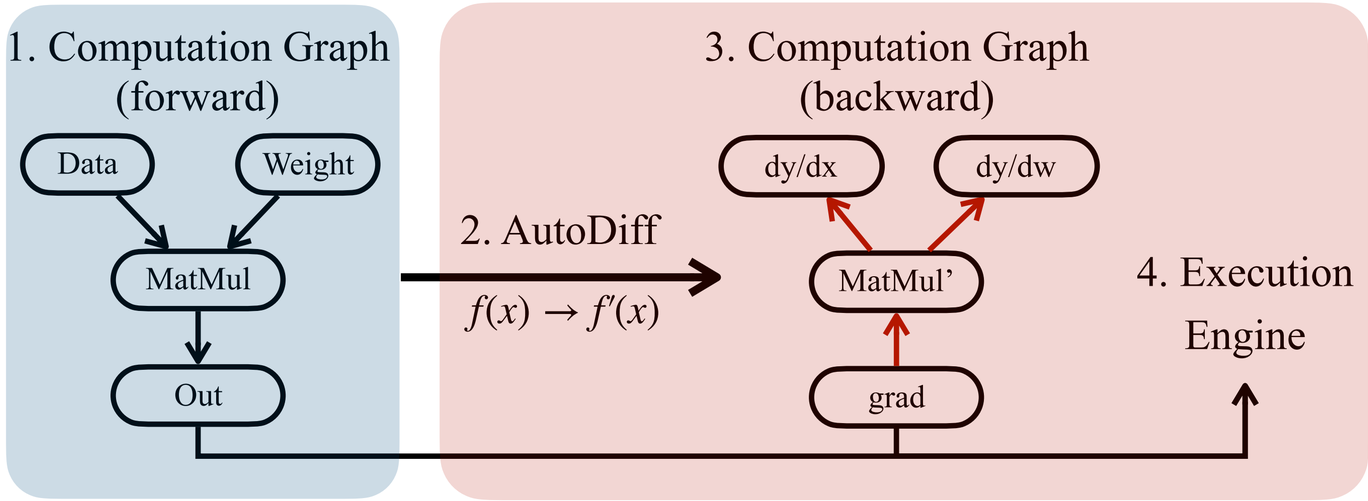
- **Runtime** is heavy
  - Heavy dependencies and large binary size (>**100MB** static memory)
  - Autodiff at runtime
  - Operators optimized for the cloud, not for edge
- **Memory** is heavy
  - A lot of intermediate (and unused) buffers
  - Has to compute full gradients

Tiny Training Engine (TTE)



Tiny Training Engine (TTE)

Compile-Time Autodiff



(a) Previous training frameworks

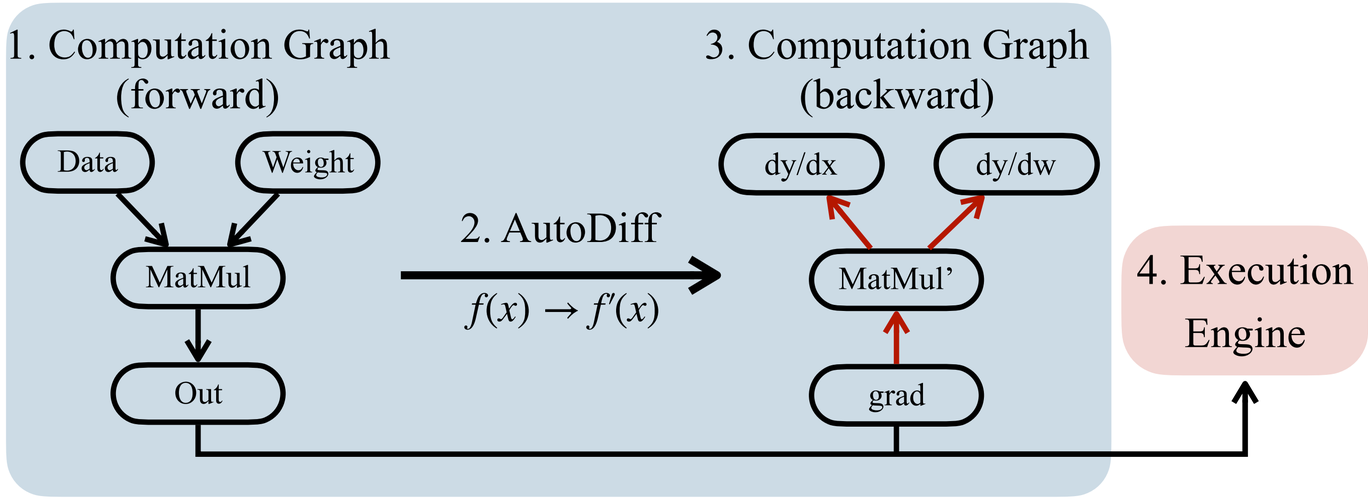
: Compile-Time

: Runtime

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

Tiny Training Engine (TTE)

Compile-Time Autodiff



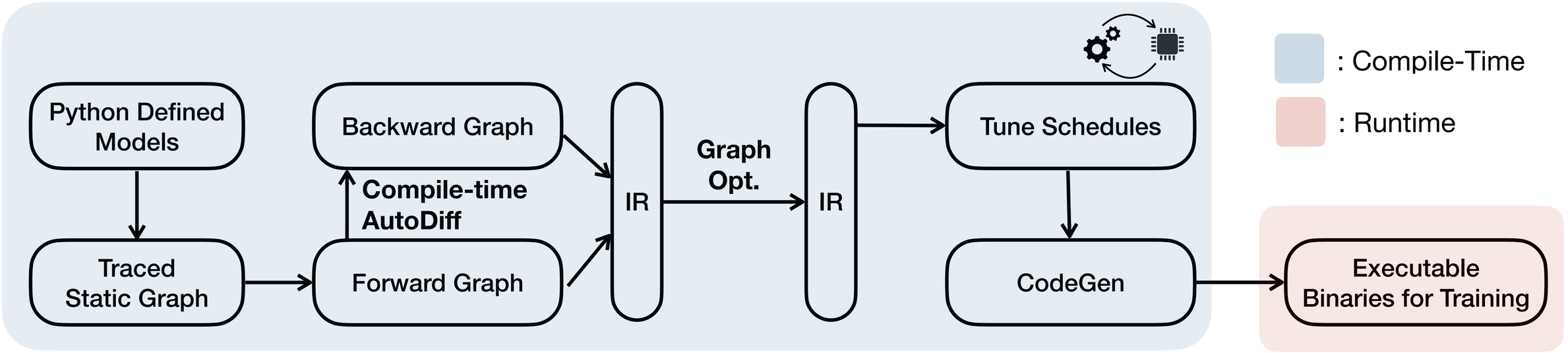
(b) Our Tiny Training Engine

: Compile-Time

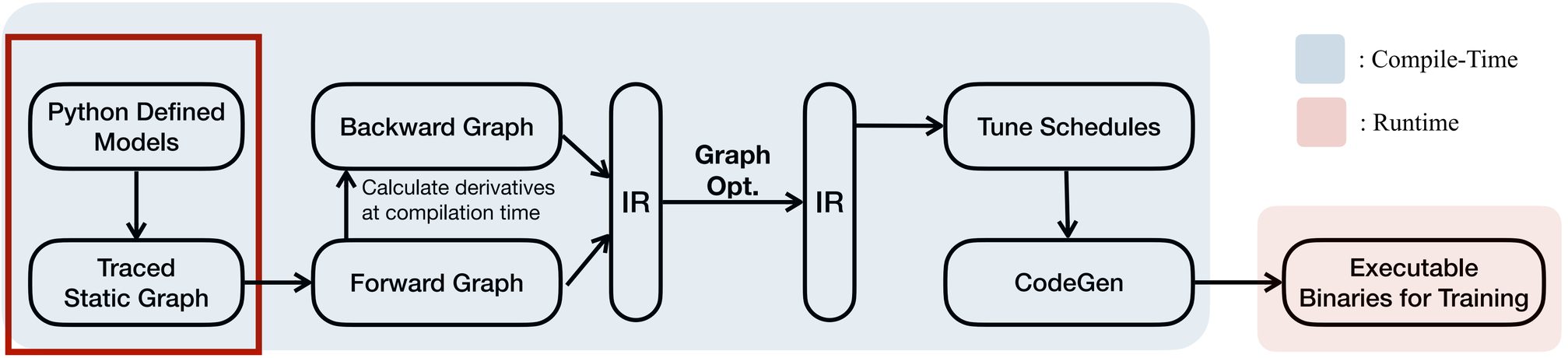
: Runtime

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

Tiny Training Engine (TTE)



Tiny Training Engine (TTE)



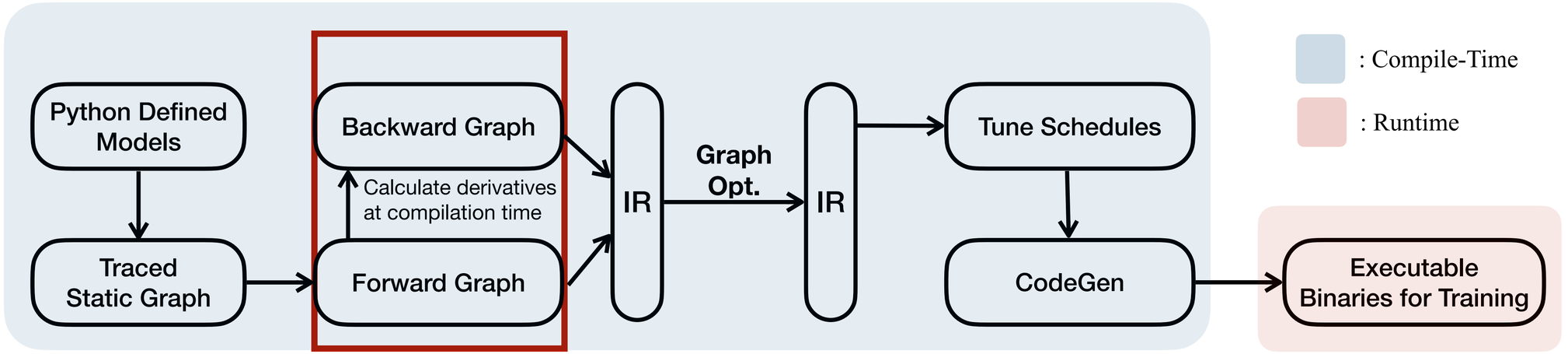
PyTorch

```
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

```
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 (TTE)



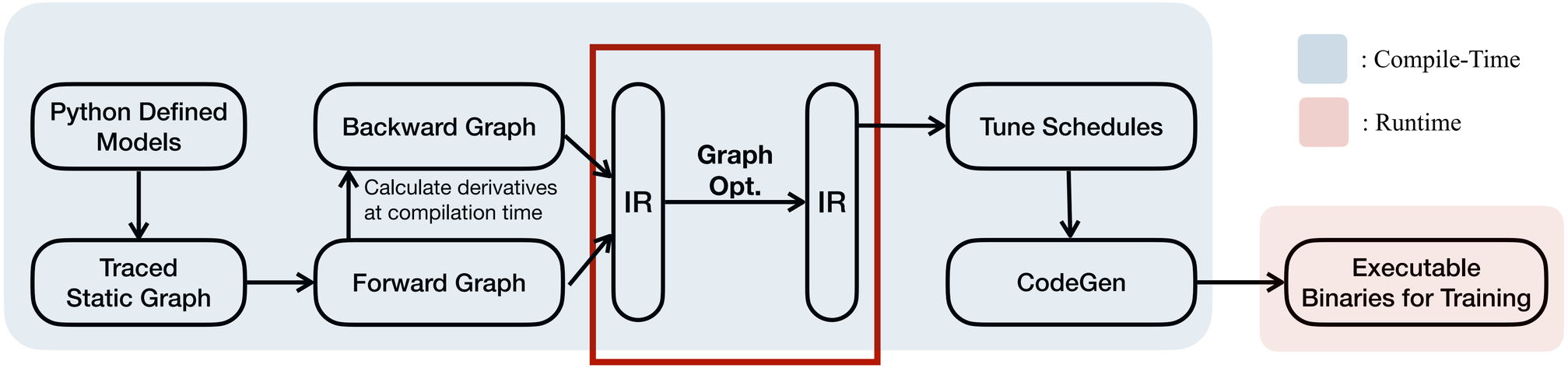
Forward IR

```
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);
}
```

Backward IR

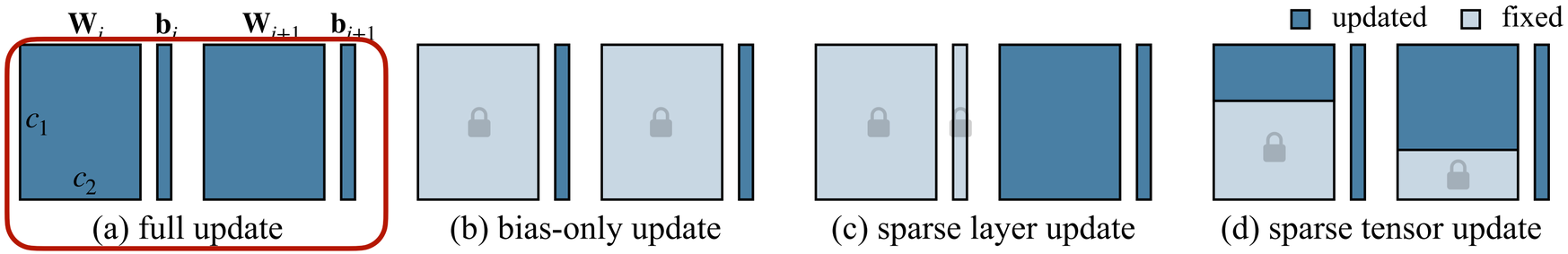
```
fn (%input0: Tensor[(1, 3, 28, 28), float32], %v0.weight: Tensor[(3,
3, 3, 3), float32], %grad_output: Tensor[(1, 3, 28, 28), float32]) {
  # forward
  %0 = nn.conv2d(%input0, %v0.weight, padding=[1, 1, 1, 1],
channels=3, kernel_size=[3, 3]);
  %1 = nn.relu(%0);
  # grad_input
  %2 = padding(%grad_output);
  %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, 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 (TTE)



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

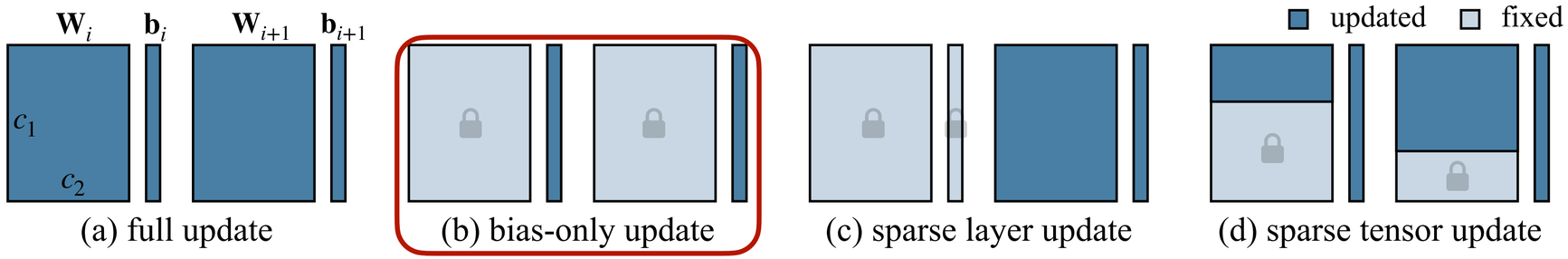
Tiny Training Engine (TTE)



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)
}
```

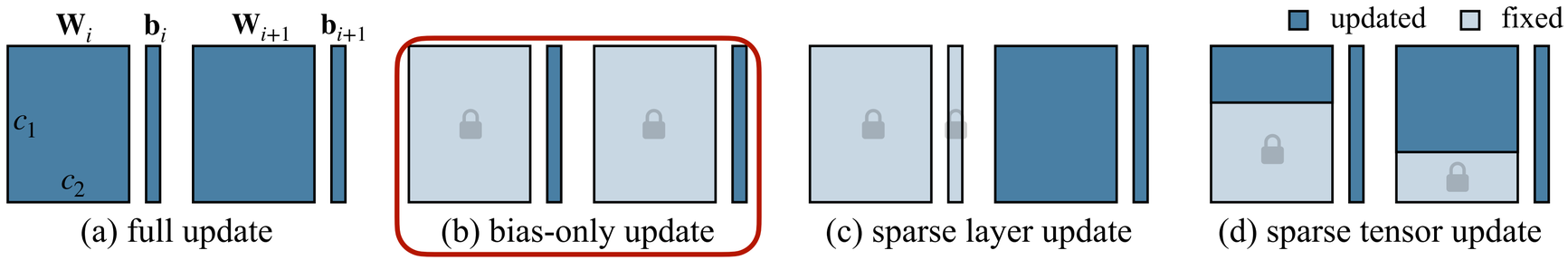
Tiny Training Engine (TTE)




```
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); ====> dy / dx
  %4 = transpose(%grad);
  %5 = multiply(%4, %x);         ====> dy / dw
  %6 = sum(%grad, axis=-1);      ====> dy / db
  (%3, %5, %6)
}
```

Annotate whether a tensor requires gradient or not

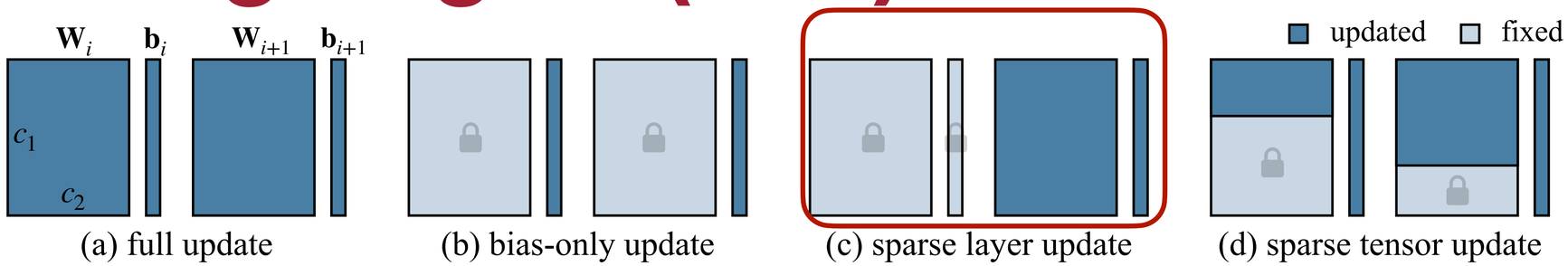
Tiny Training Engine (TTE)




```
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); ====> dy / dx
  %4 = transpose(%grad);
  %5 = multiply(%4, %x);          ====> dy / dw
  %6 = sum(%grad, axis=-1);       ====> dy / db
  (%3, %5, %6)
}
```

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

Tiny Training Engine (TTE)

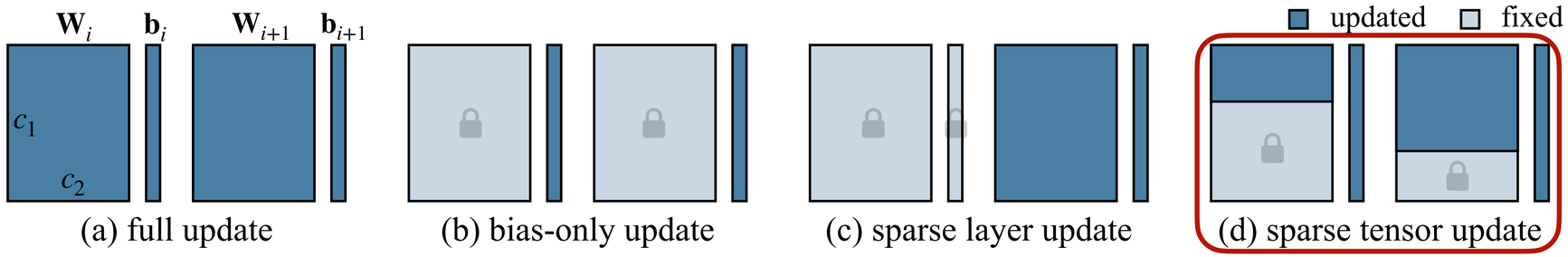



```
fn (%x: Tensor[(10, 10), float32, needs_grad=False],
    %weight1: needs_grad=False],
    %bias1: needs_grad=False],
    %weight2: needs_grad=True],
    %bias2: needs_grad=True],
    ……
    %grad: .., float32]),
{
  # …
}
```

Freely annotate **ANY** parameters

TTE will trim the computation accordingly.

Tiny Training Engine (TTE)



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

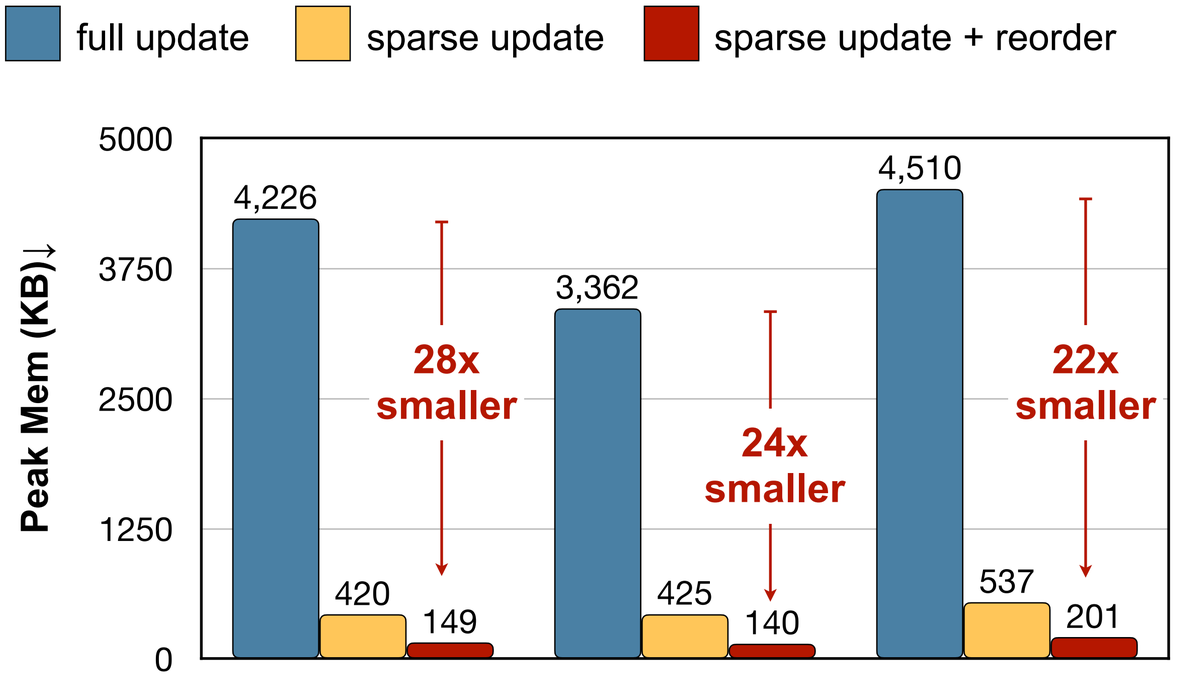
```
fn (%x: Tensor[(10, 10), float32],
    %weight: Tensor[(10, 10), float32],
    %bias: Tensor[(10), float32]),
    %grad: Tensor[(10), float32]),
{
  # forward
  %0 = multiply(%x, %weight);
  %1 = add(%0, %bias);
  # backward
  %3 = multiply(%grad, %weight);
  %4 = transpose(%grad)
  %5 = multiply(%4, %x);
  %6 = sum(%grad, axis=-1);
  (%3, %5, %6)
}
```

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

```
fn (%x: Tensor[(10, 10), float32, needs_grad=True],
    %weight: Tensor[(20, 10), float32, needs_grad=0.5],
    %bias: Tensor[(20), float32, needs_grad=True],
    %grad: Tensor[(10, 20), float32]),
{
  # forward
  %0 = multiply(%x, %weight);
  %0.1 = slice(%x, begin=[0, 0], ends=[10, 10]);
  %1 = add(%0, %bias);
  # backward
  %3 = multiply(%grad, %weight);
  %4 = transpose(%grad)
  %5 = multiply(%4, %0.1);
  %6 = sum(%grad, axis=-1);
  (%3, %5, %6)
}
```

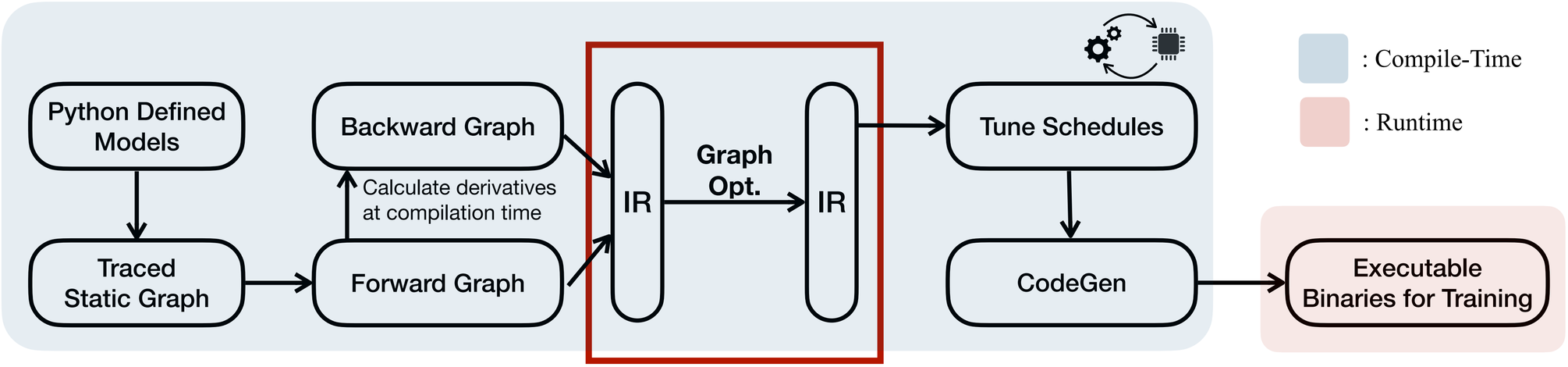
Tiny Training Engine (TTE)

Sparse update results



- 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

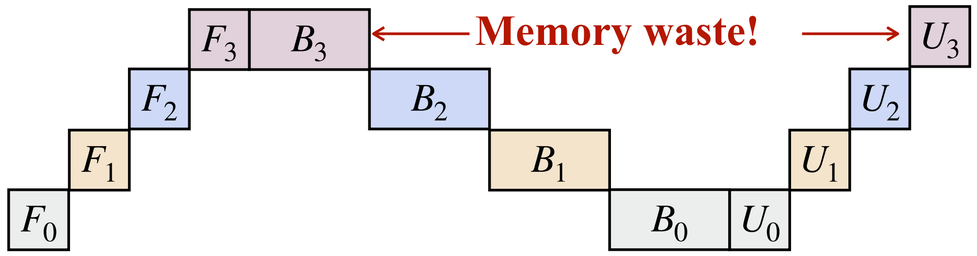
Tiny Training Engine (TTE)



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

Tiny Training Engine (TTE)

Re-ordering reduces memory footprint



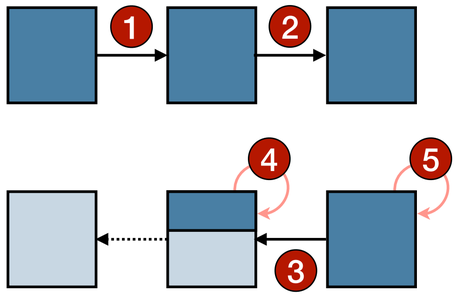
(a) Conventional way to update parameters

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

Tiny Training Engine (TTE)

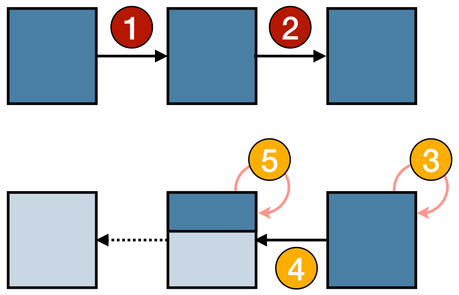
Re-ordering reduces memory footprint

```
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.

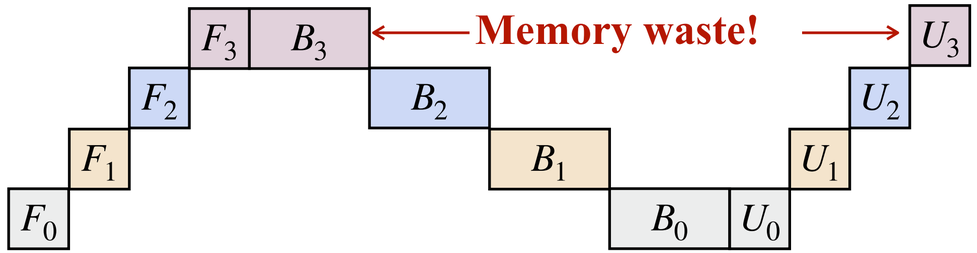
```
out = model(data)
loss = criterion(out, label)
for layers in model:
    dydx, grad = layers.backward(loss)
    optim.update(layers, grad)
    loss = dydx
```



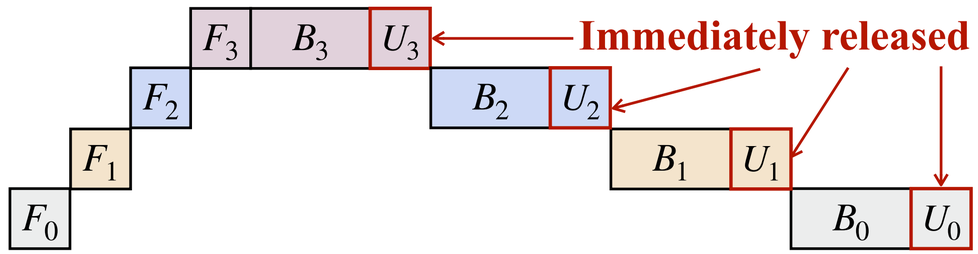
Gradient updates are **immediately applied** once calculated.
Intermediate buffers can be released.

Tiny Training Engine (TTE)

Re-ordering reduces memory footprint



(a) Conventional way to update parameters

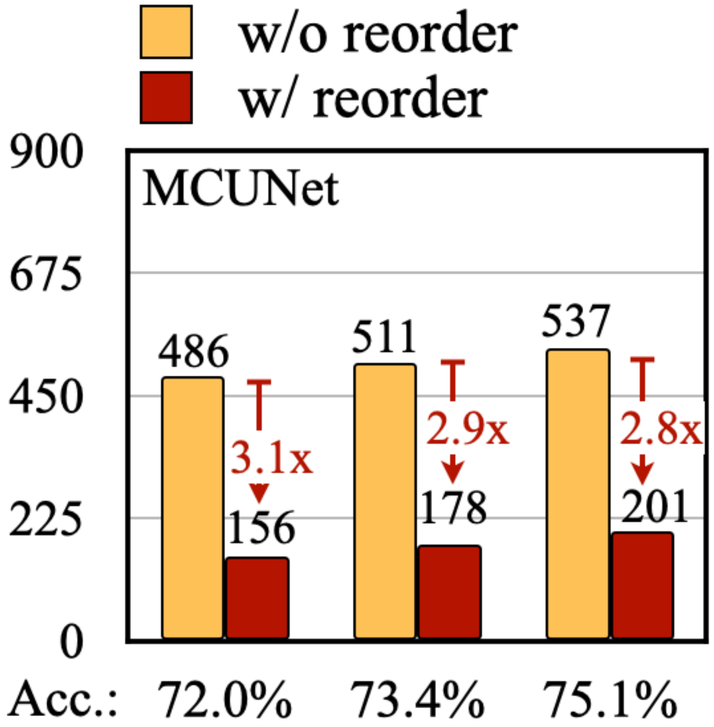


(b) Re-order to optimize gradient step.

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

Tiny Training Engine (TTE)

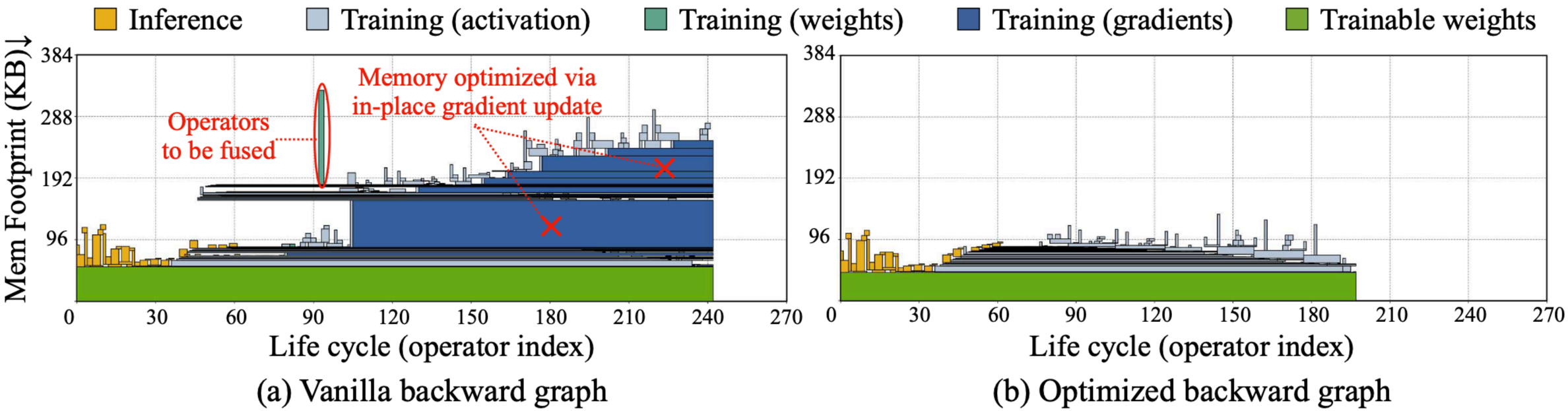
Re-ordering reduces memory footprint



By reordering, the gradient update can be immediately applied.Gradients buffer can be released earlier before before back-propagating to earlier layers, leading to **2.7x ~ 3.1x** peak memory reduction.

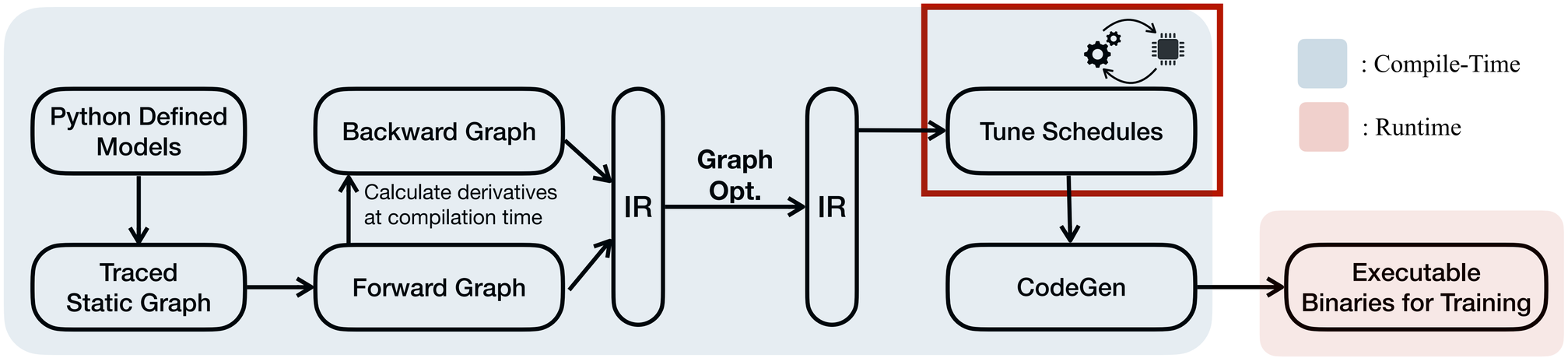
Tiny Training Engine (TTE)

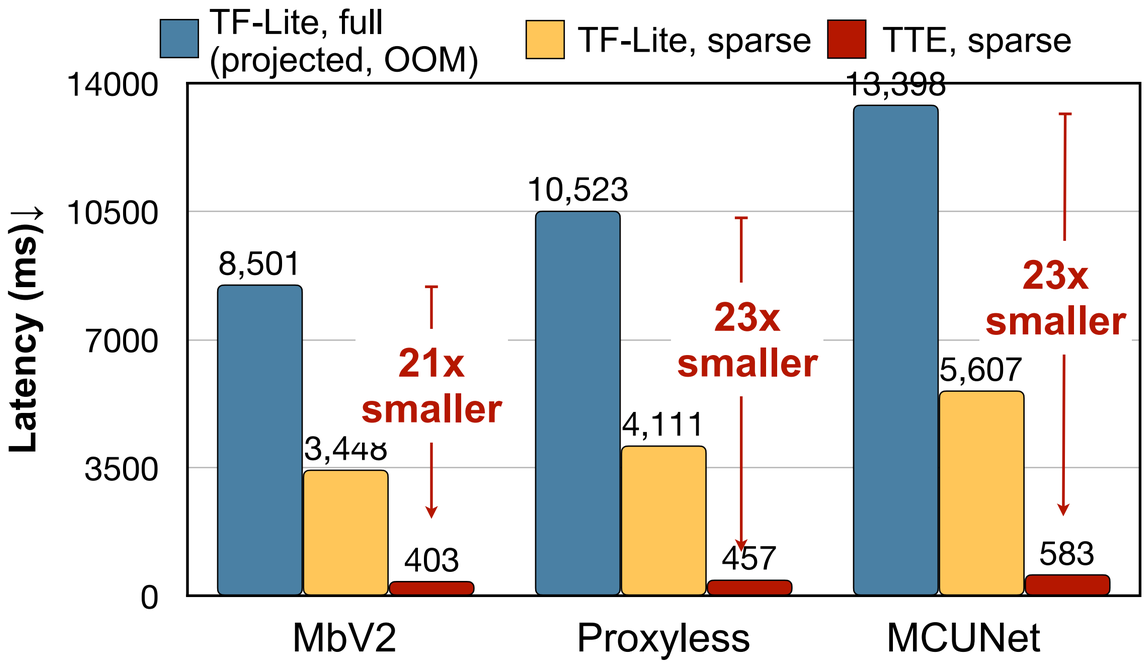
Re-ordering reduces memory footprint



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

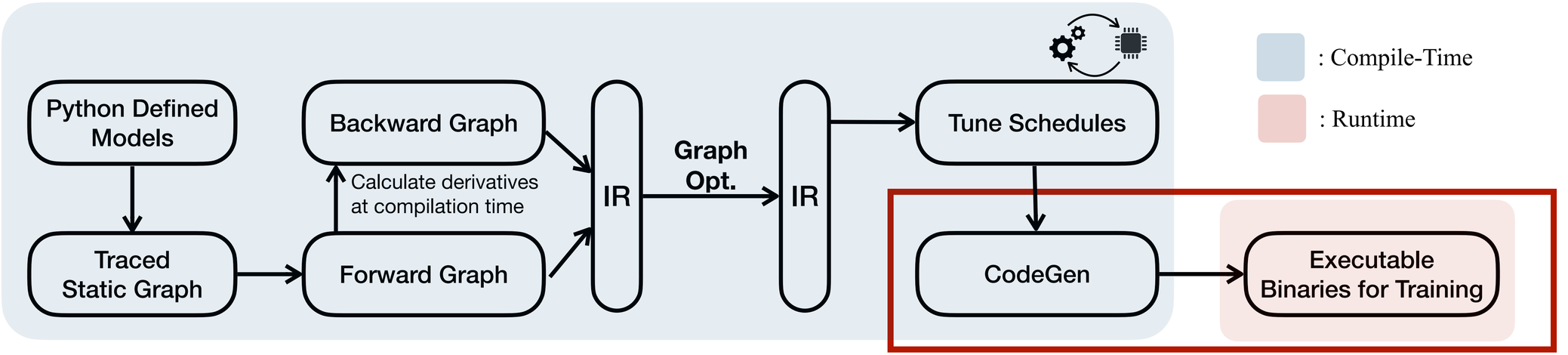
Tiny Training Engine (TTE)





Our optimized operators demonstrate **21x ~ 23x** speedup over TensorFlow-Lite.

Tiny Training Engine (TTE)



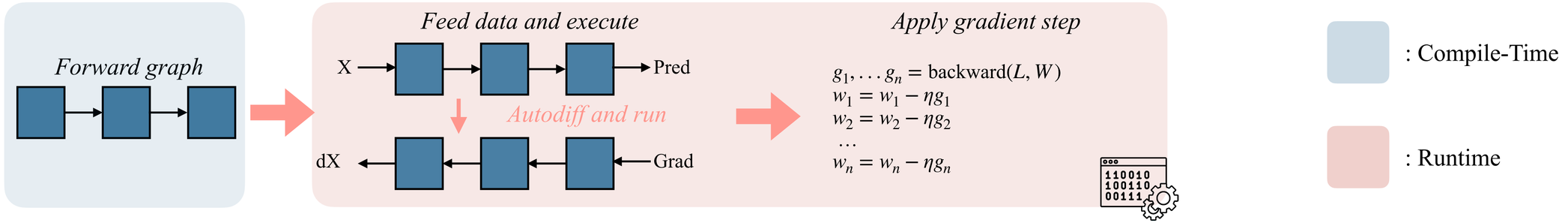

```
runtime::Module fwd_mod = runtime::Module::LoadFromFile("fwd.so");
runtime::Module bwd_mod = runtime::Module::LoadFromFile("bwd.so");

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

Our codegen only generate binaries for used operators

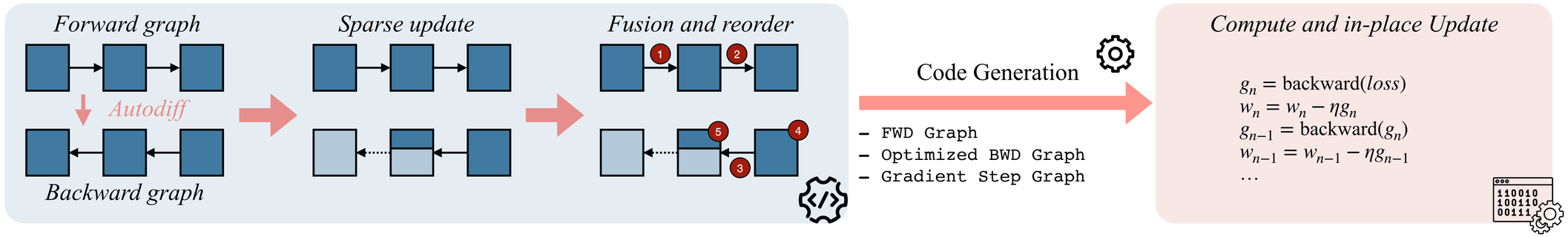
TTE finally deliver a light-weight, portable, and efficient binary.

Tiny Training Engine (TTE)

Comparison of Previous Infra and TTE



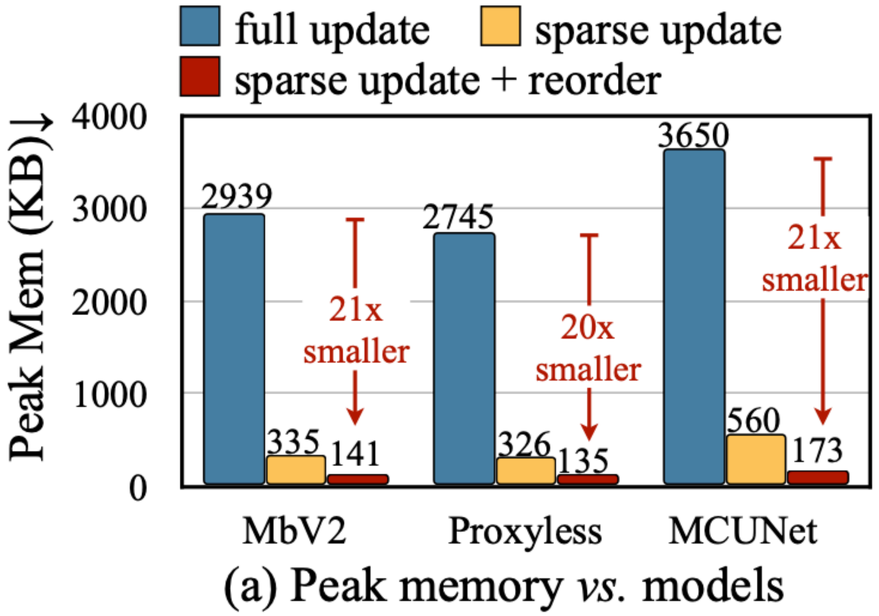
Conventional training framework performs most tasks at runtime.



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

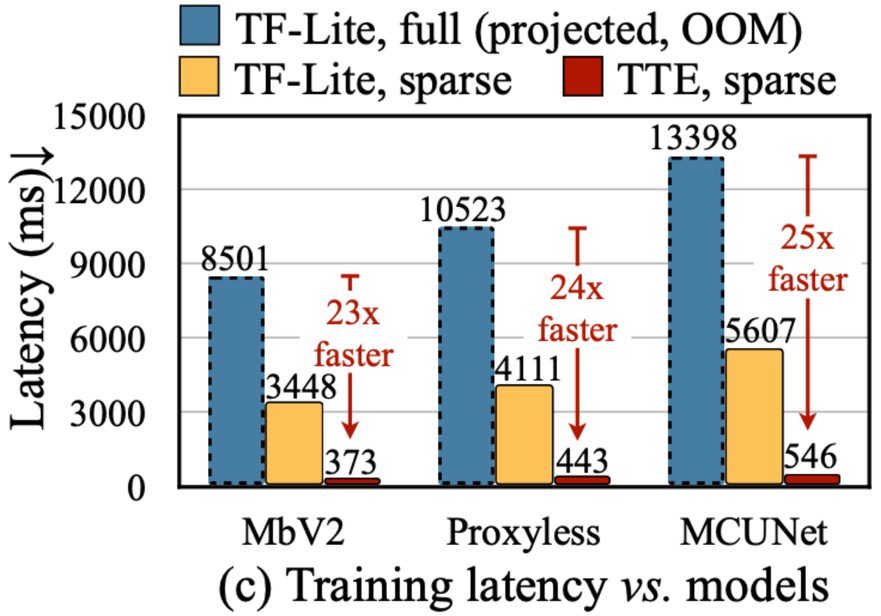
Tiny Training Engine (TTE)

Smaller memory usage, faster training speed



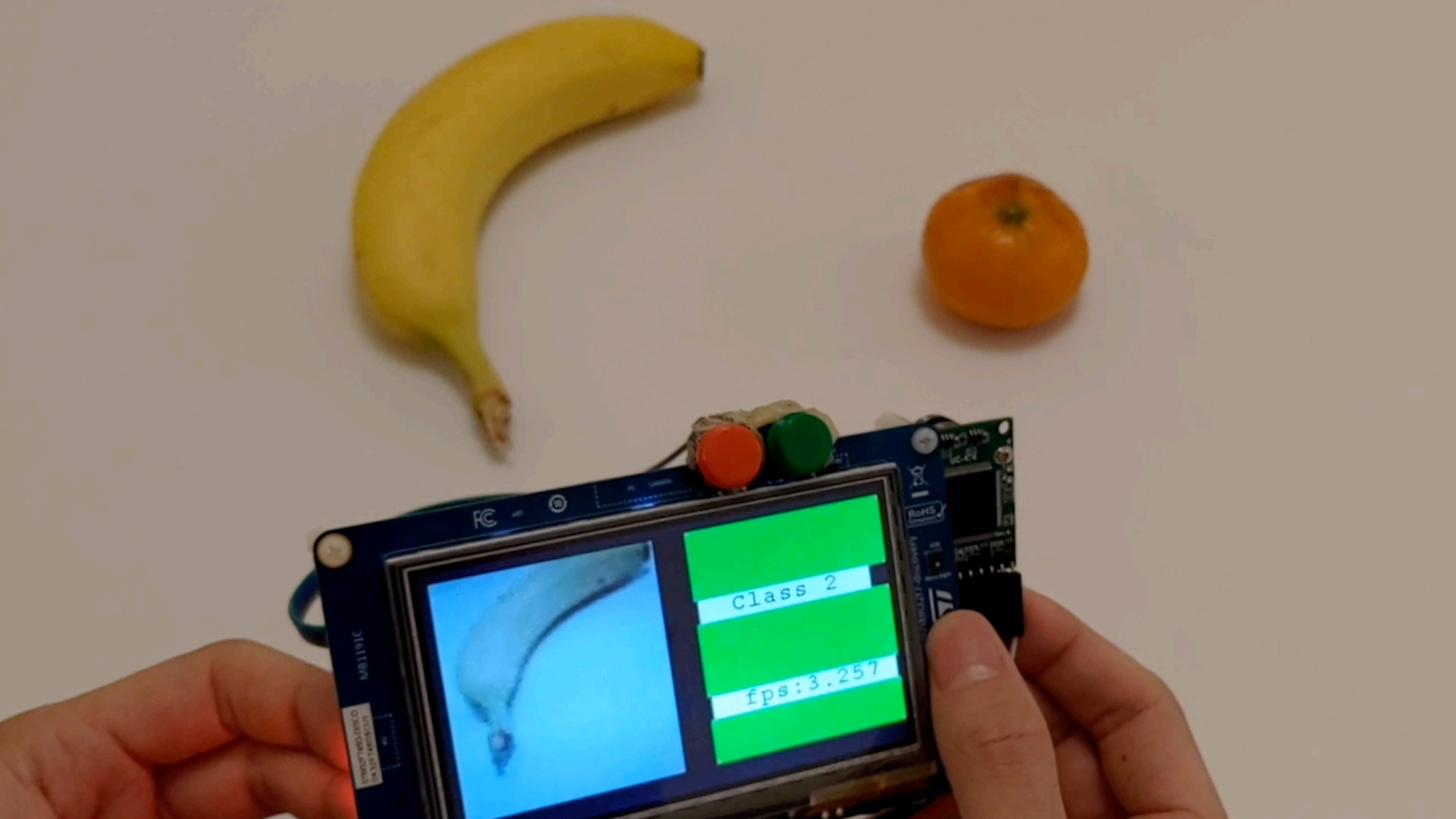
(a) Peak memory *vs.* models

20x smaller memory



(c) Training latency *vs.* models

23x faster speed

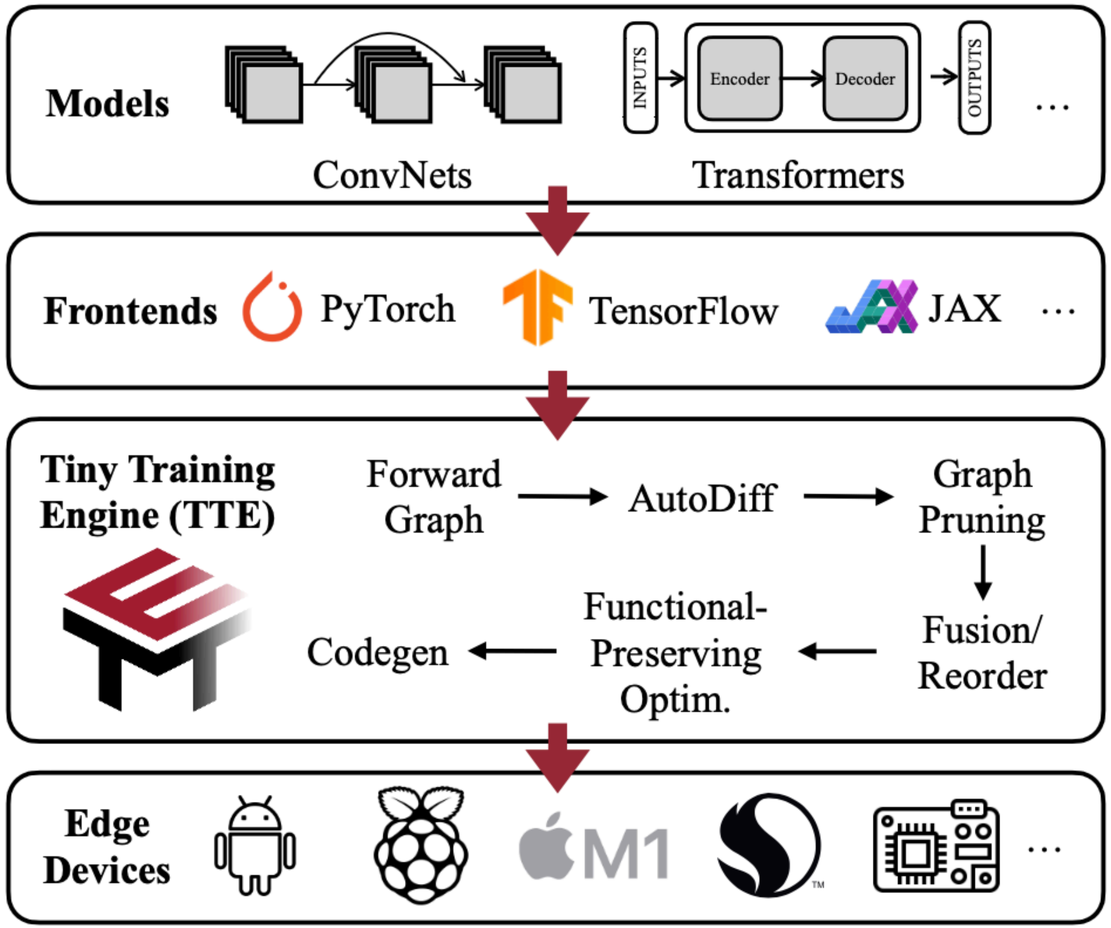

Class 2
fps:3.257

2. On-device training

Extending TTE to More Platforms

Accelerate on-device training on diverse edge hardware

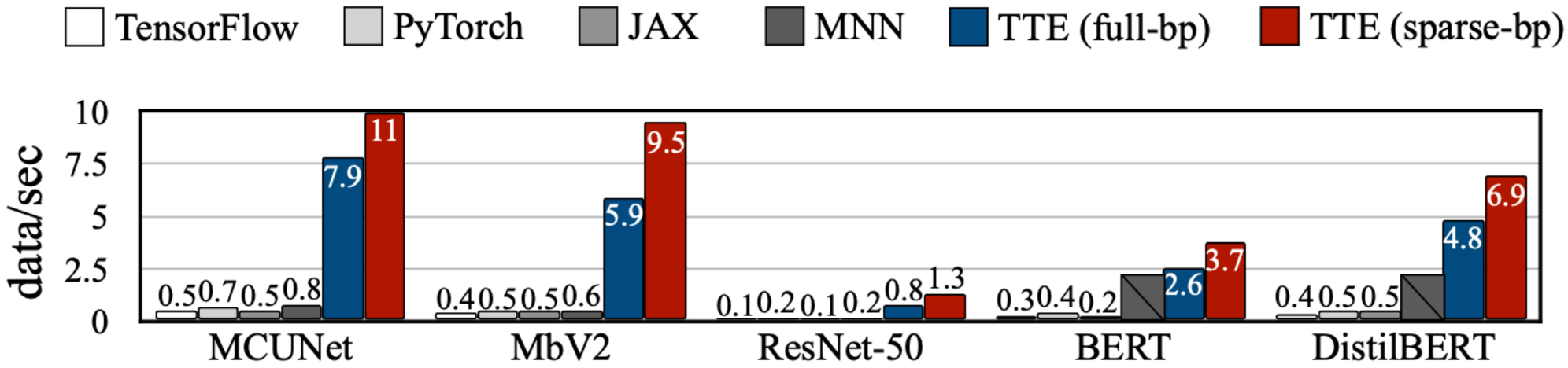
- We extend TTE to support:
  - Diverse models (CNN + Transformers)
  - Diverse frontends
    - PyTorch
    - TensorFlow
    - Jax
  - Diverse hardware backends
    - Apple M1
    - Raspberry Pi
    - Smartphones
    - …



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



Results measured on Raspberry Pi 4B+.

Explore websites, people, and locations

amy finkelstein

Top resources for

_ prospective students
_ current students
_ faculty & staff
_ alumni
_ parents
_ Covid-19 and MIT
_ all resources

Join us in building a better world.

A new technique enables AI models to continually learn from new data on intelligent “edge devices” like smartphones and sensors, reducing energy costs and privacy risks. The advance “makes deep learning more accessible,” says Song Han.

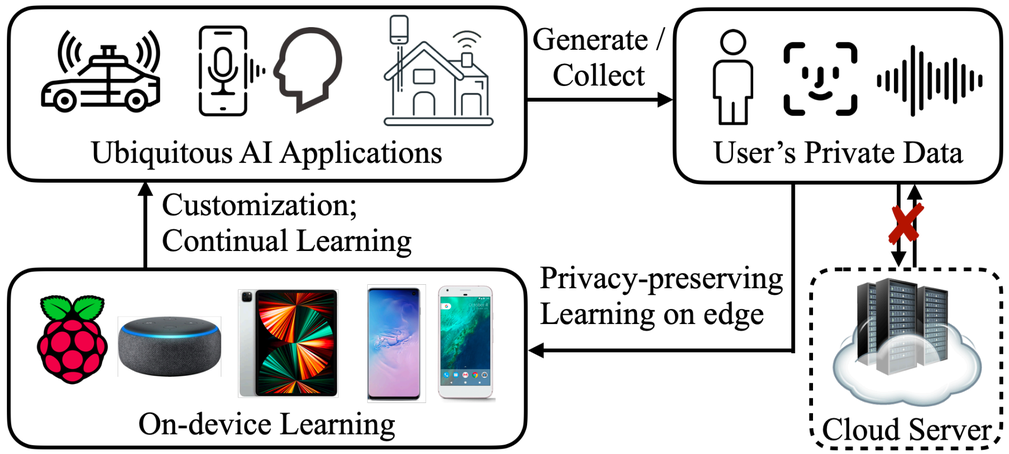
Oct 4, 2022 Full story Share: Explore more spotlights

On-Device Training Take-home

- **Quantized Training**
  - Fake quantization does not save memory
  - Real quantized graph achieves saving, however, is hard to optimize.
  - With quantization-aware scaling (QAS), we can optimize quantized graph.
- **Sparse Update**
  - Sparse learning happens in human brain
  - Update only important layers and parameters to save memory.
- **System Support**
  - Move workloads to compile-time (such as auto-diff) to minimize runtime cost.
  - Optimized schedules and kernels to improve throughput.

Summary of Today's Lecture

- We learned on-device transfer learning algorithms.
  - The training memory bottleneck is the activation.
  - Efficient transfer learning with bias-only and lite-residual.



- The co-design points of on-device training
  - Why training a quantized model is difficult and how to improve.
  - Full-update is too expensive and using sparse update for on-device training.
  - System support for efficient on-device training.

References

- Return of the devil in the details: Delving deep into convolutional nets [Chatfield. 2014]
- Do better imagenet models transfer better? [Kornblith. 2019]
- TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai et al. NeurIPS 2020]
- K for the Price of 1: Parameter-efficient Multi-task and Transfer Learning [Mudrarkarta et al., ICLR 2019]
- Do We Have Brain to Spare? [Drachman et al. 2004]
- Peter Huttenlocher (1931–2013) [Walsh. 2013]
- MCUNet: Tiny Deep Learning on IoT Devices [Lin et al 2020]
- On-Device Training Under 256KB Memory [Lin et al. 2022]