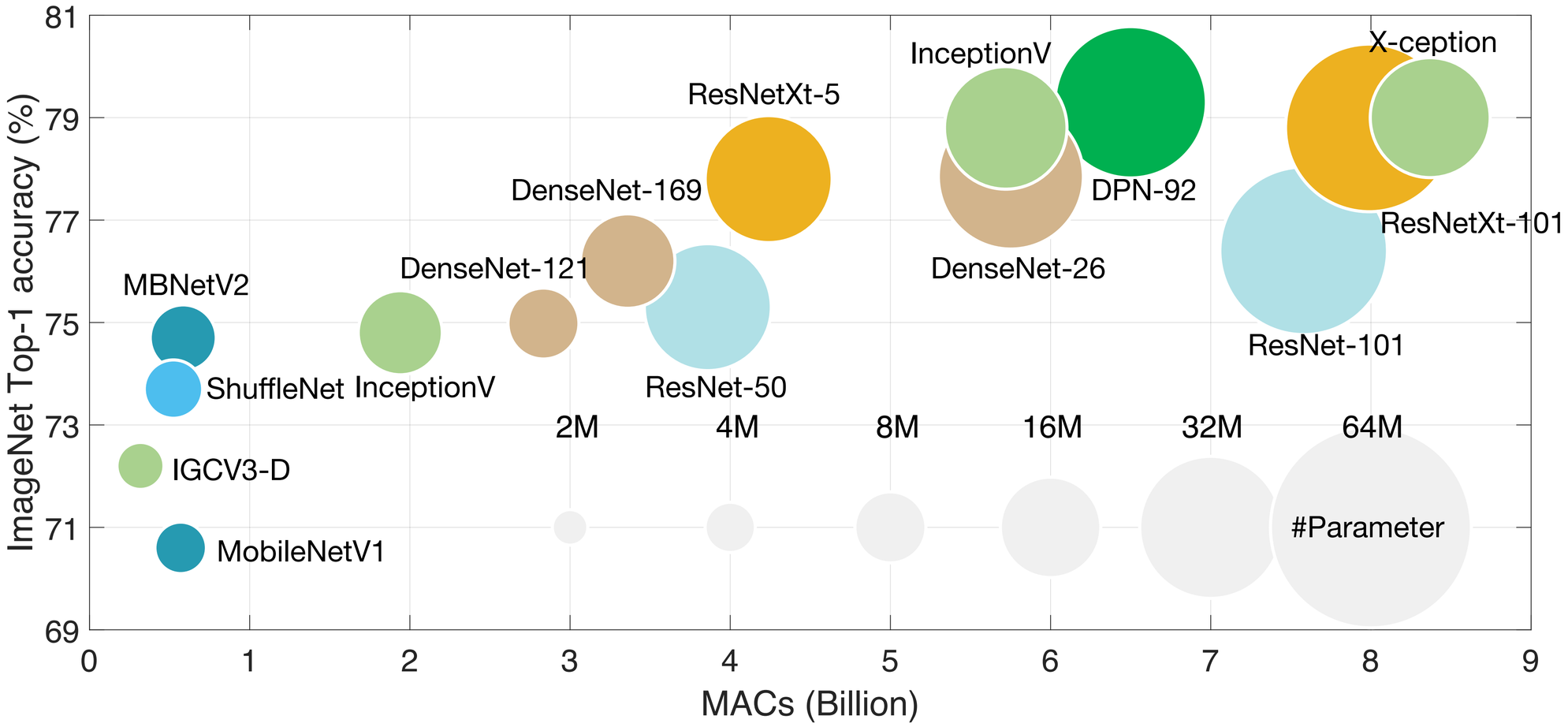
MIT HAN LAB

Algorithm-System Co-Design for TinyML

Ligeng Zhu
ligeng@mit.edu
MIT

Today's AI is too BIG

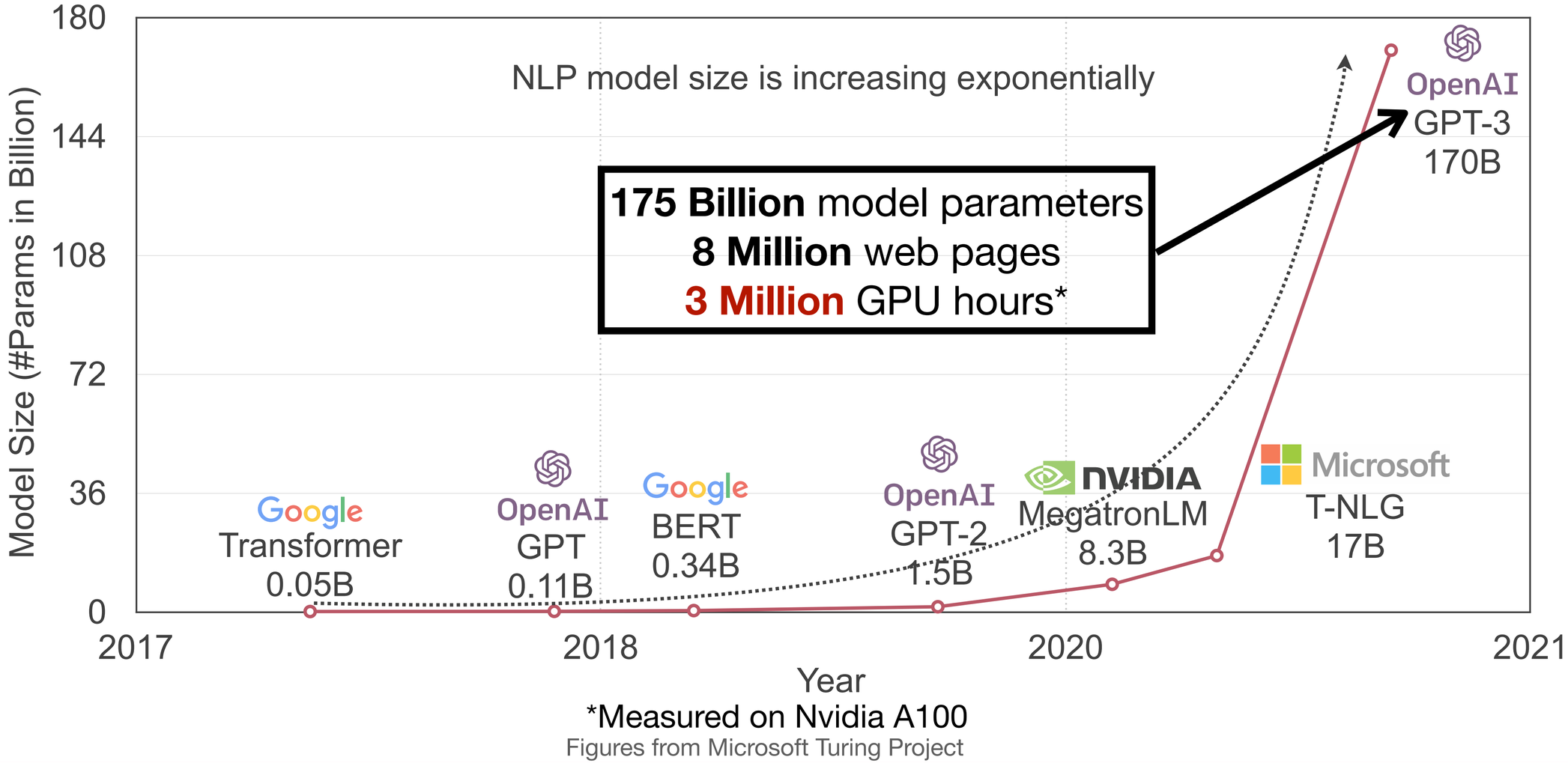
Better model always comes with higher computational cost (vision)



Figures from Once-for-all project page.

Today's AI is too BIG

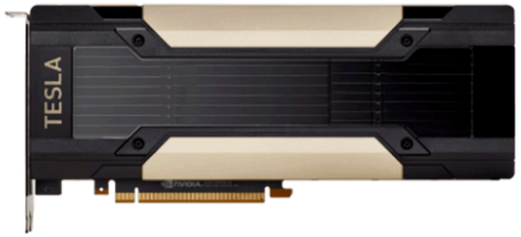
Better model always comes with higher computational cost (NLP)



Figures from Microsoft Turing Project

Deep Learning Going “Tiny”

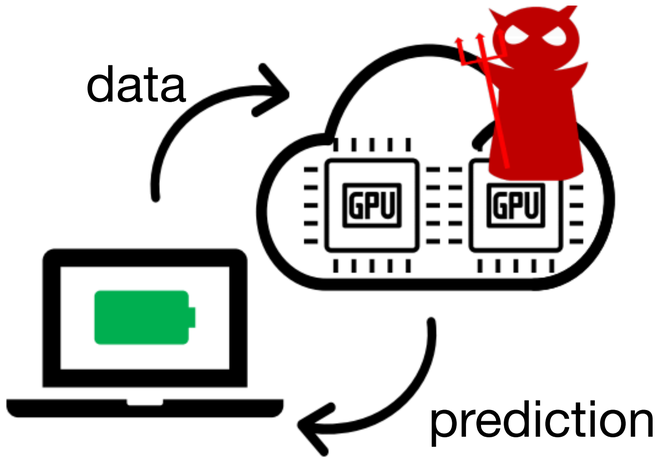
Cloud → Mobile → Tiny



Cloud AI

GPUs/TPUs

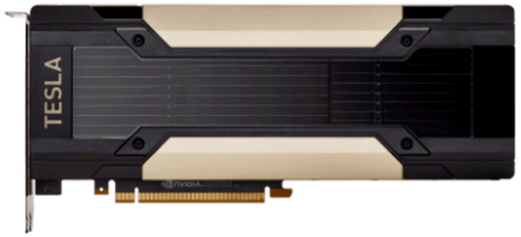
ResNet

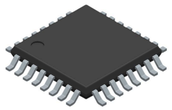


- Data uploaded to the cloud for inference/training

Deep Learning Going “Tiny”

Cloud → Mobile → Tiny





Cloud AI

GPUs/TPUs

ResNet

Mobile AI

Smartphones

MobileNet

Tiny AI

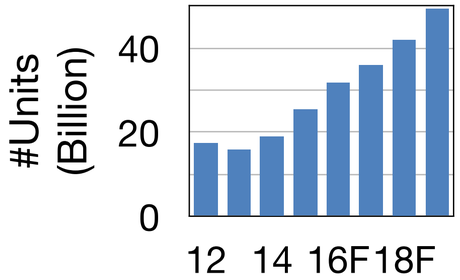
IoT/Microcontrollers

MCUNet

Deep Learning Going “Tiny”

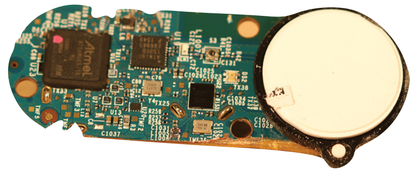
Squeezing deep learning into IoT devices

- 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



Ubiquitous

Low-cost
($0.1 - $10)



Low-power
(mW)

Deep Learning Going “Tiny”

Squeezing deep learning into IoT devices

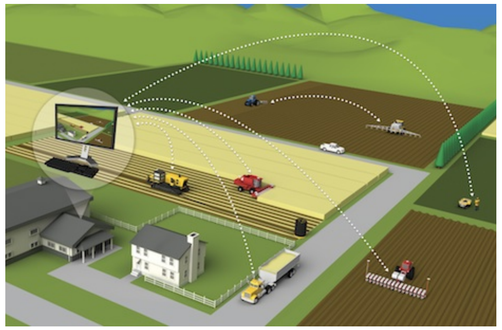
- Billions of IoT devices around the world based on **microcontrollers**
- **Low-cost**: low-income people can afford access. Democratize AI.
- **Low-power**: **green AI**, reduce carbon
- Various applications

Smart Home

Smart Manufacturing

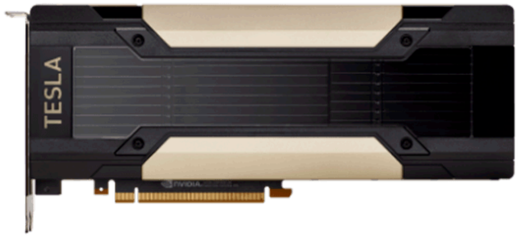
Personalized Healthcare

Precise Agriculture



TinyML is Challenging

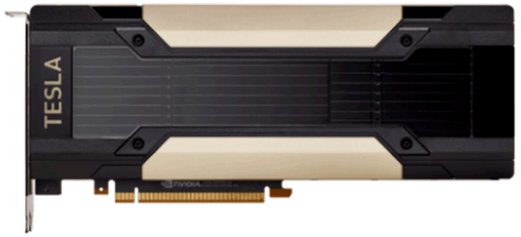
Memory size is too small to hold DNNs



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

TinyML is Challenging

Memory size is too small to hold DNNs

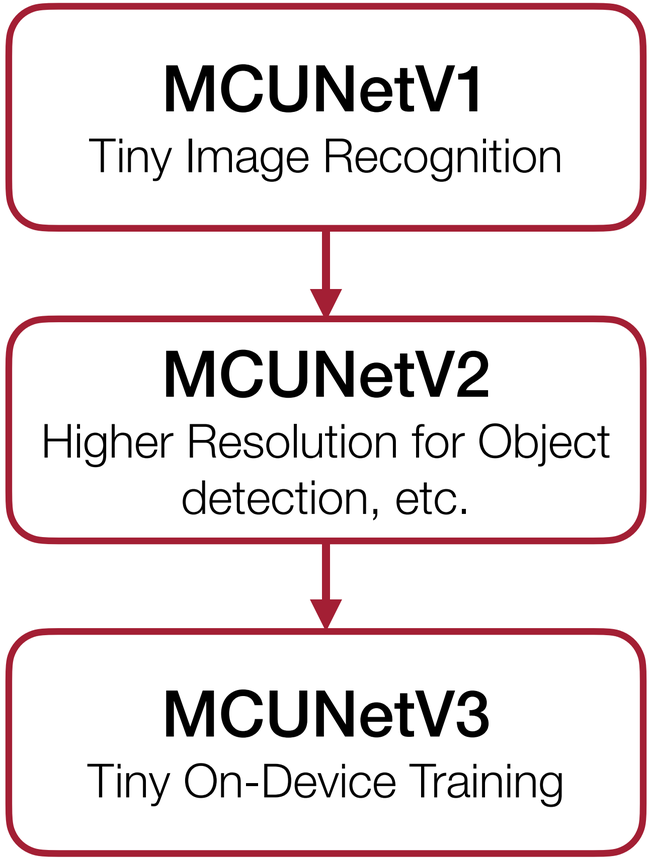


	Cloud AI	Mobile AI	Tiny AI
Memory (Activation)	32GB	4GB	320kB
Storage (Weights)	~TB/PB	256GB	1MB

13,000x smaller

100,000x smaller

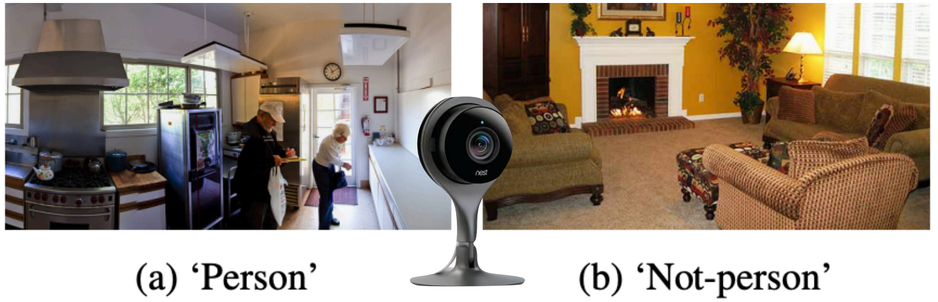
Overview



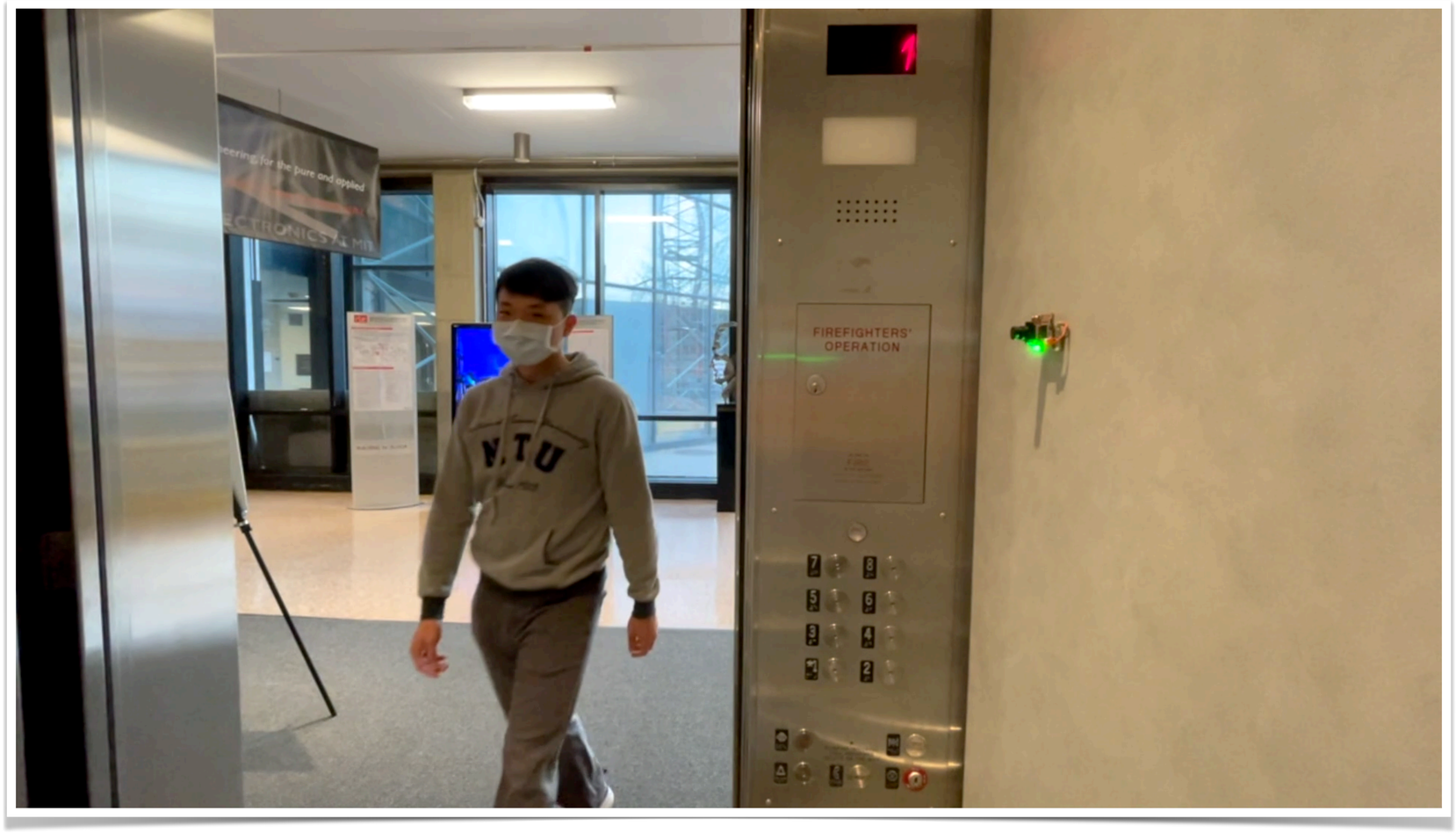
MCUNet: Tiny Deep Learning on IoT Devices [Lin *et al.*, NeurIPS 2020]
MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning [Lin *et al.*, NeurIPS 2021]
On-Device Training Under 256KB Memory [Lin *et al.*, NeurIPS 2022]

MCUNetV1 - Classification

Tiny vision application: visual wake words



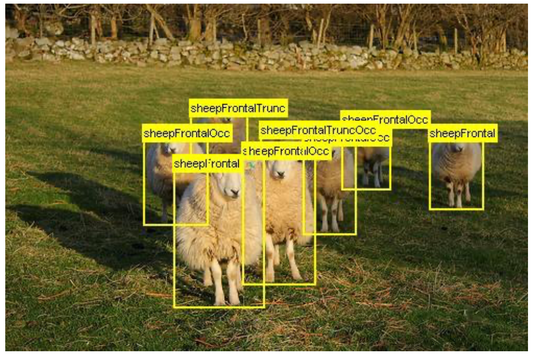
(a) 'Person' (b) 'Not-person'



Visual wake words dataset. [Chowdhery *et al.*, arXiv 2019]

MCUNetV2: Detection

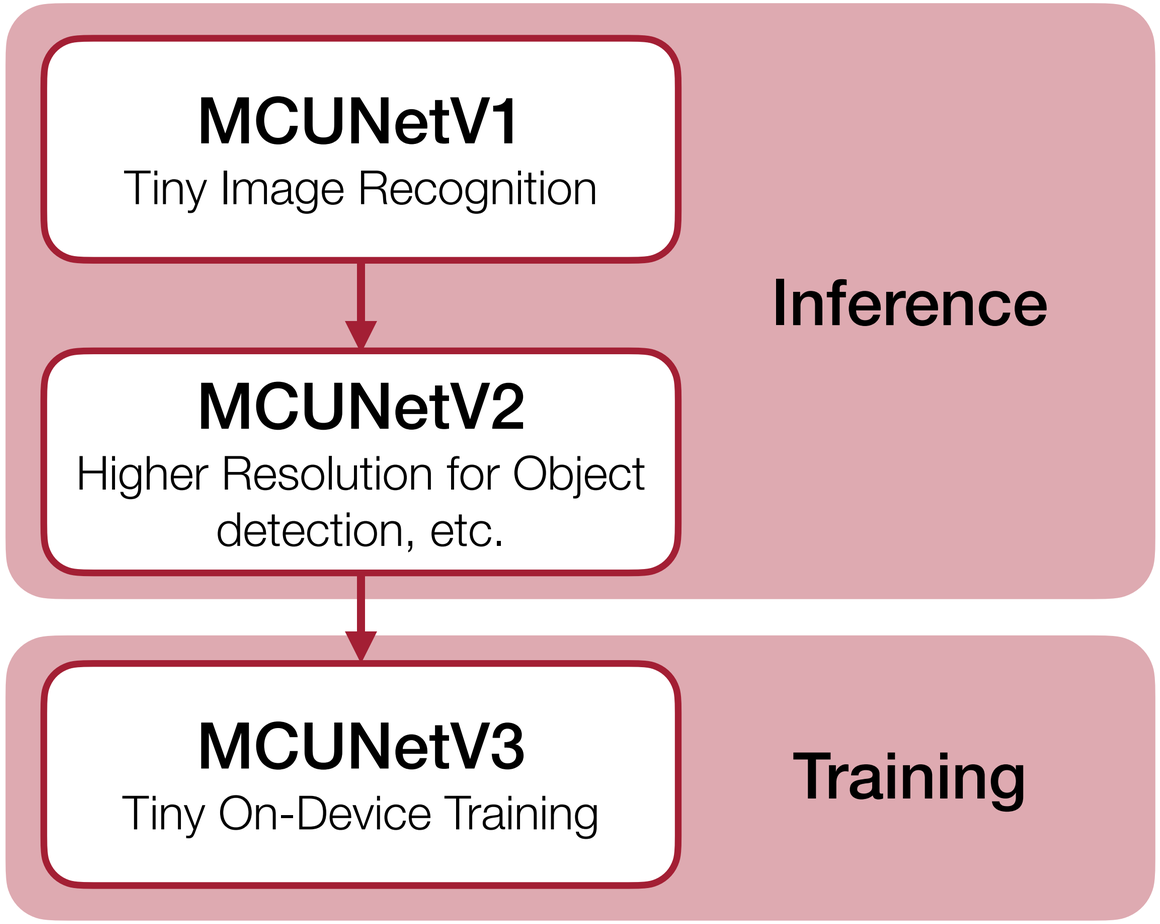
Advancing object detection by allowing a larger resolution



Face/mask detection

Person detection

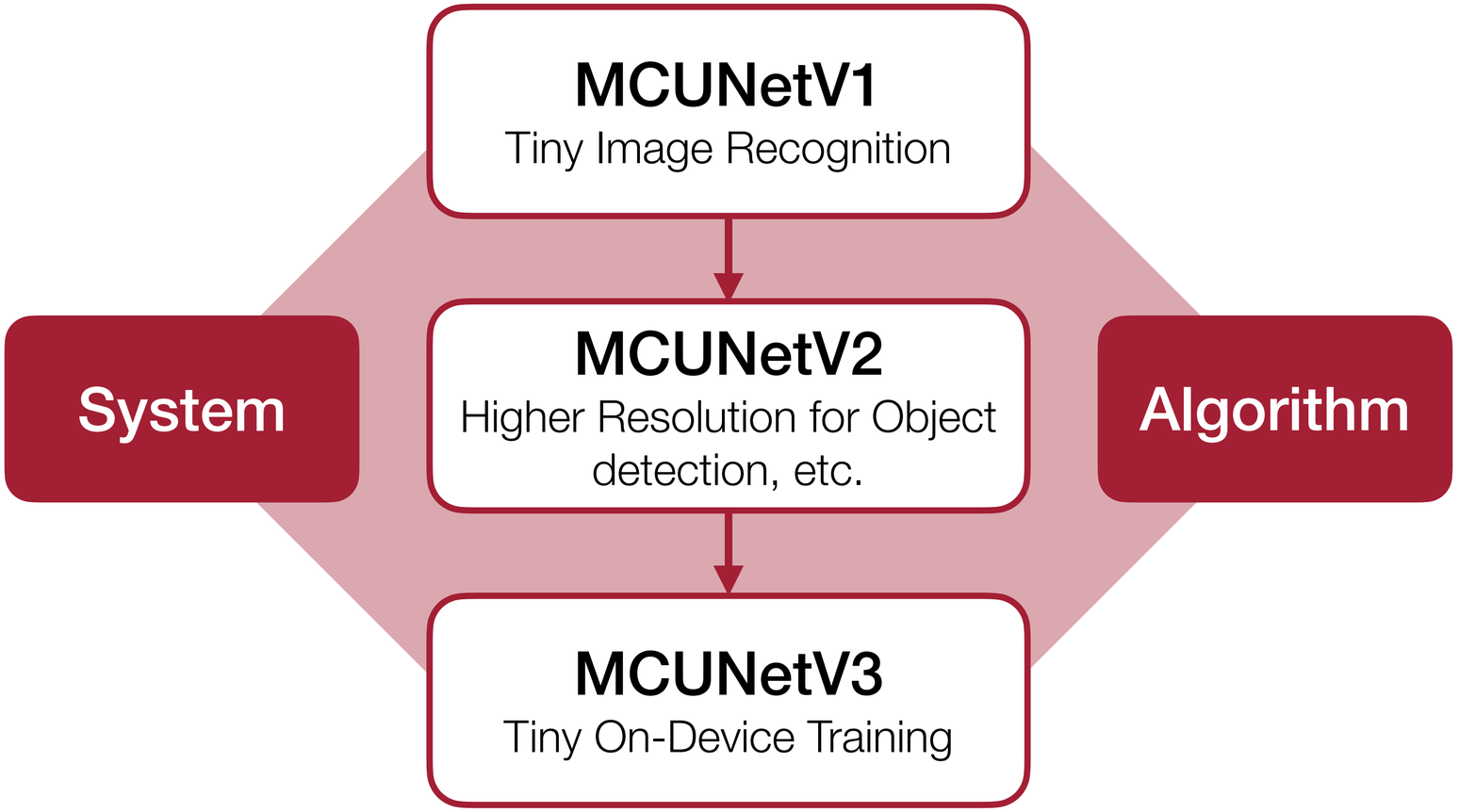
Overview



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



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)**

On-Device Training Under 256KB SRAM [Lin *et al.*, NeurIPS 2022]

Can We Learn on the Edge?

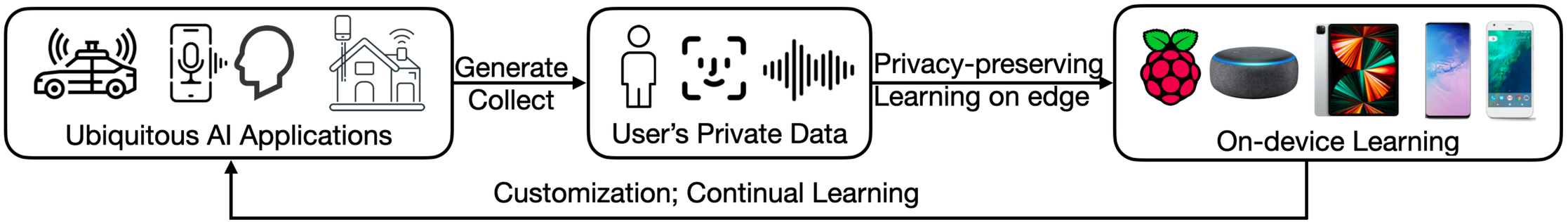
From tinyML inference to training

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

Can We Learn on the Edge?

From tinyML inference to training

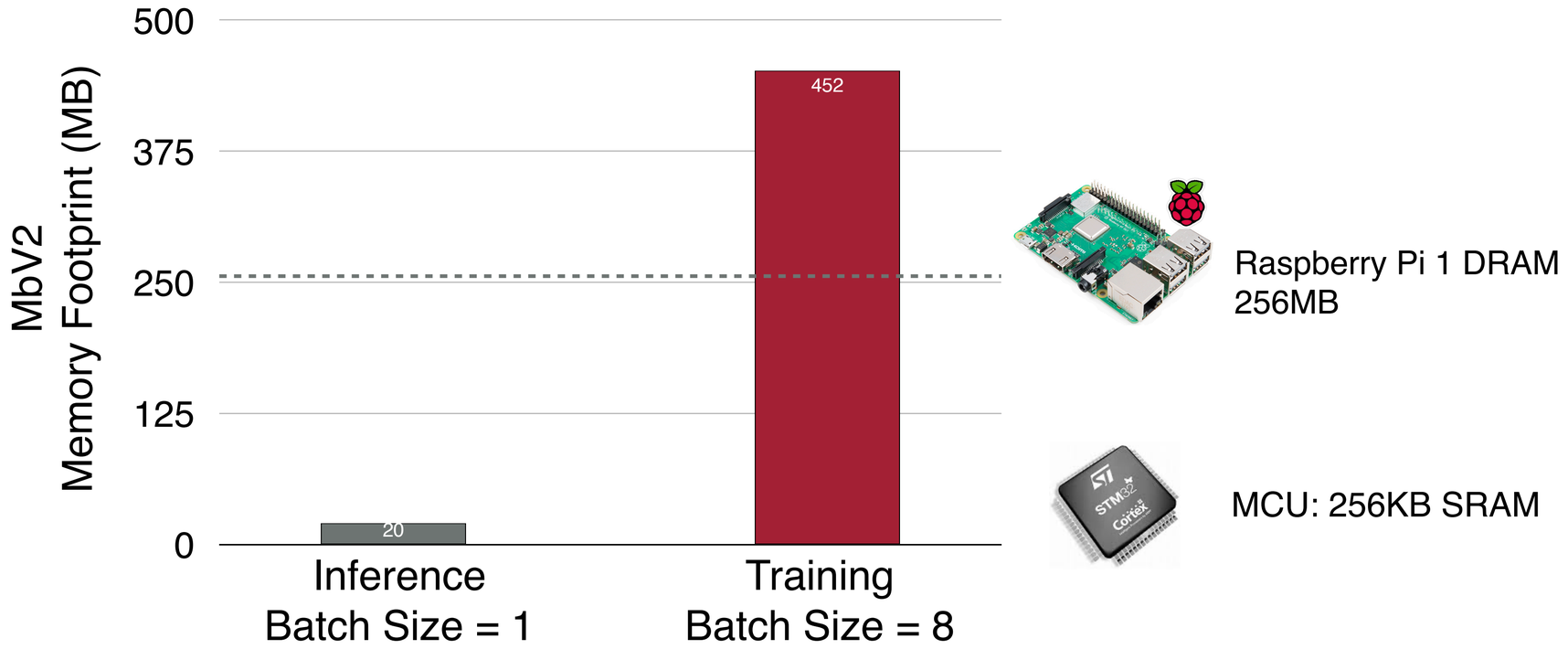
A **virtuous cycle**:



- 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.



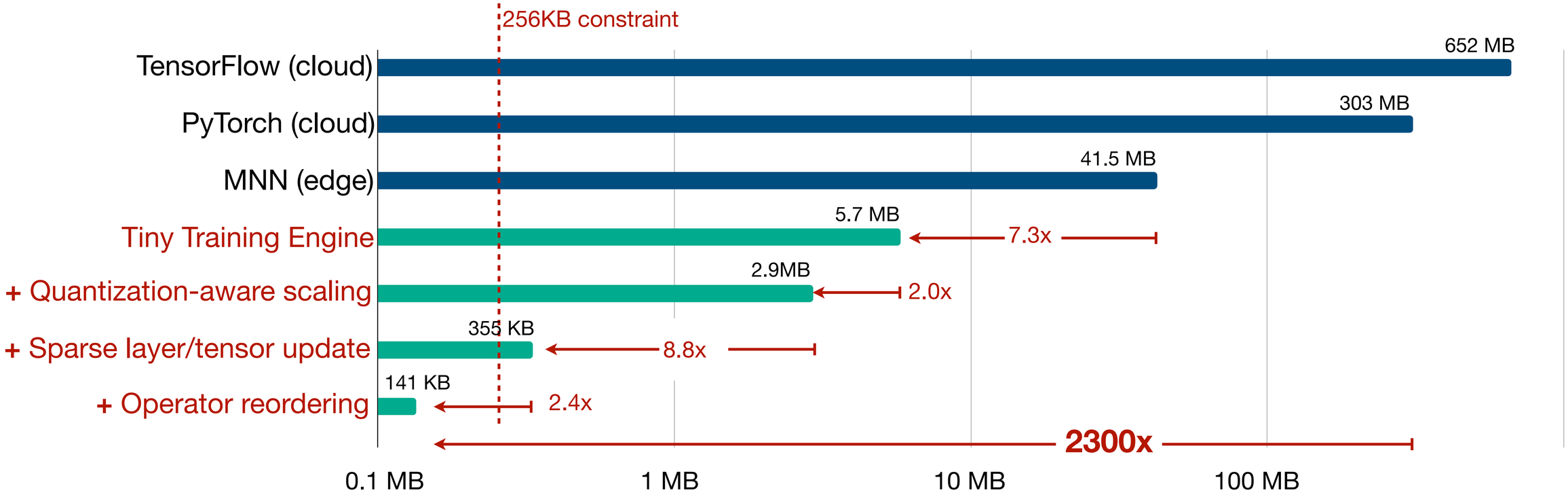
TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai *et al.*, NeurIPS 2020]

On-Device Training Under 256KB Memory

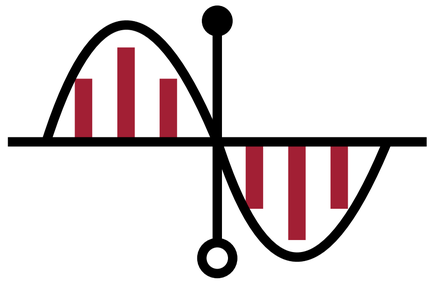
- **Training** is more expensive than **inference** due to back-propagation, making it hard to fit IoT devices (e.g., MCU only has 256KB SRAM).

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).

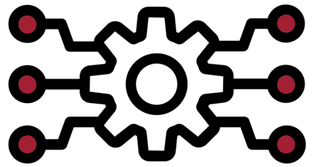


On-Device Training Under 256KB Memory



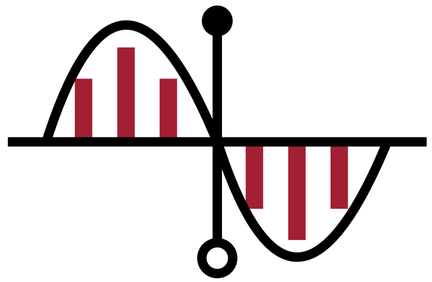
1. Quantization-aware scaling

2. Sparse layer/tensor update



3. Tiny Training Engine

On-Device Training Under 256KB Memory



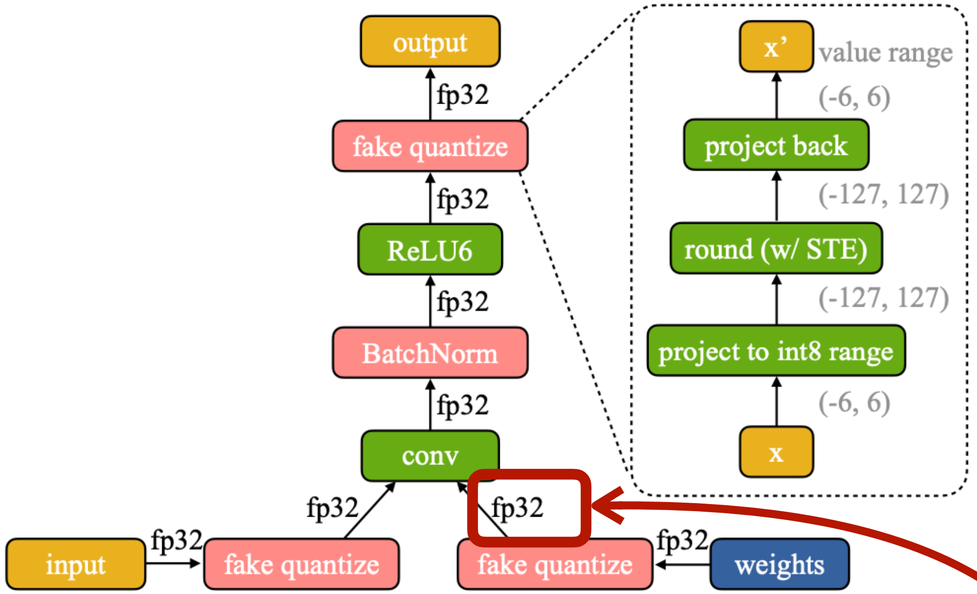
1. Quantization-aware scaling

2. Sparse layer/tensor update

3. Tiny Training Engine

1. Quantization-Aware Scaling (QAS)

Real quantized graphs save memory, but are hard to quantize

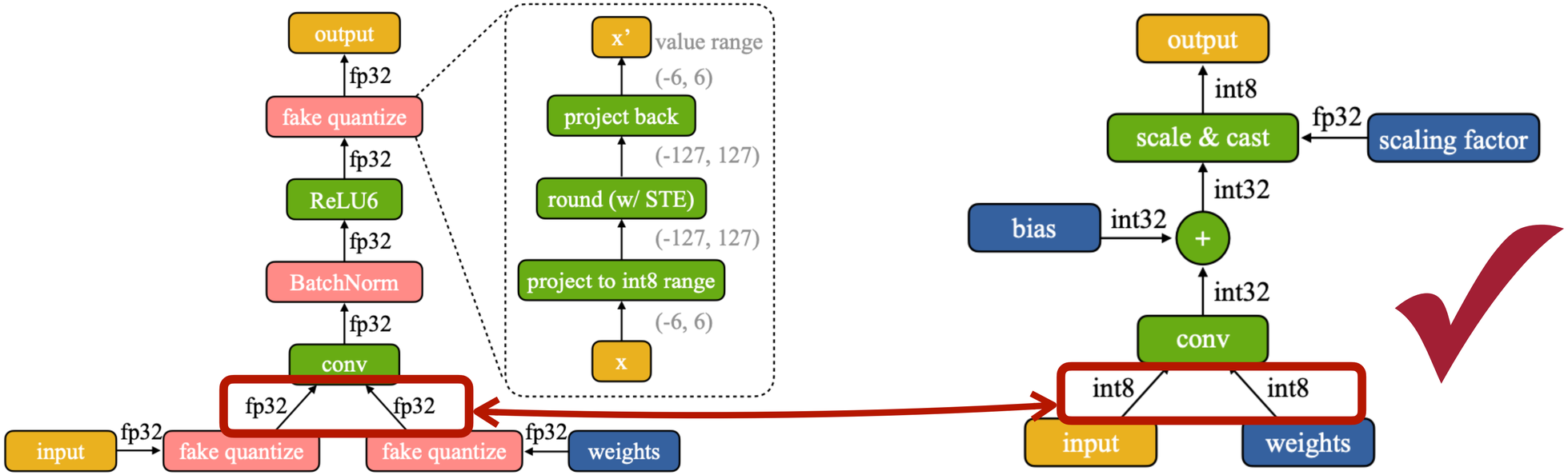


(a) Fake Quantization
(quantization aware training)

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



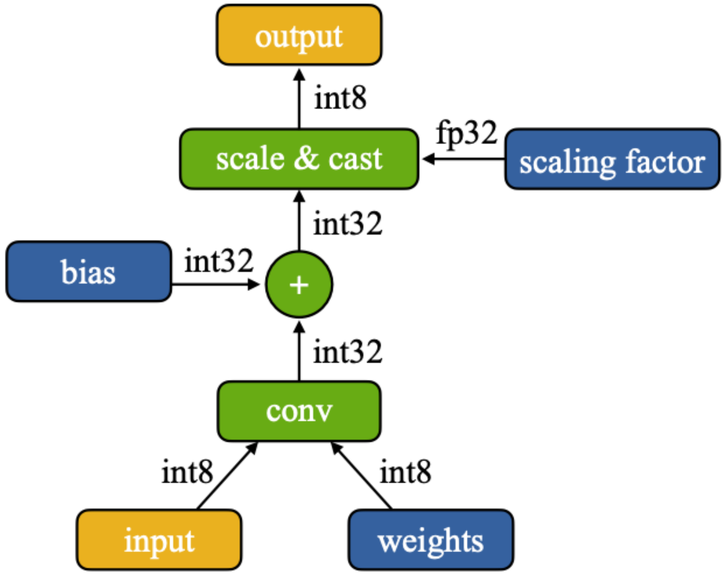
(a) Fake Quantization
(quantization aware training)

(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

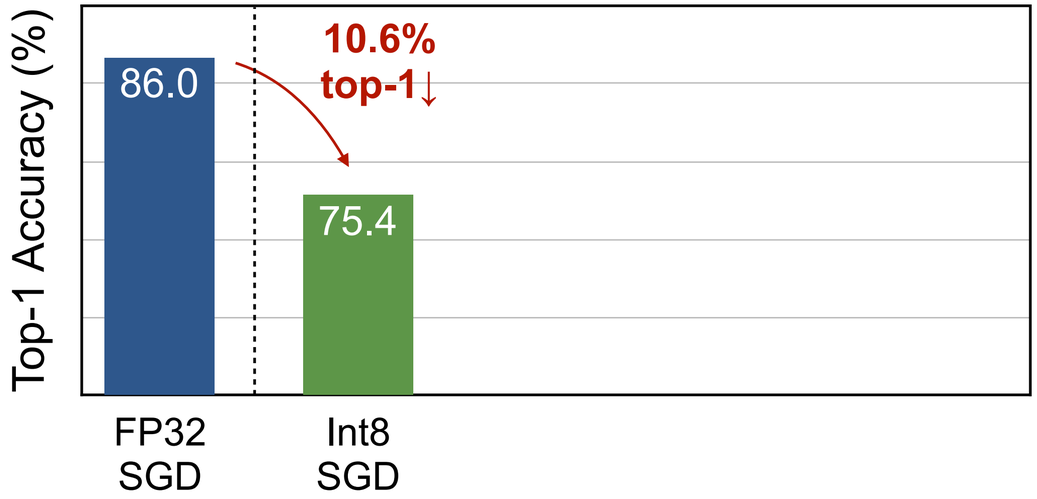


(a) Real Quantization

Difficult optimize:

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

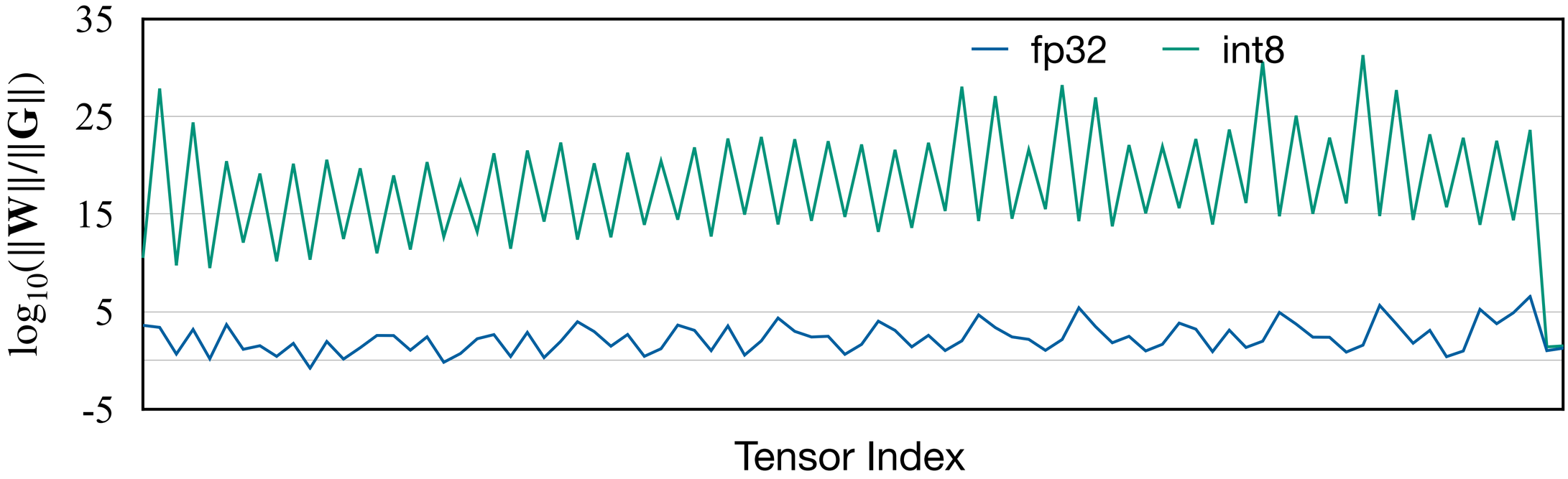
Performance Comparison (average on 10 datasets)



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

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

Per Channel scaling

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

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

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

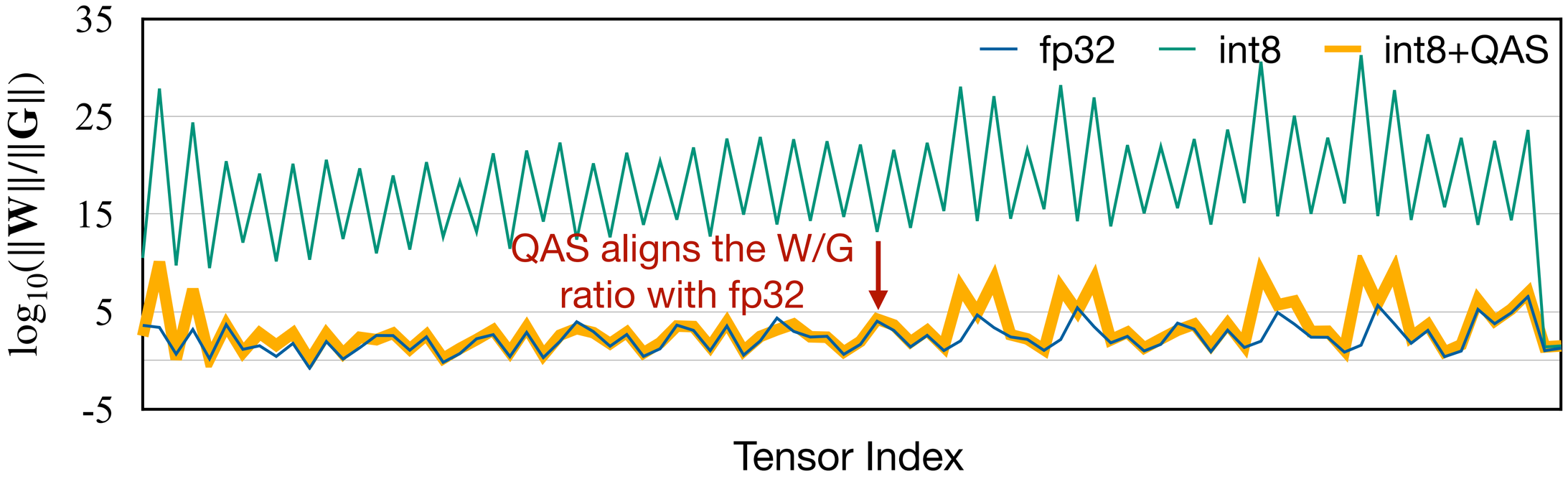
Thus, re-scale the gradients

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

1. Quantization-Aware Scaling (QAS)

QAS addresses the optimization difficulty of quantized graphs

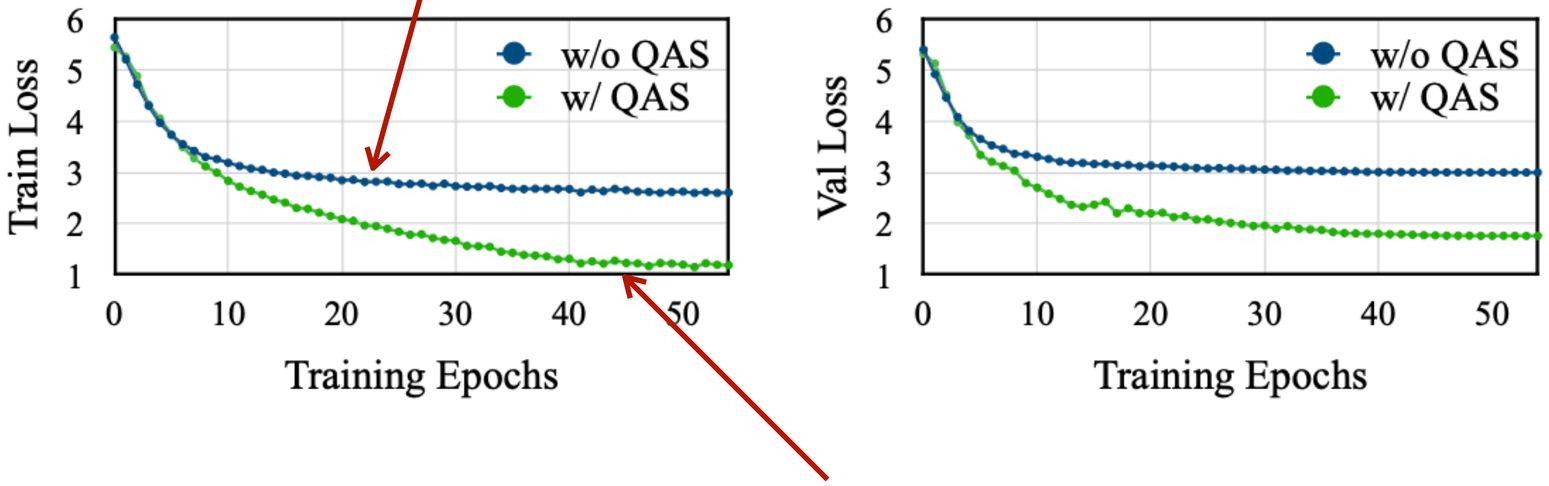
$$\tilde{\mathbf{G}}_{\bar{\mathbf{W}}} = \mathbf{G}_{\bar{\mathbf{W}}} \cdot s_{\mathbf{W}}^{-2}, \quad \tilde{\mathbf{G}}_{\bar{\mathbf{b}}} = \mathbf{G}_{\bar{\mathbf{b}}} \cdot s_{\mathbf{W}}^{-2} \cdot s_{\mathbf{x}}^{-2} = \mathbf{G}_{\bar{\mathbf{b}}} \cdot s^{-2}$$



1. Quantization-Aware Scaling (QAS)

QAS addresses the optimization difficulty of quantized graphs

Without QAS, **poor** convergence

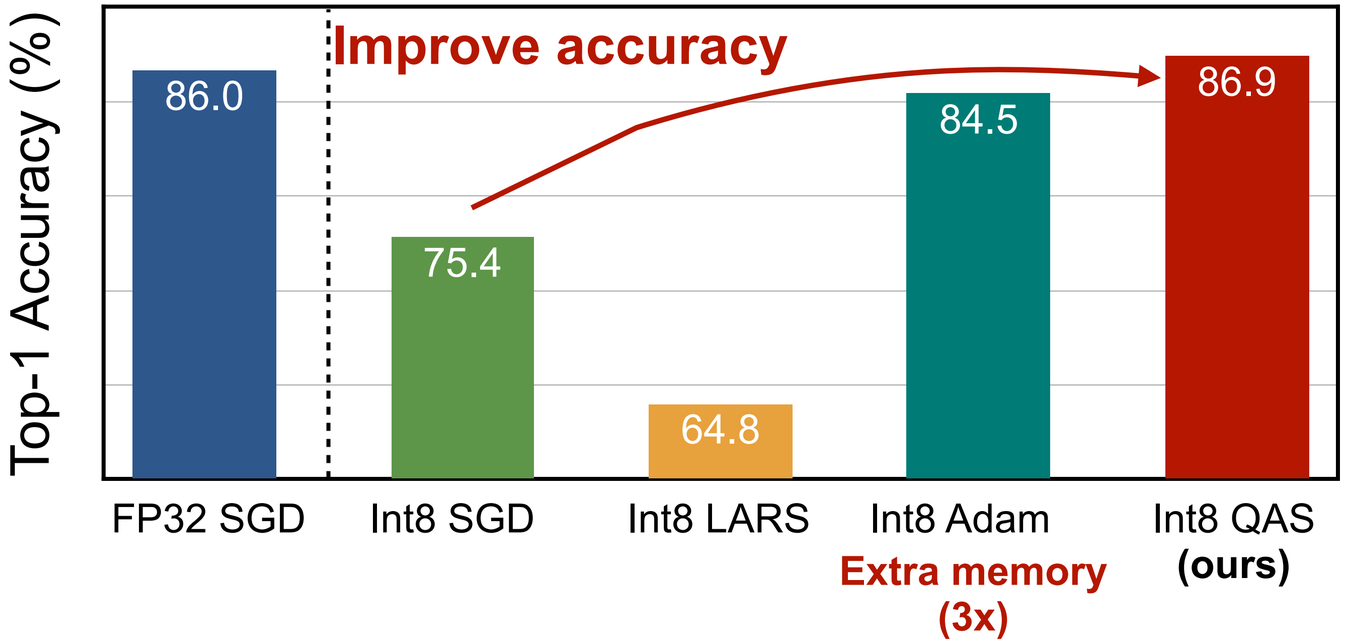


With QAS, **better** convergence

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**

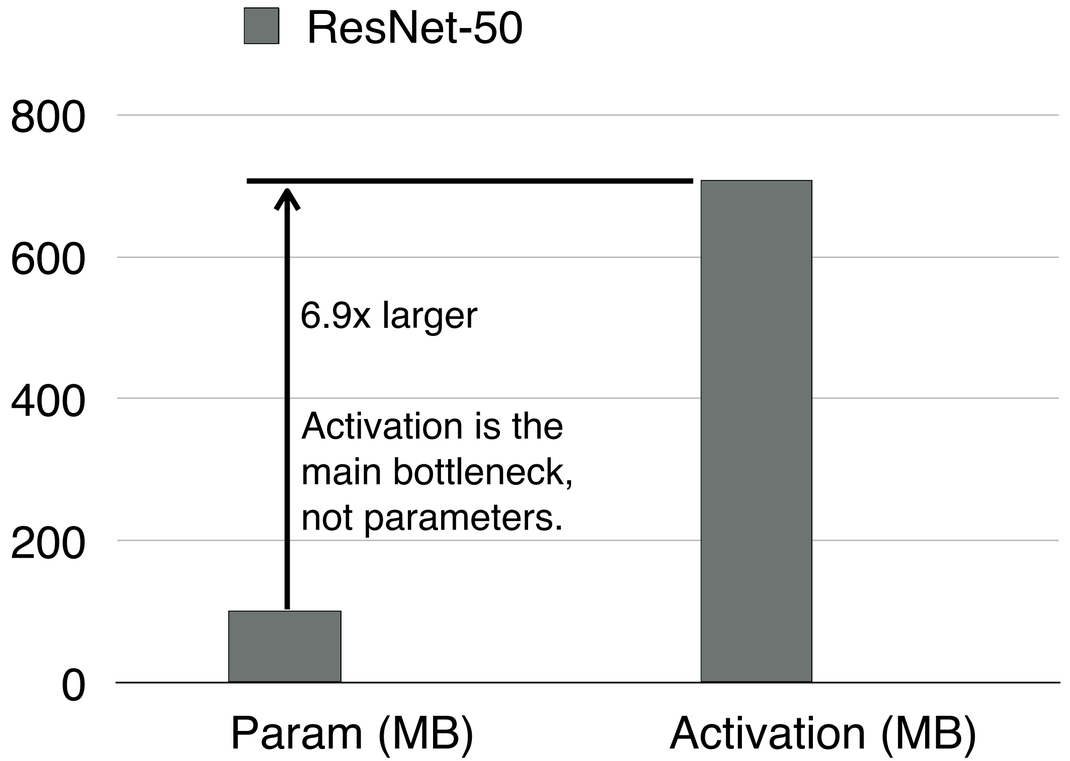
Forward: $\mathbf{a}_{i+1} = \mathbf{a}_i \mathbf{W}_i + \mathbf{b}_i$

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]

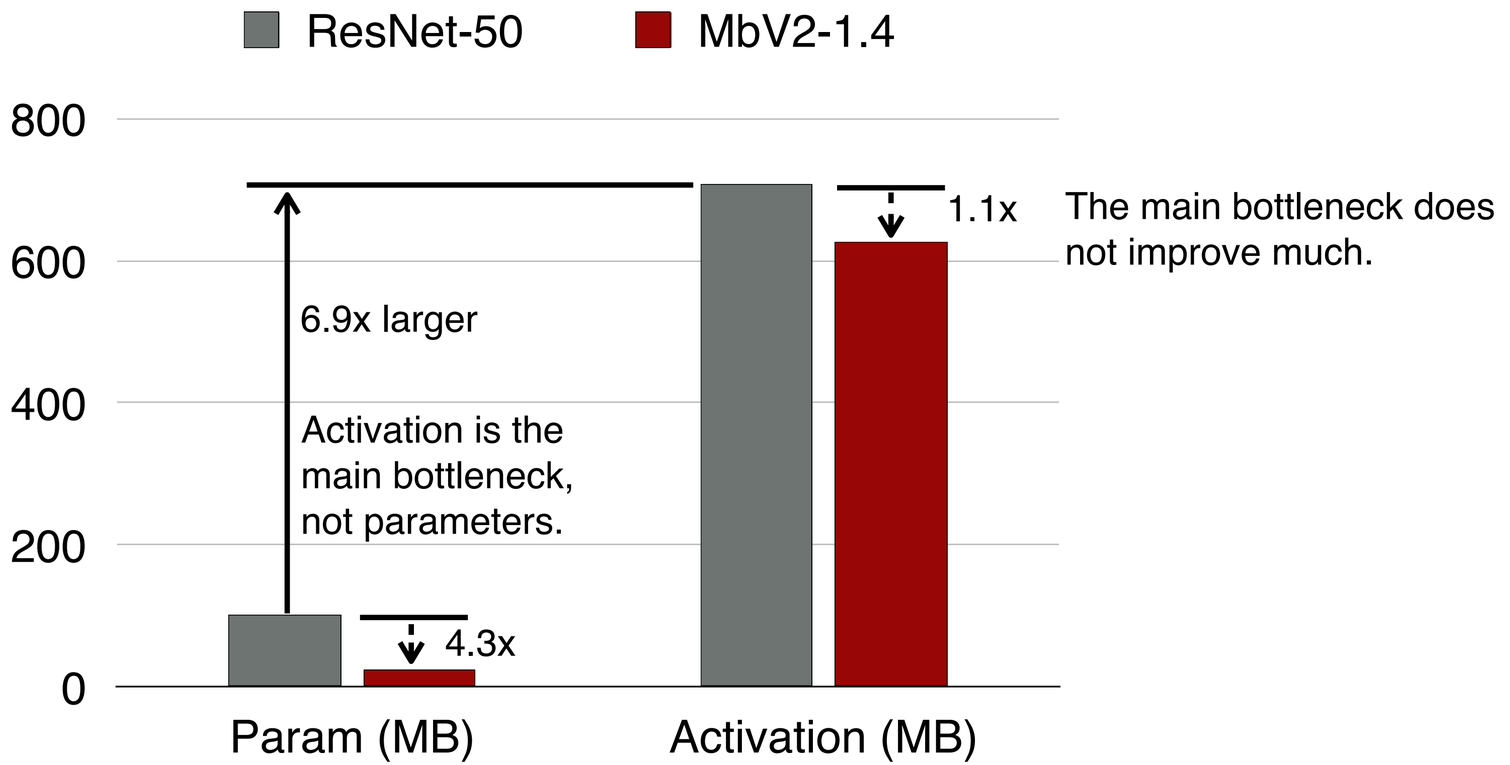
Training Memory is the Key Bottleneck



- 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]

Training Memory is the Key Bottleneck

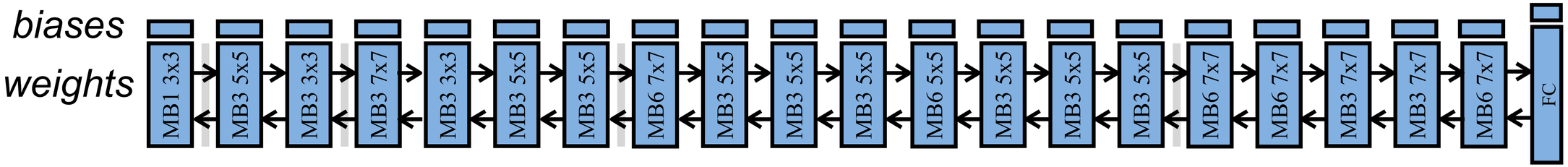


- Activation is the main bottleneck for on-device learning, not parameters.
- Previous methods focus on reducing the number of parameters or FLOPs, while the main bottleneck does not improve much.

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

2. Sparse Layer/Tensor Update

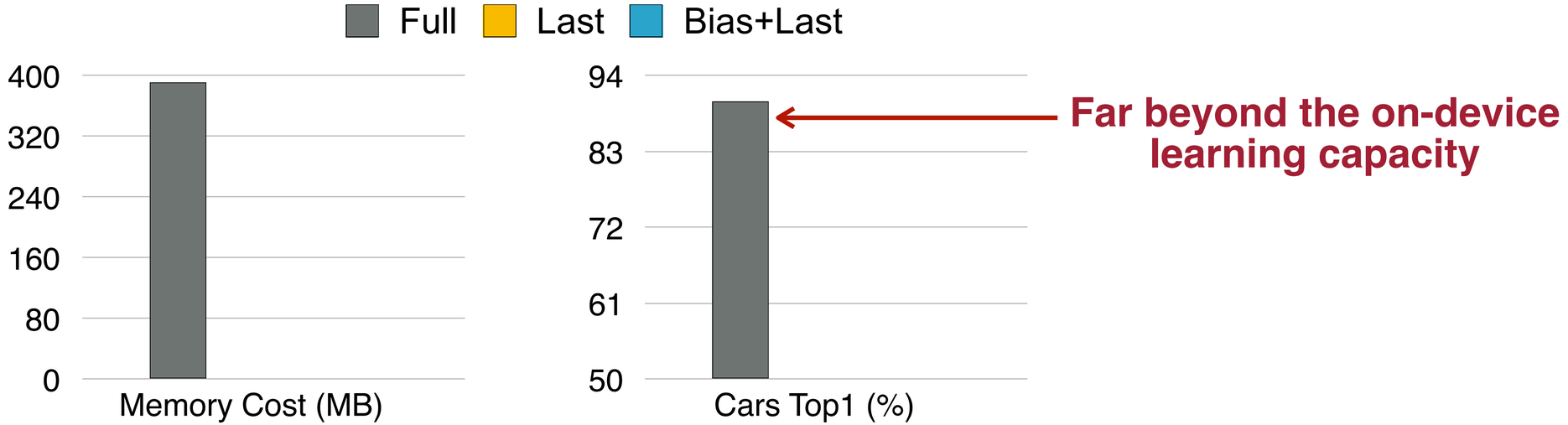
Full update



Model: ProxylessNAS-Mobile

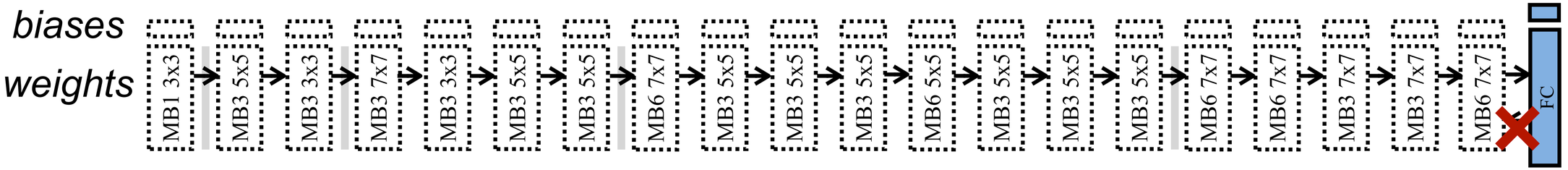
Updating the whole model is **too expensive**:

- Need to save all intermediate activation (quite large)
- Need to store the updated weights in SRAM (Flash is read-only)



2. Sparse Layer/Tensor Update

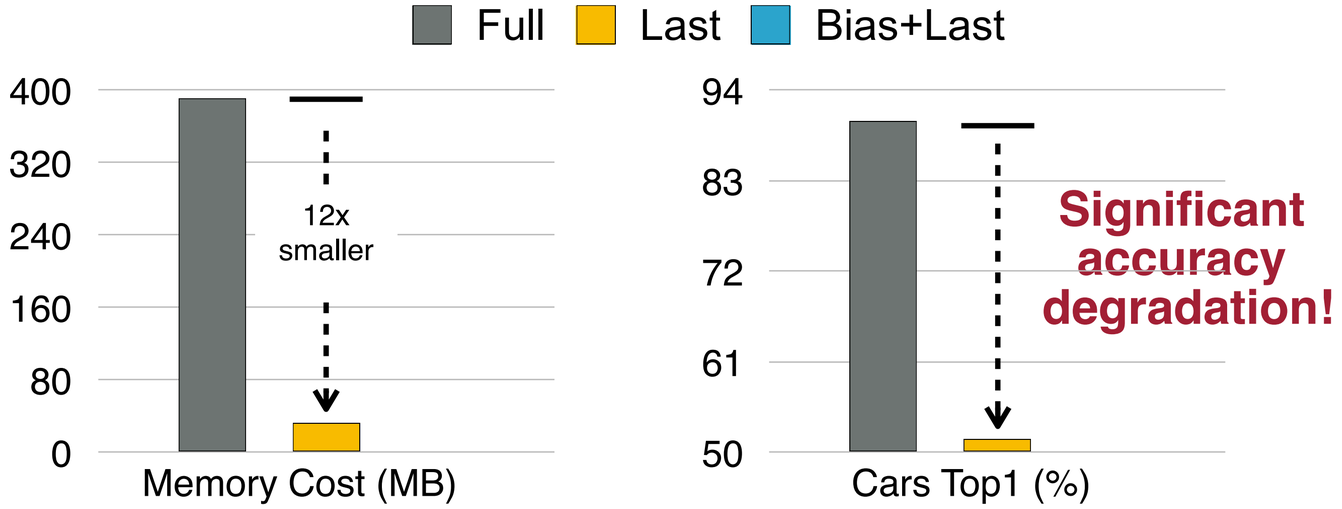
Last layer update



Model: ProxylessNAS-Mobile

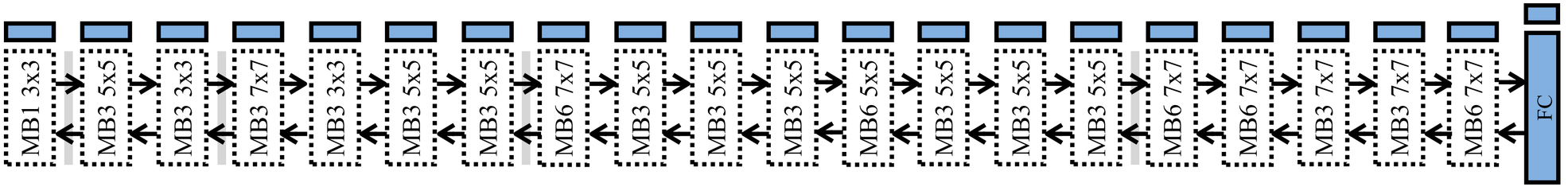
Updating only the last cheap

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



2. Sparse Layer/Tensor Update

Bias-only update



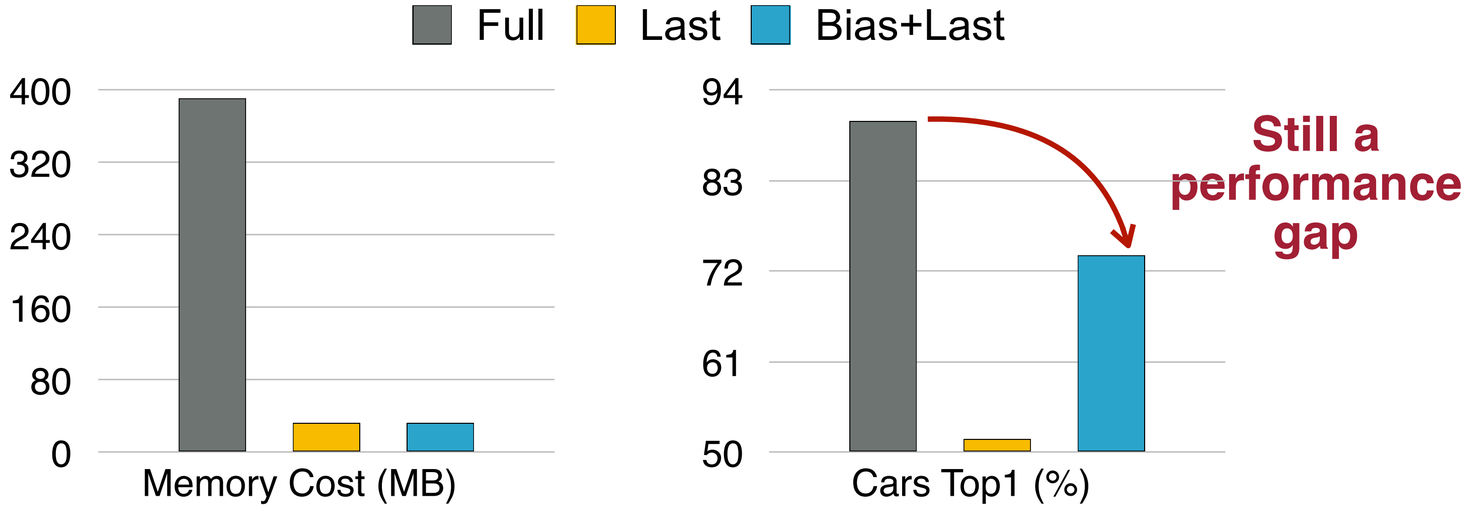
Model: ProxylessNAS-Mobile

Updating the only the bias part

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

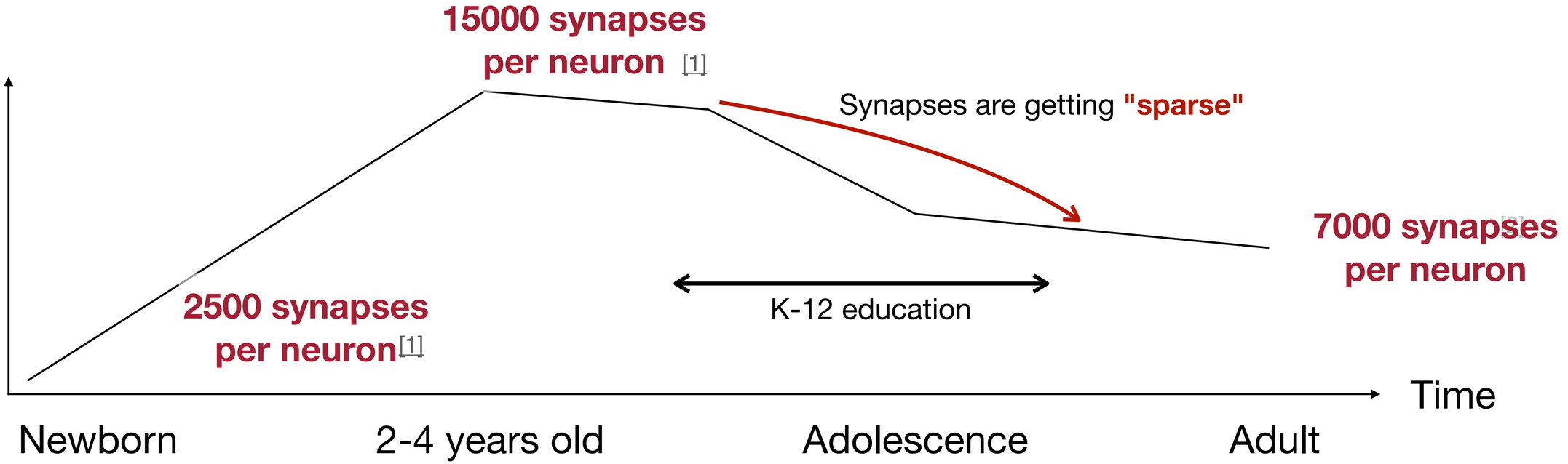
Forward: $\mathbf{a}_{i+1} = \mathbf{a}_i \mathbf{W}_i + \mathbf{b}_i$

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



2. Sparse Layer/Tensor Update

Updated synapses are sparse

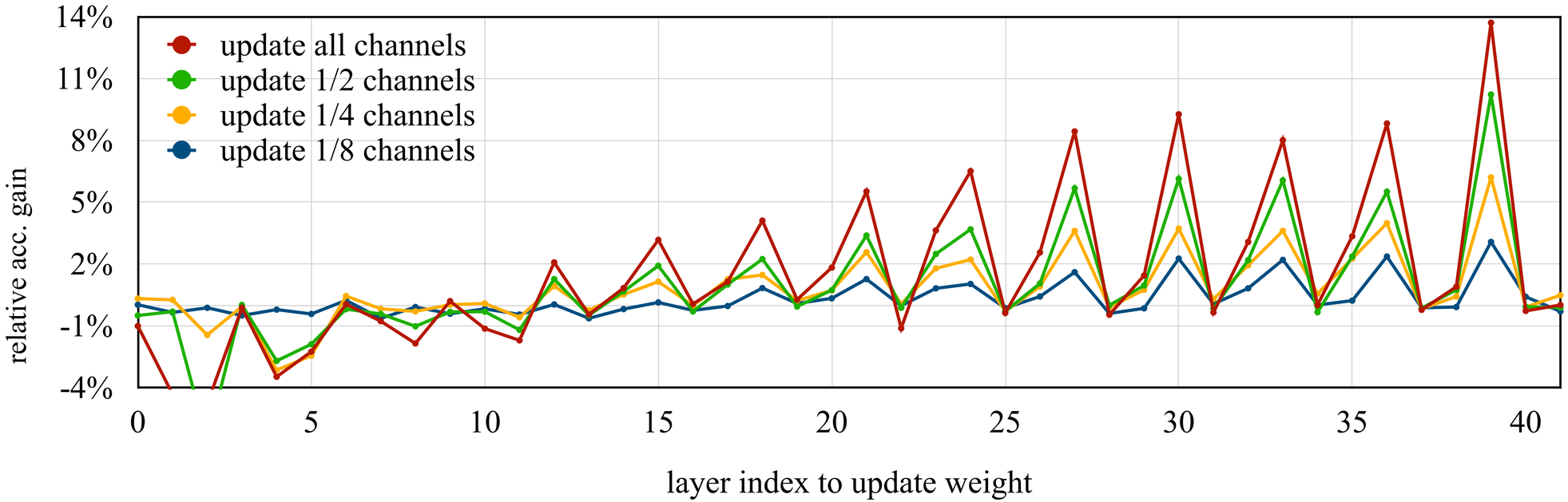


Do We Have Brain to Spare? [Drachman DA, Neurology 2004]
Peter Huttenlocher (1931–2013) [Walsh, C. A., Nature 2013]

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

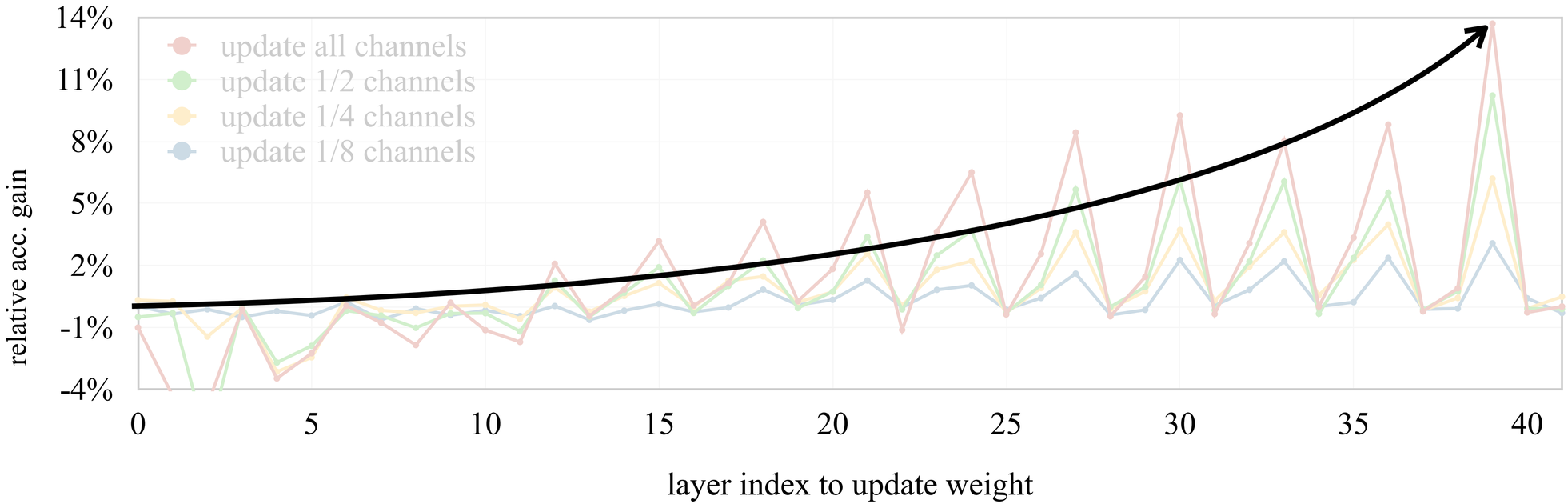
2. Sparse Layer/Tensor Update

Some layers are more important than others



2. Sparse Layer/Tensor Update

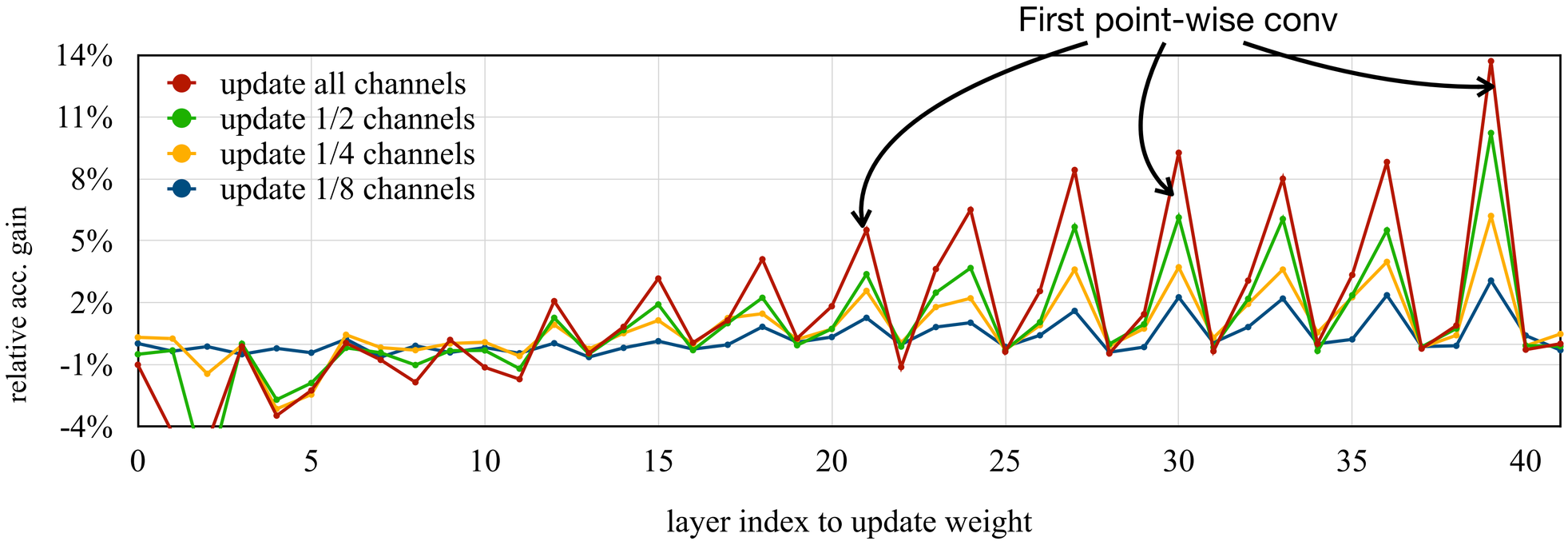
Some layers are more important than others



1. Later layers contribute more to the accuracy.

2. Sparse Layer/Tensor Update

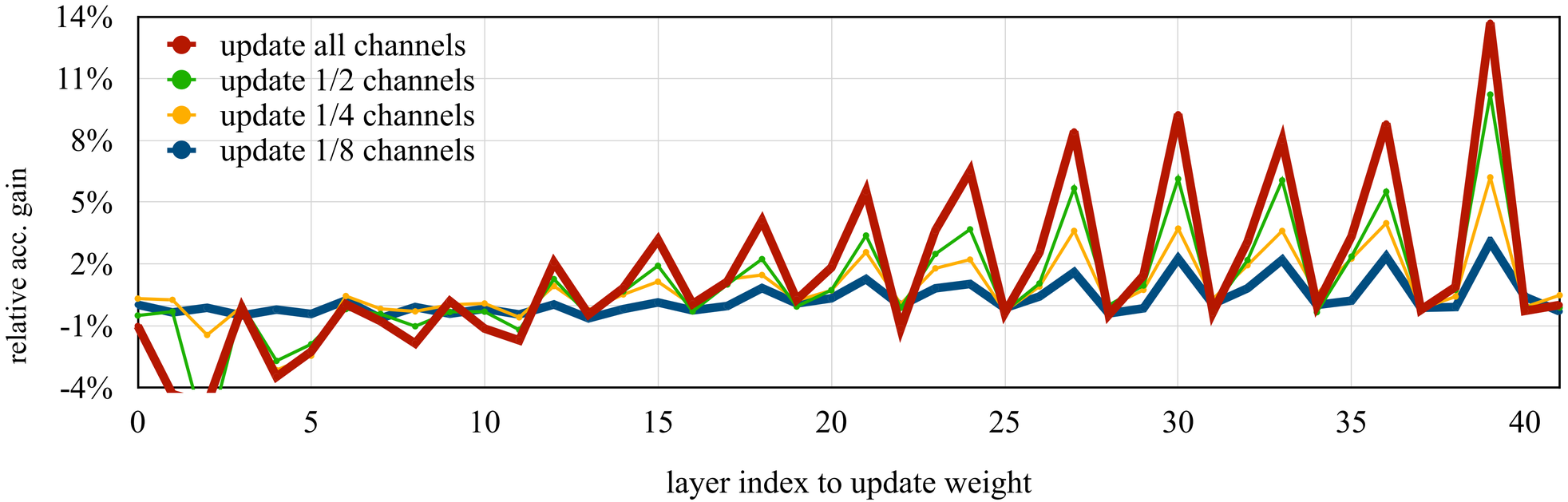
Some layers are more important than others



1. Later layers contribute more to the accuracy.
2. First point-wise conv are more important to accuracy.

2. Sparse Layer/Tensor Update

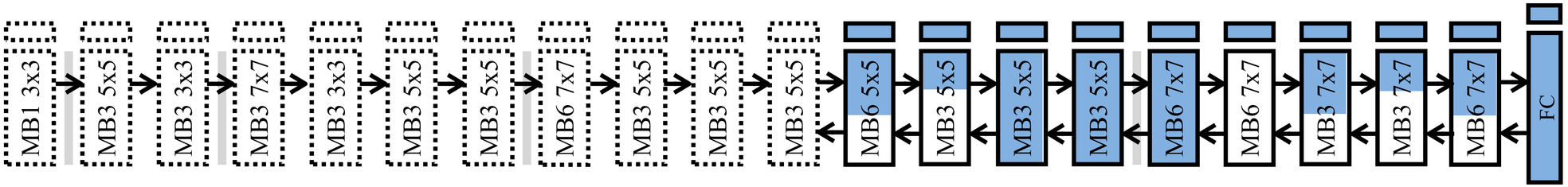
Some layers are more important than others



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

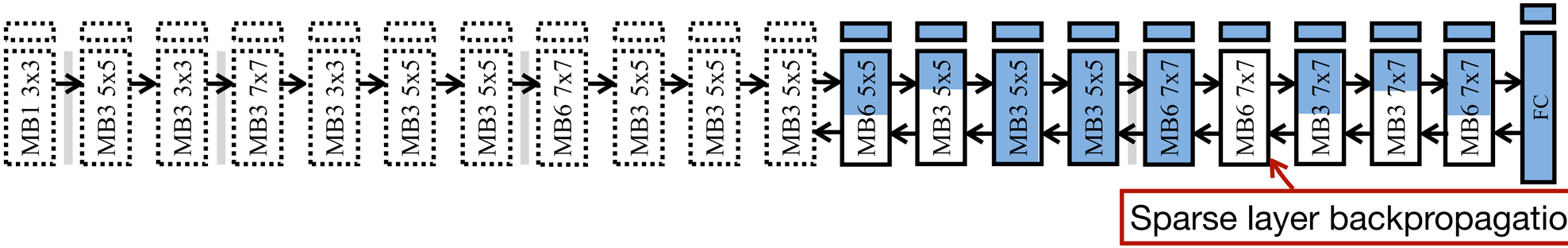
Sparse Layer/Tensor Update



Model: ProxylessNAS-Mobile

2. Sparse Layer/Tensor Update

Sparse Layer/Tensor Update

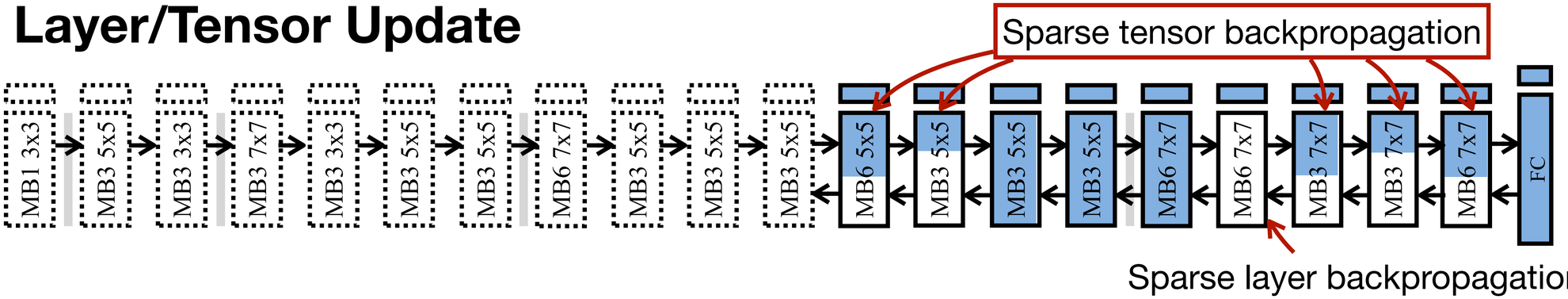


Model: ProxylessNAS-Mobile

- **Sparse layer update**: no need to store activation

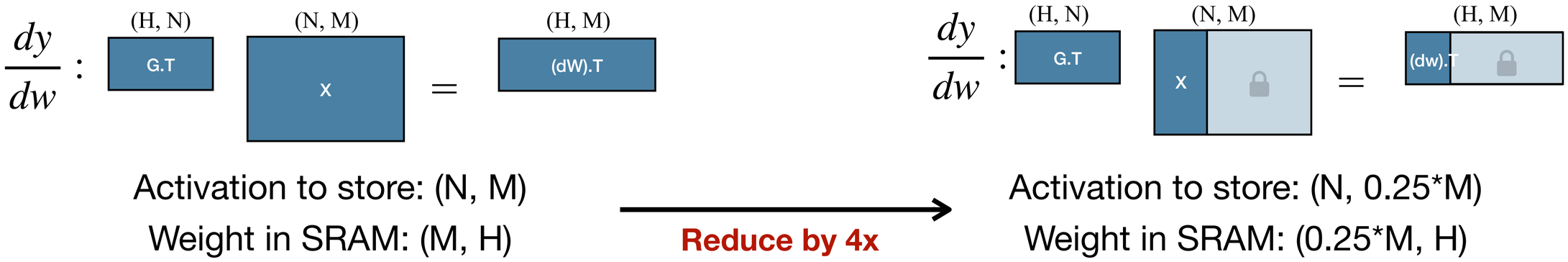
2. Sparse Layer/Tensor Update

Sparse Layer/Tensor Update



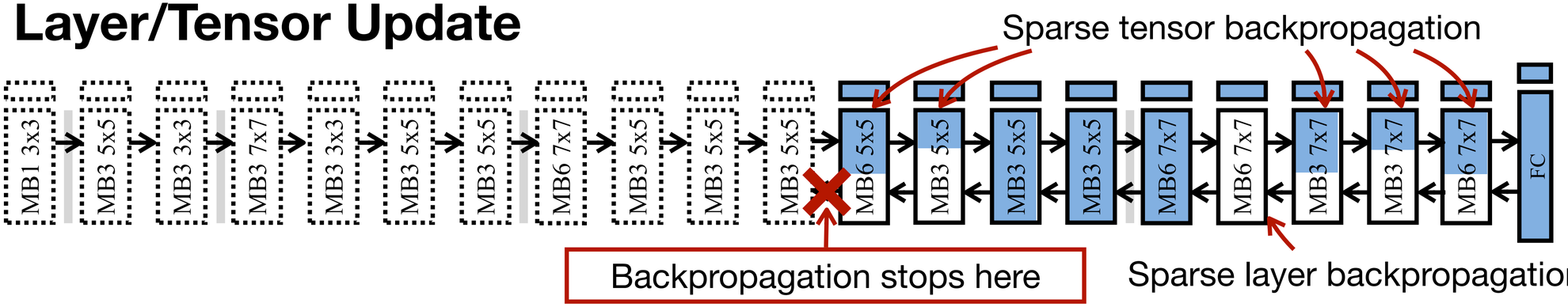
Model: ProxylessNAS-Mobile

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



2. Sparse Layer/Tensor Update

Sparse Layer/Tensor Update

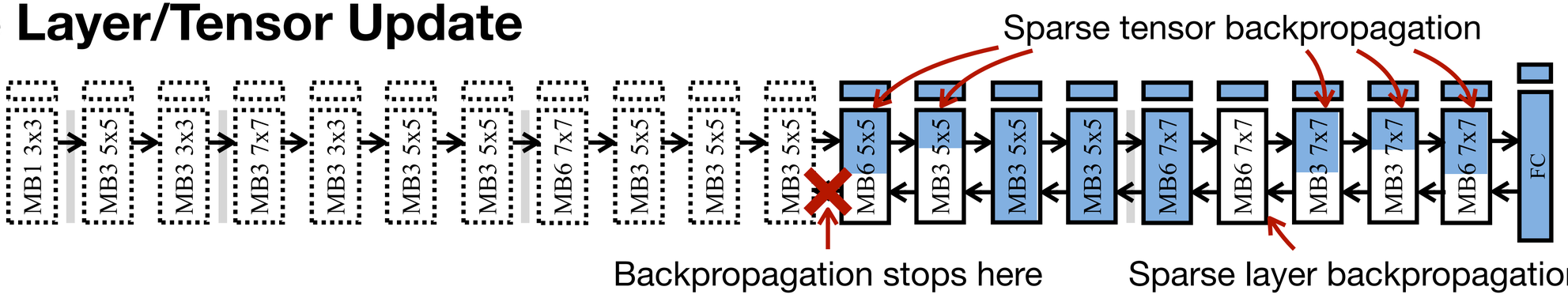


Model: ProxylessNAS-Mobile

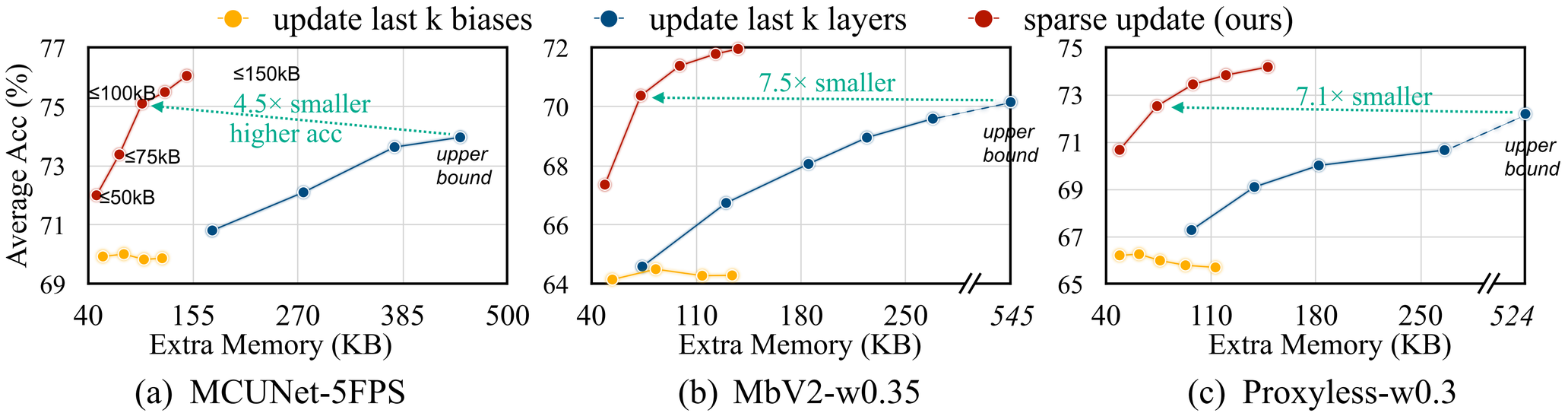
- **Sparse layer update**: no need to store activation
- **Sparse tensor update**: only store a subset of the activations.
- **Sparse update:** no need to back propagate the early layers

2. Sparse Layer/Tensor Update

Sparse Layer/Tensor Update

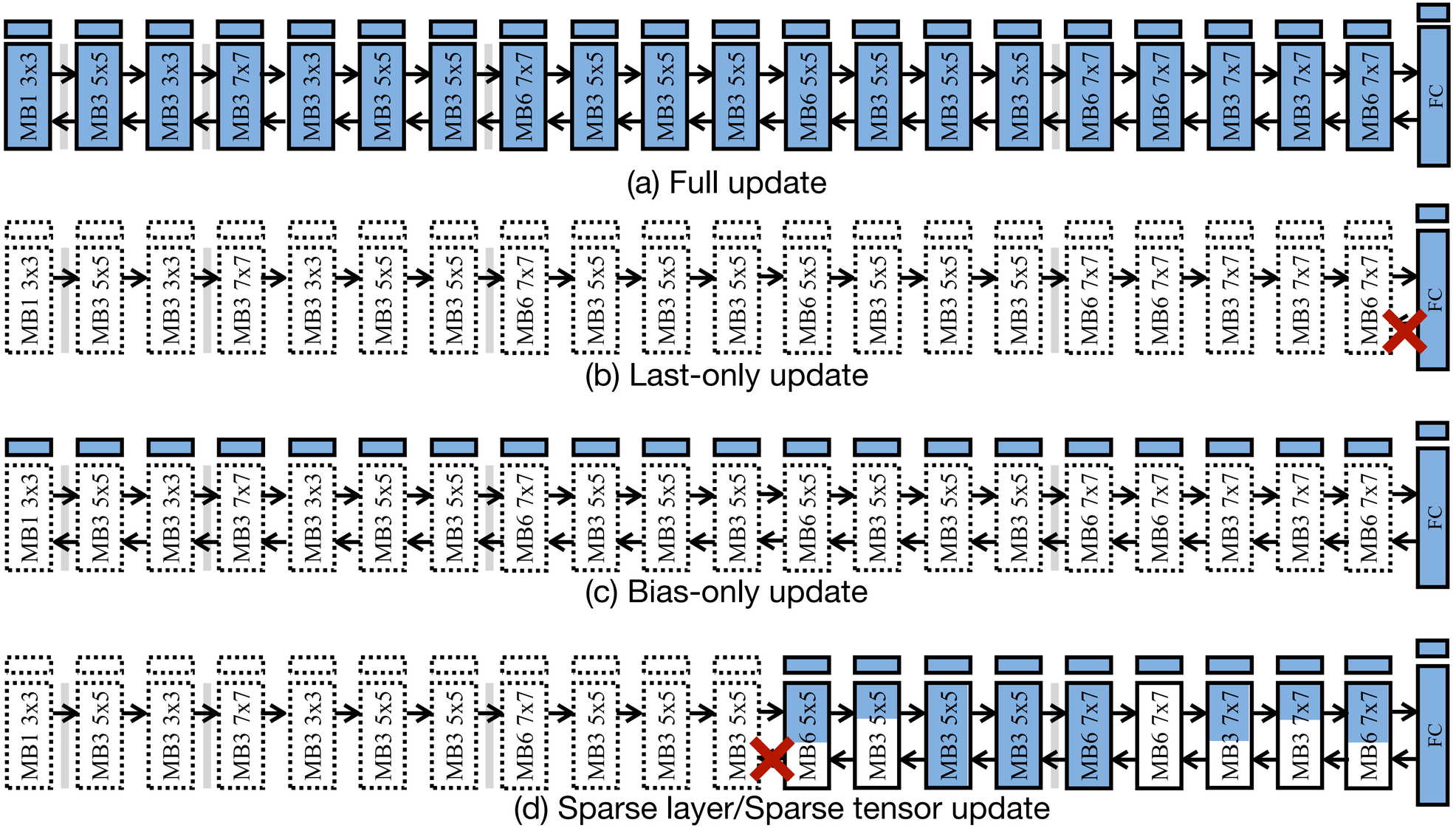


Model: ProxylessNAS-Mobile



(a) MCUNet-5FPS (b) MbV2-w0.35 (c) Proxyless-w0.3

Update Paradigms Comparison



(a) Full update

(b) Last-only update

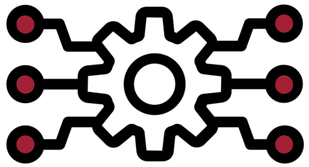
(c) Bias-only update

(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

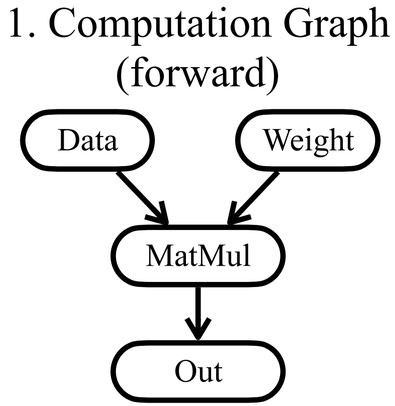
3. Tiny Training Engine (TTE)

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

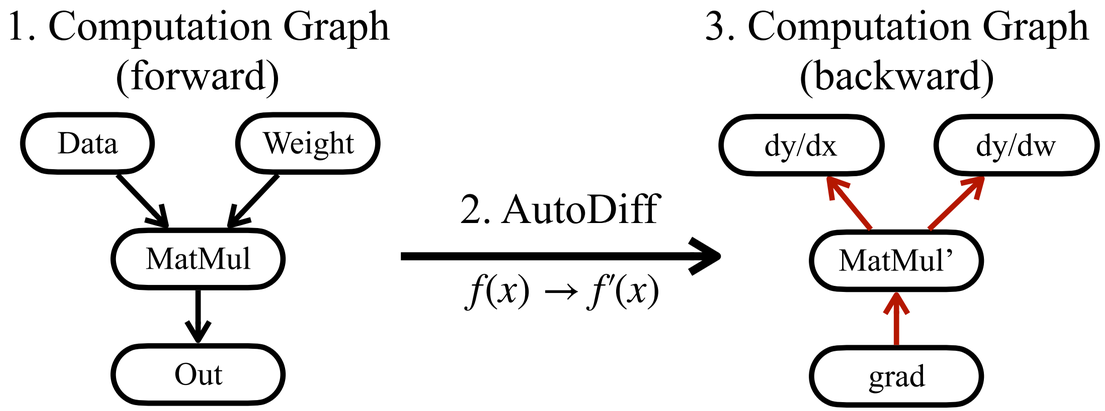
3. Tiny Training Engine (TTE)

Workflow of conventional training engine



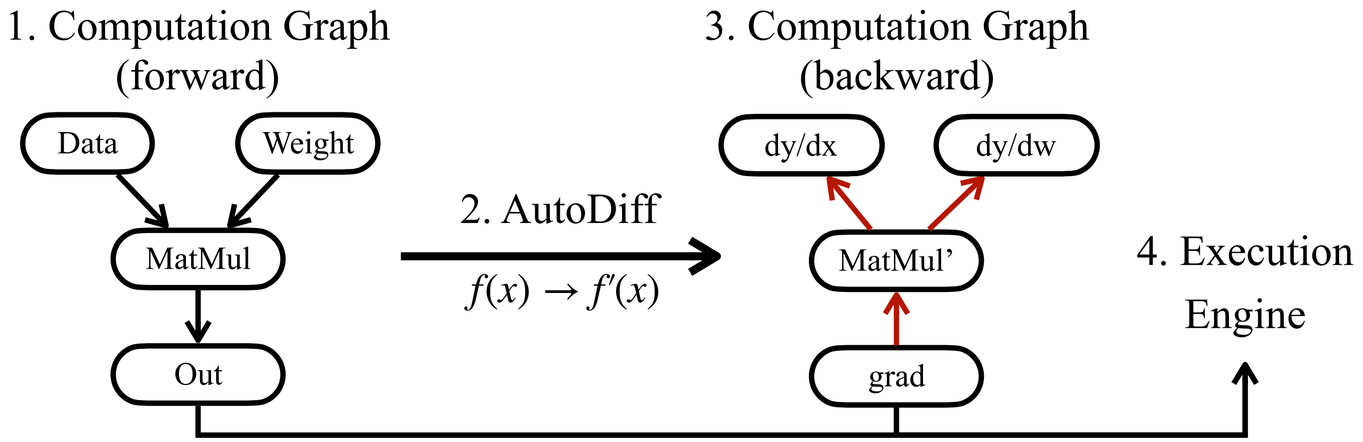
3. Tiny Training Engine (TTE)

Workflow of conventional training engine



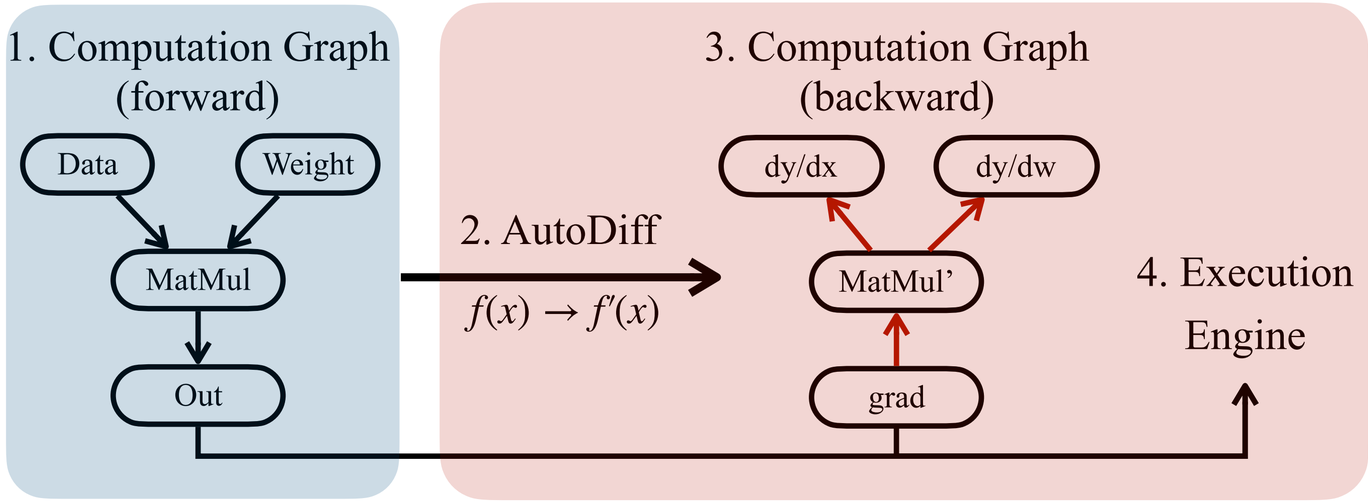
3. Tiny Training Engine (TTE)

Workflow of conventional training engine



3. Tiny Training Engine (TTE)

Workflow of conventional training engine



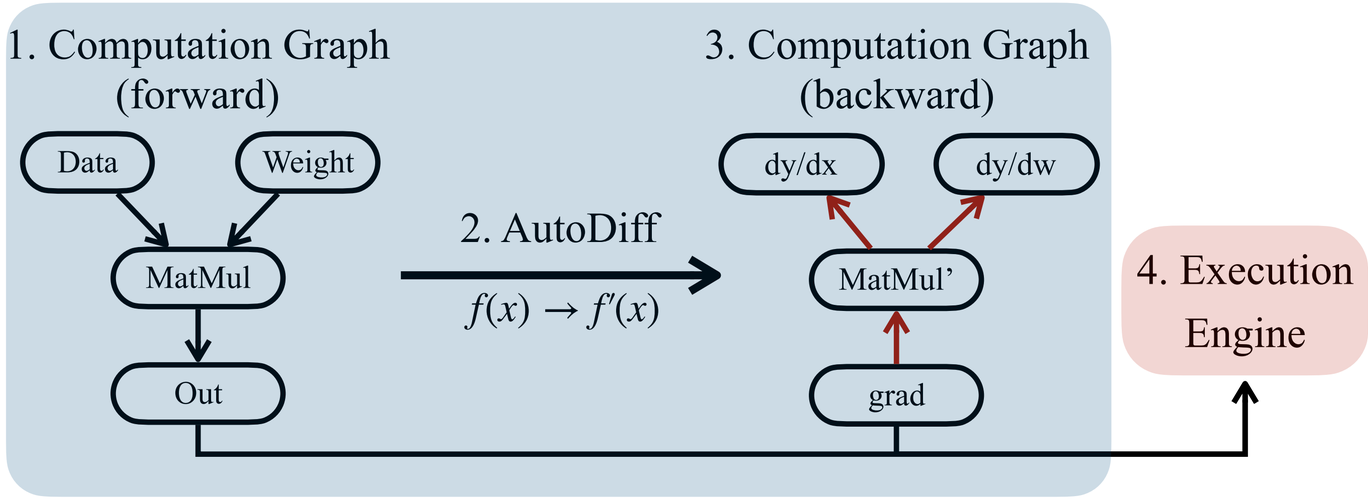
: Compile-Time

: Runtime

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

3. Tiny Training Engine (TTE)

TTE: Move workload from runtime to compile time

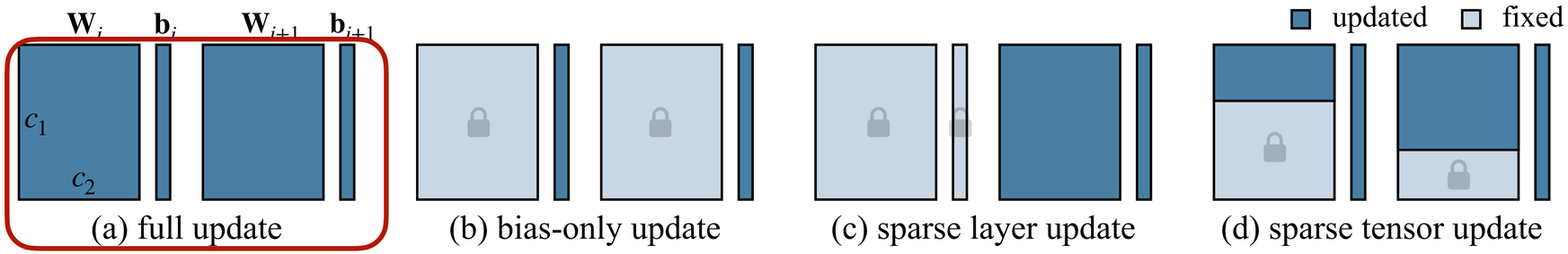


: Compile-Time

: Runtime

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)



Example from a matrix multiplication with full update

Forward

$$y = mul(x, w) + b$$

Backward

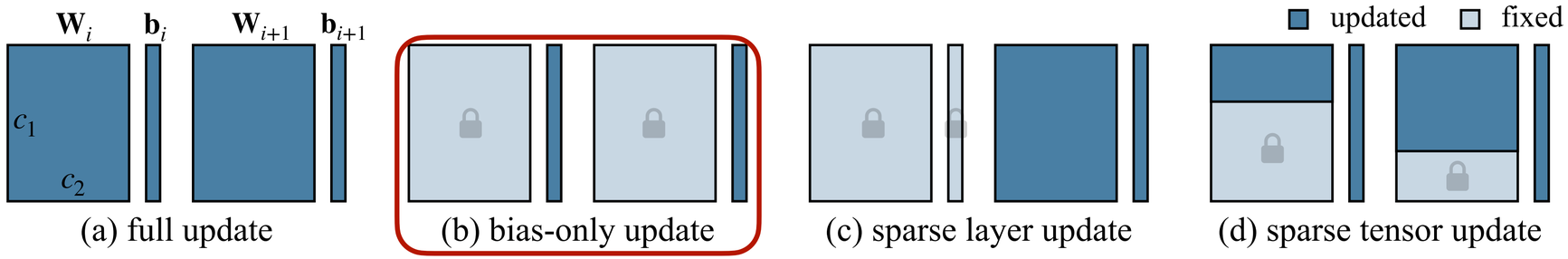
$$dy/dx = mul(G, w)$$

$$dy/dw = mul(G^T, X)$$

$$dy/db = sum(G)$$

```
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)
}
```

3. 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, 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

Forward

$$y = mul(x, w) + b$$

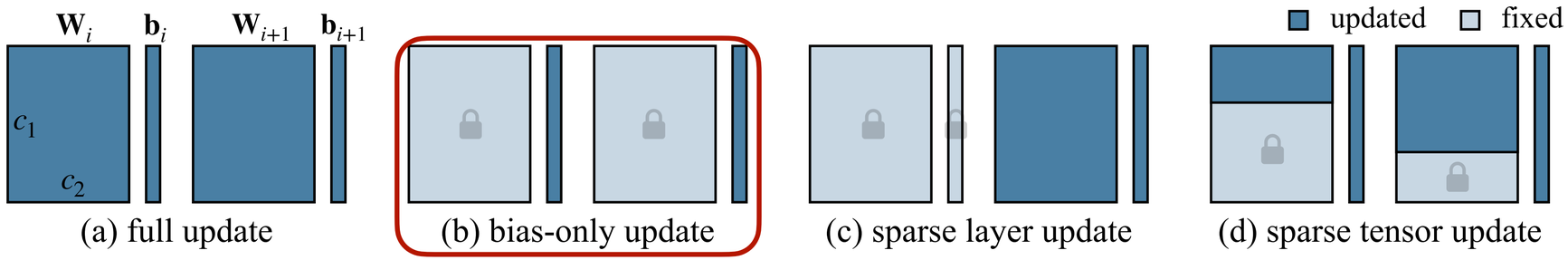
Backward

$$dy/dx = mul(G, w)$$

$$dy/dw = mul(G^T, X)$$

$$dy/db = sum(G)$$

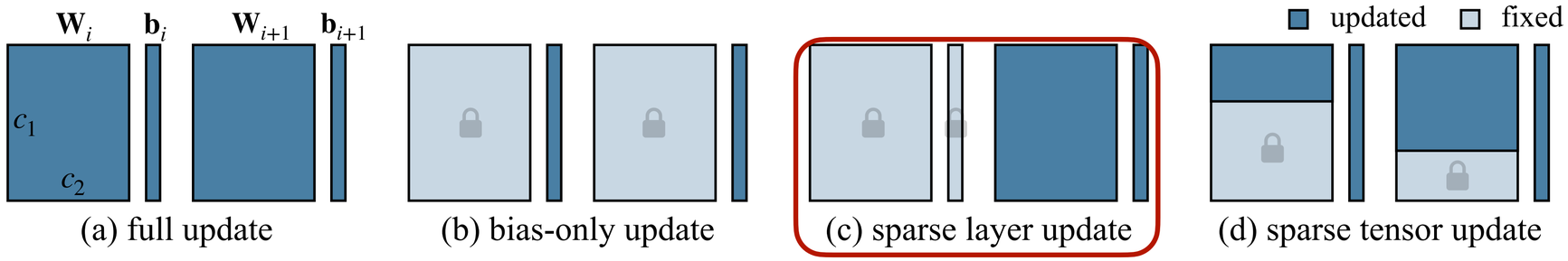
3. Tiny Training Engine (TTE)




```
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.

3. Tiny Training Engine (TTE)



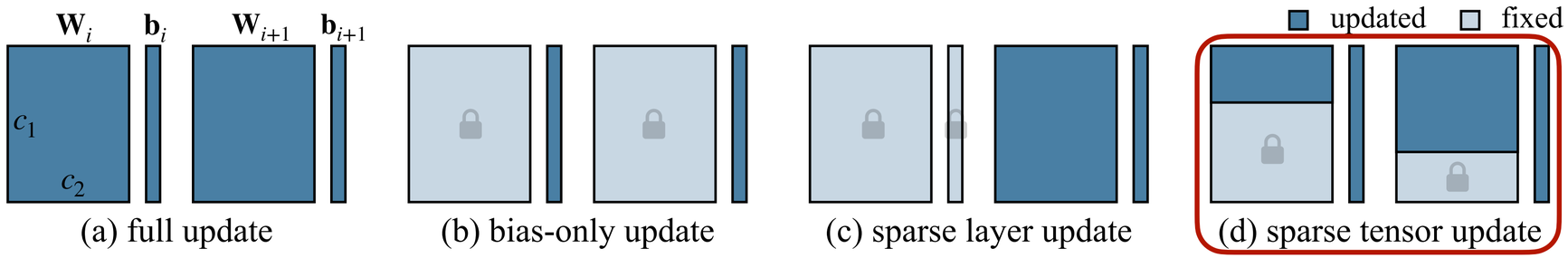
(a) full update (b) bias-only update (c) sparse layer update (d) sparse tensor update

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

Freely annotate **ANY** parameters

TTE will trim the computation accordingly.

3. Tiny Training Engine (TTE)



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

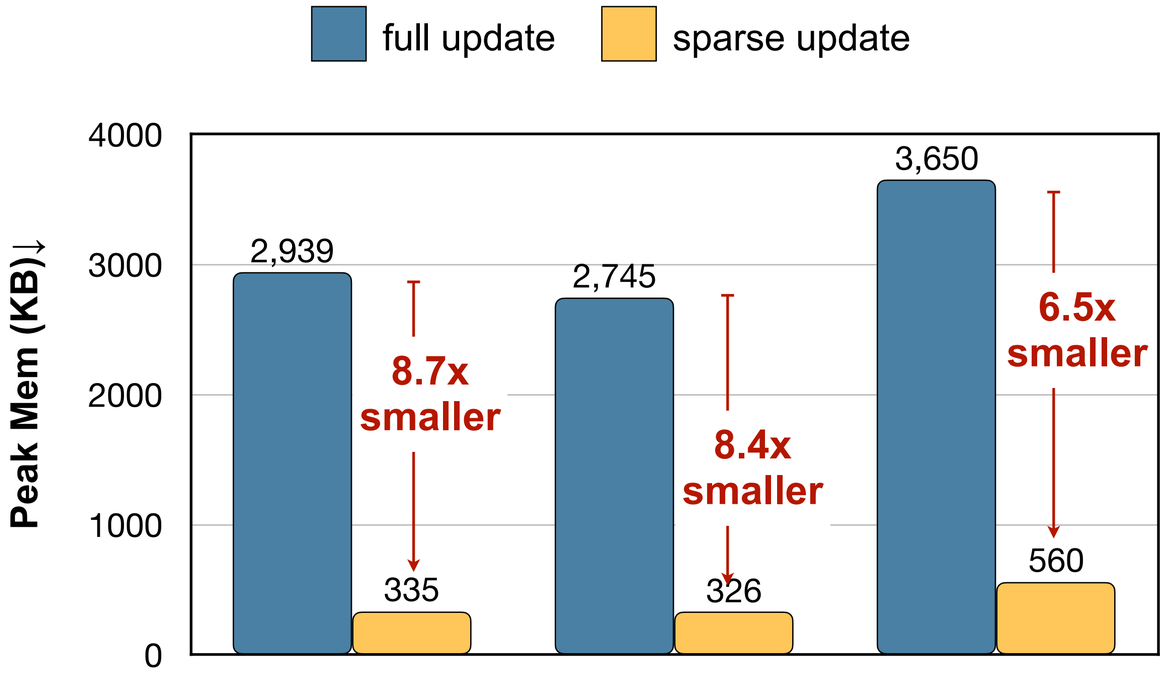
```
fn (%x: Tensor[(10, 10), float32],
    %weight: Tensor[(10, 10), float32],
    %bias: Tensor[(10), float32]),
    %grad: Tensor[(10), float32]),
{
  # forward
  %0 = multiply(%x, %weight);
  %1 = add(%0, %bias);
  # backward
  %3 = multiply(%grad, %weight);
  %4 = transpose(%grad)
  %5 = multiply(%4,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

3. Tiny Training Engine (TTE)

Sparse update results

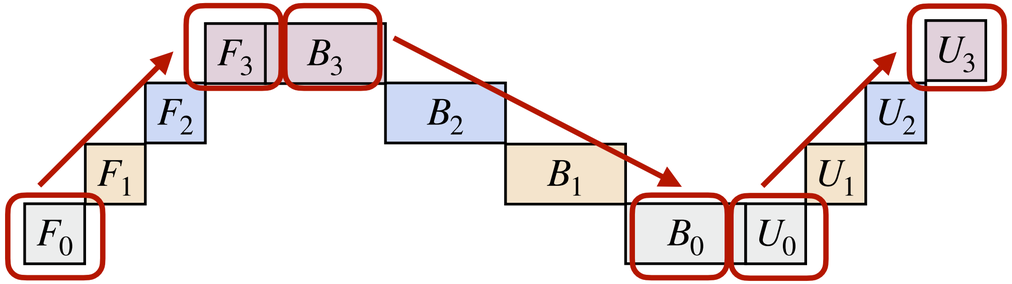


- Tiny Training Engine supports backward graph pruning and sparse update at IR-level.
- After graph pruning, un-used weights and sub-tensors are pruned from DAG => 6.5-8.7x memory saving

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



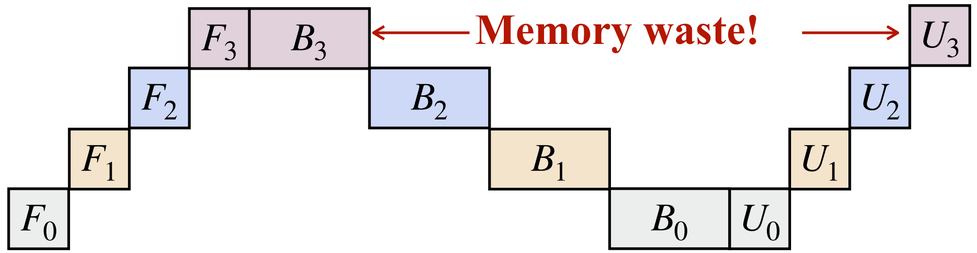
(a) Conventional way to update parameters

F: Forward, *B*: Backward, *U*: Update

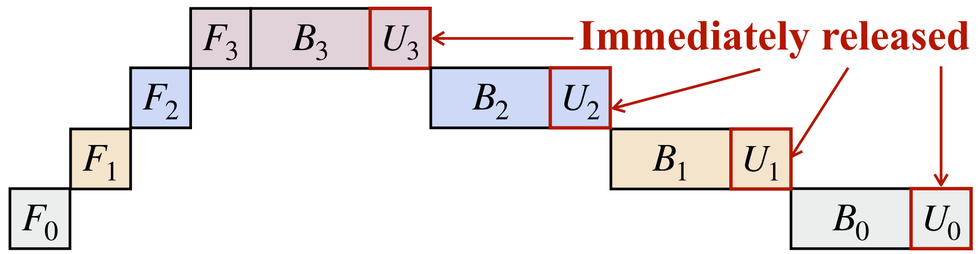
3. Tiny Training Engine (TTE)

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



(a) Conventional way to update parameters



(b) Operator re-ordering

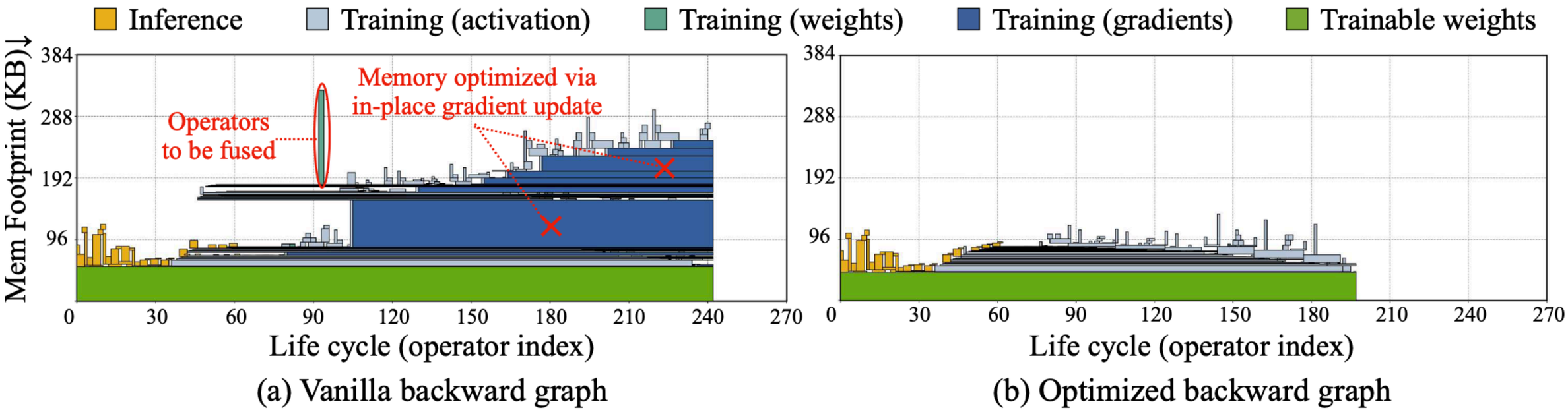
Operator life-cycle analysis reveals the **memory redundancy** in the optimization step.

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

F: Forward, *B*: Backward, *U*: Update

3. Tiny Training Engine (TTE)

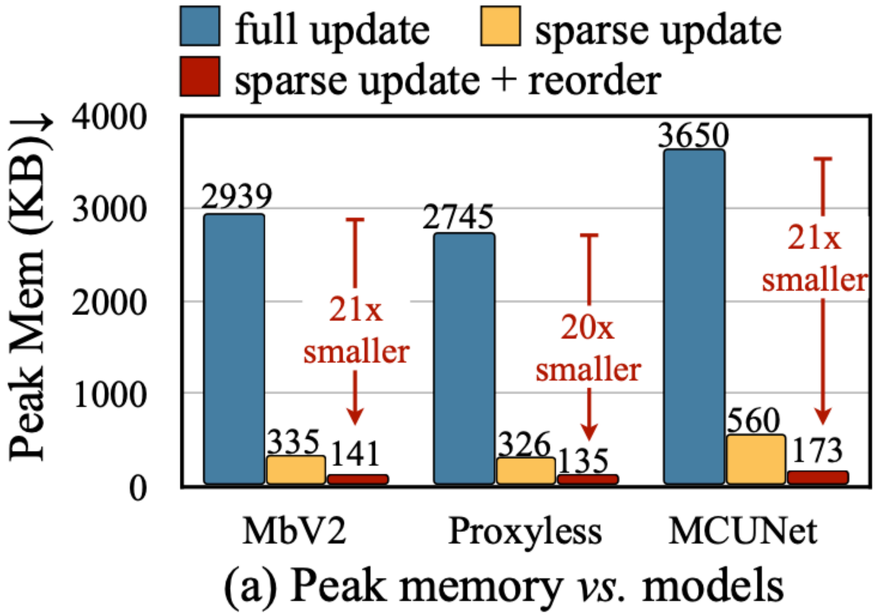
Re-ordering reduces memory footprint



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

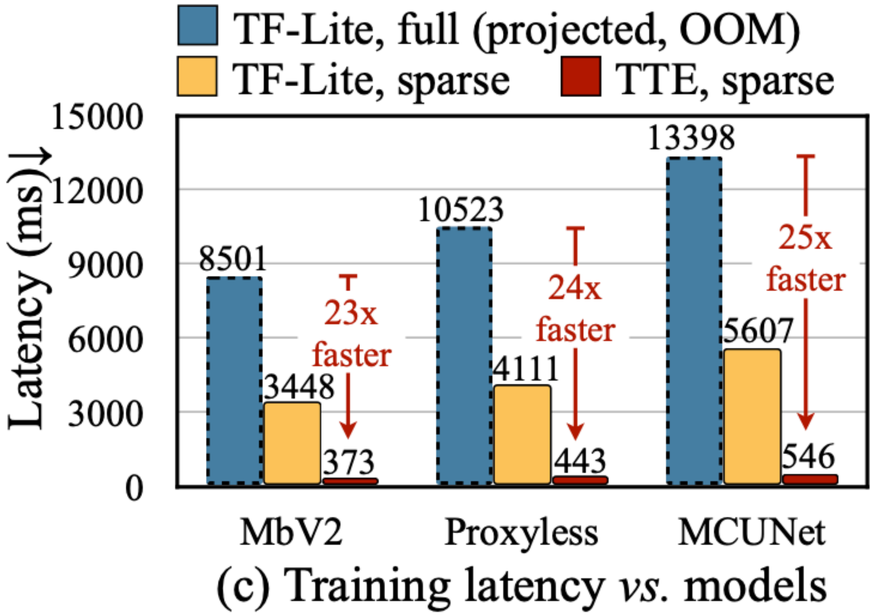
3. Tiny Training Engine (TTE)

Smaller memory usage, faster training speed



(a) Peak memory *vs.* models

20x smaller memory

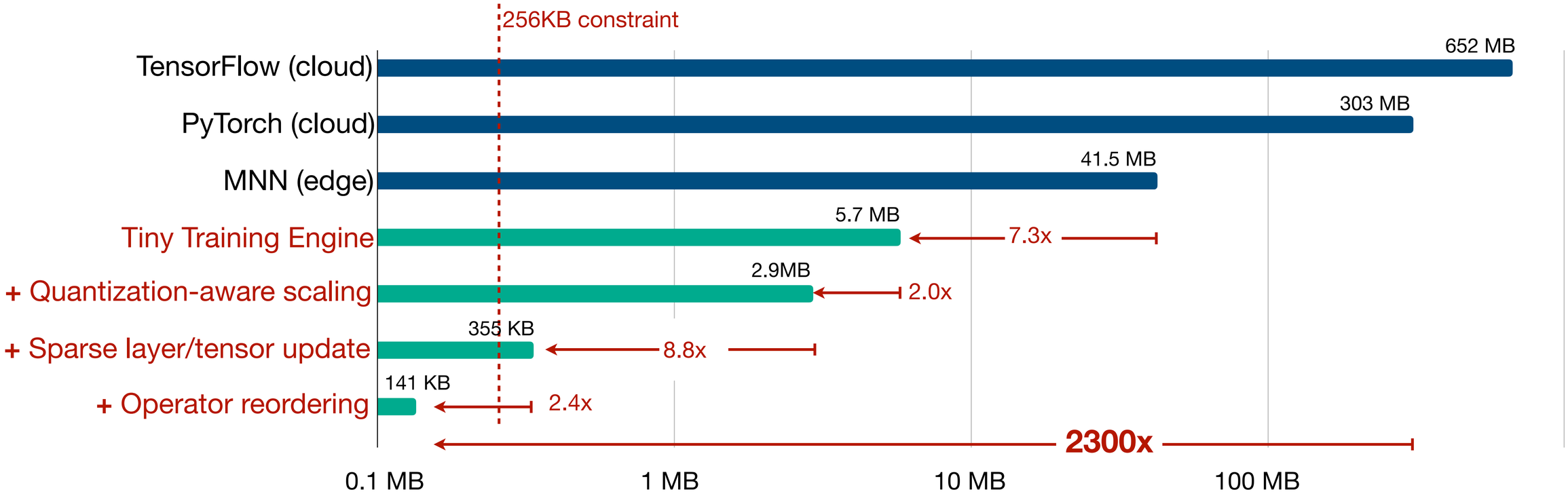


(c) Training latency *vs.* models

23x faster speed

Tiny Training

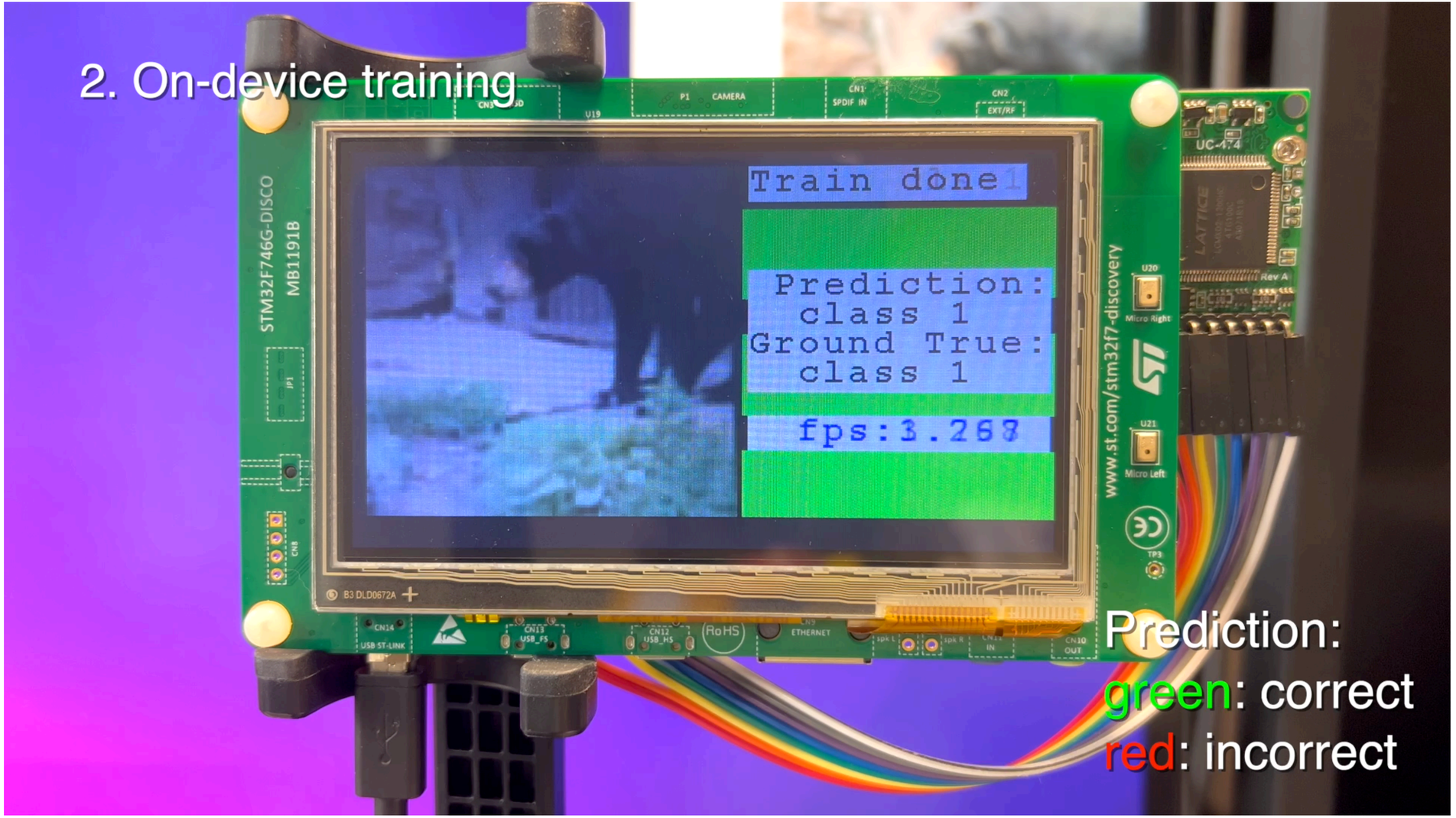
Co-design Results



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



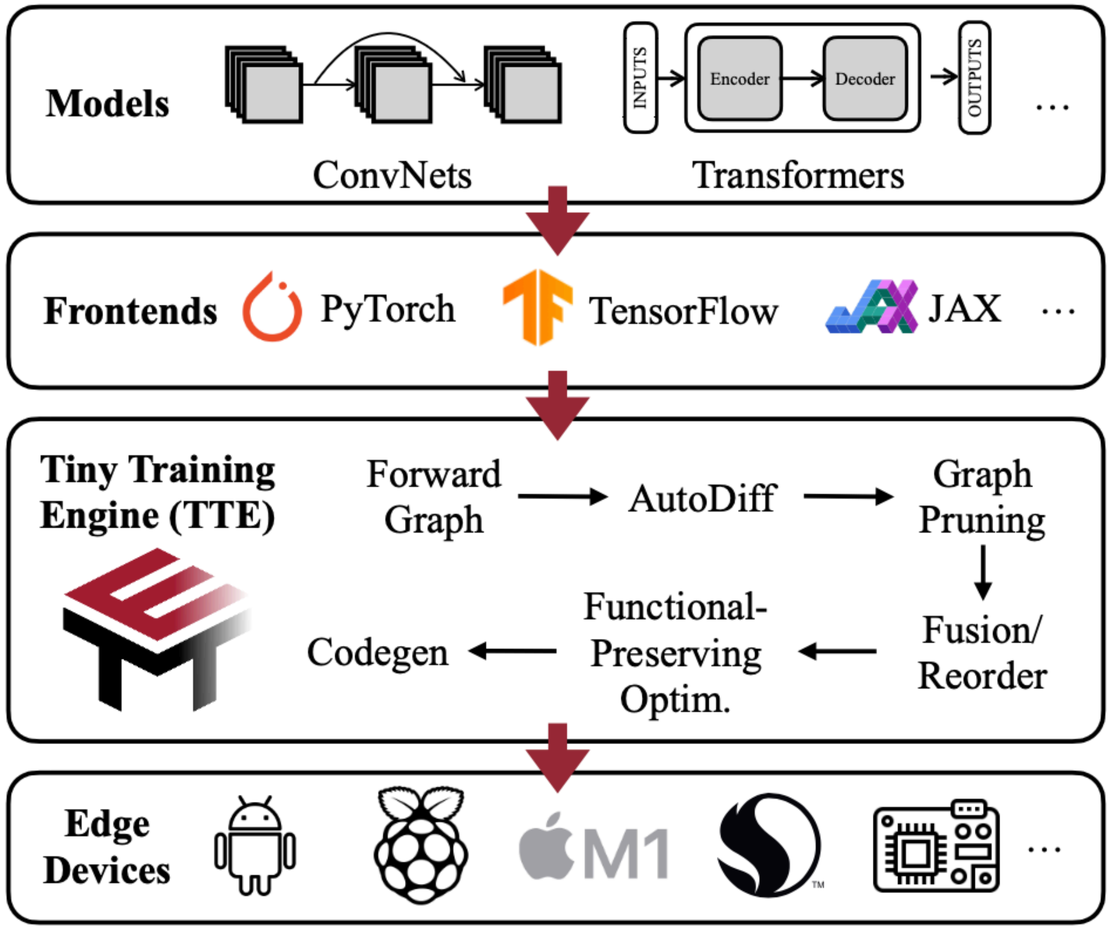
https://www.bilibili.com/video/BV1qv4y1d7MV/

https://youtu.be/XaDCO8YtmBw

Extending TTE to More Platforms

Accelerate on-device training on diverse edge hardware

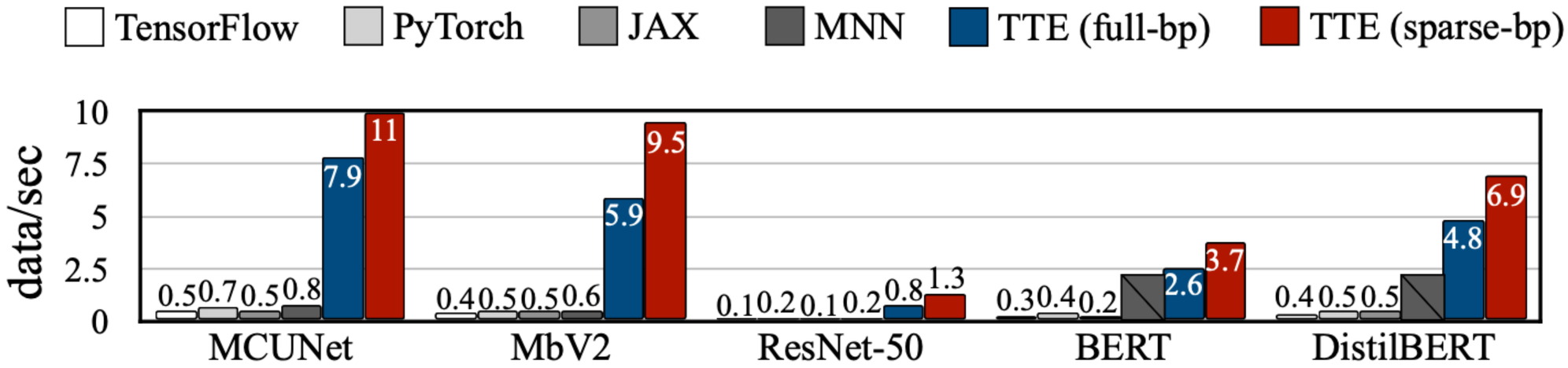
- We extend TTE to support:
  - Diverse models (CNN + Transformers)
  - Diverse frontends
    - PyTorch
    - TensorFlow
    - Jax
  - Diverse hardware backends
    - Apple M1
    - Raspberry Pi
    - Smartphones
    - …



Extending TTE to More Platforms

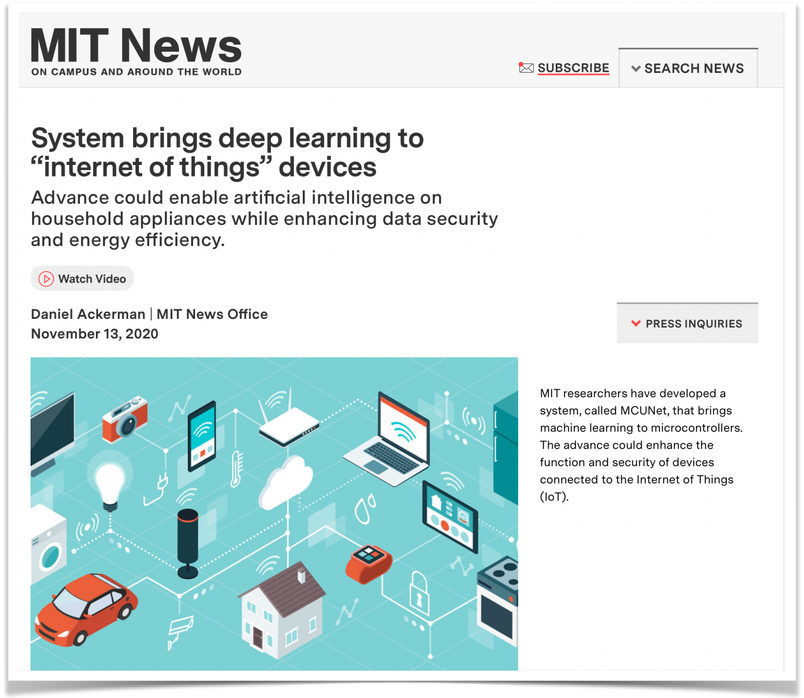
Consistently speed up training on diverse platforms

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

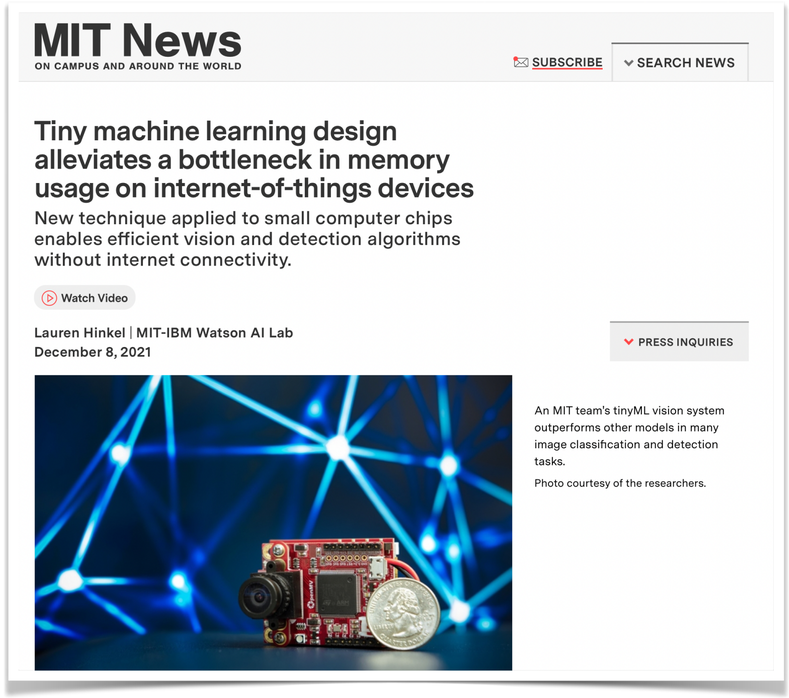


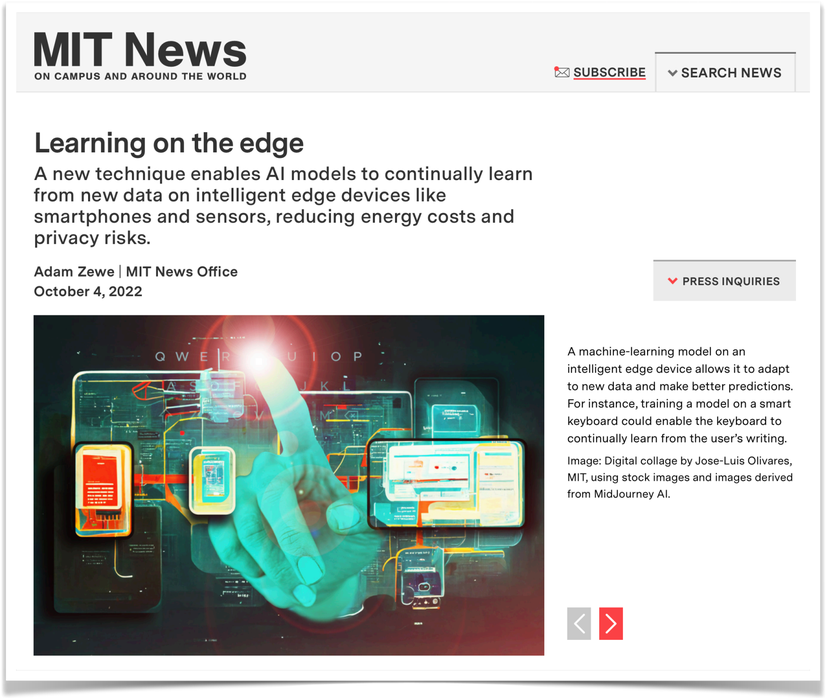
Results measured on Raspberry Pi 4B+.

Media



(Homepage highlight)





(Homepage highlight)

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]

Open Source

Product Team Enterprise Explore Marketplace Pricing Search

mit-han-lab / mcunet Public

Code Issues 6 Pull requests Actions Projects Security Insights

master 2 branches 0 tags

README.md

MCUNet: Tiny Deep

This is the official implementation of th

website | paper | paper (v2) | de

Product Team Enterprise Explore Marketplace Pricing Search

mit-han-lab / tinyengine Public

Code Issues Pull requests Actions

master 1 branch 0 tags

README.md

TinyEngine

This is the official implementation of TinyEngine,
Microcontrollers. TinyEngine is a part of MCUNet,
co-design framework for tiny deep learning on mi
tight memory budgets.

The MCUNet and TinyNAS repo is here.

MCUNetV1 | MCUNetV2 | MCUNetV3

mit-han-lab / tiny-training Public

Edit Pins Unwatch 8 Fork 0 Star 65

Code Issues 1 Pull requests Actions Projects Wiki Security Insights Settings

main 1 branch 0 tags Go to file Add file Code

Lyken17 Merge branch 'main' of https://github.com/mit-han... f8dfb50 yesterday 4 commits

algorithm	prepare open source	2 days ago
compilation	prepare open source	2 days ago
figures	refine qas_accuracy figure	yesterday
.gitignore	prepare open source	2 days ago
.gitmodules	prepare open source	2 days ago
LICENSE	prepare open source	2 days ago
README.md	minor update	yesterday
assets	prepare open source	2 days ago
configs	prepare open source	2 days ago

README.md

On-Device Training Under 256KB Memory

About

On-Device Training Under 256KB Memory [NeurIPS'22]

tinytraining.mit.edu

edge-ai on-device-training learning-on-the-edge

Readme
MIT license
65 stars
8 watching
0 forks

Releases

No releases published
Create a new release

Packages

No packages published

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!**