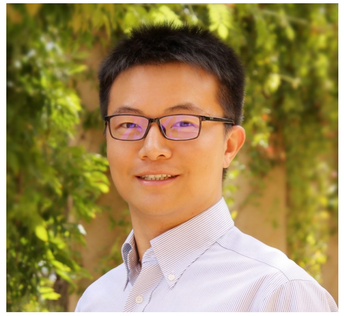
MIT HAN LAB

EfficientML.ai Lecture 21 On-Device Training and Transfer Learning



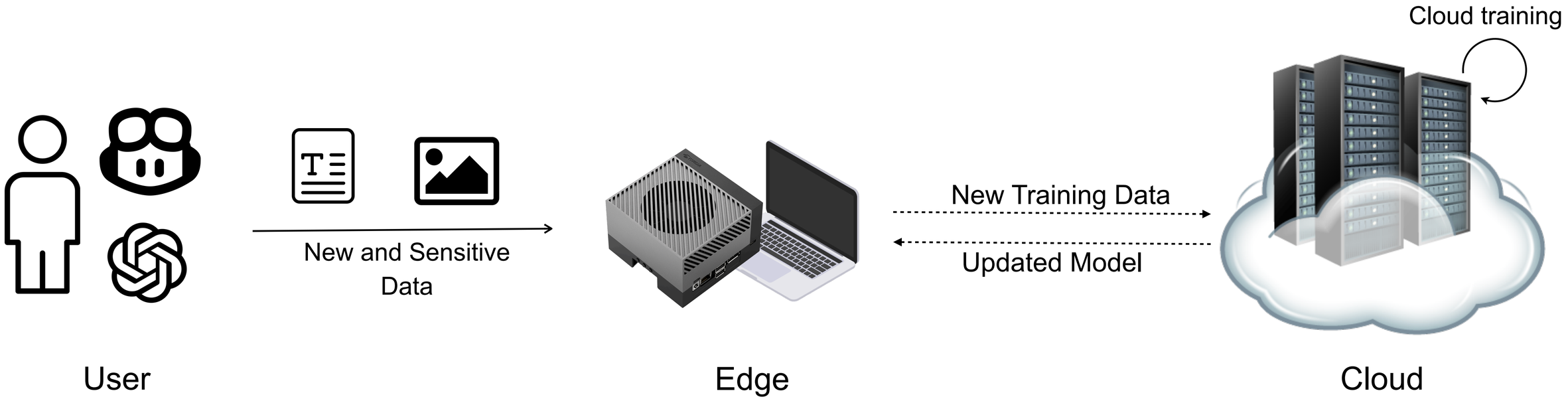
Song Han

Associate Professor, MIT
Distinguished Scientist, NVIDIA

@SongHan_MIT

On-Device Learning

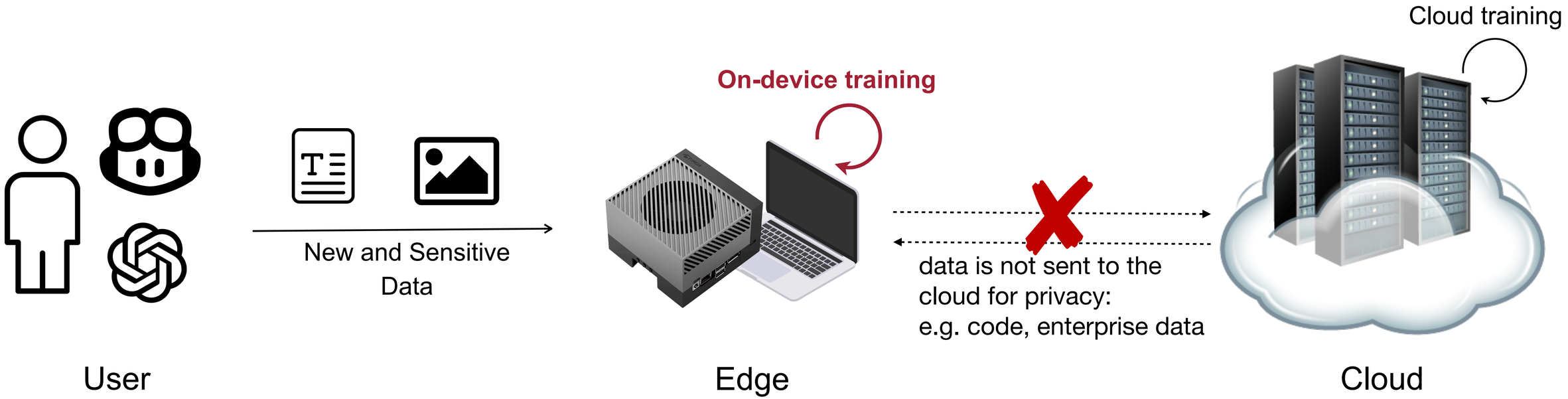
- Transfer learning at the "edge", rather than cloud.



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

On-Device Learning

- Transfer learning at the "edge", rather than cloud.



- Customization: AI systems need to continually adapt to new data collected from the sensors.
- Privacy: Data does not leave the device.

Lecture Plan

Today we will discuss:

1. Deep leakage from gradients, gradient is not safe to share
2. Memory bottleneck of on-device training
3. Tiny transfer learning (TinyTL)
4. Sparse back-propagation (SparseBP)
5. Quantized training with quantization aware scaling (QAS)
6. PockEngine: system support for sparse back-propagation

Quantized Training

Sparse Training

PockEngine

Lecture Plan

1. **Deep leakage from gradients, gradient is not safe to share**
2. Memory bottleneck of on-device training
3. Tiny transfer learning (TinyTL)
4. Sparse back-propagation (SparseBP)
5. Quantized training with quantization aware scaling (QAS)
6. PockEngine: system support for sparse back-propagation

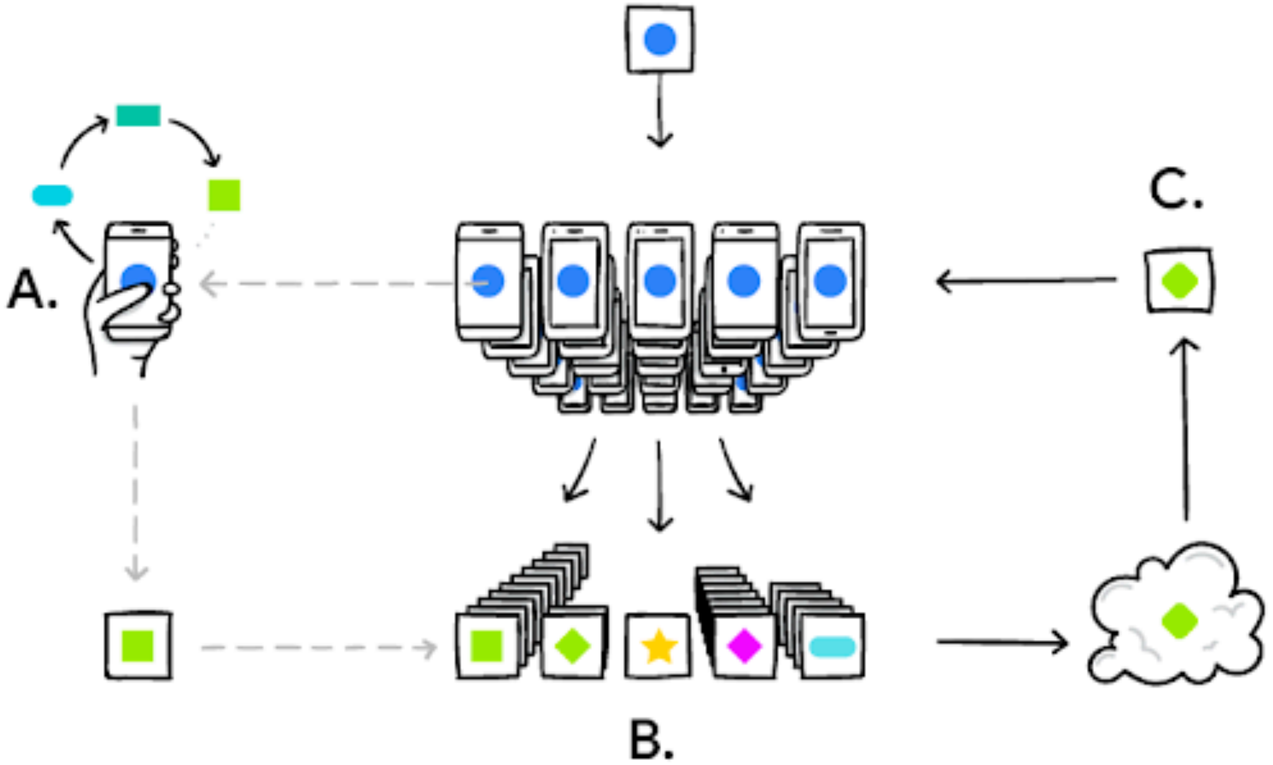
Quantized Training

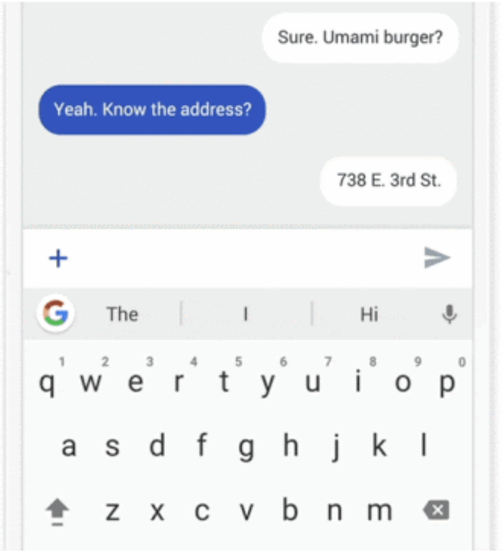
Sparse Training

PockEngine

On-Device Learning

Federated learning: only share the gradients / weights, user data stays local.

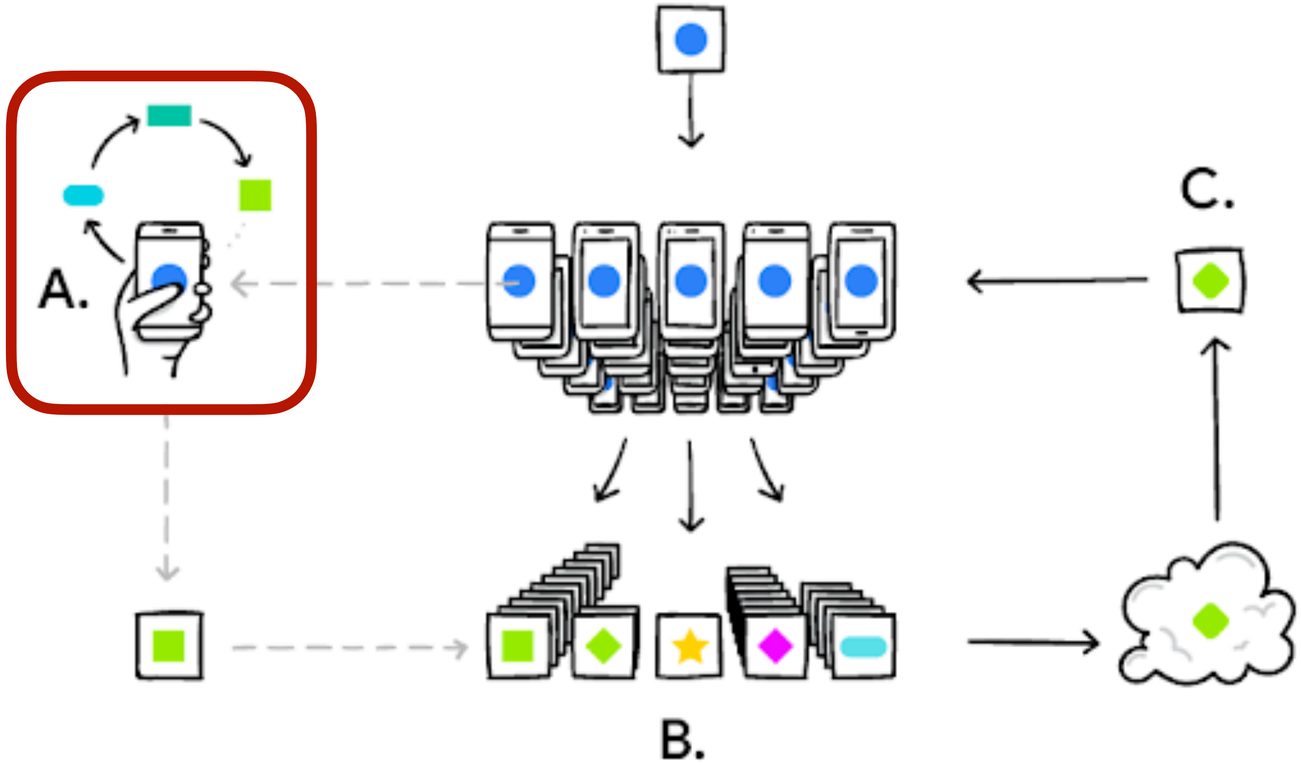




Communication-Efficient Learning of Deep Networks from Decentralized Data. [McMahan, 2016]

Background of Federated Learning

FedAvg Algorithm

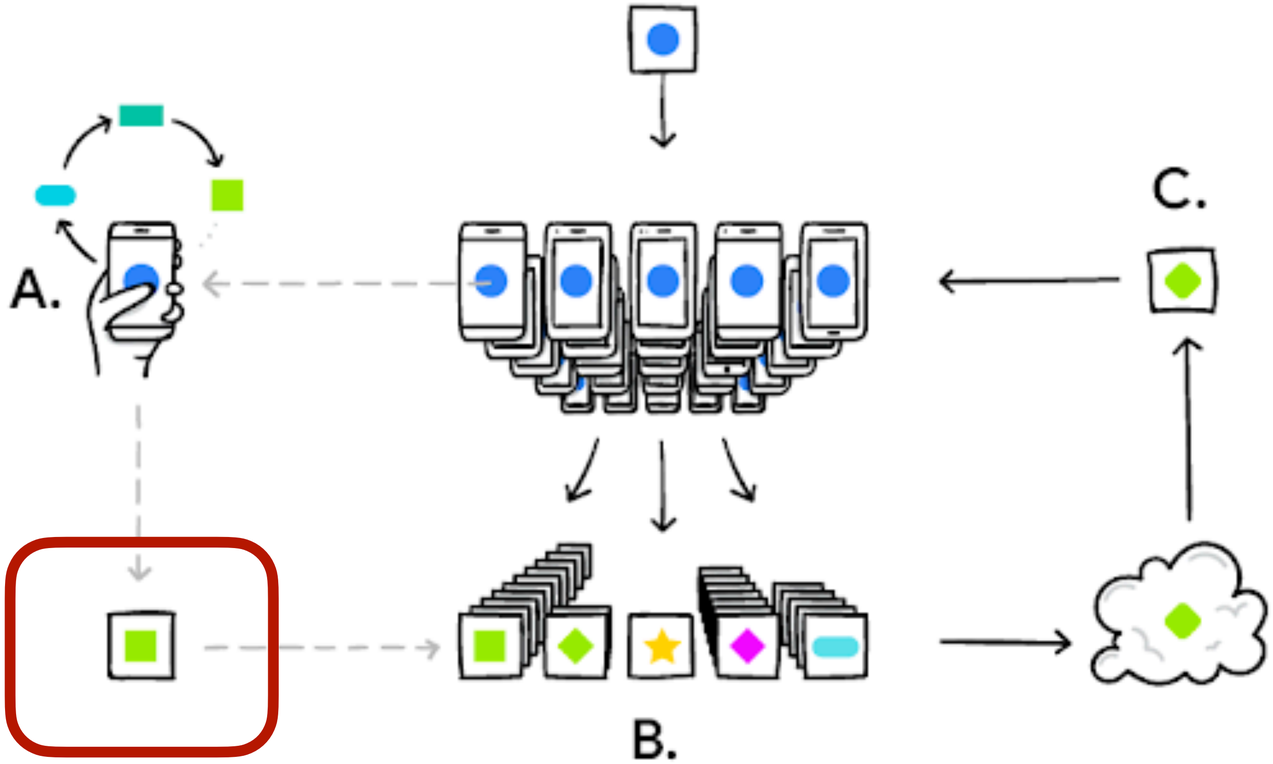


1. Users generate personal data on device and perform local training.

Communication-Efficient Learning of Deep Networks from Decentralized Data. [McMahan, 2016]

Background of Federated Learning

FedAvg Algorithm

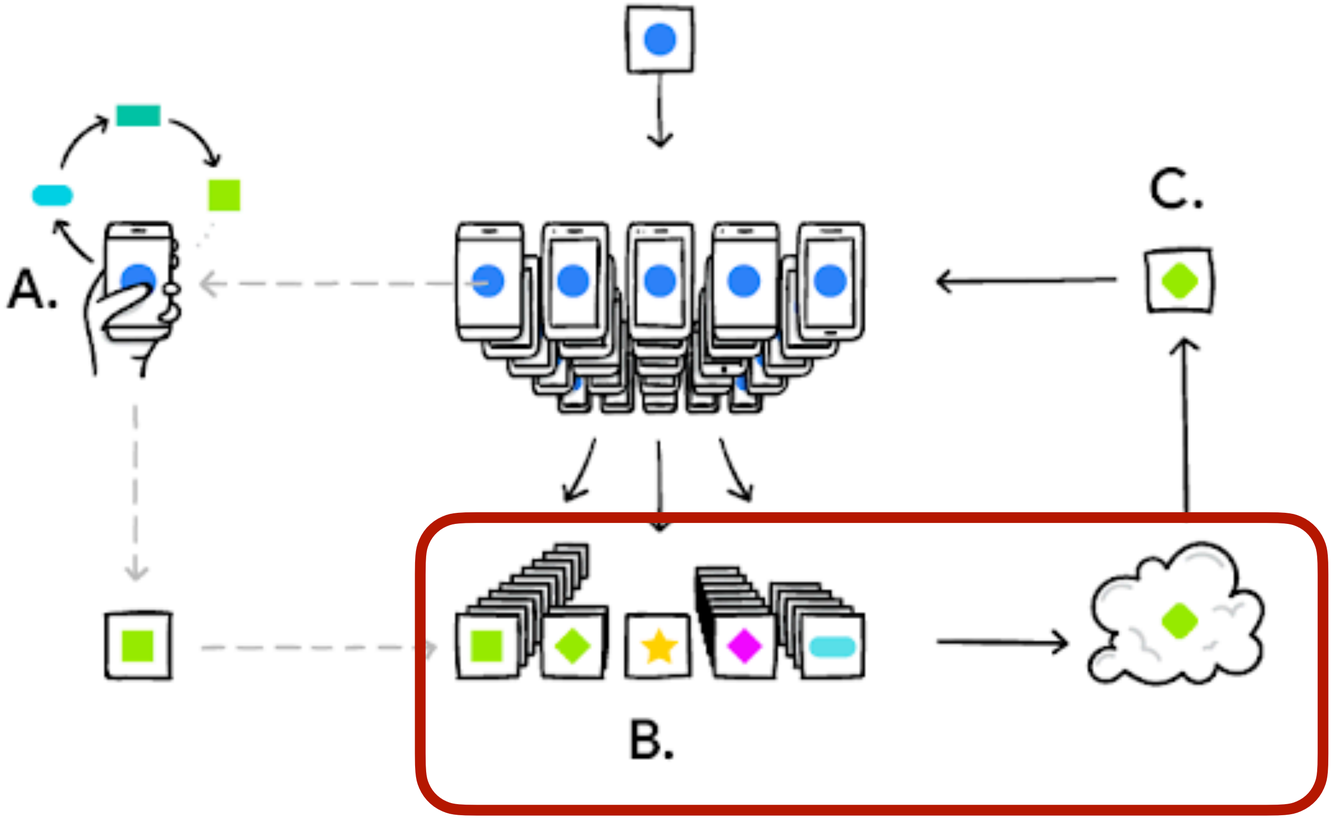


1. Users generate personal data on device and perform local training.
2. Each device update its model using local data for N iterations.

Communication-Efficient Learning of Deep Networks from Decentralized Data. [McMahan, 2016]

Background of Federated Learning

FedAvg Algorithm

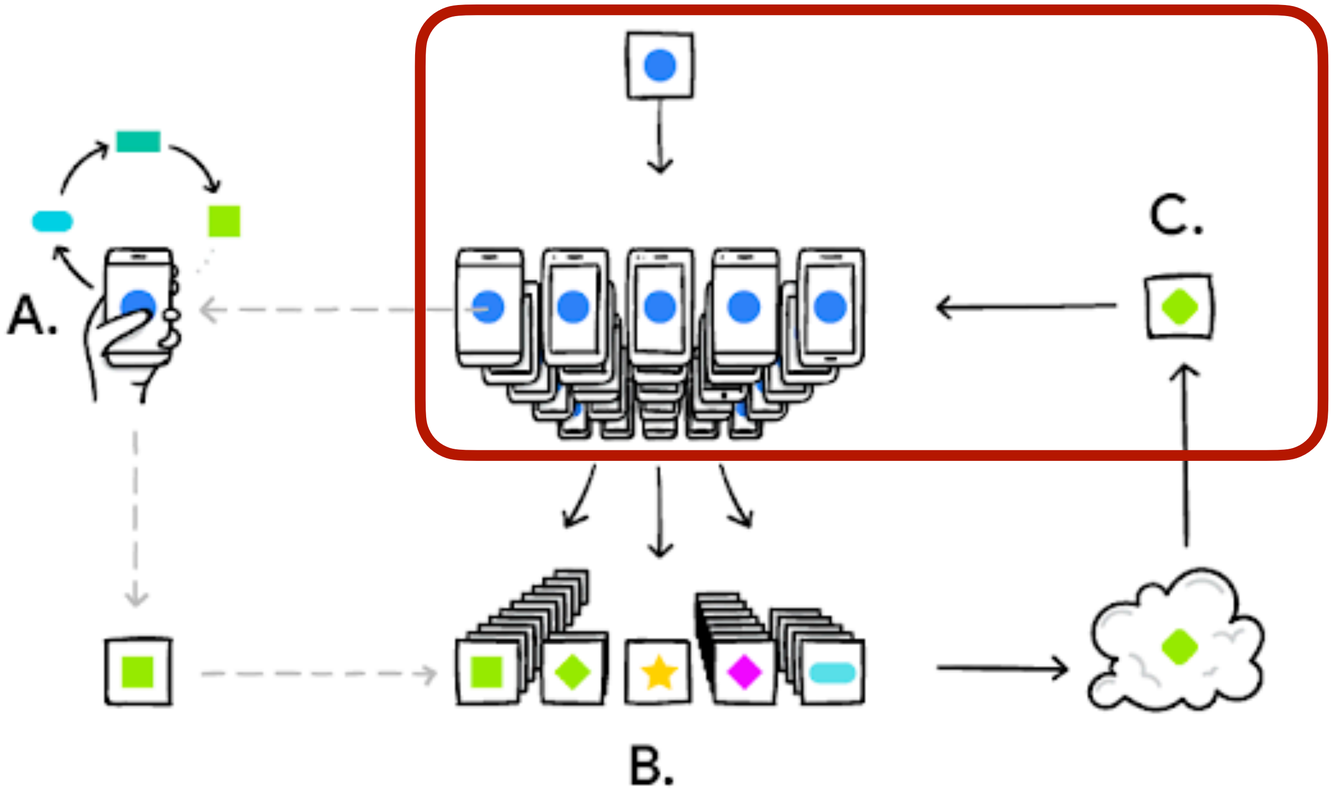


1. Users generate personal data on device and perform local training.
2. Each device update its model using local data for N iterations.
3. Updated models are sent to the server.

Communication-Efficient Learning of Deep Networks from Decentralized Data. [McMahan, 2016]

Background of Federated Learning

FedAvg Algorithm

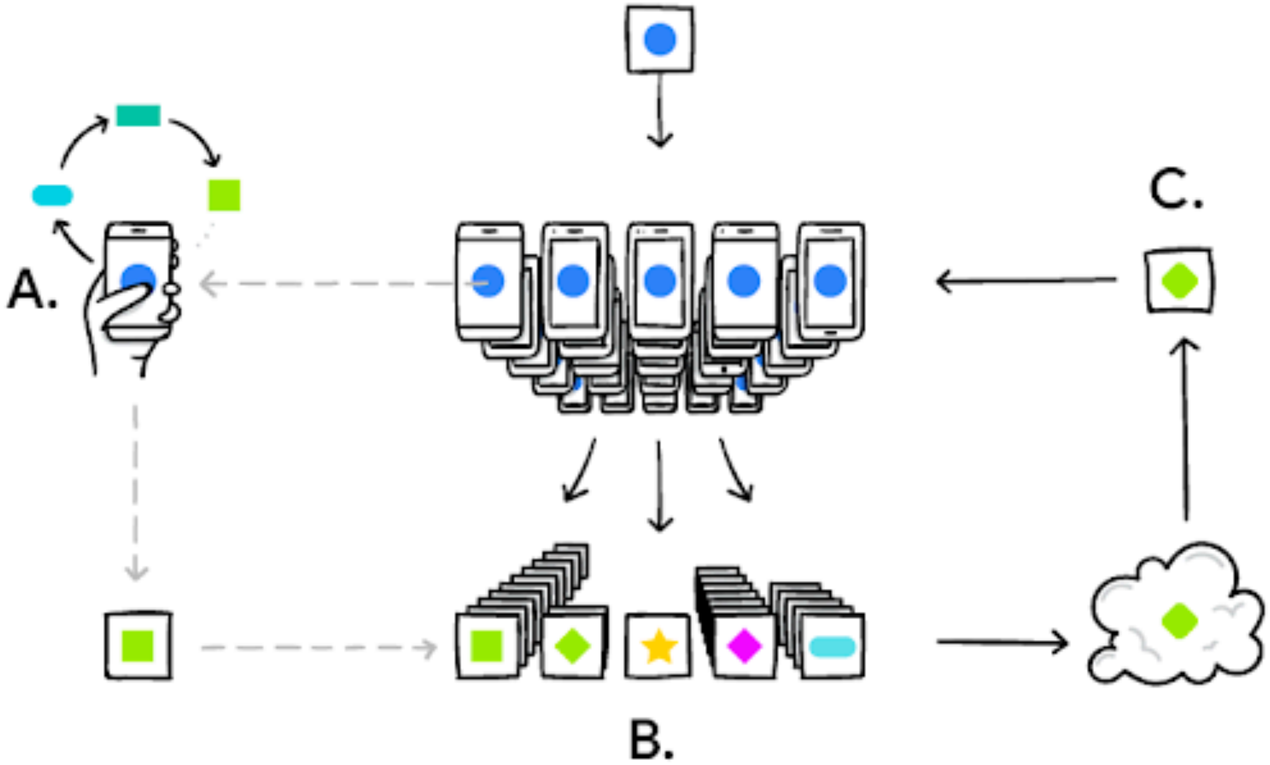


1. Users generate personal data on device and perform local training.
2. Each device update its model using local data for N iterations.
3. Updated models are sent to the server.
4. Models are averaged on the server and sent back to devices.

Communication-Efficient Learning of Deep Networks from Decentralized Data. [McMahan, 2016]

Background of Federated Learning

FedAvg Algorithm



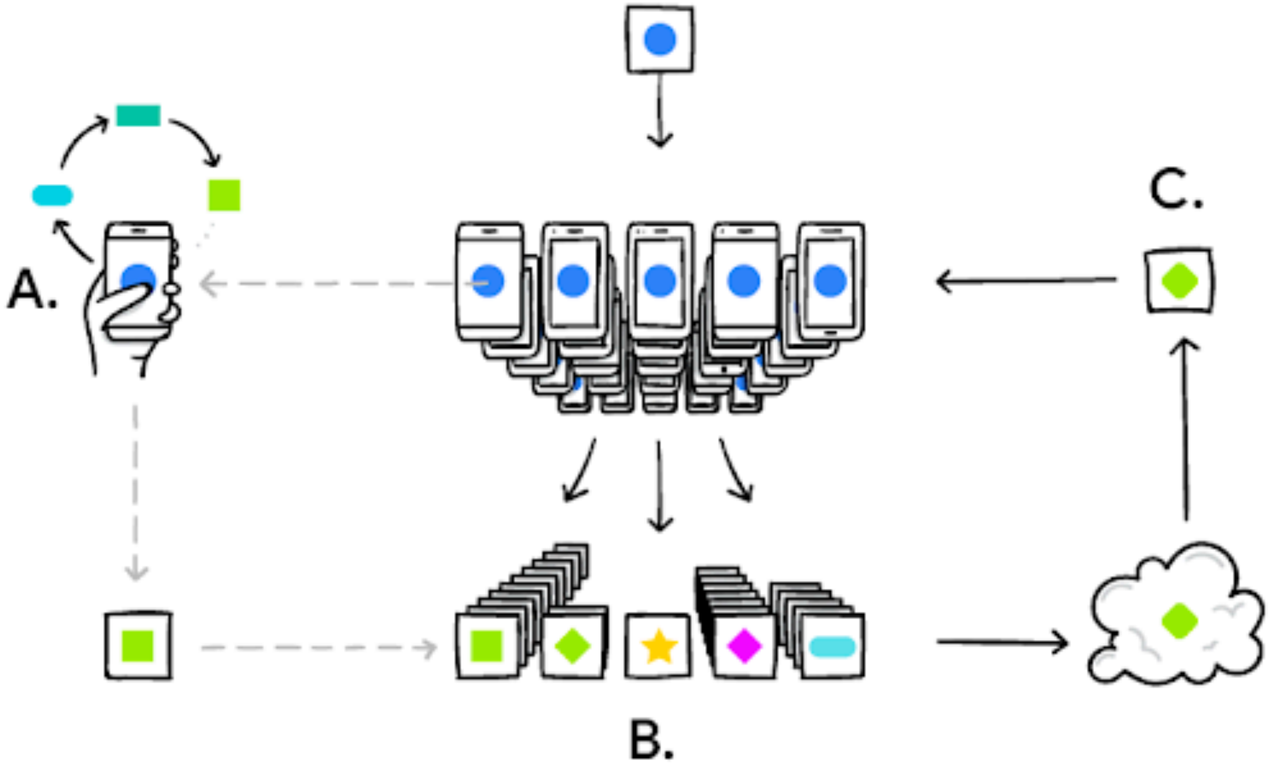
1. Users generate personal data on device and perform local training.
2. Each device update its model using local data for N iterations.
3. Updated models are sent to the server.
4. Models are averaged on the server and sent back to devices.

The important & private user data **NEVER** leaves local devices.

Communication-Efficient Learning of Deep Networks from Decentralized Data. [McMahan, 2016]

Background of Federated Learning

FedAvg Algorithm



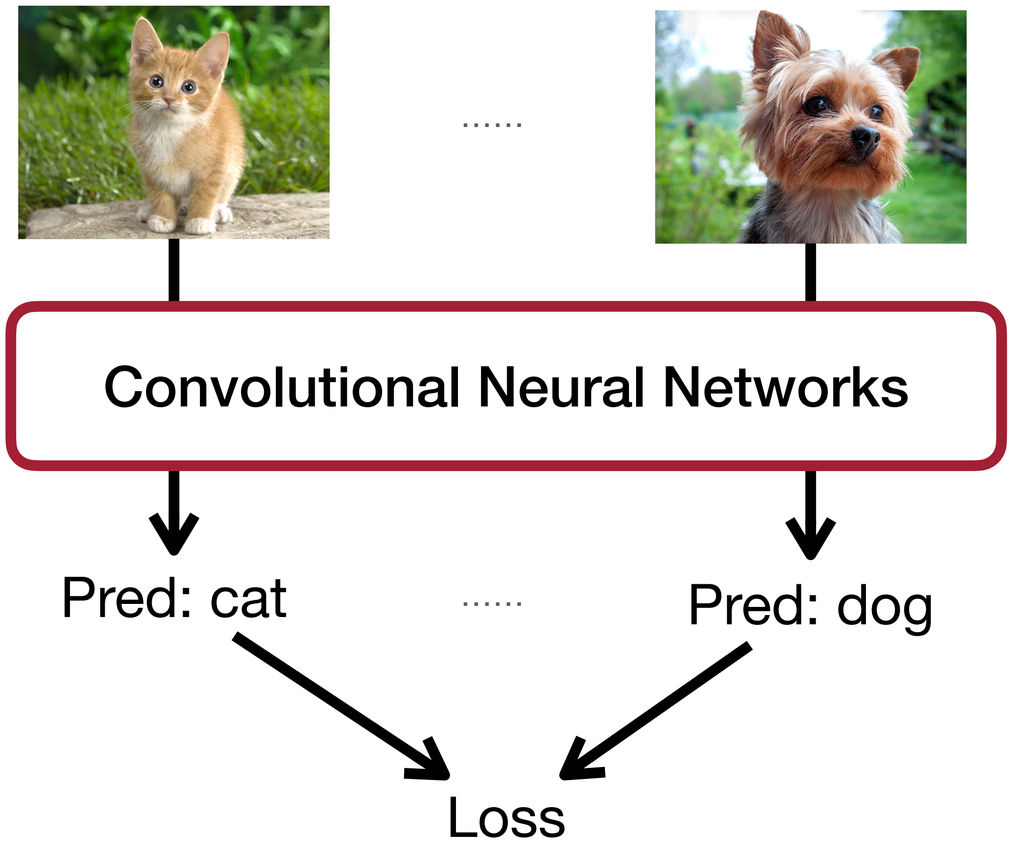
1. Users generate personal data on device and perform local training.
2. Each device update its model using local data for N iterations.
3. Updated models are sent to the server.
4. Models are averaged on the server and sent back to devices.

The important & private user data never leaves local devices.

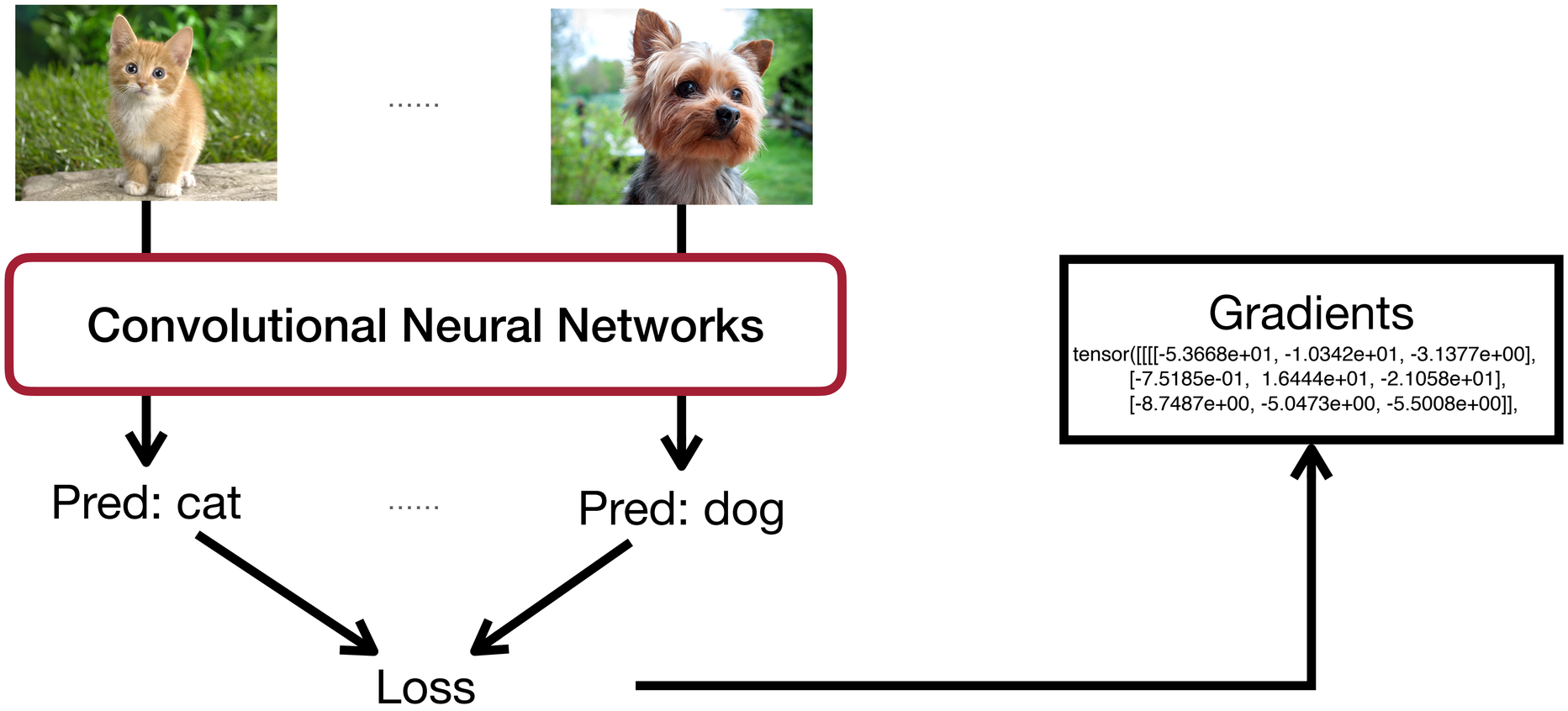
Safe…?

Communication-Efficient Learning of Deep Networks from Decentralized Data. [McMahan, 2016]

Rethink the Safety of Gradients

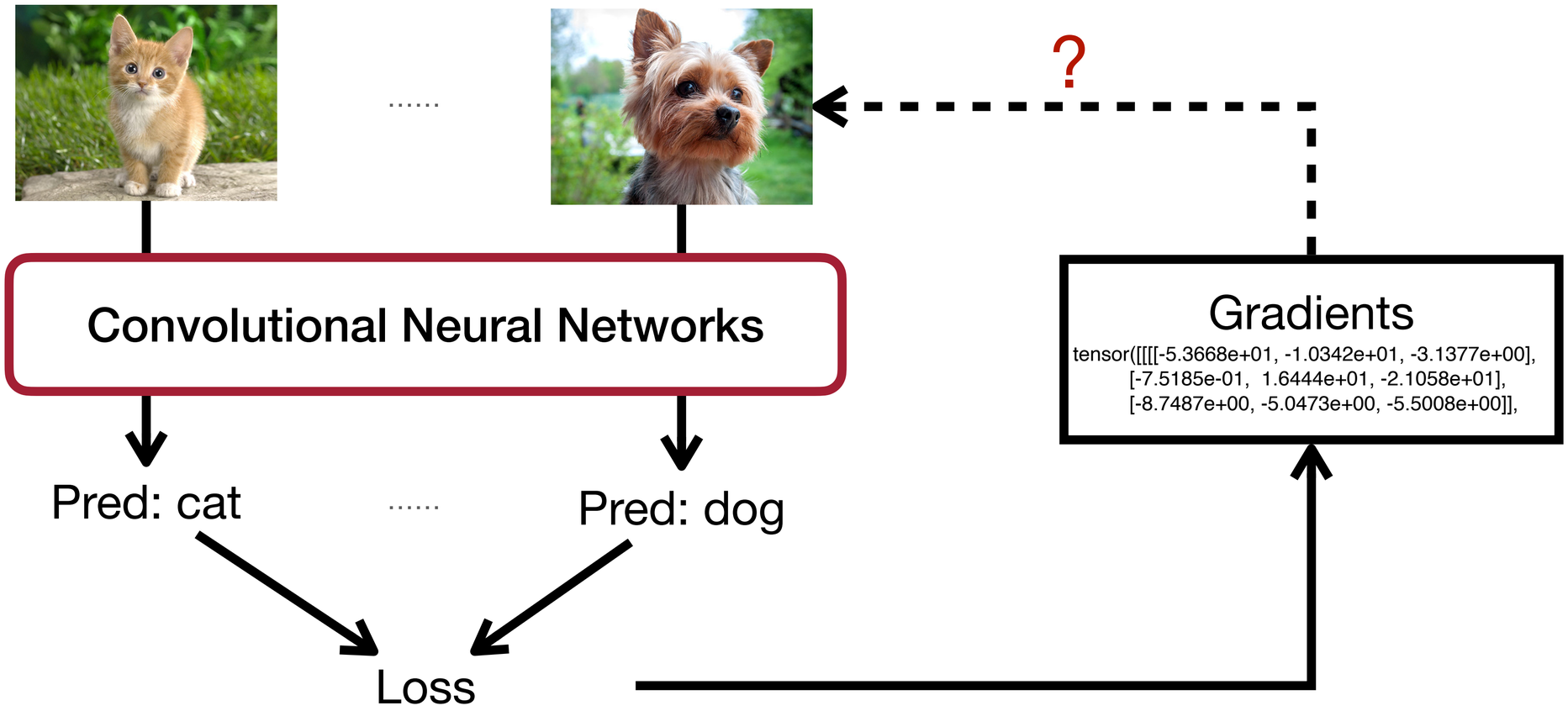


Rethink the Safety of Gradients



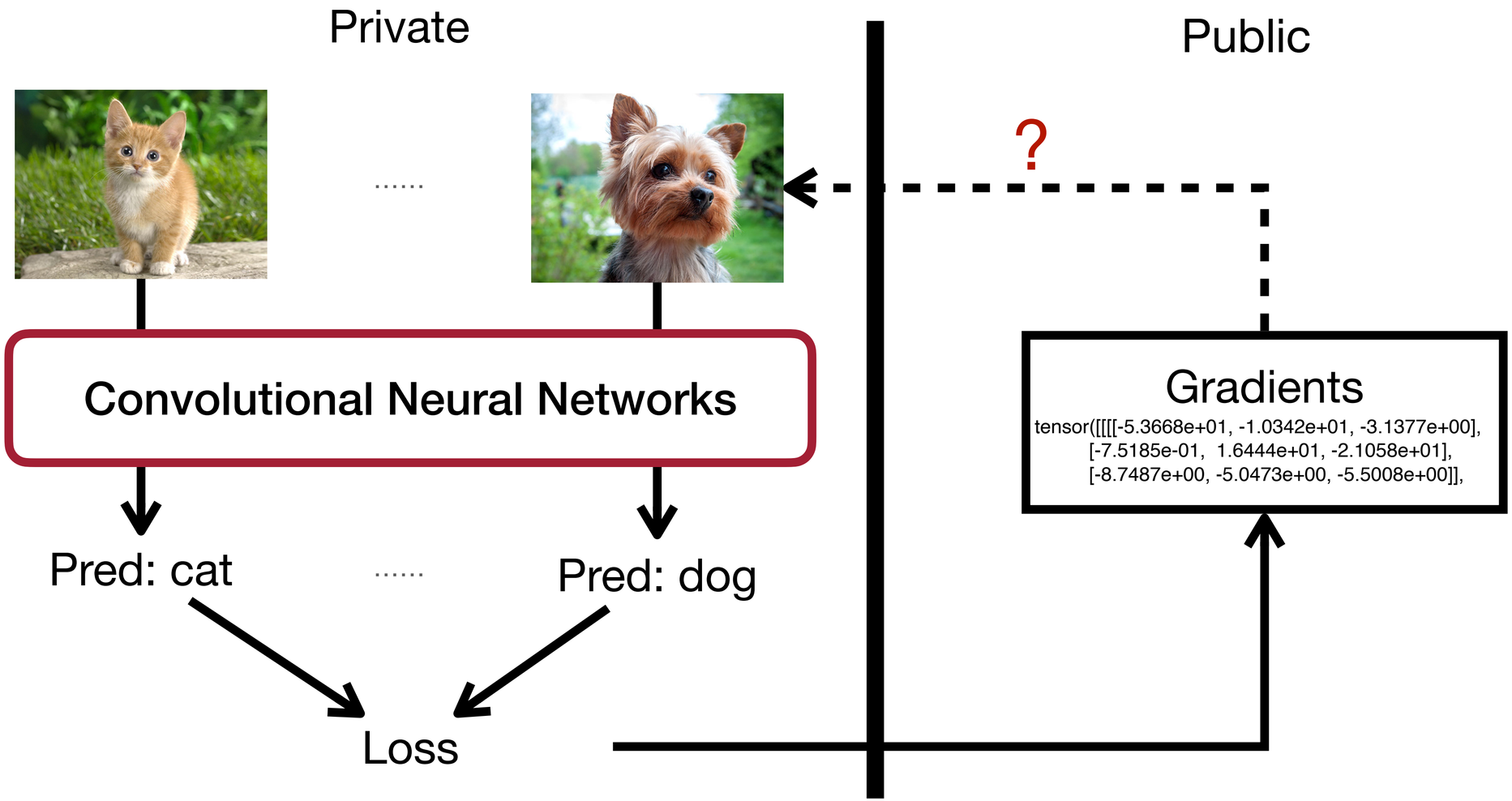
derive gradients from model and training data.

Rethink the Safety of Gradients



Can we steal the training data from gradients?

Rethink the Safety of Gradients



If that is possible, then sharing the gradient is not safe!

Rethink the Safety of Gradients

Existing Work of Gradient Inversion

Membership Inference [Shokri 2016]

- Given gradients, it's possible to find whether a data point belongs to the batch.

Property Inference [Melis 2018]

- Given gradients, it's possible to find whether a data point with certain property is in the batch.

Exploiting unintended feature leakage in collaborative learning. [Melis 2018]
Membership inference attacks against machine learning models. [Shokri 2016]

Rethink the Safety of Gradients

Existing Work of Gradient Inversion

Membership Inference [Shokri 2016]

- Given gradients, it's possible to find whether a data point belongs to the batch.

Property Inference [Melis 2018]

- Given gradients, it's possible to find whether a data point with certain property is in the batch.

Gradients contain certain information about the training data.

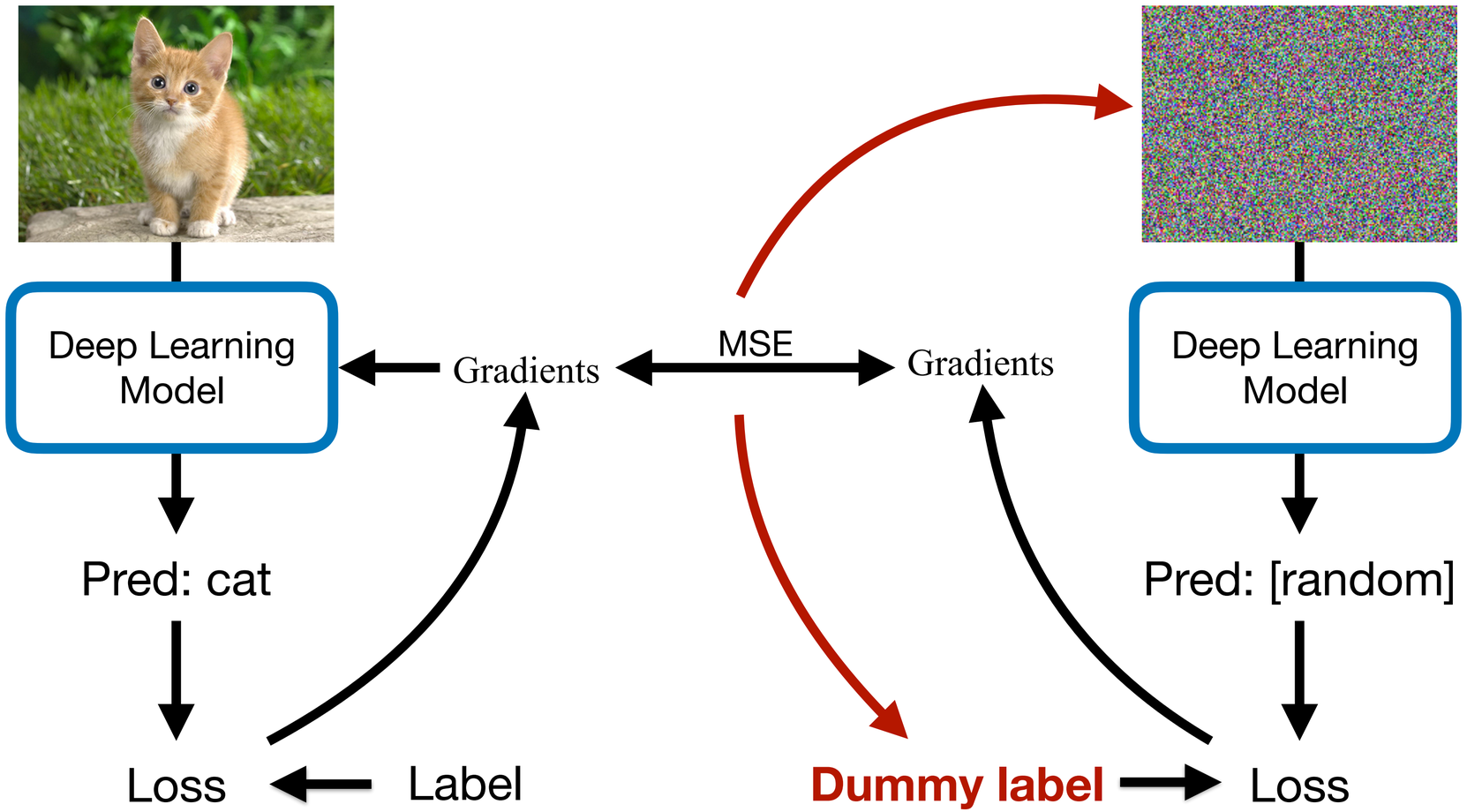
Can we obtain the **raw training data** from **gradient**?

Exploiting unintended feature leakage in collaborative learning. [Melis 2018]
Membership inference attacks against machine learning models. [Shokri 2016]

Deep Leakage from Gradients

Normal Training:
forward-backward, update **model weights**

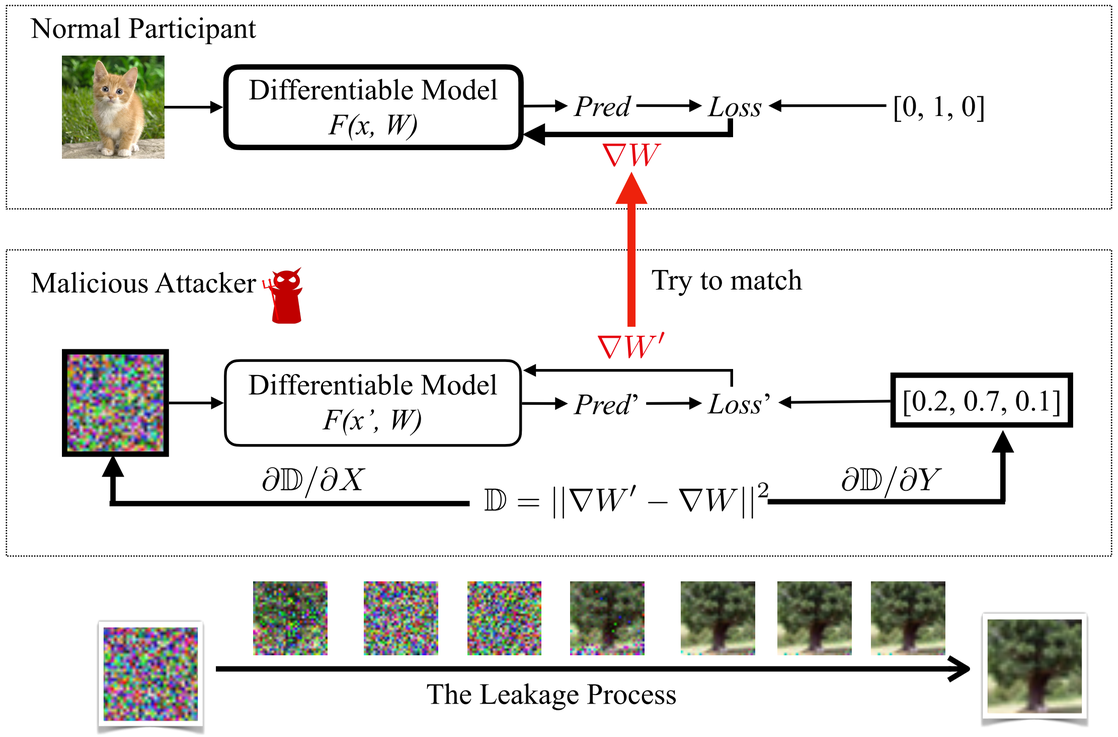
Deep Leakage Attack:
forward-backward, update **the dummy data**



Deep Leakage from Gradient. [Zhu et al, NeurIPS 2019]

Deep Leakage from Gradients

Deep Leakage Attack via Gradients Matching



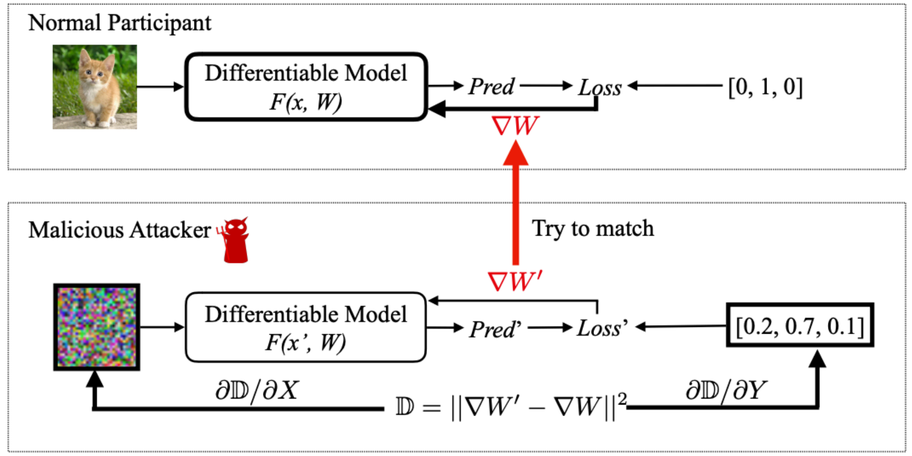
Only gradients are shared between malicious attacker and normal users.

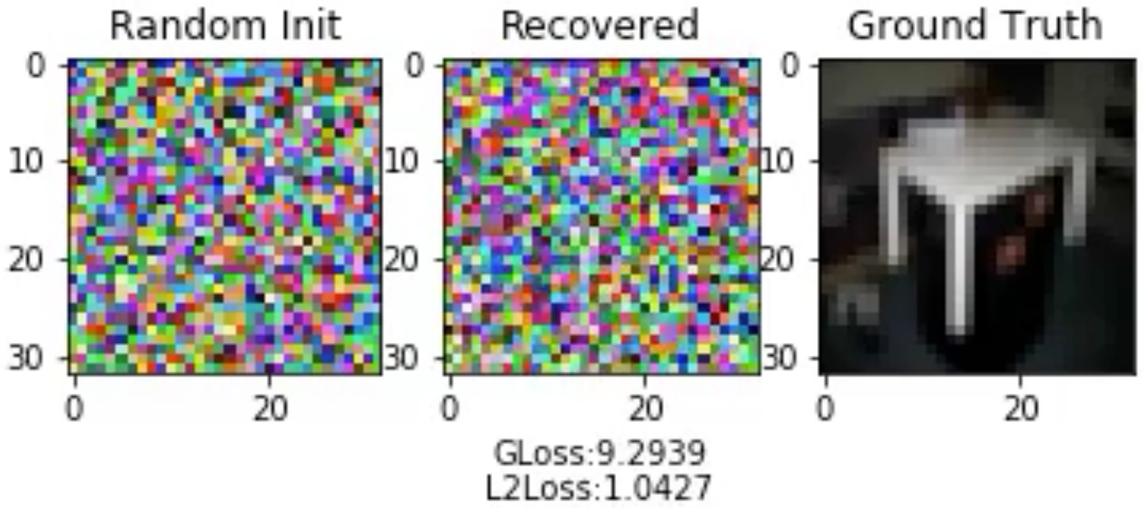
But, this action indeed **leaks the privacy**!

Deep Leakage from Gradient. [Zhu et al, NeurIPS 2019]

Deep Leakage Attack Results

Attack on Vision Model (bs=1)

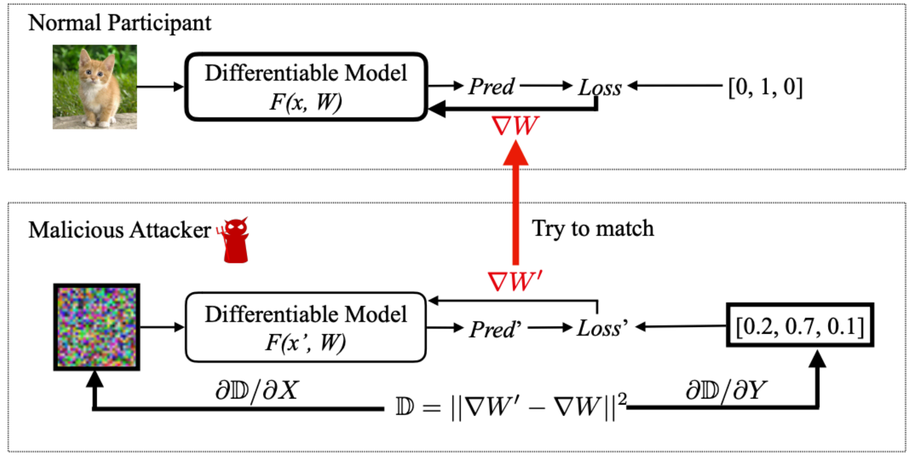


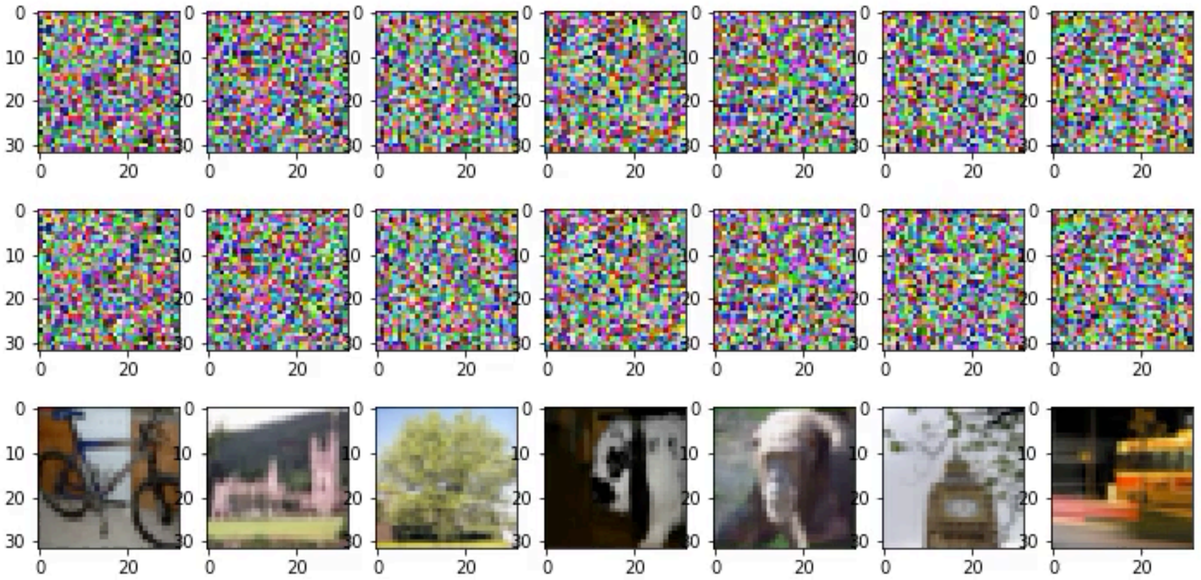


Deep Leakage from Gradient. [Zhu et al, NeurIPS 2019]

Deep Leakage Attack Results

Attack on Vision Model (bs=8)





Deep Leakage from Gradient. [Zhu et al, NeurIPS 2019]

Deep Leakage Attack Results

Attack on Language Model (BERT, Masked Language Model)

Iters=0: tilting fill given **less word **itude fine **nton overheard living vegas **vac **vation *f forte **dis cerambycidae ellison **don yards marne **kali

Iters=10: tilting fill given **less full solicitor other ligue shrill living vegas rider treatment carry played sculptures lifelong ellison net yards marne **kali

Iters=20: registration , volunteer applications , at student travel application open the ; week of played ; child care will be glare .

Iters=30: registration, volunteer applications, and student travel application open the first week of september . child care will be available

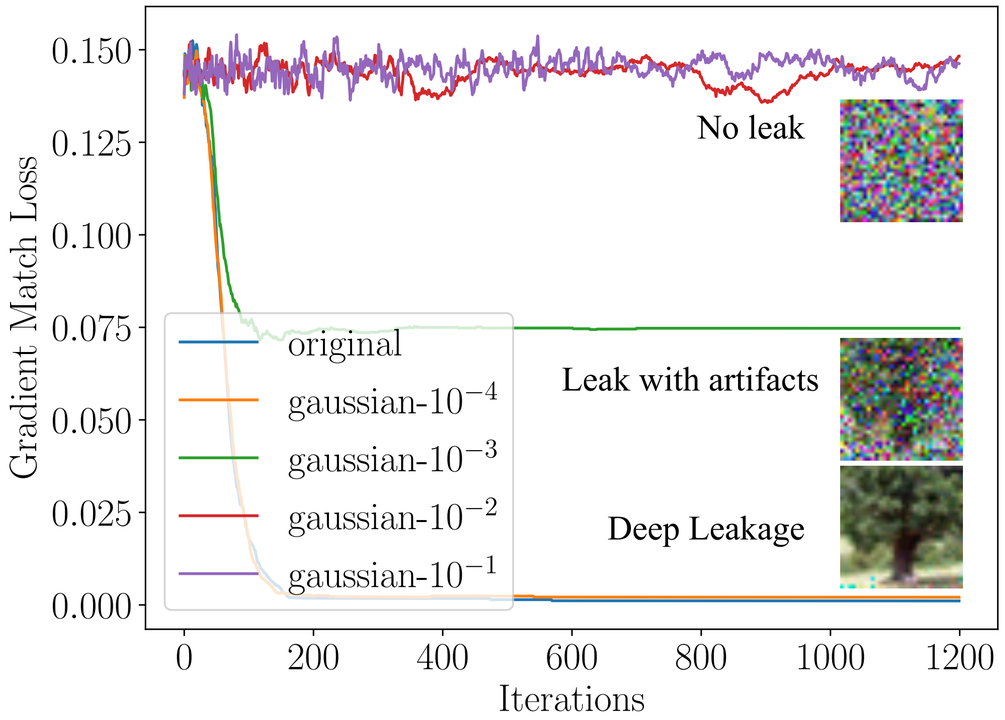
Original text: Registration, volunteer applications, and student travel application open the first week of September. Child care will be available.

Unmatched words are marked with red.

Deep Leakage from Gradient. [Zhu et al, NeurIPS 2019]

Defense Strategy for Deep Leakage

Gaussian and laplacian noise



	Original	**G**-10^{-4}	**G**-10^{-3}	**G**-10^{-2}	**G**-10^{-1}
Accuracy	76.3%	75.6%	73.3%	45.3%	≤1%
Defendability	–	✗	✗	✓	✓
		L-10^{-4}	**L**-10^{-3}	**L**-10^{-2}	**L**-10^{-1}
Accuracy	–	75.6%	73.4%	46.2%	≤1%
Defendability	–	✗	✗	✓	✓

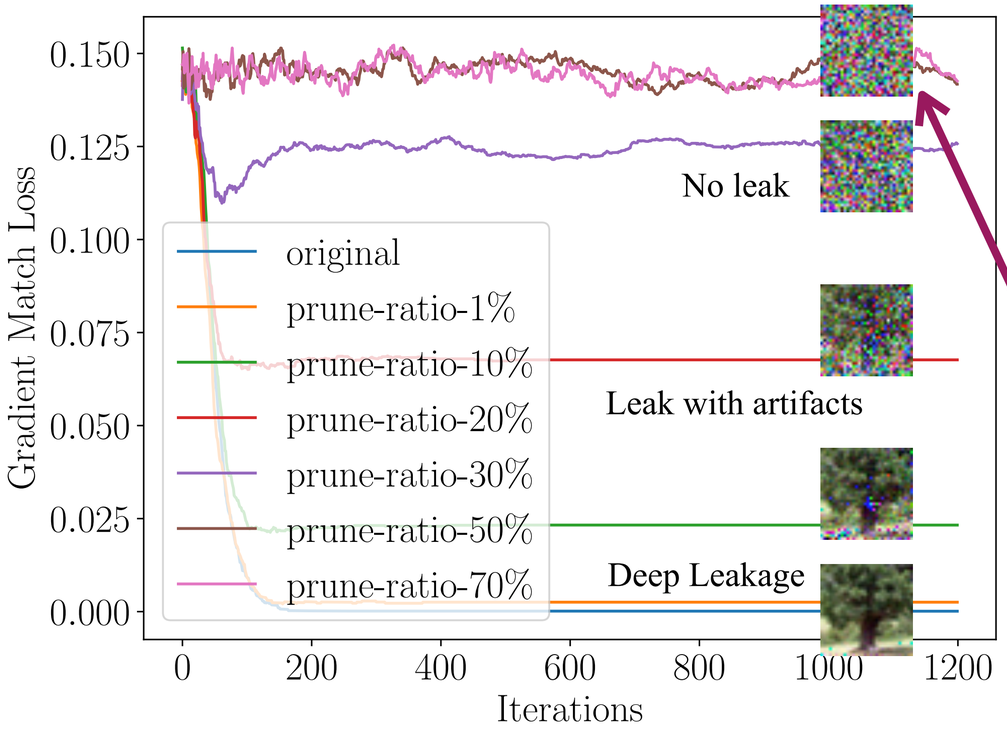
"G" denotes gaussian noise, "L" denotes laplacian noise

Simply applying noise cannot prevent deep leakage unless we allow significant accuracy drop (purple and red lines)

Deep Leakage from Gradient. [Zhu et al, NeurIPS 2019]

Defense Strategy for Deep Leakage

Gradient compression



ResNet50	Top-1	Defendablity
Prune-ratio:0%	75.96%	No
Prune-ratio:99% [2]	76.15% (+0.19%)	Yes

Gradient compression[2], **can effectively prevent** deep leakage while preserving accuracy.

Besides compression, DGC[2] further applies local accumulation to obfuscate gradients thus better protect users' privacy.

[1] Deep Leakage from Gradient. [Zhu et al, NeurIPS 2019]
[2] Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training [Lin et al, ICLR 2018]

Deep Leakage from Gradients

Sharing gradients is as dangerous as sharing the original user data!

Simple **~20 lines** PyTorch implementation

```
def deep_leakage_from_gradients(model, origin_grad):
  dummy_data = torch.randn(origin_data.size())
  dummy_label =  torch.randn(dummy_label.size())
  optimizer = torch.optim.LBFGS([dummy_data, dummy_label] )

  for iters in range(300):
    def closure():
      optimizer.zero_grad()
      dummy_pred = model(dummy_data)
      dummy_loss = criterion(dummy_pred, dummy_label)
      dummy_grad = grad(dummy_loss, model.parameters(), create_graph=True)

      grad_diff = sum(((dummy_grad - origin_grad) ** 2).sum() \
        for dummy_g, origin_g in zip(dummy_grad, origin_grad))

      grad_diff.backward()
      return grad_diff

    optimizer.step(closure)

  return  dummy_data, dummy_label
```

Over > **2400 citations** since 2019

Deep Leakage from Gradients

Authors	Ligeng Zhu, Zhijian Liu, Song Han
Publication date	2019/9
Conference	Neural Information Processing Systems (NeurIPS), 2019
Description	Passing gradient is a widely used scheme in modern multi-node learning system (eg, distributed training, collaborative learning). In a long time, people used to believe that gradients are safe to share: ie, the training set will not be leaked by gradient sharing. However, in this paper, we show that we can obtain the private training set from the publicly shared gradients. The leaking only takes few gradient steps to process and can obtain the original training set instead of look-alike alternatives. We name this leakage as\textit {deep leakage from gradient} and practically validate the effectiveness of our algorithm on both computer vision and natural language processing tasks. We empirically show that our attack is much stronger than previous approaches and thereby and raise people's awareness to rethink the gradients' safety. We also discuss some possible strategies to defend this deep leakage.
Total citations	Cited by 1545

Scholar articles
Deep leakage from gradients
L Zhu, Z Liu, S Han - Advances in neural information processing systems, 2019
Cited by 1545 Related articles All 15 versions

Github: https://github.com/mit-han-lab/dlg

Website: https://dlg.hanlab.ai

Lecture Plan

1. Deep leakage from gradients, gradient is not safe to share
2. **Memory bottleneck of on-device training**
3. Tiny transfer learning (TinyTL)
4. Sparse back-propagation (SparseBP)
5. Quantized training with quantization aware scaling (QAS)
6. PockEngine: system support for sparse back-propagation

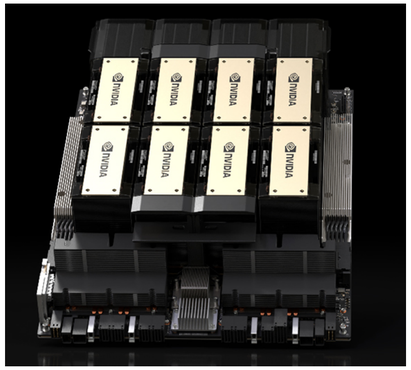
Quantized Training

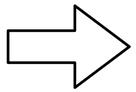
Sparse Training

PockEngine

On-Device Training is Challenging

Memory size is too small to hold DNNs

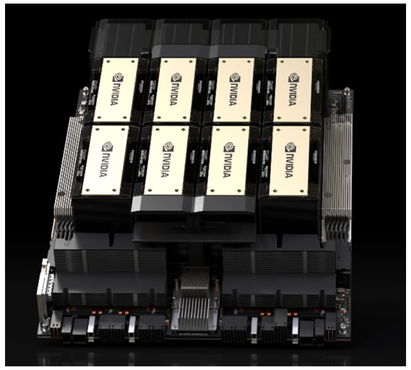


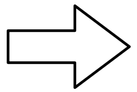


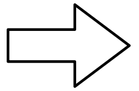
	Cloud AI	Mobile AI
Memory (Activation)	141GB	4GB
Storage (Weights)	~TB/PB	256GB

On-Device Training is Challenging

Memory size is too small to hold DNNs



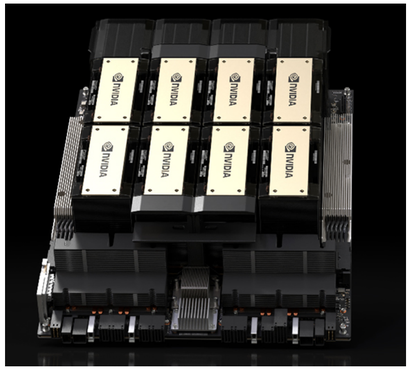


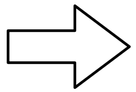


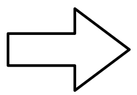
	Cloud AI	Mobile AI	Tiny AI
Memory (Activation)	141GB	4GB	320kB
Storage (Weights)	~TB/PB	256GB	1MB

On-Device Training is Challenging

Memory size is too small to hold DNNs







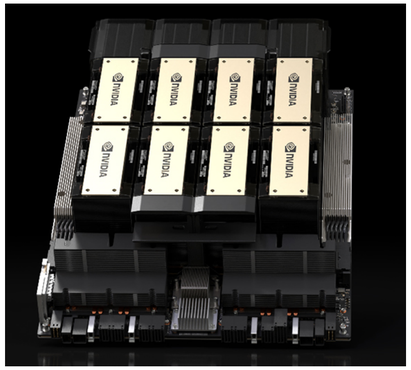
	Cloud AI	Mobile AI	Tiny AI
Memory (Activation)	141GB	4GB	320kB
Storage (Weights)	~TB/PB	256GB	1MB

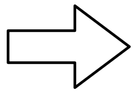
13,000x smaller

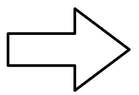
1,000,000x smaller

On-Device Training is Challenging

Memory size is too small to hold DNNs



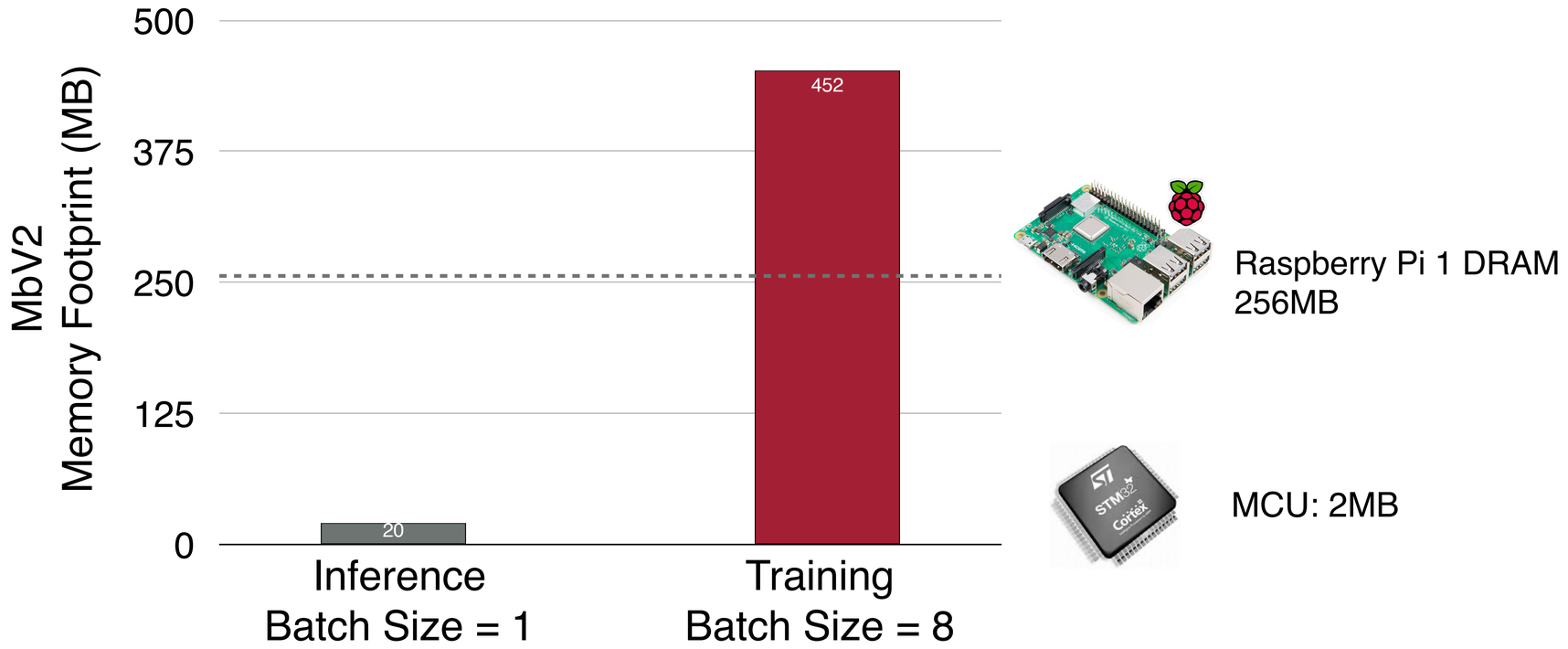




	Cloud AI	Mobile AI	Tiny AI
Memory (Activation)	141GB	4GB	320kB
Storage (Weights)	~TB/PB	256GB	1MB

- 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**

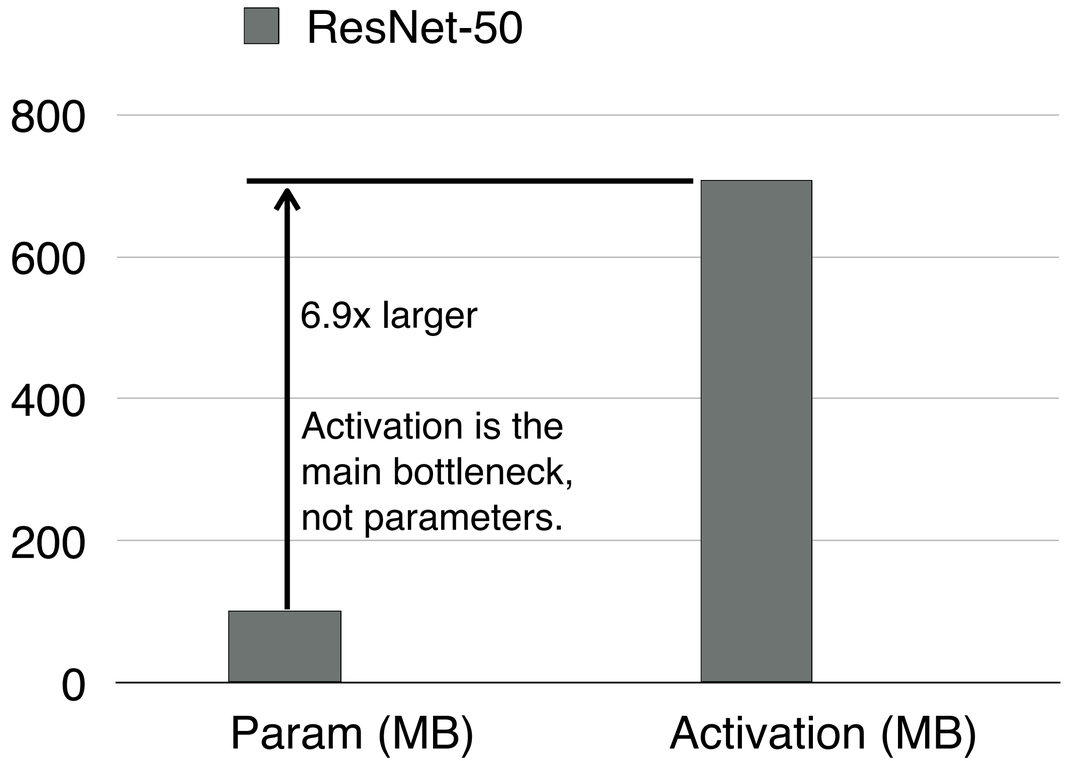
Forward: $\mathbf{a}_{i+1} = \mathbf{a}_i \mathbf{W}_i + \mathbf{b}_i$

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 beyond memory limit of many edge devices.

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

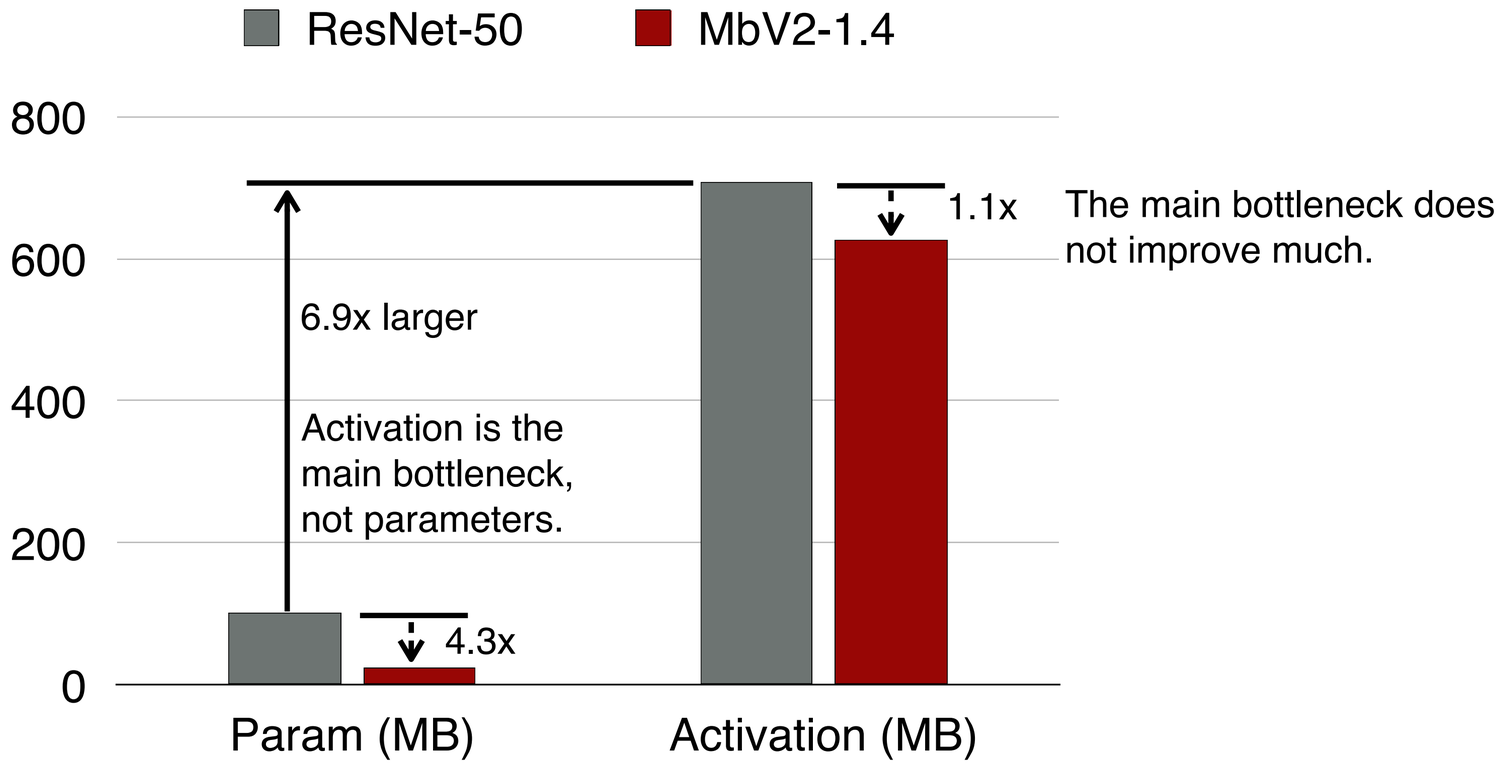
Activation is the Memory Bottleneck in CNNs



- Activation is the main bottleneck for CNN training

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

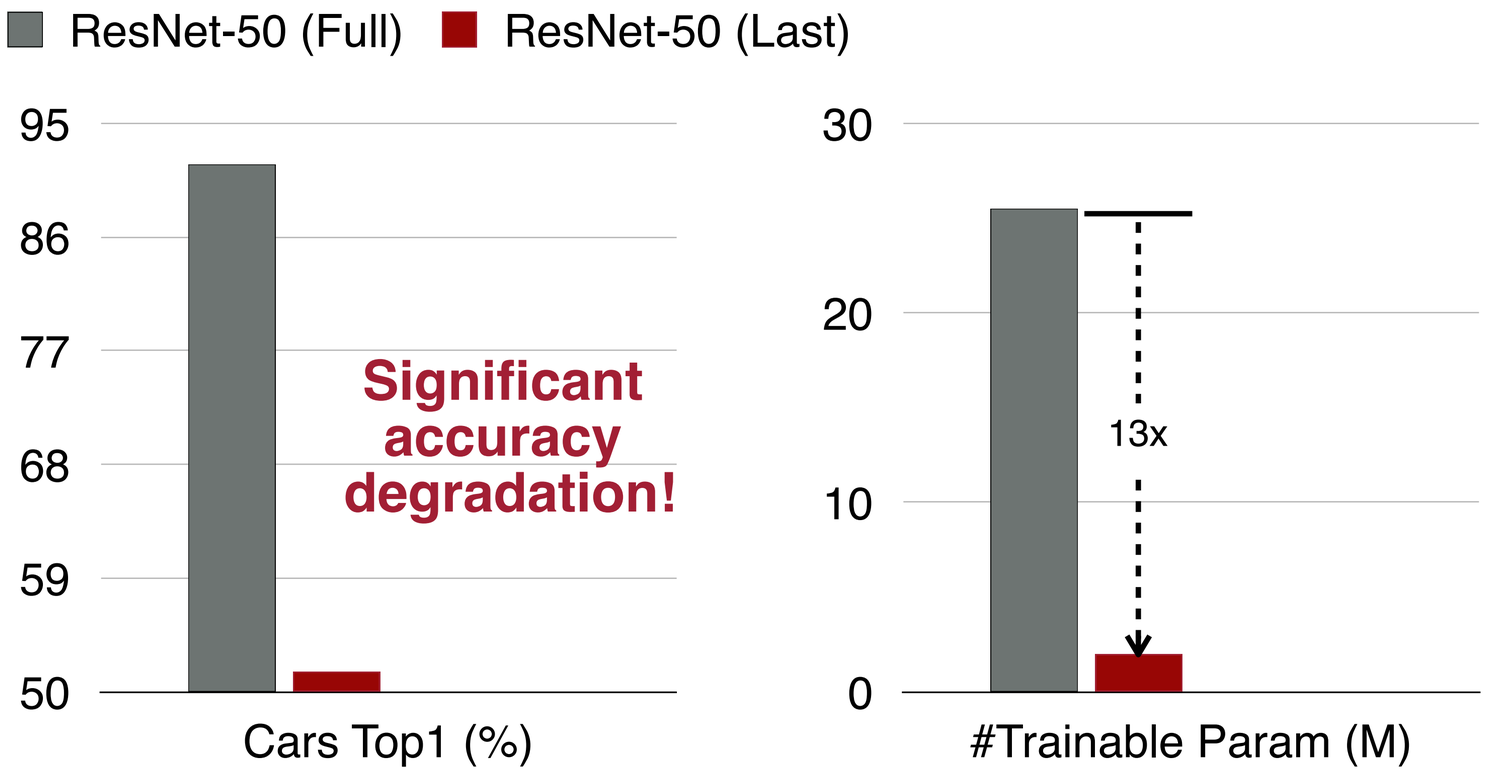
Activation is the Memory Bottleneck in CNNs



- Activation is the main bottleneck for CNN training.
- MobileNets focus on reducing the number of parameters or FLOPs, while the main bottleneck does not improve much.

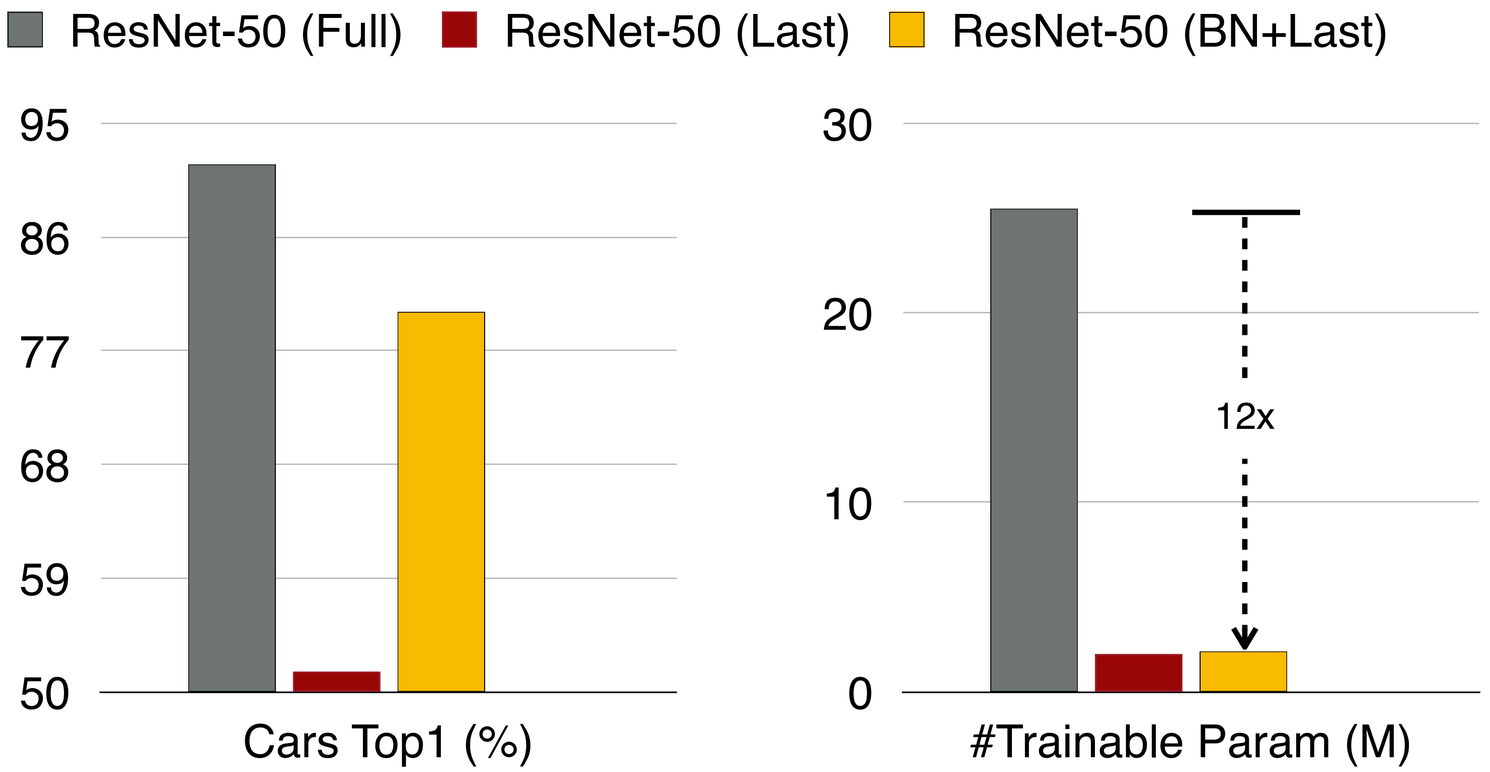
TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning [Cai *et al.*, NeurIPS 2020]

Parameter-Efficient Transfer Learning in CNNs



- 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 in CNNs

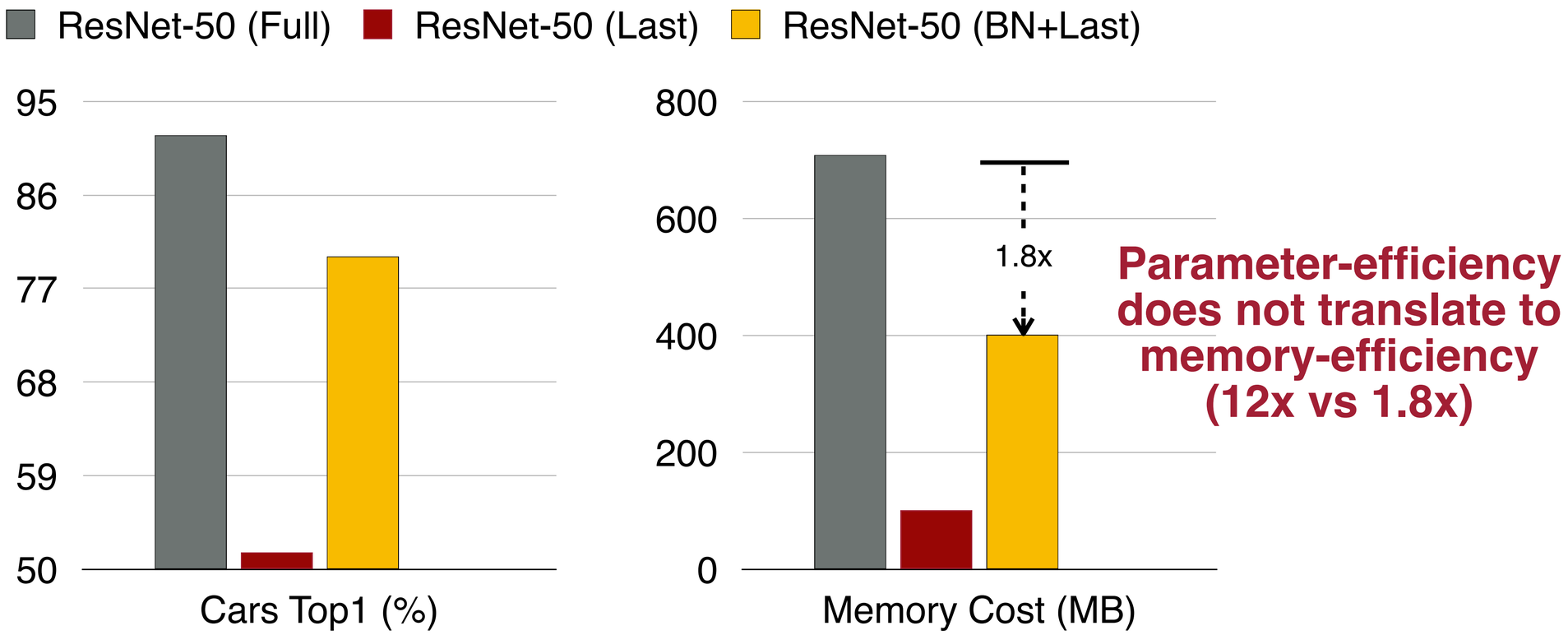


- Full: Fine-tune the full network. Better accuracy but highly inefficient.
- Last: Only fine-tune the last classifier head. Efficient but the capacity is limited.
- BN+Last: Fine-tune the BN layers and the last layer. Parameter-efficient.

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]

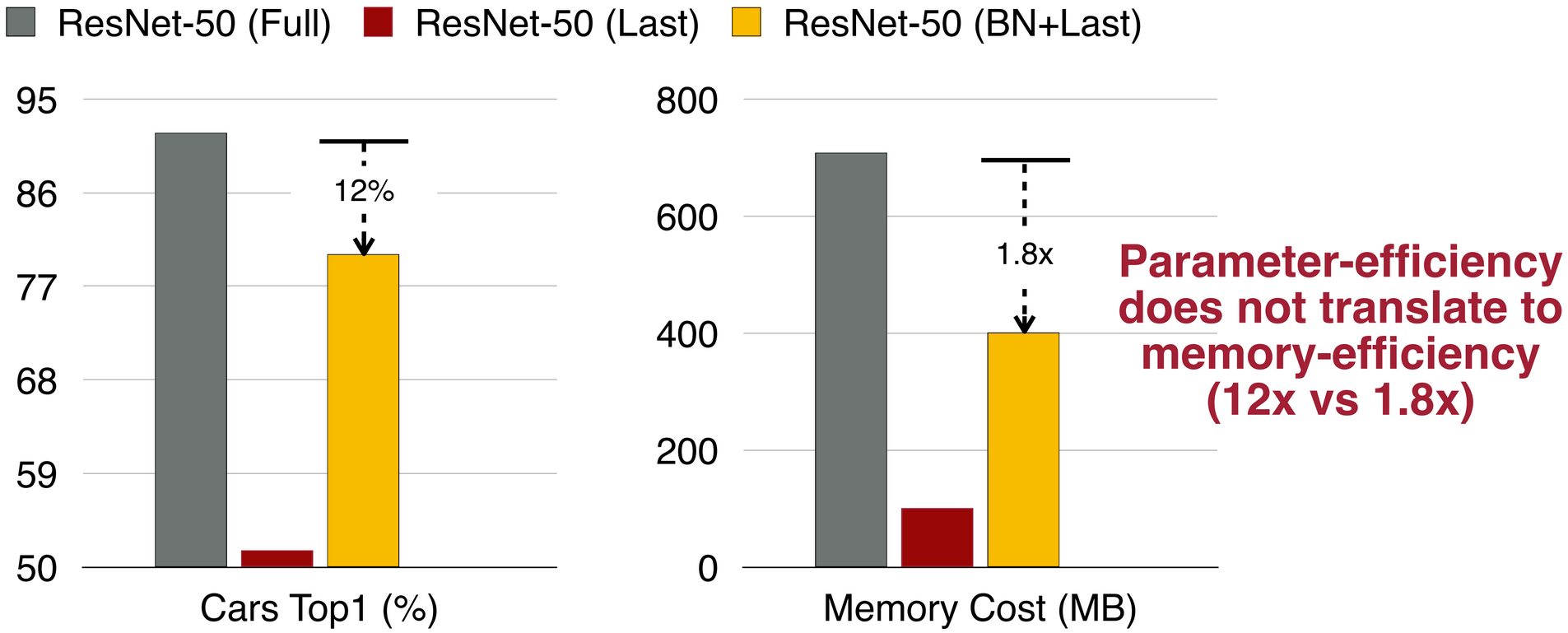
Parameter-Efficient Transfer Learning in CNNs



- 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 in CNNs



Parameter-efficiency does not 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.**

Lecture Plan

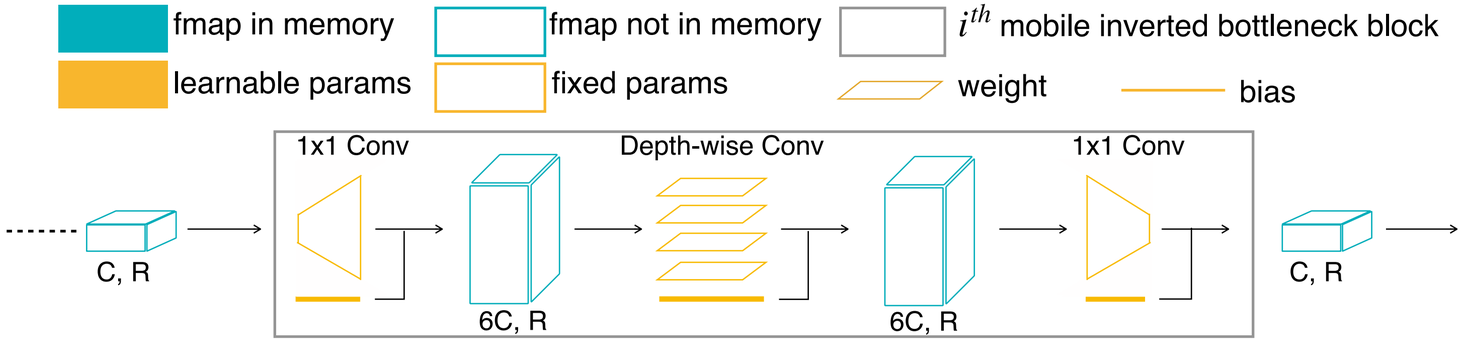
1. Deep leakage from gradients, gradient is not safe to share
2. Memory bottleneck of on-device training
3. **Tiny transfer learning (TinyTL)**
4. Sparse back-propagation (SparseBP)
5. Quantized training with quantization aware scaling (QAS)
6. PockEngine: system support for sparse back-propagation

Quantized Training

Sparse Training

PockEngine

Updating Weights is Memory-Expensive



Fine-tune bias only

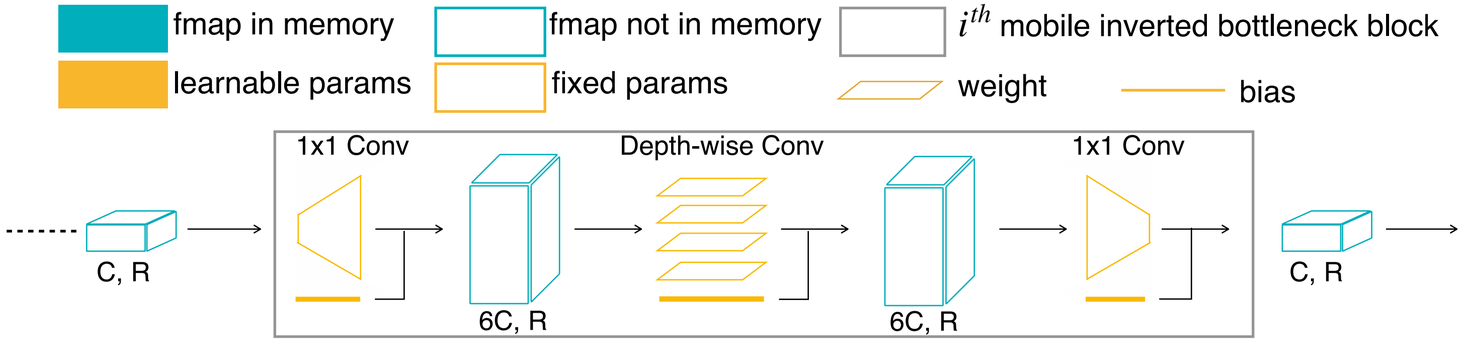
Forward: $\mathbf{a}_{i+1} = \mathbf{a}_i \mathbf{W}_i + \mathbf{b}_i$

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

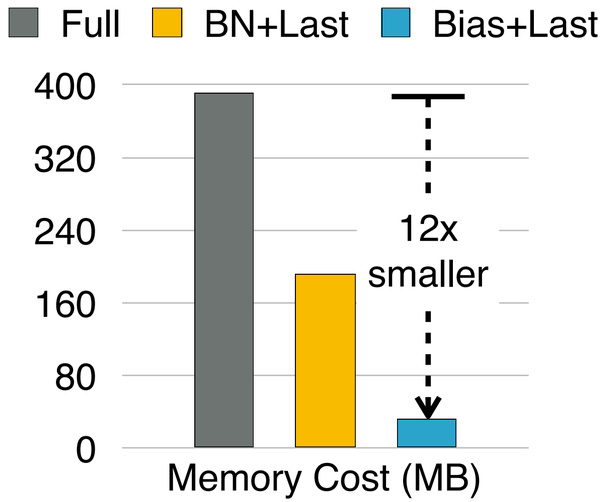
- Updating weights requires storing intermediate activations
- Updating biases does not, **is memory-efficient**

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

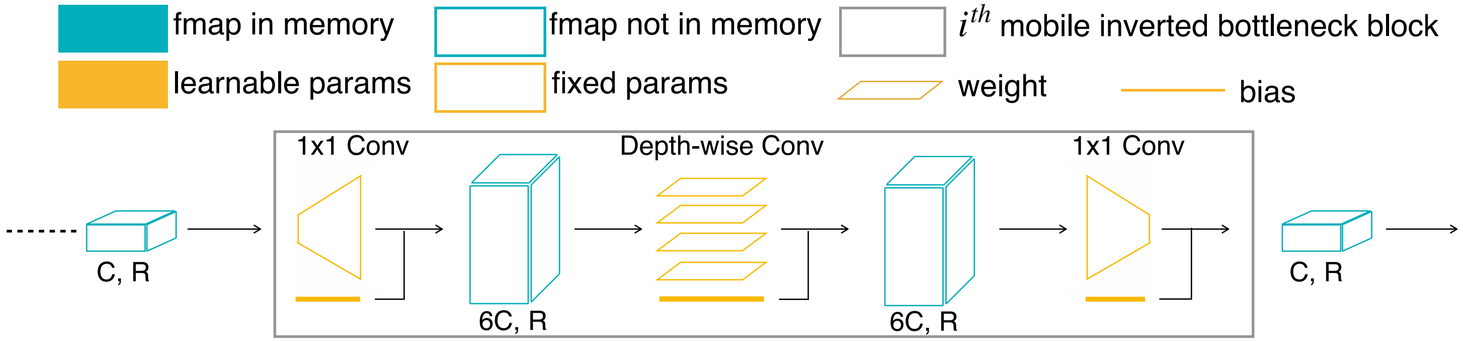


Freeze weights, only fine-tune biases

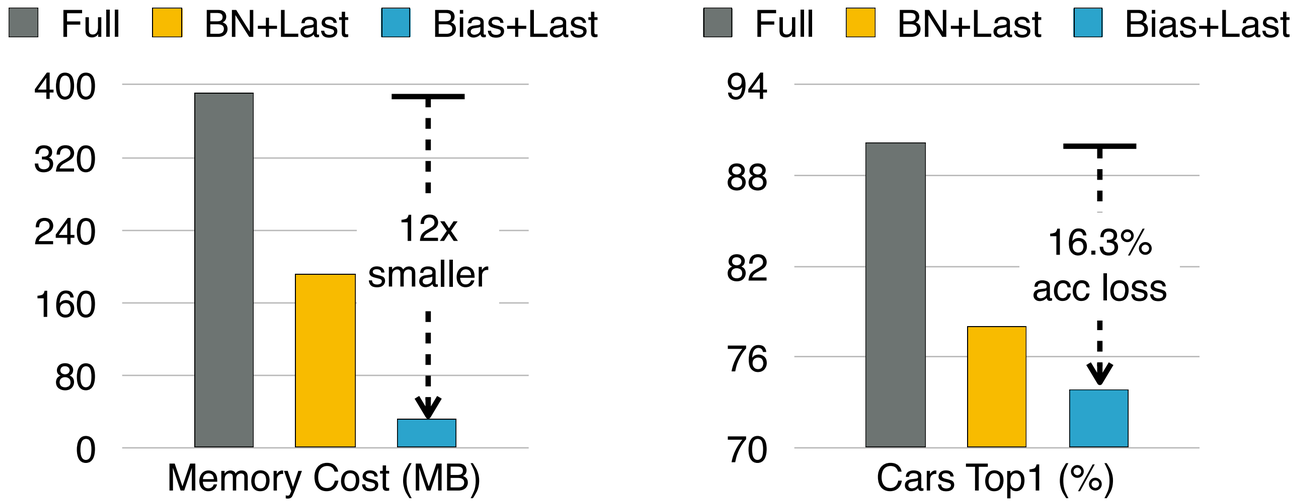
=> save 12x memory

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

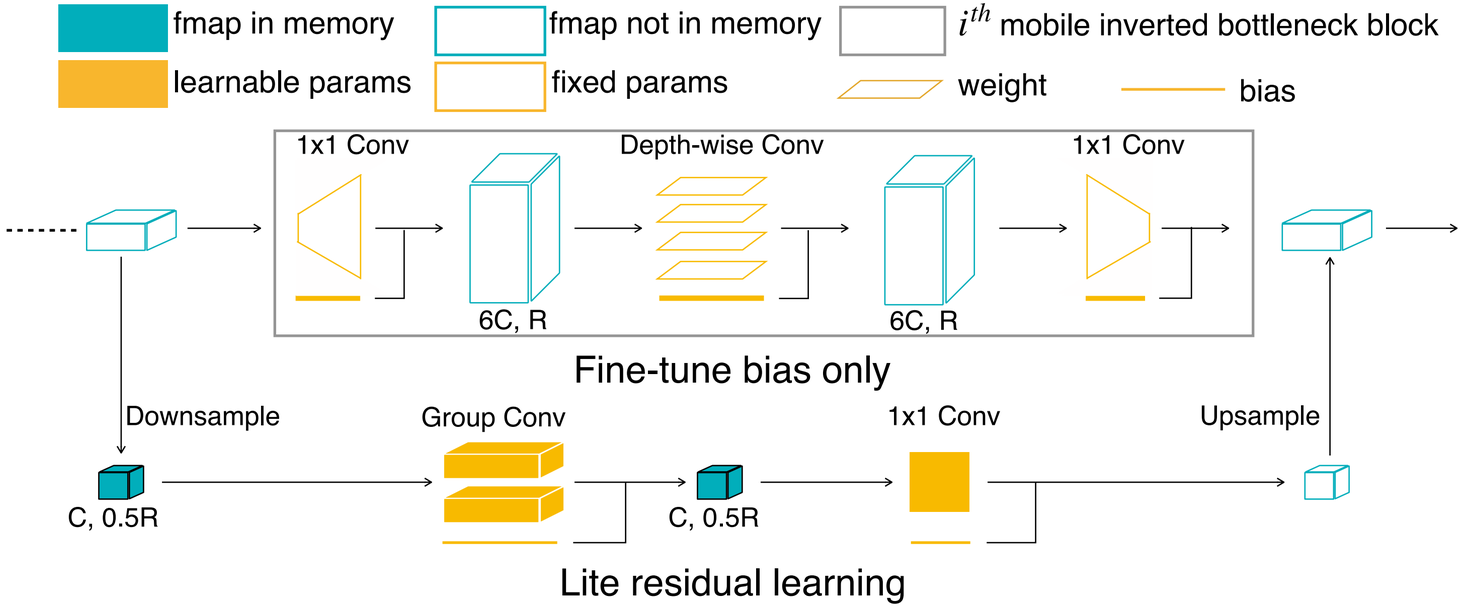


Freeze weights, only fine-tune biases

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

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

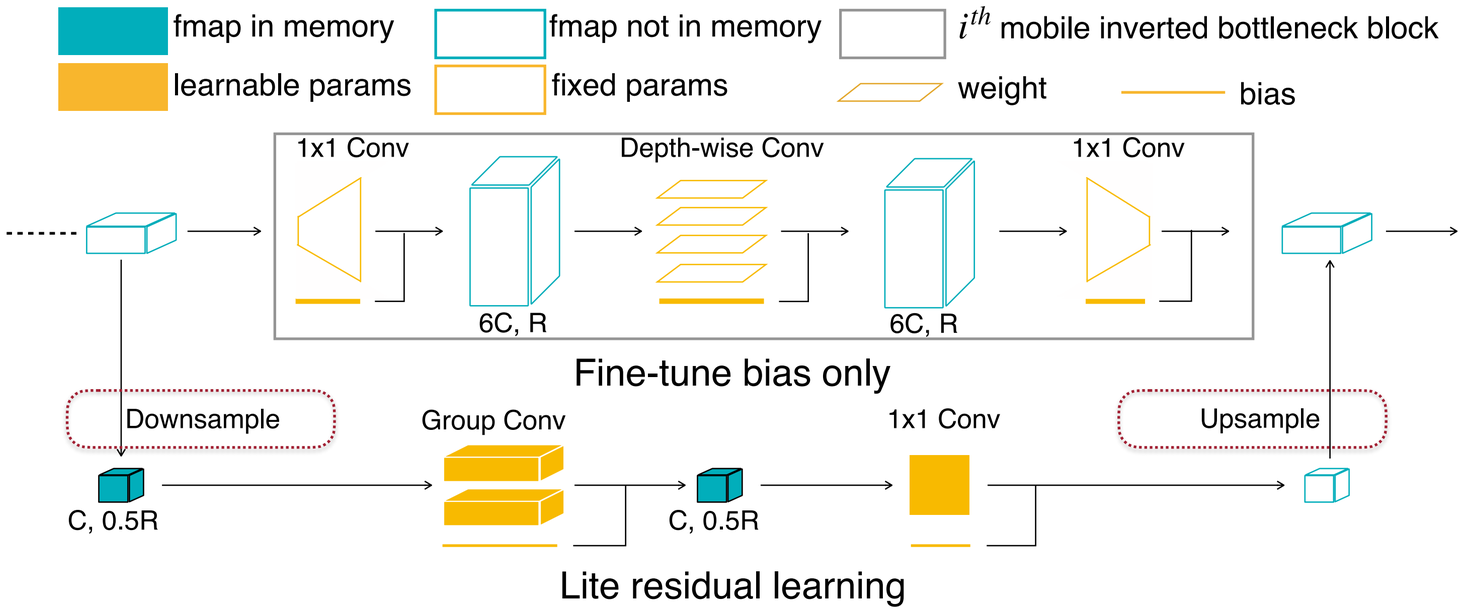
TinyTL: Lite Residual Learning



- 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]

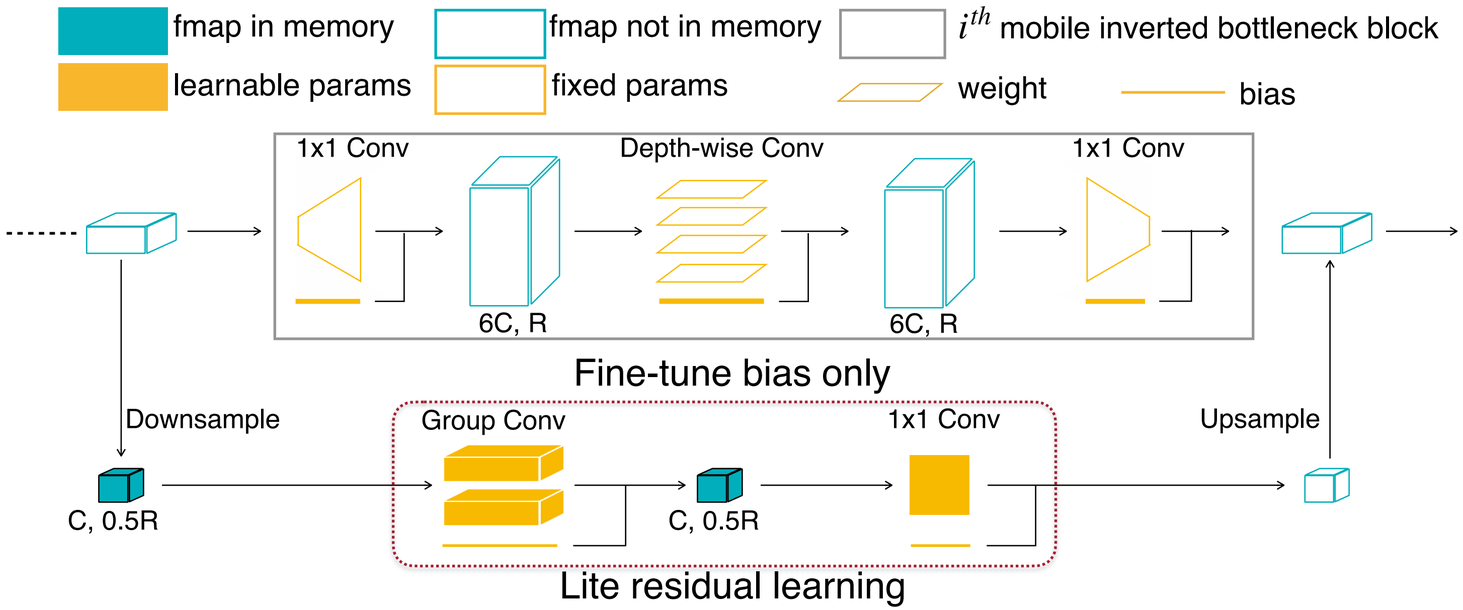
TinyTL: Lite Residual Learning



- 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]

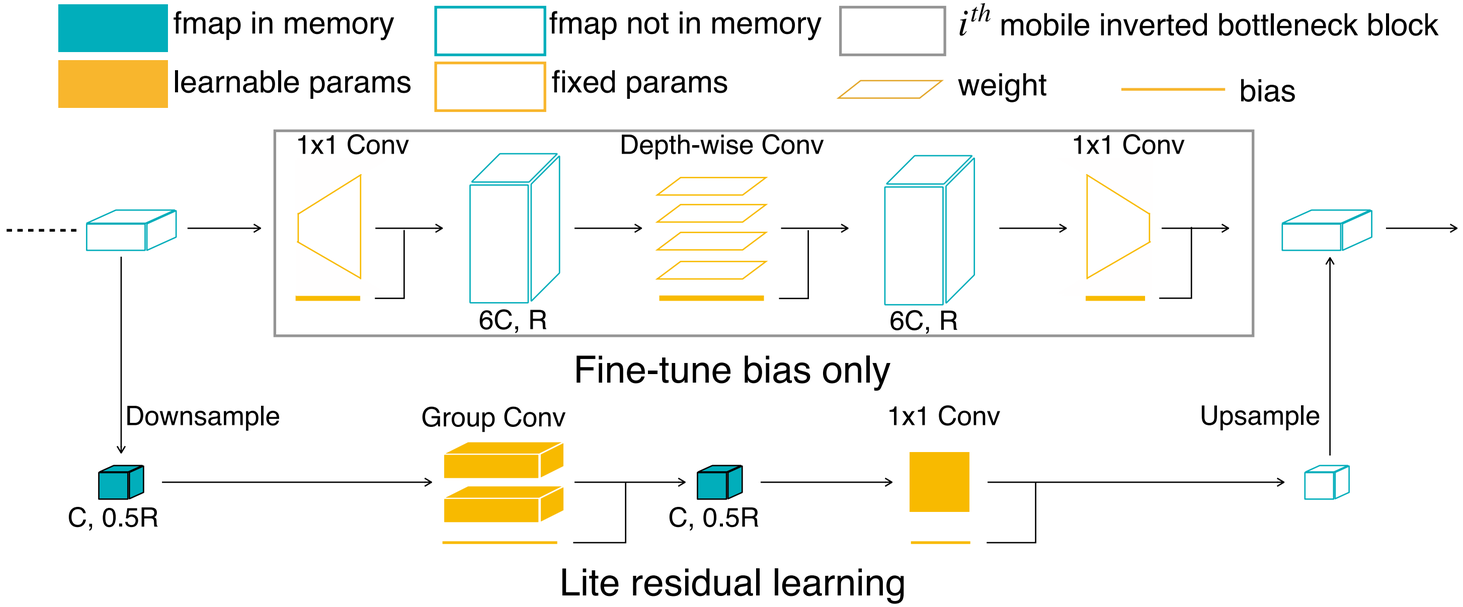
TinyTL: Lite Residual Learning



- 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

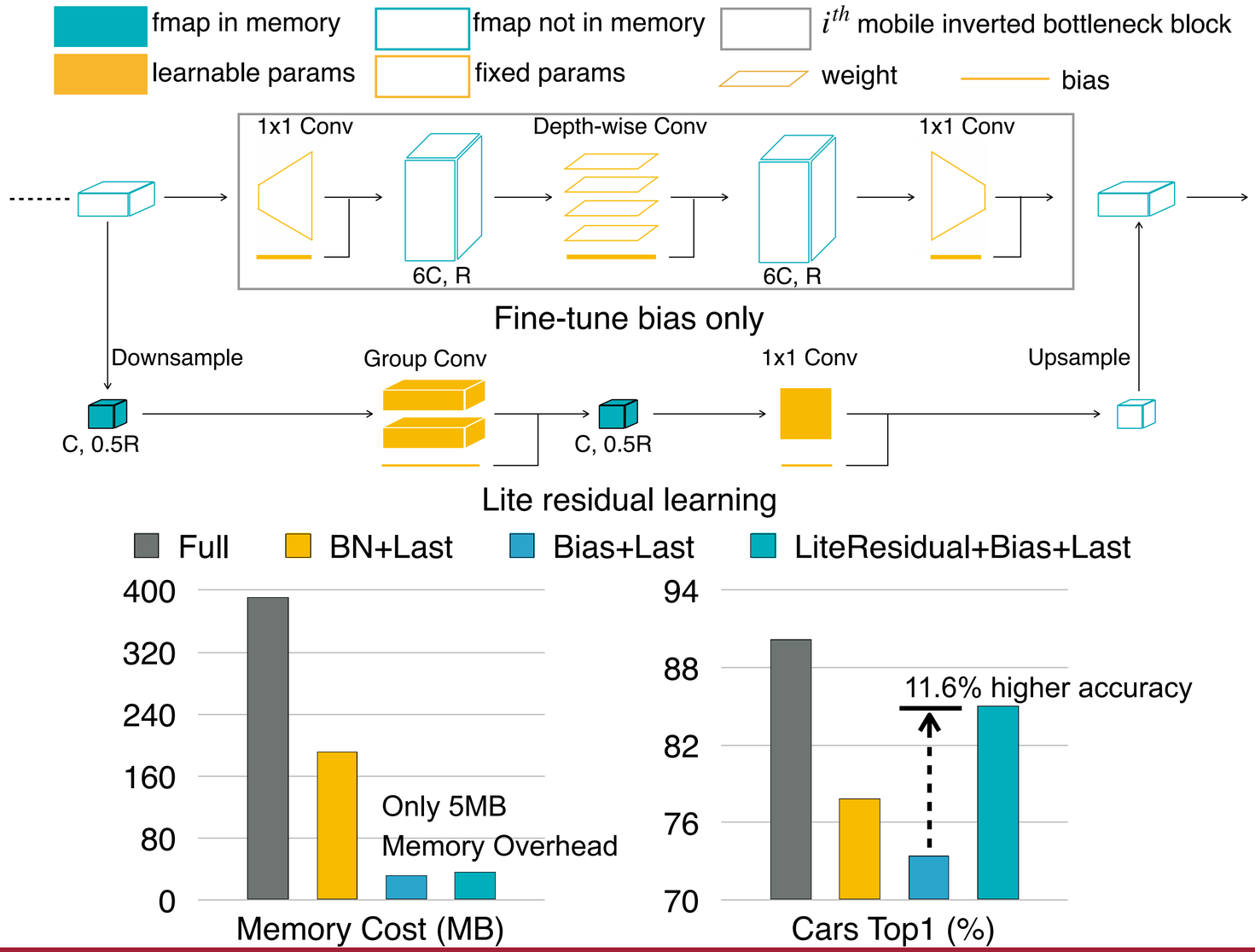


- Add lite residual modules to increase model capacity
- Key principle - keep activation size small
  1. Reduce the resolution
  2. Avoid inverted bottleneck

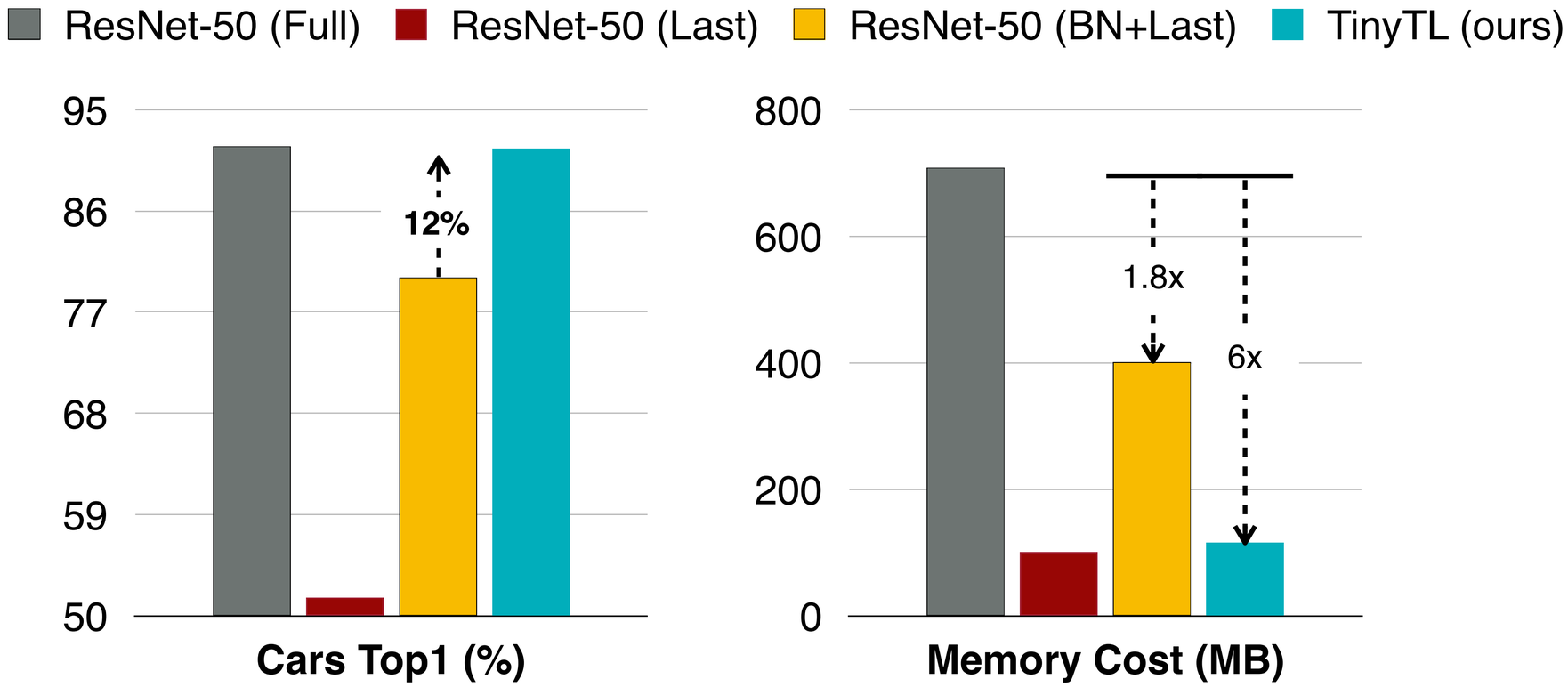
(1/6 channel, 1/2 resolution, 2/3 depth => ~4% activation size)

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

TinyTL: Lite Residual Learning



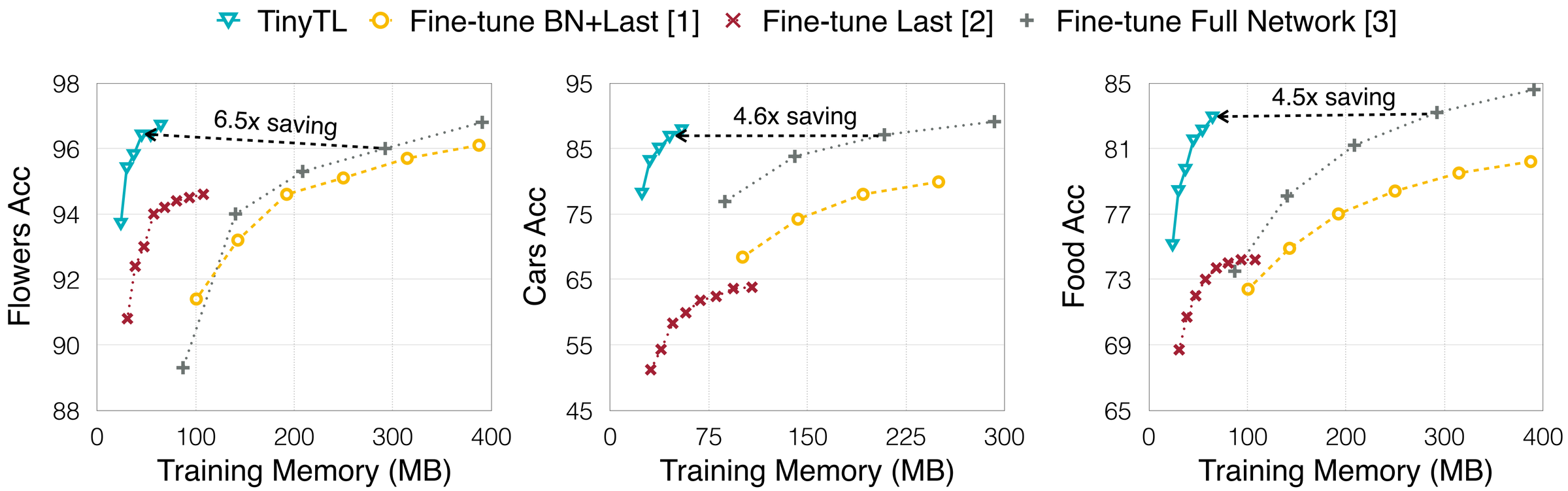
TinyTL: Memory-Efficient Transfer Learning



- 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

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

TinyTL: Up to 6.5x Memory Saving

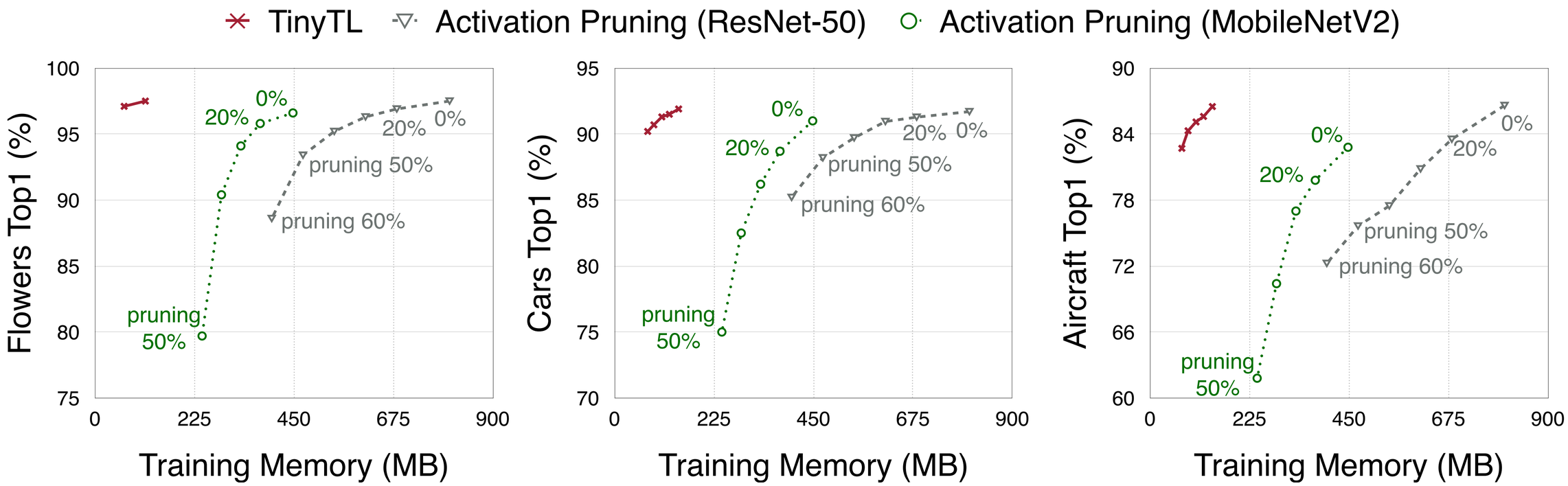


Backbone: ProxylessNAS-Mobile, Scanning over different resolutions

- 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]

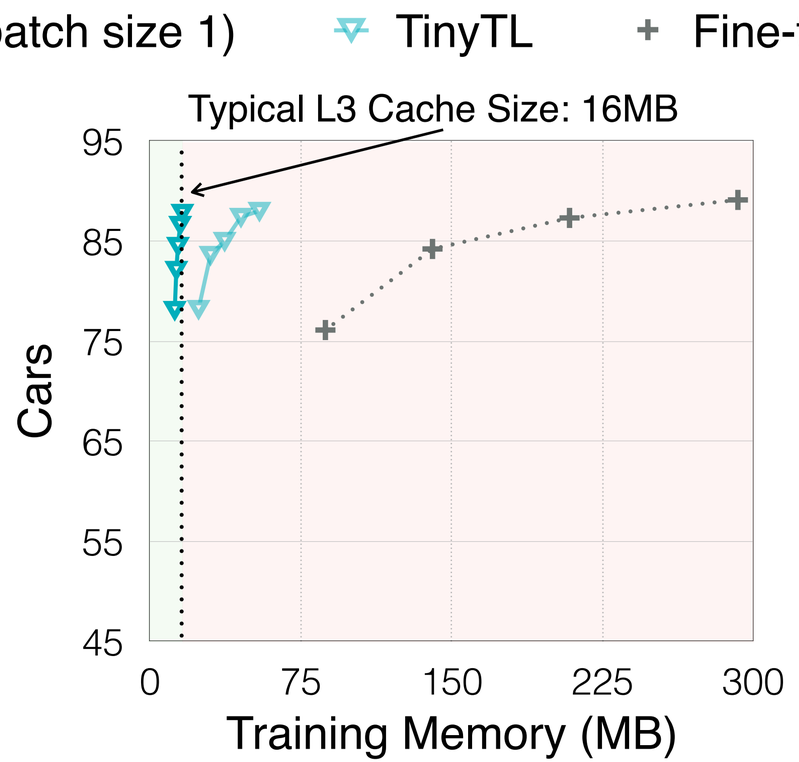
Comparison with Dynamic Activation Pruning



- 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.

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

Lecture Plan

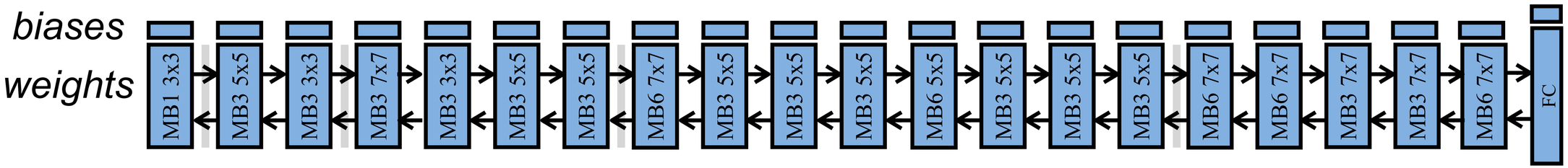
1. Deep leakage from gradients, gradient is not safe to share
2. Memory bottleneck of on-device training
3. Tiny transfer learning (TinyTL)
4. **Sparse back-propagation (SparseBP)**
5. Quantized training with quantization aware scaling (QAS)
6. PockEngine: system support for sparse back-propagation

Quantized Training

Sparse Training

PockEngine

Dense, Full Back-Propagation



Model: ProxylessNAS-Mobile

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

- Need to save all intermediate activations (quite large)

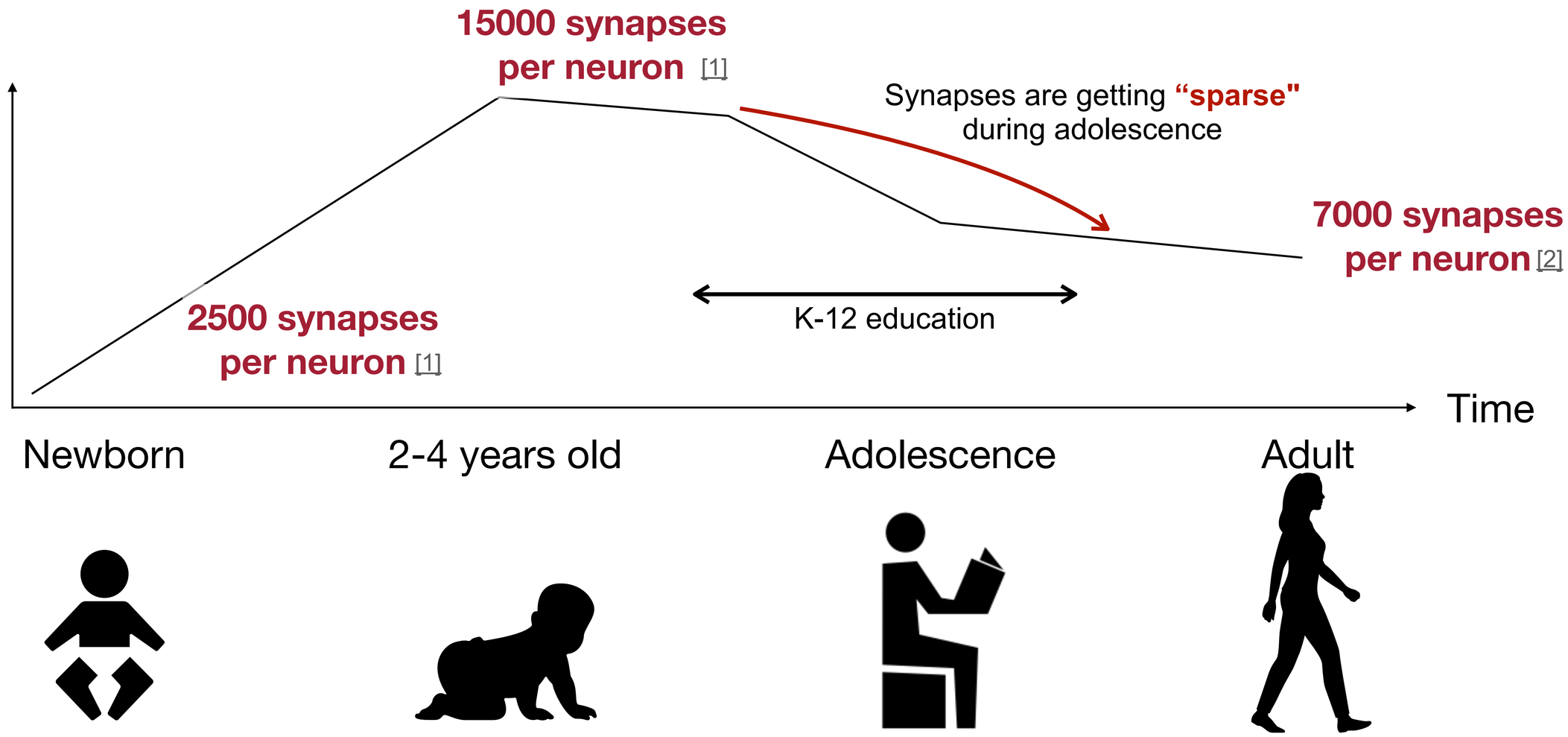
Forward: $\mathbf{a}_{i+1} = \mathbf{a}_i \mathbf{W}_i + \mathbf{b}_i$

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

- Inference does not need to store activations, training does.
- Activations grows linearly with batch size.

TinyTL: Reduce Memory, Not Parameters for Efficient On-Device Learning [Cai et al, NeurIPS 2020]

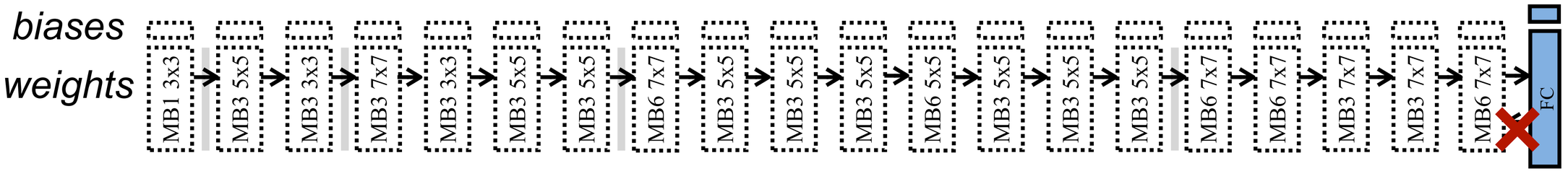
Sparse Learning



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

Slide Inspiration: Alila Medical Media

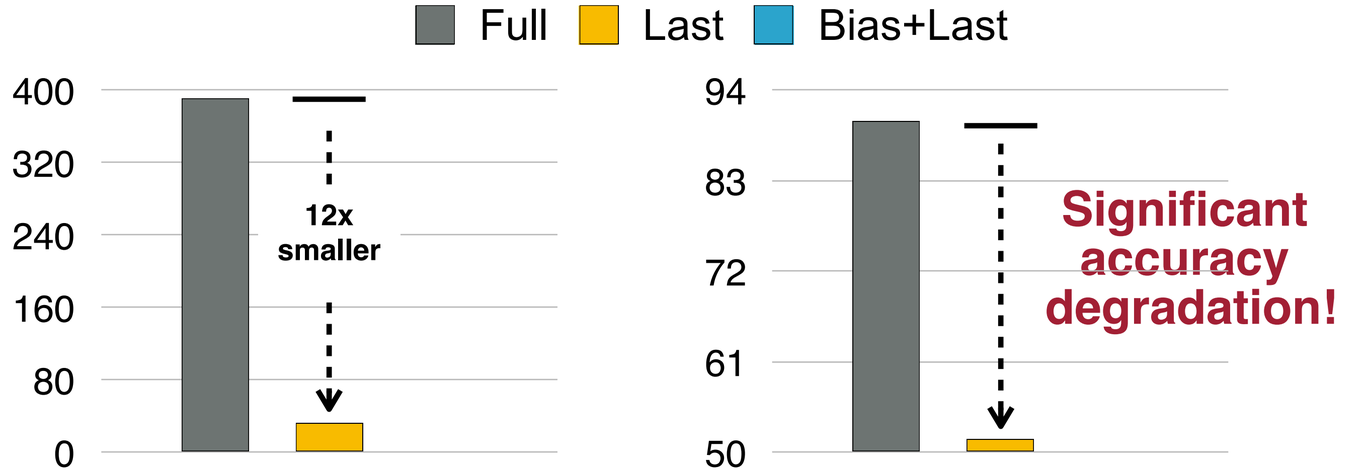
Last-Layer-Only Back-Propagation



Model: ProxylessNAS-Mobile

Updating only the last layer is cheap

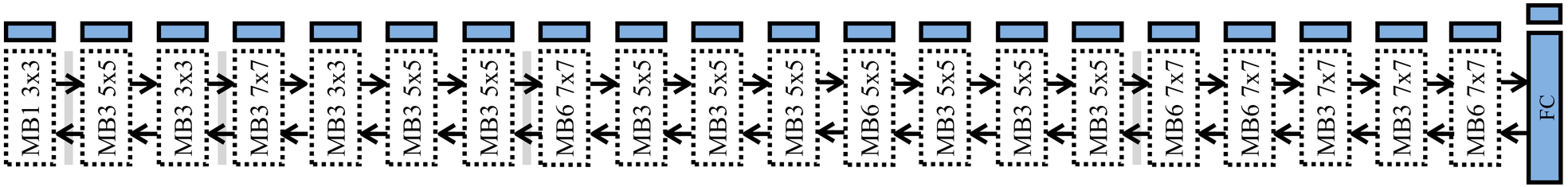
- No need to back propagate to previous layers
- But, accuracy drops significantly



TinyTL: Reduce Memory, Not Parameters for Efficient On-Device Learning [Cai et al, NeurIPS 2020]

Bias-Only Back-Propagation

LoRA is a special case



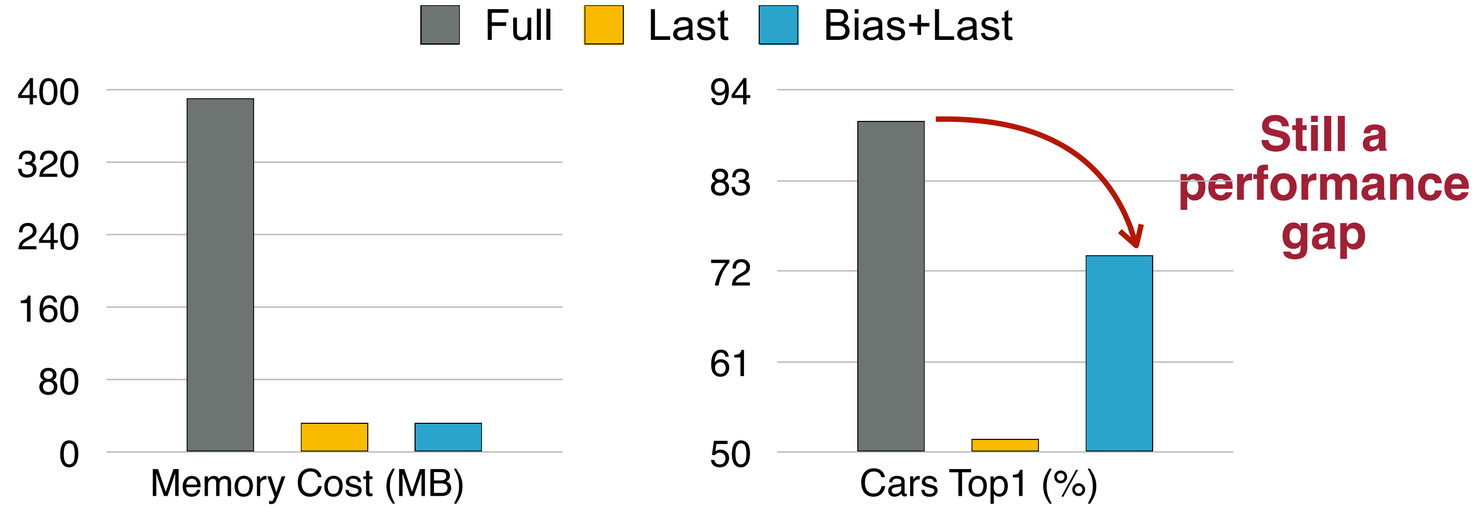
Model: ProxylessNAS-Mobile

Updating the only the bias part

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

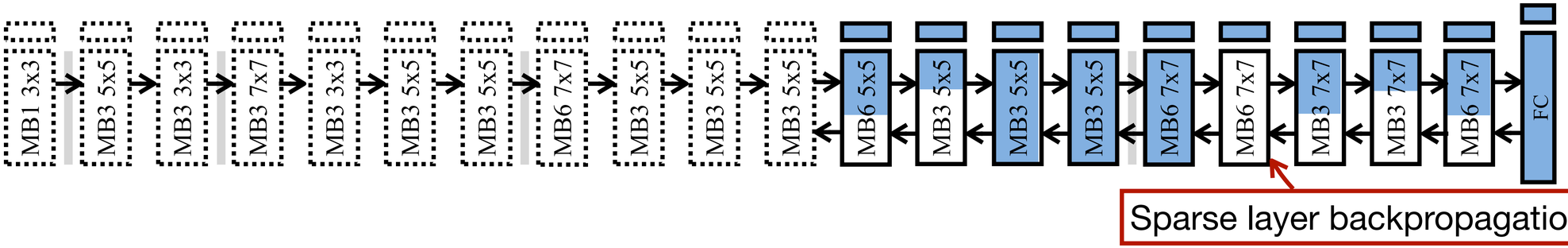
$$d\mathbf{W} = f(\mathbf{X}, d\mathbf{Y})$$

$$d\mathbf{b} = f(d\mathbf{Y})$$



TinyTL: Reduce Memory, Not Parameters for Efficient On-Device Learning [Cai et al, NeurIPS 2020]

Sparse Back-Propagation

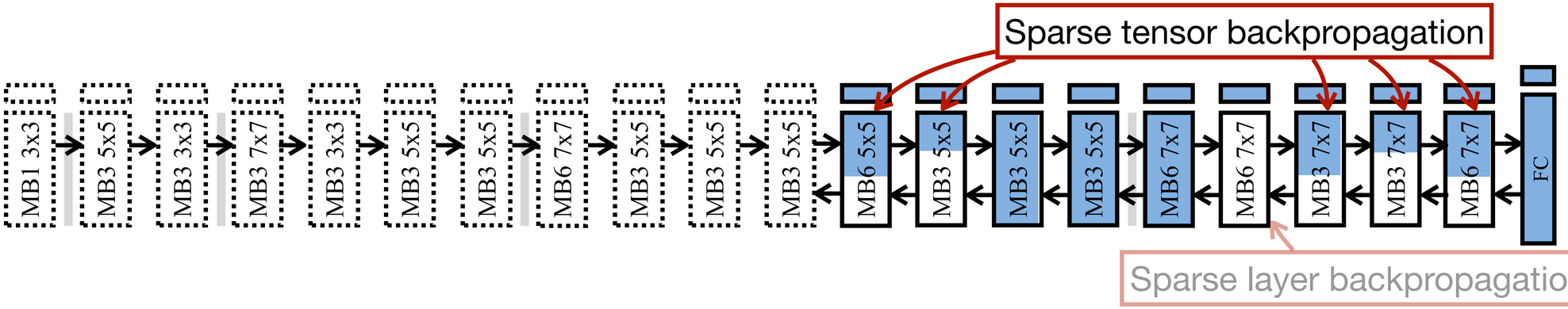


Model: ProxylessNAS-Mobile

Use sparse back-propagation to train the model

- Some **layers are not as important as** others

Sparse Back-Propagation

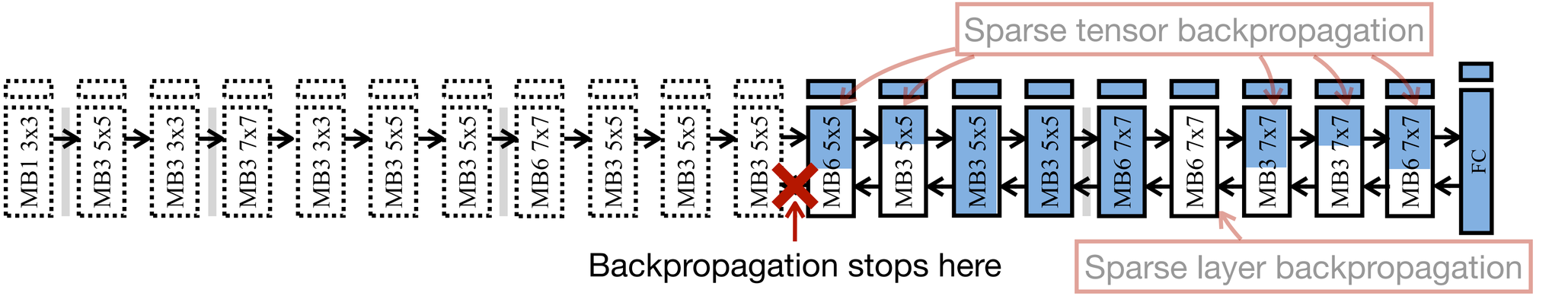


Model: ProxylessNAS-Mobile

Use sparse back-propagation to train the model

- Some layers are not as important as others
- Some **channels are not as important as** others

Sparse Back-Propagation

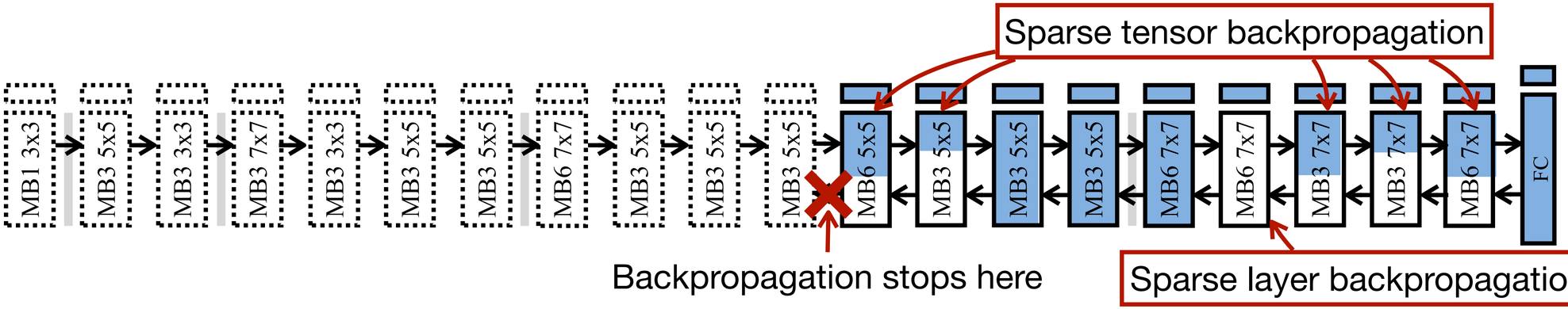


Model: ProxylessNAS-Mobile

Use sparse back-propagation to train the model

- Some layers are not as important as others
- Some channels are not as important as others
- **No need to back-propagate to the early layers**

Sparse Back-Propagation



Model: ProxylessNAS-Mobile

Use sparse back-propagation to train the model

- Some layers are not as important as others
- Some channels are not as important as others
- No need to back-propagate to the early layers
- **Only need to store and compute on a subset of the activations.**

Activation to store: (N, M)

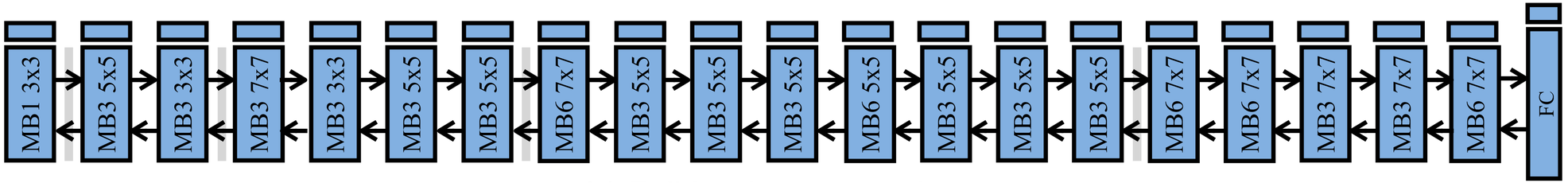
FLOPs: (M * H * N)

Reduce by 4x

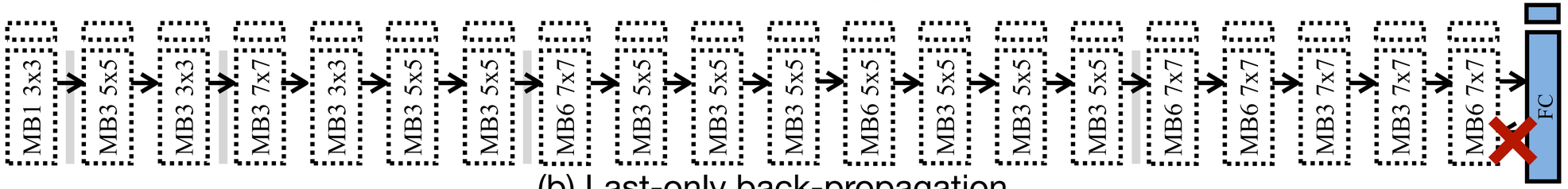
Activation to store: (N, 0.25 * M)

FLOPs: (0.25 * M * H * N)

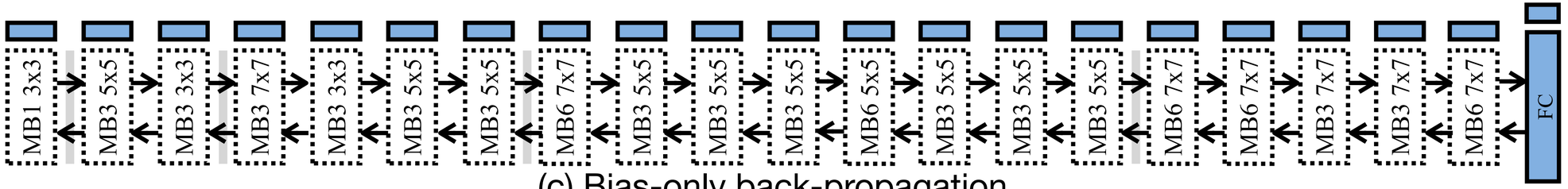
Comparison



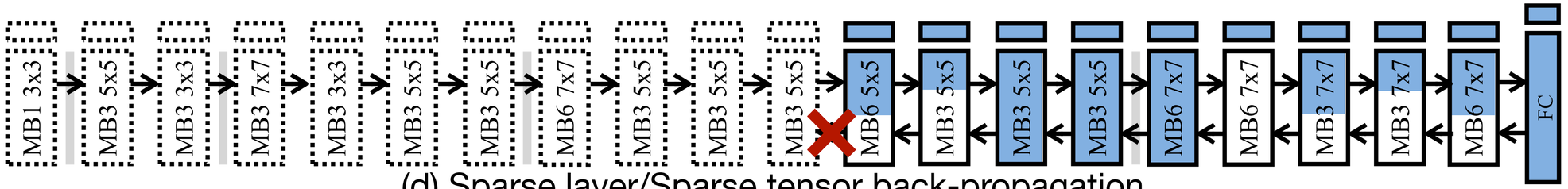
(a) Full back-propagation



(b) Last-only back-propagation



(c) Bias-only back-propagation

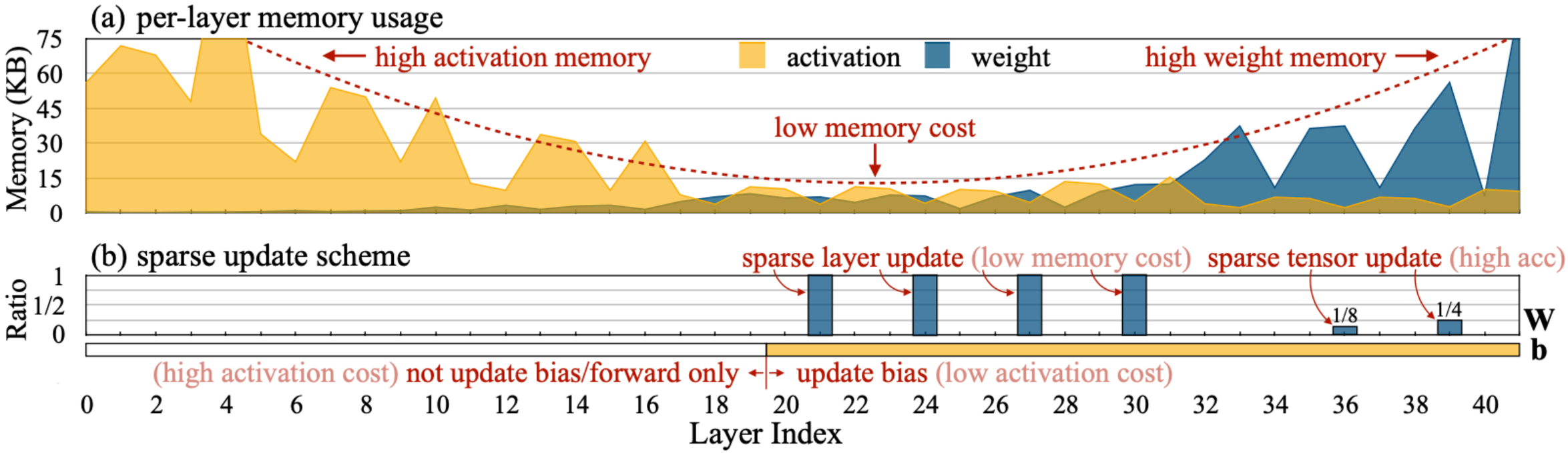


(d) Sparse layer/Sparse tensor back-propagation

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)

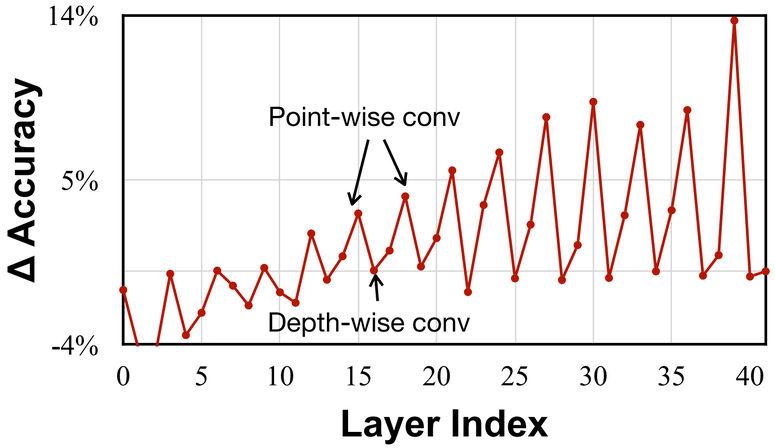


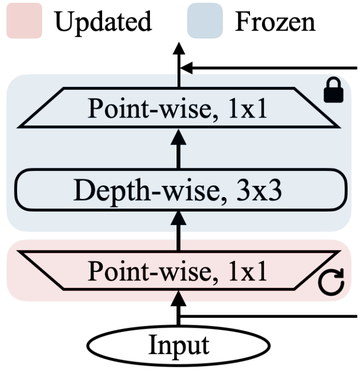
Contribution Analysis

Which layer to update?

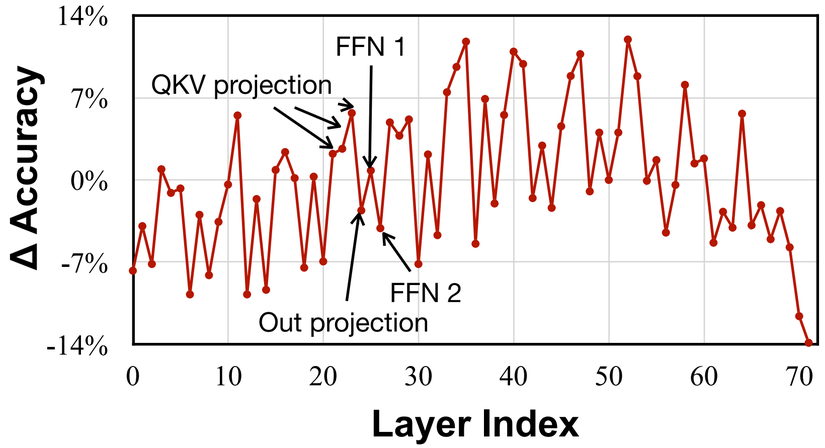
- Contribution Analysis: fine-tune only one layer on a downstream task to measure the accuracy improvement (Δaccuracy) as contributions.
- Only fine-tune the **layers with large Δaccuracy** (contributes more to performance)

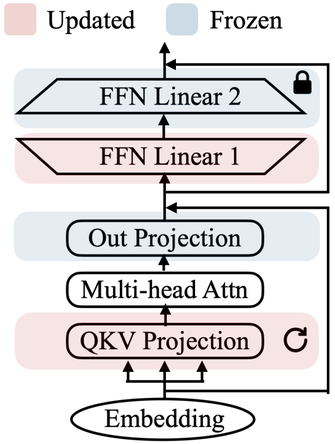
CNN model (MobileNetV2)





Transformers (BERT)



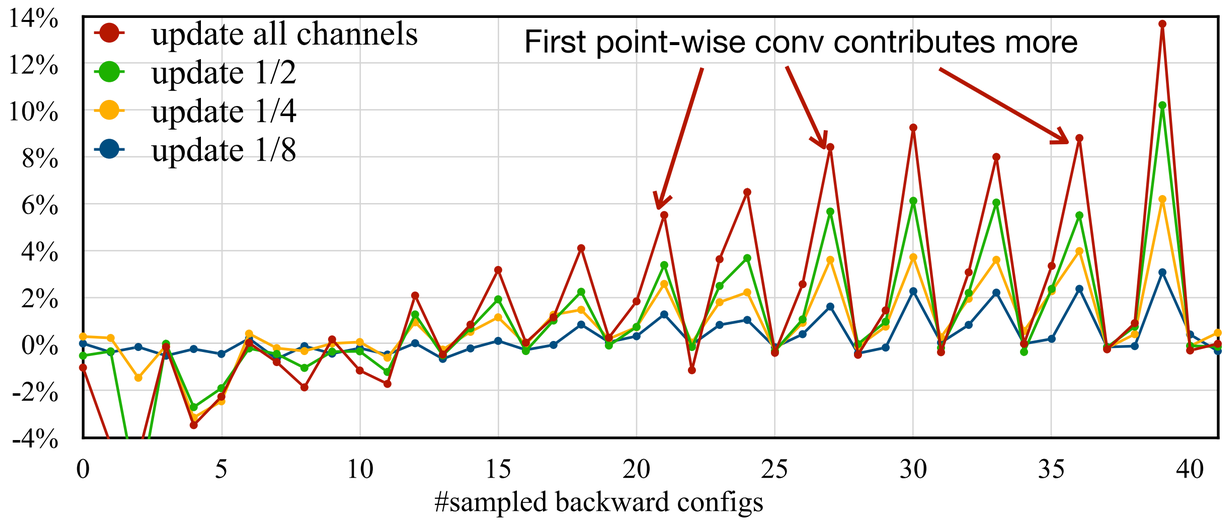


Different models prefer *different* layers for fine-tuning

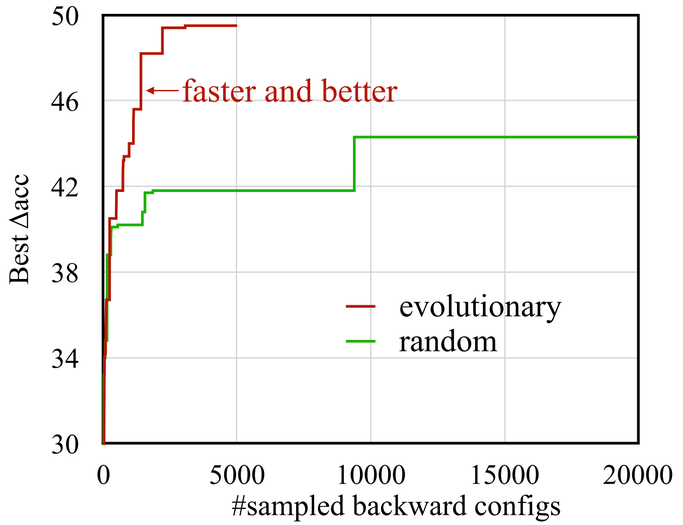
- MobilenetV2 prefers **first depth-wise conv**.
- BERT prefers **QKV projection** and **first FFN** layers.

Contribution Analysis

Which layer to update?



(a) Contribution analysis



(a) Evolution Search

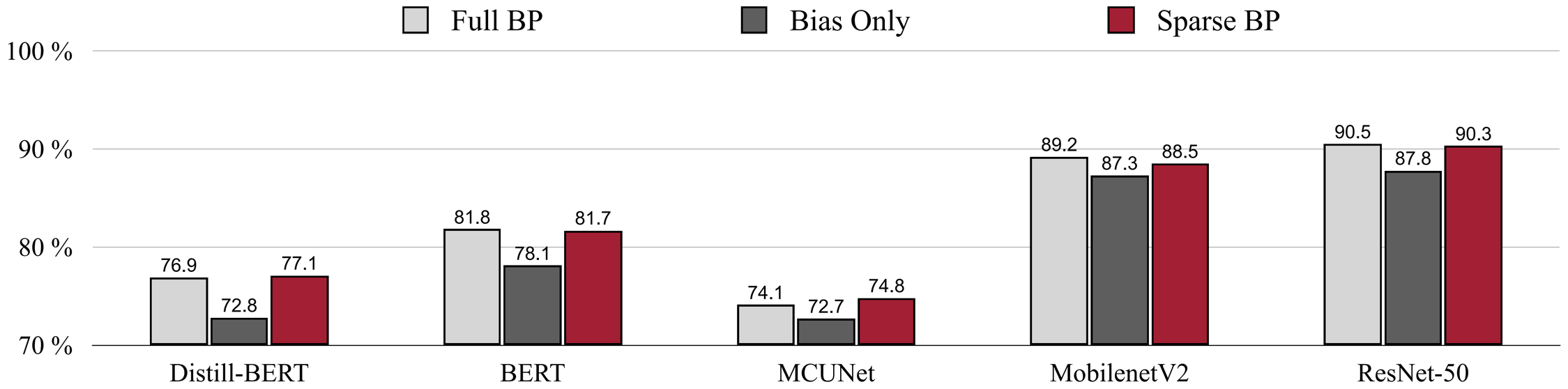
- Use **evolutionary search** to find the sparse back-propagation scheme.

$$k^*, \mathbf{i}^*, \mathbf{r}^* = \max_{k,\mathbf{i},\mathbf{r}} (\Delta \text{acc}_{\mathbf{b}[:k]} + \sum_{i \in \mathbf{i}, r \in \mathbf{r}} \Delta \text{acc}_{\mathbf{W}i,r}) \quad \text{s.t. Memory}(k, \mathbf{i}, \mathbf{r}) \leq \text{constraint}$$

- Thus we can train the model on the edge with low memory cost while achieving high accuracy.

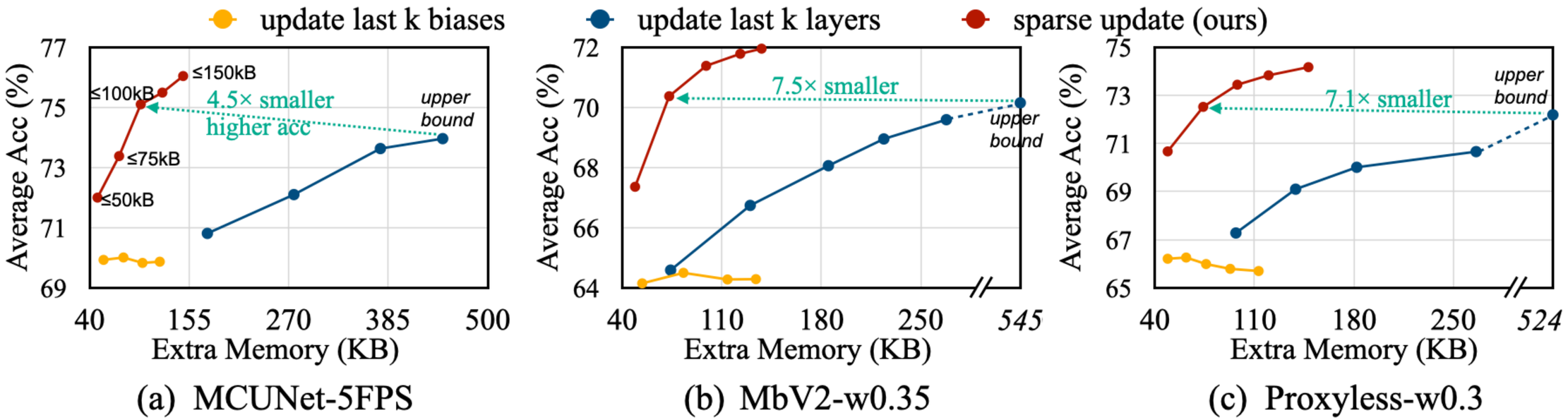
Accuracy of Sparse Back-Propagation

Well maintains the accuracy



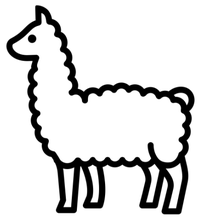
- The accuracy on DistillBERT and BERT is average from GLUE Benchmark.
- The accuracy on MCUNet, MobilenetV2, ResNet-50 is average from TinyTL Benchmark.
- Sparse-BP demonstrates **on-par performance with Full-BP** on both vision and language tasks.

Sparse BP: Lower Memory, Higher Accuracy



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

Sparse BP: Fine-tune LLM On-Device



Model: Meta/Llama-V2

Size: 7B

Instruction: What is the meaning of the following idiom?

Input: It's raining cats and dogs.
Output: The idiom "it's raining cats and dogs" means that it is raining heavily.

Example from Alpaca Dataset.

Dataset: Stanford/Alpaca

Size: 52K Instructions

Instruction: Translate the following phrase into French.

Input: I miss you.
Output: Je te manque.

Example from Alpaca Dataset.

- Although LLM's knowledge and capabilities are learnt mostly during pre-training, fine-tuning / alignment are crucial, teaching LLMs how to interact with users [1, 2].
- Thus, instruction tuning aims to **equip the model with the interaction ability** by providing examples.

[1] LIMA: Less Is More for Alignment
[2] A few more examples may be worth billions of parameters

Sparse BP: Fine-tune LLM On-Device

Fine-tuning Llama2-7B on edge GPU (Jetson Orin)

Framework	Method	Iteration Latency (↓)	GPU Memory(↓)	Loss(↓)	Alpaca-Eval Winrate(↑)	MT-Bench score(↑)
PyTorch	FT-Full	7.7s	45.1GB	0.761	44.1%	6.1
PyTorch	LoRA (rank=8)	7.3s	30.9GB	0.801	43.1%	5.1
PockEngine	FT-Full	1.8s	43.1GB	0.768	43.7%	6.1
PockEngine	Sparse	0.9s	31.2GB	0.779	43.1%	5.7

- Sparse-BP achieves **same accuracy as LoRA**, and **comparable accuracy as full-BP**.
- Sparse BP improves the training speed (1.8s -> 0.9s) compared to dense BP.

On-Device Training Under 256KB Memory [Lin et al, NeurIPS 2022]

Sparse BP: Fine-tune LLM On-Device

Qualitative analysis

Input: What is the capital of United Status?
LlamaV2-7B-original: Washington, D.C.
LlamaV2-7B-tuned (Full-BP): The capital of the United States is Washington D.C. (District of Columbia).

LlamaV2-7B-tuned (Sparse-BP): The capital of the United States is Washington, D.C.

Input: what was Michael Jackson's last album?
LlamaV2-7B-original: Michael Jackson **passed away on June 25, 2020**, so he did not release an album after that. However, before his death, he released the album "**Michael Jackson: The Ultimate Collection**," which was a compilation of his previous studio albums and bonus tracks.
LlamaV2-7B-tuned (Full-BP): Michael Jackson's last album was '**Invincible**', released in 2001.
LlamaV2-7B-tuned (Sparse-BP): Michael Jackson's last al- bum, released during his lifetime, was "**Invincible**" which was released in 2001.

For simple factual questions (*Left*), all models consistently deliver accurate responses while the tuned models also offer more comprehensive information, enriching the overall user experience.

For obscure questions (*Right*), the original Llama model no longer provides accurate answers while Full-BP and Sparse-BP models generate the proper response.

Sparse BP: Fine-tune LLM On-Device

Qualitative analysis - Reasoning task

Input: Micro was running virtually online in 2021, and held physically at Chicago in 2022 and will soon be hosted in Toronto in 2023. Now is the year of 2023, where was Micro hosted last year?

LlamaV2-7B-original: Micro was hosted **virtually in 2021**.

LlamaV2-7B-tuned (Full-BP): Micro was hosted physically in **Chicago in 2022**.

LlamaV2-7B-tuned (Sparse-BP): Based on the information provided, it seems that Micro was hosted physically at **Chicago in 2022**.

Input: Please reverse the words in the sentence "I love the Micro conference”

LlamaV2-7B-original: I **hate** the conference Micro.

LlamaV2-7B-tuned (Full-BP): The reversed result is "**Conference Micro the love I**"

LlamaV2-7B-tuned (Sparse-BP): The reversed sentence would be "**Conference the Micro love I**".

- If **not fine-tuned**, then LLMs will **mis-interoperate the question** and performs wrong actions.
- *Left*: Without fine-tuning, the model failed to understand and only simply repeats.
- *Right*: The original model misunderstood “reverse”, leading to incorrect answers

Sparse BP: Fine-tune LLM On-Device

Qualitative analysis - Math task

Input: Lee mows one lawn and charges $33. Last week he mowed 16 lawns and three customers each gave him a $10 tip. How many dollars did Lee earn mowing lawns last week?

LlamaV2-7B-original: Lee **earned $660 from mowing** lawns last week.

LlamaV2-7B-tuned (Full-BP): Lee earned $33 x 16 = $528 from mowing lawns. His total earnings from the 16 customers who gave him a $10 tip is $10 x 3 = $30. Therefore, **Lee earned a total of $528 + $30 = $558 last week** mowing lawns.

LlamaV2-7B-tuned (Sparse-BP): To determine how much Lee earned mowing lawns last week, we need to calculate the total amount of money he earned from mowing lawns and tips. Lee mowed one lawn and charged $33, so he earned $33 from that lawn. Last week, Lee mowed 16 lawns, and three customers each gave him a $10 tip, so he earned $10 x 3 = $30 in tips from those lawns and earned $33 from mowing one lawn and total 16 lawns = $528. Therefore, **Lee earned $558 last week** from mowing lawns and receiving tips.

- The original Llama2-7B model struggled with math-related questions, while the Full-BP and Sparse-BP models demonstrated certain capabilities for reasoning and properly calculating the results.

Lecture Plan

1. Deep leakage from gradients, gradient is not safe to share
2. Memory bottleneck of on-device training
3. Tiny transfer learning (TinyTL)
4. Sparse back-propagation (SparseBP)
5. **Quantized training with quantization aware scaling (QAS)**
6. PockEngine: system support for sparse back-propagation

Quantized Training

Sparse Training

PockEngine

Quantized Training: lower memory and latency

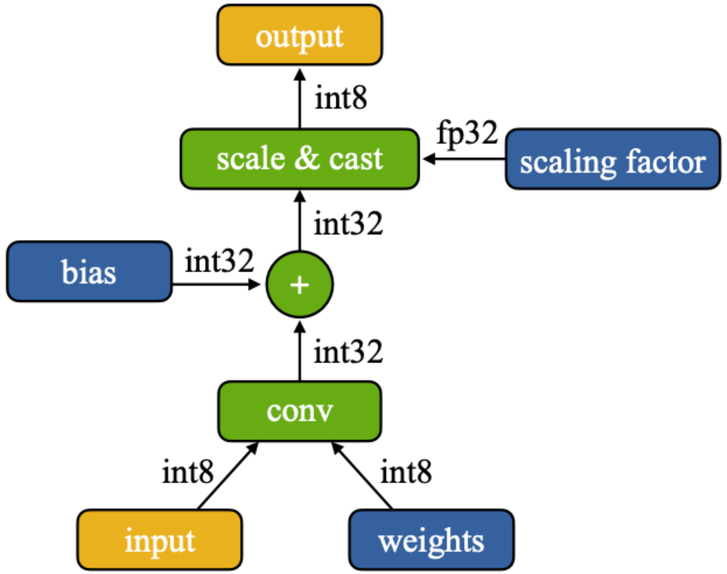
Full-precision training (32-bit)

2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

Quantized training (2-bit)

1	-2	0	-1
-1	-1	-2	1
-2	1	-1	-2
1	-1	0	0

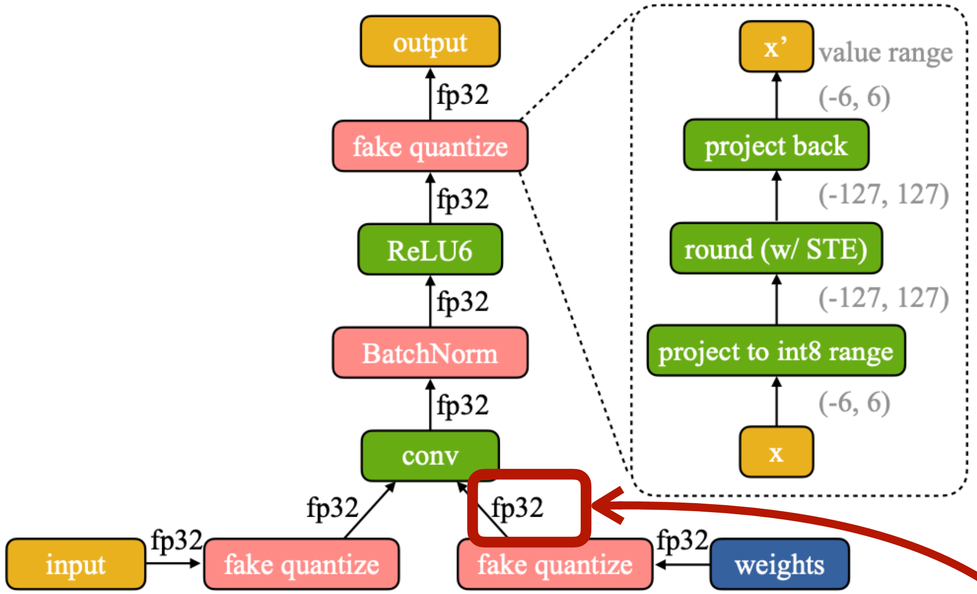
More efficient, but **difficult** to update



Real quantized graphs
(integer tensors)

Difficulty of Quantized Graphs

Fake quantization of QAT does not save training memory

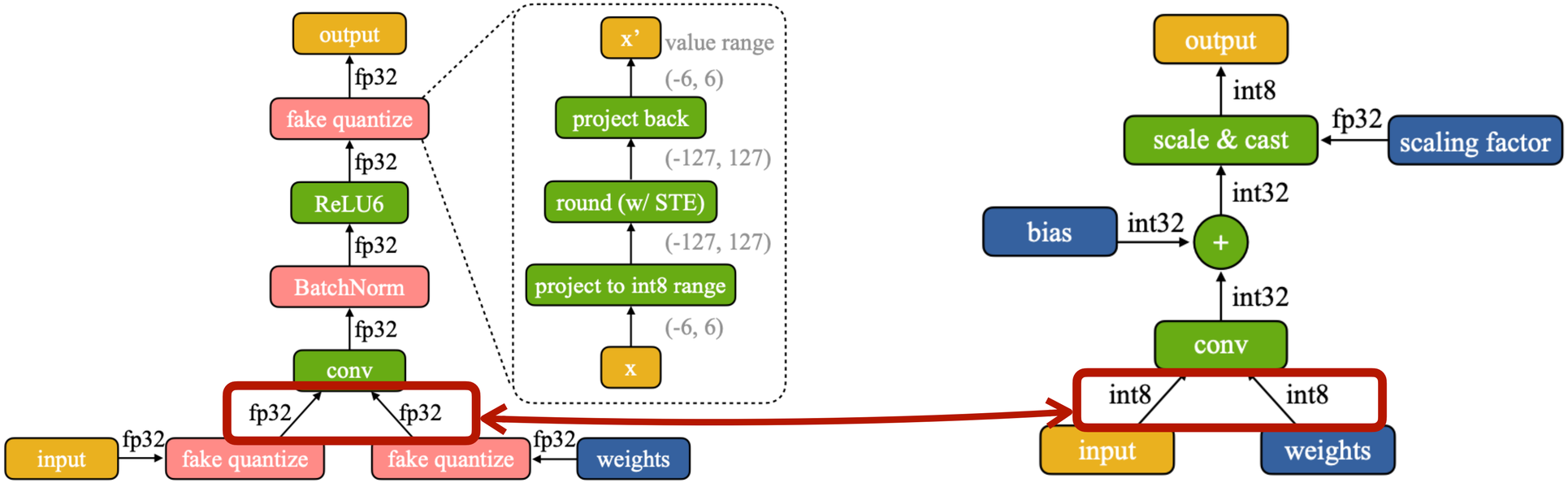


(a) Fake Quantization
(quantization aware training)

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)

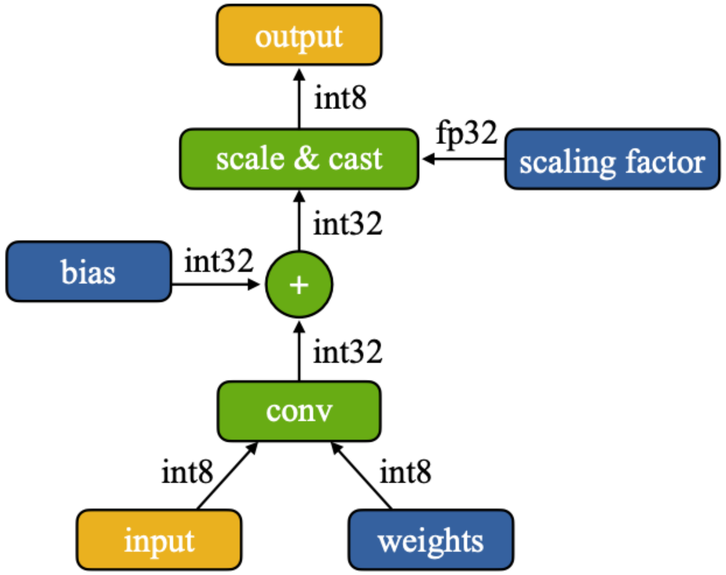
(b) Real Quantization
(inference/on-device training)

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

Real quantized graphs save memory, but hard to quantize

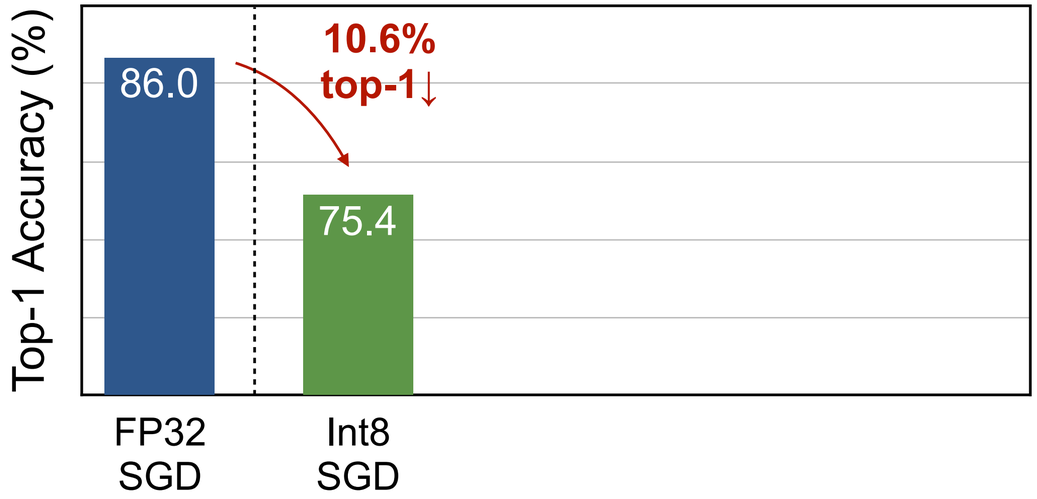


(a) Real Quantization

Making training difficult:

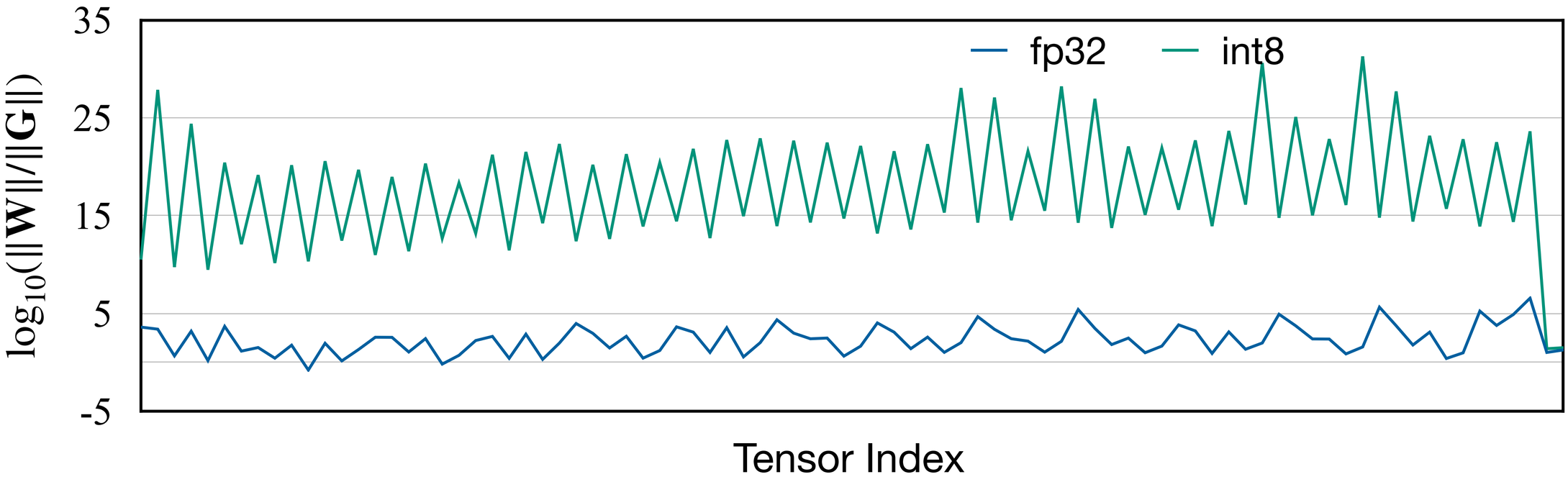
- Mixed precisions: int8/int32/fp32…
- Lack BatchNorm

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 $S_{\mathbf{W}}^{-2}$

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

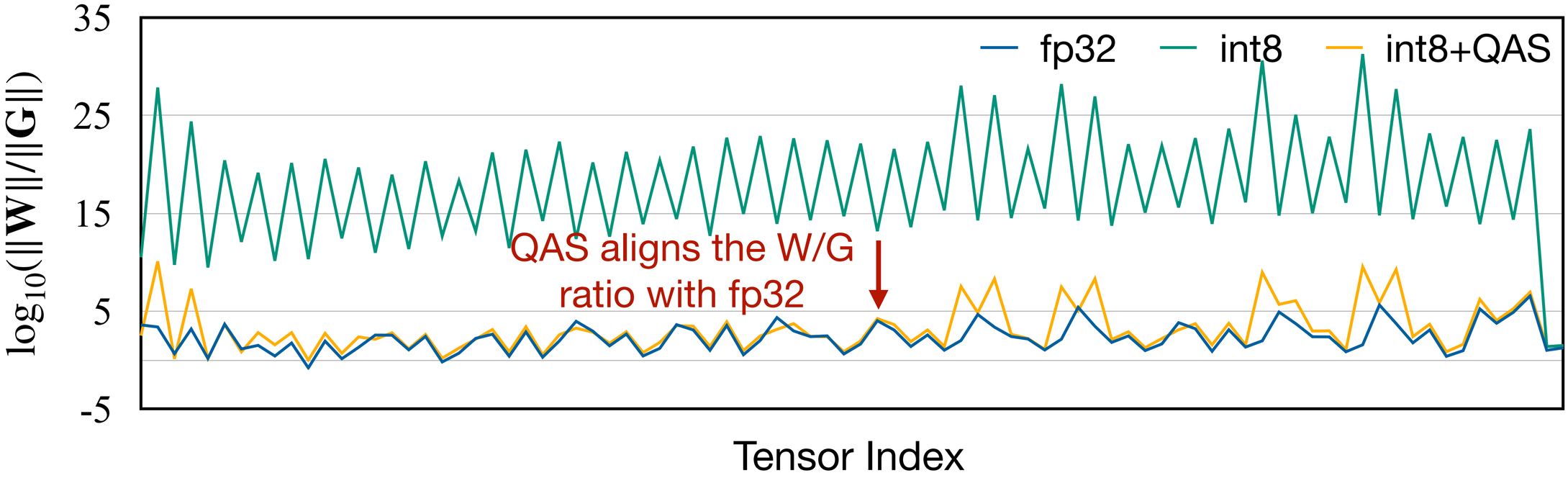
Thus, re-scale the gradients

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

QAS: Quantization-Aware Scaling

QAS addresses the optimization difficulty of quantized graphs

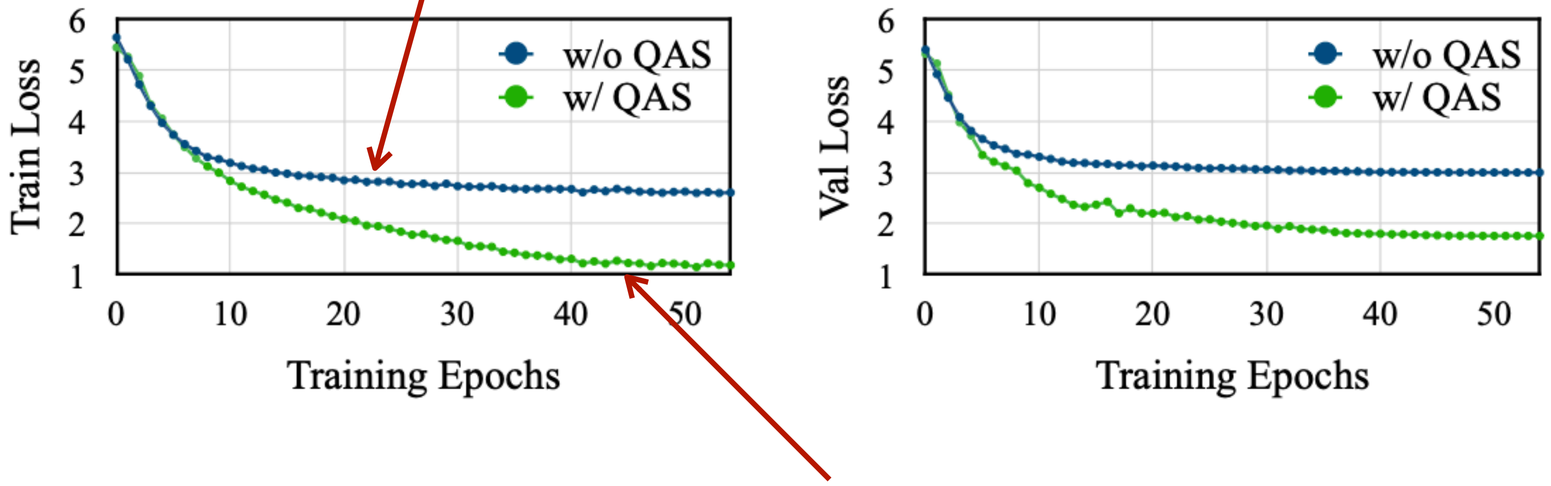
$$\tilde{\mathbf{G}}_{\bar{\mathbf{W}}} = \mathbf{G}_{\bar{\mathbf{W}}} \cdot s_{\mathbf{W}}^{-2}, \quad \tilde{\mathbf{G}}_{\bar{\mathbf{b}}} = \mathbf{G}_{\bar{\mathbf{b}}} \cdot s_{\mathbf{W}}^{-2} \cdot s_{\mathbf{x}}^{-2} = \mathbf{G}_{\bar{\mathbf{b}}} \cdot s^{-2}$$



QAS: Quantization-Aware Scaling

QAS addresses the optimization difficulty of quantized graphs

Without QAS, **poor** convergence

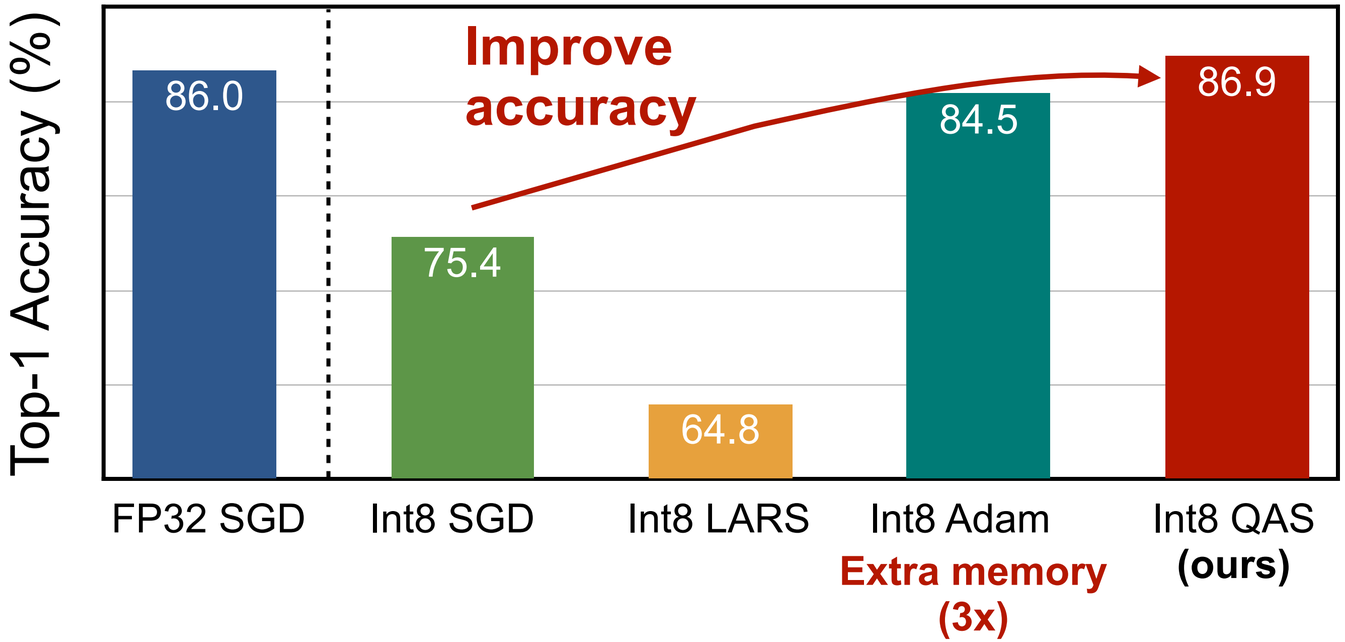


With QAS, **better** convergence

After applying QAS, the convergence of real quantized is stable.

QAS: Quantization-Aware Scaling

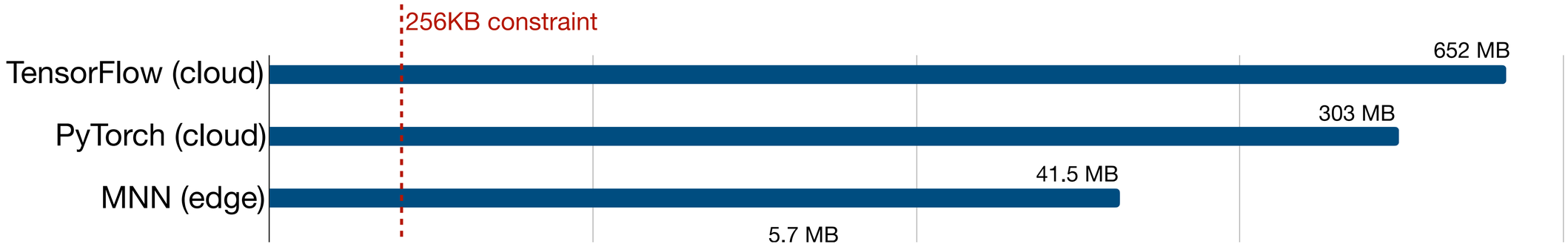
QAS addresses the optimization difficulty of quantized graphs



QAS improves the accuracy over naive int8 training, and shows no inferior performance than fp32 results.

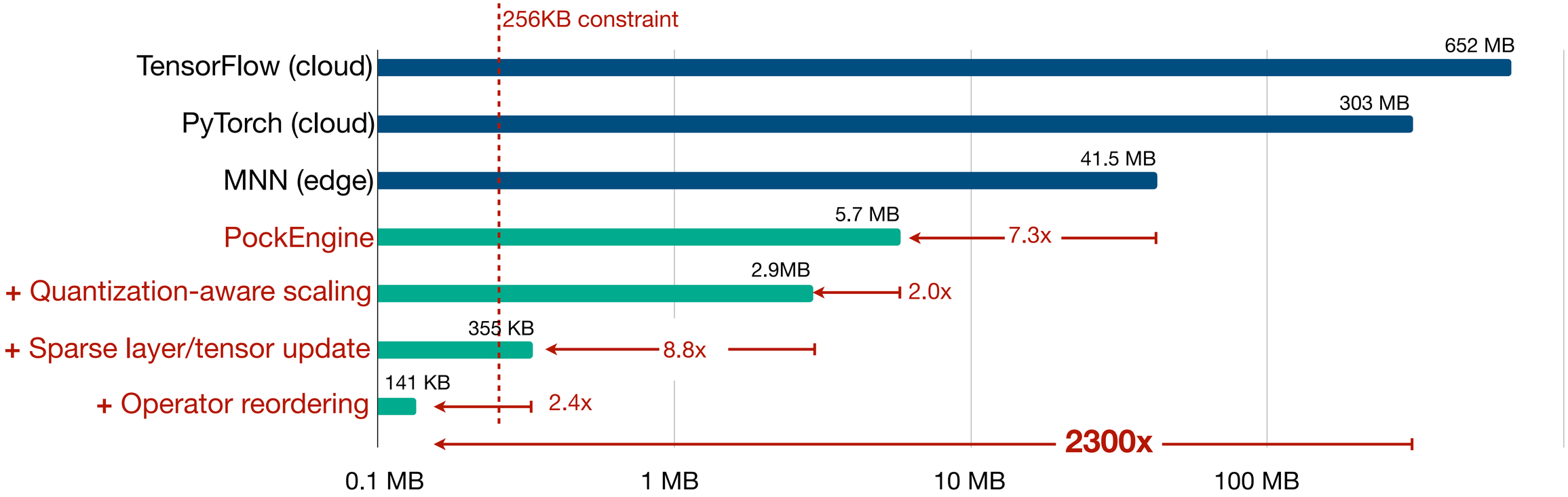
On-Device Training Under 256KB Memory

- **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



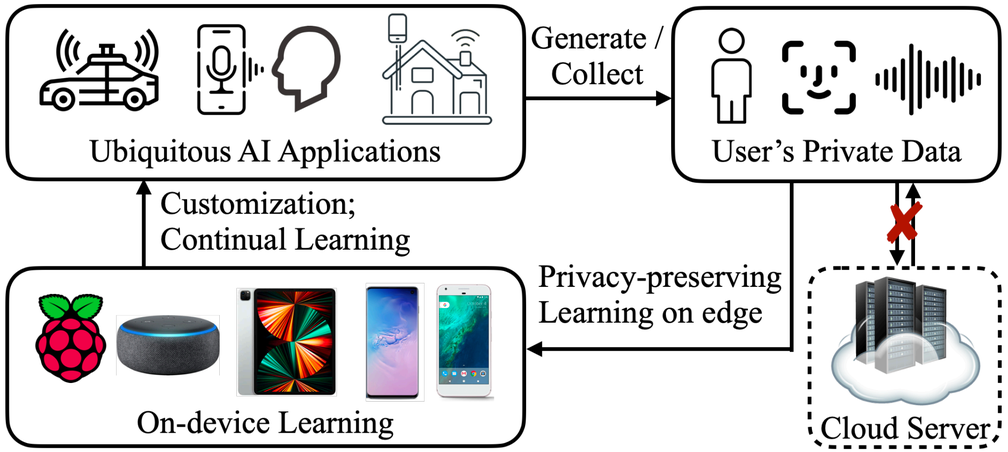
On-Device Training Under 256KB Memory

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

ImageNet Pre-trained MCUNet -> VWW
Running on OpenMV Cam H7 MCU
SRAM: 185KB / Flash: 467KB

Summary

- Gradient is not safe to share. Staying local is important.
- CNN's training memory bottleneck is the activation.
- Efficient transfer learning with bias-only and lite-residual.
- Full-update is too expensive and using sparse back-propagation for on-device training.
- Why training a quantized model is difficult and how to improve by QAS.



How can we translate the algorithmic improvement into actual savings?

Algorithm System Co-Design.

Lecture Plan

1. Deep leakage from gradients, gradient is not safe to share
2. Memory bottleneck of on-device training
3. Tiny transfer learning (TinyTL)
4. Sparse back-propagation (SparseBP)
5. Quantized training with quantization aware scaling (QAS)
6. **PockEngine: system support for sparse back-propagation**

Quantized Training

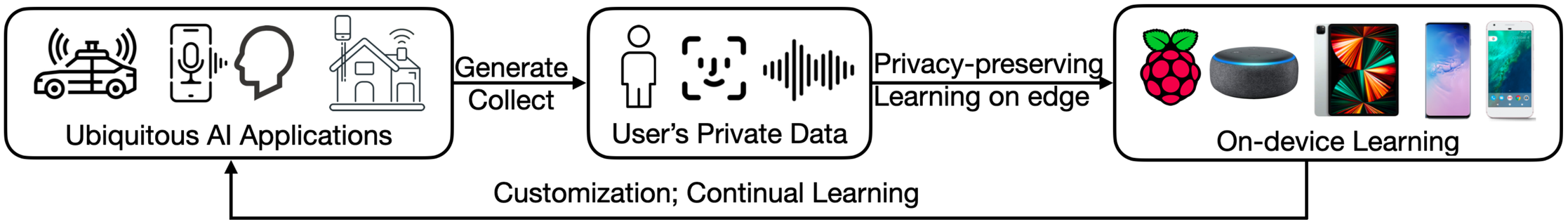
Sparse Training

PockEngine

The Benefits of Learning on the Edge

From tiny inference to training

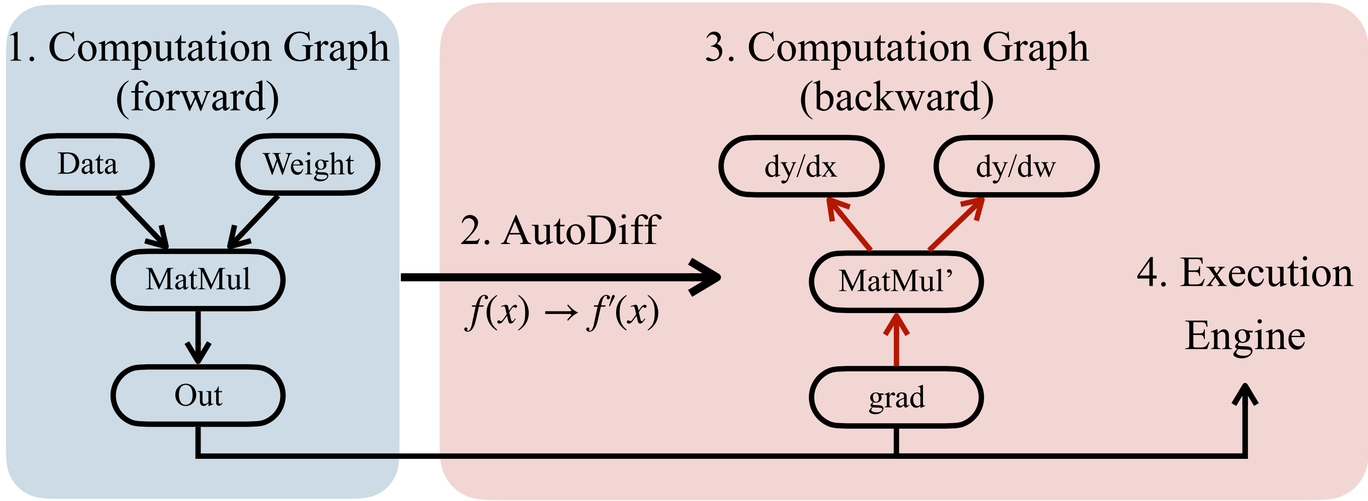
A **virtuous cycle**:



- On-device learning:
  - **customization** by adapting to user data / **life-long** learning
  - better **privacy**, lower **cost**
- Training is more **expensive** than inference
  - For example, store intermediate activation, extra back-propagation, etc.

PockEngine

Previous DL Training



Cannot apply many graph optimizations during runtime

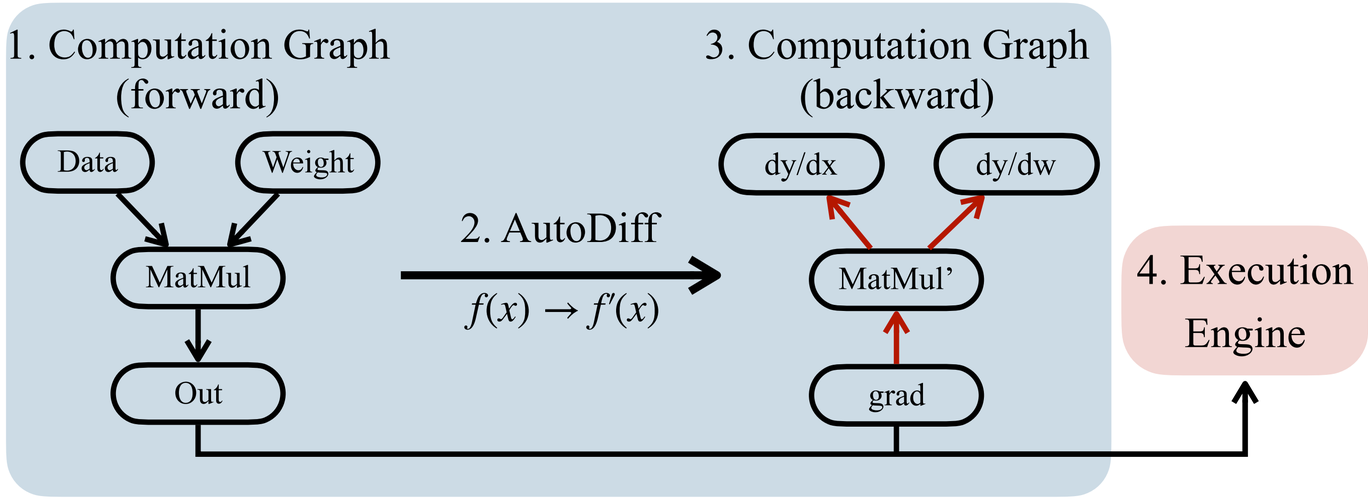
: Compile-Time

: Runtime

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

PockEngine

Compile-Time Autodiff



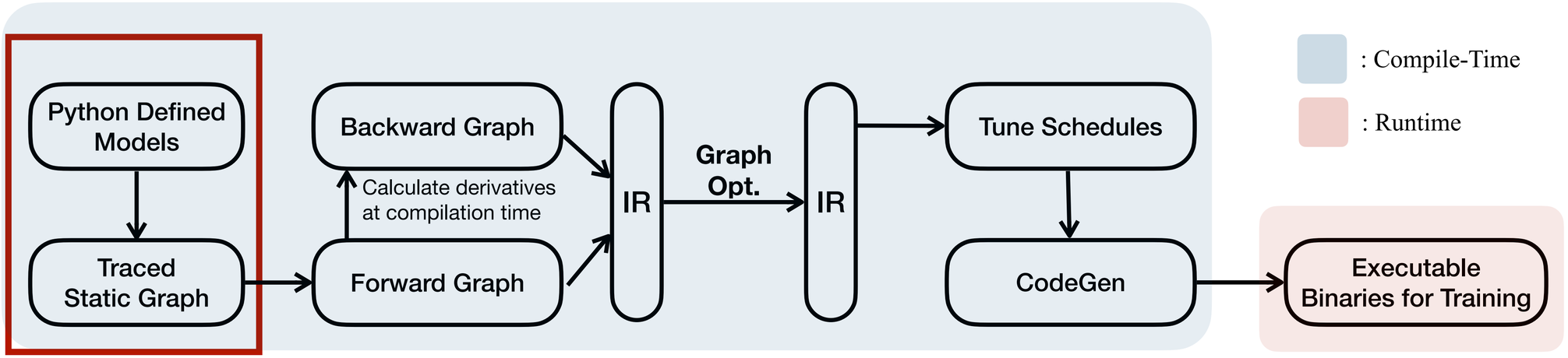
(b) Our PockEngine

: Compile-Time

: Runtime

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

PockEngine



PyTorch

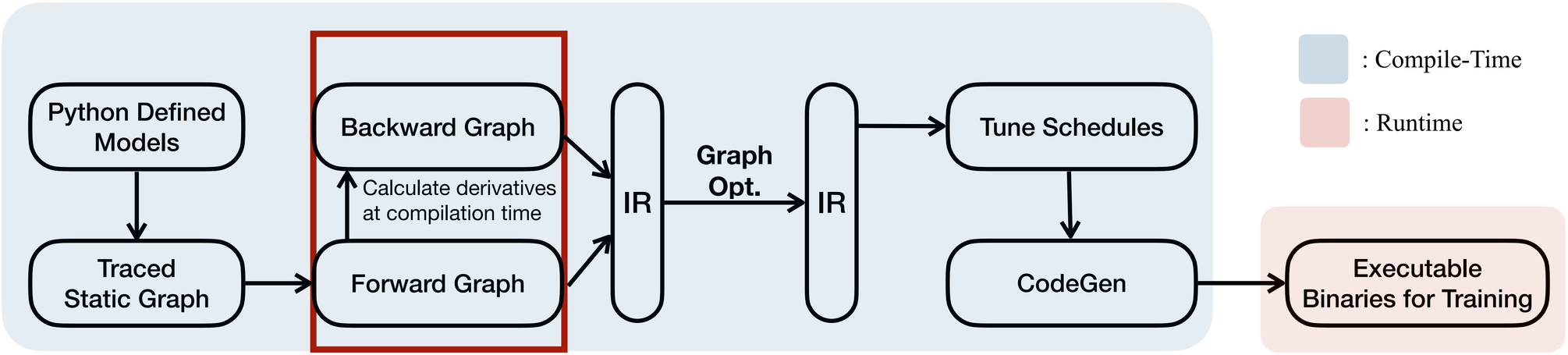
```
net = nn.Sequential(
    nn.Conv2d(3, 3, kernel=3, padding=1),
    nn.ReLU()
)

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)
}
```

On-Device Training Under 256KB Memory [Lin et al, NeurIPS 2022]
PockEngine: Sparse and Efficient Fine-tuning in a Pocket [Zhu et al, Micro 2023]

PockEngine



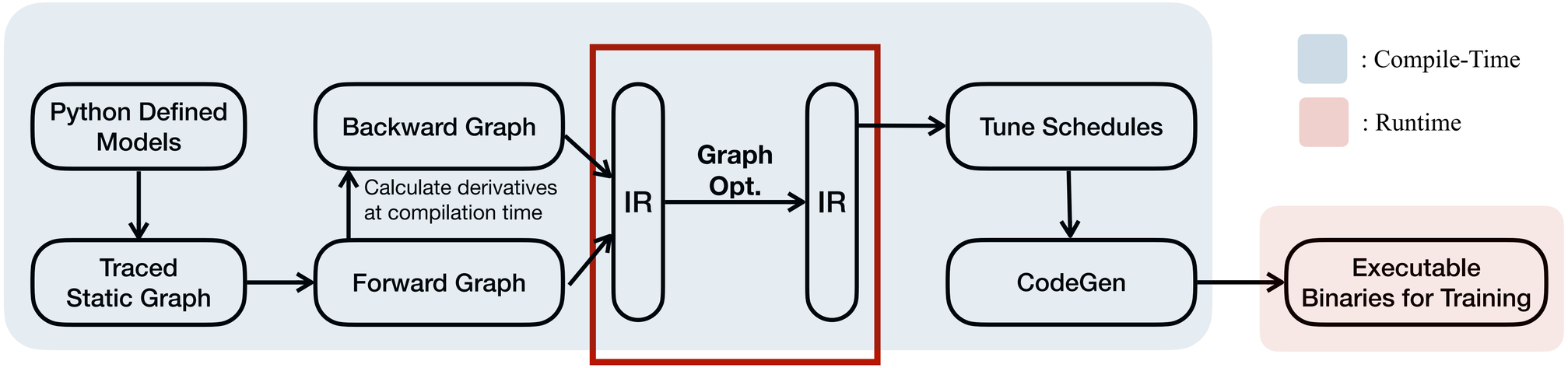
Forward IR

```
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)
}
```

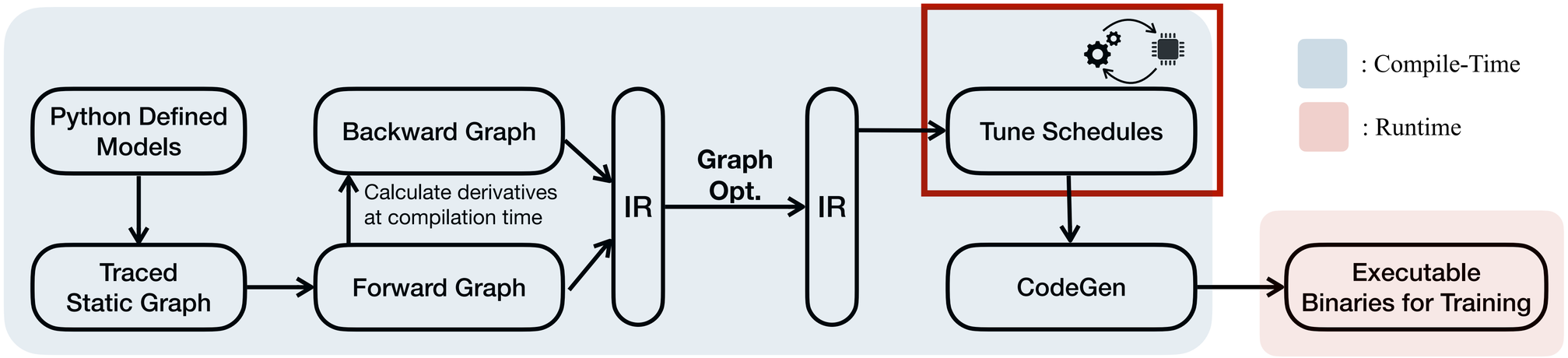
PockEngine

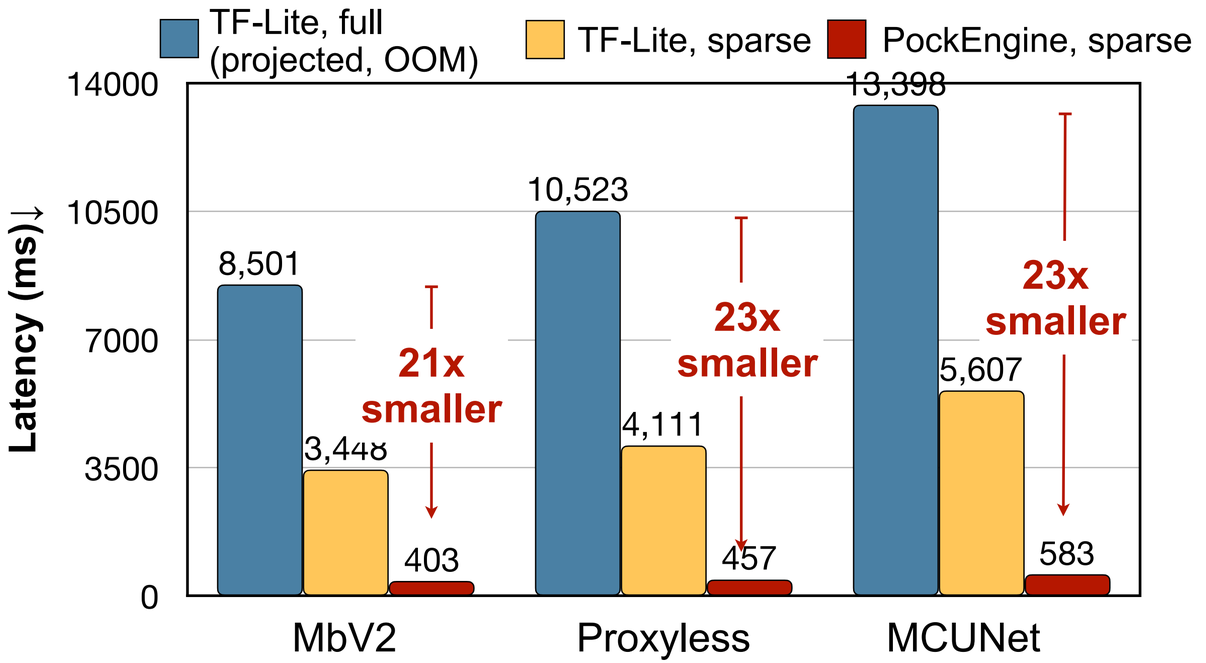


- Graph-level optimizations:
  - Sparse layer / sparse tensor update
  - Operator reordering and in-place update
  - Constant folding
  - Dead-code elimination

On-Device Training Under 256KB Memory [Lin et al, NeurIPS 2022]
PockEngine: Sparse and Efficient Fine-tuning in a Pocket [Zhu et al, Micro 2023]

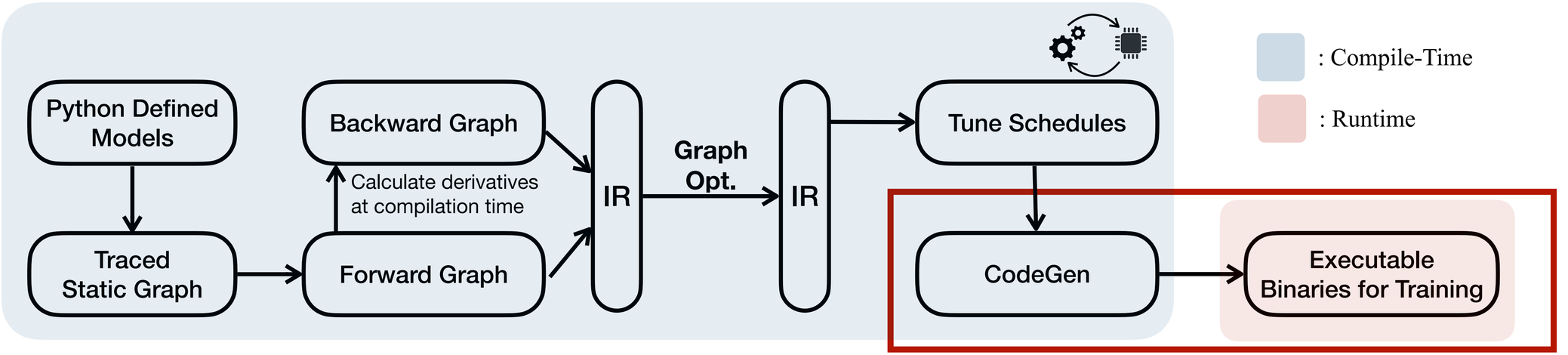
PockEngine





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

PockEngine




```
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);
```

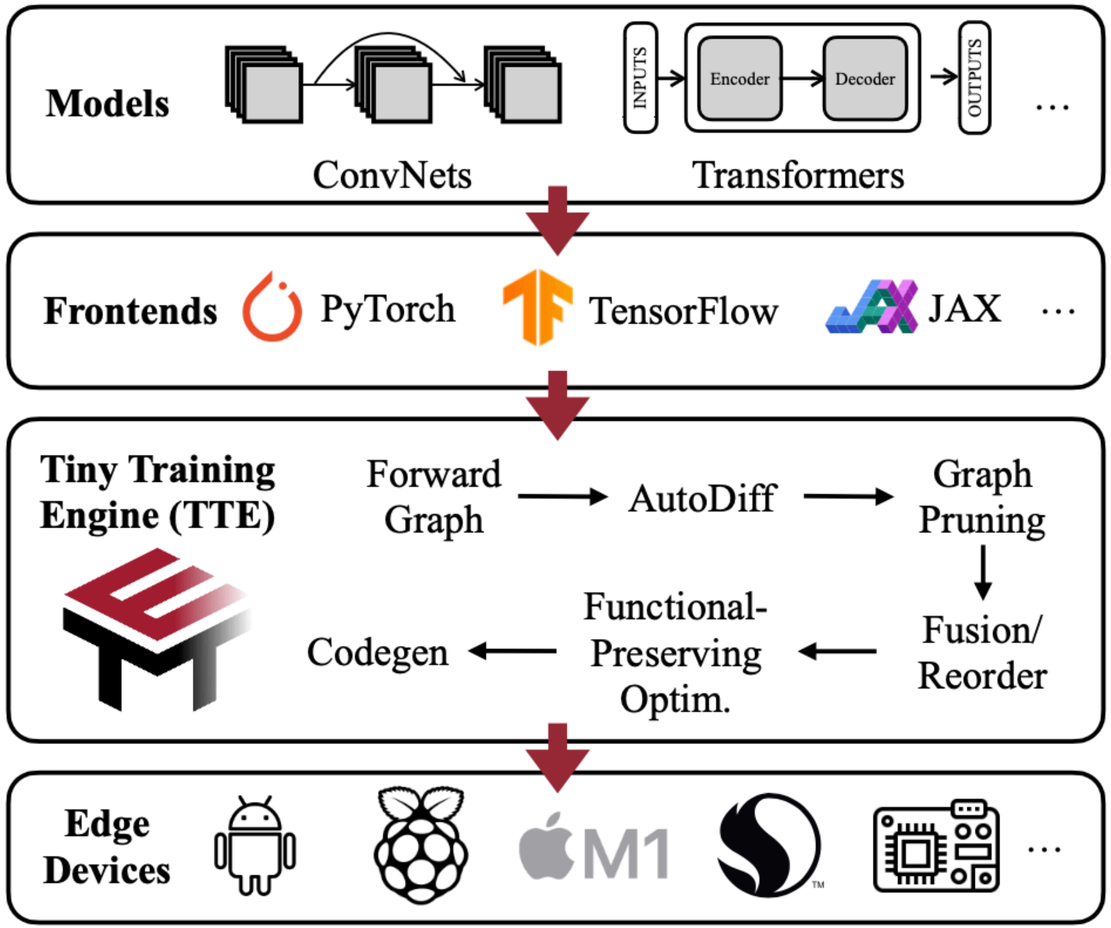
Our codegen only generate binaries for used operators

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

Extending PockEngine to More Platforms

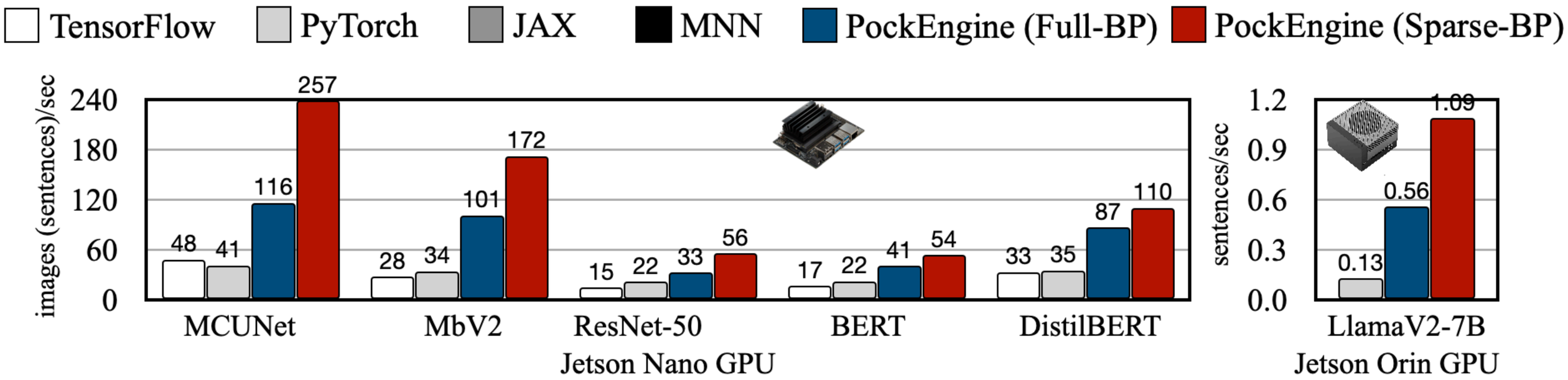
Accelerate on-device training on diverse edge hardware

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



PockEngine: Extend to More Platforms

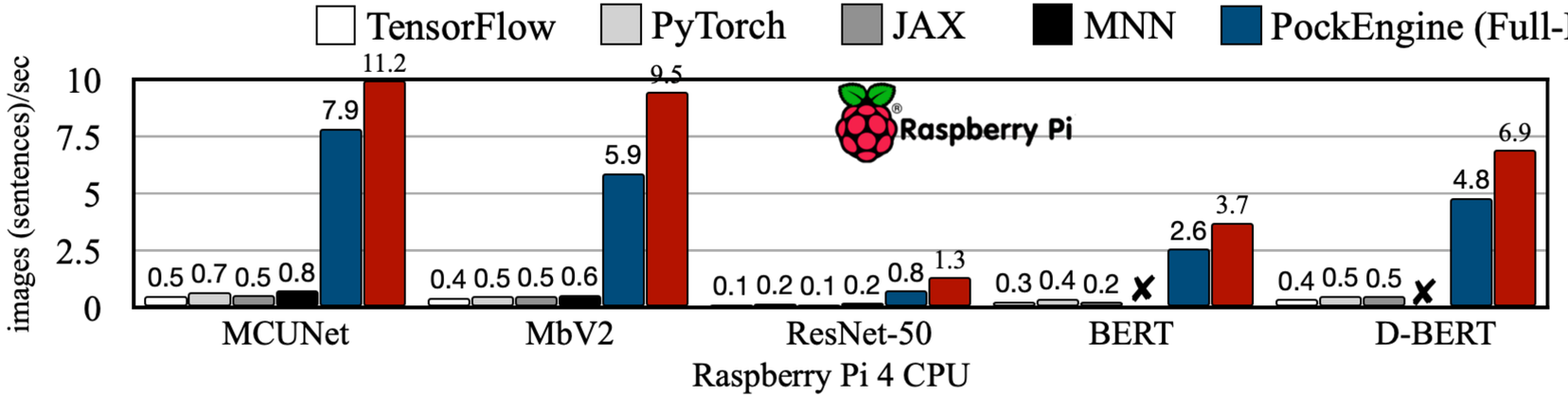
Speedup comparison: Edge GPU

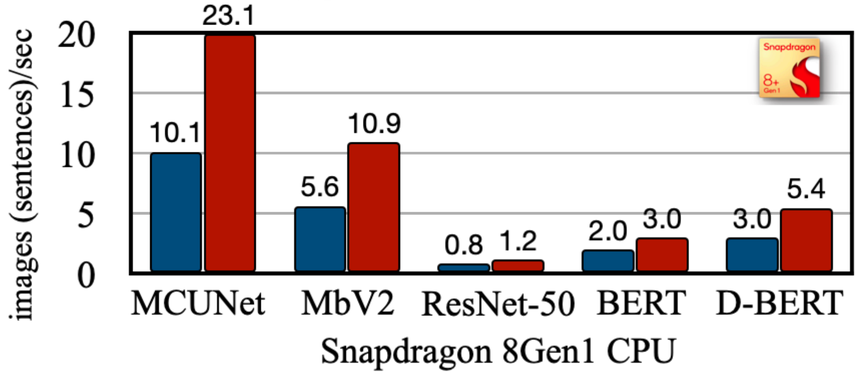


- **2-4x speedup** on Jetson Nano and Orin.
- Speedup comes from compilation. The host language (Python) is slow on low-frequency CPUs.
- Commercial inference frameworks (e.g., TensorRT) also benefits from compilation, but not for training.

PockEngine: Extend to More Platforms

Speedup comparison: ARM CPU

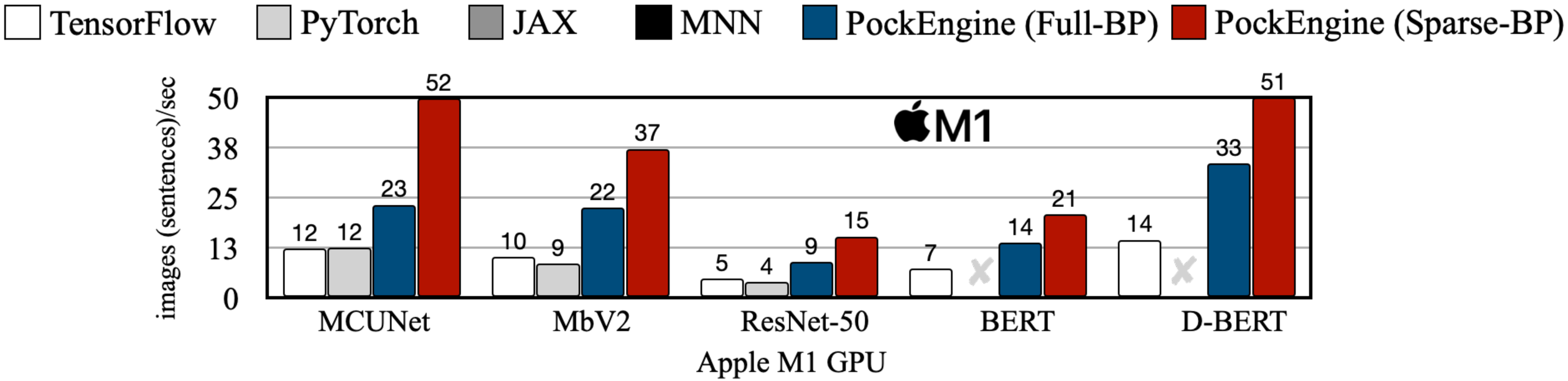




- PockEngine shows **13 to 21x speedup** on Raspberry Pi 4 B+ platforms.
  - Existing framework are mostly optimized for inference only.
  - Compilation enables kernel tuning, thus accelerates training.
    - Most kernel implementations focus on GPU and x86 CPU.
    - ARM CPU (especially for training) is highly under optimized.
  - Sparse-BP can further speedup fine-tuning throughput.

PockEngine: Extend to More Platforms

Speedup comparison: Apple M-Chip

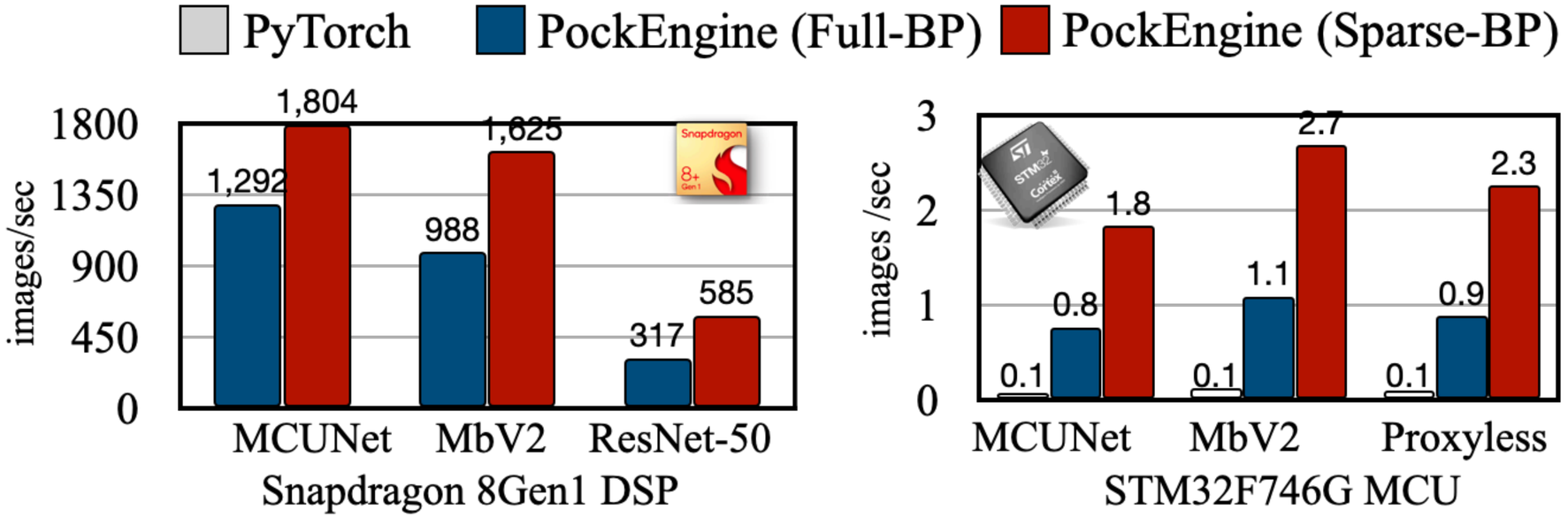


- Apple M1/2 is a new, and the **compatibility is not ideal** for PyTorch and TensorFlow:
  - For PyTorch*, even with a recent build (commit ID: c9913cf), transformer training throws errors on M1.
  - For TensorFlow, the GPU training support is preliminary and incomplete on M1.
- PockEngine compiles the training graph to Metal, **providing better compatibility** and **faster training speeds**.

*https://github.com/pytorch/pytorch/issues/77764

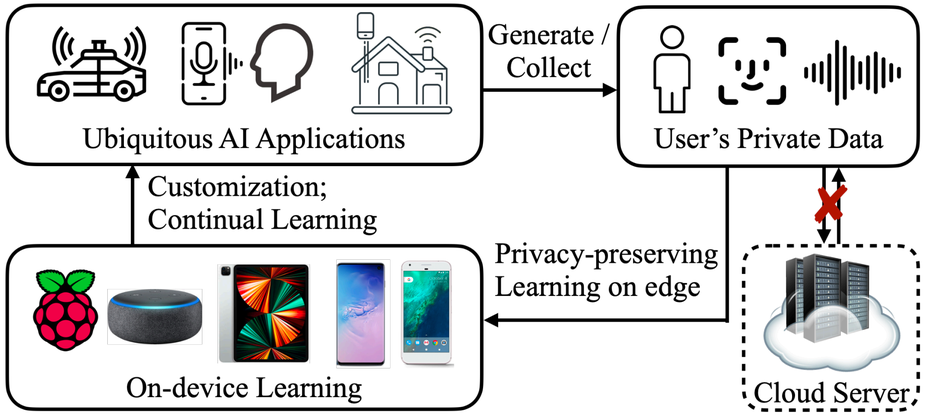
PockEngine: Extend to More Platforms

Speedup comparison: DSP and MCUs



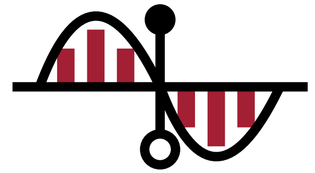
- We integrate SNPE for Qualcomm DSPs and TinyEngine for Microcontrollers
- PockEngine's compilation workflow is (1) kernel agnostic and (2) shares the same set of OPs between FWD and BWD, thus enables **previous inference-only framework to support training**.

Summary of Today's Lecture



Today we will discuss:

1. Deep leakage from gradients, gradient is not safe to share
2. Memory bottleneck of on-device training
3. Tiny transfer learning (TinyTL)
4. Sparse back-propagation (SparseBP)
5. Quantized training with quantization aware scaling (QAS)
6. PockEngine: system support for sparse back-propagation



Quantized Training

Sparse Training

PockEngine

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]