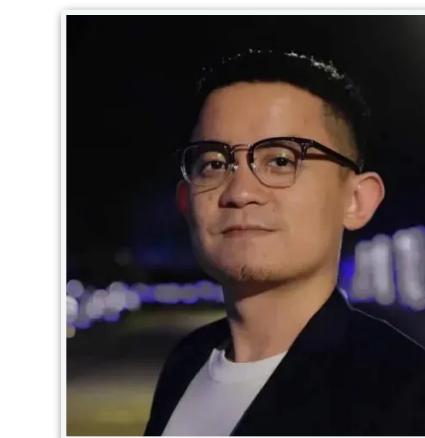
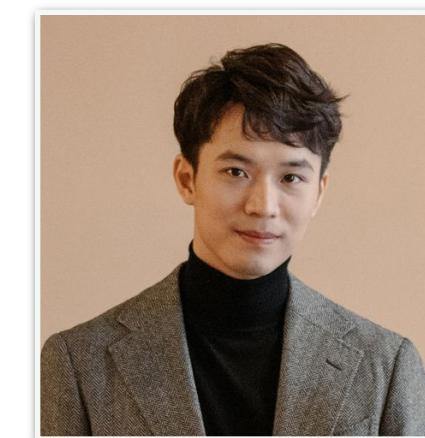


# PockEngine: Sparse and Efficient Fine-tuning in a Pocket

56th IEEE/ACM International Symposium on Microarchitecture (MICRO 2023)

Ligeng Zhu<sup>1</sup>, Lanxiang Hu<sup>2</sup>, Ji Lin<sup>1</sup>, Wei-Chen Wang<sup>1</sup>, Wei-Ming Chen<sup>1</sup>, Chuang Gan<sup>3</sup>, Song Han<sup>1,4</sup>

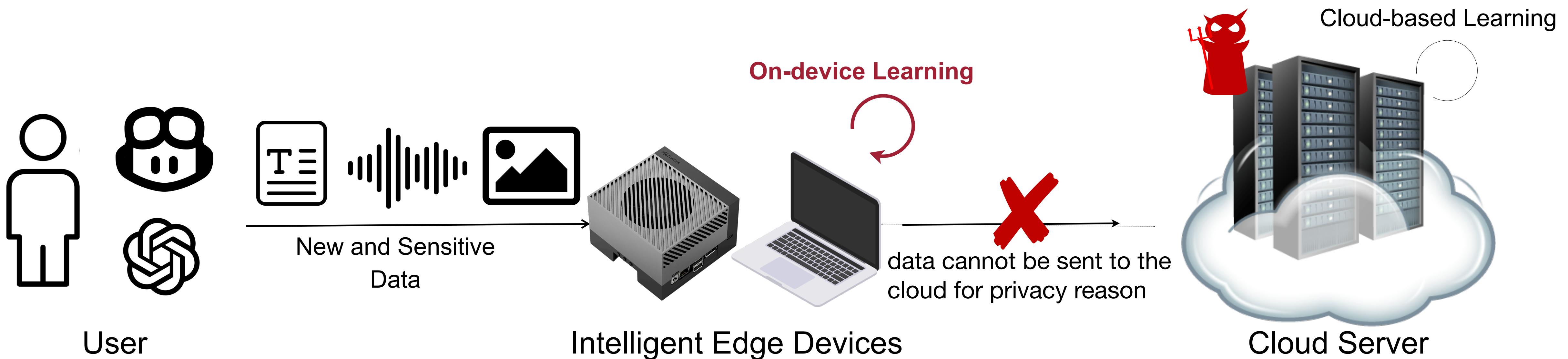
MIT<sup>1</sup>, UCSD<sup>2</sup>, MIT-IBM Watson AI Lab<sup>3</sup>, NVIDIA<sup>4</sup>



# Can We Learn on the Edge?

AI systems need to continually adapt to new data collected from the edge  
-> requires training

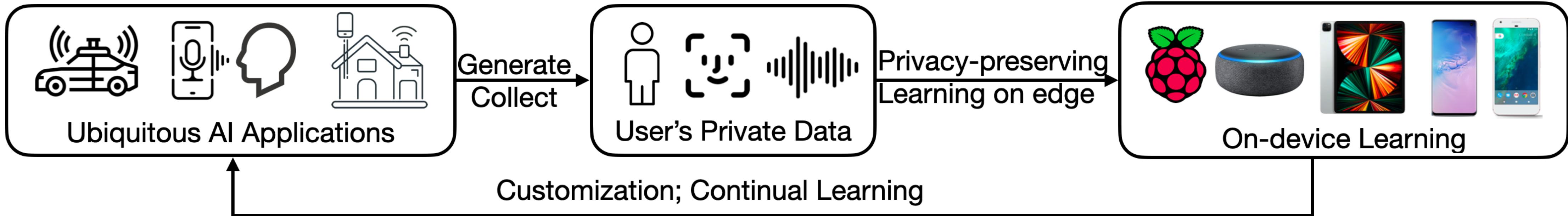
- On-device learning: **better privacy, lower cost, customization, life-long learning**
- Training is more **expensive** than inference, hard to fit edge hardware (limited memory)



# Push Training to be On-Device

Squeezing *training* into local devices

A **virtuous cycle**:



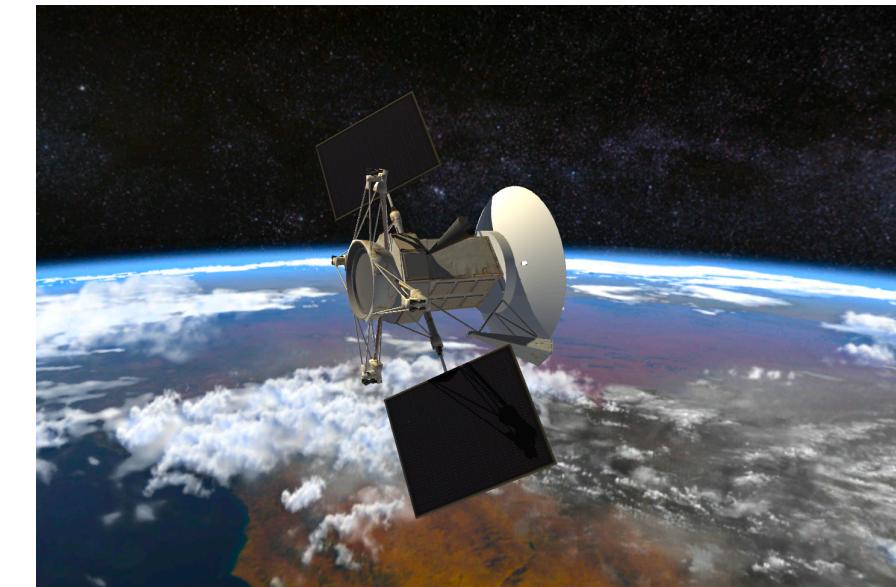
Privacy  
Coding Assistant



Personalized Healthcare



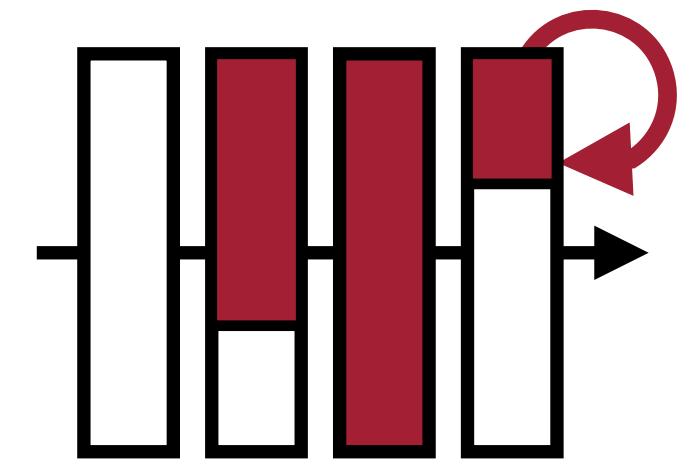
Connectivity  
Space Exploration



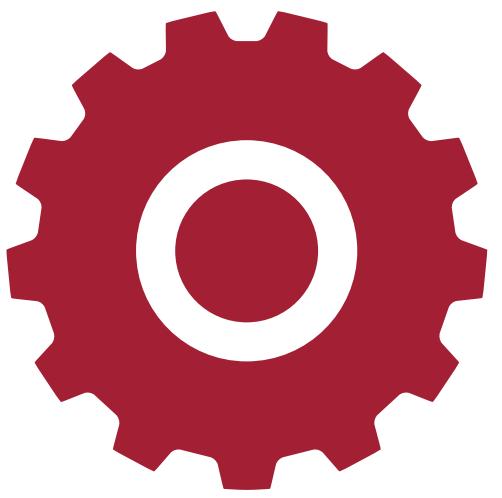
Ocean Sensing

no physical connection to the internet.

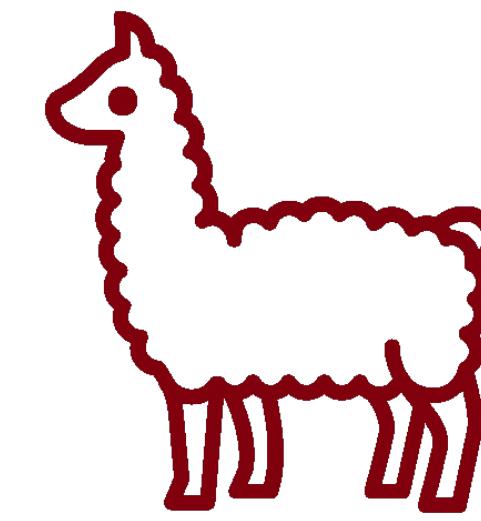
# PockEngine



**1. Sparse Layer/Tensor  
Back-propagation**

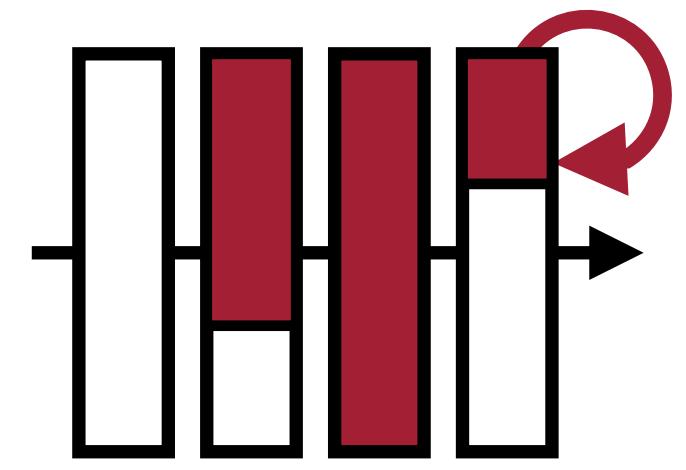


**2. Compiler Support**

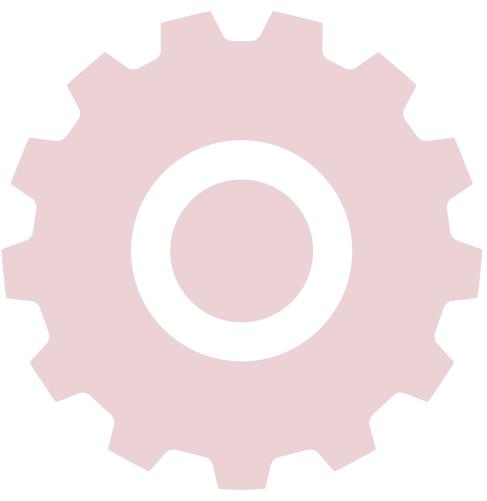


**3. Fine-tune Llama2 on Orin**

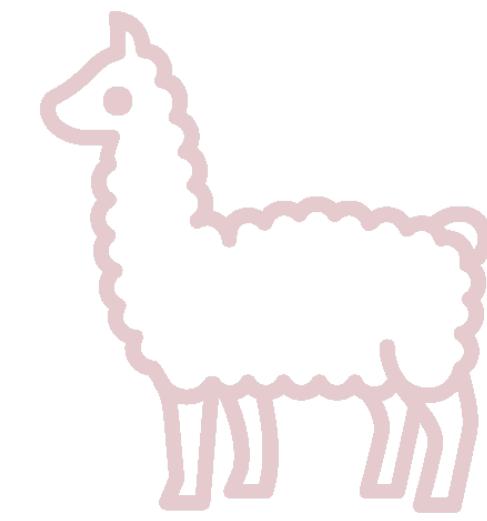
# PockEngine



**1. Sparse Layer/Tensor  
Back-propagation**



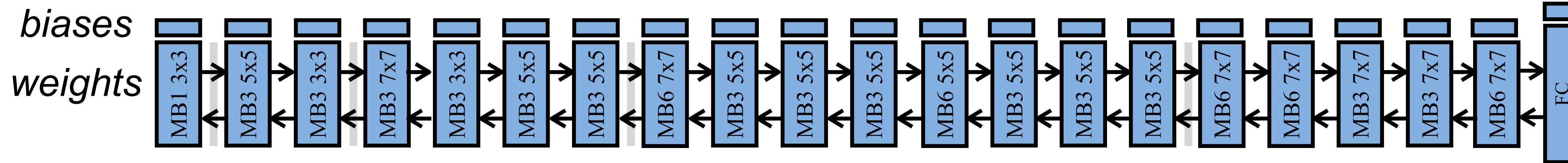
**2. Compilation-Based**



**3. Fine-tune Llama2 on Orin**

# Sparse Layer/Tensor Back-propagation

# Full update



# Model: ProxylessNAS-Mobile

# Updating the whole model is too expensive:

- Need to save all intermediate activation (quite large)

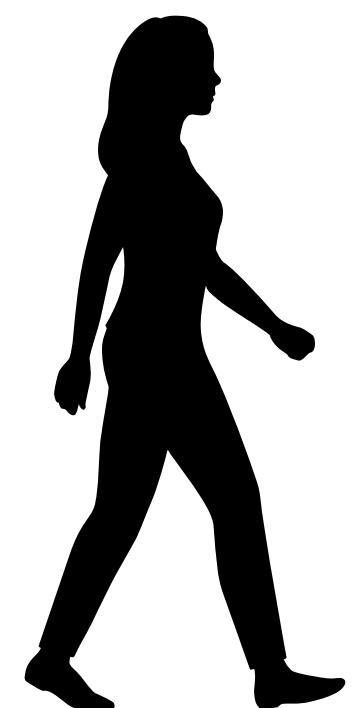
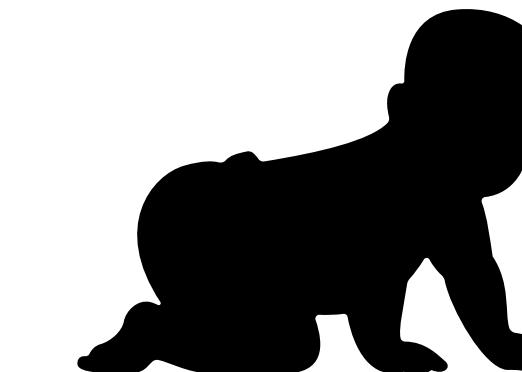
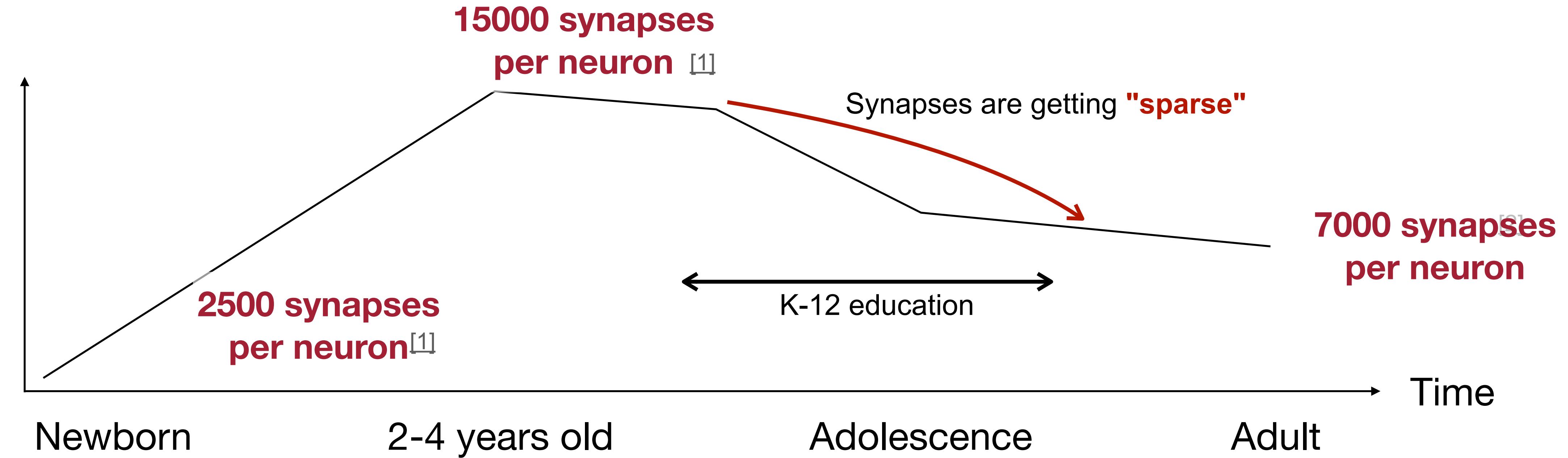
Forward:  $\mathbf{a}_{i+1} = \mathbf{a}_i \mathbf{W}_i + \mathbf{b}_i$

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

- 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.

# Sparse Layer/Tensor Back-propagation

## The idea of Sparse Update



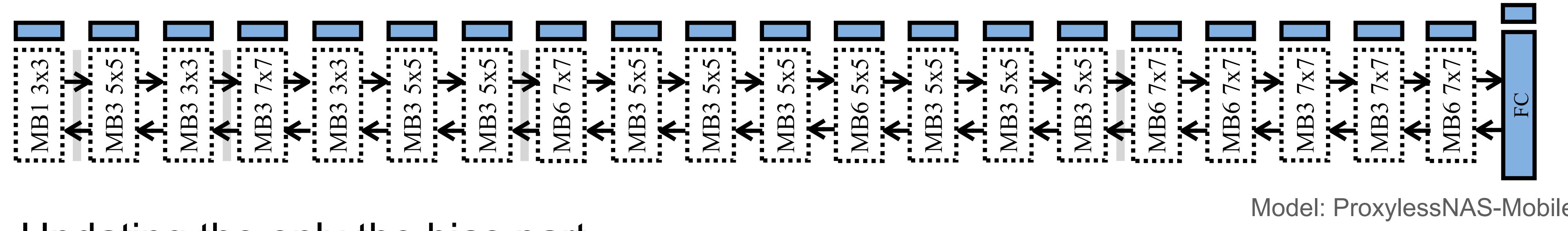
Do We Have Brain to Spare? [Drachman DA, Neurology 2004]  
Peter Huttenlocher (1931–2013) [Walsh, C. A., Nature 2013]

Data Source: 1, 2

Slide Inspiration: Alila Medical Media

# Sparse Layer/Tensor Back-propagation

## Bias-only update

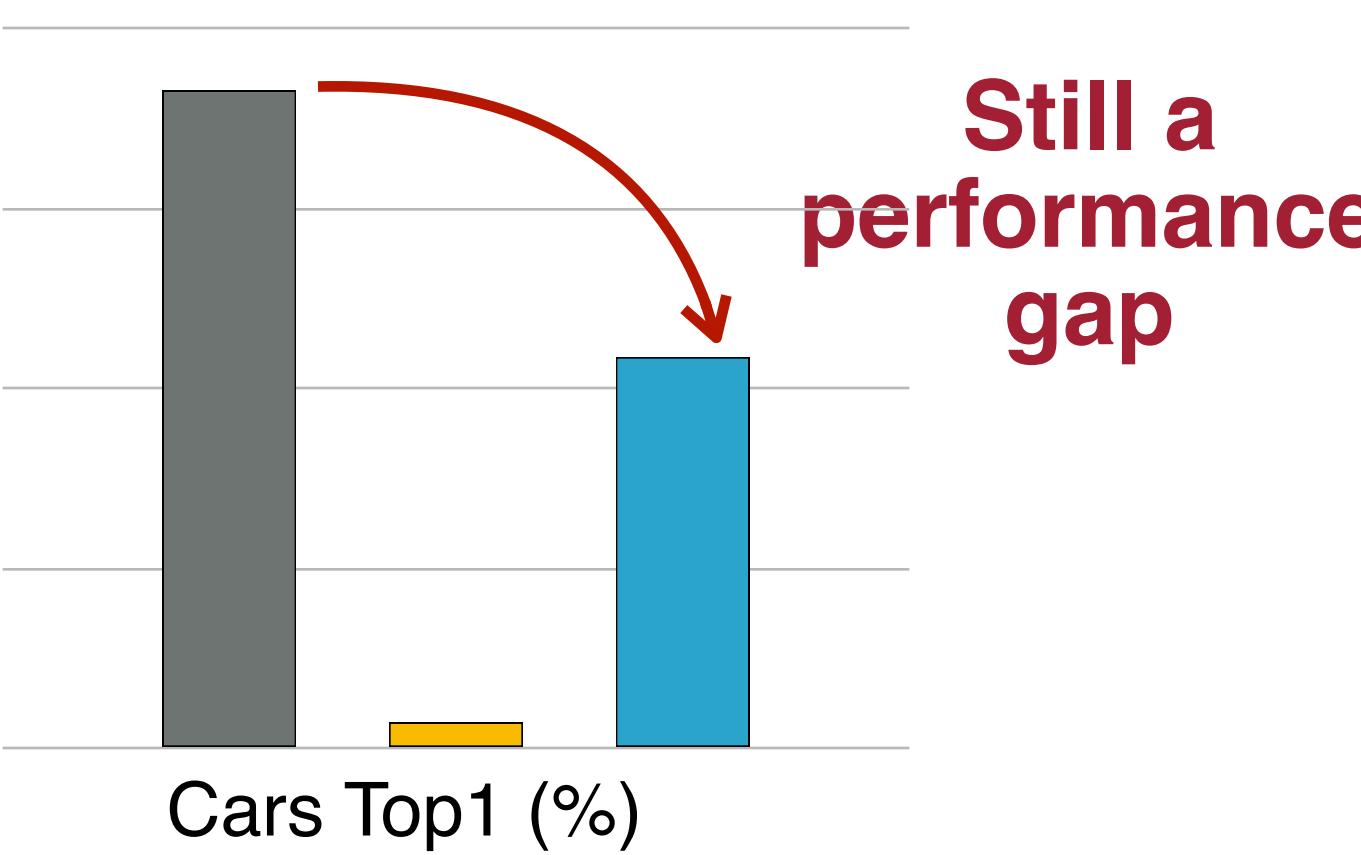
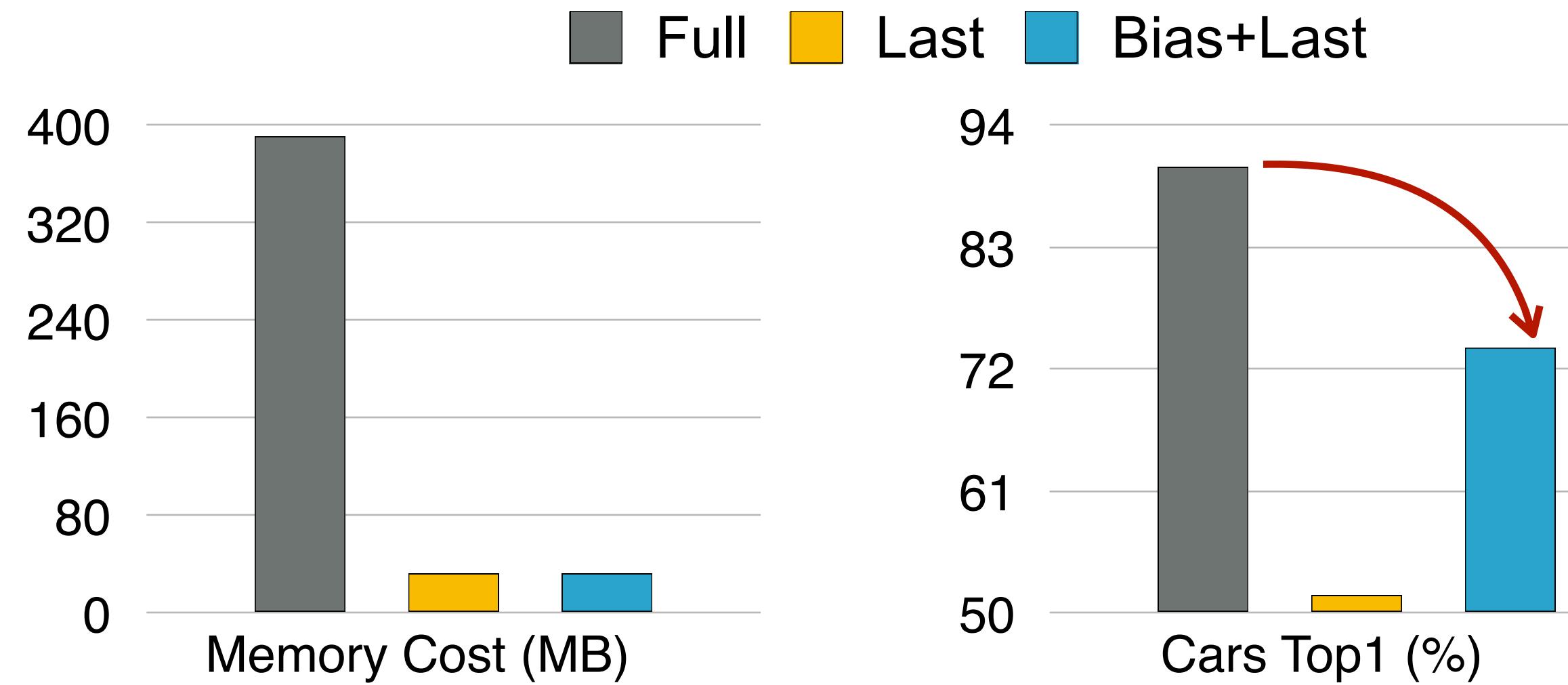


Updating the only the bias part

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

$$dW = f(\mathbf{X}, dY)$$

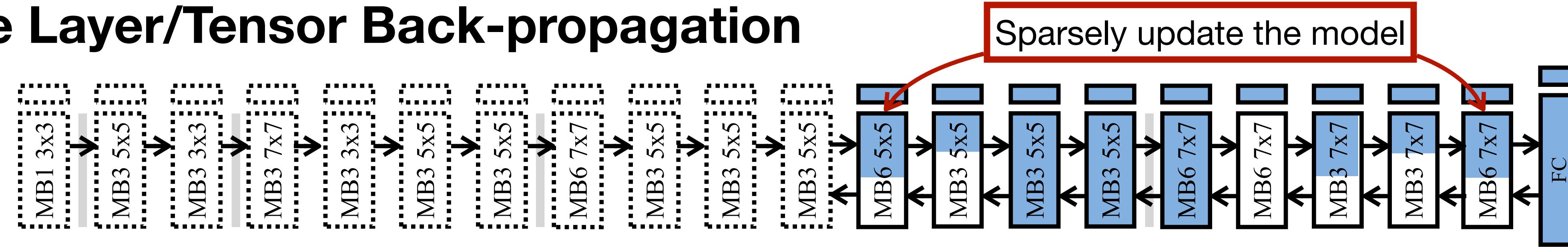
$$db = f(dY)$$



Still a  
performance  
gap

# Sparse Layer/Tensor Back-propagation

## Sparse Layer/Tensor Back-propagation



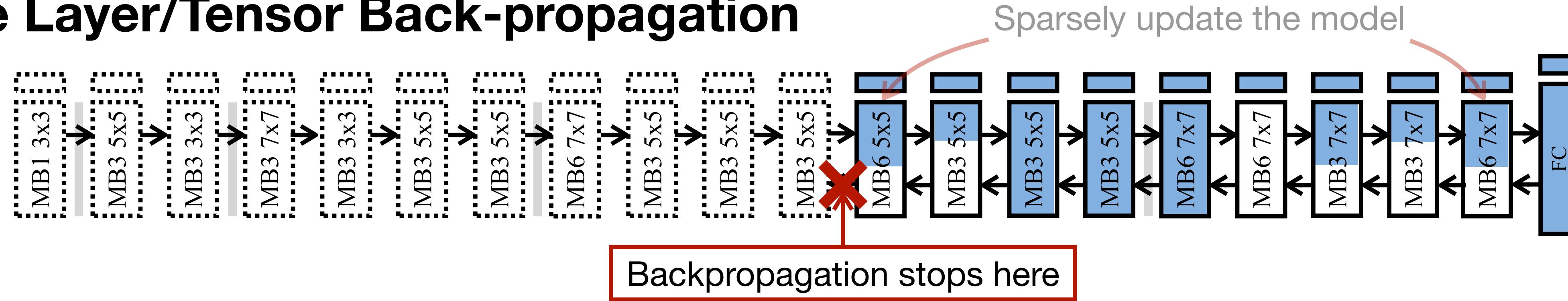
Use sparse update to train the model

- Some layers are **more important than others**

Model: ProxylessNAS-Mobile

# Sparse Layer/Tensor Back-propagation

## Sparse Layer/Tensor Back-propagation



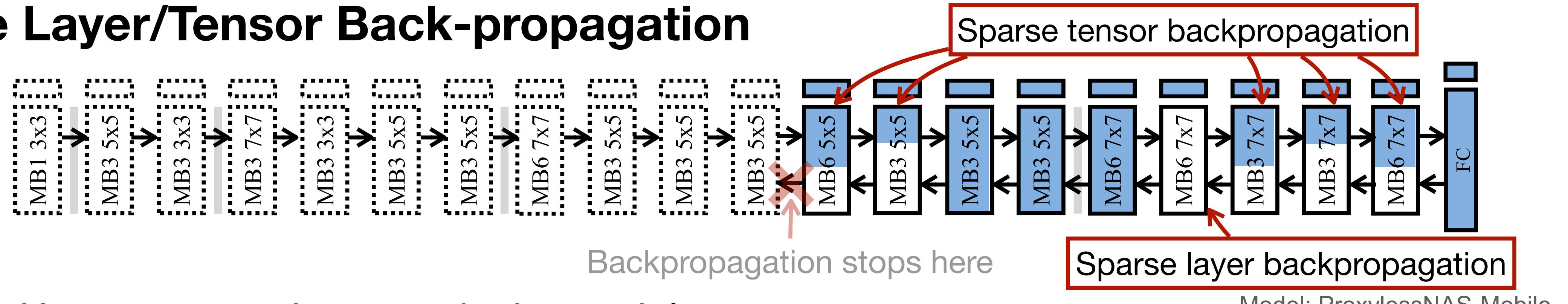
Use sparse update to train the model

- Some layers are more important than others
- **No need to back-propagate the early layers**

Model: ProxylessNAS-Mobile

# Sparse Layer/Tensor Back-propagation

## Sparse Layer/Tensor Back-propagation



Use sparse update to train the model

- Some layers are more important than others
- No need to back propagate the early layers
- **Only need to store a subset of the activations.**

$$\frac{dy}{dw} : \begin{matrix} (H, N) \\ \text{G.T} \end{matrix} \quad \begin{matrix} (N, M) \\ x \end{matrix} = \begin{matrix} (H, M) \\ (dW).T \end{matrix}$$

Activation to store: (N, M)

Weight in SRAM: (M, H)

$$\frac{dy}{dw} : \begin{matrix} (H, N) \\ \text{G.T} \end{matrix} \quad \begin{matrix} (N, M) \\ x \end{matrix} = \begin{matrix} (H, M) \\ (dw).T \end{matrix}$$

Activation to store: (N, 0.25\*M)

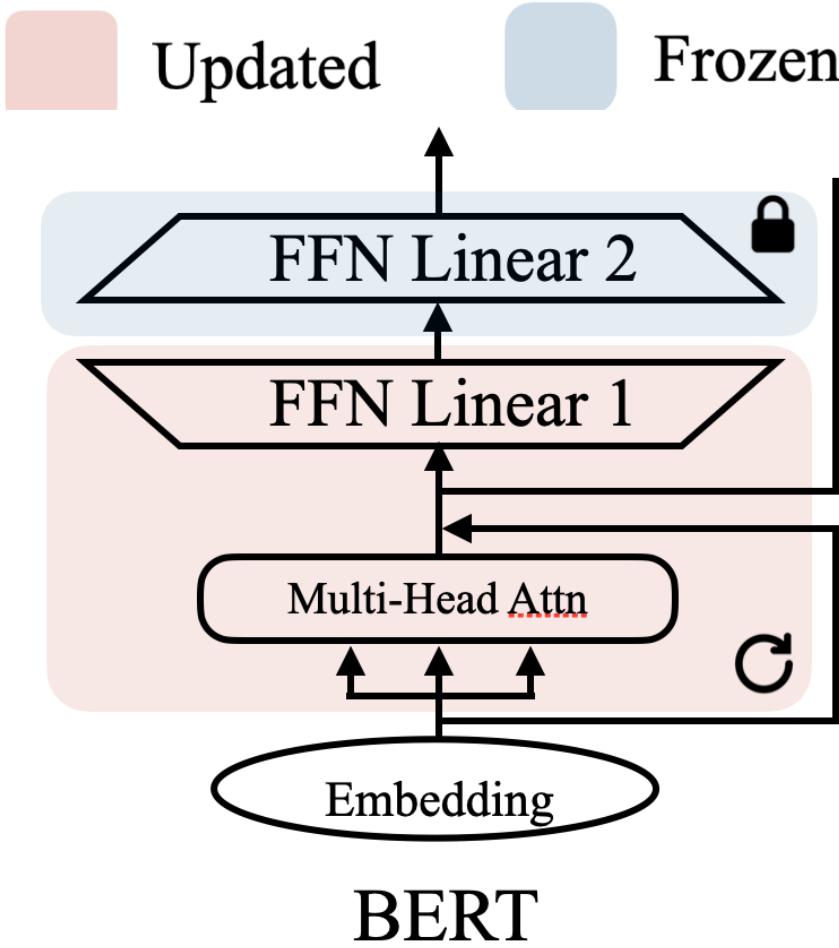
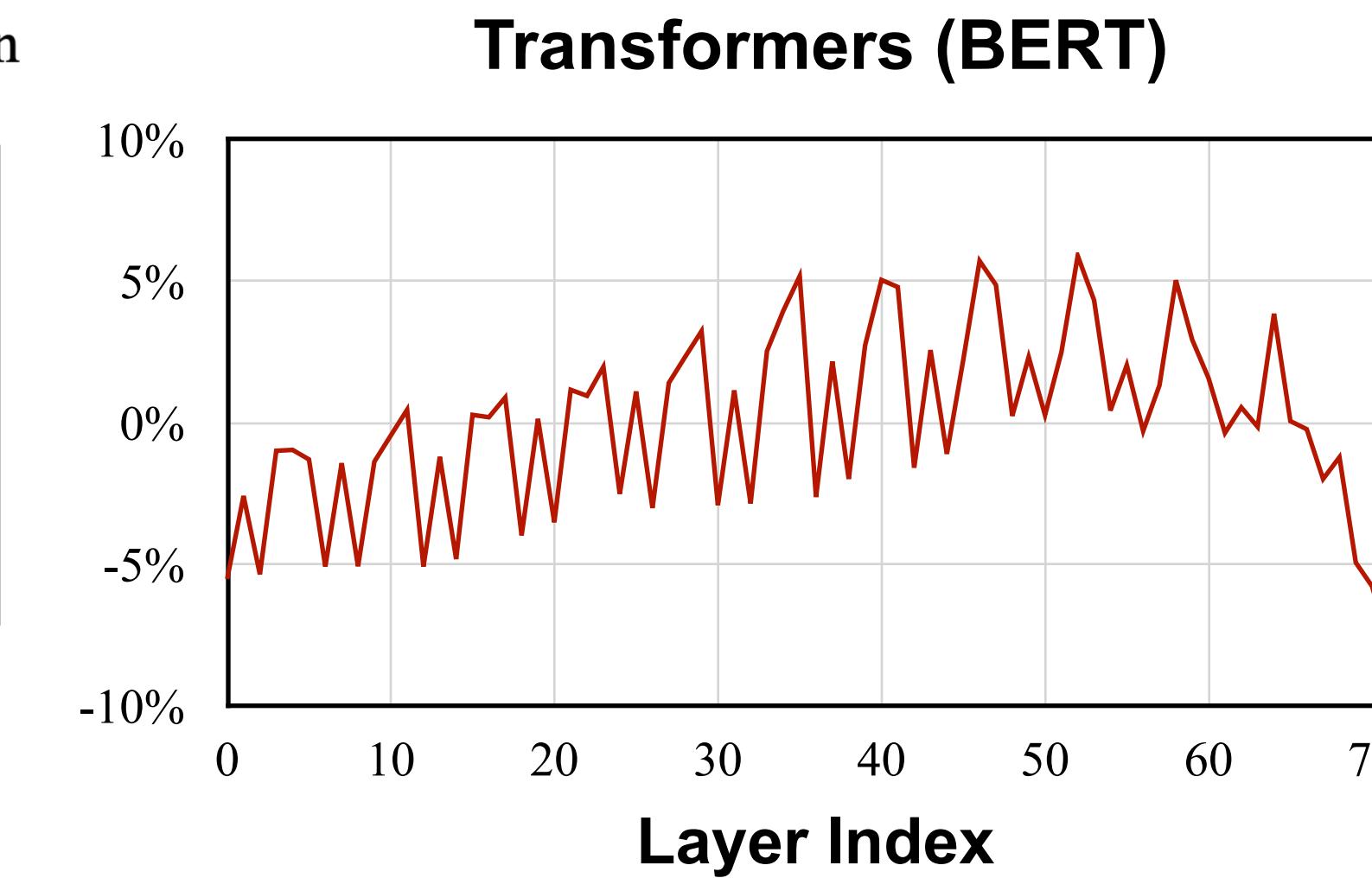
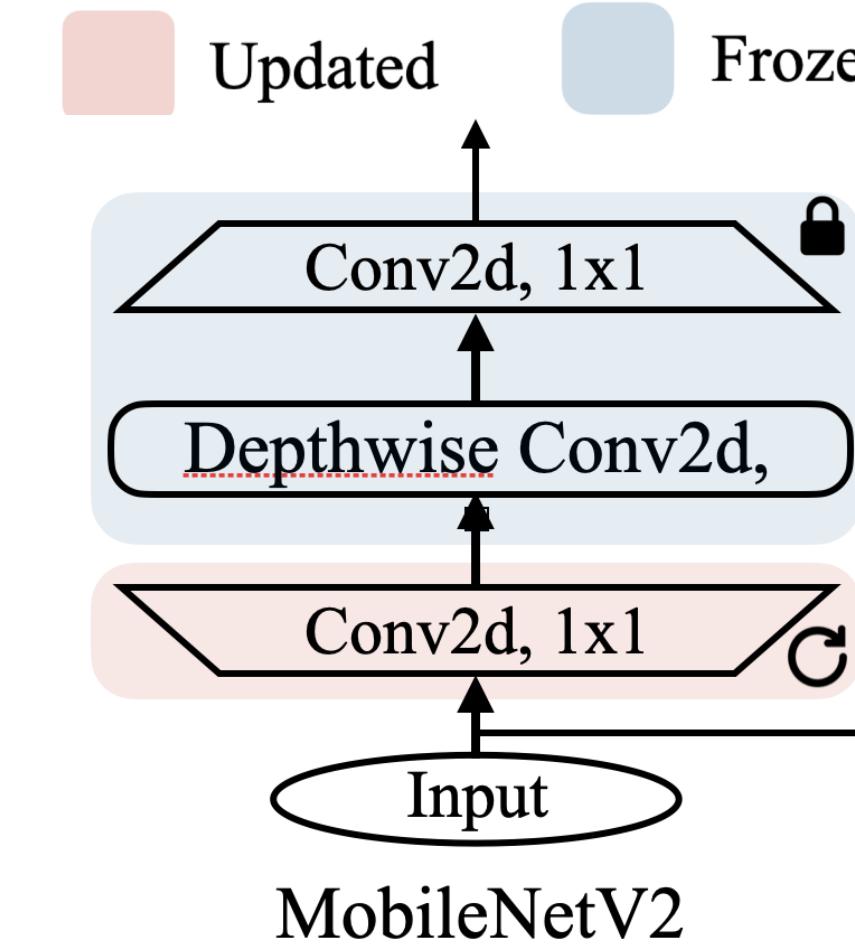
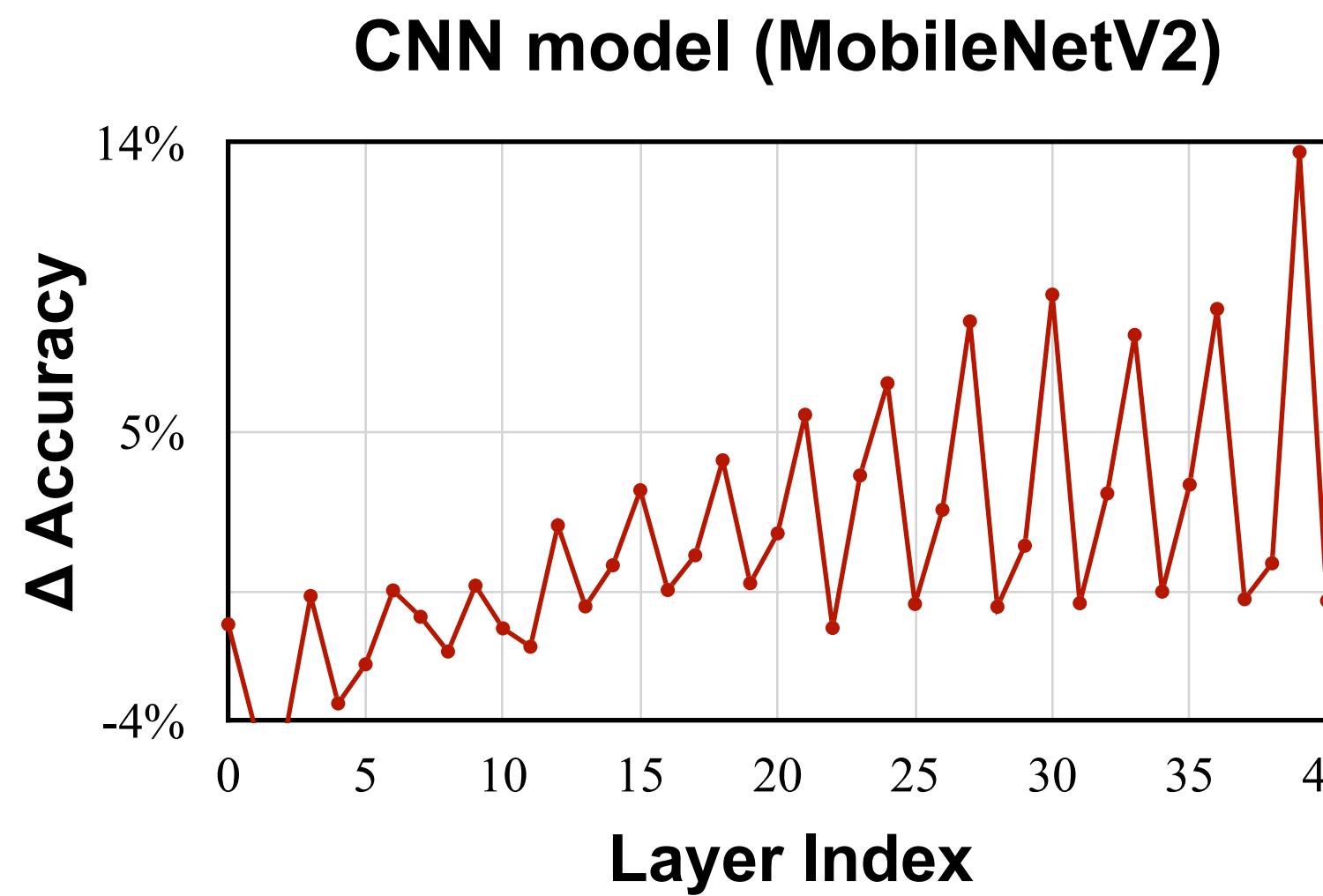
Weight in SRAM: (0.25\*M, H)

Reduce by 4x

# Sparse Layer/Tensor Back-propagation

## Searching Important Layers to Sparsely Update

- Fine-tune each layer on a downstream task to measure accuracy improvement as **contributions**



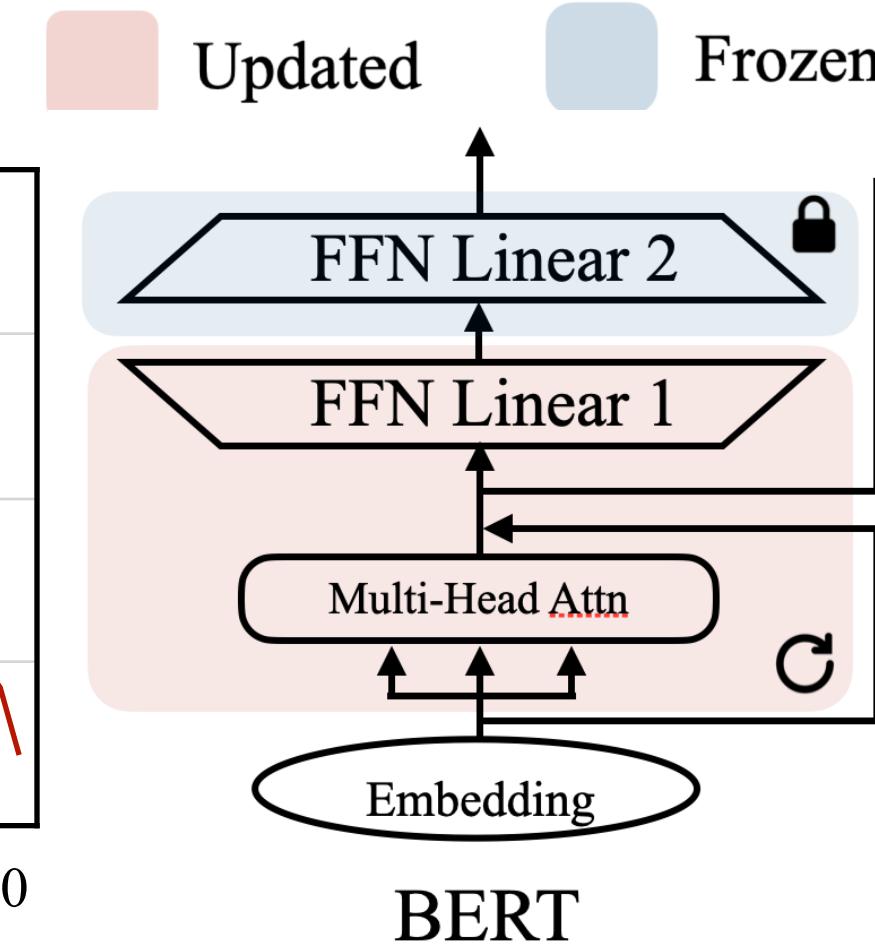
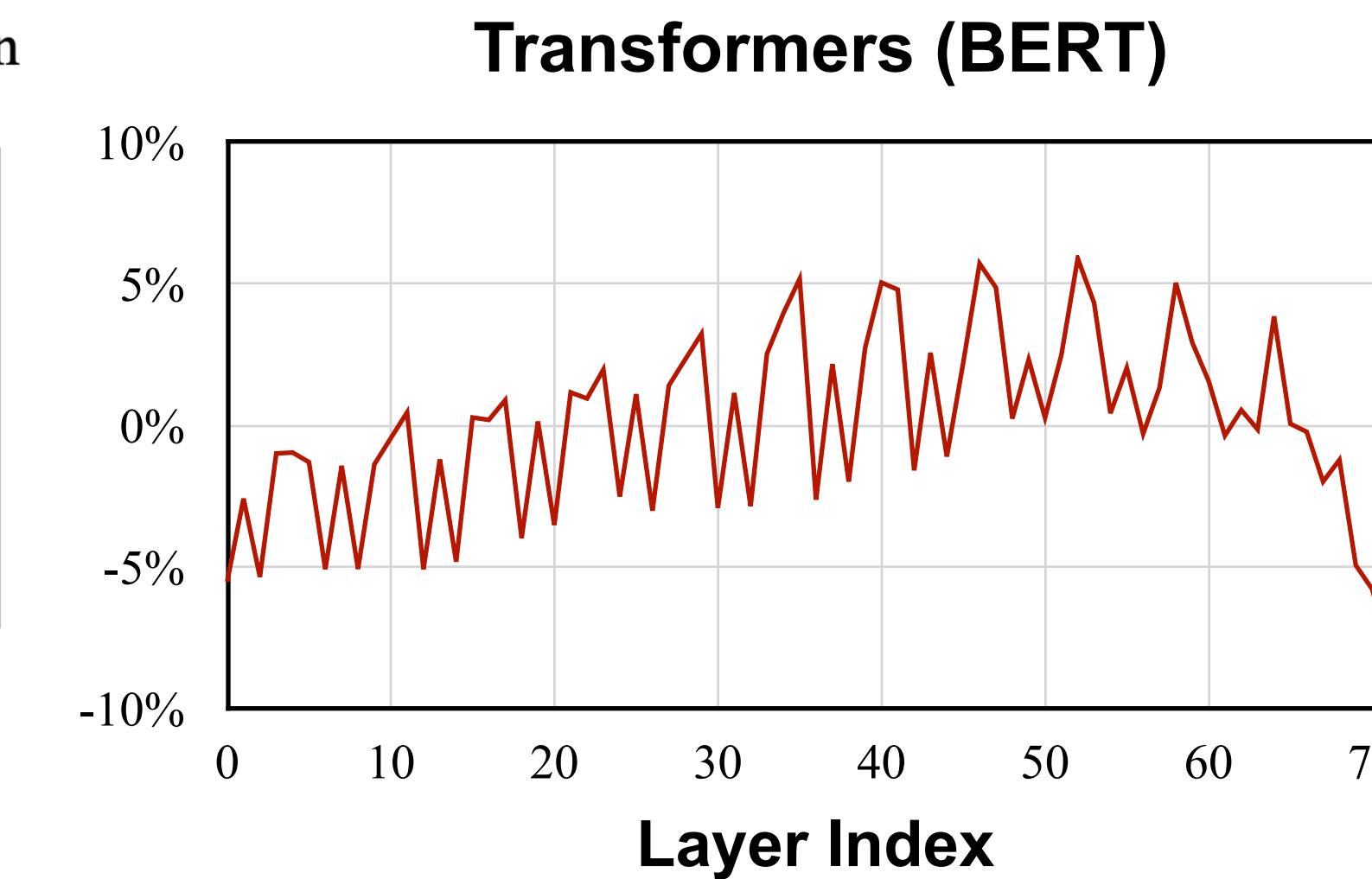
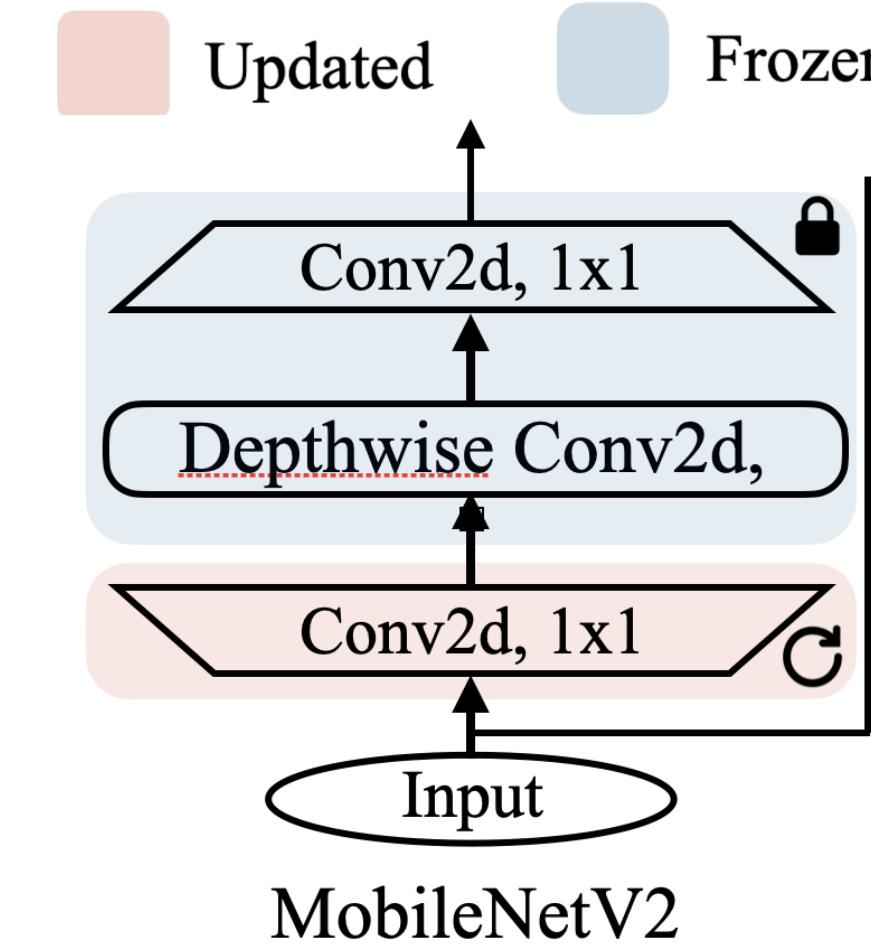
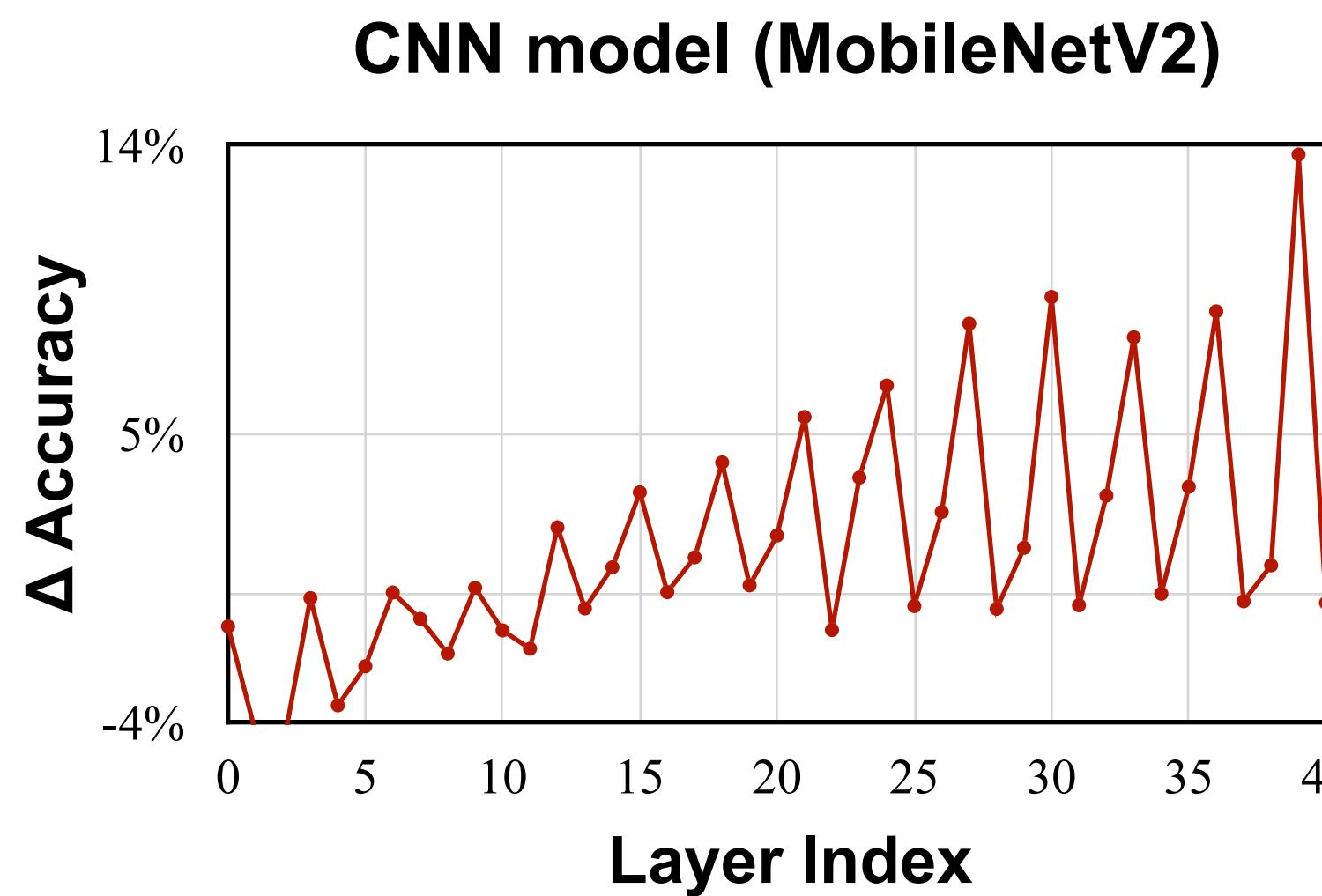
- **Later layers** are more important
- The **first point-wise conv** in each block contributes more

- **Middle layers** are more important
- **Attention and first FFN layers** contribute more.

# Sparse Layer/Tensor Back-propagation

## Searching Important Layers to Sparsely Update

- Fine-tune each layer on a downstream task to measure accuracy improvement as **contributions**



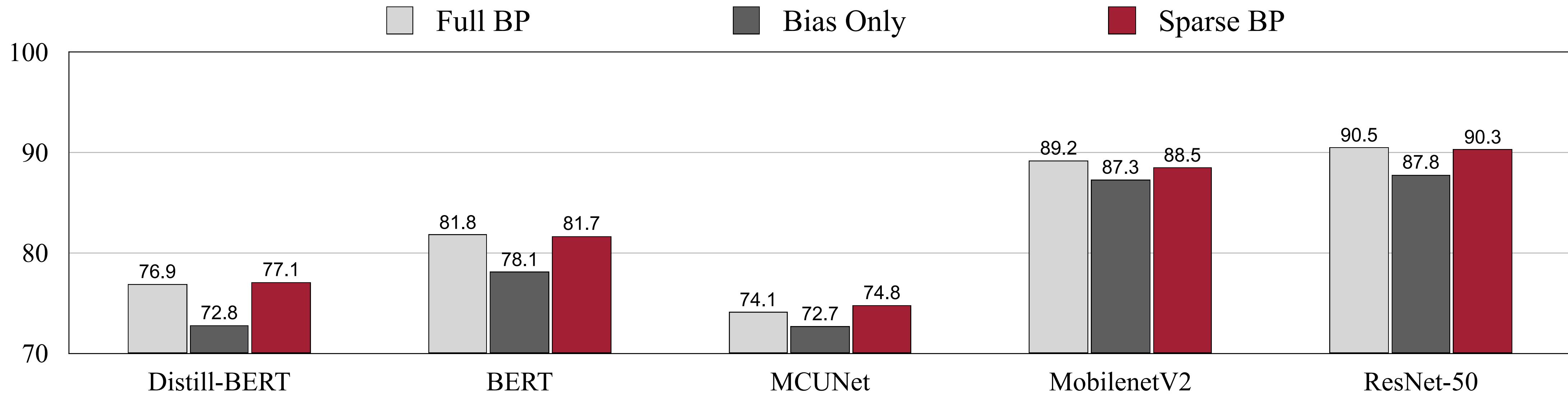
$$k^*, i^*, r^* = \max_{k, i, r} \left( \sum_{k \in i} \Delta acc_{b_k} + \sum_{i \in i, r \in r} \Delta acc_{W_{i,r}} \right) \text{ s.t. } \text{Memory}(k, i, r) \leq \text{constraint}$$

Where  $k$  is the bias index,  $i$  is the weight index and  $r$  is sparsity ratio.

- Use **evolution search** to find the sparse update scheme.
- Find the **pareto-optimal** for best accuracy-cost tradeoff.

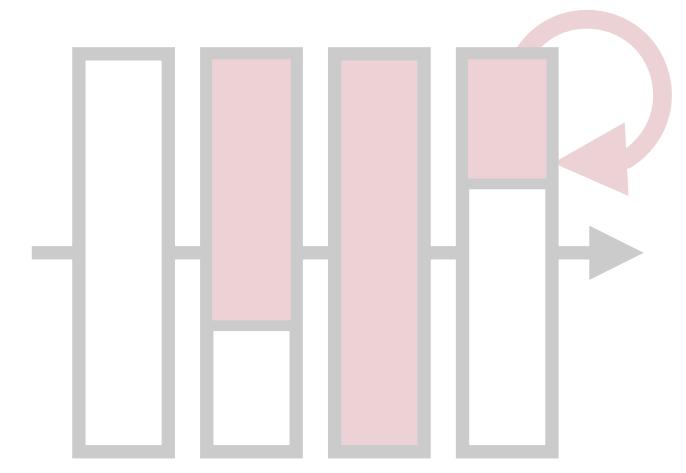
# Sparse Layer/Tensor Back-propagation

## Searching Important Layers to Sparsely Update

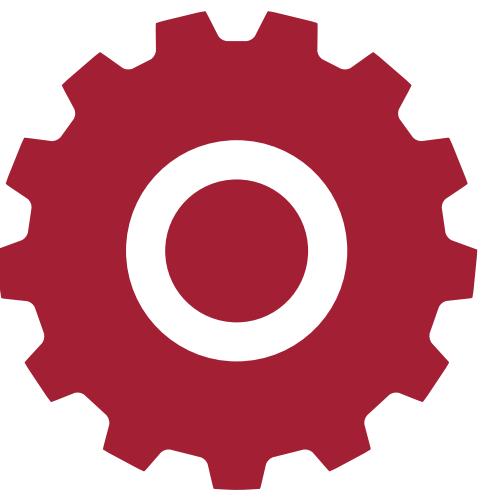


- The accuracy on DistillBERT and BERT is average from GLUE Benchmark.
- The accuracy on MCUNet, MobilenetV2, ResNet-50 is average from TinyTL Benchmark.
- Sparse-BP demonstrates **on-par performance with Full-BP** on both vision and language tasks.

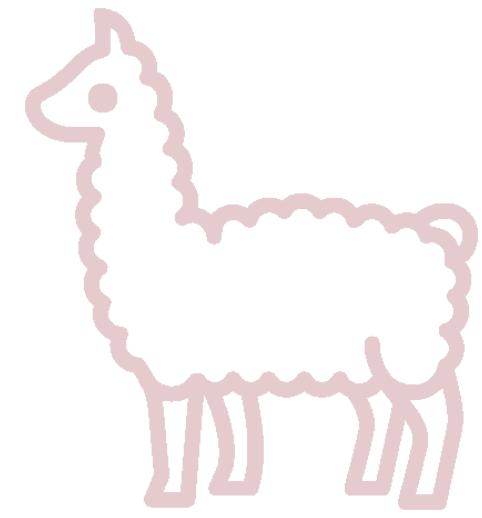
# PockEngine



1. Sparse Layer/Tensor  
Back-propagation



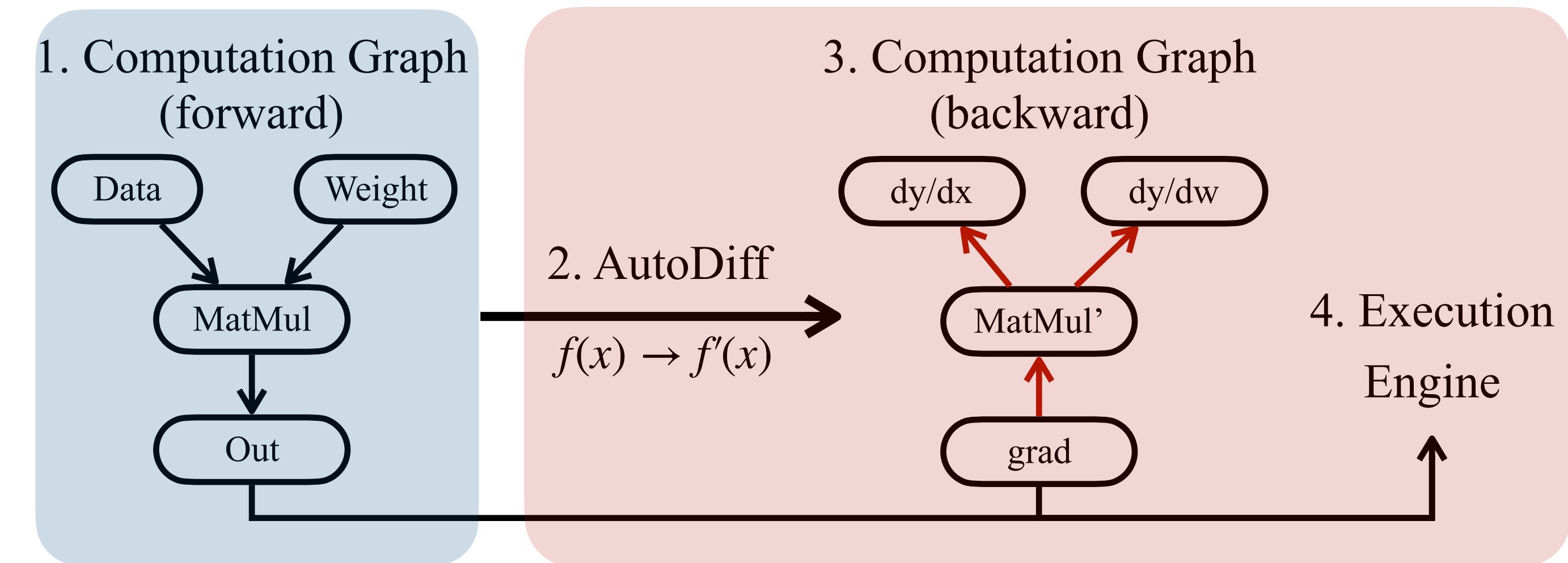
2. Compilation-Based



3. Fine-tune Llama2 on Orin

# PockEngine: Compilation-Based

## The Workflow of Conventional Training Framework

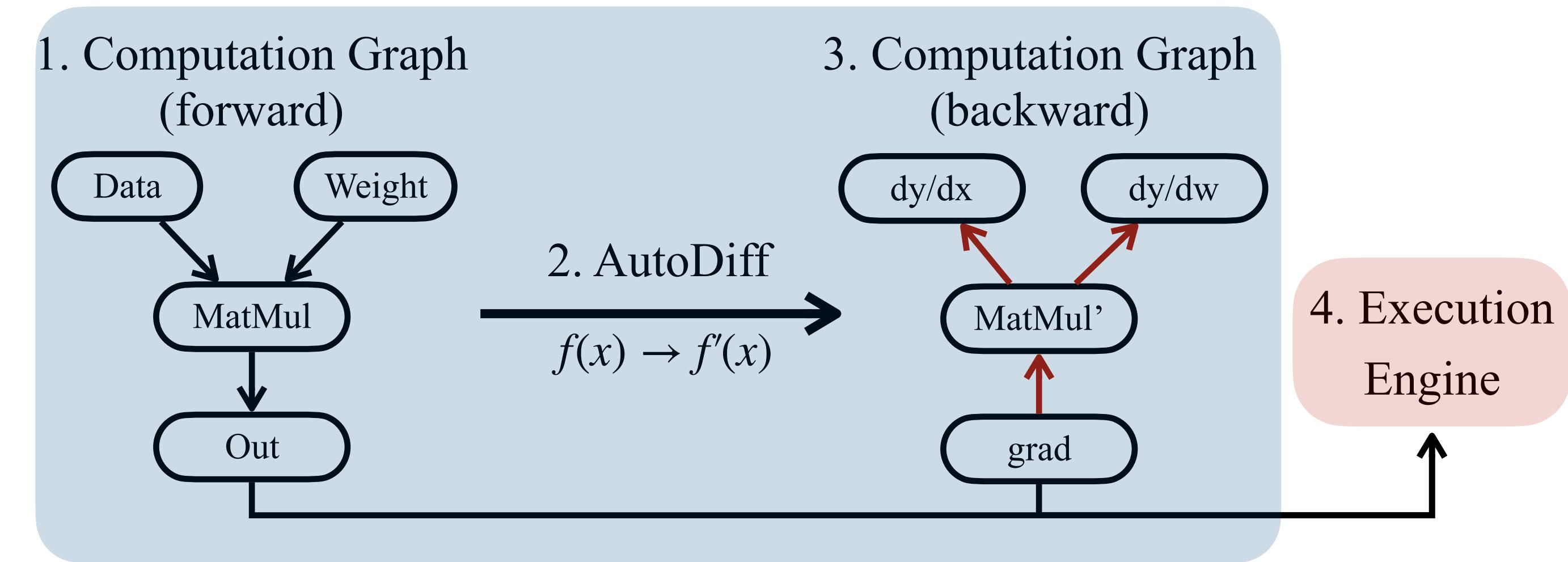


- : Compile-Time
- : Runtime

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

# PockEngine: Compilation-Based

## The Workflow of PockEngine



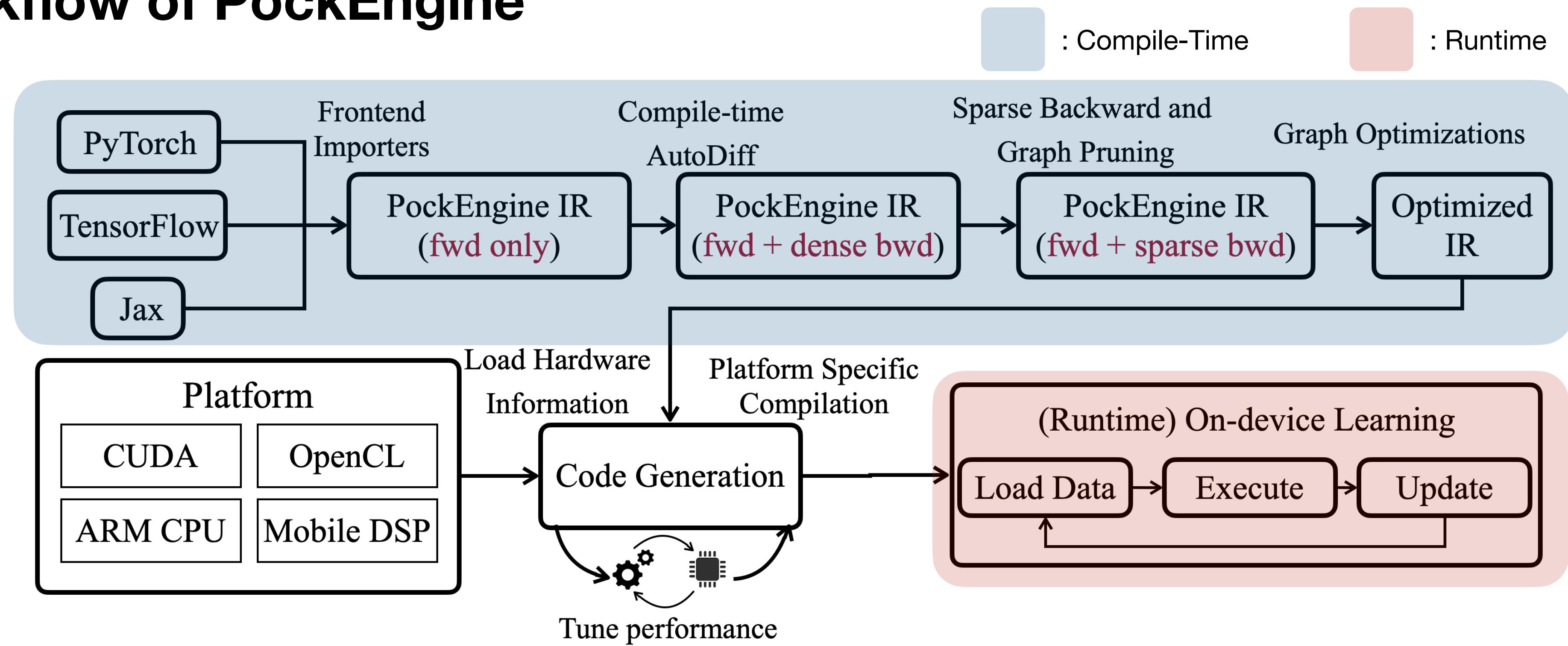
- : Compile-Time
- : Runtime

PockEngine moves most workload (like autodiff, memory scheduling, execution planning) from runtime to **compile-time**, thus minimizes the **runtime overhead**

This also enables opportunities for **extensive graph optimizations**.

# PockEngine: Compilation-Based

## The Workflow of PockEngine

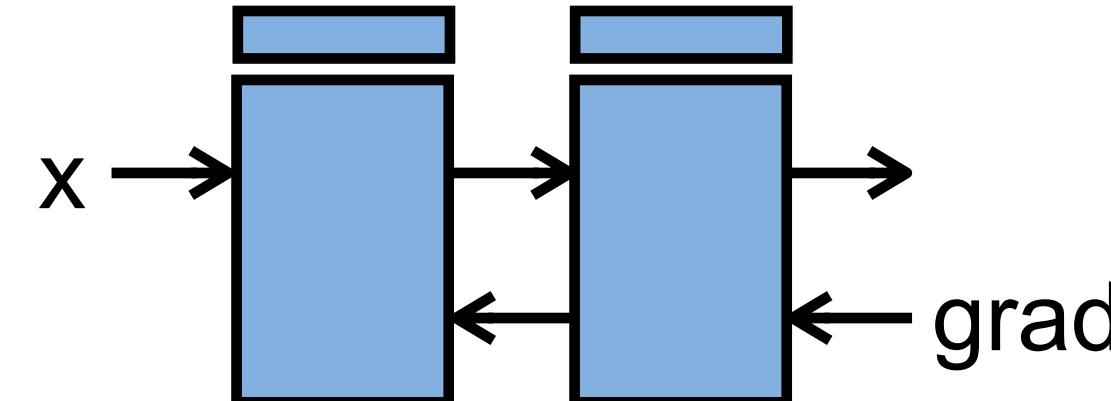


PockEngine features:

- Offload AutoDiff from runtime to compile-time, thus provides global information during compiling
- Enable various and extensive graph optimizations during compilation, improve runtime efficiency
- Support Diverse Frontends/Backends

# PockEngine: Compilation-Based

## Step 1 - Translate Models into a unified IR

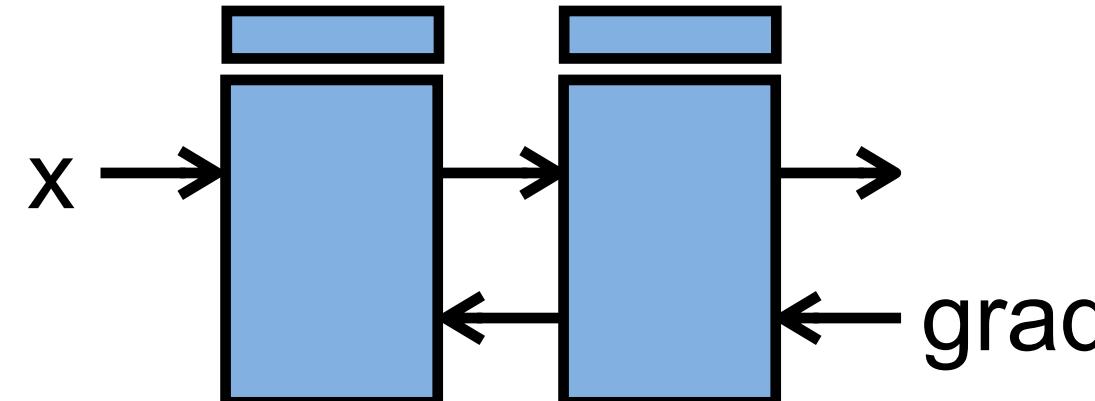


(a) Two-layer NN

Consider a simple two layer Neural Network

# PockEngine: Compilation-Based

## Step 1 - Translate Models into a unified IR



(a) Two-layer NN

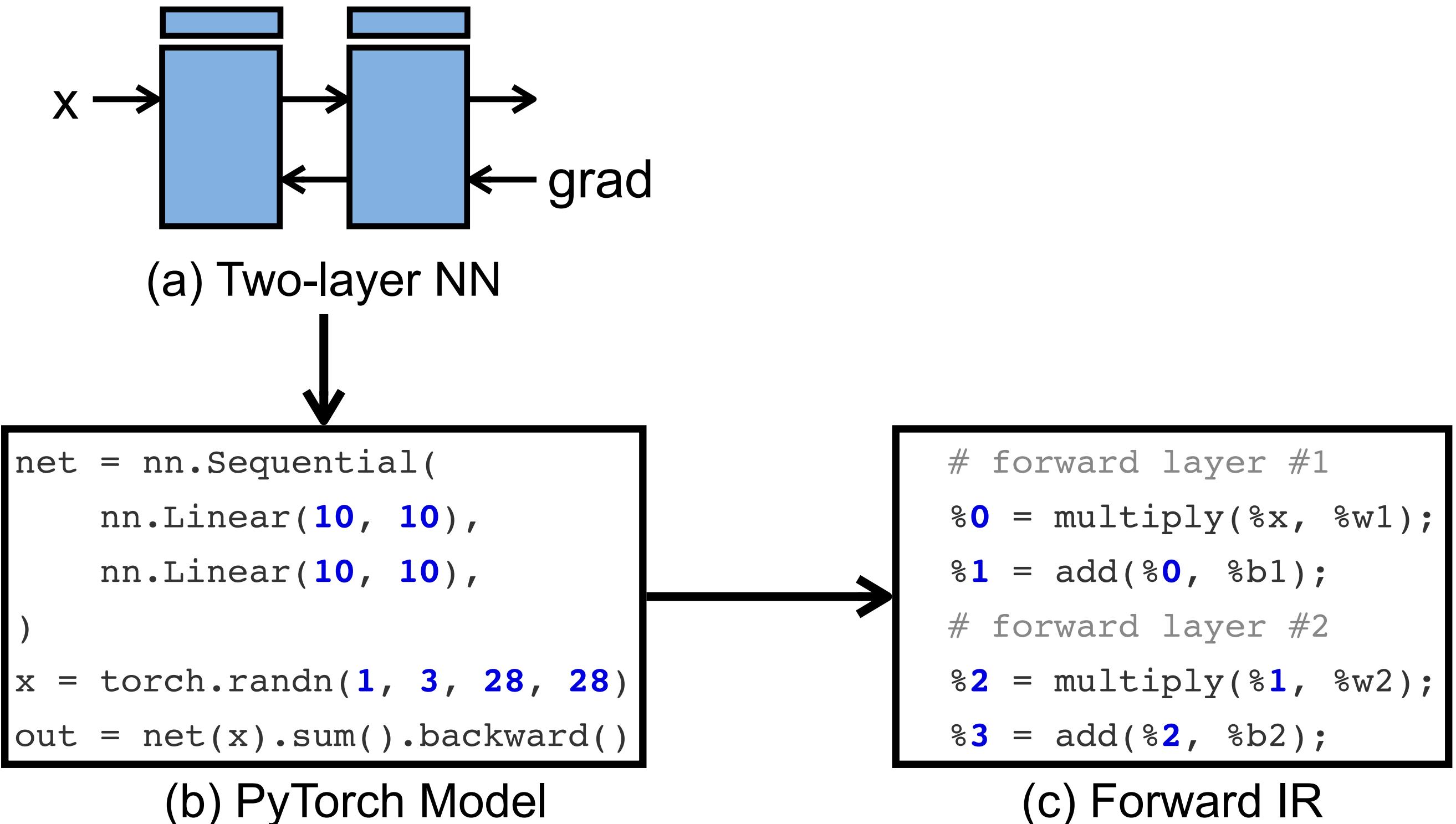
```
net = nn.Sequential(  
    nn.Linear(10, 10),  
    nn.Linear(10, 10),  
)  
x = torch.randn(1, 3, 28, 28)  
out = net(x).sum().backward()
```

(b) PyTorch Model

PockEngine takes models defined in PyTorch/TensorFlow/Jax

# PockEngine: Compilation-Based

## Step 1 - Translate Models into a unified IR



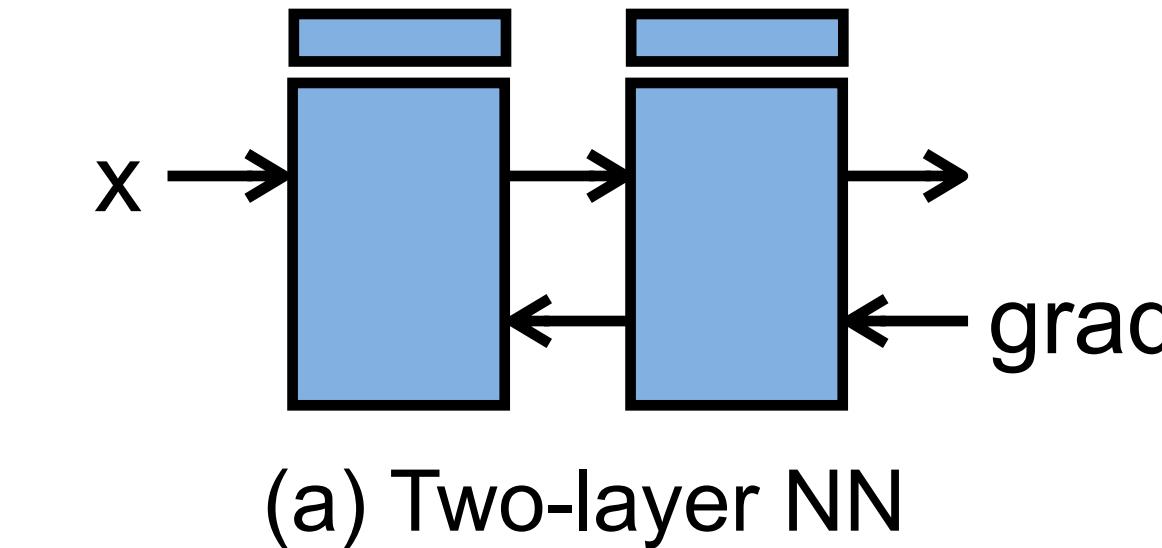
(b) PyTorch Model

(c) Forward IR

PockEngine translates the models into IR where graph optimizations can be applied.

# PockEngine: Compilation-Based

## Step 2 - AutoDiff the Graph at Compile Time



(a) Two-layer NN

```
net = nn.Sequential(  
    nn.Linear(10, 10),  
    nn.Linear(10, 10),  
)  
x = torch.randn(1, 3, 28, 28)  
out = net(x).sum().backward()
```

(b) PyTorch Model

```
# forward layer #1  
%0 = multiply(%x, %w1);  
%1 = add(%0, %b1);  
# forward layer #2  
%2 = multiply(%1, %w2);  
%3 = add(%2, %b2);
```

(c) Forward IR

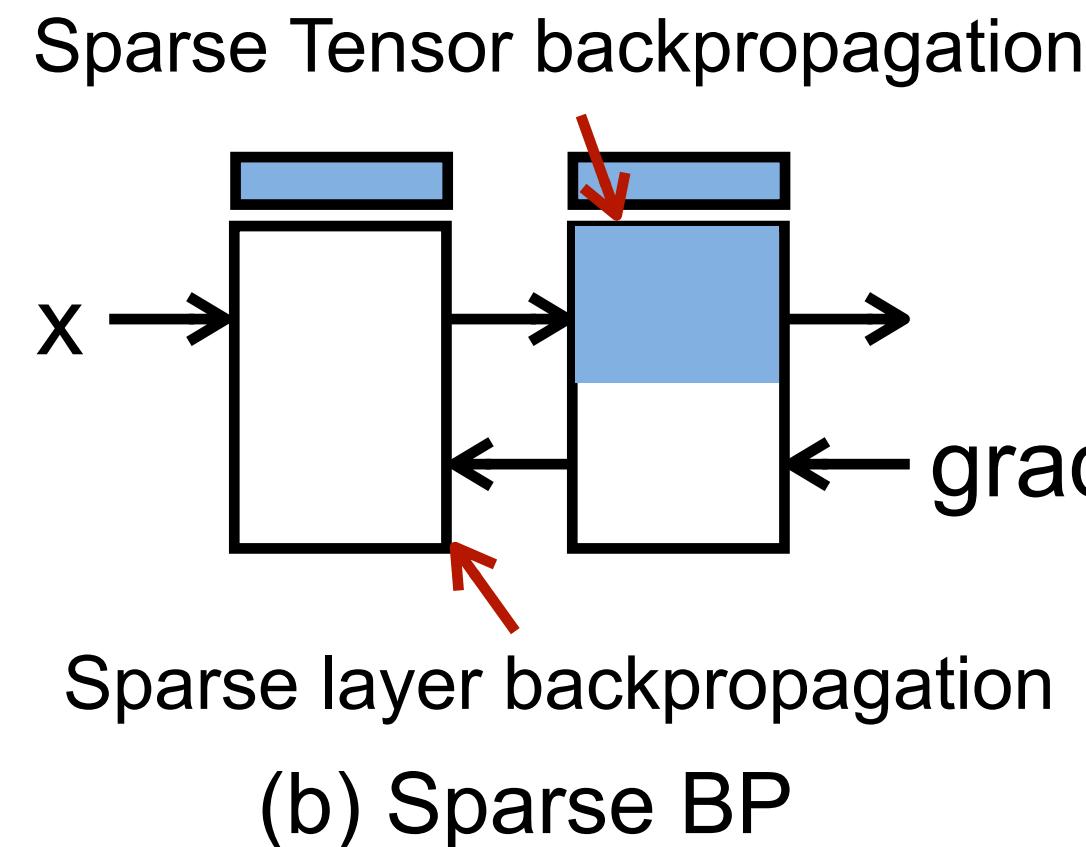
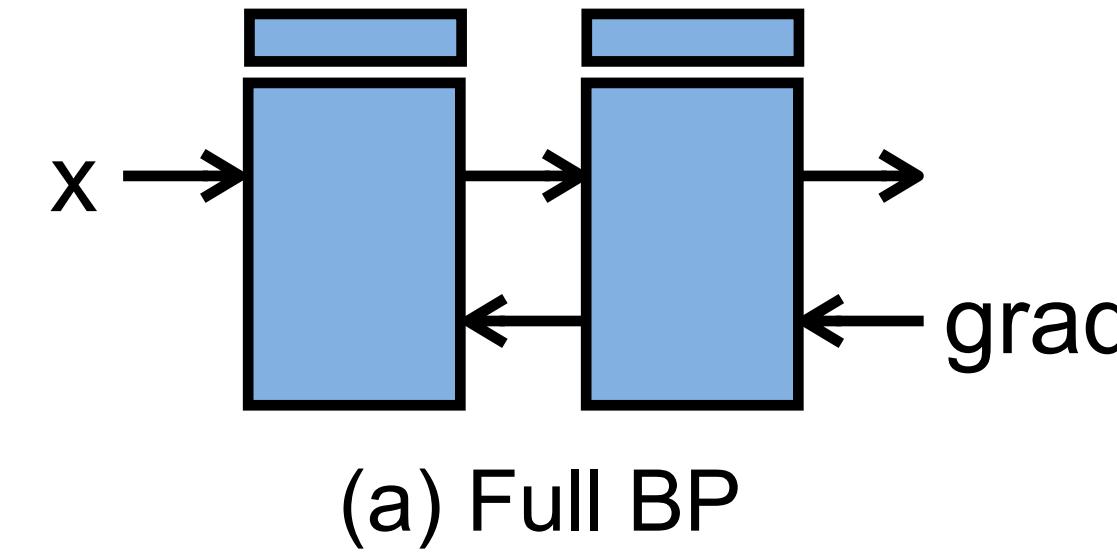
```
# forward layer #1  
%0 = multiply(%x, %w1);  
%1 = add(%0, %b1);  
# forward layer #2  
%2 = multiply(%1, %w2);  
%3 = add(%2, %b2);  
# backward layer #2  
%4 = multiply(%grad, %w2);  
%5 = transpose(%grad);  
%6 = multiply(%5, %1);  
%7 = sum(%grad, axis=-1);  
# backward layer #1  
%8 = multiply(%6, %w1);  
%9 = transpose(%6);  
%10 = multiply(%9, %x);  
%11 = sum(%6, axis=-1);  
return (%6, %7, %10, %11)
```

(d) Forward + Backward IR

Instead of binding each operator with a backward implementation (e.g., `matmul`, `matmul_backward`), PockEngine uses the same set of primitive operations as inference to construct the training graph

# PockEngine: Compilation-Based

## Step 3 - Graph Pruning and Sparse Back-Propagation



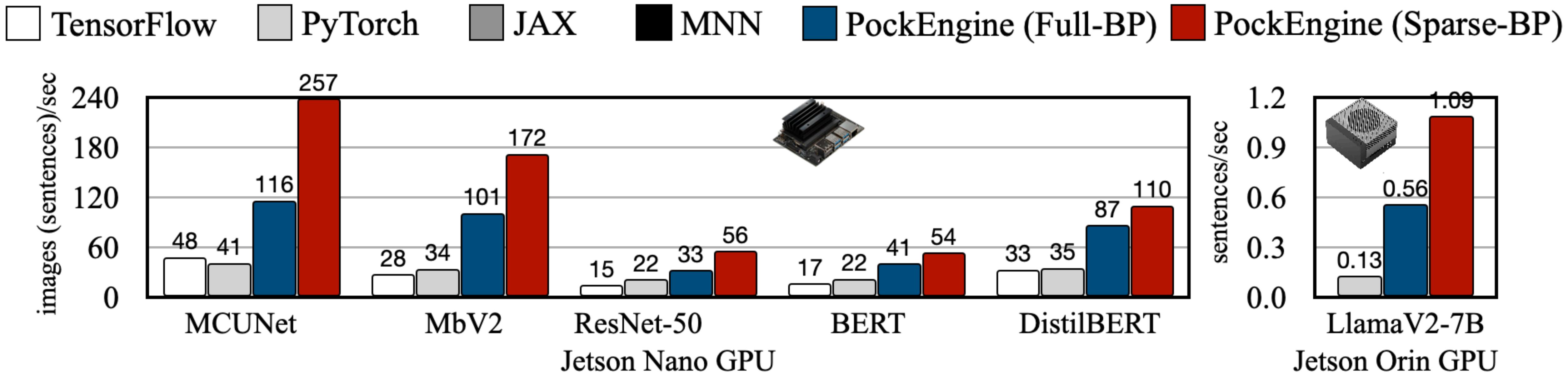
```
# forward layer #1  
%0 = multiply(%x, %w1);  
%1 = add(%0, %b1);  
# forward layer #2  
%2 = multiply(%1, %w2);  
%3 = add(%2, %b2);  
# backward layer #2  
%4 = multiply(%grad, %w2);  
%5 = transpose(%grad);  
%6 = multiply(%5, %1);  
%7 = sum(%grad, axis=-1);  
# backward layer #1  
%8 = multiply(%6, %w1);  
%9 = transpose(%6);  
%10 = multiply(%9, %x);  
%11 = sum(%6, axis=-1);  
return (%6, %7, %10, %11)
```

```
# forward layer #1  
%0 = multiply(%x, %w1);  
%1 = add(%0, %b1);  
# forward layer #2  
%2 = multiply(%1, %w2);  
%1.1 = slice(%1, range=[0:10, 0:10]);  
%3 = add(%2, %b2);  
# backward layer #2 [Sparse Tensor BP]  
%4 = multiply(%grad, %w2);  
%5 = transpose(%grad);  
%6 = multiply(%5, %1.1);  
%7 = sum(%grad, axis=-1);  
# backward layer #1 [Sparse Layer BP]  
%8 = multiply(%6, %w1);  
%9 = transpose(%6);  
%10 = multiply(%9, %x);  
%11 = sum(%6, axis=-1);  
return (%6, %7, %10, %11)
```

Training DAG is automatically transformed and trimmed using Dead Code Elimination

# PockEngine: Compilation-Based

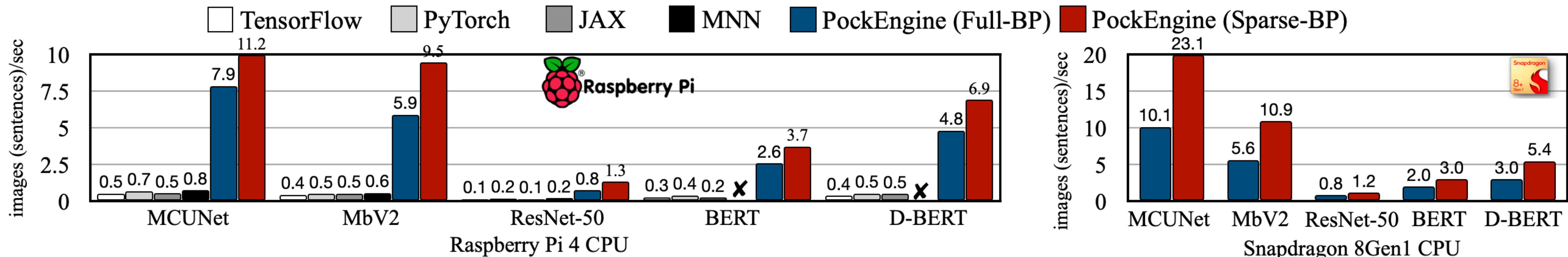
## Speedup Comparison: Edge GPU



- **2.2x to 2.6x speedup** on Jetson Nano, and **4.4x speedup** on Orin.
- Main speedup from compilation as the host language (Python) is slow on low-free CPUs.
- Other frameworks (e.g., TensorRT) may also achieve this, but do provide the training support.

# PockEngine: Compilation-Based

## Speedup Comparison: Edge CPU

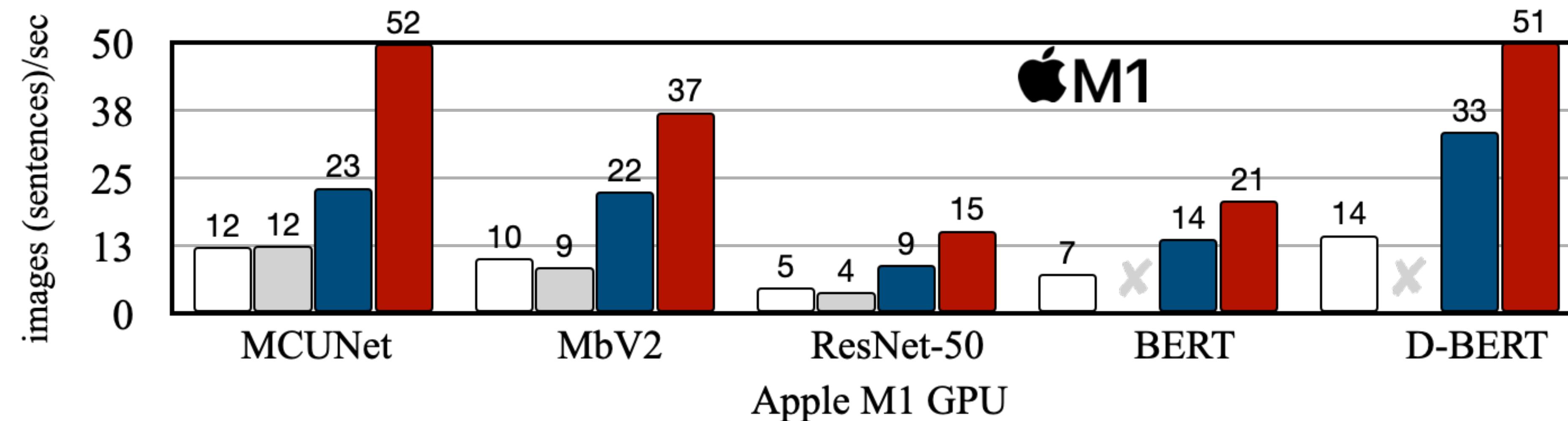


- PockEngine shows **13 to 21x speedup** on Raspberry Pi 4 B+ platforms.
  - Speedup mainly from kernel tuning
  - Most kernel implementations focus on GPU and x86 CPU with AVX instructions (PyTorch, Jax)
  - Framework Optimize most for the inference pipeline and operators (MNN).

# PockEngine: Compilation-Based

## Speedup Comparison: Apple M-Chip

TensorFlow PyTorch JAX MNN PockEngine (Full-BP) PockEngine (Sparse-BP)

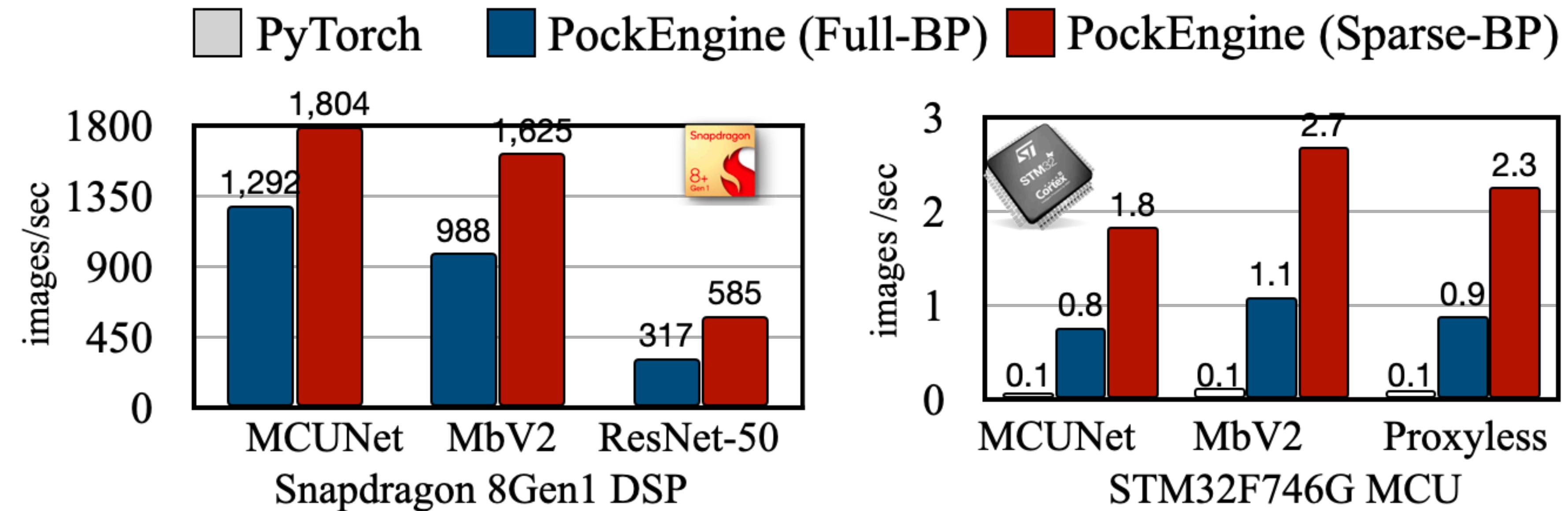


- Apple M1/2 is a new, and the **compatibility is not ideal** for PyTorch and TensorFlow:
  - For PyTorch\*, even with the a recent build (commit ID: c9913cf), transformer training throws errors on M1.
  - For TensorFlow, the GPU training support is preliminary and incomplete on M1.
- PockEngine compiles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

\*<https://github.com/pytorch/pytorch/issues/77764>

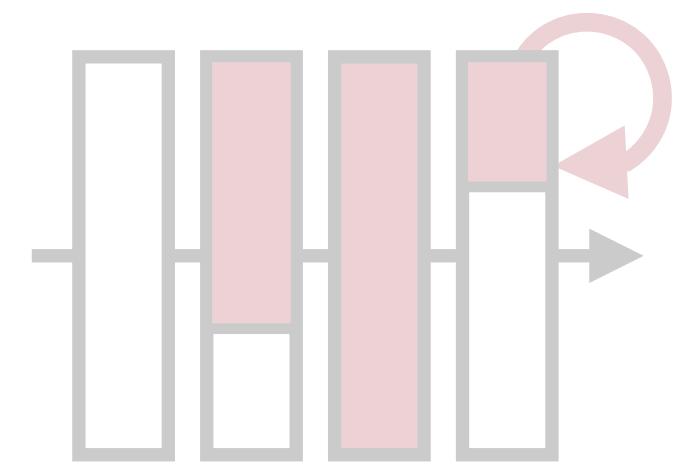
# PockEngine: Compilation-Based

## Speedup Comparison: DSP and MCUs

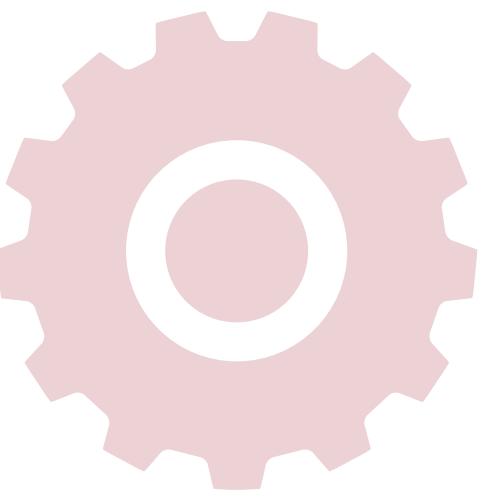


- We integrate SNPE for Qualcomm DSPs and TinyEngine for Micro-Controllers
  - On MCU, Pock-Engine is much faster than TF-Lite-Micro (projected latency).
  - On DSP, the peak performance is impressive and even on par with edge GPUs.
- PockEngine's compilation workflow is (1) kernel agnostic and (2) shares the same set of OPs between FWD and BWD, thus enables **previous inference-only framework to support training**.

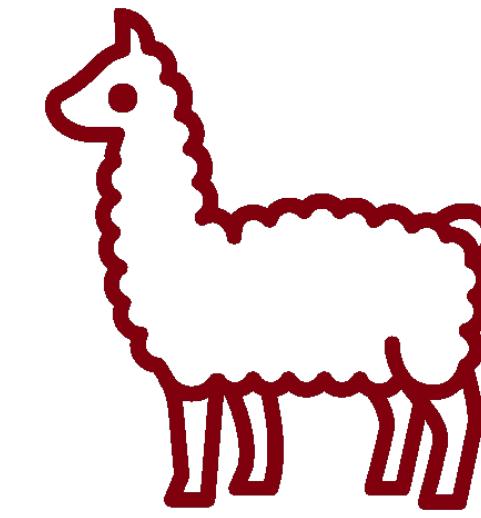
# PockEngine



1. Sparse Layer/Tensor  
Back-propagation



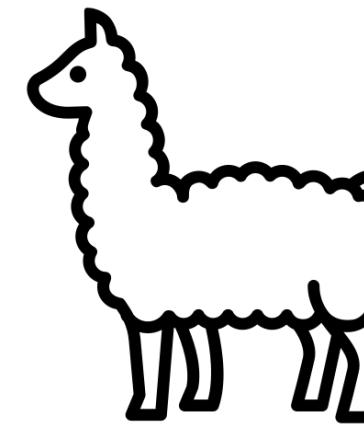
2. Compilation-Based



3. Fine-tune Llama2 on Orin

# PockEngine: Build your own ChatBot

## LLMs Fine-tuning Setup



Model: Llama-V2

	Params	Dimensions	N heads	N layers	Pretrain Tokens	Content Length
	6.7B	4096	32	32	2.0T	4K
	13.0B	5120	40	40	2.0T	4K
	32.5B	6656	52	60	2.0T	4K

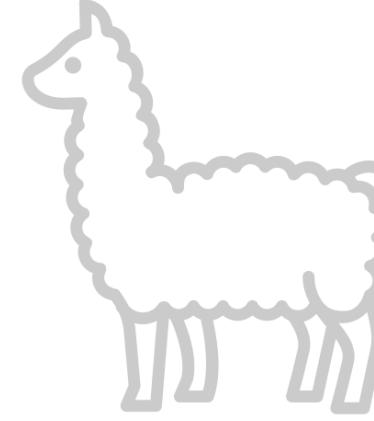
We choose the Llama-V2 foundation model from Meta, and adapt the 7B model from the Llama family.

We use LION optimizer to efficiently tune the model, with LR  $10^{-4}$  and gradient accumulation.

[1] LLaMA: Open and Efficient Foundation Language Models  
[2] Llama 2: Open Foundation and Fine-Tuned Chat Models

# PockEngine: Build your own ChatBot

## LLMs Fine-tuning Setup



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.

A common hypothesis is that a model's knowledge and capabilities are learnt almost entirely during pre-training, while fine-tuning / alignment teaches 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

# PockEngine: Build your own ChatBot

## Quantitive Analysis

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

- **PockEngine Accelerates Training:**
  - Fine-tuning on 1000 records would require 2 hours using PyTorch while PockEngine can finish in less than half an hour (**4.4x**).

# PockEngine: Build your own ChatBot

## Quantitive Analysis

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 Accelerates Training and Comparable Accuracy**
  - LoRA reduce the memory footprint (45.1GB → 30.9GB), but training cost is still higher.
  - Sparse BP significantly improves training speed (1768ms → 914ms, 1.9× faster).
  - Further, sparse-BP demonstrates similar performance when compared to full-BP.
- LoRA needs to **back-propagate to the very first layer** while SparseBP doesn't.

# PockEngine: Build your own ChatBot

## Qualitative Analysis - Factual Check

**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.

When asking simple factual questions, all models consistently deliver accurate responses while the tuned models also offer more comprehensive information, enriching the overall user experience.

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

# PockEngine: Build your own ChatBot

## Qualitative Analysis - Logic Inference

**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".

**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.

If not fine-tuned, then LLMs will mis-interoperate the question and performs wrong actions.

*Left:* The original model misunderstood “reverse”, leading to incorrect answers

*Right:* Without fine-tuning, the model failed to understand and only simply repeated

# PockEngine: Build your own ChatBot

## Qualitative Analysis - Math

**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  $\$33 \times 16 = \$528$  from mowing lawns. His total earnings from the 16 customers who gave him a \$10 tip is  $\$10 \times 3 = \$30$ . 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  $\$10 \times 3 = \$30$  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 LlamaV2 model struggled with math-related questions, while the Full-BP and Sparse-BP models demonstrated certain capabilities for reasoning and properly calculating the results.

# Conclusion

- We present **sparse backpropagation**, a more efficient option for fine-tuning/transfer learning without harming the accuracy
- PockEngine design is **compilation-based**, and provides **system-level support** for both **dense** and **sparse** backpropagation
- PockEngine enables many **inference-only frameworks to conduct training** and consistently **demonstrates faster performance** compared to previous frameworks on different platforms.
- Notably, PockEngine shows the ability to **fine-tune LLMs on Jetson Orin locally**, enabling the possibility for everyone to build their own ChatBot.

# Q & A?