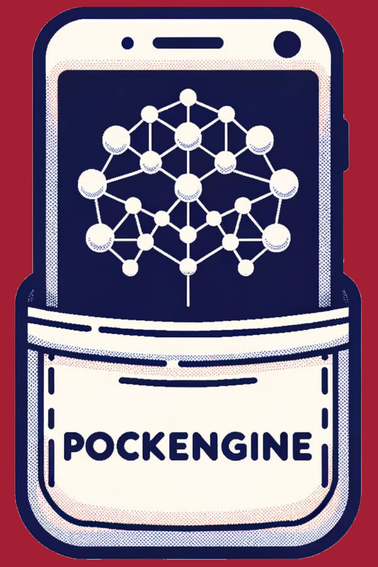
PockEngine: Sparse and Efficient Fine-tuning in a Pocket



Ligeng Zhu[1], Lanxiang Hu[2], Ji : A,o bWei-Chen Wang[1], Wei-Ming Chen[1], Chuang Gan[3], Song Han[1,4]

MIT[1], UCSD[2], MIT-IBM Watson AI Lab[3], NVIDIA[4]



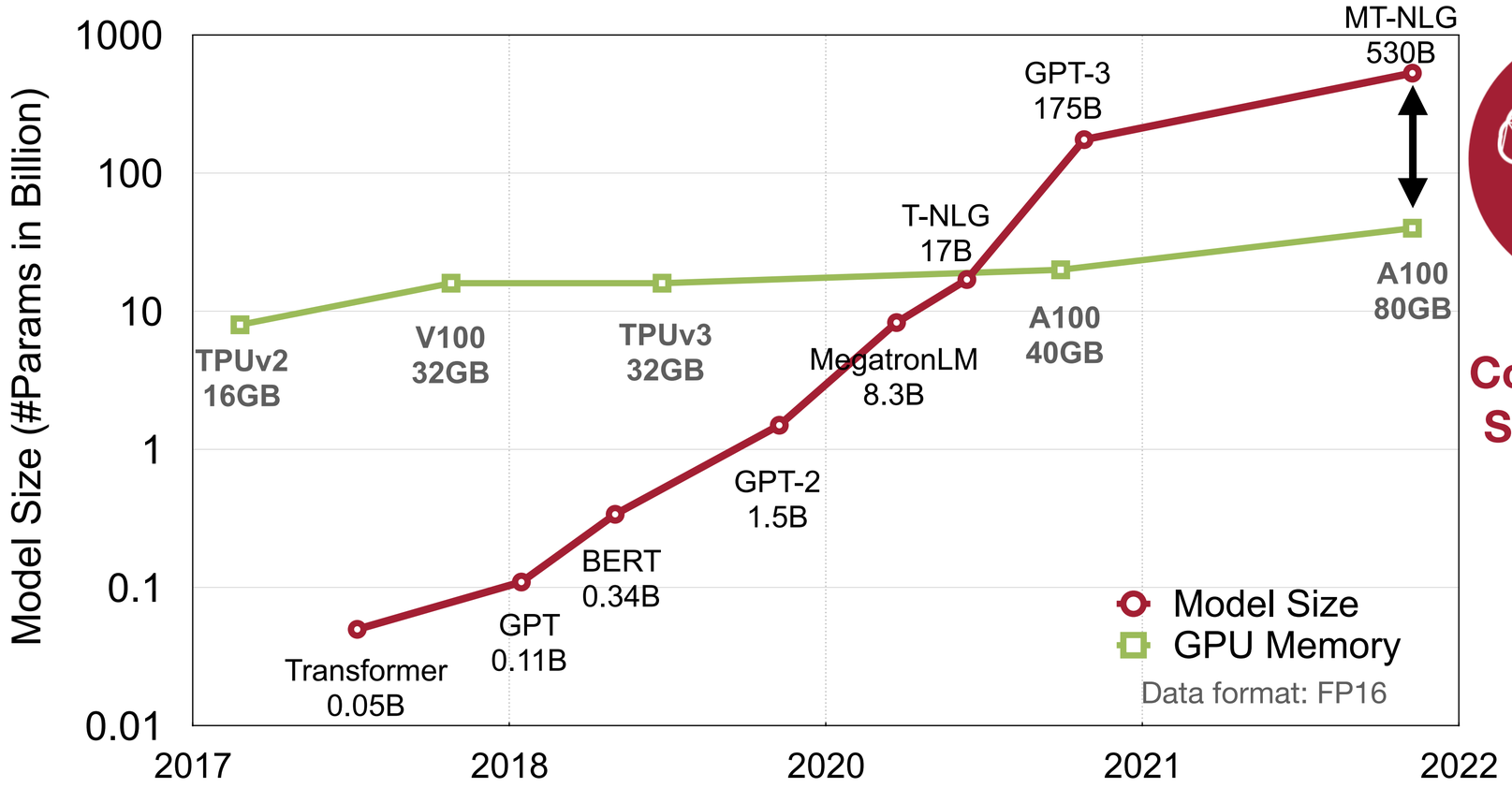




Problem: AI Models Outgrow Hardware

Model size is growing faster than Moore's law.

Co-design is essential: bridge the gap between the supply and demand of AI computing.



Model Compression, Sparsity, co-Design

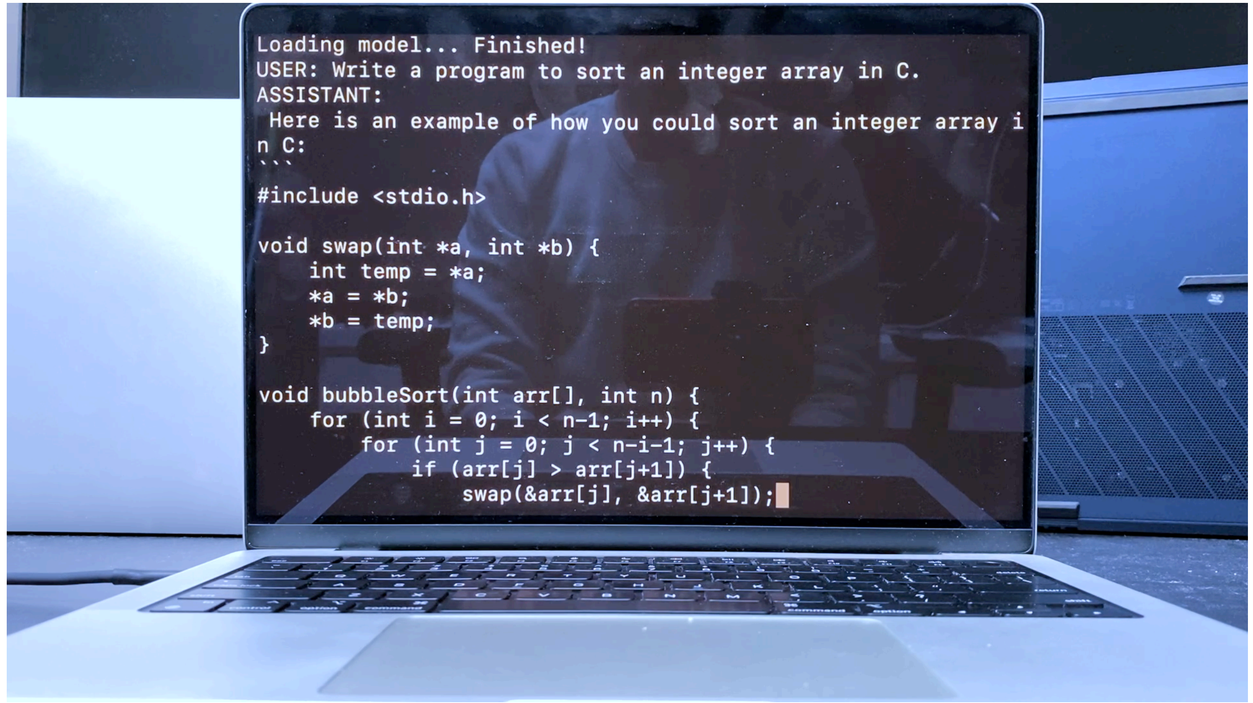
SmoothQuant, ICML' 2023

Prior Work: On-Device LLM Inference

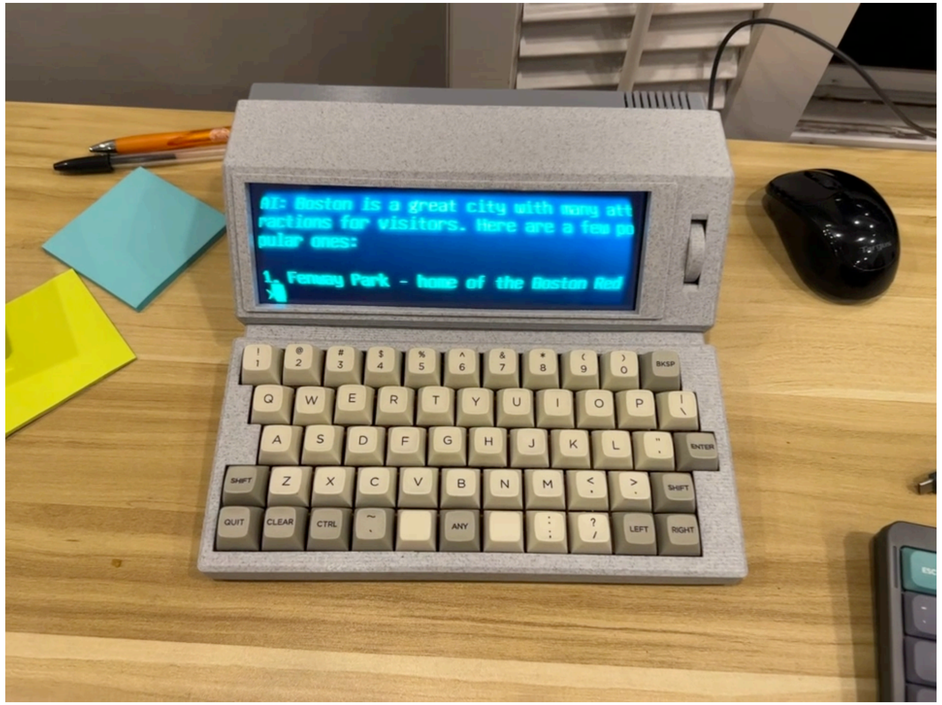
SmoothQuant, AWQ (4bit weight, 16bit arithmetic, same as EIE)

TinyChat implements the 4bit LLM, 3x measured speedup

Open sourced and integrated by NVIDIA TensorRT-LLM



- AWQ runs CodeLlama-7B on Macbook

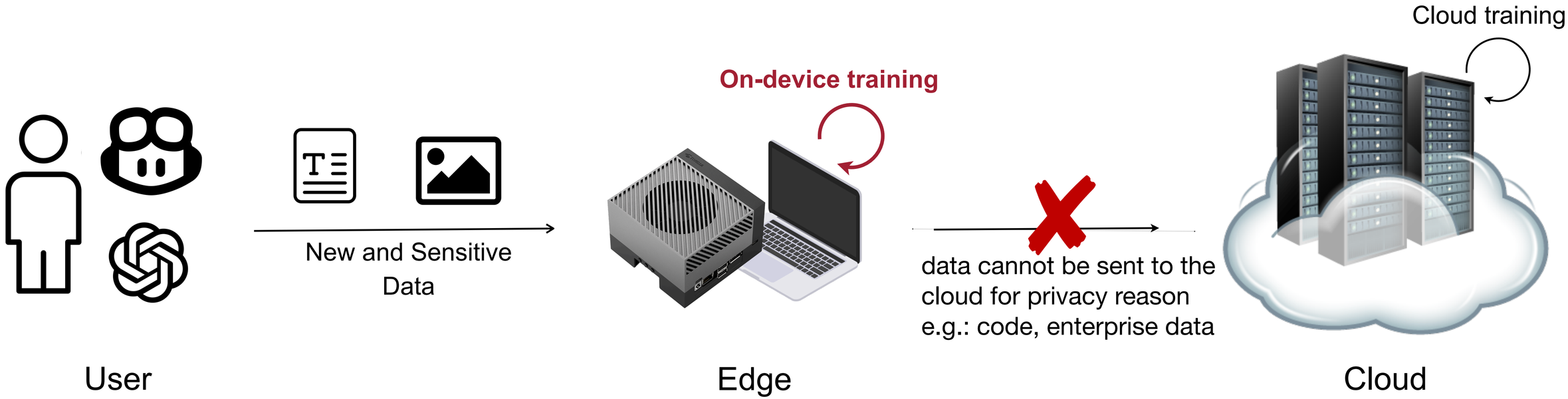


- AWQ runs Llama2-7B on Jetson Orin Nano

Can We Learn on the Edge?

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

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

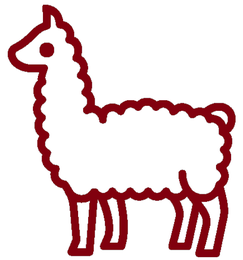


- Fine-tune a foundation model => domain-specific LLMs for law, finance, coding, etc.

PockEngine

1. Sparse Back-Propagation

2. Compiler Support



3. Fine-tune Llama2 on Orin

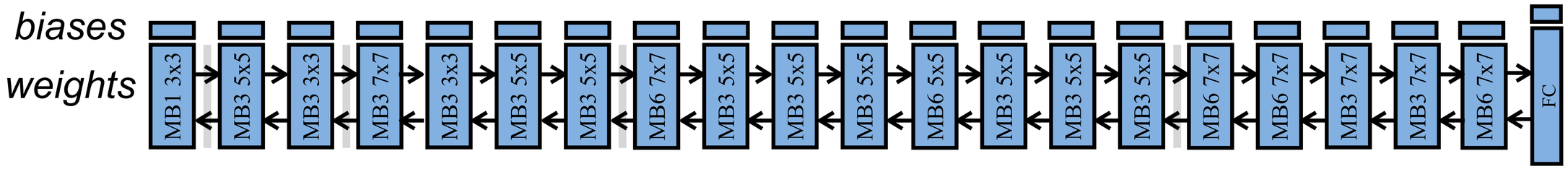
PockEngine

1. Sparse Back-Propagation

2. Compiler Support

3. Fine-tune Llama2 on Orin

Dense, Full Back-Propagation



Model: ProxylessNAS-Mobile

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

- Need to save all intermediate activations (quite large)

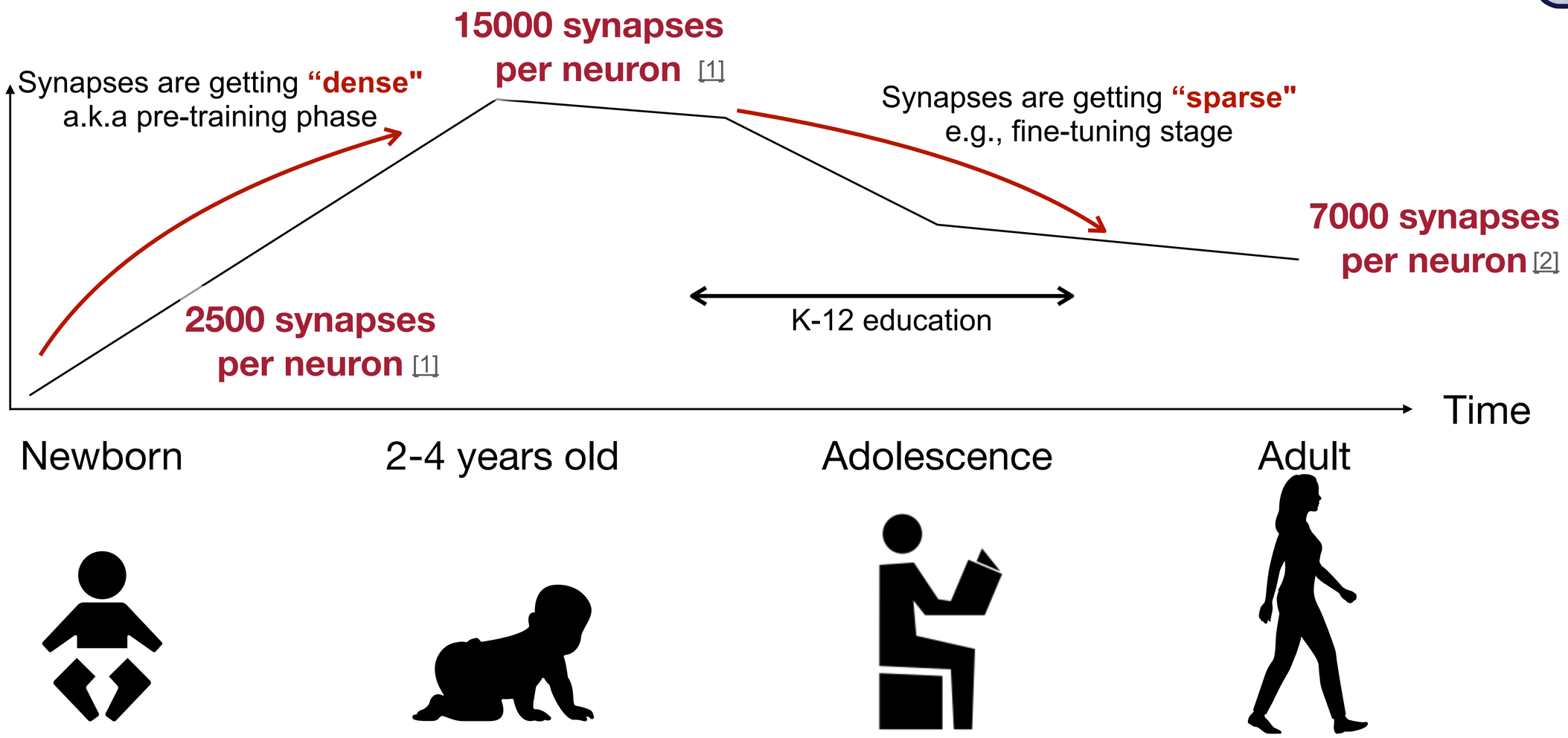
Forward: $\mathbf{a}_{i+1} = \mathbf{a}_i \mathbf{W}_i + \mathbf{b}_i$

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]

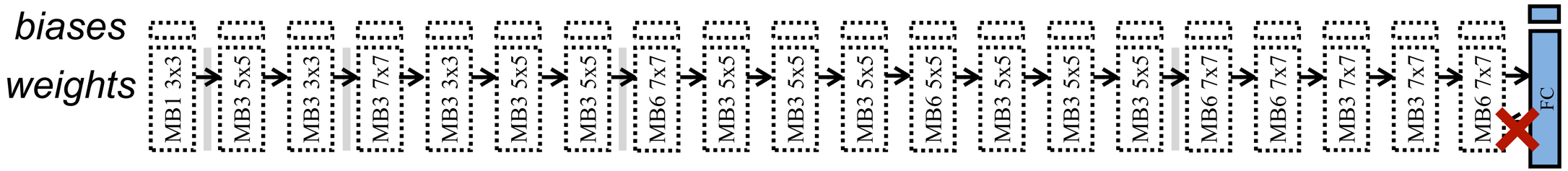
Sparse Back-Propagation



[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

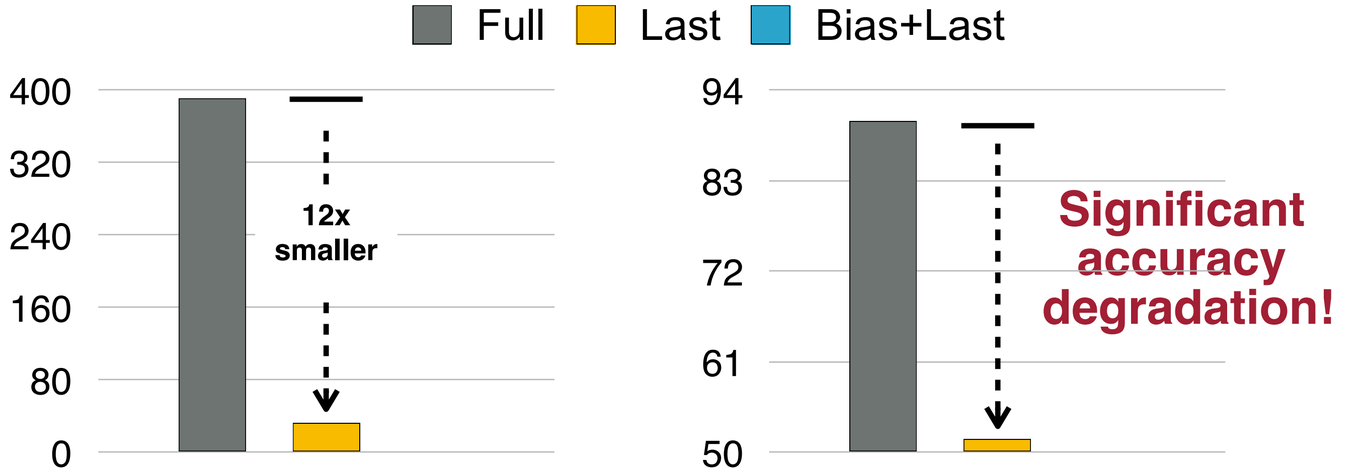
Last-Layer-Only Back-Propagation



Model: ProxylessNAS-Mobile

Updating only the last layer is cheap

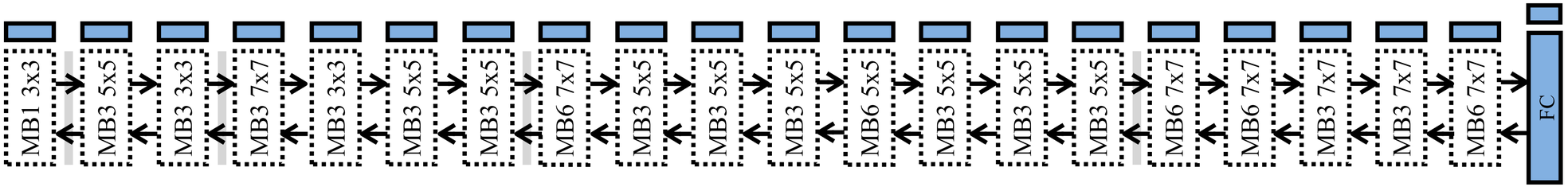
- No need to back propagate to previous layers
- But, accuracy drops significantly



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

Bias-Only Back-Propagation

LoRA is a special case



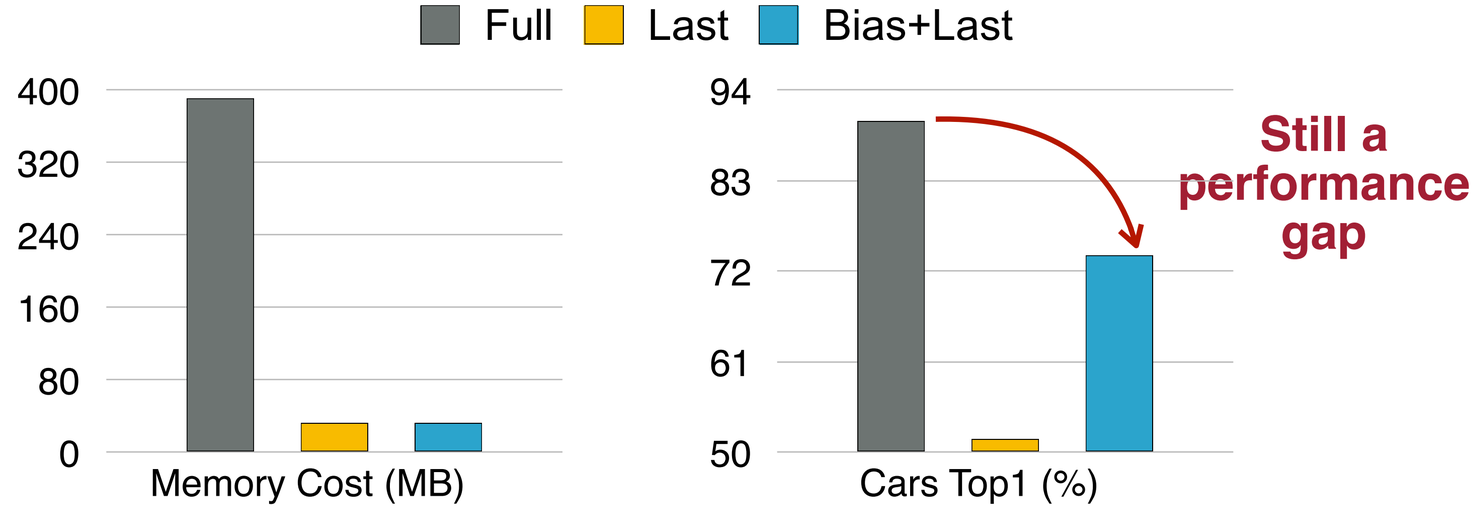
Model: ProxylessNAS-Mobile

Updating the only the bias part

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

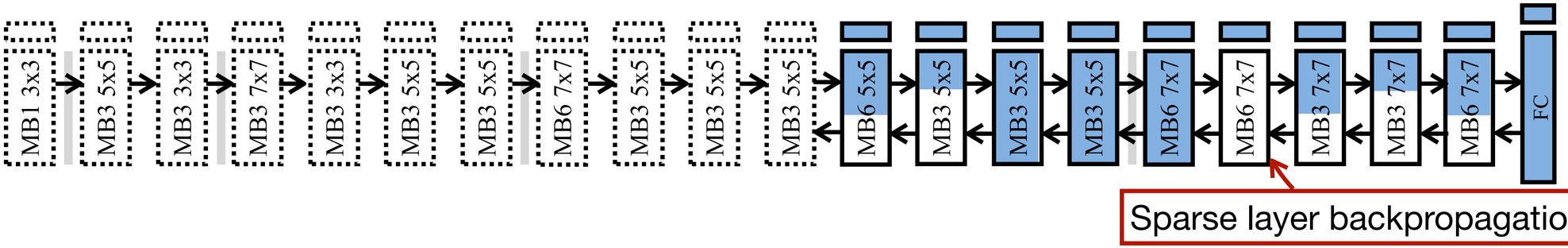
$$d\mathbf{W} = f(\mathbf{X}, d\mathbf{Y})$$

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



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

Sparse Back-Propagation

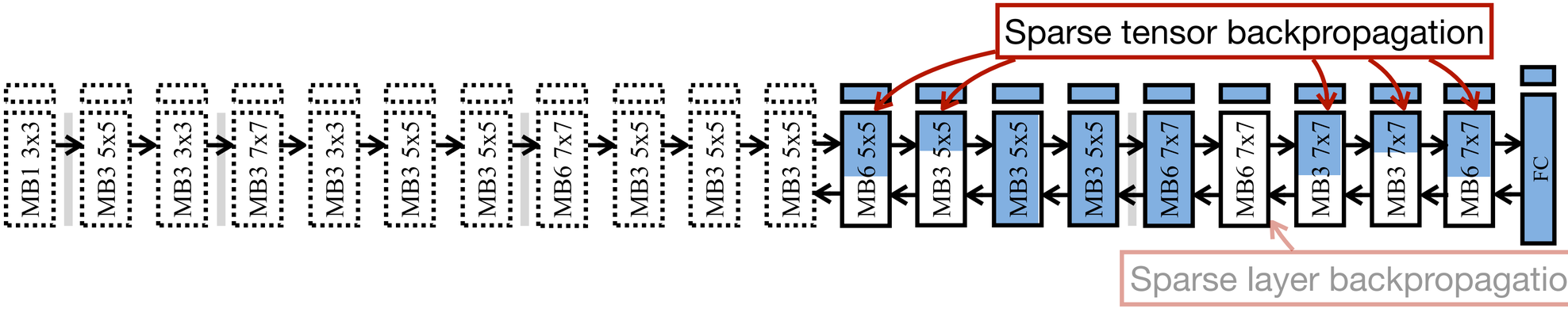


Model: ProxylessNAS-Mobile

Use sparse update to train the model

- Some **<u>layers</u> are not as important as** others

Sparse Back-Propagation

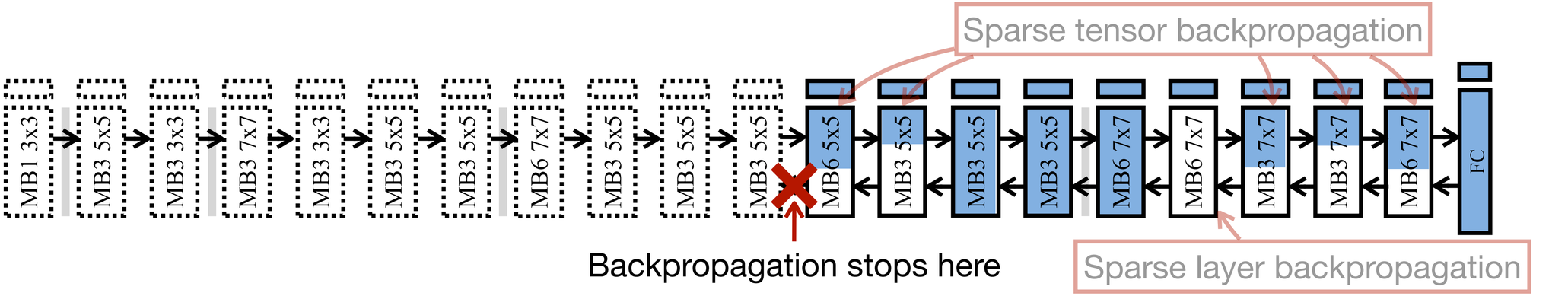


Model: ProxylessNAS-Mobile

Use sparse update to train the model

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

Sparse Back-Propagation

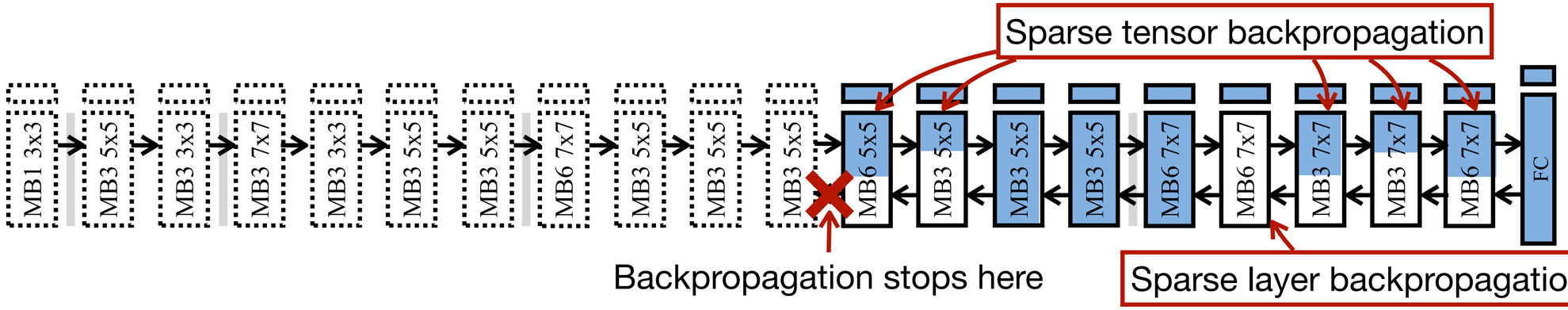


Model: ProxylessNAS-Mobile

Use sparse update 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 update 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)

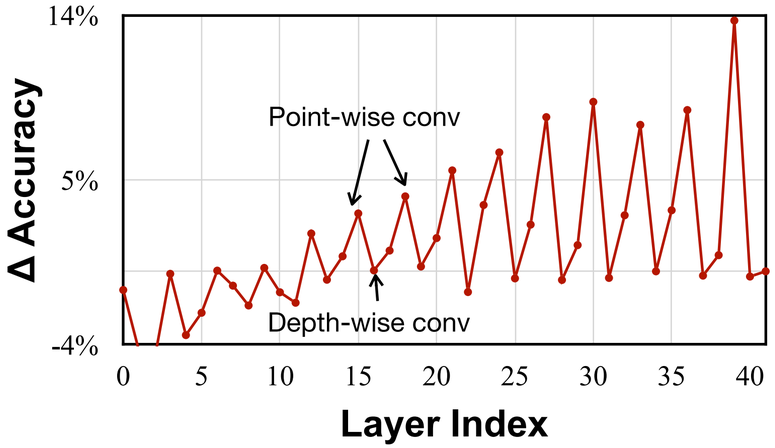
FLOPs: (0.25 * M * H * N)

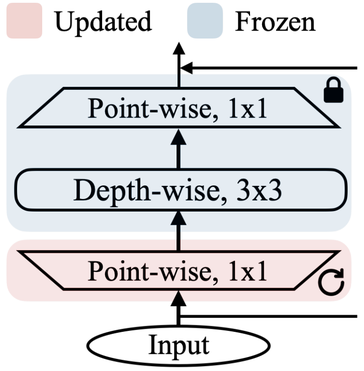
Contribution Analysis

Searching which layers to sparsely update

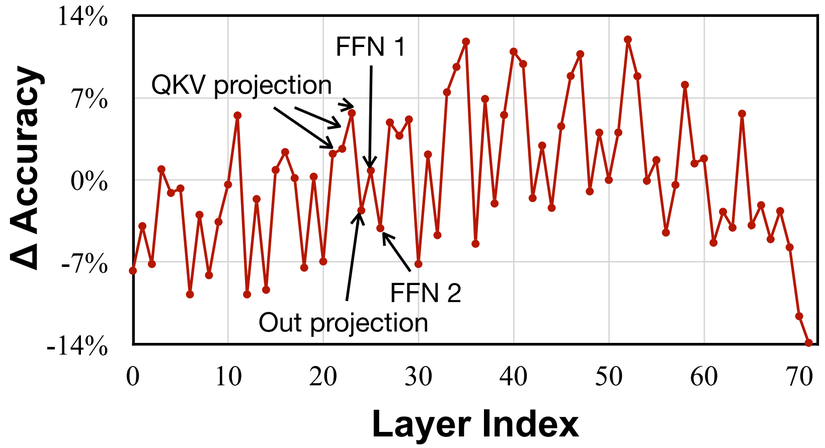
- Contribution Analysis: fine-tune only one layer on a downstream task, measure the accuracy improvement (Δaccuracy) as contributions.
- Only fine-tune the **layers with large Δaccuracy** (contributes more to performance)

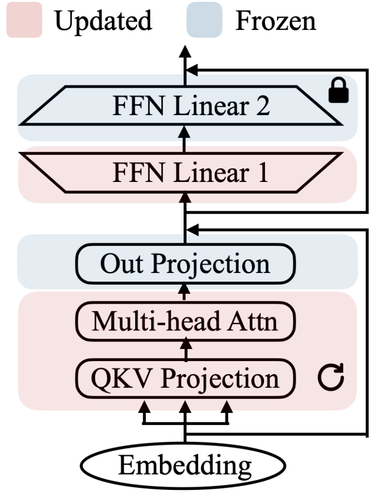
CNN model (MobileNetV2)





Transformers (BERT)



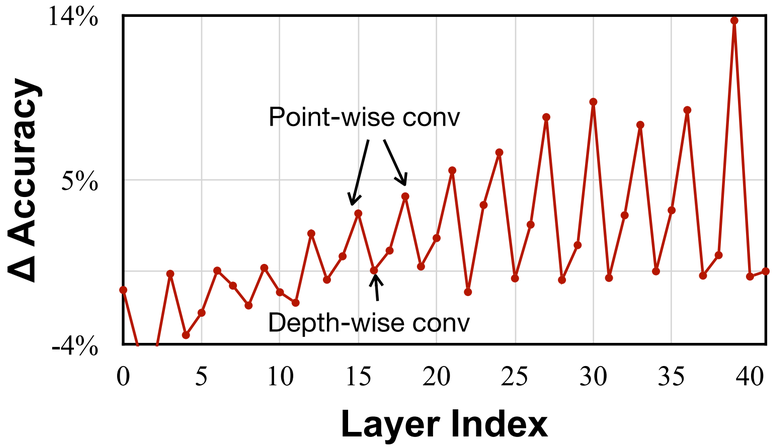


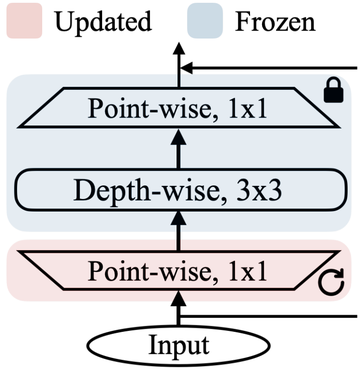
Contribution Analysis

Searching which layers to sparsely update

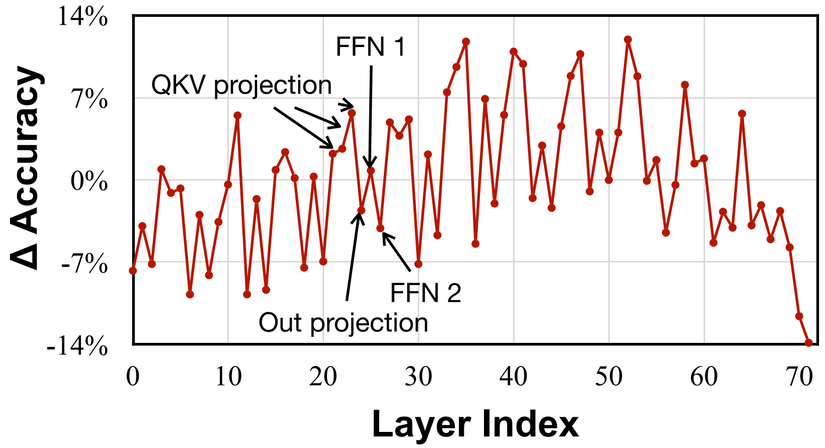
- Contribution Analysis: fine-tune only one layer on a downstream task to measure accuracy improvement (Δaccuracy) as contributions.
- Only fine-tune the **layers with large Δaccuracy** (contributes more to performance)

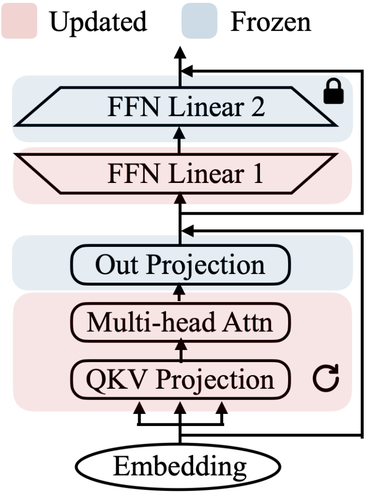
CNN model (MobileNetV2)





Transformers (BERT)





Different models prefer *different* layers for fine-tuning

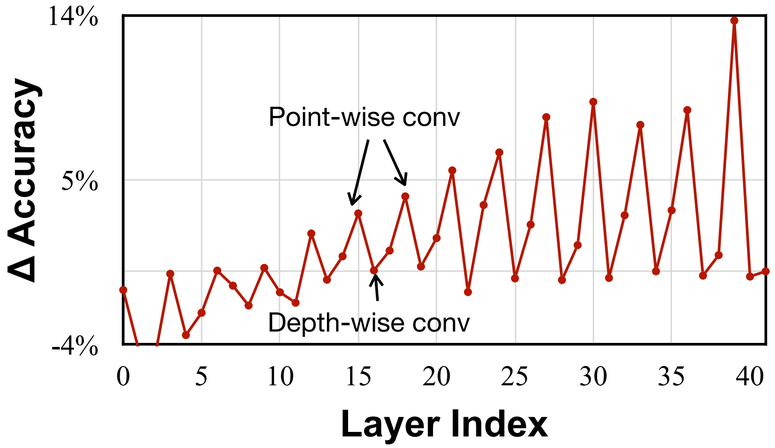
- MobilenetV2 prefers **first depth-wise conv**.
- BERT prefers **QKV projection** and **first FFN** layers.

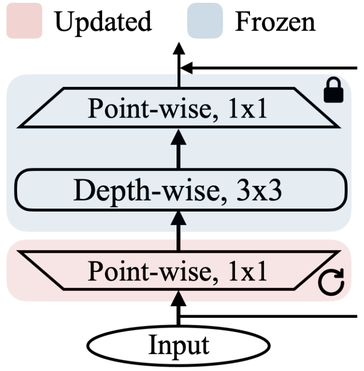
Contribution Analysis

Searching which layers to sparsely update

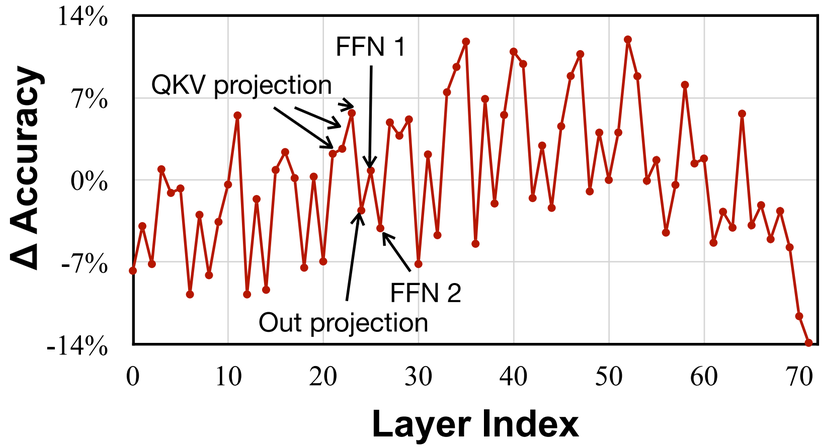
- Contribution Analysis: fine-tune only one layer on a downstream task to measure accuracy improvement (Δaccuracy) as contributions.
- Only fine-tune the **layers with large Δaccuracy** (contributes more to performance)

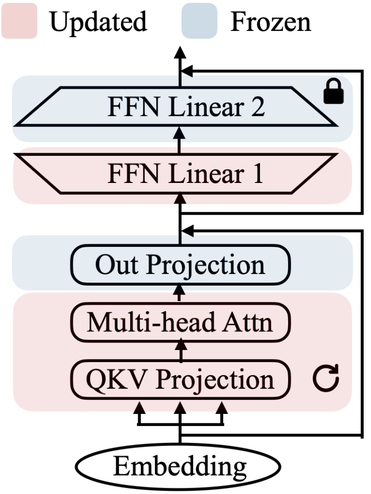
CNN model (MobileNetV2)





Transformers (BERT)





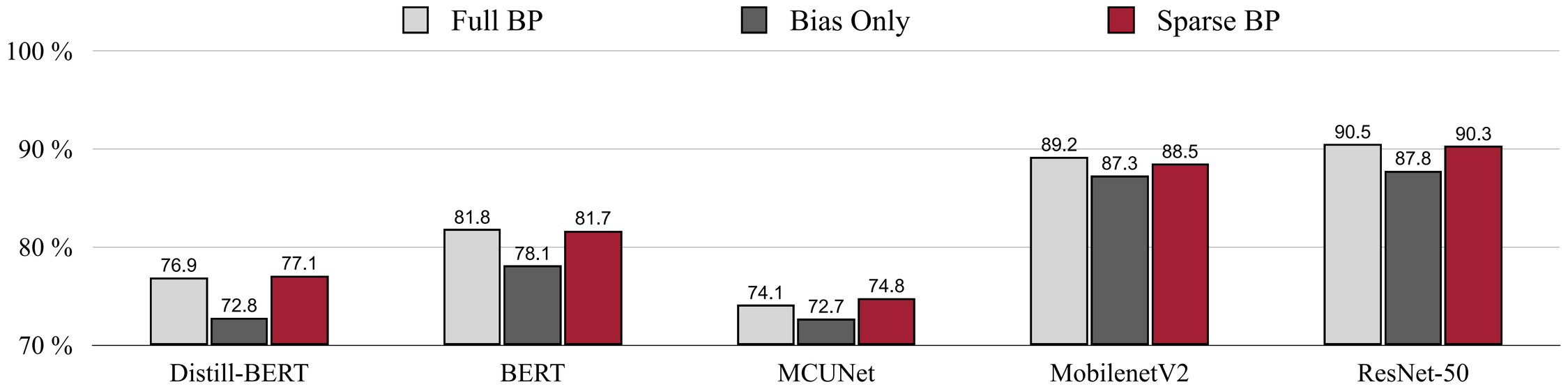
$$\mathbf{k}^*, \mathbf{i}^*, \mathbf{r}^* = \max_{\mathbf{k},\mathbf{i},\mathbf{r}} \left(\sum_{\mathbf{k} \in \mathbf{i}} \Delta \text{acc}_{\text{b}_k} + \sum_{i \in \mathbf{i}, r \in \mathbf{r}} \Delta \text{acc}_{\text{W}_{i,r}} \right) \quad \text{s.t. Memory}(\mathbf{k}, \mathbf{i}, \mathbf{r})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

k is the bias index, i is the layer index and r is the sparsity ratio.

- Use **evolutionary search** to find the sparse update scheme.

Accuracy of Sparse Back-Propagation

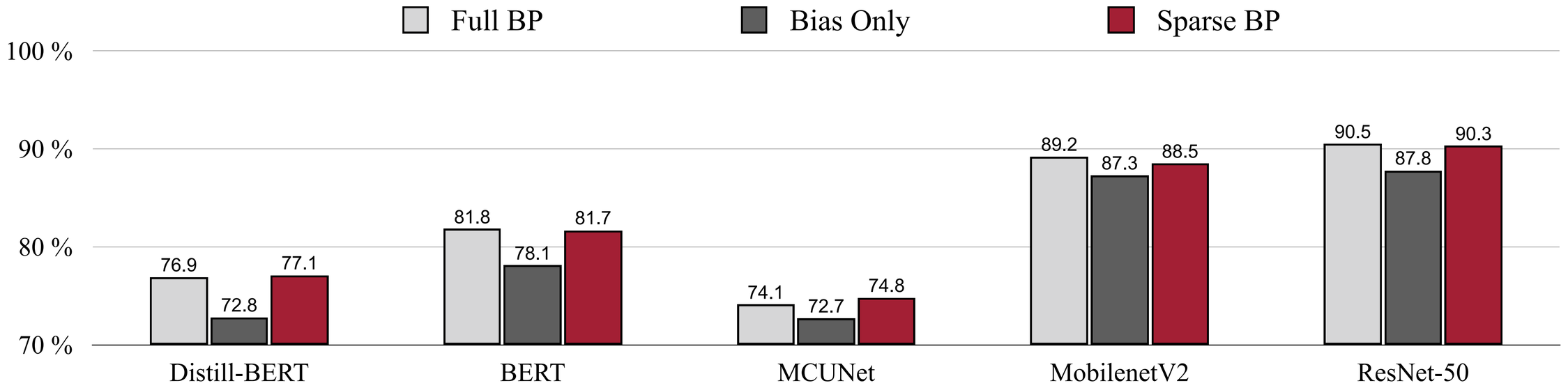
Well maintains the accuracy



- 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.

Accuracy of Sparse Back-Propagation

Can’t translate to measured speedup on existing framework



- Existing frameworks:
  - Assume all weights will be updated — can not handle for frozen weights (masking only).
  - Each OP have its own BP kernels — supporting sparse-BP for all OPs is engineering-expensive.

Sparse-BP needs **compiler support** to *translate theoretical* saving into *measured speedup*.

PockEngine

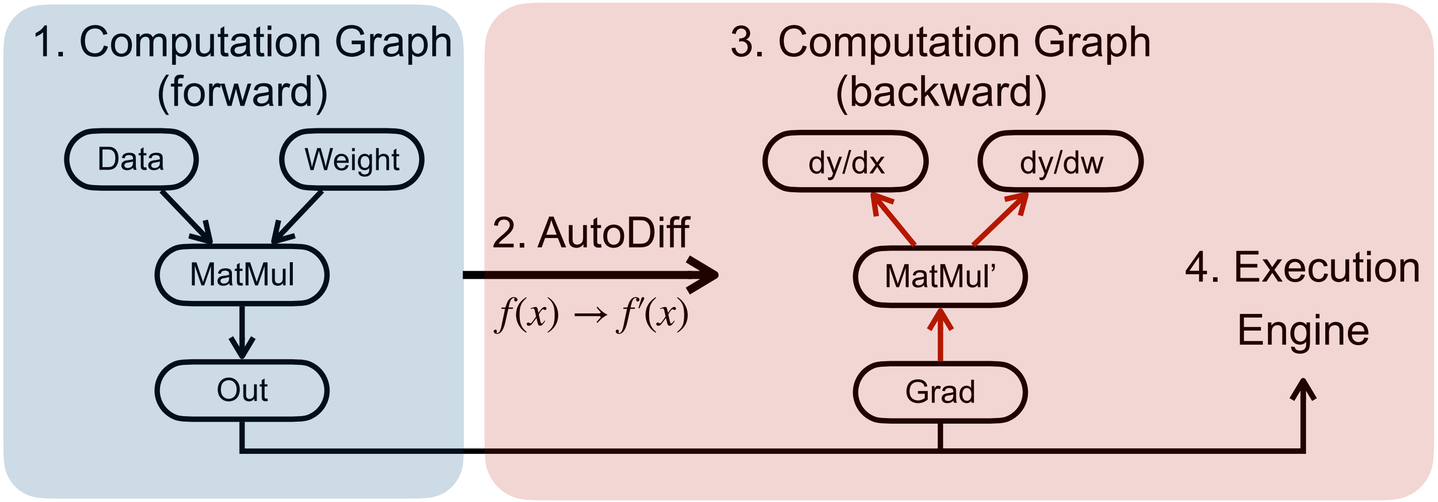
1. Sparse Back-Propagation

2. Compiler Support

3. Fine-tune Llama2 on Orin

PockEngine: Compiler Support

Conventional training frameworks



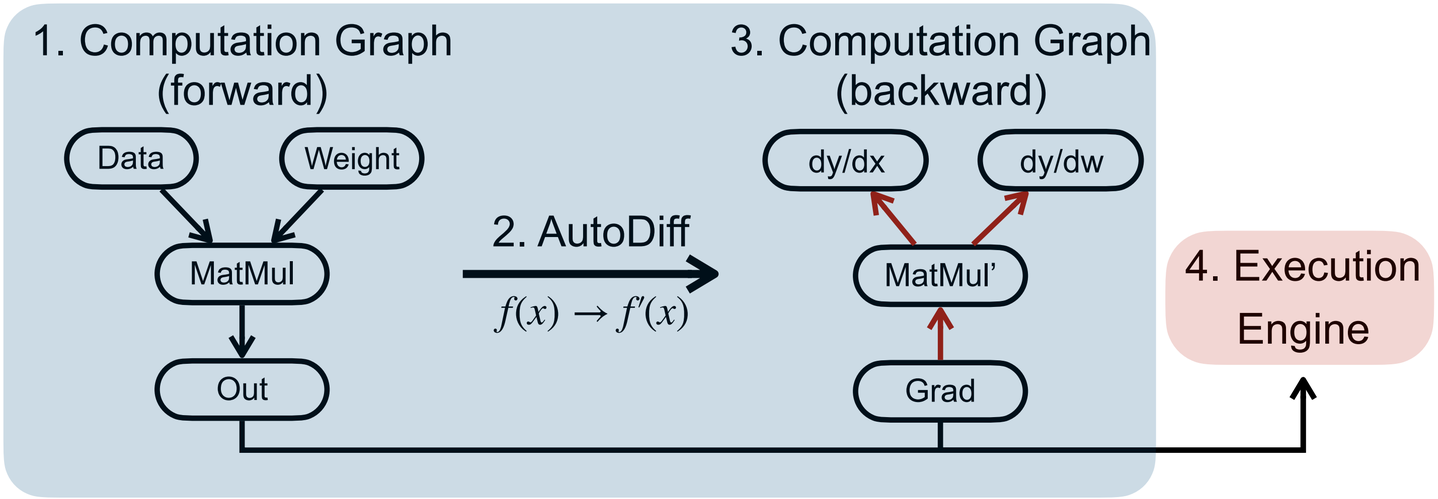
: Compile-Time

: Runtime

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

PockEngine: Compiler Support

New workflow: more@compile time, less@runtime



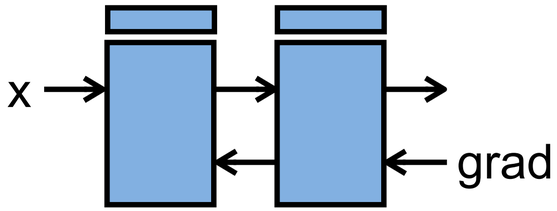
: Compile-Time

: Runtime

- PockEngine shifts the workload from runtime to **compile-time** (autodiff, memory scheduling, execution planning), minimizes the **runtime overhead**
- Enables compile-time **training graph optimizations**.

PockEngine: Compiler Support

Step 1 - Translate models into a unified IR



(a) Full BP

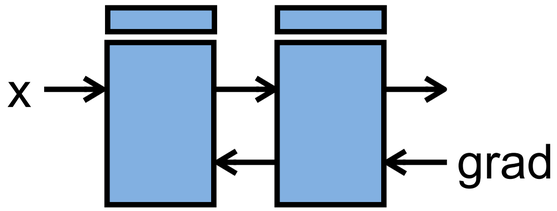
```
net = nn.Sequential(
    nn.Linear(10, 10),
    nn.Linear(10, 10),
)
x = torch.randn(1, 3, 28, 28)
out = net(x).sum().backward()
```

(b) PyTorch Model

PockEngine takes models defined in PyTorch/TensorFlow/Jax.

PockEngine: Compiler Support

Step 1 - Translate models into a unified IR



(a) Full BP

```
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

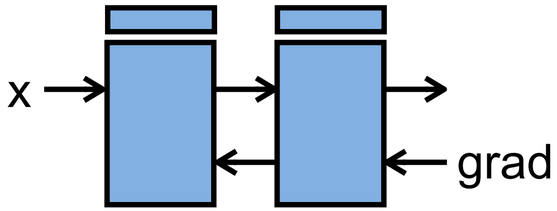
```
# forward layer #1
%0 = multiply(%x, %w1);
%1 = add(%0, %b1);
# forward layer #2
%2 = multiply(%1, %w2);
%3 = add(%2, %b2);
```

(c) Forward IR

PockEngine first generates forward IR

PockEngine: Compiler Support

Step 2 - AutoDiff the graph at compile-time



(a) Full BP

```
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

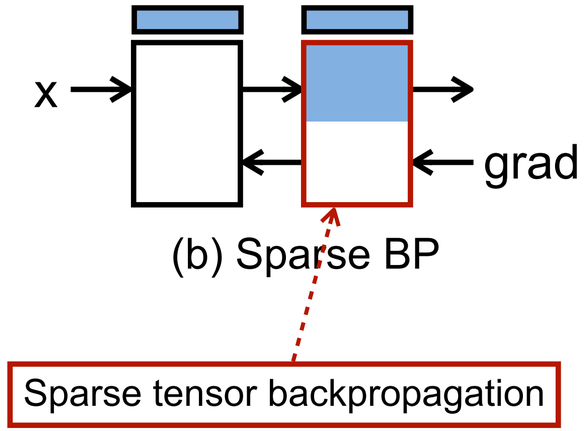
```
# forward layer #1
%0 = multiply(%x, %w1);
%1 = add(%0, %b1);
# forward layer #2
%2 = multiply(%1, %w2);
%3 = add(%2, %b2);
# backward layer #2
%4 = multiply(%grad, %w2);
%5 = transpose(%grad);
%6 = multiply(%5, %1);
%7 = sum(%grad, axis=-1);
# backward layer #1
%8 = multiply(%6, %w1);
%9 = transpose(%6);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

(d) Forward + Backward IR

PockEngine constructs the training graph by **re-using the same set of primitive OPs** as inference, easily extend existing **inference-only frameworks** with training capabilities

PockEngine: Compiler Support

Step 3 - Computation graph pruning for sparse tensor BP



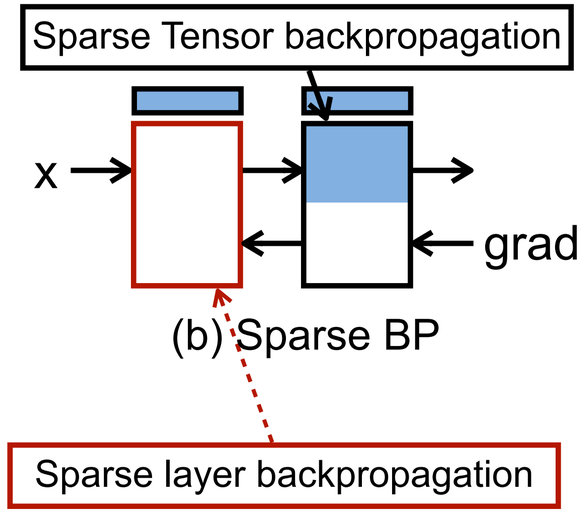

```
# forward layer #1
%0 = multiply(%x, %w1);
%1 = add(%0, %b1);
# forward layer #2
%2 = multiply(%1, %w2);
%3 = add(%2, %b2);
# backward layer #2
%4 = multiply(%grad, %w2);
%5 = transpose(%grad);
%6 = multiply(%5, %1);
%7 = sum(%grad, axis=-1);
# backward layer #1
%8 = multiply(%6, %w1);
%9 = transpose(%6);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

PockEngine traces the graph and transforms the computation nodes.

PockEngine: Compiler Support

Step 3 - Computation graph pruning for sparse layer BP



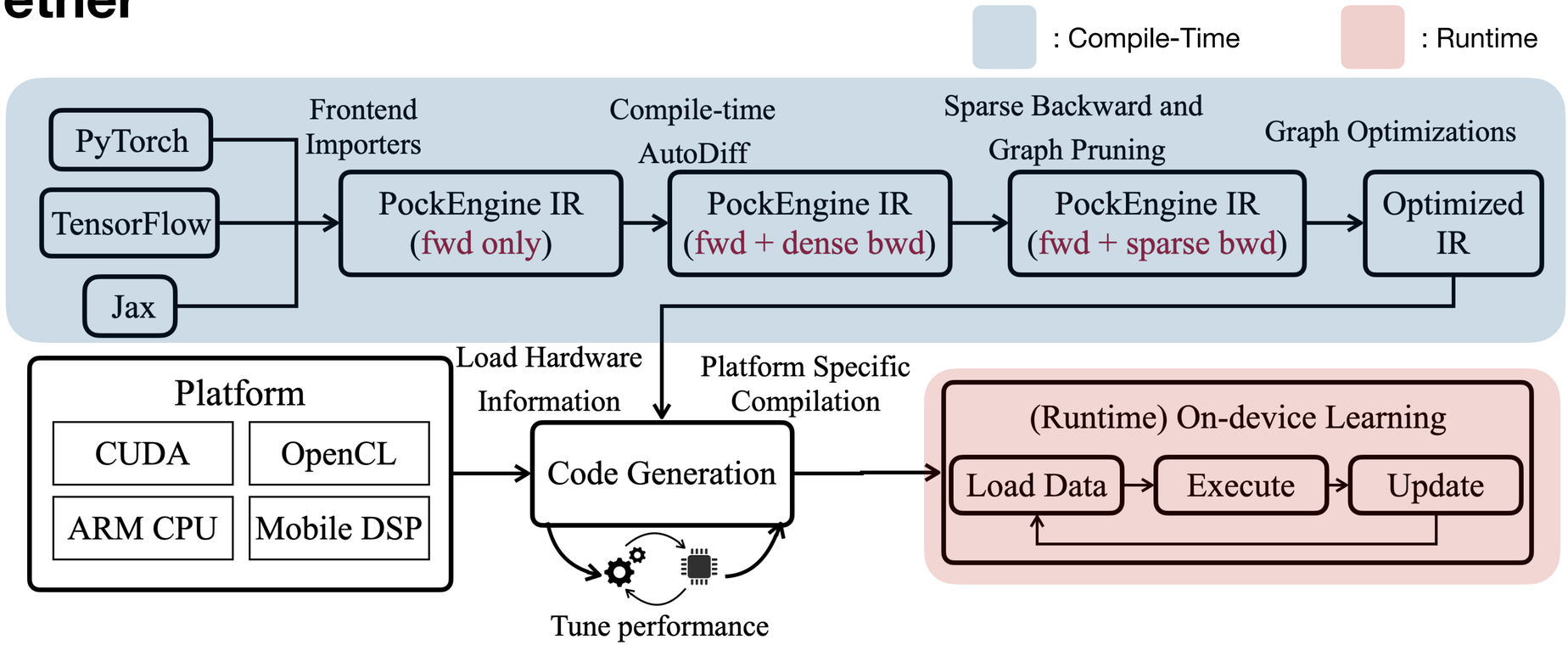

```
# forward layer #1
%0 = multiply(%x, %w1);
%1 = add(%0, %b1);
# forward layer #2
%2 = multiply(%1, %w2);
%3 = add(%2, %b2);
# backward layer #2
%4 = multiply(%grad, %w2);
%5 = transpose(%grad);
%6 = multiply(%5, %1);
%7 = sum(%grad, axis=-1);
# backward layer #1
%8 = multiply(%6, %w1);
%9 = transpose(%6);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

PockEngine **prunes the computation graph** by dependency analysis and dead code elimination

PockEngine: Compiler Support

Put it together

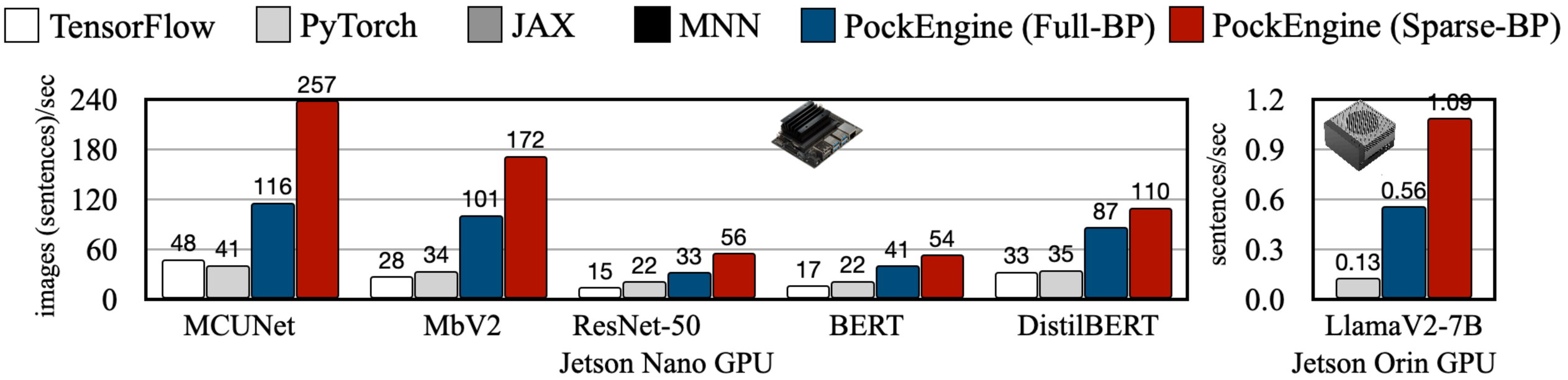


PockEngine features:

- Offload AutoDiff from runtime to compile-time.
- Prunes the computation graph, transforms the computation nodes.
- Support diverse frontends/backends, and diverse NN architectures (CNN, LLM).

PockEngine: Compiler Support

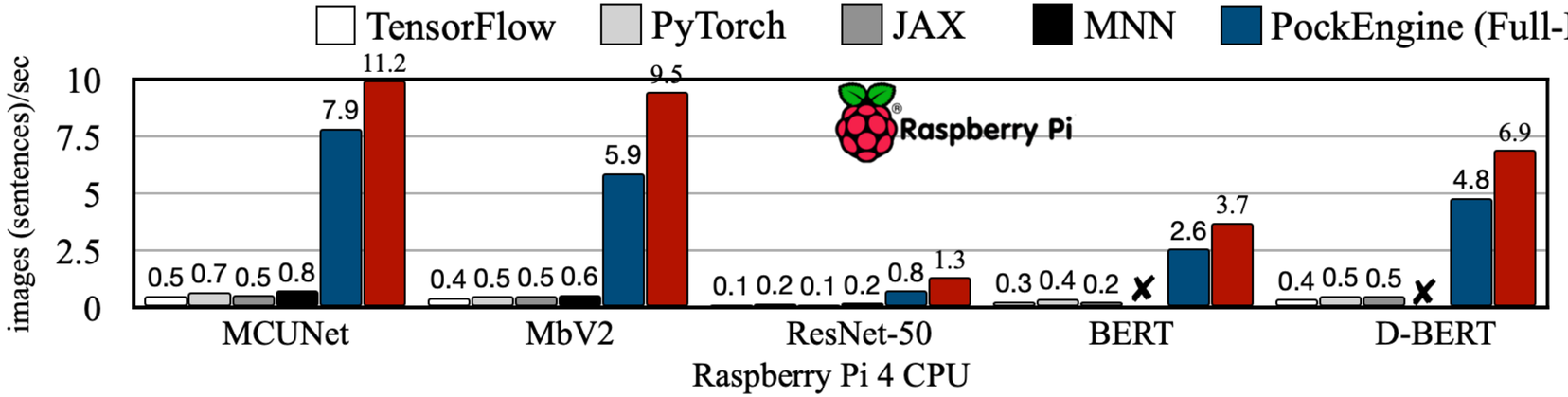
Speedup comparison: Edge GPU

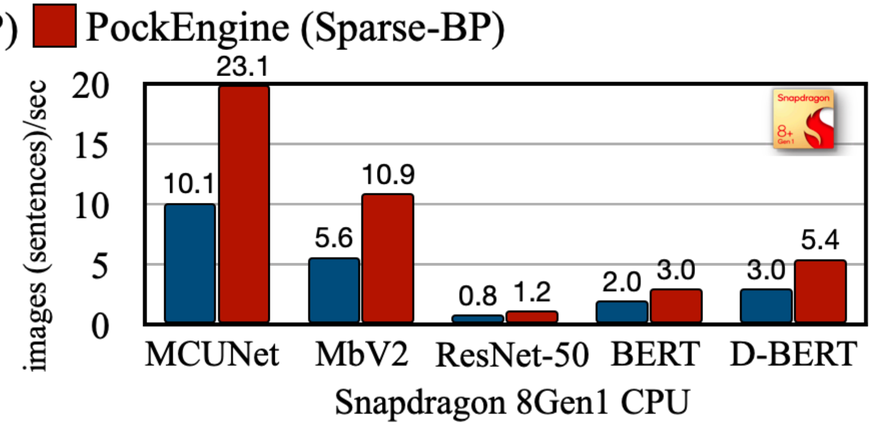


- **2-4x speedup** on Jetson Nano and Orin.
- Speedup comes from compilation. The host language (Python) is slow on low-frequency CPUs.
- Commercial inference frameworks (e.g., TensorRT) also benefits from compilation, but not for training.

PockEngine: Compiler Support

Speedup comparison: ARM CPU

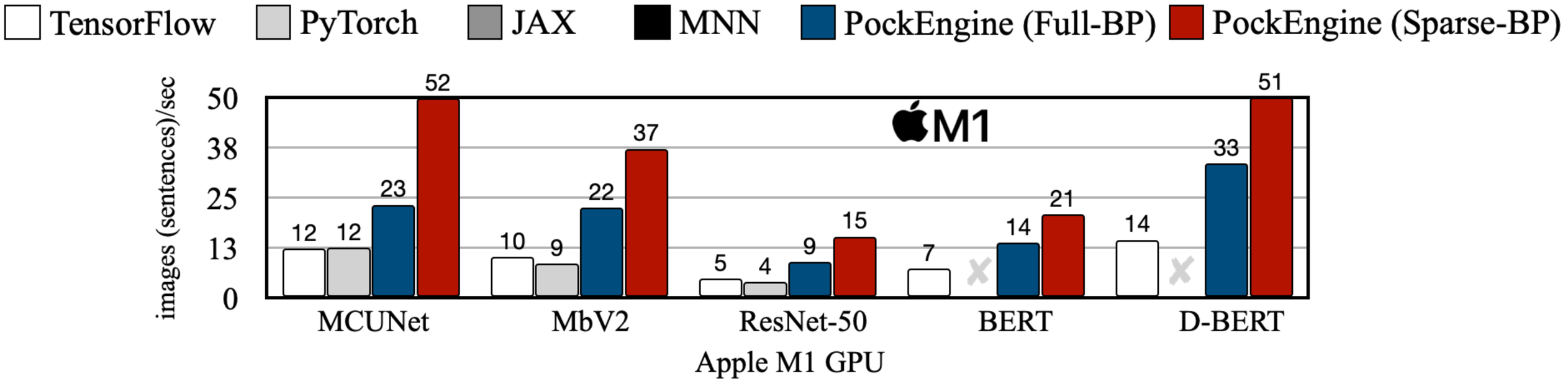




- PockEngine shows **13 to 21x speedup** on Raspberry Pi 4 B+ platforms.
  - Existing framework are mostly optimized for inference only.
  - Compilation enables kernel tuning, thus accelerates training.
    - Most kernel implementations focus on GPU and x86 CPU.
    - ARM CPU (especially for training)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

PockEngine: Compiler Support

Speedup comparison: Apple M-Chip

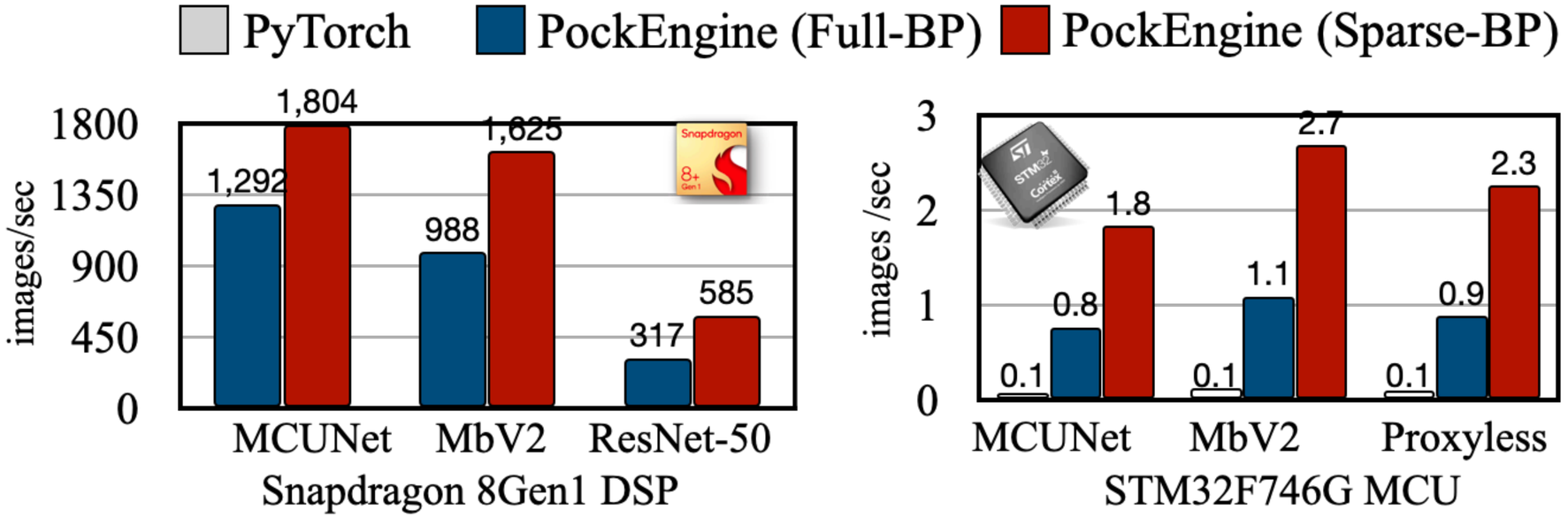


- Apple M1/2 is a new, and the **compatibility is not ideal** for PyTorch and TensorFlow:
  - For PyTorch*, even with a recent build (commit ID: c9913cf), transformer training throws errors on M1.
  - For TensorFlow, the GPU training support is preliminary and incomplete on M1.
- PockEngine compiles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

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

PockEngine: Compiler Support

Speedup comparison: DSP and MCUs

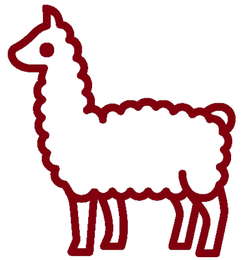


- We integrate SNPE for Qualcomm DSPs and TinyEngine for Microcontrollers
- PockEngine's compilation workflow is (1) kernel agnostic and (2) shares the same set of OPs between FWD and BWD, thus enables **previous inference-only framework to support training**.

PockEngine

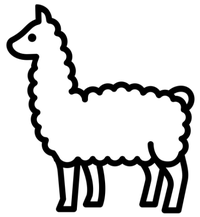
1. Sparse Back-Propagation

2. Compiler Support



3. Fine-tune Llama2 on Orin

PockEngine: 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

PockEngine: 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

- LoRA is parameter efficient (45.1GB -> 30.9GB), but the training time is not improved much.
- LoRA needs to **back-propagate to the very first layer** while SparseBP doesn't.
- Fine-tuning on 1000 instruction-tuning samples takes 2 hours using PyTorch
- while PockEngine needs < half an hour (**4.4x faster**).

PockEngine: 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.

PockEngine: 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.

PockEngine: 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**.

Without fine-tuning, the model failed to understand and only simply repeats.

PockEngine: 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.

PockEngine Conclusion

- AI models outgrow hardware. There's plenty of room at the top. Co-design is essential.
- **Sparse back-propagation** is efficient for fine-tuning (both CNN and LLM).
- PockEngine provides **compiler support** for both **dense** and **sparse** backpropagation.
- Enables **inference-only** frameworks to conduct **training**.
- Consistent **speedup** on diverse hardware platforms.
- Highlight: **fine-tune Llama2-7B locally on edge device (Jetson Orin).**

Sister projects:

LLM Quantization: SmoothQuant, AWQ (integrated by NVIDIA, Intel)

On-device LLM: TinyChat (deploy Llama2-7B on Rasp Pi and Mac)

Long Conversation: StreamingLLM (infinitely long chat)