EECE695D: Efficient ML Systems

Network Architectures

(part 2)

Recap

Architectures. Focused on achieving SOTA parameter-accuracy tradeoff (hoping for latency improvements), mainly for the inference efficiency.

Methodology. Slowly transitioned from handcrafting key modules (e.g., inverted residual) to automated search (neural architecture search)

Today. Asking a different question—training efficiency!
+ Models that run on memory-scarce devices

EfficientNet V2

Tan & Le (ICML 2021)

Tan & Le, "EfficientNetV2: Smaller Models and Faster Training," ICML 2021.

Twists

- EfficientNet is good, but there are at least three limitations.

  (1) Slow Training—large image size. Using full-sized images ⇒ smaller batch size

  Solution. Use smaller image patches during training, just like FixRes (Touvron et al., 2019).
  (In FixRes, we also additionally fine-tune some layers; in EffNet, not!)
  This somehow leads to a better accuracy, too! (Why?)

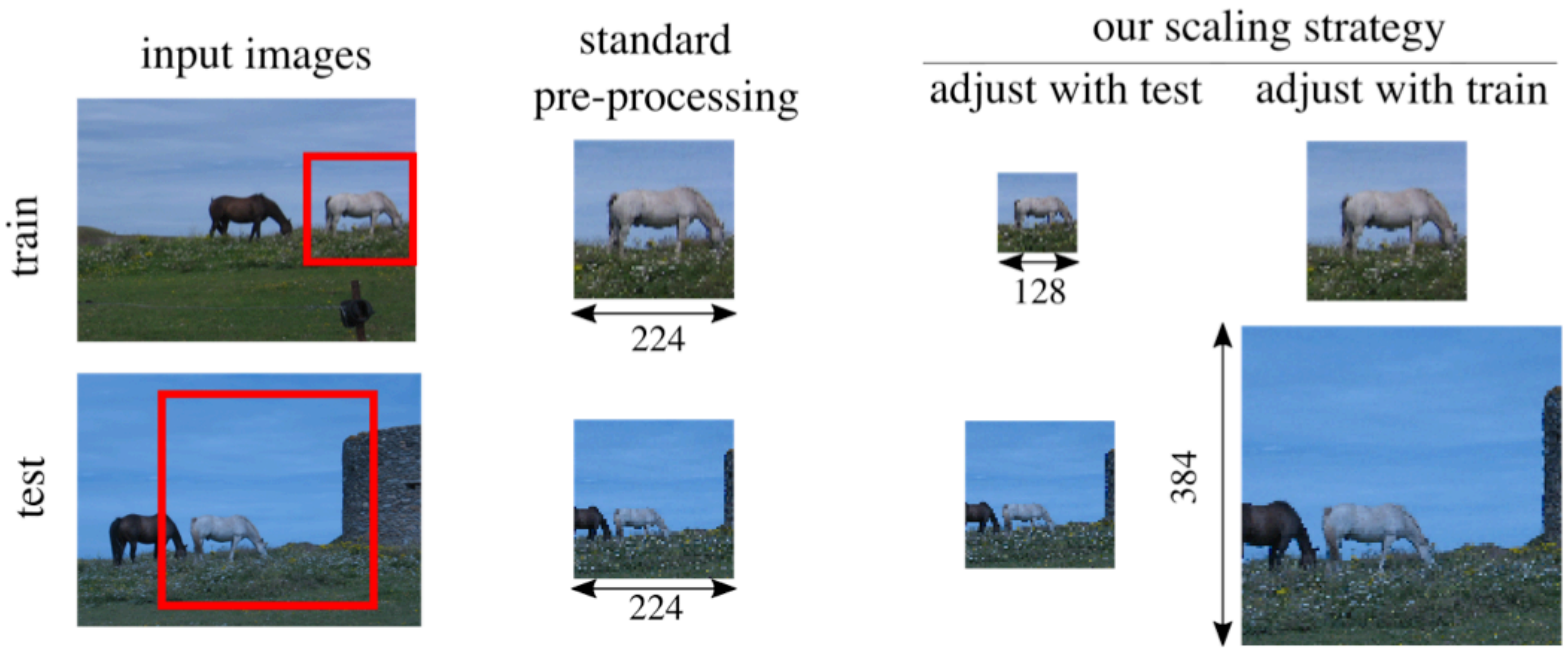


Table 2. EfficientNet-B6 accuracy and training throughput for different batch sizes and image size.

	Top-1 Acc.	TPUv3 imgs/sec/core batch=32	TPUv3 imgs/sec/core batch=128	V100 imgs/sec/gpu batch=12	V100 imgs/sec/gpu batch=24
train size=512	84.3%	42	OOM	29	OOM
train size=380	84.6%	76	93	37	52

Touvron et al., "Fixing the train-test resolution discrepancy," arXiv 2019 (updated 2022)

Twists

- EfficientNet is good, but there are at least three limitations.

(1) Slow Training—large image size. Using full-sized images ⇒ smaller batch size

Solution. Use smaller image patches during training, just like FixRes (Touvron et al., 2019).
(In FixRes, we also additionally fine-tune some layers; in EffNet, not!)
This somehow leads to a better accuracy, too! (Why?)
Also develops a well-tailored progressive training technique,
combined with adaptive regularization.

Twists

- EfficientNet is good, but there are at least three limitations.

(2) DW Convs are slow in early layers. Early layers have small # input channels—using DWConv there means underutilization of GPU/TPU.

Solution. Use Fused-MBConv in early layers.
Use NAS to figure out how early.

Table 3. Replacing MBConv with Fused-MBConv. `No fused` denotes all stages use MBConv, `Fused stage1-3` denotes replacing MBConv with Fused-MBConv in stage {2, 3, 4}.

	Params (M)	FLOPs (B)	Top-1 Acc.	TPU imgs/sec/core	V100 imgs/sec/gpu
No fused	19.3	4.5	82.8%	262	155
Fused stage1-3	20.0	7.5	83.1%	362	216
Fused stage1-5	43.4	21.3	83.1%	327	223
Fused stage1-7	132.0	34.4	81.7%	254	206

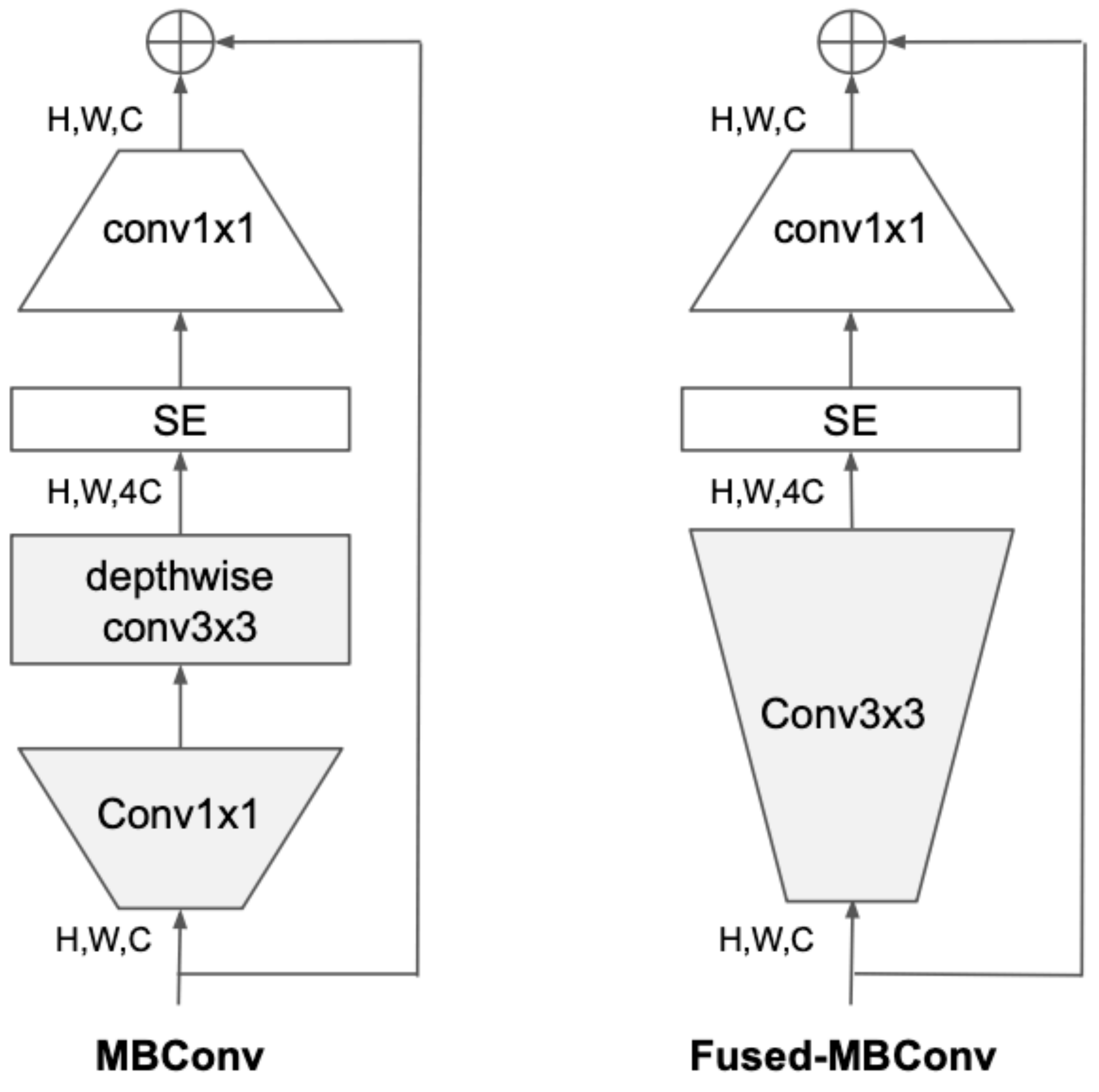


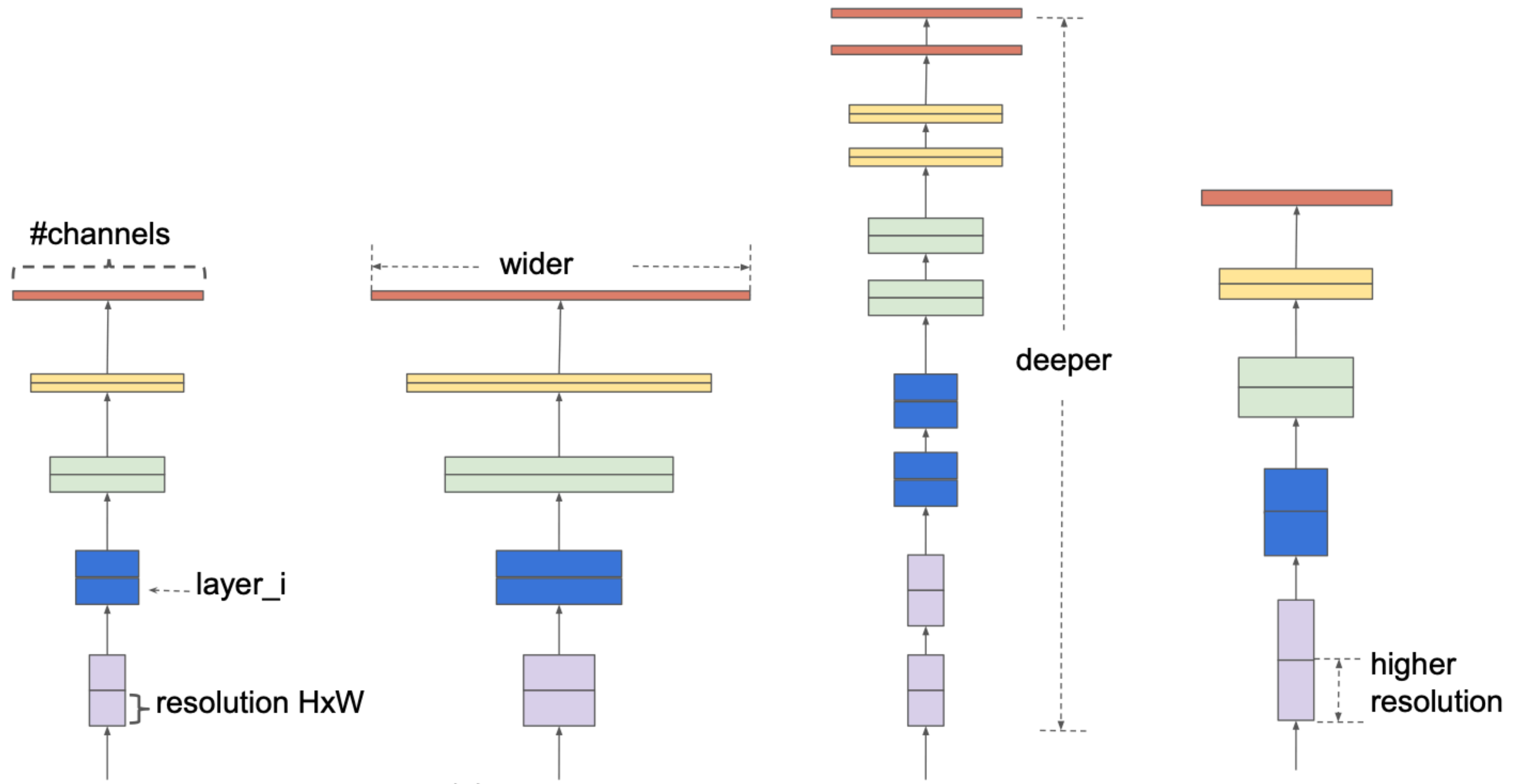
Figure 2. Structure of MBConv and Fused-MBConv.

Twists

- EfficientNet is good, but there are at least three limitations.

(3) Equi-layer scale-up is bad. Empirically tryouts show that doubling depth of all layers is highly suboptimal, in terms of efficiency.

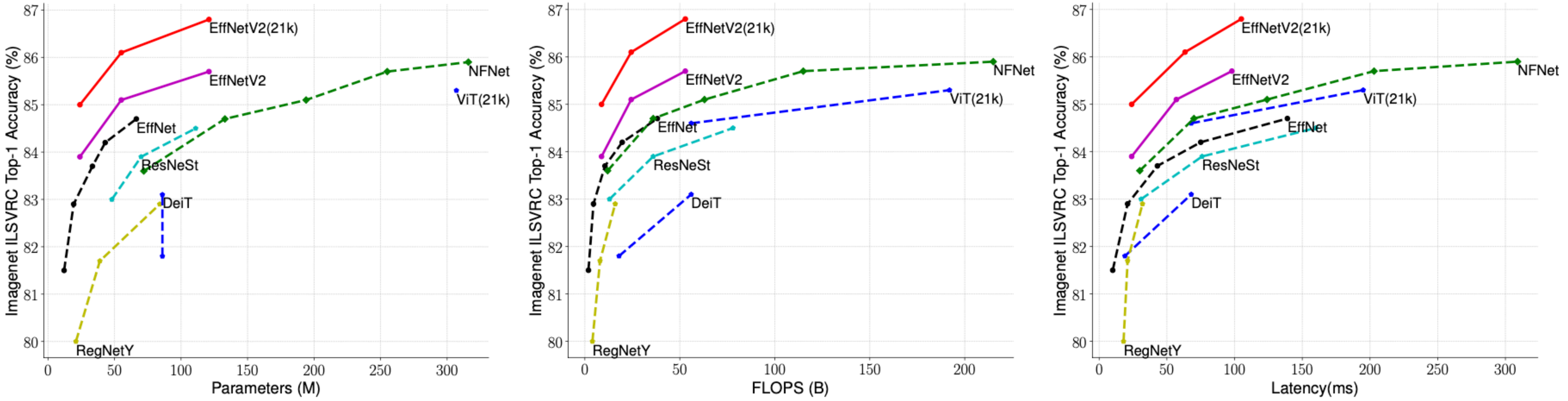
Solution. Modify the scaling law: Depth scaling—more on later layers
Resolution scaling—less preferred.



(a) baseline (b) width scaling (c) depth scaling (d) resolution scaling

Training-aware NAS

- Authors call a NAS that takes into account these twists a **training-aware NAS**
- This training-aware NAS finds: (1) Fused-MBConv in early layers.
  (2) Smaller expansion ratio of MBConv (less memory access overhead)
  (3) Prefers 3x3 conv than 5x5, with added number of layers



(a) Parameters (b) FLOPs (c) GPU V100 Latency (batch 16)

Figure 5. **Model Size, FLOPs, and Inference Latency** – Latency is measured with batch size 16 on V100 GPU. `21k` denotes pretrained on ImageNet21k images, others are just trained on ImageNet ILSVRC2012. Our EfficientNetV2 has slightly better parameter efficiency with EfficientNet, but runs 3x faster for inference.

Model	Top-1 Acc.	Params	FLOPs	Infer-time(ms)	Train-time (hours)
EfficientNet-B3 (Tan & Le, 2019a)	81.5%	12M	1.9B	19	10
EfficientNet-B4 (Tan & Le, 2019a)	82.9%	19M	4.2B	30	21
EfficientNet-B5 (Tan & Le, 2019a)	83.7%	30M	10B	60	43
EfficientNet-B6 (Tan & Le, 2019a)	84.3%	43M	19B	97	75
EfficientNet-B7 (Tan & Le, 2019a)	84.7%	66M	38B	170	139
EfficientNetV2-S	83.9%	22M	8.8B	24	7.1
EfficientNetV2-M	85.1%	54M	24B	57	13
EfficientNetV2-L	85.7%	120M	53B	98	24

NFNet

Brock et al. (ICML 2021)

Brock et al., "High-performance large-scale image recognition without normalization," ICML 2021.

Motivation

- **Batch normalization.** accelerates training (reduces # epochs),
  stabilizes the training of deeper networks,
  smoothens loss landscape to enable larger learning rate & batch size,
  and has some regularization effects…

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

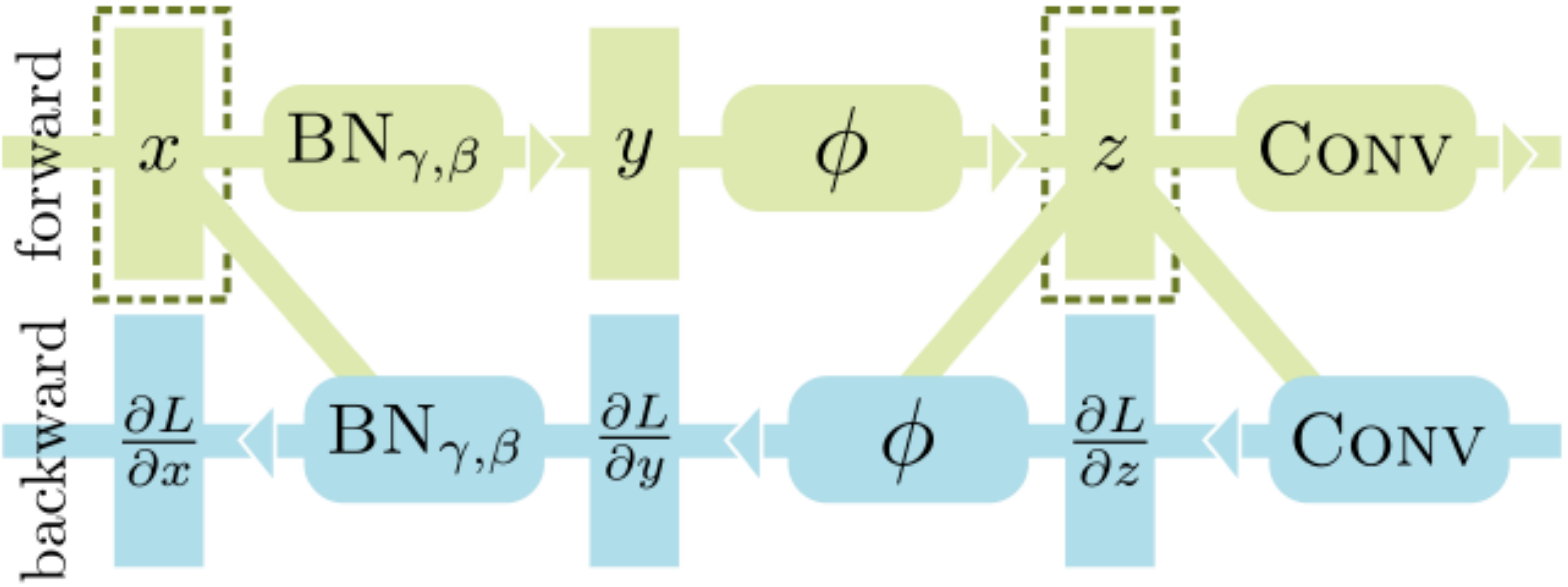
Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

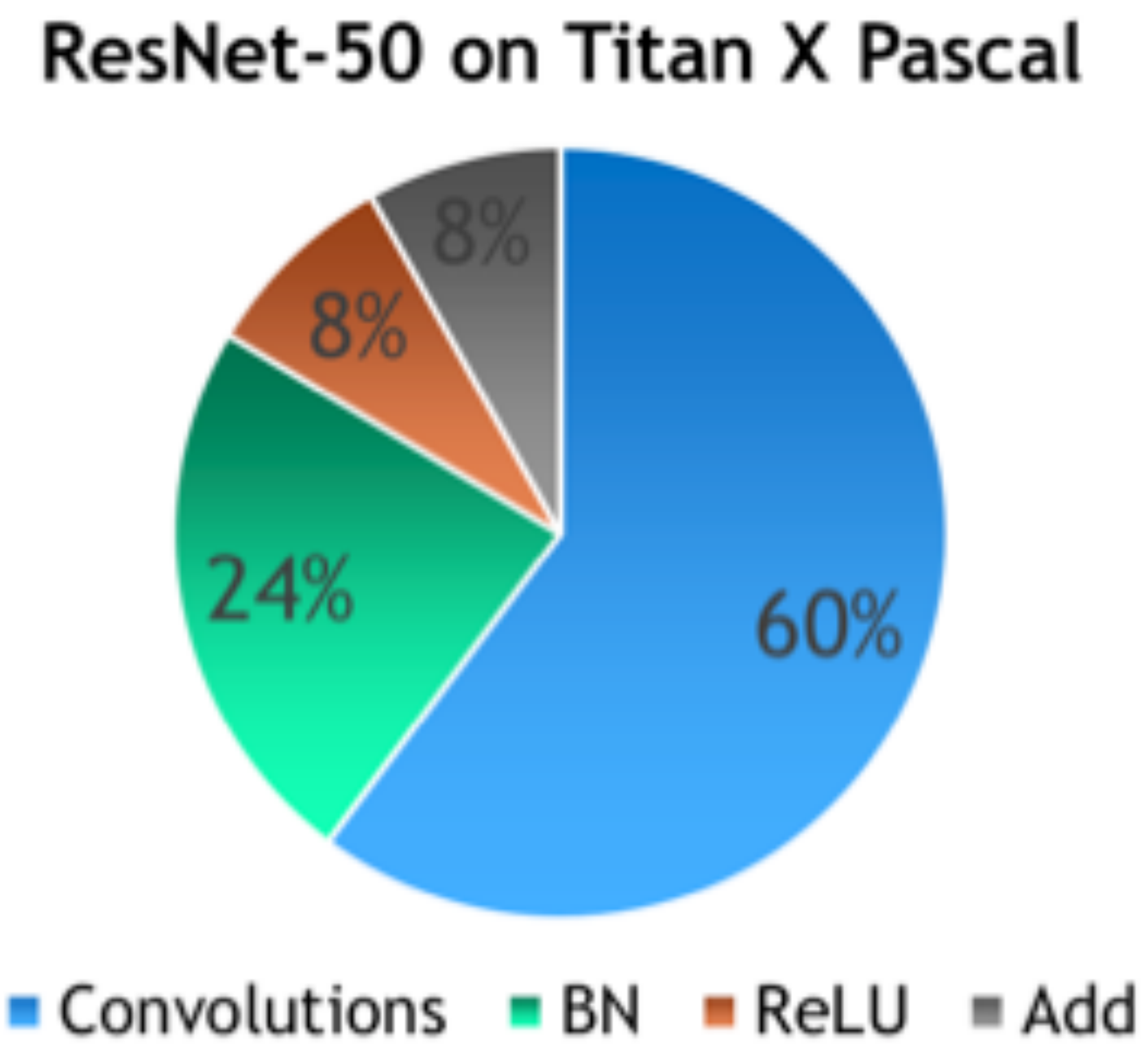
Ioffe & Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," arXiv 2015

Motivation

- Batch normalization has some practical disadvantages

  (1) Expensive. Incurs memory overhead
  Increases the time required to evaluate gradient in some NNs.





Need to additionally store x for computing $\frac{\partial L}{\partial x}, \frac{\partial L}{\partial \gamma}$

Rota Bulò et al., "In-Place Activated BatchNorm for Memory-Optimized Training of DNNs," CVPR 2018
Gitman and Ginsburg, "Comparison of BN and WN algorithms for the Large-scale Image Classification," arXiv 2017

Motivation

- Batch normalization has some practical disadvantages

  (2) Shift. Introduces a discrepancy between the training time & inference time behavior.
  (let's not care too much about this now)

  (3) Dependency. Breaks the independence among training examples in the mini-batch.
  - BN causes subtle errors in distributed training (Pham et al., 2019)
  - Source of information leakage in contrastive learning (Chen et al., 2020)
  - Does not work well in small-batch training (Hoffer et al., 2017)
  + Some issues with privacy / adversarial robustness

Pham et al., "Cradle: Cross-backend validation to detect and localize bugs in DL libraries," ICSE 2019

Chen et al., "A simple framework for contrastive learning of visual representations," ICML 2020

Hoffer et al., "Train longer, generalize better: Closing the generalization gap in large batch training of NNs," NeurIPS 2017

Idea

- Remove the BatchNorm from the model, by replicating its positive effect with
    - **Scaled weight standardization.** Reparameterizing the weights of conv layer as

$$\hat{w} = \gamma \cdot \frac{w - \mu_w}{\sigma_w \sqrt{N}}$$

    - **Adaptive gradient clipping.** Clipping gradient based on the ratio $||\nabla_\theta||/||\theta||$
- **Intuition.** Weight standardization normalizes the layer output,
  reduce remaining exploding gradient with AGC.

Brock et al., "Characterizing signal propagation to close the performance gap in unnormalized ResNets," ICLR 2021

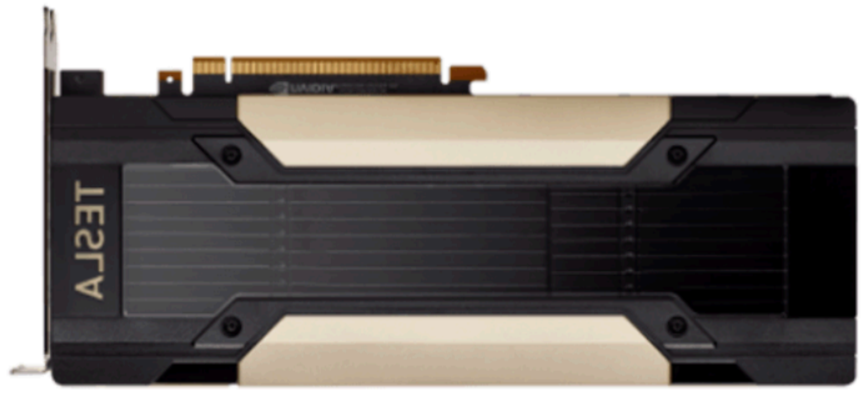
Model	#FLOPs	#Params	Top-1	Top-5	TPUv3 Train	GPU Train
ResNet-50	4.10B	26.0M	78.6	94.3	41.6ms	35.3ms
EffNet-B0	0.39B	5.3M	77.1	93.3	51.1ms	44.8ms
SENet-50	4.09B	28.0M	79.4	94.6	64.3ms	59.4ms
NFNet-F0	**12.38B**	**71.5M**	**83.6**	**96.8**	**73.3ms**	**56.7ms**
SENet-350	52.90B	115.2M	83.8	96.6	593.6ms	–
EffNet-B5	9.90B	30.0M	83.7	96.7	450.5ms	458.9ms
LambdaNet-350	–	105.8M	84.5	97.0	471.4ms	–
BoTNet-77-T6	23.30B	53.9M	84.0	96.7	578.1ms	–
NFNet-F3	**114.76B**	**254.9M**	**85.7**	**97.5**	**532.2ms**	**524.5ms**
LambdaNet-420	–	124.8M	84.8	97.0	593.9ms	–
EffNet-B6	19.00B	43.0M	84.0	96.8	775.7ms	868.2ms
BoTNet-128-T7	45.80B	75.1M	84.7	97.0	804.5ms	–
NFNet-F4	**215.24B**	**316.1M**	**85.9**	**97.6**	**1033.3ms**	**1190.6ms**

MCUNet
Lin et al. (NeurIPS 2020)

Lin et al., "MCUNet: Tiny Deep Learning on IoT Devices," NeurIPS 2020.

Motivation

- **Microcontroller.** If we want to make a model that runs on MCU, the real bottleneck is the **storage / peak memory**.



Cloud AI

Mobile AI

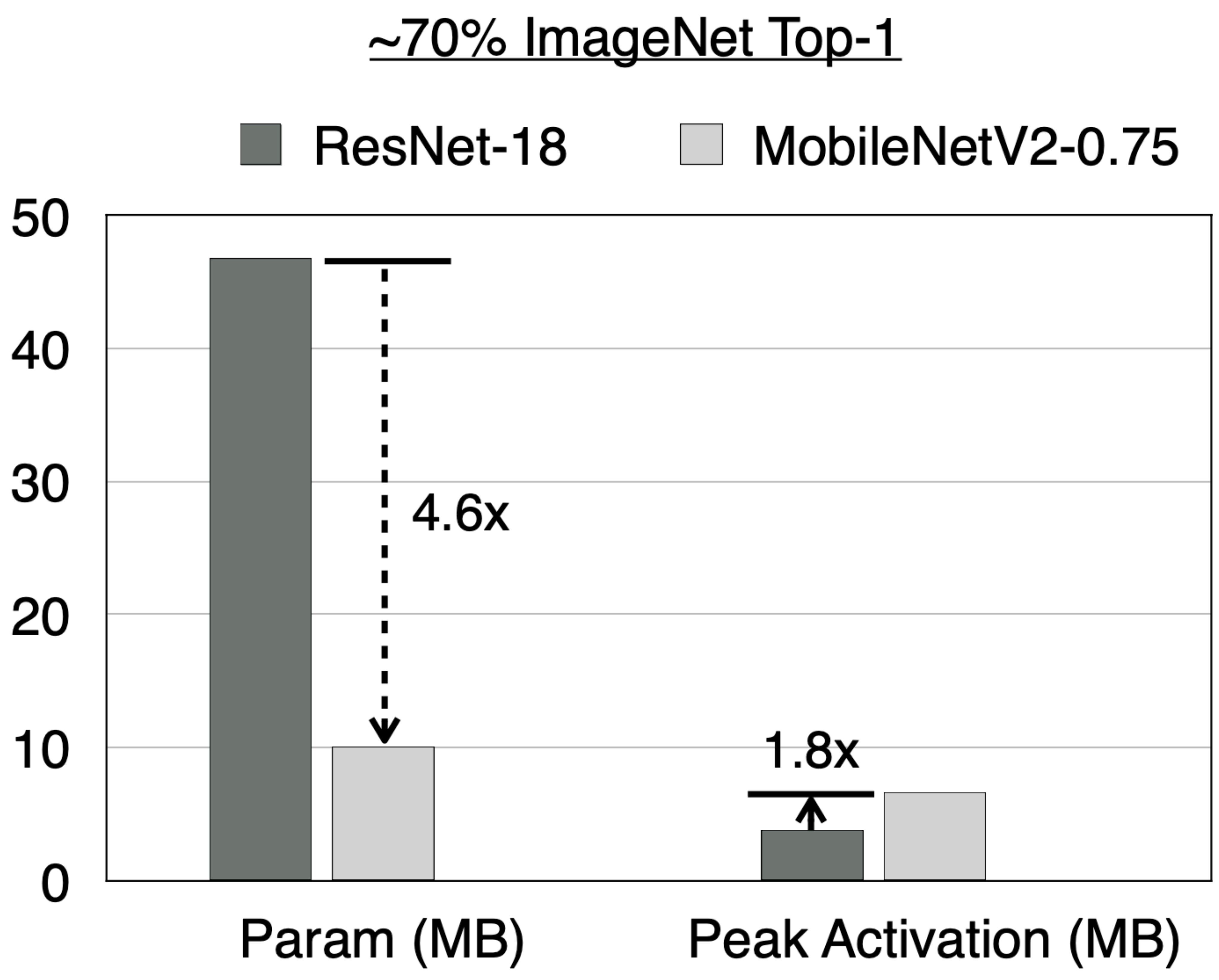
Tiny AI

	Cloud AI (NVIDIA V100)		**Mobile AI** (iPhone 11)		**Tiny AI** (STM32F746)		**ResNet-50**	**MobileNetV2**	**MobileNetV2 (int8)**
Memory	16 GB	$\xrightarrow{4\times}$	4 GB	$\xrightarrow{3100\times}$	**320 kB**	←gap→	7.2 MB	6.8 MB	1.7 MB
Storage	TB~PB	$\xrightarrow{1000\times}$	>64 GB	$\xrightarrow{64000\times}$	**1 MB**	←gap→	102MB	13.6 MB	3.4 MB

Motivation

- **Microcontroller.** If we want to make a model that trains on MCU, the real bottleneck is the **storage / peak memory**.

- **Problem.** The so-called "efficient" models do not really consider the **activation size**!

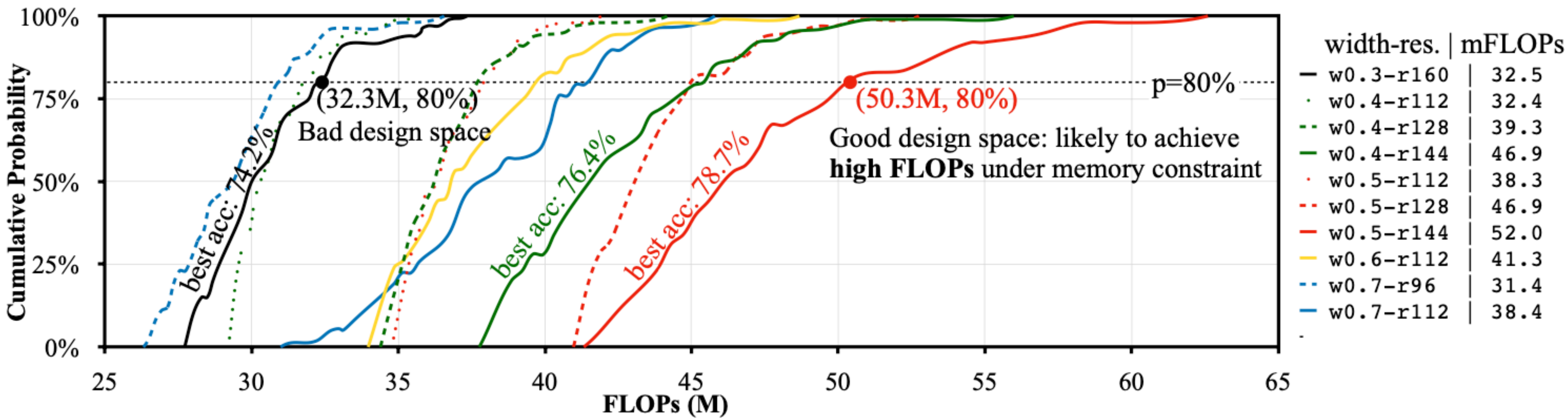


Idea

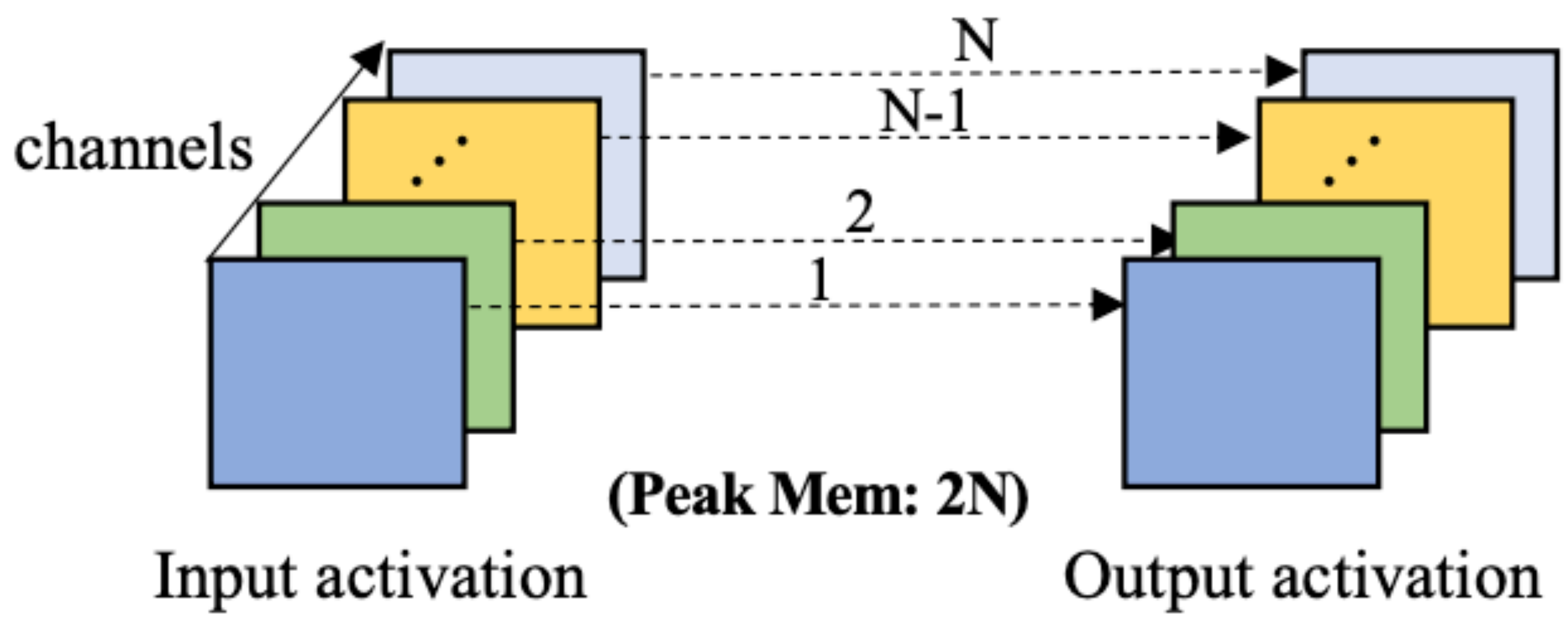
- **TinyNAS.** Run NAS with additional design dimension:

  kernel size × expansion ratio × depth × input resolution × width multiplier

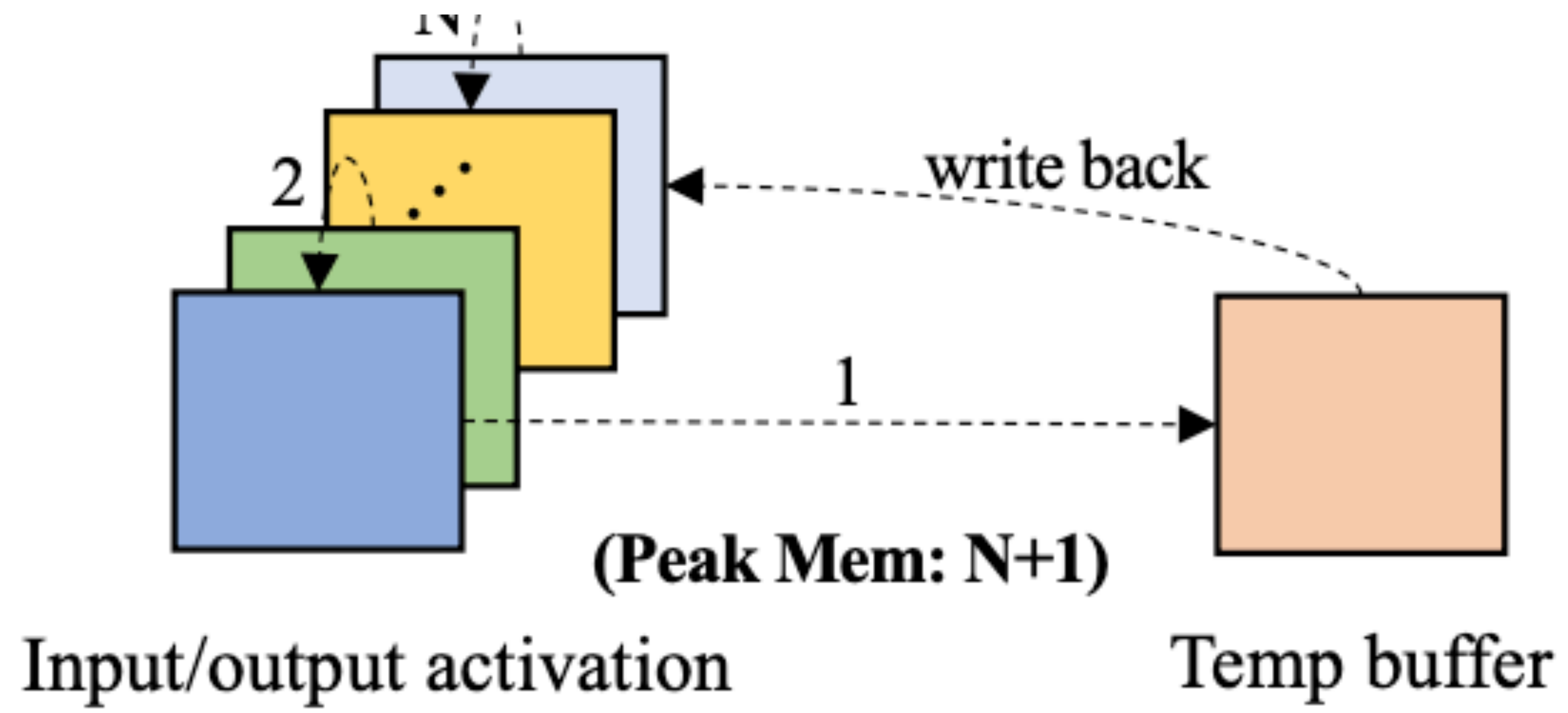
  - **FLOPs.** No longer a big issue—use more FLOPs for better performance!
    To search over the NAS search space, compare FLOPs instead of accuracy



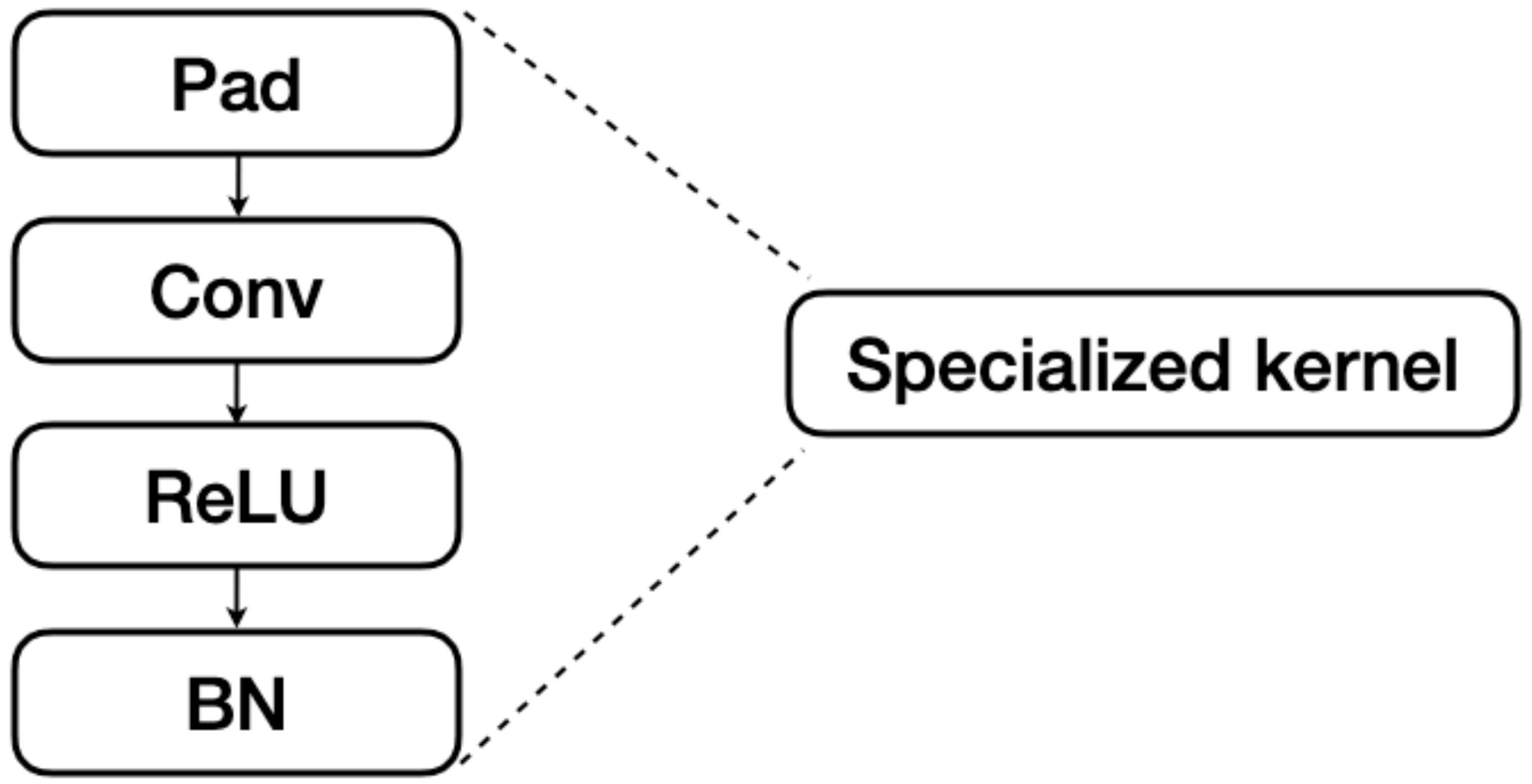
- **TinyEngine.** An inference library (such as TF-Lite Micro) with smaller peak memory
  (advanced kernel / compiler stuff... out of scope!)



(a) Depth-wise convolution



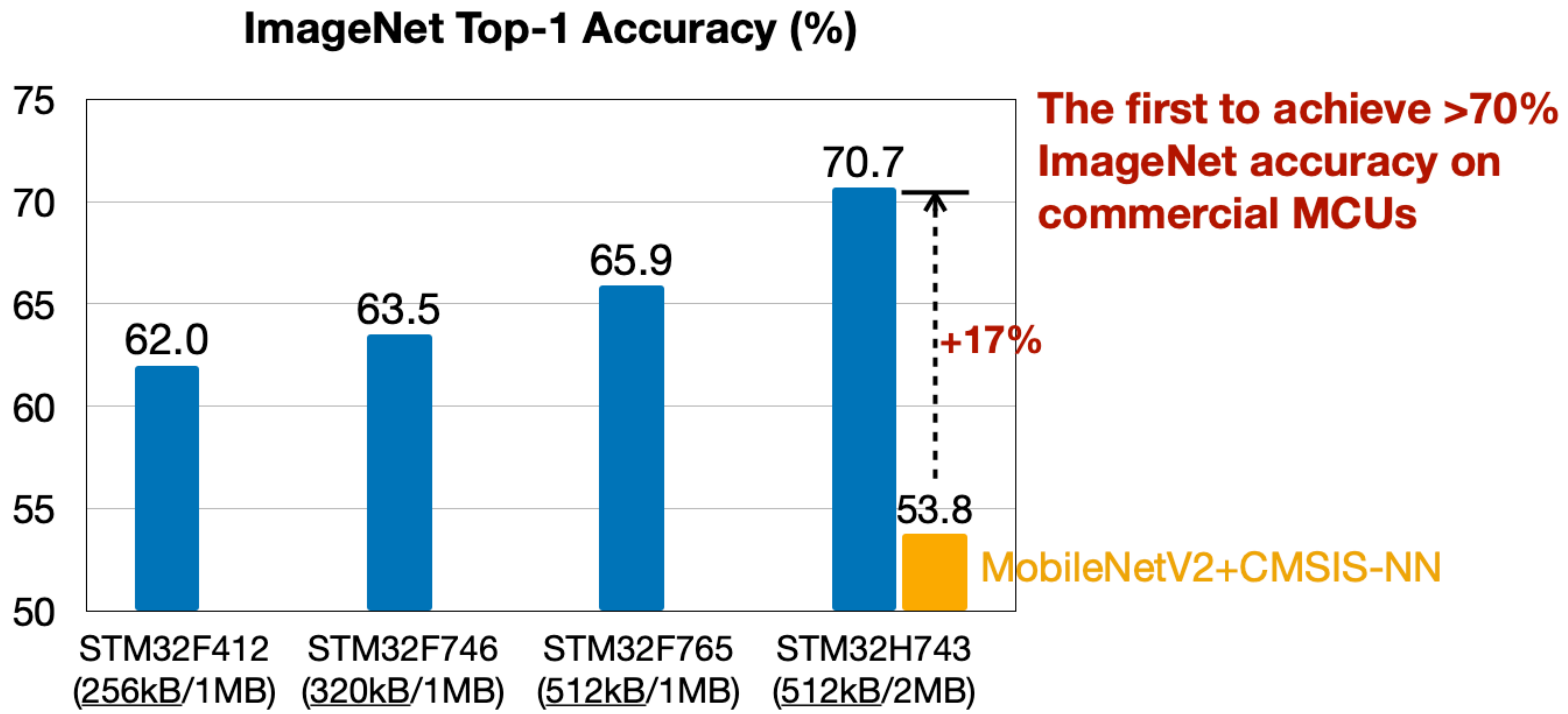
(b) In-place depth-wise convolution




```
/* dot products of convolution */
for (i = 0; i < kernel_x; i++)
    for(j = 0; j < kernel_y; j++)
        sum += x[i][j] * w[i][j];
```

e.g., fully unroll for 3x3 conv

```
/* dot products of convolution */
sum += x[0][0] * w[0][0];
sum += x[0][1] * w[0][1];
              ⋮
sum += x[2][2] * w[2][2];
```



ImageNet Top-1 Accuracy (%)
75
70
65
60
55
50
62.0
63.5
65.9
70.7
53.8
+17%
The first to achieve >70% ImageNet accuracy on commercial MCUs
MobileNetV2+CMSIS-NN
STM32F412 (256kB/1MB)
STM32F746 (320kB/1MB)
STM32F765 (512kB/1MB)
STM32H743 (512kB/2MB)

MCUNet V2

Lin et al. (NeurIPS 2021)

Lin et al., "MCUNetV2: Memory-efficient Patch-based inference for Tiny Deep Learning," NeurIPS 2021.

Motivation

- The peak memory requirement is different from layer to layer!

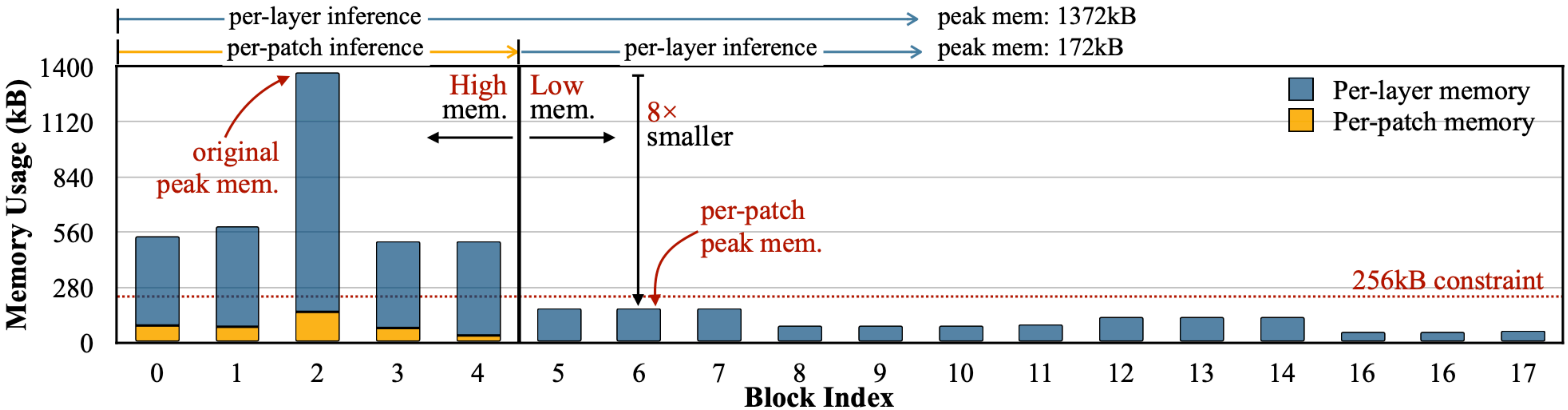
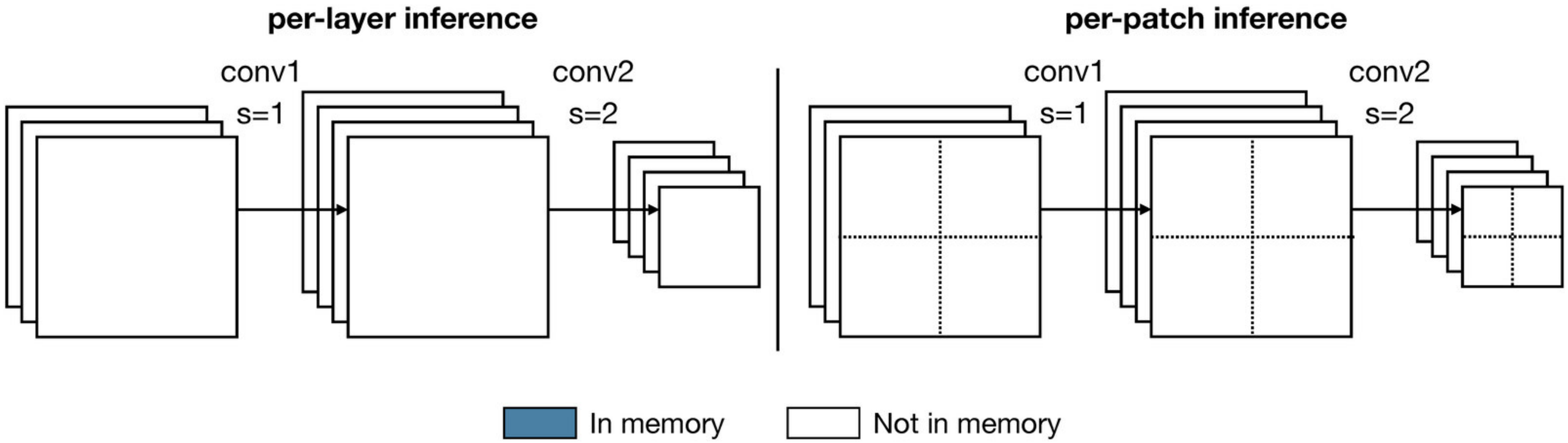


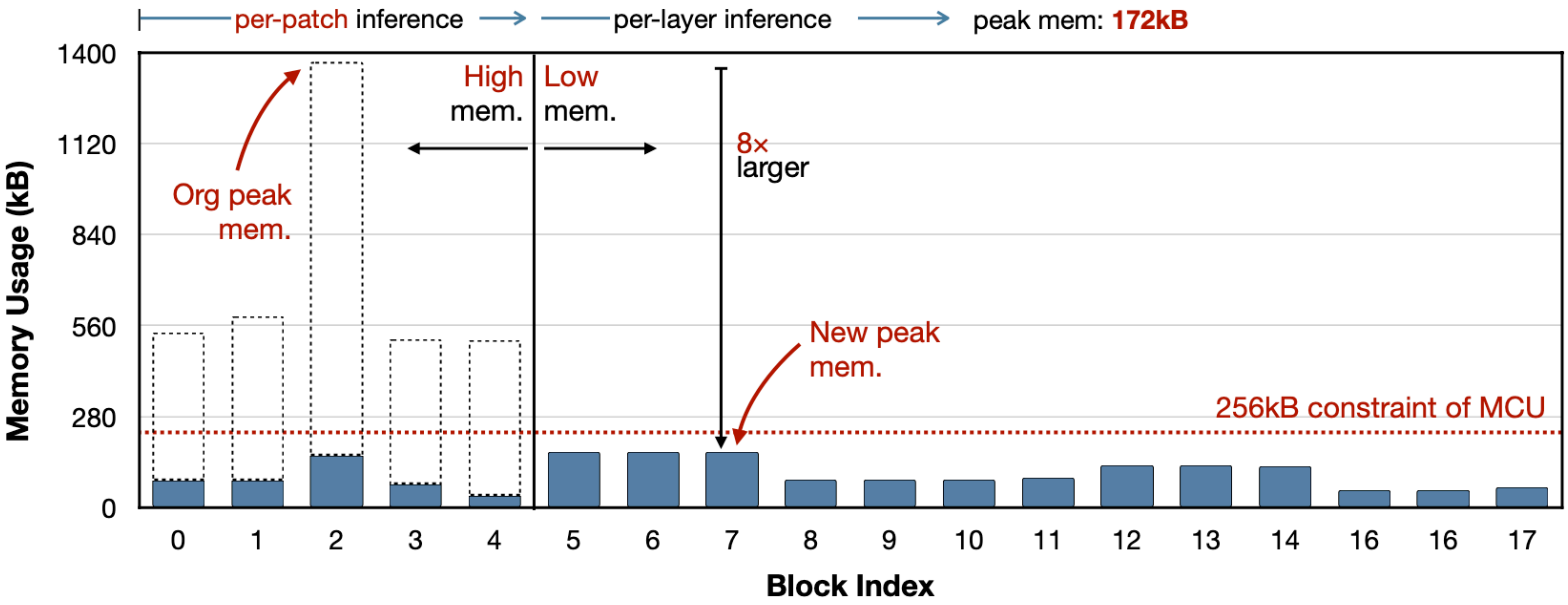
Figure 1. MobileNetV2 [44] has a very *imbalanced memory usage distribution*. The peak memory is determined by the first 5 blocks with high peak memory, while the later blocks all share a small memory usage. By using per-patch inference (4 × 4 patches), we are able to significantly reduce the memory usage of the first 5 blocks, and reduce the overall peak memory by 8×, fitting MCUs with a 256kB memory budget. Notice that the model architecture and accuracy are not changed for the two settings. The memory usage is measured in `int8`.

Solution

- For big layers, divide activations into patches for a smaller footprint inference.



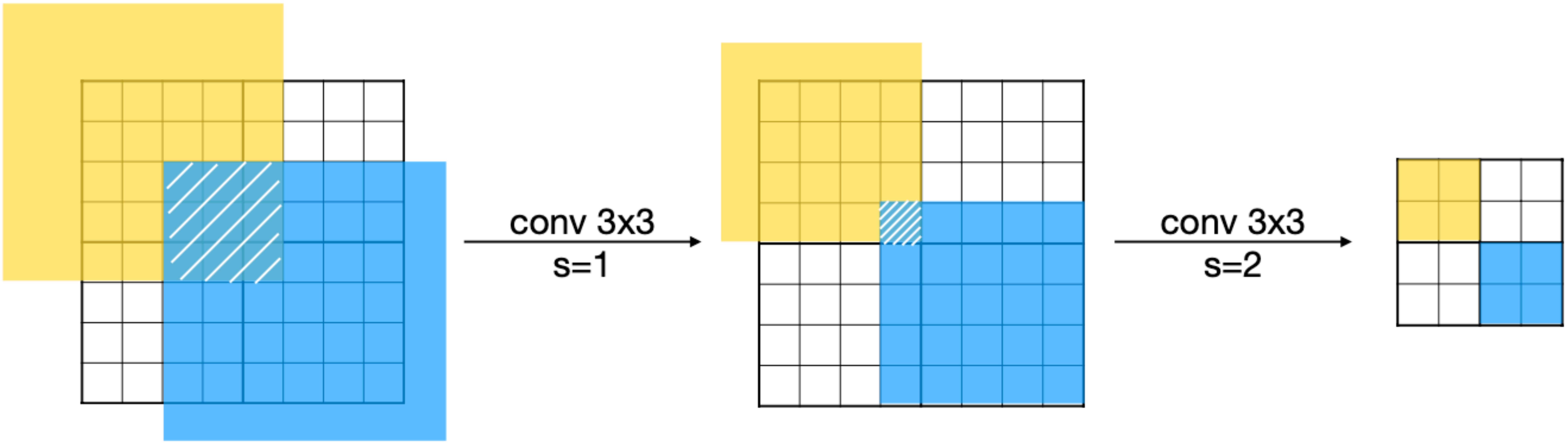
- Applying to MobileNetV2



Problem

- When you do patch-based inference, you get some computation overhead from overlaps. (10~20% MACs increase)

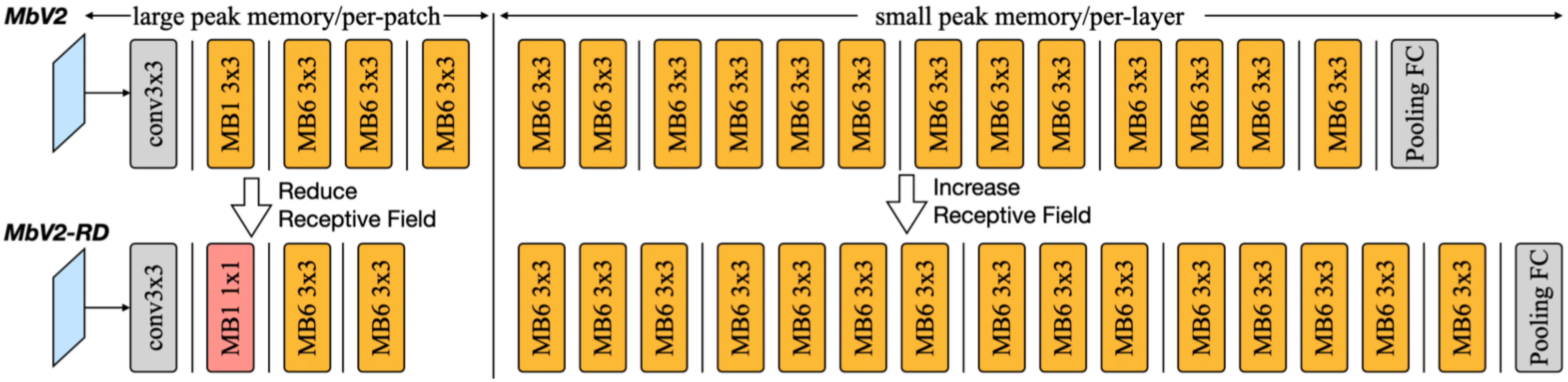
  - Using 2x2 patches



Spatial overlapping gets larger as **receptive field** grows!

Solution

- Play with the receptive fields—decrease for patch-based layers and increase for others.



- In fact, use NAS to optimize the receptive field distribution.

MCUNet V3

Lin et al. (NeurIPS 2022)

Lin et al., "On-Device Training Under 256KB Memory," NeurIPS 2022.

Motivation

- **Motivation.** Finally, we want to do some on-device training.
  But the memory bottleneck (for backdrop) is real!

  - We use low-precision weights, but it adds instability to the training...

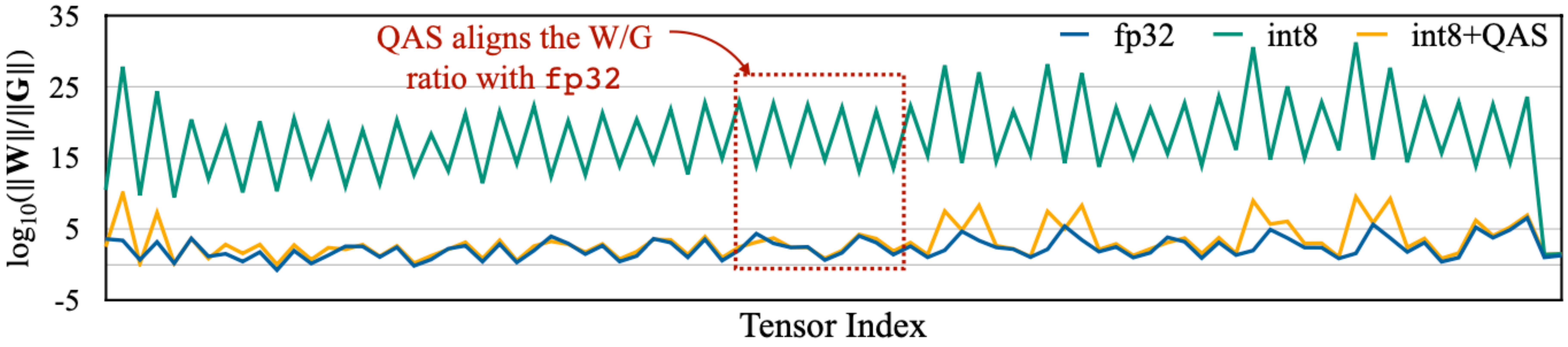


Figure 2. The quantized model has a very different weight/gradient norm ratio (*i.e.*, $\|\mathbf{W}\|/\|\mathbf{G}\|$) compared to the floating-point model at training time. QAS stabilizes the $\|\mathbf{W}\|/\|\mathbf{G}\|$ ratio and helps optimization. For example, in the highlighted area, the ratios of the quantized model fluctuate dramatically in a zigzag pattern (weight, bias, weight, bias, ...); after applying QAS, the pattern stabilizes and matches the `fp32` counterpart.

Solution

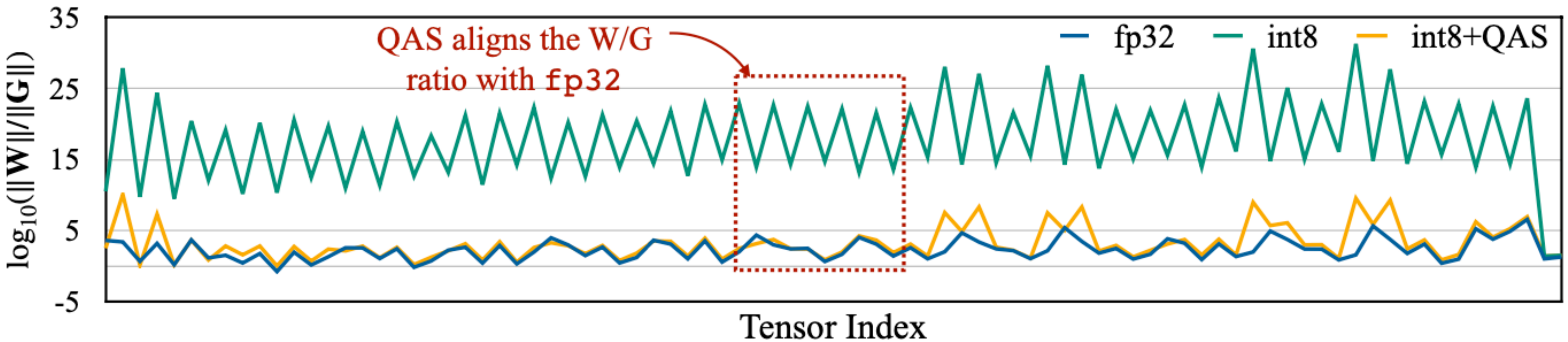
- **Quantization-aware Scaling.** We use different learning rate for each layer.
  - **Why Zigzag?** Let us do 8-bit quantization; we reparameterize weight $\mathbf{W}$ as

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

where $\bar{\mathbf{W}}$ has the maximum magnitude $127 = 2^7 - 1$. Then,

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

  - **Fix.** Use the gradient $\tilde{\mathbf{G}}_{\bar{\mathbf{W}}} = \mathbf{G}_{\bar{\mathbf{W}}} \cdot s_{\mathbf{W}}^{-2}$.



Solution

- + Also use sparse channel / layer updates...

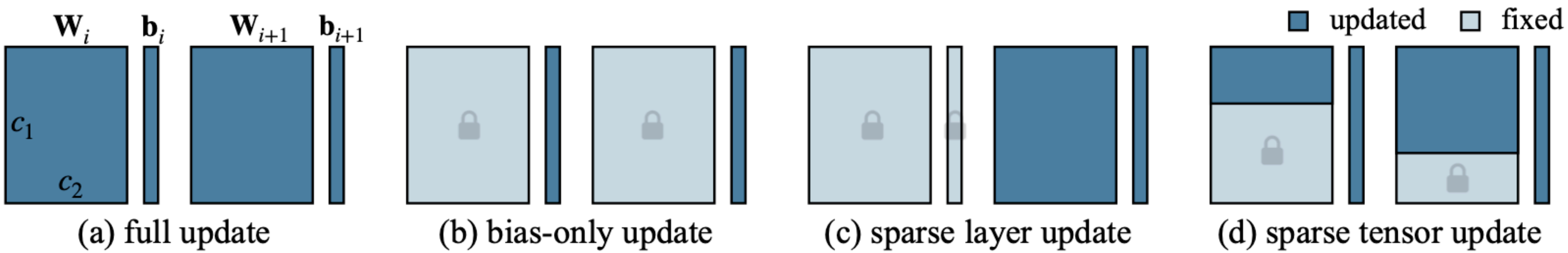


Figure 3. Different update paradigms of two linear layers in a deep neural network.

Figure 1. Algorithm and system co-design reduces the training memory from 303MB (PyTorch) to 149KB with the same transfer learning accuracy, leading to 2077× reduction. The numbers are measured with MobilenetV2-w0.35 [61], batch size 1 and resolution 128×128. It can be deployed to a microcontroller with 256KB SRAM.

Thoughts

Transition 1. From handcrafting to automated search (NAS),
but how to do design NAS search space and how to
evaluate the searched model is still up to us.

Transition 2. From inference efficiency to training efficiency,
but training efficiency research usually starts from
a model/method with a good inference efficiency
∵ key metrics overlap to a certain degree
∴ model compression techniques matter

Modules. When certain properties matter (e.g. privacy),
some modules (e.g., BN) could be viewed inefficient