DepthShrinker: A New Compression Paradigm Towards Boosting Real-Hardware Efficiency of Compact Neural Networks

Yonggan Fu[† 1] **Haichuan Yang**[2] **Jiayi Yuan**[1] **Meng Li**[2] **Cheng Wan**[1] **Raghuraman Krishnamoorthi**[2] **Vikas Chandra**[2] **Yingyan Lin**[1]

Abstract

Efficient deep neural network (DNN) models equipped with compact operators (e.g., depthwise convolutions) have shown great potential in reducing DNNs' theoretical complexity (e.g., the total number of weights/operations) while maintaining a decent model accuracy. However, existing efficient DNNs are still limited in fulfilling their promise in boosting real-hardware efficiency, due to their commonly adopted compact operators' low hardware utilization. In this work, we open up a new compression paradigm for developing real-hardware efficient DNNs, leading to boosted hardware efficiency while maintaining model accuracy. Interestingly, we observe that while some DNN layers' activation functions help DNNs' training optimization and achievable accuracy, they can be properly removed after training without compromising the model accuracy. Inspired by this observation, we propose a framework dubbed DepthShrinker, which develops hardware-friendly compact networks via shrinking the basic building blocks of existing efficient DNNs that feature irregular computation patterns into dense ones with much improved hardware utilization and thus real-hardware efficiency. Excitingly, our DepthShrinker framework delivers hardware-friendly compact networks that outperform both state-of-the-art efficient DNNs and compression techniques, e.g., a 3.06% higher accuracy and 1.53× throughput on Tesla V100 over SOTA channel-wise pruning method MetaPruning. Our codes are available at: https://github.com/facebookresearch/DepthShrinker.

[1]Department of Electrical and Computer Engineering, Rice University [2]Meta Inc. Correspondence to: Yingyan Lin <yingyan.lin@rice.edu>.

Proceedings of the 39^{th} International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022.

1. Introduction

Recent breakthroughs in deep neural networks (DNNs) have fueled a growing demand for deploying DNNs in real-world devices. However, the prohibitive complexity of DNNs stands at odds with the often constrained on-device resources. As such, many techniques aiming to boost DNNs' hardware efficiency have been developed, including pruning (Han et al., 2015a), quantization (Zhu et al., 2016; Zhou et al., 2016), and efficient DNN models (Howard et al., 2017; Google., 2020) leveraging compact operators (e.g., depthwise convolutions). Yet, the resulting DNN models from the above techniques mostly require dedicated DNN accelerators to achieve the desired hardware efficiency.

In parallel, *there exists a dilemma between the trends of efficient DNN design and modern computing platform advances*: while modern computing platforms (e.g., GPUs and TPUs) have consistently advanced to favor a higher degree of parallel computing, existing efficient DNN models often adopt light-weight operations that suffer from low hardware utilization and thus inferior achievable hardware efficiency. For instance, depthwise convolutions (Howard et al., 2017), commonly adopted in compact DNNs such as MobileNetV2 and EfficientNet, feature much more irregular computation patterns as compared to standard convolution layers, making it difficult to make good use of on-device resources due to their reduced data reuse opportunities and limiting existing efficient DNNs to unleash their theoretical potential (Chen et al., 2019). Therefore, there has been an increasing interest in developing more hardware friendly DNNs with improved hardware utilization to better leverage the power of parallelism in modern computing platforms (Chen & Zhao, 2018; Elkerdawy et al., 2020; Zhou et al., 2021).

To tackle the aforementioned gap between (1) the low hardware utilization of existing efficient DNNs and (2) the continuously increasing degree of computing parallelism of modern computing platforms, we ask an intriguing question: "*How do we design efficient DNNs that can simultaneously enjoy both the powerful expressiveness of state-of-the-*

[†]Work done in collaboration between Meta Reality Labs and Rice EIC lab during internship

art (SOTA) efficient DNN structures and boosted parallel-computing capability of modern computing platforms?" Inspired by RepVGG (Ding et al., 2021), which merges parallel branches to build decent single-branch networks, one natural thought is to merge consecutive layers into one single layer with dense computation patterns and improved hardware utilization. Nevertheless, it is non-trivial to merge layers along the depth dimension due to the associated activation functions, which are desired for introducing more non-linearity to empower the model capacity.

Interestingly, we observe that while some DNN layers' activation functions help DNNs' training optimization and thus achievable accuracy, they can be properly removed after training without compromising the model accuracy. An exciting outcome is that the remaining consecutive linear operations between which the activation functions are removed can be merged into one single linear operation. **Notably, if the two activation functions in an inverted residual block (Sandler et al., 2018), the basic building block of SOTA efficient DNNs (Sandler et al., 2018; Tan & Le, 2019; Wu et al., 2019; Howard et al., 2019), are removed, its two pointwise convolution layers and one depthwise convolution layer as well as the associated residual connection can be merged to one dense convolution with (1) a kernel of the same size as the original depthwise convolution and (2) the same number of input/output channels as the original inverted residual block**. Excitingly, the resulting dense convolution enjoys a much improved hardware utilization as compared to that of both the pointwise convolutions of kernel size 1×1 and depthwise convolution in the original inverted residual block, enabling the derived DNN to win boosted hardware efficiency while maintaining the original accuracy.

Driven by the above exciting discovery, we propose a new compression paradigm towards real-hardware efficient compact networks and make the following contributions:

- We conduct experiments to show that the commonly adopted building blocks in existing efficient DNNs are inferior in hardware efficiency as compared to dense operations with the same theoretical complexity.
- Motivated by the above, we propose DepthShrinker that advocates **merging consecutive layers**, between which the activation functions are learned to be unimportant for inference, **into one single dense layer**. DepthShrinker's derived DNNs can largely leverage the high degree of parallelism in modern computing platforms and thus boost hardware efficiency while maintaining the original models' accuracy.
- DepthShrinker opens up a new perspective towards powerful and hardware-efficient DNNs, and can be viewed as some sort of soft layer pruning, in contrast to layer-wise pruning, i.e., *merging* vs. *hard pruning*. Notably, DepthShrinker delivers DNNs that outperform both SOTA channel- and layer-wise pruning techniques, e.g., a 3.06% higher accuracy and 1.53× throughput on Tesla V100 over SOTA channel-wise pruning method MetaPruning (Liu et al., 2019).
- Extensive experiments and ablation studies validate that DepthShrinker can (1) largely push forward the frontier of DNNs' achievable accuracy-efficiency trade-off, and (2) serve as an augmentation technique for boosting tiny DNNs' accuracy.

2. Related Works

Efficient DNNs. Various efficient DNNs have been developed. Early efficient DNNs mainly rely on human experts' manual design, e.g., MobileNets (Howard et al., 2017; Sandler et al., 2018) boost model efficiency and accuracy trade-offs via depthwise convolution, which has become a standard operator for efficient DNNs. In parallel, hardware efficient operators have been proposed (Wu et al., 2017; Chen et al., 2020) as alternatives for convolution. Thanks to the great success of neural architecture search (NAS) (Zoph & Le, 2016; Zoph et al., 2018), automated efficient DNN design via reinforcement learning (Tan et al., 2019; Howard et al., 2019; Tan & Le, 2019) and differentiable search (Liu et al., 2018; Wu et al., 2019; Cai et al., 2018) have been proposed. However, existing efficient DNNs are still limited in their hardware efficiency due to the low hardware utilization of their basic building blocks, e.g., depthwise convolutions (Howard et al., 2017).

DNN compression techniques. Existing DNN compression techniques reduce the model complexity by pruning (Han et al., 2015b;a; Wen et al., 2016; He et al., 2018; Liu et al., 2019; He et al., 2017; 2020; 2019; Dong et al., 2017), quantization (Courbariaux et al., 2015; 2016; Rastegari et al., 2016; Fu et al., 2020; 2021a), low-rank decomposition (Yin et al., 2021; Sainath et al., 2013; Nakkiran et al., 2015), or dynamic inference (Teerapittayanon et al., 2016; Wang et al., 2018; Shen et al., 2020), while striving to maintain a decent accuracy. Nevertheless, it is well known that general computing platforms (e.g., GPUs and CPUs) cannot fully benefit from DNN compression via low-bit quantization, low-rank decomposition, or dynamic inference in terms of hardware efficiency, and are still limited in fulfilling the efficiency improvement from pruning.

Layer-wise pruning. The most relevant work to DepthShrinker is layer-wise pruning (Chen & Zhao, 2018; Elkerdawy et al., 2020; Zhou et al., 2021; Xu et al., 2020), which prunes an entire layer/block motivated by the fact that pruning a layer is more effective in reducing hardware latency (Xu et al., 2020) compared with channel-wise pruning.

Table 1. Measured throughput of both the MobileNetV2 (including EfficientNet-Lite0) and ResNet families, as well as their corresponding dense counterparts on three commercial devices. All the reported numbers are real-device Frame-Per-Second (FPS).

Model	GFLOPs	Tesla V100 GPU		RTX 2080Ti GPU		TX2 Edge GPU	
		Original	Dense	Original	Dense	Original	Dense
MobileNetV2	0.33	3088	**12090** (↑**3.91**×)	2364	**9351** (↑**3.96**×)	115	**397** (↑**3.45**×)
MobileNetV2-1.4	0.63	2127	**8846** (↑**4.16**×)	1617	**6869** (↑**4.25**×)	73	**267** (↑**3.66**×)
Efficientnet-Lite0	0.41	2731	**11174** (↑**4.09**×)	2185	**9577** (↑**4.38**×)	98	**360** (↑**3.67**×)
ResNet-50	4.14	1079	**2182** (↑**2.02**×)	874	**1862** (↑**2.13**×)	45	**53** (↑**1.18**×)
ResNet-101	7.88	642	**1509** (↑**2.35**×)	538	**1279** (↑**2.38**×)	28	**44** (↑**1.57**×)
ResNet-152	11.62	449	**1082** (↑**2.41**×)	378	**917** (↑**2.43**×)	19	**30** (↑**1.58**×)

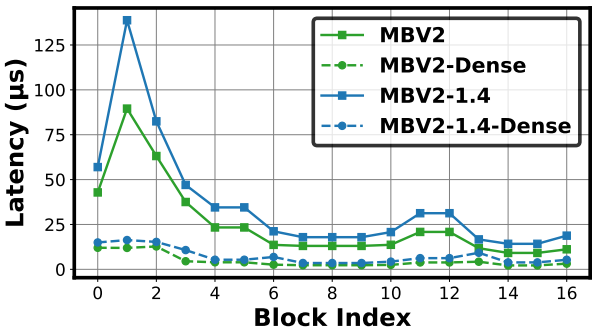

Figure 1. Visualizing the block-wise latency of the inverted residual blocks (a total of 17) in MobileNetV2/MobileNetV2-1.4 (*solid lines*) and their corresponding dense convolutions (*dashed lines*) on an RTX 2080Ti GPU. "MBV2" denotes MobileNetV2.

Specifically, (Chen & Zhao, 2018; Elkerdawy et al., 2020) prune layers based on their proposed criteria; while (Zhou et al., 2021) and (Xu et al., 2020) determine which layers/blocks to be pruned via evolutionary search and differentiable optimization, respectively. However, aggressive layer pruning inevitably suffers from non-trivial accuracy drops under large compression ratios due to the difficulty in restoring the pruned models' accuracy. Instead of hard pruning, our DepthShrinker merges consecutive linear operations into one dense operation after training, and can win both accuracy and hardware efficiency thanks to (1) its better maintained model expressiveness and (2) the high utilization and thus lower latency of the merged dense operation.

3. Motivating Inspiration and Observations

3.1. Inspiration Drawn from Previous Works

Shallow networks with a higher utilization favor real-hardware efficiency. Recent works (Elkerdawy et al., 2020; Xu et al., 2020) show that shallow networks favor a higher degree of parallel processing and thus higher hardware utilization, leading to better real-hardware efficiency on modern computing devices, e.g., GPUs, over their deeper counterparts with a comparable computational cost; this is also further validated by our profiling experiments in Sec. 3.2. Nevertheless, existing shallow networks are still not able to approach the accuracy of their deep counterparts, motivating us to innovate shallow networks with boosted accuracy.

Linear operations in DNNs can be merged. It is known that linear operations can be properly merged. RepVGG (Ding et al., 2021) shows that linear operations in parallel branches of a DNN can be merged to deliver competitive single-branch networks. Inspired by this, one natural thought for building powerful compact networks is to merge consecutive layers in a SOTA DNN to reduce the model depth. However, layers cannot be directly merged along the depth dimension due to the non-linear activation functions. This motivates us to question "whether some activation functions can be properly removed for inference".

The role of activation functions. We hypothesize that the answer to the above question is positive based on existing DNN compression (Han et al., 2016; Jacob et al., 2018) and training (Zhou et al., 2020; Cai et al., 2021) works, which show that DNNs' higher complexity benefits training but can be trimmed down during inference without hurting the accuracy. Specifically, iterative pruning (Han et al., 2016) and quantization-aware training (Jacob et al., 2018) train DNNs with their original complexity and then sparsify/quantize the model for inference without hurting the accuracy; Meanwhile, (Zhou et al., 2020; Cai et al., 2021) augment DNNs via expanding their widths during training for improved accuracy, while the models during inference remain the same. One inspiration from these prior arts is that activation functions can be viewed as one specific model dimension in enhancing DNNs' complexity and expressiveness, and thus some might be properly removed after training without hurting the accuracy.

3.2. Motivating Observations from Real-device Profiling

Since our work is hardware-driven and aims to improve real-device efficiency instead of theoretical ones, we conduct extensive real-device profiling experiments to validate our hypothesis and to gain better understandings of the design space, and they are summarized in this subsection.

Key hypothesis/motivation. While the commonly used bottleneck blocks (He et al., 2016) and more efficient inverted residual blocks (Sandler et al., 2018) have shown

impressive theoretical efficiency and accuracy trade-offs, their real-device efficiency is inferior as compared to their dense counterparts under the same computational complexity, due to their more irregular computation patterns that cause reduced data reuse and lower hardware utilization.

Experiment setup. In our profiling, we replace *each building block* in both the MobileNetV2 (Sandler et al., 2018) (including EfficientNet-Lite0 (Google., 2020)) and ResNet (He et al., 2016) families with *one dense convolution layer* (1) of the same kernel size as the second convolution layer of each block, which is the only convolution with a kernel size larger than 1×1 within a bottleneck/inverted residual block and (2) with a scaled number of channels to **maintain the same floating-point operations (FLOPs) as the original block**. We summarize the real-device throughput of both these two network families featuring two different types of basic building blocks and their dense counterparts on ImageNet with a resolution of 224×224 in Tab. 1.

Considered devices and measurement settings. We consider three commercial devices, including (1) NVIDIA Tesla V100 GPU (NVIDIA., c), (2) NVIDIA RTX 2080Ti GPU (NVIDIA., b), and (3) Jetson TX2 Edge GPU (NVIDIA., a), to cover both Desktop and edge GPUs. We adopt a batch size of 128 for the first two devices, following (Ding et al., 2021), and 64 for the last device, and Frame-Per-Second (FPS) as the efficiency metric.

Results and analysis. As shown in Tab. 1, we can observe that (1) the dense counterparts consistently achieve a higher throughput against the original networks with the same FLOPs, regardless of the model families and profiling devices. Specifically, the dense convolution counterparts on top of the MobileNetV2 family boost the throughput on a Tesla V100/RTX 2080Ti GPU and TX2 Edge GPU by $3.91\times$ $\sim$$4.38\times$ and $3.45\times$ $\sim$$3.67\times$, respectively; and similarly, they increase the throughput on a Tesla V100/RTX 2080Ti GPU and TX2 Edge GPU by $2.02\times$ $\sim$$2.43\times$ and $1.18\times$ $\sim$$1.58\times$, respectively, on top of the ResNet family. To further understand this, we also visualize the block-wise latency of MobileNetV2/MobileNetV2-1.4 on a RTX 2080Ti GPU in Fig. 1, including the latency of both the original block and the corresponding dense convolution. It shows that the dense counterpart for each block can consistently reduce the latency by up to 88.2%.

This set of profiling experiments indicates that (1) replacing the commonly adopted building blocks with dense operations of the same FLOPs can notably boost real-device efficiency, thanks to the improved utilization of hardware resources; (2) The throughput improvement is more notable on top of the MobileNetV2 family than the ResNet family, because depthwise convolutions in the former and widely adopted in existing efficient DNNs introduce more irregular computation patterns and thus we see a more pronounced improvement after replacing them with dense ones; and (3) throughput improvement is consistently observed across different devices, and is larger on the Tesla V100/2080Ti GPU than the TX2 Edge GPU since the former has a higher degree of parallel-processing that favors the achievable throughput of the dense counterparts, indicating an even larger efficiency improvement of our DepthShrinker, in line with more parallelism trend of modern AI-driven computing platforms.

Remark. The hardware utilization and thus real-device efficiency improvements are mainly attributed to two perspectives: (1) from the operation perspective, irregular operations have less data reuse opportunities and thus require more data movement costs (Chen et al., 2019), e.g., in our profile experiment, both the standard convolutions with a kernel size of 1×1 and the depthwise convolutions are replaced with dense convolutions with a larger kernel size of 3×3, leading to more data reuse opportunities and less data movements under the same FLOPs; and (2) from the depth/width trade-off perspective, replacing a building block with one dense convolution of the same FLOPs would both shallow and widen the original network, and thus favor a higher utilization when running on modern computing platforms featuring an increasingly high degree of parallelism. We further study the independent impact of the above perspective (2) in the Appendix. C.

4. The Proposed DepthShrinker Framework

4.1. Overview

Key idea. DepthShrinker aims to develop real-hardware efficient DNNs favoring high hardware utilization by removing redundant activation functions and then merging the resulting consecutive linear operations. The key idea is that by removing the two activation functions within an inverted residual block (Sandler et al., 2018), i.e., the basic building block in most efficient DNNs, the entire block can then be merged into one dense convolution layer with the same kernel size as the original block's depthwise convolution and the same number of input/output channels as the original block. The exciting outcome is much improved real-hardware efficiency as profiled in Sec. 3.2.

Framework overview. To achieve the above aim, two non-trivial challenges exist: which activation functions to be removed and how to restore the accuracy with fewer remaining activation functions after the removal. To tackle these, DepthShrinker built on top of SOTA efficient DNNs integrates a three-stage effort as shown in Fig. 2: (1) identify redundant activation functions, (2) remove the identified activation functions and fine-tune the resulting DNNs from stage (1), and (3) merge consecutive layers between which the activation functions are removed to deliver the final networks. Note that we apply our DepthShrinker on top of

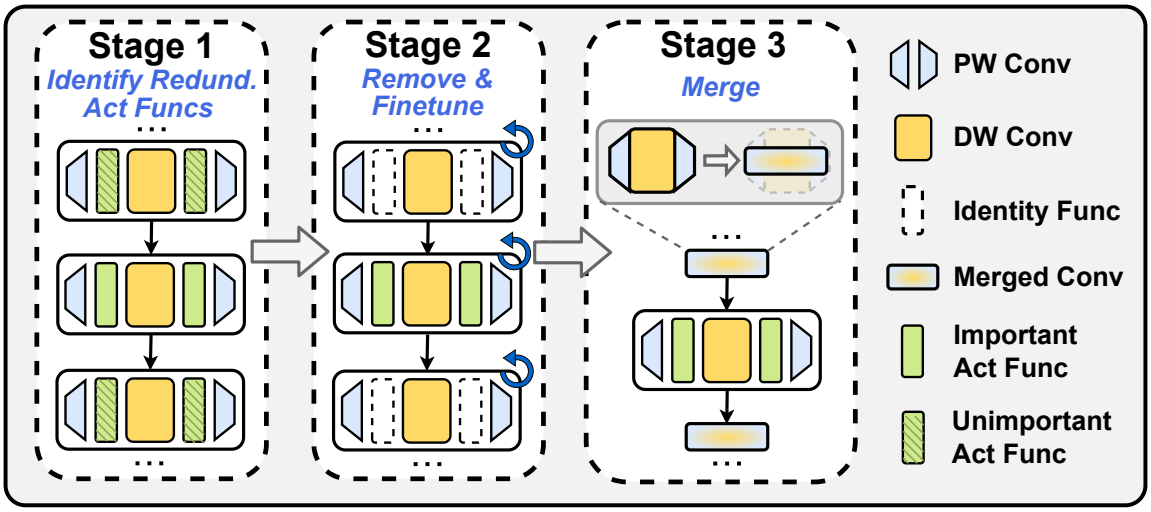

Figure 2. Overview of our DepthShrinker framework and its three-stage design. "PW" and "DW" denote pointwise/depthwise convolutions, respectively. During merging, we merge the two pointwise convolutions and one depthwise convolution in blocks whose activation functions are removed, into one *dense* convolution.

publicly available pretrained models, following common practice in model compression (He et al., 2018; Wen et al., 2016; Han et al., 2015a; Jacob et al., 2018).

4.2. Stage 1: Identify Redundant Activation Functions

To identify unimportant activation functions, predefined criteria like those for layer-wise pruning (Li et al., 2016; Wen et al., 2016) may not be appropriate as activation functions of different layers are coupled, e.g., removing preceding activation functions might change the feature distributions of the following layers. Therefore, we propose a differentiable search method to learn the importance of all activation functions considering their joint influence, as inspired by recent pruning works (Kang & Han, 2020; Ramanujan et al., 2020; Fu et al., 2021b).

Search method overview. Our search method assigns a learnable mask $m \in \mathbb{R}^N$ (N is the total number of activation functions) to all activation functions, serving as a proxy of the activation functions' importance score. When updating m during search, the coupling effect among different activation functions should be considered, while ensuring that the sparsity of m is sufficiently high, e.g., higher than $(1 - k/N)$ where k is the number of remained activation functions, to satisfy the target efficiency after the merging stage (see sec. 4.4). In DepthShrinker, the search method jointly learns the mask m together with model weights θ.

Search method formulation. Deriving the optimal θ and m is essentially a bi-level optimization problem (Liu et al., 2018). In DepthShrinker, we approximate it as a one-level optimization formulation (Xie et al., 2018; Hu et al., 2020; Bi et al., 2020) to jointly update θ and m:

$$\arg\min_{\theta,m} \sum_i \ell(\hat{y}_{\theta,m}(x_i), y_i) \quad \text{s.t. } \|m\|_0 \leqslant k \qquad (1)$$

where ℓ is the loss function, x_i and y_i are the i-th input and label pair, and $\hat{y}_{\theta,m}(x_i)$ is the predicted label under the parameters θ and activation mask m. To achieve the target sparsity in m, we impose an L_0 constraint on m via activating only its top k elements during forward. Specifically, we adopt a binary mask $\hat{m} \in \{0,1\}^N$ to approximate the top k elements of m using 1 and 0 otherwise during forward, while all the elements in m are updated via straight-through estimation (Bengio et al., 2013) during backward. In particular, the forward function can be formulated as:

$$f^{(l)}_{\theta,\hat{m}}(\cdot) := (\hat{m}_l \cdot \sigma + (1 - \hat{m}_l) \cdot \mathbb{1}) \circ \mathcal{T}_{\theta_l} \circ f^{(l-1)}_{\theta,\hat{m}}(\cdot) \qquad (2)$$

where $f^{(l)}_{\theta,\hat{m}}$ is the network function for the first l layers, $\circ$ is the operator of function composition, σ and $\mathbb{1}$ denote an activation function and identity mapping, respectively, and $\mathcal{T}_{\theta_l}$ is a transformation (e.g., convolution or other linear operations) parameterized by θ_l. The binary mask $\hat{m}_l$ in Eq. (2) guarantees that an activation function in the l-th layer is either fully enabled or disabled. During backward, we directly pass the gradients of the binary mask $\hat{m}$ to m, i.e., $\frac{\partial \ell}{\partial m} \approx \frac{\partial \ell}{\partial \hat{m}}$. Since only the activation functions corresponding to the top k values of mask m participate in the forward process, activation functions with larger m values are more likely to be kept, thus larger m values after training indicate higher importance.

Search method implementation. Our search method makes two settings: (1) block-wise shrink and (2) latency-aware decay on m. For the former, since we aim to merge the whole inverted residual blocks into one dense convolution, we share the mask values of m corresponding to the two activation functions in one block, i.e., both of the two activation functions are either removed or kept. For the latter, we additionally add an L_1 decay on each element of m weighted by the corresponding block's latency during

search to penalize the importance of the costly blocks. Note that in this work we directly adopt the pre-measured latency on RTX 2080Ti GPU during search, and recognize that it is straightforward to make it platform-aware for further boosting the efficiency at the cost of a longer search time.

4.3. Stage 2: How to Fine-tune

After the above search process, the least important activation functions with the smallest m are removed, and fine-tuning is performed to restore the accuracy. The following solutions have been proposed and validated:

Adding additional activation functions for free. There is no nonlinear function following the dense convolutions after the merge stage, since an inverted residual block (Sandler et al., 2018) contains only two activation functions. To boost the achieved accuracy, we additionally add an activation function (i.e., ReLU6 in this work) after each merge convolution, which incurs a negligible hardware cost.

Self-distillation. In DepthShrinker, we can optionally enable a self-distillation mechanism during fine-tuning, i.e., conducting knowledge distillation (Hinton et al., 2015) under the guidance of the original network with all activation functions on to further boost the derived network's accuracy. Note that we only assort to the original network as the teacher without introducing extra models.

4.4. Stage 3: How to Merge

After fine-tuning the resulting network with unimportant activation functions removed, the final step is to merge adjacent linear operations (e.g., convolutional/fully-connected, average pooling, or batch normalization layers).

Merging two adjacent layers. Without loss of generality, here we consider two adjacent convolution layers with an input feature $X \in \mathbb{R}^{H_1 \times W_1 \times c_1}$, intermediate feature $Z \in \mathbb{R}^{H_2 \times W_2 \times c_2}$, output feature $Y \in \mathbb{R}^{H_3 \times W_3 \times c_3}$, and kernel weights $K^{(1)} \in \mathbb{R}^{d_1 \times d_1 \times c_1 \times c_2}$ and $K^{(2)} \in \mathbb{R}^{d_2 \times d_2 \times c_2 \times c_3}$ for the first and second layers, between which the activation is removed, and assume:

$$Z_{m,n,s} = \sum_{i=0}^{d_1-1} \sum_{j=0}^{d_1-1} \sum_{r=0}^{c_1-1} X_{m-i,n-j,r} K^{(1)}_{i,j,r,s} \quad (3)$$

$$Y_{m,n,t} = \sum_{i=0}^{d_2-1} \sum_{j=0}^{d_2-1} \sum_{s=0}^{c_2-1} Z_{m-i,n-j,s} K^{(2)}_{i,j,s,t} \quad (4)$$

The above two layers can be merged into one single layer. Assuming the stride of both layers s_1 and s_2 is 1, we have:

$$Y_{m,n,t} = \sum_{i=0}^{d-1} \sum_{j=0}^{d-1} \sum_{r=0}^{c_1-1} X_{m-i,n-j,r} K_{i,j,r,t} \quad (5)$$

where $d = d_1 + d_2 - 1$ and

$$K_{i,j,r,t} = \sum_{p=\underline{p}}^{\overline{p}} \sum_{q=\underline{q}}^{\overline{q}} \sum_{s=0}^{c_2-1} K^{(1)}_{i-p,j-q,r,s} K^{(2)}_{p,q,s,t} \quad (6)$$

where $K_{i,j,r,t}$ is the merged kernel of size $d \times d$, and $\underline{p} = \max(0, i - d_1 + 1)$, $\overline{p} = \min(d_2 - 1, i)$, $\underline{q} = \max(0, j - d_1 + 1)$, and $\overline{q} = \min(d_2 - 1, j)$. Note that when the stride of layers s_1 and s_2 is larger than 1, the kernels can still be merged into one with a stride of $s_1 \times s_2$ and kernel size of $((d_2 - 1) \times s_1 + d_1) \times ((d_2 - 1) \times s_1 + d_1)$.

Merging inverted residual blocks. An important insight from the above analysis is that when consecutive convolution layers are merged into one convolution layer, the number of both the input and output channels for the resulting convolution layer is only determined by the number of the input channels in the first convolution layer and the number of output channels in the last convolution layer, respectively, **regardless of the intermediate layer structures**. As a result, **the design rule of inverted residual blocks (Sandler et al., 2018), i.e., three convolution layers with their number of channels first expanded and then decreased, is naturally favorable to our DepthShrinker's derived networks consisting of merged convolutions with only the decreased number of input and output channels**. We believe this also sheds light on future hardware-efficient DNN designs.

4.5. DepthShrinker$^+$: Expand-then-Shrink

The vanilla design of our DepthShrinker described above leverages the insight that *unimportant activation functions can be properly removed after training without hurting the inference accuracy*. Excitingly, this insight can also be leveraged to improve DNN training. Specifically, we propose to train a given DNN via an Expand-then-Shrink strategy, and term it as DepthShrinker$^+$. In a DepthShrinker$^+$ training, we first (1) expand one or some of the convolution layers to become inverted residual blocks, which benefits the training optimization thanks to the increased overparameterization in the expanded model, (2) train the expanded DNN, and then (3) apply DepthShrinker to merge the aforementioned newly introduced blocks to recover the original network structure. As such, this training scheme can be viewed as augmenting the original DNN with enhanced complexity during training to favor its training optimization and thus achievable accuracy, while the inference efficiency remains to be the same.

5. Experiment Results

5.1. Experiment Setup

Networks and datasets. We apply DepthShrinker to both the MobileNetV2 (Howard et al., 2017) and EfficientNet-Lite (Google., 2020) (i.e., a hardware-efficient variant of

Table 2. Benchmark DepthShrinker with SOTA channel-wise pruning method MetaPruning (Liu et al., 2019) and uniform pruning on MobileNetV2-1.4@ImageNet in terms of FPS measured on three devices. All baseline accuracies are their reported ones (Liu et al., 2019).

Model	Acc (%)	MFLOPs	Tesla V100	RTX 2080Ti	TX2
MBV2-1.4	75.30	630	2127 (↑1.00×)	1617 (↑1.00×)	73 (↑1.00×)
MetaPruning-1.0×	73.20	332	3159 (↑1.49×)	2527 (↑1.56×)	115 (↑1.58×)
MBV2-1.4-DS-A	74.65/75.29/75.65	519	3827 (↑1.80×)	2881 (↑1.78×)	134 (↑1.84×)
MBV2-1.4-DS-B	73.67/74.8/75.13	502	4778 (↑2.25×)	3356 (↑2.08×)	163 (↑2.23×)
MBV2-1.4-DS-C	73.38/74.55/74.91	492	4597 (↑2.16×)	3537 (↑2.19×)	159 (↑2.18×)
Uniform-0.65×	67.20	182	4004 (↑1.88×)	3147 (↑1.95×)	161 (↑2.21×)
MetaPruning-0.65×	71.70	160	4336 (↑2.04×)	3691 (↑2.28×)	179 (↑2.45×)
MBV2-1.4-DS-D	72.51/73.93/74.50	484	5560 (↑2.61×)	3926 (↑2.43×)	184 (↑2.52×)
MBV2-1.4-DS-E	72.20/73.85/74.43	474	5317 (↑2.50×)	4175 (↑2.58×)	179 (↑2.45×)
Uniform-0.35×	54.60	68	6266 (↑2.95×)	5607 (↑3.47×)	263 (↑3.60×)
MetaPruning-0.35×	64.50	52	7044 (↑3.31×)	6938 (↑4.29×)	377 (↑5.16×)
MBV2-1.4-DS-F	67.56/69.04/70.13	415	10804 (↑5.08×)	7687 (↑4.75×)	344 (↑4.71×)

Table 3. Benchmark DepthShrinker with SOTA channel-wise pruning method AMC (He et al., 2018) on top of MobileNetV2@ImageNet in terms of FPS measured on three devices.

Model	FLOPS (M)	Acc (%)	Tesla V100	RTX 2080Ti	TX2
MBV2	330	72.30	3088	2364	115
AMC	220	70.80	3943	3159	152
MBV2-DS-A	287	72.43/72.48/72.50	5012	3505	177
MBV2-DS-B	272	71.54/72.06/72.09	5448	4074	199
MBV2-DS-C	261	70.90/71.21/71.56	6189	4691	226
MBV2-DS-D	253	69.40/70.15/70.58	6776	5257	258

EfficientNet (Tan & Le, 2019)) families, on top of the ImageNet dataset (Russakovsky et al., 2015).

Search settings. We adopt the same training hyper-parameters as the fine-tuning stage (see below), and find that the important activation functions can be quickly identified and the search becomes stable within 20 epochs.

Fine-tuning settings. By default, we fine-tune for 180 epochs with an SGD optimizer and a cosine learning rate, equipping with label smoothing (Müller et al., 2019) and RandAugment (Cubuk et al., 2020) following (Wang et al., 2021). **Unless explicitly specified, we do not enable self-distillation in experiments of the reported results.**

Devices and measurement settings. We consider three commonly used computing platforms, including NVIDIA Tesla V100 GPUs (NVIDIA., c), RTX 2080Ti GPUs (NVIDIA., b), and Jetson TX2 Edge GPUs (NVIDIA., a), and adopt the same measurement setting as in Sec. 3.2.

5.2. Benchmark with SOTA Pruning Methods

We first benchmark DepthShrinker with SOTA pruning techniques, including both channel- and layer-wise ones.

Benchmark with channel-wise pruning. We benchmark with two channel-wise pruning methods, AMC (He et al., 2018) and MetaPruning (Liu et al., 2019), achieving SOTA performance in compressing efficient DNNs, as well as a uniform channel-wise pruning baseline in (Liu et al., 2019), on ImageNet. As shown in Tabs. 2 and 3, we annotate our DepthShrinker's delivered model families with "DS-X" (detailed structures are in the Appendix. E), and report their accuracy under three training settings: standard training for 180 epochs, training with self-distillation in Sec. 4.3 for 180 and 360 epochs, respectively.

Results and analysis. We can observe that (1) under the standard training setting, DepthShrinker consistently achieves better accuracy-efficiency trade-offs over all the three baselines on all three devices. In particular, DepthShrinker achieves 1.40× throughput with a 0.18% higher accuracy OR a 1.45% higher accuracy with 1.14× throughput over MetaPruning-1.0×, and 1.48× throughput with a 0.1% higher accuracy over AMC on MobileNetV2 measured on a RTX 2080Ti GPU; (2) Equipping with self-distillation and more training epochs, DepthShrinker's achievable accuracy is notably boosted by up to 2.57%, further enlarging the accuracy gap with the channel-wise pruning baselines; (3) DepthShrinker shows decent scalability to different compression ratios and outperforms SOTA channel-wise pruning especially under extremely efficient cases, e.g., a 3.06% higher accuracy with 1.53× throughput over MetaPruning-0.35× on Tesla V100; and (4) DepthShrinker favors better data reuses and higher utilization, since the FLOPs of its delivered models are larger while their real-hardware efficiency is better compared with those of channel-wise pruning methods.

Benchmark with layer-wise pruning. To benchmark with layer-wise pruning methods (which prune a whole block in our case), we directly remove the entire blocks identified by our DepthShrinker's differentiable search scheme for a fair comparison, based on the hypothesis that the blocks with re-

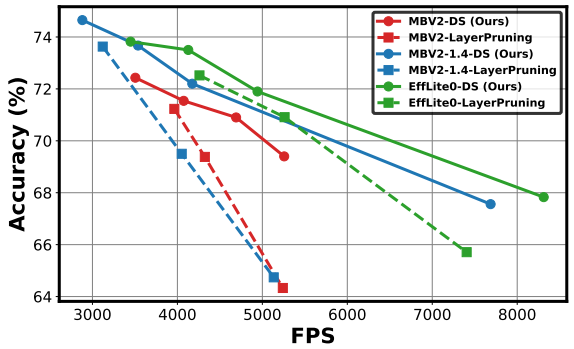

Figure 3. Benchmark DepthShrinker (*solid line*) with layer-wise pruning (*dashed line*) on top of three models in terms of FPS measured on an RTX 2080Ti GPU. "MBV2" and "Efflite0" denote MobileNetV2 and EfficientNet-Lite0, respectively.

dundant activation functions are also redundant themselves since their complexity contributes less to the final accuracy. As shown in Fig. 3, we can see that (1) DepthShrinker still consistently achieves better accuracy-efficiency trade-offs across all three models; (2) DepthShrinker notably outperforms layer-wise pruning under high compression ratios with more blocks pruned, since the latter suffers from a larger accuracy drop, e.g., DepthShrinker achieves a 2.80% higher accuracy with 1.50× throughput on MobileNetV2-1.4 over the smallest model from layer pruning. This indicates merging is better than hard pruning in scalability.

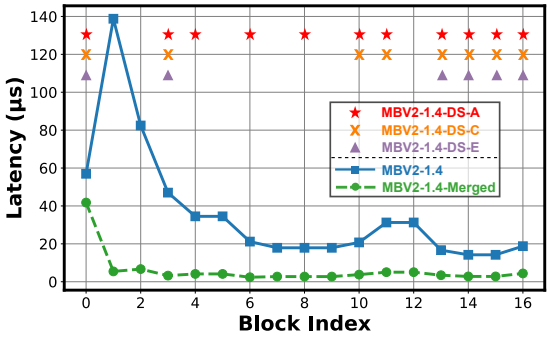

Figure 4. Visualizing the block-wise latency of the blocks in MobileNetV2-1.4 (*solid lines*) and their merged counterparts (*dashed lines*) on an RTX 2080Ti GPU. We also annotate blocks where the activation functions are remained, using different symbols for the three model variants delivered by DepthShrinker.

Visualization. We visualize the remained activation functions of DepthShrinker's delivered model variants, as well as the block-wise latency breakdown before and after merging each block on top of MobileNetV2-1.4 in Fig. 4. We can see that (1) shrinking the building blocks to dense convolutions can notably reduce the latency by up to 96.1%, and (2) DepthShrinker can successfully identify bottleneck layers in terms of latency, thanks to the latency-aware decay (see Sec. 4.2). Note that a merged dense convolution has the same number of input/output channels as the original block, which is different from the setting in Sec. 3.2 where the input/output channels are scaled to keep the same FLOPs.

We also visualize the block-wise memory footprint, including both that of weights and peak activation maps, i.e., the maximal sum of the input/output/residual activation maps when executing each convolution in a block, before and after applying DepthShrinker in Fig. 5 (assuming 16-bit precision). We can see that DepthShrinker effectively reduces the peak activation usage which mostly dominates the memory footprint, as it removes both the channel expansion and residual connections, leading to reduced data movement cost and thus boosted real-hardware efficiency.

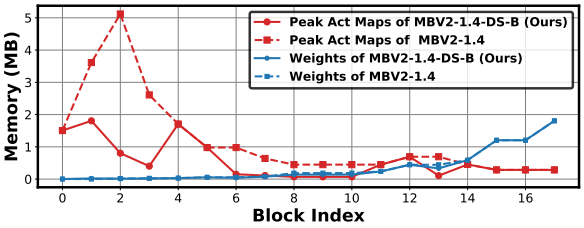

Figure 5. Visualizing the memory footprint, including both that of weights and peak activation maps, of MobileNetV2-1.4 before and after applying DepthShrinker.

Remark. Our DepthShrinker opens up a new compression paradigm which provides a cost-effective perspective for compressing efficient DNN structures, wining the advantages of both channel- and layer-/block-wise pruning, i.e., achieving the high accuracy of the former together with the decent hardware efficiency of the latter.

5.3. Benchmark with SOTA Efficient DNNs

We apply our DepthShrinker to MobileNetV2 with a channel scale of 1.4, and EfficientNet-Lite0 with self-distillation enabled to acquire a set of new model families made up of inverted residual blocks and dense convolution layers, which are compared with SOTA efficient DNN families (Howard et al., 2019; Lin et al., 2020; Wu et al., 2019; Lin et al., 2020; Tan et al., 2019).

Results and analysis. We show the accuracy and FPS trade-off of different models in Fig. 6. We can observe that (1) DepthShrinker's generated models push forward the frontier of the accuracy-efficiency trade-off over SOTA efficient DNNs, including the NAS-based ones, e.g., a 1.59% higher accuracy with 1.17× throughput over MobileNetV3 (Howard et al., 2019); (2) DepthShrinker scales better to high compression ratio scenarios, e.g., a 2.87% higher accuracy under comparable throughput (0.96×) compared with the smallest model in the MobileNetV3 family. This set of experiments indicates that shrinking manually designed models via DepthShrinker can match or even outperform advanced NAS-based models in terms of real-hardware efficiency. Note that the key idea of DepthShrinker can be combined with NAS methods to deliver more real-hardware efficient model families, which we leave as a future work.

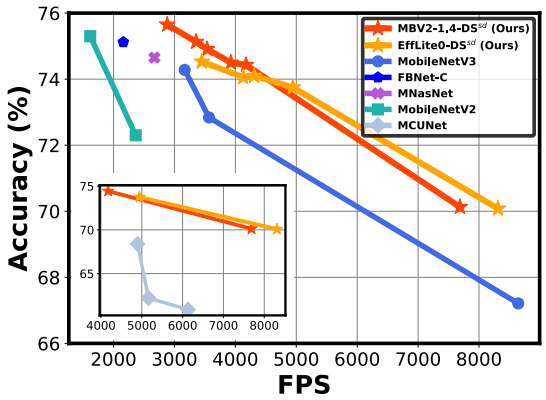

Figure 6. Benchmark DepthShrinker's delivered models with SOTA efficient DNNs in terms of FPS measured on an RTX 2080Ti GPU. "DS^{sd}" denotes that self-distillation is enabled. The zoom-in figure shows the comparison with MCUNet (Lin et al., 2020).

Table 4. Evaluating the effectiveness of starting from pretrained models and adding free activation functions.

Model	From Scratch	+ Pretrain	+ Free Act Func
MBV2-1.4-DS-A	73.82	74.40 (+0.58)	**74.65 (+0.83)**
MBV2-1.4-DS-C	72.45	72.87 (+0.42)	**73.67 (+1.22)**
MBV2-1.4-DS-E	70.12	71.39 (+1.27)	**72.20 (+2.08)**
MBV2-DS-A	71.51	72.39 (+0.88)	**72.43 (+0.92)**
MBV2-DS-B	70.50	71.29 (+0.79)	**71.54 (+1.04)**
MBV2-DS-C	69.92	70.51 (+0.59)	**70.90 (+0.98)**

5.4. Ablation study of DepthShrinker

Contributions of pretraining and adding free activation functions. As shown in Tab. 4, given the identified redundant activation functions, the target network is (1) trained from scratch, (2) fine-tuned from pretrained models, or (3) fine-tuned with one free activation function being added after each merged convolution (see Sec. 4.3). We observe that (1) pretraining improves the accuracy by 0.42%∼1.27%, which echoes the role of activation functions in DNN training in Sec. 3.1; and (2) Adding the free activation functions can further boost the accuracy by up to 0.81%.

Comparison with random search. We benchmark the proposed differentiable search method with a random search baseline which randomly keeps the same amount of activation functions as our searched ones. Compared with the random search counterparts (averaged over five runs), MBV2-1.4-DS-E and MBV2-DS-D in Tabs. 2 and 3 achieve a 4.30%/5.46% higher accuracy with 1.24×/1.19× throughput on the RTX 2080Ti GPU, respectively. This implies random search can hardly hit decent strategies and leads to both inferior accuracy and throughput. More ablation studies with the EfficientNet-Lite family are in the Appendix. A.

Measurement on CPU devices. In addition to GPUs and Edge GPUs considered by aforementioned experiments, we also measure the latency of DepthShrinker's delivered models on two CPU devices, including the Google Pixel 3 mobile phone and Raspberry Pi 4 (Raspi 4) with a batch size of 1, where all Pytorch models are converted to ONNX and then compiled to the TFLite format, following (Li et al., 2021). As shown in Tab. 5, we can see DepthShrinker still notably reduces the latency under comparable accuracy (according to Tab. 2 and 3), thanks to the reduced data movements with more data reuses of dense convolutions, indicating the general applicability of DepthShrinker across various commercial devices.

Table 5. Measure the latency of DepthShrinker's delivered models on two CPU devices, i.e., Google Pixel 3 and Raspi 4.

Model	Pixel 3 (s)	Raspi 4 (s)	Model	Pixel 3 (s)	Raspi 4 (s)
MBV2	0.073	0.200	MBV2-1.4	0.127	0.299
MBV2-DS-A	0.065 (↓10.9%)	0.147 (↓26.5%)	**MbV2-1.4-DS-A**	0.089 (↓29.9%)	0.204 (↓31.8%)
MBV2-DS-B	0.049 (↓32.9%)	0.133 (↓33.5%)	**MbV2-1.4-DS-B**	0.105 (↓17.3%)	0.196 (↓34.4%)
MBV2-DS-C	0.047 (↓35.6%)	0.124 (↓38.0%)	**MbV2-1.4-DS-C**	0.083 (↓34.6%)	0.185 (↓38.1%)
MBV2-DS-D	0.045 (↓38.4%)	0.116 (↓42.0%)	**MbV2-1.4-DS-E**	0.079 (↓37.8%)	0.170 (↓43.1%)

5.5. Evaluate DepthShrinker$^+$

We evaluate the proposed DepthShrinker$^+$, i.e., the Expand-Then-Shrink training strategy in Sec. 4.5, on top of VGG11/VGG13 (Simonyan & Zisserman, 2014)/MCUNet (Lin et al., 2020)/MobileNetV2 (Sandler et al., 2018) on ImageNet via replacing their intermediate blocks with inverted residual blocks, which are then merged using our DepthShrinker principle. More details about how to expand each network are in the Appendix. D.

Table 6. Evaluating DepthShrinker$^+$ on five models on ImageNet. "rXX" denotes the input resolution, following (Cai et al., 2021).

Model	VGG11	VGG13	MCUNet (r176)	MBV2-0.5 (r160)	MBV2 (r160)
Baseline (%)	71.51	71.64	61.50	61.40	69.60
DepthShrinker$^+$ (%)	**72.95**	**73.26**	**62.77**	**62.72**	**70.86**

Results. As shown in Tab. 6, DepthShrinker$^+$ consistently boosts the accuracy by 1.26%∼1.62% over standard training across all the five models. This indicates the potential of DepthShrinker$^+$ in aiding tiny network training.

6. Conclusion

To tackle the limitations of existing efficient DNNs in fulfilling their promise in boosting real-hardware efficiency due to their low hardware utilization, we open up a new compression paradigm and propose DepthShrinker to develop hardware efficient compact DNNs via merging irregular blocks into dense operations with much improved real-hardware efficiency. Extensive experiments validate our DepthShrinker wins both the high accuracy of channel-wise pruning and the decent efficiency of layer-wise pruning, opening up a cost-effective dimension for DNN compression.

Acknowledgements

The work performed by Yonggan Fu, Jiayi Yuan, Cheng Wan, and Yingyan Lin is supported by the National Science Foundation (NSF) through the SCH program (Award number: 1838873), the NeTS program (Award number: 1801865), and the MLWiNS program (Award number: 2003137).

References

Bengio, Y., Léonard, N., and Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Bi, K., Xie, L., Chen, X., Wei, L., and Tian, Q. Gold-nas: Gradual, one-level, differentiable. *arXiv preprint arXiv:2007.03331*, 2020.

Cai, H., Zhu, L., and Han, S. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

Cai, H., Gan, C., Lin, J., and Han, S. Network augmentation for tiny deep learning. *arXiv preprint arXiv:2110.08890*, 2021.

Chen, H., Wang, Y., Xu, C., Shi, B., Xu, C., Tian, Q., and Xu, C. Addernet: Do we really need multiplications in deep learning? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1468–1477, 2020.

Chen, S. and Zhao, Q. Shallowing deep networks: layer-wise pruning based on feature representations. *IEEE transactions on pattern analysis and machine intelligence*, 41(12):3048–3056, 2018.

Chen, Y.-H., Yang, T.-J., Emer, J., and Sze, V. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

Courbariaux, M., Bengio, Y., and David, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pp. 3123–3131, 2015.

Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

Cubuk, E. D., Zoph, B., Shlens, J., and Le, Q. V. Randaugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pp. 702–703, 2020.

Ding, X., Zhang, X., Ma, N., Han, J., Ding, G., and Sun, J. Repvgg: Making vgg-style convnets great again. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 13733–13742, 2021.

Dong, X., Chen, S., and Pan, S. Learning to prune deep neural networks via layer-wise optimal brain surgeon. *Advances in Neural Information Processing Systems*, 30, 2017.

Elkerdawy, S., Elhoushi, M., Singh, A., Zhang, H., and Ray, N. To filter prune, or to layer prune, that is the question. In *Proceedings of the Asian Conference on Computer Vision*, 2020.

Fu, Y., You, H., Zhao, Y., Wang, Y., Li, C., Gopalakrishnan, K., Wang, Z., and Lin, Y. Fractrain: Fractionally squeezing bit savings both temporally and spatially for efficient dnn training. *Advances in Neural Information Processing Systems*, 33:12127–12139, 2020.

Fu, Y., Guo, H., Li, M., Yang, X., Ding, Y., Chandra, V., and Lin, Y. Cpt: Efficient deep neural network training via cyclic precision. *arXiv preprint arXiv:2101.09868*, 2021a.

Fu, Y., Yu, Q., Zhang, Y., Wu, S., Ouyang, X., Cox, D., and Lin, Y. Drawing robust scratch tickets: Subnetworks with inborn robustness are found within randomly initialized networks. *Advances in Neural Information Processing Systems*, 34, 2021b.

Google. Efficientnet-lite. https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet/lite, 2020.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015a.

Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28: 1135–1143, 2015b.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

He, Y., Zhang, X., and Sun, J. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE international conference on computer vision*, pp. 1389–1397, 2017.

He, Y., Lin, J., Liu, Z., Wang, H., Li, L.-J., and Han, S. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 784–800, 2018.

He, Y., Liu, P., Wang, Z., Hu, Z., and Yang, Y. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proceedings of the IEEE/CVF*

Conference on Computer Vision and Pattern Recognition, pp. 4340–4349, 2019.

He, Y., Ding, Y., Liu, P., Zhu, L., Zhang, H., and Yang, Y. Learning filter pruning criteria for deep convolutional neural networks acceleration. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 2009–2018, 2020.

Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., et al. Searching for mobilenetv3. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1314–1324, 2019.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. URL http://arxiv.org/abs/1704.04861.

Hu, S., Xie, S., Zheng, H., Liu, C., Shi, J., Liu, X., and Lin, D. Dsnas: Direct neural architecture search without parameter retraining. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12084–12092, 2020.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2018.

Kang, M. and Han, B. Operation-aware soft channel pruning using differentiable masks. In *International Conference on Machine Learning*, pp. 5122–5131. PMLR, 2020.

Li, C., Yu, Z., Fu, Y., Zhang, Y., Zhao, Y., You, H., Yu, Q., Wang, Y., and Lin, Y. Hw-nas-bench: Hardware-aware neural architecture search benchmark. *arXiv preprint arXiv:2103.10584*, 2021.

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

Lin, J., Chen, W.-M., Lin, Y., Cohn, J., Gan, C., and Han, S. Mcunet: Tiny deep learning on iot devices. *arXiv preprint arXiv:2007.10319*, 2020.

Liu, H., Simonyan, K., and Yang, Y. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

Liu, Z., Mu, H., Zhang, X., Guo, Z., Yang, X., Cheng, K.-T., and Sun, J. Metapruning: Meta learning for automatic neural network channel pruning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 3296–3305, 2019.

Müller, R., Kornblith, S., and Hinton, G. When does label smoothing help? *arXiv preprint arXiv:1906.02629*, 2019.

Nakkiran, P., Alvarez, R., Prabhavalkar, R., and Parada, C. Compressing deep neural networks using a rank-constrained topology. 2015.

NVIDIA. NVIDIA Jetson TX2, a. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/, accessed 2020-09-01.

NVIDIA. NVIDIA RTX 2080Ti, b. https://www.nvidia.com/en-me/geforce/graphics-cards/rtx-2080-ti/.

NVIDIA. NVIDIA Tesla V100, c. https://www.nvidia.com/en-us/data-center/v100/.

Ramanujan, V., Wortsman, M., Kembhavi, A., Farhadi, A., and Rastegari, M. What's hidden in a randomly weighted neural network? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11893–11902, 2020.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pp. 525–542. Springer, 2016.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3): 211–252, 2015.

Sainath, T. N., Kingsbury, B., Sindhwani, V., Arisoy, E., and Ramabhadran, B. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 6655–6659. IEEE, 2013.

Sandler, M., Howard, A. G., Zhu, M., Zhmoginov, A., and Chen, L. Mobilenetv2: Inverted residuals and linear bottlenecks. mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018. URL http://arxiv.org/abs/1801.04381.

Shen, J., Wang, Y., Xu, P., Fu, Y., Wang, Z., and Lin, Y. Fractional skipping: Towards finer-grained dynamic cnn inference. In *AAAI*, pp. 5700–5708, 2020.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Tan, M. and Le, Q. V. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., and Le, Q. V. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2820–2828, 2019.

Teerapittayanon, S., McDanel, B., and Kung, H.-T. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pp. 2464–2469. IEEE, 2016.

Wang, D., Gong, C., Li, M., Liu, Q., and Chandra, V. Alphanet: Improved training of supernet with alpha-divergence. *arXiv preprint arXiv:2102.07954*, 2021.

Wang, X., Yu, F., Dou, Z.-Y., Darrell, T., and Gonzalez, J. E. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 409–424, 2018.

Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29:2074–2082, 2016.

Wu, B., Wan, A., Yue, X., Jin, P. H., Zhao, S., Golmant, N., Gholaminejad, A., Gonzalez, J., and Keutzer, K. Shift: A zero flop, zero parameter alternative to spatial convolutions. *CoRR*, abs/1711.08141, 2017. URL http://arxiv.org/abs/1711.08141.

Wu, B., Dai, X., Zhang, P., Wang, Y., Sun, F., Wu, Y., Tian, Y., Vajda, P., Jia, Y., and Keutzer, K. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 10734–10742, 2019.

Xie, S., Zheng, H., Liu, C., and Lin, L. Snas: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*, 2018.

Xu, P., Cao, J., Shang, F., Sun, W., and Li, P. Layer pruning via fusible residual convolutional block for deep neural networks. *arXiv preprint arXiv:2011.14356*, 2020.

Yin, M., Sui, Y., Liao, S., and Yuan, B. Towards efficient tensor decomposition-based dnn model compression with optimization framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10674–10683, 2021.

Zhou, D., Ye, M., Chen, C., Meng, T., Tan, M., Song, X., Le, Q., Liu, Q., and Schuurmans, D. Go wide, then narrow: Efficient training of deep thin networks. In *International Conference on Machine Learning*, pp. 11546–11555. PMLR, 2020.

Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H., and Zou, Y. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

Zhou, Y., Yen, G. G., and Yi, Z. Evolutionary shallowing deep neural networks at block levels. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

Zhu, C., Han, S., Mao, H., and Dally, W. J. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

A. Evaluate DepthShrinker on EfficientNet-Lite Families

Setup. We apply DepthShrinker on top of EfficientNet-Lite3 (Google., 2020) on ImageNet, equipped with the self-distillation mechanism mentioned in Sec. 4.3, to generate a new model family annotated as "EffLite3-DS-X" in Tab. 7. For a fair comparison, we adopt the same training schedule as introduced in Sec. 5.1 to train the EfficientNet-Lite baselines from scratch.

Results and analysis. From Tab. 7, we can observe that DepthShrinker's delivered models again push forward the achievable accuracy-efficiency trade-off. In particular, EffLite3-DS-B achieves a 1.32% higher accuracy with 1.21× throughput on Tesla V100 over EfficientNet-Lite2 and EffLite3-DS-A achieves a 1.38% higher accuracy with comparable throughput (e.g., 0.95× on Tesla V100). As mentioned in the main text, our DepthShrinker can be combined with NAS methods to deliver new model families featuring much improved real-hardware efficiency and we leave this as our future work.

Table 7. Evaluating DepthShrinker on top of EfficientNet-Lite3 on ImageNet. "Efflite" denotes EfficientNet-Lite.

Model	Acc (%)	MFLOPs	Throughput		
			Tesla V100	RTX 2080Ti	TX2
EffLite1	75.41	651	1773	1403	64
EffLite2	76.14	924	1301	1081	44
EffLite3-DS-A	76.79	991	1676	1337	53
EffLite3-DS-B	77.46	952	1573	1264	50
EffLite3-DS-C	77.63	918	1431	1152	46
EffLite3-DS-D	77.84	905	1250	1030	41

B. More Benchmark with Layer-wise Pruning

We also benchmark with LayerPrune (Elkerdawy et al., 2020) for compressing three efficient models based on their provided implementation, as a complement of Sec.5.2 in the main text. As shown in Fig. 7, DepthShrinker still consistently outperforms LayerPrune, especially under large compression ratios.

C. More Real-device Profiling Results

We benchmark the efficiency of SOTA DNN families under different depth/width trade-offs under the same FLOPs for better understanding the causes of the improved hardware utilization in Sec. 3.2.

Setup. To construct models featuring different depth/width trade-offs with the same FLOPs, we uniformly scale the channel number of the networks within the same model family and benchmark their throughput on different devices.

Results and analysis. As shown in Tab. 8, we can observe that (1) shallower networks consistently win better throughput compared with the deeper counterparts under the same FLOPs across different model families and devices, e.g., channel-scaled ResNet-50 achieves 1.30× over channel-scaled ResNet-152. This indicates the preference for shallow-wide networks of the mapping strategies of existing commercial devices; (2) shallower networks reduce the latency by up to 58.9% on TX2 Edge GPU (i.e., the inverse of the throughput measured with a batch size of one), which is another perspective for measuring the real-time processing capability on edge devices; (3) Based on the comparison between Tab. 8 and Tab. 1, dense operations win real-hardware efficiency thanks to both of the two aforementioned aspects and reducing operation-wise irregularity may contribute more, especially on more powerful devices with a higher degree of parallelism.

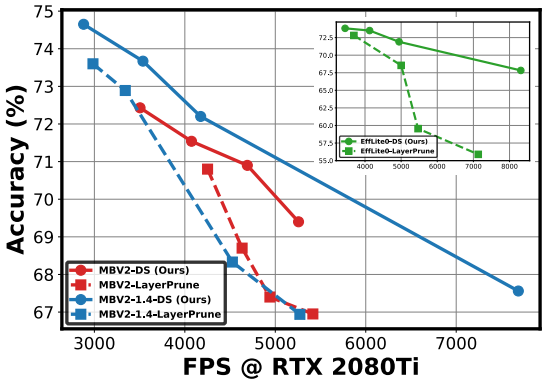

Figure 7. Benchmark DepthShrinker (*solid line*) with LayerPrune (Elkerdawy et al., 2020) (*dashed line*) on top of three models in terms of FPS measured on an RTX 2080Ti GPU.

D. Design Details of DepthShrinker$^+$

In Sec. 5.5, we evaluate the proposed DepthShrinker$^+$, on top of VGG11/VGG13 (Simonyan & Zisserman, 2014)/MCUNet (Lin et al., 2020)/MobileNetV2 (Sandler et al., 2018) on ImageNet. In particular, for the last two models, we follow the definition in (Cai et al., 2021). Without bells and whistles, we design empirical rules to determine which layer to be expanded for all the networks to demonstrate the general effectiveness of our DepthShrinker technique as a training technique for boosting accuracy.

Expand VGG. For all VGG networks, we expand all the 3×3 convolution layers to a standard inverted residual block (Sandler et al., 2018) with an expansion ratio of 6, except the first two convolution layers and the last convolution layer.

Expand MCUNet/MobileNetV2. For all networks made up of inverted residual blocks, we apply DepthShrinker$^+$ to expand one block in every two consecutive blocks. In addition, to expand one specific inverted residual block, we only expand the first pointwise convolution to a new inverted residual block (Sandler et al., 2018) with an expansion ratio

Table 8. Measured throughput of the ResNet family and the MobileNetV2 family with scaled channel numbers to maintain the same FLOPs. All the reported numbers are FPS. The number of blocks in each stage of MobileNetV2 is annotated in the Depth column.

Model	Depth	Width Scale	Throughput (FPS)		
			RTX 2080Ti	TX2 (bs=32)	TX2 (bs=1)
ResNet-18	18	1.535	1476	85	28
ResNet-34	34	1.07	1388	61	27
ResNet-50	50	1.00	874	46	20
ResNet-101	101	0.73	792	36	15
ResNet-152	152	0.60	674	31	11
MobileNetV2	[1,1,1,1,1,1,1]	1.45	2499	112	56
	[1,2,2,2,1,1,1]	1.25	2167	98	52
	[1,2,3,3,2,2,1]	1.11	2114	103	39
	[1,2,3,4,3,3,1]	1.00	2149	103	33
	[1,3,4,6,5,5,1]	0.85	1916	96	23

of 6 and a depthwise kernel size of 1 to ensure the original model structure can be recovered.

Integrating with more advanced expansion strategies, our DepthShrinker can potentially achieve more notable improvements, which will be our future work.

Table 9. Visualizing the remained activation functions in DepthShrinker's generated model families.

Model	Remained Activation Functions
MBV2-1.4-DS-A	[1 0 0 1 1 0 1 0 1 0 1 1 0 1 1 1 1]
MBV2-1.4-DS-B	[0 0 0 1 1 0 0 0 0 0 1 1 0 1 1 1 1]
MBV2-1.4-DS-C	[1 0 0 1 0 0 0 0 0 0 1 1 0 1 1 1 1]
MBV2-1.4-DS-D	[0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 1]
MBV2-1.4-DS-E	[1 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1]
MBV2-1.4-DS-F	[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
MBV2-DS-A	[0 0 1 0 1 1 1 0 0 1 1 1 1 1 1 1 1]
MBV2-DS-B	[1 0 0 0 1 1 1 0 0 1 1 1 1 1 1 1 1]
MBV2-DS-C	[1 0 0 1 1 0 1 0 0 0 1 0 0 1 1 1 1]
MBV2-DS-D	[1 0 0 1 0 0 0 0 0 0 1 1 0 1 0 1 1]
Eff-Lite0-A	[0 0 1 1 0 1 0 0 1 0 0 1 1 1 1 1]
Eff-Lite0-B	[0 0 0 1 1 0 0 0 1 0 0 1 1 1 1 1]
Eff-Lite0-C	[0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1]
Eff-Lite0-D	[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
EffLite3-DS-A	[0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 1]
EffLite3-DS-B	[0 0 0 1 1 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 1 1 1]
EffLite3-DS-C	[0 0 0 1 1 0 0 0 0 0 1 0 1 1 1 0 0 1 1 1 1 1 1]
EffLite3-DS-D	[1 0 0 1 1 0 1 1 0 0 0 0 1 0 0 1 1 1 1 1 1 1 1]

E. Details about DepthShrinker's Delivered Model Families

We apply DepthShrinker on top of the given efficient DNNs to generate new model families via varying the number of remained activation functions k in Eq. 1 and the decay strength on m discussed in Sec. 4.2, which constrains the overall efficiency of the delivered network. For all the reported models in the main text, we provide their remained activation functions identified by our DepthShrinker in Tab. 9, where each element in the list indicates whether the activation functions of the corresponding block are kept, i.e., "1" denotes the activation functions in the block are remained.