i-TAP
NN Compression

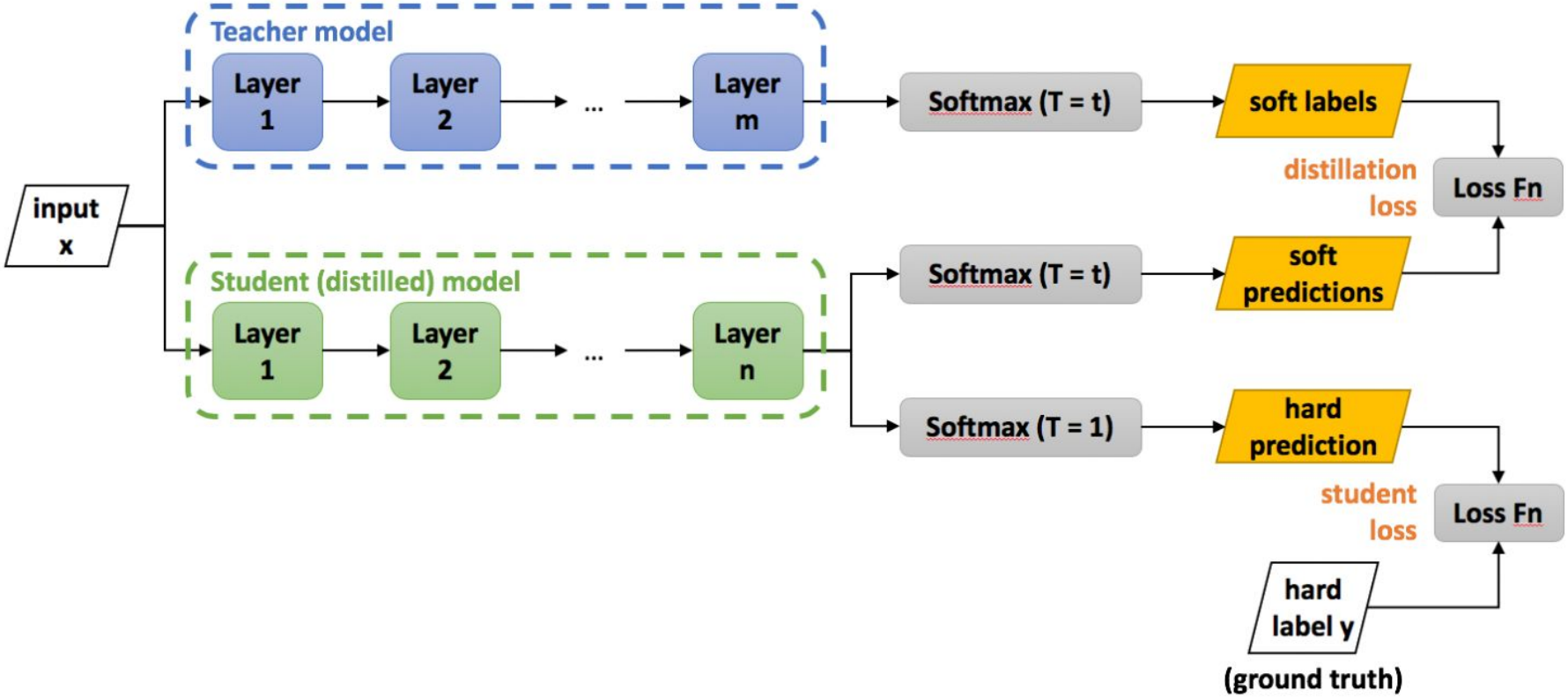
홍익대학교
노승문
2021. 4. 22.

Distilling the Knowledge in a Neural Network

Geoffrey Hinton[*†]
Google Inc.
Mountain View
geoffhinton@google.com

Oriol Vinyals[†]
Google Inc.
Mountain View
vinyals@google.com

Jeff Dean
Google Inc.
Mountain View
jeff@google.com



https://intellabs.github.io/distiller/knowledge_distillation.html

Neural networks typically produce class probabilities by using a "softmax" output layer that converts the logit, z_i, computed for each class into a probability, q_i, by comparing z_i with the other logits.

$$q_i = \frac{exp(z_i/T)}{\sum_j exp(z_j/T)} \tag{1}$$

where T is a temperature that is normally set to 1. Using a higher value for T produces a softer probability distribution over classes.

In the simplest form of distillation, knowledge is transferred to the distilled model by training it on a transfer set and using a soft target distribution for each case in the transfer set that is produced by using the cumbersome model with a high temperature in its softmax. The same high temperature is used when training the distilled model, but after it has been trained it uses a temperature of 1.

2.1 Matching logits is a special case of distillation

Each case in the transfer set contributes a cross-entropy gradient, dC/dz_i, with respect to each logit, z_i of the distilled model. If the cumbersome model has logits v_i which produce soft target probabilities p_i and the transfer training is done at a temperature of T, this gradient is given by:

$$\frac{\partial C}{\partial z_i} = \frac{1}{T}(q_i - p_i) = \frac{1}{T}\left(\frac{e^{z_i/T}}{\sum_j e^{z_j/T}} - \frac{e^{v_i/T}}{\sum_j e^{v_j/T}}\right) \tag{2}$$

If the temperature is high compared with the magnitude of the logits, we can approximate:

$$\frac{\partial C}{\partial z_i} \approx \frac{1}{T}\left(\frac{1 + z_i/T}{N + \sum_j z_j/T} - \frac{1 + v_i/T}{N + \sum_j v_j/T}\right) \tag{3}$$

If we now assume that the logits have been zero-meaned separately for each transfer case so that $\sum_j z_j = \sum_j v_j = 0$ Eq. 3 simplifies to:

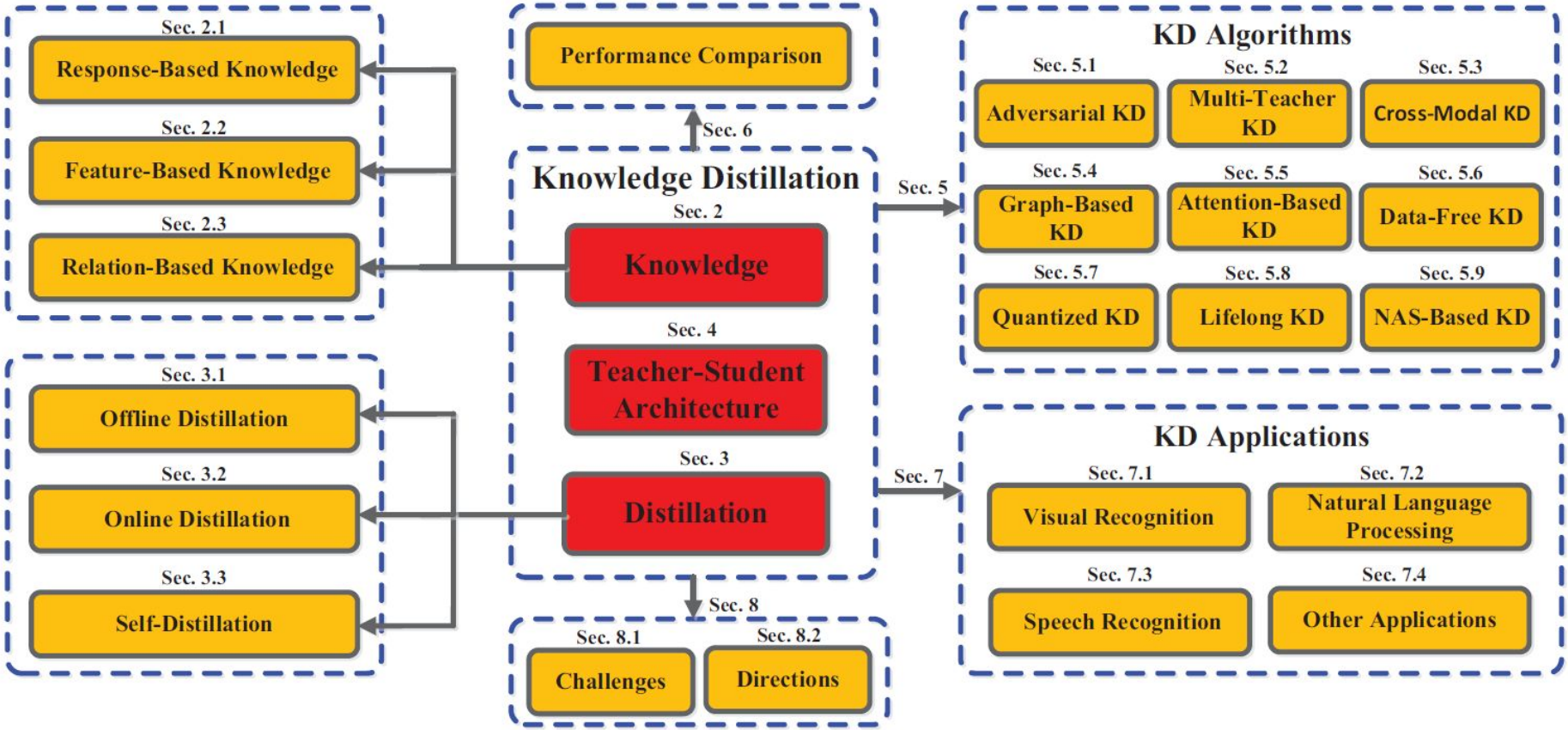
$$\frac{\partial C}{\partial z_i} \approx \frac{1}{NT^2}(z_i - v_i) \tag{4}$$

So in the high temperature limit, distillation is equivalent to minimizing $1/2(z_i - v_i)^2$, provided the logits are zero-meaned separately for each transfer case. At lower temperatures, distillation pays much less attention to matching logits that are much more negative than the average. This is potentially advantageous because these logits are almost completely unconstrained by the cost function used for training the cumbersome model so they could be very noisy. On the other hand, the very negative logits may convey useful information about the knowledge acquired by the cumbersome model. Which of these effects dominates is an empirical question. We show that when the distilled model is much too small to capture all of the knowledege in the cumbersome model, intermediate temperatures work best which strongly suggests that ignoring the large negative logits can be helpful.

Noname manuscript No.
(will be inserted by the editor)

Knowledge Distillation: A Survey

Jianping Gou[1] · **Baosheng Yu** [1] · **Stephen J. Maybank** [2] · **Dacheng Tao**[1]


Sec. 2.1
Response-Based Knowledge
Sec. 2.2
Feature-Based Knowledge
Sec. 2.3
Relation-Based Knowledge
Sec. 3.1
Offline Distillation
Sec. 3.2
Online Distillation
Sec. 3.3
Self-Distillation
Performance Comparison
Sec. 6
Knowledge Distillation
Sec. 2
Knowledge
Sec. 4
Teacher-Student Architecture
Sec. 3
Distillation
Sec. 8
Sec. 8.1
Challenges
Sec. 8.2
Directions
Sec. 5
Sec. 7
KD Algorithms
Sec. 5.1
Adversarial KD
Sec. 5.2
Multi-Teacher KD
Sec. 5.3
Cross-Modal KD
Sec. 5.4
Graph-Based KD
Sec. 5.5
Attention-Based KD
Sec. 5.6
Data-Free KD
Sec. 5.7
Quantized KD
Sec. 5.8
Lifelong KD
Sec. 5.9
NAS-Based KD
KD Applications
Sec. 7.1
Visual Recognition
Sec. 7.2
Natural Language Processing
Sec. 7.3
Speech Recognition
Sec. 7.4
Other Applications

MODEL COMPRESSION VIA DISTILLATION AND QUANTIZATION

Antonio Polino
ETH Zürich
antonio.polino1@gmail.com

Razvan Pascanu
Google DeepMind
razp@google.com

Dan Alistarh
IST Austria
dan.alistarh@ist.ac.at

ABSTRACT

Deep neural networks (DNNs) continue to make significant advances, solving tasks from image classification to translation or reinforcement learning. One aspect of the field receiving considerable attention is efficiently executing deep models in resource-constrained environments, such as mobile or embedded devices. This paper focuses on this problem, and proposes two new compression methods, which jointly leverage *weight quantization* and *distillation* of larger networks, called "teachers," into compressed "student" networks. The first method we propose is called *quantized distillation* and leverages distillation during the training process, by incorporating distillation loss, expressed with respect to the teacher network, into the training of a smaller student network whose weights are quantized to a limited set of levels. The second method, *differentiable quantization*, optimizes the *location of quantization points* through stochastic gradient descent, to better fit the behavior of the teacher model. We validate both methods through experiments on convolutional and recurrent architectures. We show that quantized shallow students can reach similar accuracy levels to state-of-the-art full-precision teacher models, while providing up to order of magnitude compression, and inference speedup that is almost linear in the depth reduction. In sum, our results enable DNNs for resource-constrained environments to leverage architecture and accuracy advances developed on more powerful devices.

3 QUANTIZED DISTILLATION

The context is the following: given a task, we consider a trained state-of-the-art deep model solving it–the *teacher*, and a compressed *student* model. The student is compressed in the sense that 1) it is *shallower* than the teacher; and 2) it is *quantized*, in the sense that its weights are expressed at limited bit width. The strategy, as for standard distillation (Ba & Caruana, 2013; Hinton et al., 2015) is for the student to leverage the converged teacher model to reach similar accuracy. We note that distillation has been used previously to obtain compact high-accuracy encodings of ensembles (Hinton et al., 2015); however, we believe this is the first time it is used for model compression via quantization.

The second question is how to employ distillation loss in the context of a *quantized* neural network. An intuitive approach is to rely on projected gradient descent, where a gradient step is taken as in full-precision training, and then the new parameters are *projected* to the set of valid solutions. Critically, we accumulate the error at each projection step into the gradient for the next step. One can think of this process as if *collecting evidence for whether each weight needs to move to the next quantization point or not*. Crucially, the error accumulation prevents the algorithm from getting stuck in the current solution if gradients are small, which would occur in a naive projected gradient approach. This is similar to the approach taken by BinaryConnect technique, with some differences. Li et al. (2017) also examines these dynamics in detail. Compared to BinnaryConnect, we use distillation rather than learning from scratch, hence learning more efficiently. We also do not restrict ourselves to binary representation, but rather use *variable bit-width* quantization functions and *bucketing*, as defined in Section 2.

An alternative view of this process, illustrated in Figure 1, is that we perform the SGD step on the *full-precision* model, but computing the gradient on the *quantized model*, expressed with respect to the *distillation loss*. See Algorithm 1 for details.

Algorithm 1 Quantized Distillation

procedure QUANTIZED DISTILLATION
    Let w *be the network weights*
loop
    $w^q \leftarrow \text{quant\_function}(w, s)$
    Run forward pass and compute distillation loss $l(w^q)$
    Run backward pass and compute $\frac{\partial l(w^q)}{\partial w^q}$
    Update original weights using SGD ***in full precision*** $w = w - \nu \cdot \frac{\partial l(w^q)}{\partial w^q}$
Finally quantize the weights before returning: $w^q \leftarrow \text{quant\_function}(w, s)$
return w^q

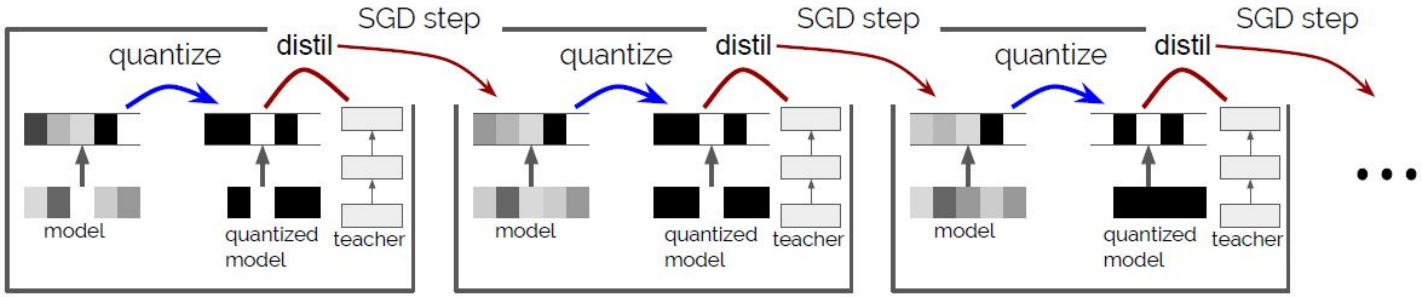


Figure 1: Depiction of the steps of quantized distillation. Note the accumulation over multiple steps of gradients in the unquantized model leads to a switch in quantization (e.g. top layer left most square).

4 DIFFERENTIABLE QUANTIZATION

A major problem in quantizing neural networks is the fact that the decision of which p_i should replace a given weight is discrete, hence the gradient is zero: $\frac{\partial Q(v,p)}{\partial v} = 0,$ almost everywhere. This implies that we cannot backpropagate the gradients through the quantization function. To solve this problem, typically a variant of the straight-through estimator is used, see e.g. Bengio et al. (2013); Hubara et al. (2016). On the other hand, the model as a function of the chosen p_i is *continuous* and *can be differentiated*; the gradient of $Q(v,p)_i$ with respect to p_j is well defined almost everywhere, and it is simply

$$\frac{\partial Q(v,p)_i}{\partial p_j} = \begin{cases} \alpha_i, & \text{if } v_i \text{ has been quantized to } p_j \\ 0, & \text{otherwise,} \end{cases} \tag{6}$$

where α_i is i-th element of the scaling factor, assuming we are using a bucketing scheme. If no bucketing is used, then $\alpha_i = \alpha$ for every i. Otherwise it changes depending on which bucket the weight v_i belongs to.

Algorithm 2 Differentiable Quantization

procedure DIFFERENTIABLE QUANTIZATION
    Let w *be the networks weights and* p *the initial quantization points*
loop
    $w^q \leftarrow$ quant_function(w, p)
    Run forward pass and compute loss $l(w^q)$
    Run backward pass and compute $\frac{\partial l(w^q)}{\partial w^q}$
    Use equation 6 *to compute* $\frac{\partial l(w^q)}{\partial p}$
    Update quantization points using SGD or similar: $p = p - \nu \cdot \frac{\partial l(w^q)}{\partial p}$
return p

Residual Distillation: Towards Portable Deep Neural Networks without Shortcuts

Guilin Li[1*†], **Junlei Zhang**[1*], **Yunhe Wang** [1], **Chuanjian Liu**[1],
Matthias Tan[2], **Yunfeng Lin**[1], **Wei Zhang**[1], **Jiashi Feng**[3], **Tong Zhang**[4]
[1]Noah's Ark Lab, Huawei Technologies [2]CityU, [3]NUS, [4]HKUST
hiliguilin@gmail.com, zjlnbnb@163.com
yunhe.wang@huawei.com, tongzhang@tongzhang-ml.org

NIPS 2020

Abstract

By transferring both features and gradients between different layers, shortcut connections explored by ResNets allow us to effectively train very deep neural networks up to hundreds of layers. However, the additional computation costs induced by those shortcuts are often overlooked. For example, during online inference, the shortcuts in ResNet-50 account for about 40 percent of the entire memory usage on feature maps, because the features in the preceding layers cannot be released until the subsequent calculation is completed. In this work, for the first time, we consider training the CNN models with shortcuts and deploying them without. In particular, we propose a novel joint-training framework to train plain CNN by leveraging the gradients of the ResNet counterpart. During forward step, the feature maps of the early stages of plain CNN are passed through later stages of both itself and the ResNet counterpart to calculate the loss. During backpropagation, gradients calculated from a mixture of these two parts are used to update the plainCNN network to solve the gradient vanishing problem. Extensive experiments on ImageNet/CIFAR10/CIFAR100 demonstrate that the plainCNN network without shortcuts generated by our approach can achieve the same level of accuracy as that of the ResNet baseline while achieving about $1.4\times$ speed-up and $1.25\times$ memory reduction. We also verified the feature transferability of our ImageNet pretrained plain-CNN network by fine-tuning it on MIT 67 and Caltech 101. Our results show that the performance of the plain-CNN is slightly higher than that of its baseline ResNet-50 on these two datasets. The code will be available at https://github.com/leoozy/JointRD_Neurips2020 and the MindSpore code will be available at https://www.mindspore.cn/resources/hub.

Table 1: Max memory (KB) on the mobile NPU

Input size	256	384	512
ResNet50	438737	832733	OOM
plain-CNN 50	356657	669397	1106989
Reduction	18.7%	19.6%	-

Table 2: Latency (ms) of 50-layer plain-CNN vs ResetNet50 evaluated on the mobile NPU

input size	224	1024	1536	2048
ResNet50	22.45 ± 0.34	346.96 ± 0.64	756.03 ± 1.94	out of memory
plain-CNN 50	18.03 ± 0.36	247.65 ± 1.52	519.01 ± 3.26	944.59 ± 4.27
Speedup	19.69%	28.62 %	31.35 %	-

In this paper, we propose to solve the problem mentioned above using the teacher-student paradigm. Different from the current teacher-student framework, our method, for the first time, propose to pass the gradients calculated from the teacher network to the student network. This can also be seen as training the network by adding some auxiliary architecture to assist the convergence. While during inference, the auxiliary part is abandoned. Specially, we develop a Joint-training framework based on Residual Distillation (JointRD), as shown in Figure 1. In practice, the original ResNets are selected as teacher networks, and the students are generated by directly removing shortcuts from the teachers. Then, each of the stages in the plain student network is connected with both later stages of the teacher and student networks. During the back-propagation, gradients are calculated and integrated from both of these two parts. In the early training iterations, gradients from the teacher network play a larger role, which will be gradually reduced until the convergence. By exploiting the proposed joint-training framework, we can effectively integrate the benefit of shortcuts into the training of plain student networks and obtain excellent portable networks without shortcuts.

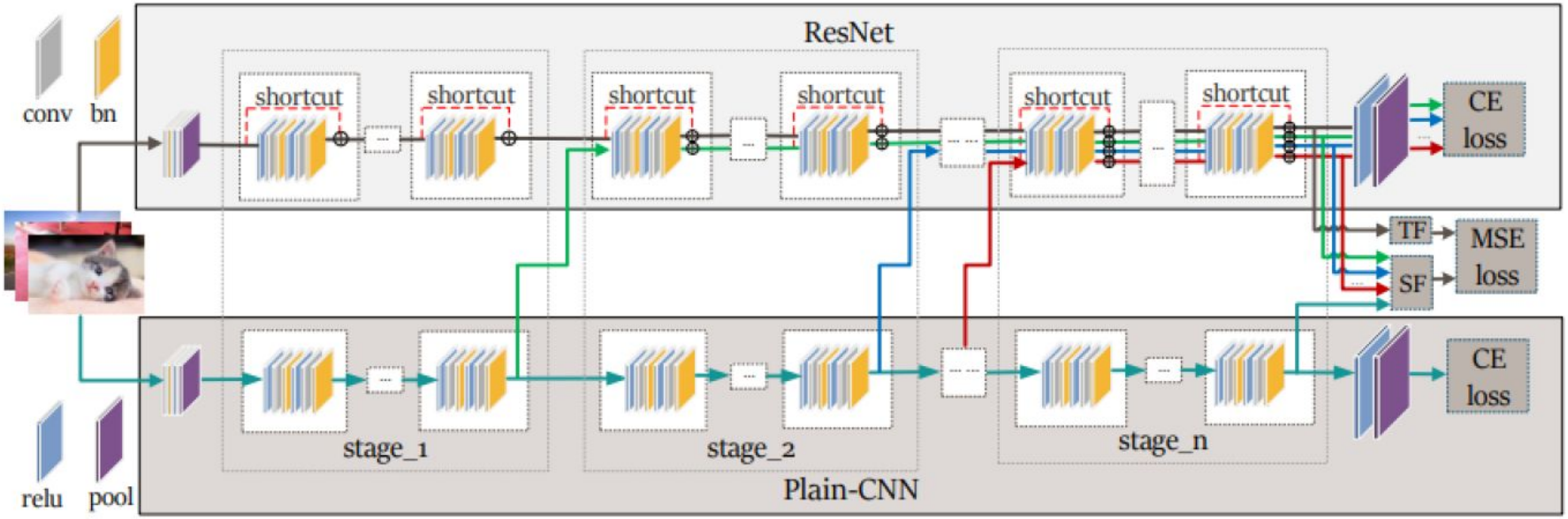


Figure 1: Joint-training framework: early stages of plain-CNN is connected to later stages of ResNet. CE loss represents for the cross-entropy loss, TF represents for teacher feature map and SF represents for students feature maps.

During the forward step, there would be four forward paths passing through student network s, teacher network t, and the mixture of them, network st. Define W and V as the parameters for the student and teacher network respectively, we have the four paths as:

1. **Path 1:** The input is passed through the first stage of plain-CNN and fed into the second stage of ResNet (the green arrow in Figure 1), following by the third and fourth stages of ResNet, to obtain feature map $f\_st^1(W;V)$ and cross-entropy loss $L\_st^1(W)$.
2. **Path 2:** The input is passed through the first and second stage of plain-CNN and feed into the third stage of ResNet (the blue arrow in Figure 1), following by the fourth stage of ResNet, to obtain feature map $f\_st^2(W;V)$ and cross-entropy loss $L\_st^2(W)$.
3. **Path 3:** The input is passed through the first, second and third stage of plain-CNN and feed into the fourth stage of ResNet (the red arrow in Figure 1), to obtain feature map $f\_st^3(W;V)$ and cross-entropy loss $L\_st^3(W)$.
4. **Path 4:** The input is passed through the whole plain-CNN network (network s) until the last stage to obtain feature map $f\_st^4(W)$ and cross entropy loss $L\_s(W)$.

Here, $f\_st(\cdot)$ represents for the student deep nested functions up to the last convolution layer of the last stage, right before they are passed to the relu and pooling layer. Note that although we also adopt regular KD [31, 32] where intermediate features $f\_t(\cdot)$ of teacher models are used to guide the student model in this framework, a key difference is that we also pass the gradients of the teacher to the student.

Final Layer KD Although using the KD loss alone would end up with very poor performance on some datasets such as CIFAR100, we found that including it in the joint-training framework would stabilize the training process and improve the final performance of the plain-CNN model. Especially, we minimize the mean square error (MSE) between the final layer output of the teacher model $f_t(\cdot)$ and that of the student models $f^i_{st}(\cdot), i = 1, \cdots, N$, with the feature maps from the student models transformed by a fully-connected regressor :

$$\sum_{i=1}^{N} ||f_t(V), r(f^i_{st}(W; V), w_r)||^2 \tag{2}$$

where r is a fully-connected regressor with parameter w_r

Table 3: Benchmark results on CIFAR-10/CIFAR-100. Plain-CNN: all shortcuts removed

Dataset	Model	JointRD (ours)	Naive	KD (MSE) + Dirac	ResNet
CIFAR-100	plain-CNN 18	78.24 ± 0.04	77.44 ± 0.10	$\mathbf{78.39 \pm 0.10}$	77.92 ± 0.26
	plain-CNN 34	$\mathbf{78.47 \pm 0.22}$	72.30 ± 2.62	77.86 ± 0.73	78.58 ± 0.21
	plain-CNN 50	$\mathbf{78.16 \pm 0.20}$	55.39 ± 4.29	50.38 ± 7.11	78.39 ± 0.40
CIFAR-10	plain-CNN 18	$\mathbf{95.11 \pm 0.08}$	94.81 ± 0.06	94.72 ± 0.11	95.19 ± 0.04
	plain-CNN 34	$\mathbf{94.78 \pm 0.25}$	93.73 ± 0.07	94.50 ± 0.10	95.39 ± 0.16
	plain-CNN 50	$\mathbf{94.40 \pm 0.08}$	91.13 ± 0.36	92.27 ± 0.27	95.31 ± 0.08

Table 4: Benchmark results on CIFAR-10/CIFAR-100. Plain-CNN*: removed all shortcuts together with pointwise convolutions of downsampling layers

Dataset	Model	JointRD (ours)	Naive	KD (MSE) + Dirac	ResNet
CIFAR-100	plain-CNN 18*	$\mathbf{77.91 \pm 0.21}$	76.67 ± 0.01	77.81 ± 0.26	77.92 ± 0.26
	plain-CNN 34*	$\mathbf{78.42 \pm 0.54}$	72.72 ± 0.41	78.41 ± 0.12	78.58 ± 0.21
	plain-CNN 50*	$\mathbf{77.68 \pm 0.58}$	54.53 ± 5.57	59.79 ± 7.68	78.39 ± 0.40
CIFAR-10	plain-CNN 18*	$\mathbf{95.03 \pm 0.03}$	94.78 ± 0.13	94.67 ± 0.13	95.19 ± 0.04
	plain-CNN 34*	$\mathbf{94.62 \pm 0.16}$	93.78 ± 0.13	$\mathbf{94.62 \pm 0.03}$	95.39 ± 0.16
	plain-CNN 50*	$\mathbf{94.36 \pm 0.36}$	90.59 ± 0.59	93.11 ± 0.25	95.31 ± 0.08

Questions?

Better Network Design

- Not pruning
- Not distilling

Pruning with initialization

This also shows that there might be a better `light' structure inherently

We may not have to train huge network then prune

MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications

Andrew G. Howard Menglong Zhu Bo Chen Dmitry Kalenichenko
Weijun Wang Tobias Weyand Marco Andreetto Hartwig Adam

Google Inc.

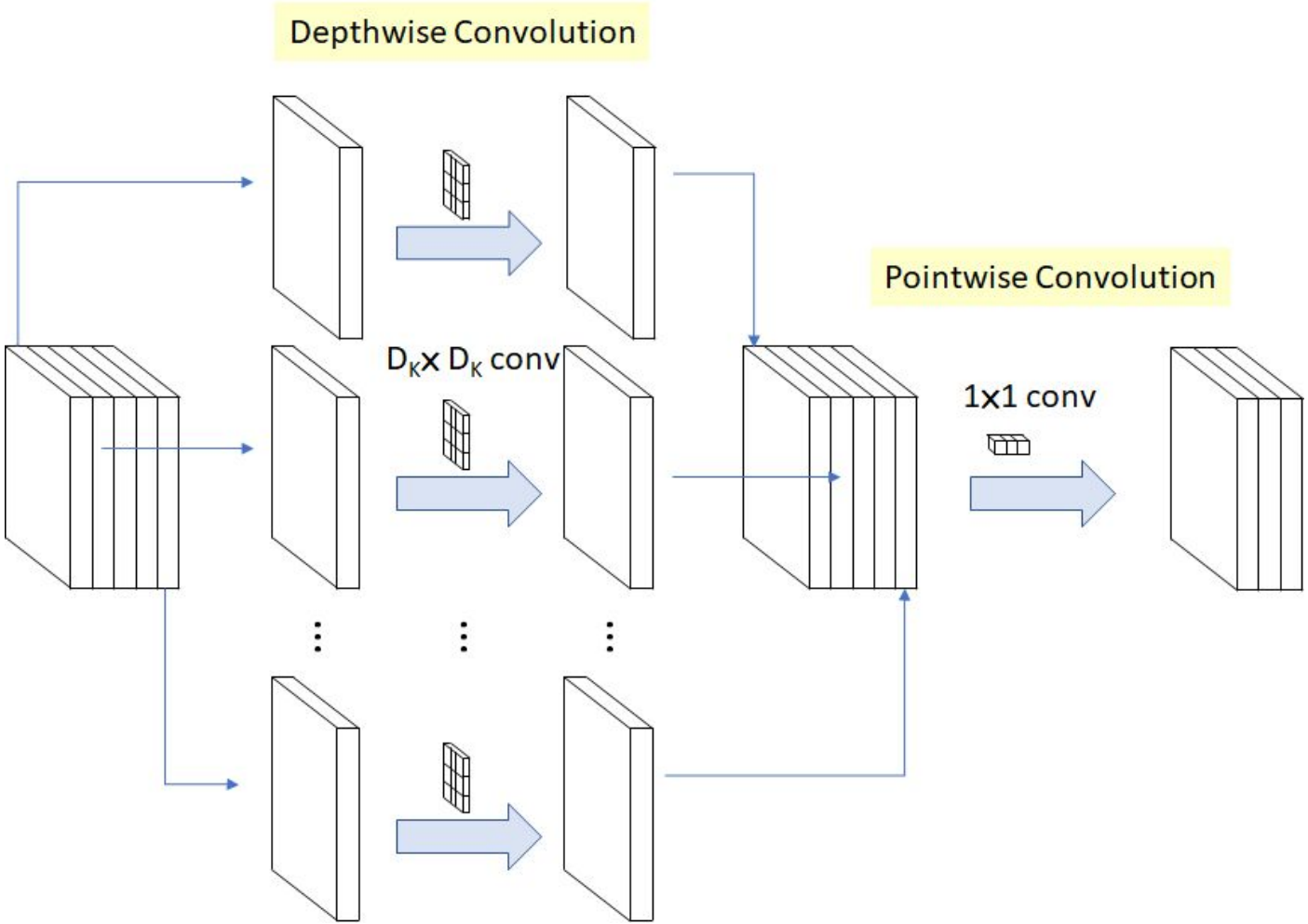
{howarda,menglong,bochen,dkalenichenko,weijunw,weyand,anm,hadam}@google.com

There has been rising interest in building small and efficient neural networks in the recent literature, e.g. [16, 34, 12, 36, 22]. Many different approaches can be generally categorized into either compressing pretrained networks or training small networks directly. This paper proposes a class of network architectures that allows a model developer to specifically choose a small network that matches the resource restrictions (latency, size) for their application. MobileNets primarily focus on optimizing for latency but also yield small networks. Many papers on small networks focus only on size but do not consider speed.

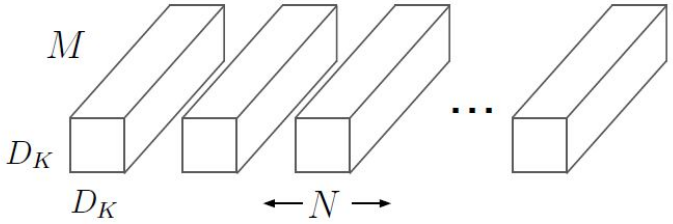
MobileNets are built primarily from depthwise separable convolutions initially introduced in [26] and subsequently used in Inception models [13] to reduce the computation in the first few layers. Flattened networks [16] build a network

3.1. Depthwise Separable Convolution

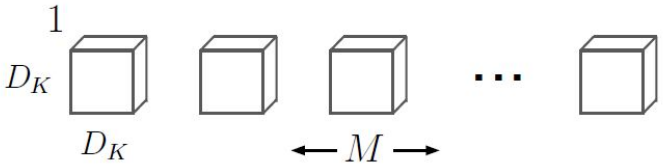
The MobileNet model is based on depthwise separable convolutions which is a form of factorized convolutions which factorize a standard convolution into a depthwise convolution and a 1×1 convolution called a pointwise convolution. For MobileNets the depthwise convolution applies a single filter to each input channel. The pointwise convolution then applies a 1×1 convolution to combine the outputs the depthwise convolution. A standard convolution both filters and combines inputs into a new set of outputs in one step. The depthwise separable convolution splits this into two layers, a separate layer for filtering and a separate layer for combining. This factorization has the effect of drastically reducing computation and model size. Figure 2 shows how a standard convolution 2(a) is factorized into a depthwise convolution 2(b) and a 1×1 pointwise convolution 2(c).



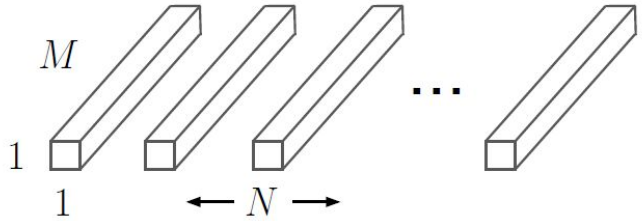
https://medium.com/dataseries/the-evolution-of-googles-mobilenet-architectures-to-improve-computer-vision-models-ffb483ffcc0a



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

A standard convolutional layer takes as input a $D_F \times D_F \times M$ feature map $\mathbf{F}$ and produces a $D_F \times D_F \times N$ feature map $\mathbf{G}$ where D_F is the spatial width and height of a square input feature map[1], M is the number of input channels (input depth), D_G is the spatial width and height of a square output feature map and N is the number of output channel (output depth).

The standard convolutional layer is parameterized by convolution kernel $\mathbf{K}$ of size $D_K \times D_K \times M \times N$ where D_K is the spatial dimension of the kernel assumed to be square and M is number of input channels and N is the number of output channels as defined previously.

The output feature map for standard convolution assuming stride one and padding is computed as:

$$\mathbf{G}_{k,l,n} = \sum_{i,j,m} \mathbf{K}_{i,j,m,n} \cdot \mathbf{F}_{k+i-1,l+j-1,m} \tag{1}$$

Standard convolutions have the computational cost of:

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F \tag{2}$$

where the computational cost depends multiplicatively on the number of input channels M, the number of output channels N the kernel size $D_k \times D_k$ and the feature map size $D_F \times D_F$. MobileNet models address each of these

VS

Depthwise convolution with one filter per input channel (input depth) can be written as:

$$\hat{\mathbf{G}}_{k,l,m} = \sum_{i,j} \hat{\mathbf{K}}_{i,j,m} \cdot \mathbf{F}_{k+i-1,l+j-1,m} \tag{3}$$

where $\hat{\mathbf{K}}$ is the depthwise convolutional kernel of size $D_K \times D_K \times M$ where the m_{th} filter in $\hat{\mathbf{K}}$ is applied to the m_{th} channel in $\mathbf{F}$ to produce the m_{th} channel of the filtered output feature map $\hat{\mathbf{G}}$.

Depthwise convolution has a computational cost of:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F \tag{4}$$

Depthwise convolution is extremely efficient relative to standard convolution. However it only filters input channels, it does not combine them to create new features. So an additional layer that computes a linear combination of the output of depthwise convolution via 1×1 convolution is needed in order to generate these new features.

The combination of depthwise convolution and 1×1 (pointwise) convolution is called depthwise separable convolution which was originally introduced in [26].

Depthwise separable convolutions cost:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F \tag{5}$$

which is the sum of the depthwise and 1×1 pointwise convolutions.

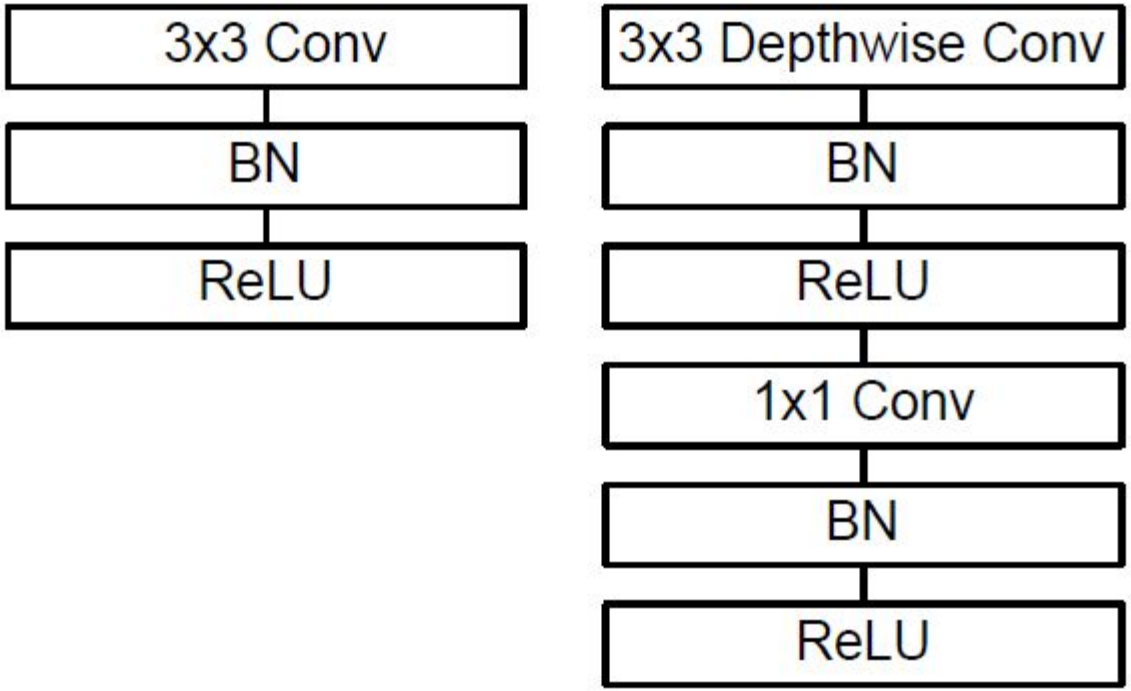


Figure 3. Left: Standard convolutional layer with batchnorm and ReLU. Right: Depthwise Separable convolutions with Depthwise and Pointwise layers followed by batchnorm and ReLU.

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Table 2. Resource Per Layer Type

Type	Mult-Adds	Parameters
Conv 1×1	94.86%	74.59%
Conv DW 3×3	3.06%	1.06%
Conv 3×3	1.19%	0.02%
Fully Connected	0.18%	24.33%

and width multiplier α, the number of input channels M becomes αM and the number of output channels N becomes αN.

The computational cost of a depthwise separable convolution with width multiplier α is:

$$D_K \cdot D_K \cdot \alpha M \cdot D_F \cdot D_F + \alpha M \cdot \alpha N \cdot D_F \cdot D_F \quad (6)$$

where $\alpha \in (0, 1]$ with typical settings of 1, 0.75, 0.5 and 0.25. $\alpha = 1$ is the baseline MobileNet and $\alpha < 1$ are reduced MobileNets. Width multiplier has the effect of re-

We can now express the computational cost for the core layers of our network as depthwise separable convolutions with width multiplier α and resolution multiplier ρ:

$$D_K \cdot D_K \cdot \alpha M \cdot \rho D_F \cdot \rho D_F + \alpha M \cdot \alpha N \cdot \rho D_F \cdot \rho D_F \quad (7)$$

where $\rho \in (0, 1]$ which is typically set implicitly so that the input resolution of the network is 224, 192, 160 or 128. $\rho = 1$ is the baseline MobileNet and $\rho < 1$ are reduced computation MobileNets. Resolution multiplier has the effect of reducing computational cost by ρ^2.

Table 3. Resource usage for modifications to standard convolution. Note that each row is a cumulative effect adding on top of the previous row. This example is for an internal MobileNet layer with $D_K = 3, M = 512, N = 512, D_F = 14$.

Layer/Modification	Million Mult-Adds	Million Parameters
Convolution	462	2.36
Depthwise Separable Conv	52.3	0.27
$\alpha = 0.75$	29.6	0.15
$\rho = 0.714$	15.1	0.15

Table 4. Depthwise Separable vs Full Convolution MobileNet

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
Conv MobileNet	71.7%	4866	29.3
MobileNet	70.6%	569	4.2

Table 5. Narrow vs Shallow MobileNet

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
0.75 MobileNet	68.4%	325	2.6
Shallow MobileNet	65.3%	307	2.9

Table 6. MobileNet Width Multiplier

Width Multiplier	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
0.75 MobileNet-224	68.4%	325	2.6
0.5 MobileNet-224	63.7%	149	1.3
0.25 MobileNet-224	50.6%	41	0.5

Table 7. MobileNet Resolution

Resolution	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
1.0 MobileNet-192	69.1%	418	4.2
1.0 MobileNet-160	67.2%	290	4.2
1.0 MobileNet-128	64.4%	186	4.2

MobileNetV2: Inverted Residuals and Linear Bottlenecks

Mark Sandler Andrew Howard Menglong Zhu Andrey Zhmoginov Liang-Chieh Chen
Google Inc.
{sandler, howarda, menglong, azhmogin, lcchen}@google.com

This paper introduces a new neural network architecture that is specifically tailored for mobile and resource constrained environments. Our network pushes the state of the art for mobile tailored computer vision models, by significantly decreasing the number of operations and memory needed while retaining the same accuracy.

Our main contribution is a novel layer module: the inverted residual with linear bottleneck. This module takes as an input a low-dimensional compressed representation which is first expanded to high dimension and filtered with a lightweight depthwise convolution. Features are subsequently projected back to a low-dimensional representation with a *linear convolution*. The official implementation is available as part of TensorFlow-Slim model library in [4].

3.1. Depthwise Separable Convolutions

Depthwise Separable Convolutions are a key building block for many efficient neural network architectures [27, 28, 20] and we use them in the present work as well. The basic idea is to replace a full convolutional operator with a factorized version that splits convolution into two separate layers. The first layer is called a depthwise convolution, it performs lightweight filtering by applying a single convolutional filter per input channel. The second layer is a 1×1 convolution, called a pointwise convolution, which is responsible for building new features through computing linear combinations of the input channels.

3.2. Linear Bottlenecks

Consider a deep neural network consisting of n layers L_i each of which has an activation tensor of dimensions $h_i \times w_i \times d_i$. Throughout this section we will be discussing the basic properties of these activation tensors, which we will treat as containers of $h_i \times w_i$ "pixels" with d_i dimensions. Informally, for an input set of real images, we say that the set of layer activations (for any layer L_i) forms a "manifold of interest". It has been long assumed that manifolds of interest in neural networks could be embedded in low-dimensional subspaces. In other words, when we look at all individual d-channel pixels of a deep convolutional layer, the information encoded in those values actually lie in some manifold, which in turn is embeddable into a low-dimensional subspace[2].

At a first glance, such a fact could then be captured and exploited by simply reducing the dimensionality of a layer thus reducing the dimensionality of the operating space. This has been successfully exploited by MobileNetV1 [27] to effectively trade off between computation and accuracy via a width multiplier parameter, and has been incorporated into efficient model designs of other networks as well [20]. Following that intuition, the width multiplier approach allows one to reduce the dimensionality of the activation space until the manifold of interest spans this entire space. However, this intuition breaks down when we recall that deep convolutional neural networks actually have non-linear per coordinate transformations, such as ReLU. For example, ReLU applied to a line in 1D space produces a 'ray', where as in $\mathcal{R}^n$ space, it generally results in a piece-wise linear curve with n-joints.

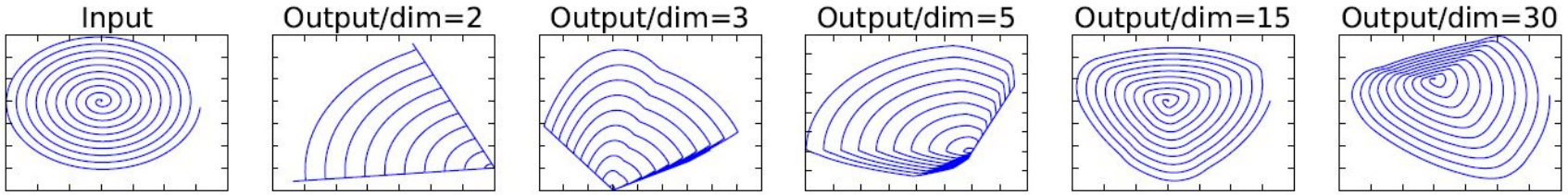
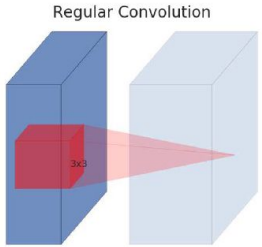


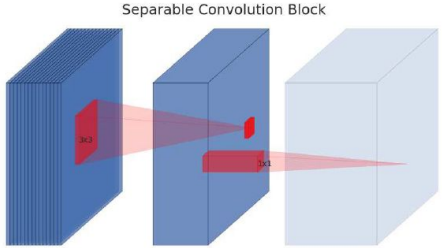
Figure 1: Examples of ReLU transformations of low-dimensional manifolds embedded in higher-dimensional spaces. In these examples the initial spiral is embedded into an n-dimensional space using random matrix T followed by ReLU, and then projected back to the 2D space using T^{-1}. In examples above $n = 2, 3$ result in information loss where certain points of the manifold collapse into each other, while for $n = 15$ to 30 the transformation is highly non-convex.

These two insights provide us with an empirical hint for optimizing existing neural architectures: assuming the manifold of interest is low-dimensional we can capture this by inserting *linear bottleneck* layers into the convolutional blocks. Experimental evidence suggests that using linear layers is crucial as it prevents non-linearities from destroying too much information. In Section 6, we show empirically that using non-linear layers in bottlenecks indeed hurts the performance by several percent, further validating our hypothesis[3]. We

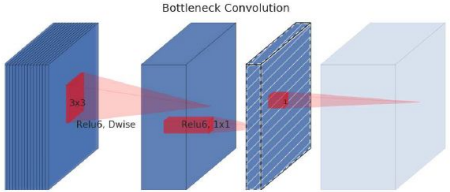
(a) Regular



(b) Separable



(c) Separable with linear bottleneck



(d) Bottleneck with expansion layer

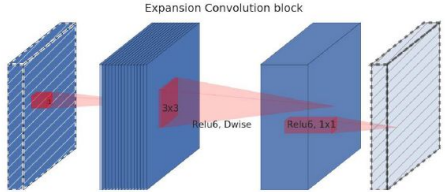
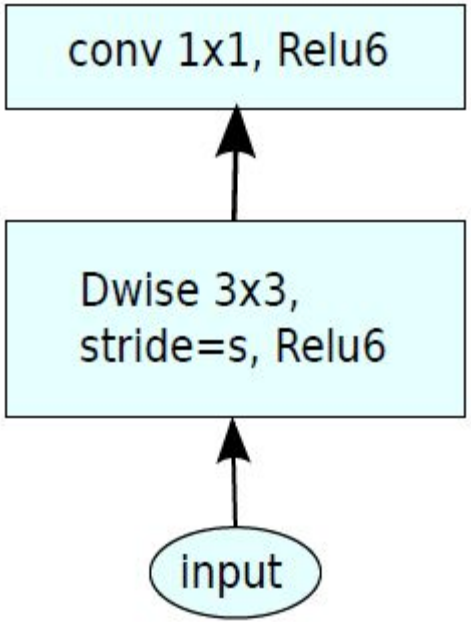
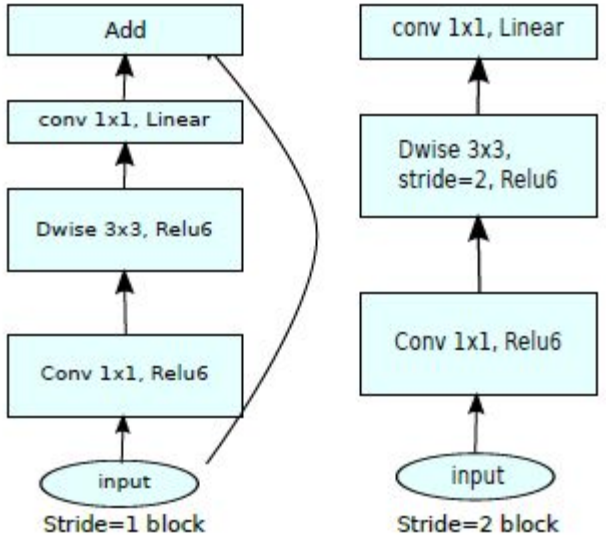


Figure 2: Evolution of separable convolution blocks. The diagonally hatched texture indicates layers that do not contain non-linearities. The last (lightly colored) layer indicates the beginning of the next block. Note: 2d and 2c are equivalent blocks when stacked. Best viewed in color.



(b) MobileNet[27]



(d) Mobilenet V2

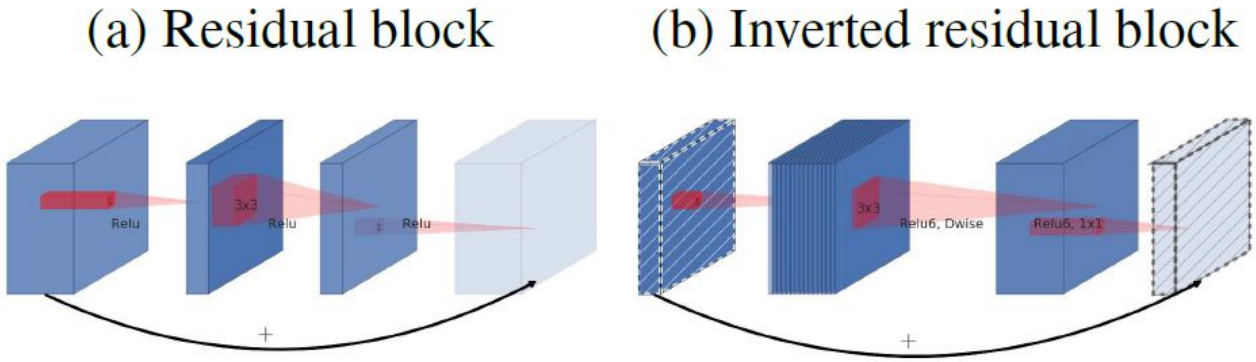


Figure 3: The difference between residual block [8, 30] and inverted residual. Diagonally hatched layers do not use non-linearities. We use thickness of each block to indicate its relative number of channels. Note how classical residuals connects the layers with high number of channels, whereas the inverted residuals connect the bottlenecks. Best viewed in color.

Table 1. The architecture of MobileNetV2 contains the initial fully convolution layer with 32 filters, followed by 19 *residual bottleneck* layers described in the Table 2. We use ReLU6 as the non-linearity because of its robustness when used with low-precision computation [27]. We always use kernel size 3×3 as is standard for modern networks, and utilize dropout and batch normalization during training.

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Table 2: MobileNetV2 : Each line describes a sequence of 1 or more identical (modulo stride) layers, repeated n times. All layers in the same sequence have the same number c of output channels. The first layer of each sequence has a stride s and all others use stride 1. All spatial convolutions use 3×3 kernels. The expansion factor t is always applied to the input size as described in Table 1.

Size	MobileNetV1	MobileNetV2	ShuffleNet (2x,g=3)
112x112	1/O(1)	1/O(1)	1/O(1)
56x56	128/800	32/200	48/300
28x28	256/400	64/100	400/600K
14x14	512/200	160/62	800/310
7x7	1024/199	320/32	1600/156
1x1	1024/2	1280/2	1600/3
max	800K	**200K**	600K

Table 3: The max number of channels/memory (in Kb) that needs to be materialized at each spatial resolution for different architectures. We assume 16-bit floats for activations. For ShuffleNet, we use $2x, g = 3$ that matches the performance of MobileNetV1 and MobileNetV2. For the first layer of MobileNetV2 and ShuffleNet we can employ the trick described in Section 5 to reduce memory requirement. Even though ShuffleNet employs bottlenecks elsewhere, the non-bottleneck tensors still need to be materialized due to the presence of shortcuts between non-bottleneck tensors.

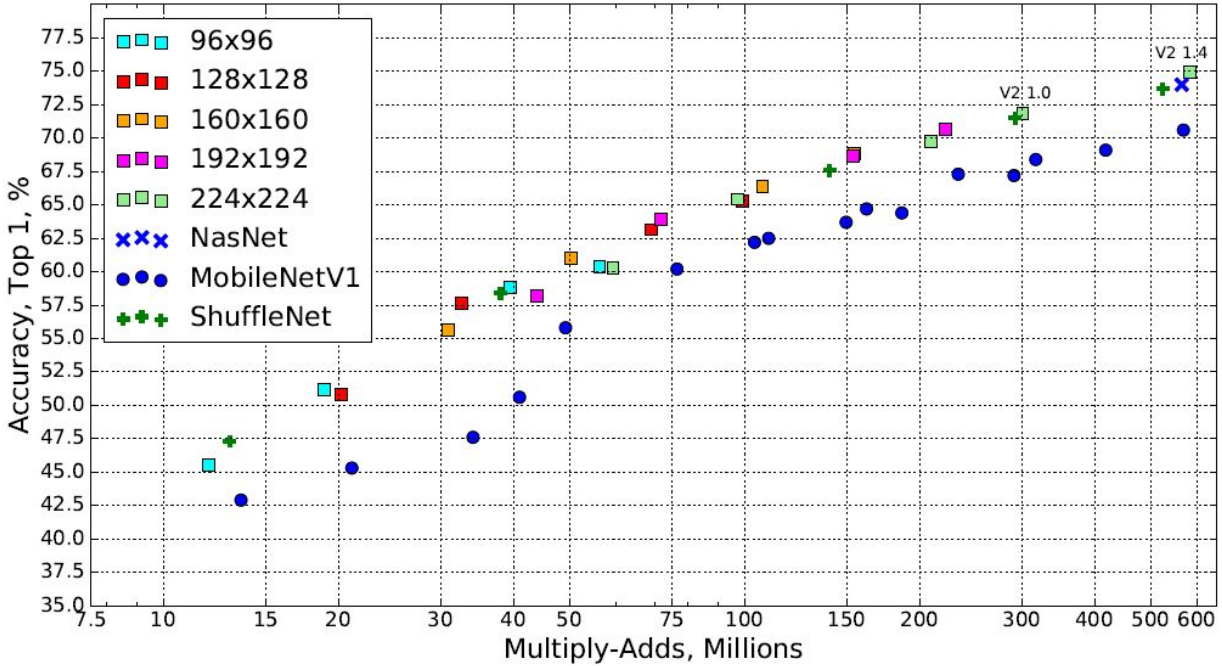
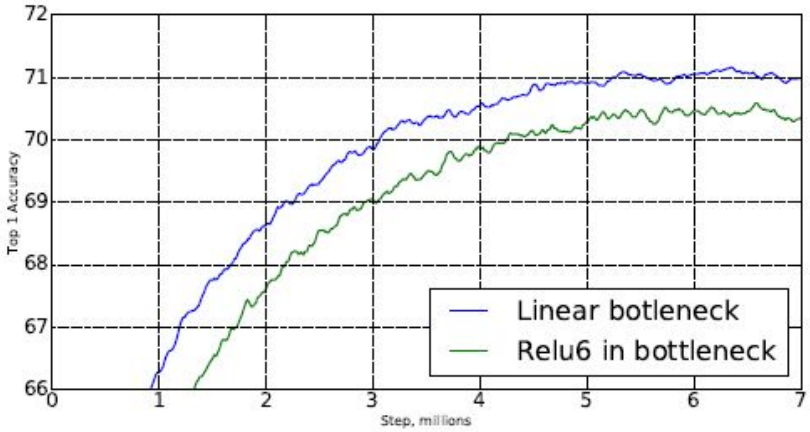
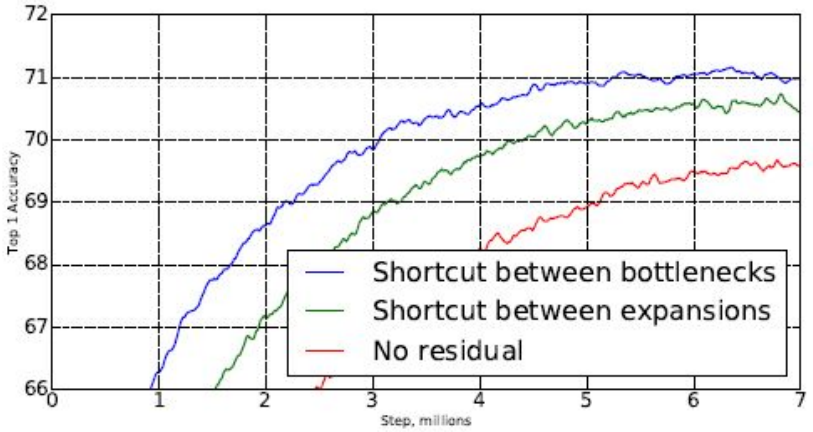


Figure 5: Performance curve of MobileNetV2 vs MobileNetV1, ShuffleNet, NAS. For our networks we use multipliers 0.35, 0.5, 0.75, 1.0 for all resolutions, and additional 1.4 for for 224. Best viewed in color.



(a) Impact of non-linearity in the bottleneck layer.



(b) Impact of variations in residual blocks.

Figure 6: The impact of non-linearities and various types of shortcut (residual) connections.

Network	Top 1	Params	MAdds	CPU
MobileNetV1	70.6	4.2M	575M	113ms
ShuffleNet (1.5)	71.5	**3.4M**	292M	-
ShuffleNet (x2)	73.7	5.4M	524M	-
NasNet-A	74.0	5.3M	564M	183ms
MobileNetV2	**72.0**	**3.4M**	**300M**	**75ms**
MobileNetV2 (1.4)	**74.7**	6.9M	585M	**143ms**

Table 4: Performance on ImageNet, comparison for different networks. As is common practice for ops, we count the total number of Multiply-Adds. In the last column we report running time in milliseconds (ms) for a single large core of the Google Pixel 1 phone (using TF-Lite). We do not report ShuffleNet numbers as efficient group convolutions and shuffling are not yet supported.

Searching for MobileNetV3

Andrew Howard[1] **Mark Sandler**[1] **Grace Chu**[1] **Liang-Chieh Chen**[1] **Bo Chen**[1] **Mingxing Tan**[2]
Weijun Wang[1] **Yukun Zhu**[1] **Ruoming Pang**[2] **Vijay Vasudevan**[2] **Quoc V. Le**[2] **Hartwig Adam**[1]
[1]Google AI, [2]Google Brain
{howarda, sandler, cxy, lcchen, bochen, tanmingxing, weijunw, yukun, rpang, vrv, qvl, hadam}@google.com

Mobile models have been built on increasingly more efficient building blocks. MobileNetV1 [19] introduced *depthwise separable convolutions* as an efficient replacement for traditional convolution layers. Depthwise separable convo-

MobileNetV2 [39] introduced the linear bottleneck and inverted residual structure in order to make even more efficient layer structures by leveraging the low rank nature of the problem. This structure is shown on Figure 3 and is

MnasNet [43] built upon the MobileNetV2 structure by introducing lightweight attention modules based on squeeze and excitation into the bottleneck structure. Note that the

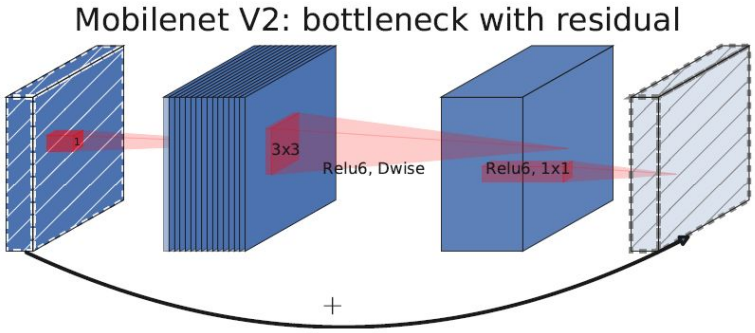


Figure 3. MobileNetV2 [39] layer (Inverted Residual and Linear Bottleneck). Each block consists of narrow input and output (bottleneck), which don't have nonlinearity, followed by expansion to a much higher-dimensional space and projection to the output. The residual connects bottleneck (rather than expansion).

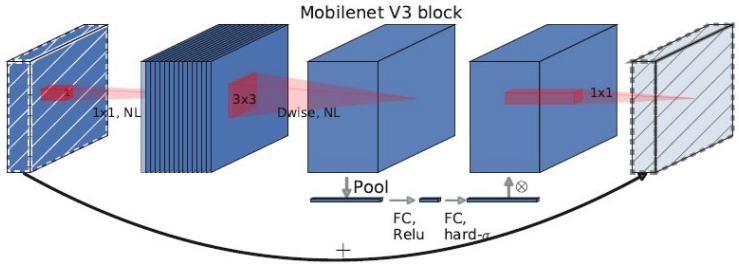


Figure 4. MobileNetV2 + Squeeze-and-Excite [20]. In contrast with [20] we apply the squeeze and excite in the residual layer. We use different nonlinearity depending on the layer, see section 5.2 for details.

4. Network Search

Network search has shown itself to be a very powerful tool for discovering and optimizing network architectures [53, 43, 5, 48]. For MobileNetV3 we use platform-aware NAS to search for the global network structures by optimizing each network block. We then use the NetAdapt algorithm to search per layer for the number of filters. These techniques are complementary and can be combined to effectively find optimized models for a given hardware platform.

optimized for small mobile models. Specifically, it uses a multi-objective reward $ACC(m) \times [LAT(m)/TAR]^w$ to approximate Pareto-optimal solutions, by balancing model accuracy $ACC(m)$ and latency $LAT(m)$ for each model m based on the target latency TAR. We observe that accuracy changes much more dramatically with latency for small models; therefore, we need a smaller weight factor $w = -0.15$ (vs the original $w = -0.07$ in [43]) to compensate for the larger accuracy change for different latencies.

1. Starts with a seed network architecture found by platform-aware NAS.
2. For each step:
    (a) Generate a set of new *proposals*. Each proposal represents a modification of an architecture that generates at least δ reduction in latency compared to the previous step.
    (b) For each proposal we use the pre-trained model from the previous step and populate the new proposed architecture, truncating and randomly initializing missing weights as appropriate. Fine-tune each proposal for T steps to get a coarse estimate of the accuracy.
    (c) Selected best proposal according to some metric.
3. Iterate previous step until target latency is reached.

This process is repeated until the latency reaches its target, and then we re-train the new architecture from scratch. We use the same proposal generator as was used in [48] for MobilenetV2. Specifically, we allow the following two types of proposals:

1. Reduce the size of any expansion layer;

2. Reduce bottleneck in all blocks that share the same bottleneck size - to maintain residual connections.

5.1. Redesigning Expensive Layers

Once models are found through architecture search, we observe that some of the last layers as well as some of the earlier layers are more expensive than others. We propose some modifications to the architecture to reduce the latency of these slow layers while maintaining the accuracy. These modifications are outside of the scope of the current search space.

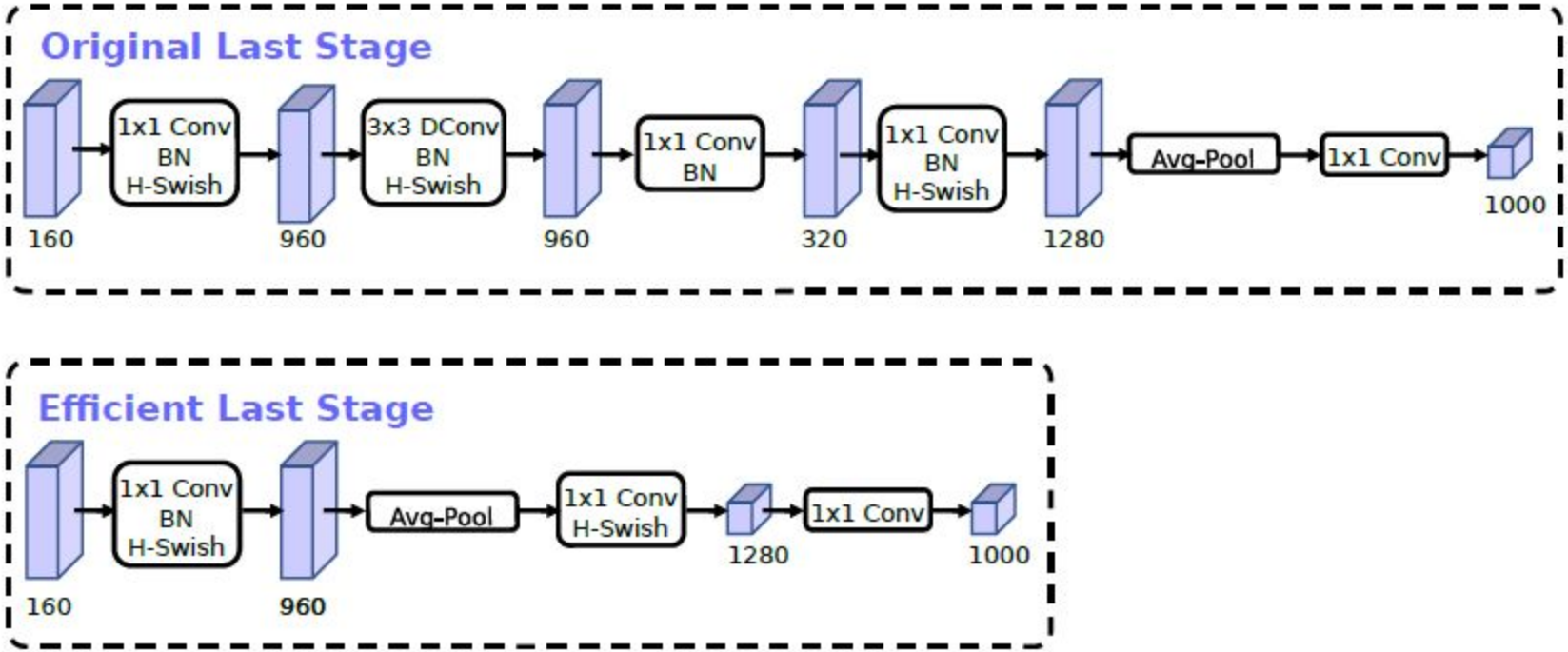


Figure 5. Comparison of original last stage and efficient last stage. This more efficient last stage is able to drop three expensive layers at the end of the network at no loss of accuracy.

5.2. Nonlinearities

In [36, 13, 16] a nonlinearity called *swish* was introduced that when used as a drop-in replacement for ReLU, that significantly improves the accuracy of neural networks. The nonlinearity is defined as

$$\text{swish}\, x = x \cdot \sigma(x)$$

While this nonlinearity improves accuracy, it comes with non-zero cost in embedded environments as the sigmoid function is much more expensive to compute on mobile devices. We deal with this problem in two ways.

1. We replace sigmoid function with its piece-wise linear hard analog: $\frac{\text{ReLU6}(x+3)}{6}$ similar to [11, 44]. The minor difference is we use ReLU6 rather than a custom clipping constant. Similarly, the hard version of swish becomes

$$\text{h-swish}[x] = x\frac{\text{ReLU6}(x+3)}{6}$$

2. The cost of applying nonlinearity decreases as we go deeper into the network, since each layer activation memory typically halves every time the resolution drops. Incidentally, we find that most of the benefits swish are realized by using them only in the deeper layers. Thus in our architectures we only use h-swish at the second half of the model. We refer to the tables 1 and 2 for the precise layout.

Input	Operator	exp size	#out	SE	NL	s
$224^2 \times 3$	conv2d	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	-	RE	1
$112^2 \times 16$	bneck, 3x3	64	24	-	RE	2
$56^2 \times 24$	bneck, 3x3	72	24	-	RE	1
$56^2 \times 24$	bneck, 5x5	72	40	✓	RE	2
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 3x3	240	80	-	HS	2
$14^2 \times 80$	bneck, 3x3	200	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	480	112	✓	HS	1
$14^2 \times 112$	bneck, 3x3	672	112	✓	HS	1
$14^2 \times 112$	bneck, 5x5	672	160	✓	HS	2
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	conv2d, 1x1	-	960	-	HS	1
$7^2 \times 960$	pool, 7x7	-	-	-	-	1
$1^2 \times 960$	conv2d 1x1, NBN	-	1280	-	HS	1
$1^2 \times 1280$	conv2d 1x1, NBN	-	k	-	-	1

Table 1. Specification for MobileNetV3-Large. SE denotes whether there is a Squeeze-And-Excite in that block. NL denotes the type of nonlinearity used. Here, HS denotes h-swish and RE denotes ReLU. NBN denotes no batch normalization. s denotes stride.

Input	Operator	exp size	#out	SE	NL	s
$224^2 \times 3$	conv2d, 3x3	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	✓	RE	2
$56^2 \times 16$	bneck, 3x3	72	24	-	RE	2
$28^2 \times 24$	bneck, 3x3	88	24	-	RE	1
$28^2 \times 24$	bneck, 5x5	96	40	✓	HS	2
$14^2 \times 40$	bneck, 5x5	240	40	✓	HS	1
$14^2 \times 40$	bneck, 5x5	240	40	✓	HS	1
$14^2 \times 40$	bneck, 5x5	120	48	✓	HS	1
$14^2 \times 48$	bneck, 5x5	144	48	✓	HS	1
$14^2 \times 48$	bneck, 5x5	288	96	✓	HS	2
$7^2 \times 96$	bneck, 5x5	576	96	✓	HS	1
$7^2 \times 96$	bneck, 5x5	576	96	✓	HS	1
$7^2 \times 96$	conv2d, 1x1	-	576	✓	HS	1
$7^2 \times 576$	pool, 7x7	-	-	-	-	1
$1^2 \times 576$	conv2d 1x1, NBN	-	1024	-	HS	1
$1^2 \times 1024$	conv2d 1x1, NBN	-	k	-	-	1

Table 2. Specification for MobileNetV3-Small. See table 1 for notation.

Network	Top-1	MAdds	Params	P-1	P-2	P-3
V3-Large 1.0	**75.2**	**219**	5.4M	**51**	**61**	**44**
V3-Large 0.75	73.3	155	4.0M	39	46	40
MnasNet-A1	75.2	315	3.9M	71	86	61
Proxyless[5]	74.6	320	4.0M	72	84	60
V2 1.0	72.0	300	3.4M	64	76	56
V3-Small 1.0	**67.4**	**56**	2.5M	**15.8**	**19.4**	**14.4**
V3-Small 0.75	65.4	44	2.0M	12.8	15.6	11.7
Mnas-small [43]	64.9	65.1	1.9M	20.3	24.2	17.2
V2 0.35	60.8	59.2	1.6M	16.6	19.6	13.9

Table 3. Floating point performance on the Pixel family of phones (P-n denotes a Pixel-n phone). All latencies are in ms and are measured using a single large core with a batch size of one. Top-1 accuracy is on ImageNet.

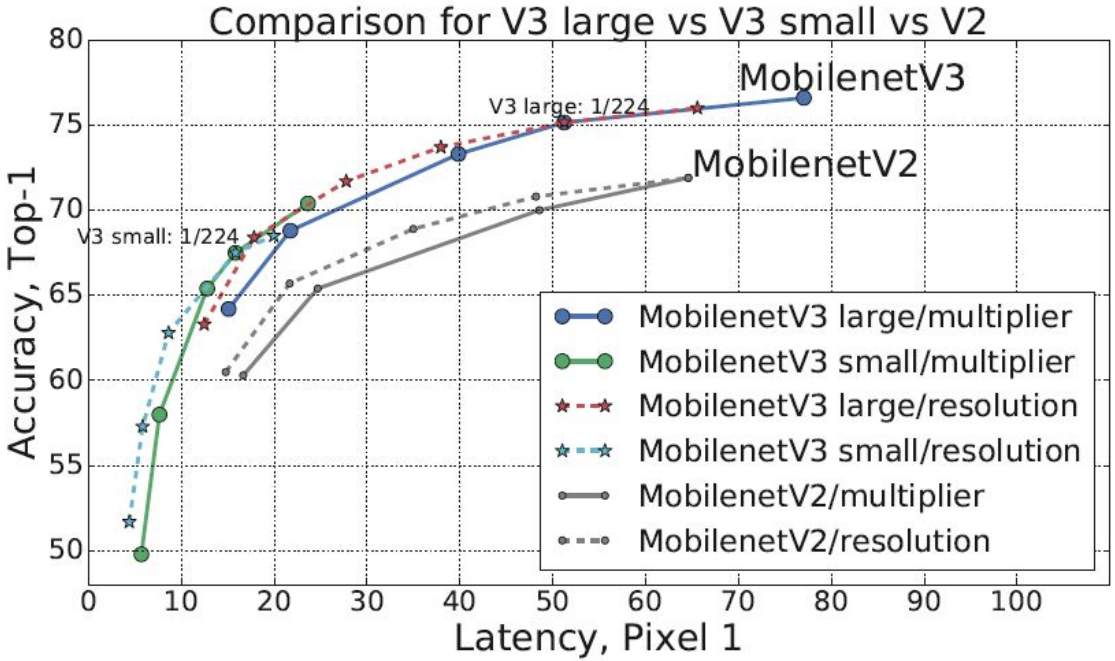


Figure 7. Performance of MobileNetV3 as a function of different multipliers and resolutions. In our experiments we have used multipliers 0.35, 0.5, 0.75, 1.0 and 1.25, with a fixed resolution of 224, and resolutions 96, 128, 160, 192, 224 and 256 with a fixed depth multiplier of 1.0. Best viewed in color. Top-1 accuracy is on ImageNet and latency is in ms.

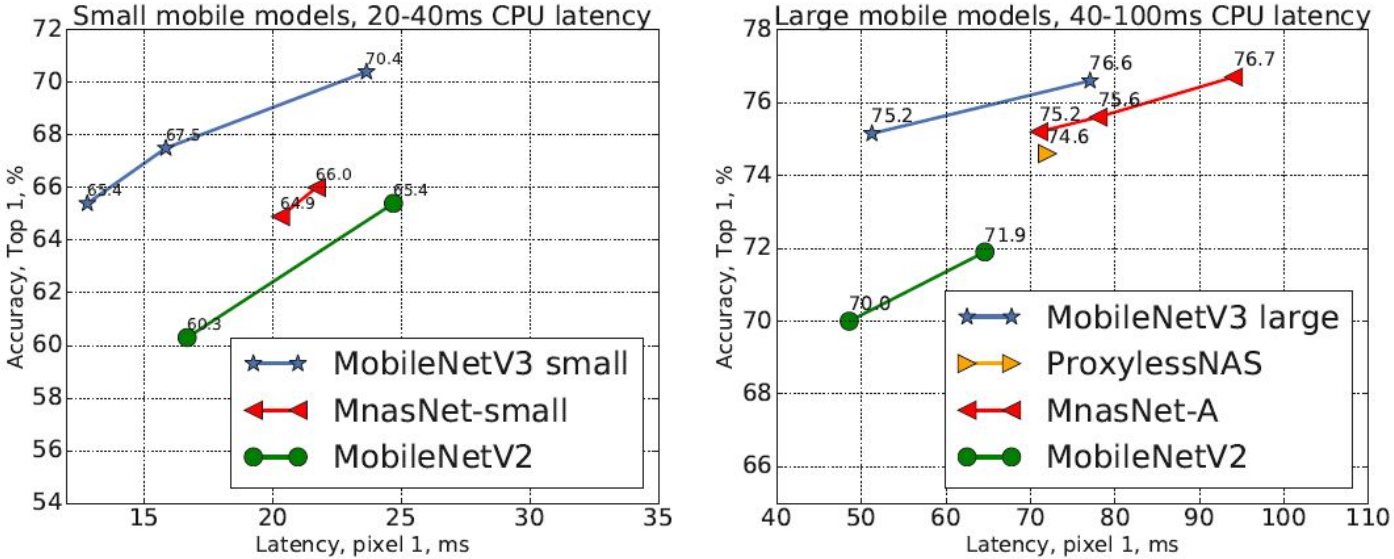


Figure 1. The trade-off between Pixel 1 latency and top-1 ImageNet accuracy. All models use the input resolution 224. V3 large and V3 small use multipliers 0.75, 1 and 1.25 to show optimal frontier. All latencies were measured on a single large core of the same device using TFLite[1]. MobileNetV3-Small and Large are our proposed next-generation mobile models.

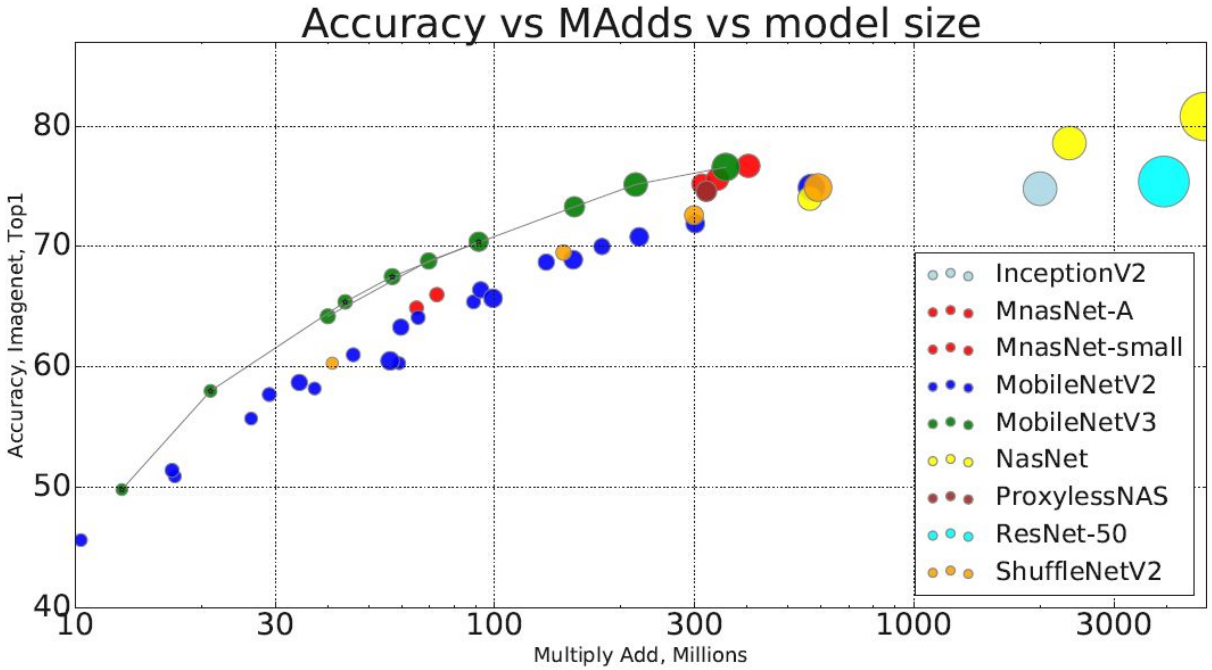


Figure 2. The trade-off between MAdds and top-1 accuracy. This allows to compare models that were targeted different hardware or software frameworks. All MobileNetV3 are for input resolution 224 and use multipliers 0.35, 0.5, 0.75, 1 and 1.25. See section 6 for other resolutions. Best viewed in color.

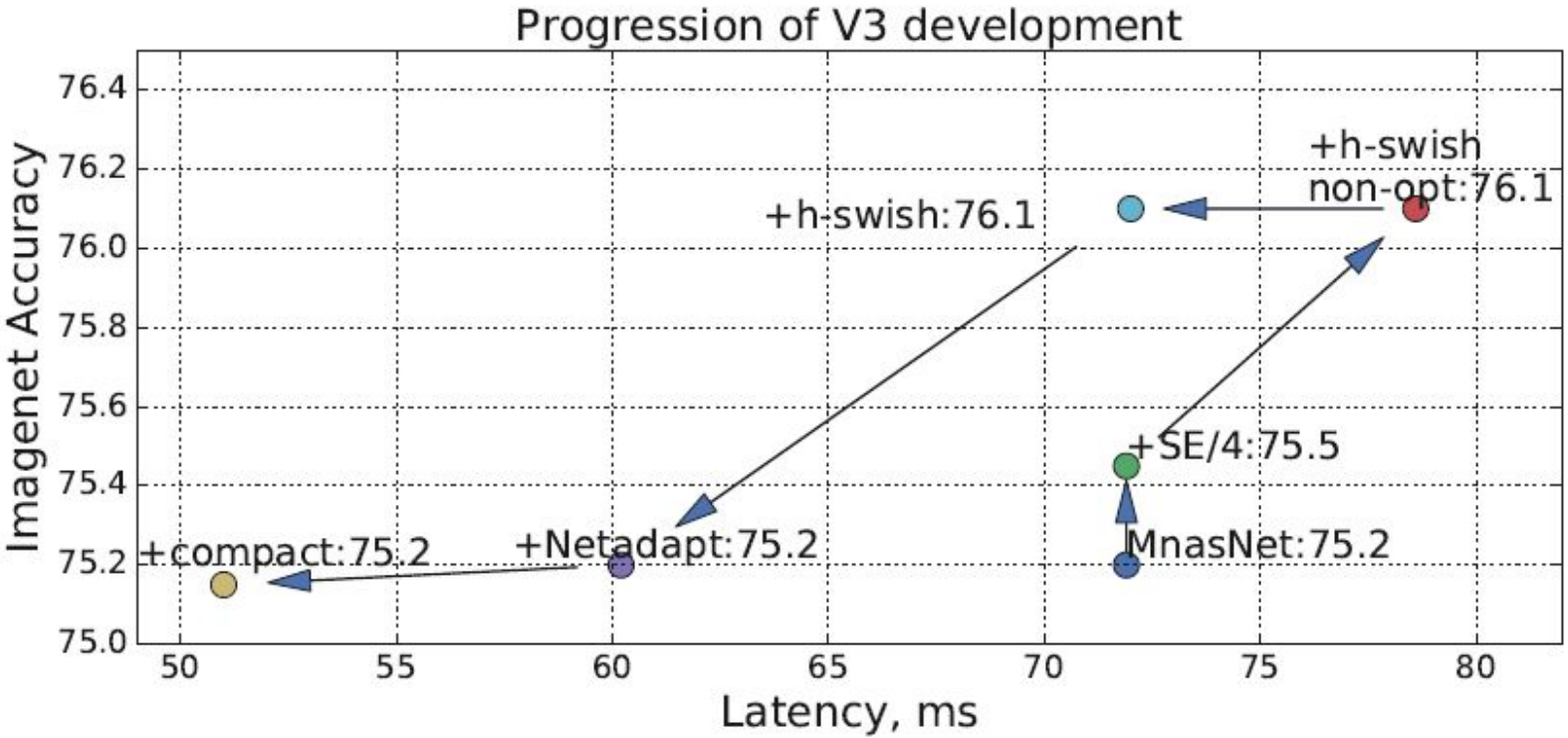


Figure 9. Impact of individual components in the development of MobileNetV3. Progress is measured by moving up and to the left.

EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks

Mingxing Tan [1] **Quoc V. Le** [1]

ICML 2019

Abstract

Convolutional Neural Networks (ConvNets) are commonly developed at a fixed resource budget, and then scaled up for better accuracy if more resources are available. In this paper, we systematically study model scaling and identify that carefully balancing network depth, width, and resolution can lead to better performance. Based on this observation, we propose a new scaling method that uniformly scales all dimensions of depth/width/resolution using a simple yet highly effective *compound coefficient*. We demonstrate the effectiveness of this method on scaling up MobileNets and ResNet.

1. Introduction

Scaling up ConvNets is widely used to achieve better accuracy. For example, ResNet (He et al., 2016) can be scaled up from ResNet-18 to ResNet-200 by using more layers; Recently, GPipe (Huang et al., 2018) achieved 84.3% ImageNet top-1 accuracy by scaling up a baseline model four time larger. However, the process of scaling up ConvNets has never been well understood and there are currently many

of them with constant ratio. Based on this observation, we propose a simple yet effective *compound scaling method*. Unlike conventional practice that arbitrary scales these factors, our method uniformly scales network width, depth, and resolution with a set of fixed scaling coefficients. For example, if we want to use 2^N times more computational resources, then we can simply increase the network depth by α^N, width by β^N, and image size by γ^N, where α, β, γ are constant coefficients determined by a small grid search on the original small model. Figure 2 illustrates the difference between our scaling method and conventional methods.

Observation 1 – Scaling up any dimension of network width, depth, or resolution improves accuracy, but the accuracy gain diminishes for bigger models.

Observation 2 – In order to pursue better accuracy and efficiency, it is critical to balance all dimensions of network width, depth, and resolution during ConvNet scaling.

In this paper, we propose a new **compound scaling method**, which use a compound coefficient ϕ to uniformly scales network width, depth, and resolution in a principled way:

$$\begin{aligned} \text{depth: } & d = \alpha^\phi \\ \text{width: } & w = \beta^\phi \\ \text{resolution: } & r = \gamma^\phi \\ \text{s.t. } & \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2 \\ & \alpha \geq 1, \beta \geq 1, \gamma \geq 1 \end{aligned} \tag{3}$$

where α, β, γ are constants that can be determined by a small grid search. Intuitively, ϕ is a user-specified coeffi-

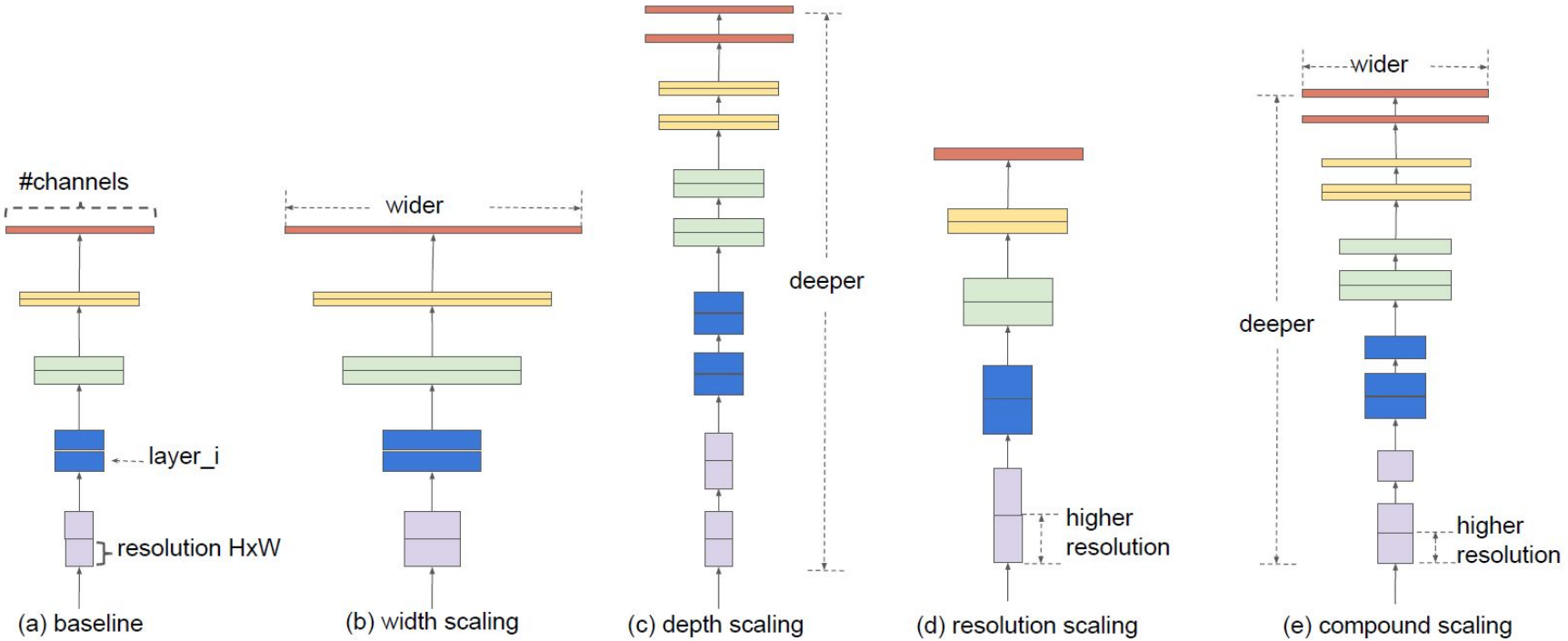


Figure 2. **Model Scaling.** (a) is a baseline network example; (b)-(d) are conventional scaling that only increases one dimension of network width, depth, or resolution. (e) is our proposed compound scaling method that uniformly scales all three dimensions with a fixed ratio.

Table 1. **EfficientNet-B0 baseline network** – Each row describes a stage i with $\hat{L}_i$ layers, with input resolution $\langle \hat{H}_i, \hat{W}_i \rangle$ and output channels $\hat{C}_i$. Notations are adopted from equation 2.

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	28×28	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

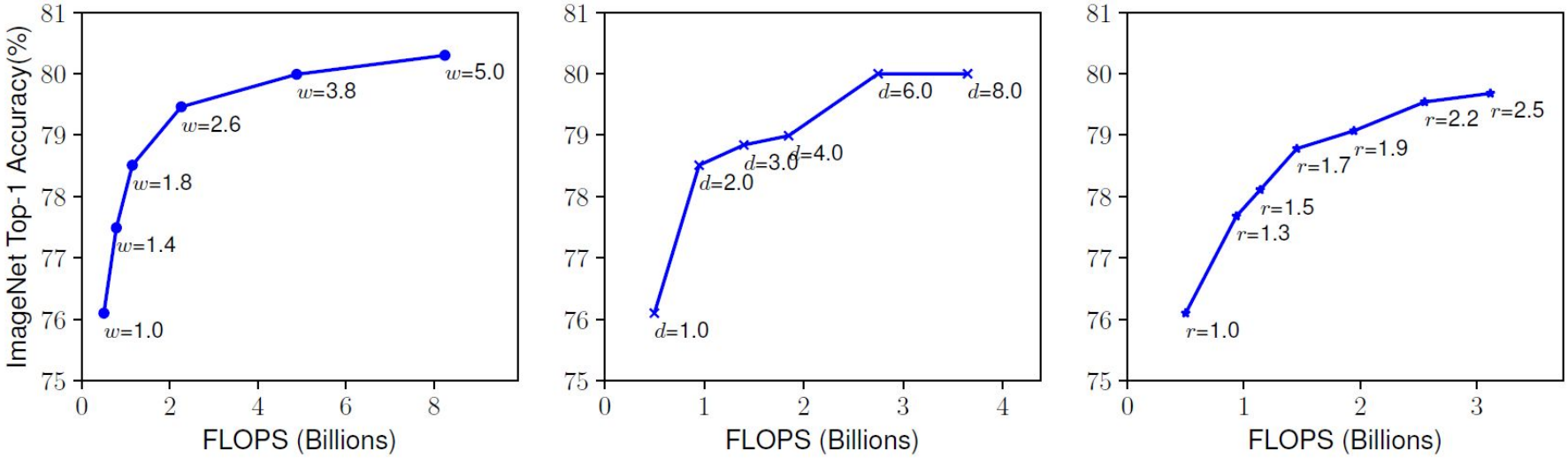
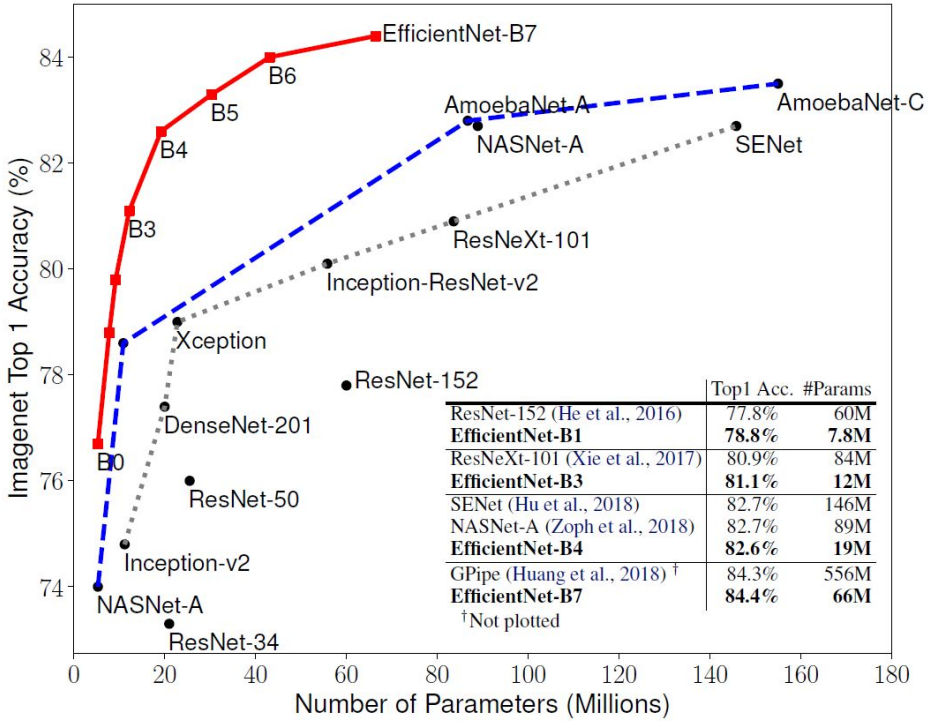


Figure 3. **Scaling Up a Baseline Model with Different Network Width (w), Depth (d), and Resolution (r) Coefficients.** Bigger networks with larger width, depth, or resolution tend to achieve higher accuracy, but the accuracy gain quickly saturate after reaching 80%, demonstrating the limitation of single dimension scaling. Baseline network is described in Table 1.

Table 2. **EfficientNet Performance Results on ImageNet** (Russakovsky et al., 2015). All EfficientNet models are scaled from our baseline EfficientNet-B0 using different compound coefficient ϕ in Equation 3. ConvNets with similar top-1/top-5 accuracy are grouped together for efficiency comparison. Our scaled EfficientNet models consistently reduce parameters and FLOPS by an order of magnitude (up to 8.4x parameter reduction and up to 16x FLOPS reduction) than existing ConvNets.

Model	Top-1 Acc.	Top-5 Acc.	#Params	Ratio-to-EfficientNet	#FLOPS	Ratio-to-EfficientNet
EfficientNet-B0	**76.3%**	**93.2%**	**5.3M**	**1x**	**0.39B**	**1x**
ResNet-50 (He et al., 2016)	76.0%	93.0%	26M	4.9x	4.1B	11x
DenseNet-169 (Huang et al., 2017)	76.2%	93.2%	14M	2.6x	3.5B	8.9x
EfficientNet-B1	**78.8%**	**94.4%**	**7.8M**	**1x**	**0.70B**	**1x**
ResNet-152 (He et al., 2016)	77.8%	93.8%	60M	7.6x	11B	16x
DenseNet-264 (Huang et al., 2017)	77.9%	93.9%	34M	4.3x	6.0B	8.6x
Inception-v3 (Szegedy et al., 2016)	78.8%	94.4%	24M	3.0x	5.7B	8.1x
Xception (Chollet, 2017)	79.0%	94.5%	23M	3.0x	8.4B	12x
EfficientNet-B2	**79.8%**	**94.9%**	**9.2M**	**1x**	**1.0B**	**1x**
Inception-v4 (Szegedy et al., 2017)	80.0%	95.0%	48M	5.2x	13B	13x
Inception-resnet-v2 (Szegedy et al., 2017)	80.1%	95.1%	56M	6.1x	13B	13x
EfficientNet-B3	**81.1%**	**95.5%**	**12M**	**1x**	**1.8B**	**1x**
ResNeXt-101 (Xie et al., 2017)	80.9%	95.6%	84M	7.0x	32B	18x
PolyNet (Zhang et al., 2017)	81.3%	95.8%	92M	7.7x	35B	19x
EfficientNet-B4	**82.6%**	**96.3%**	**19M**	**1x**	**4.2B**	**1x**
SENet (Hu et al., 2018)	82.7%	96.2%	146M	7.7x	42B	10x
NASNet-A (Zoph et al., 2018)	82.7%	96.2%	89M	4.7x	24B	5.7x
AmoebaNet-A (Real et al., 2019)	82.8%	96.1%	87M	4.6x	23B	5.5x
PNASNet (Liu et al., 2018)	82.9%	96.2%	86M	4.5x	23B	6.0x
EfficientNet-B5	**83.3%**	**96.7%**	**30M**	**1x**	**9.9B**	**1x**
AmoebaNet-C (Cubuk et al., 2019)	83.5%	96.5%	155M	5.2x	41B	4.1x
EfficientNet-B6	**84.0%**	**96.9%**	**43M**	**1x**	**19B**	**1x**
EfficientNet-B7	**84.4%**	**97.1%**	**66M**	**1x**	**37B**	**1x**
GPipe (Huang et al., 2018)	84.3%	97.0%	557M	8.4x	-	-

We omit ensemble and multi-crop models (Hu et al., 2018), or models pretrained on 3.5B Instagram images (Mahajan et al., 2018).



	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**78.8%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.1%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**82.6%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.4%**	**66M**

†Not plotted

Figure 1. **Model Size vs. ImageNet Accuracy.** All numbers are for single-crop, single-model. Our EfficientNets significantly outperform other ConvNets. In particular, EfficientNet-B7 achieves new state-of-the-art 84.4% top-1 accuracy but being 8.4x smaller and 6.1x faster than GPipe. EfficientNet-B1 is 7.6x smaller and 5.7x faster than ResNet-152. Details are in Table 2 and 4.

Table 3. **Scaling Up MobileNets and ResNet.**

Model	FLOPS	Top-1 Acc.
Baseline MobileNetV1 (Howard et al., 2017)	0.6B	70.6%
Scale MobileNetV1 by width (w=2)	2.2B	74.2%
Scale MobileNetV1 by resolution (r=2)	2.2B	72.7%
compound scale (d=1.4, w=1.2, r=1.3)	**2.3B**	**75.6%**
Baseline MobileNetV2 (Sandler et al., 2018)	0.3B	72.0%
Scale MobileNetV2 by depth (d=4)	1.2B	76.8%
Scale MobileNetV2 by width (w=2)	1.1B	76.4%
Scale MobileNetV2 by resolution (r=2)	1.2B	74.8%
MobileNetV2 compound scale	**1.3B**	**77.4%**
Baseline ResNet-50 (He et al., 2016)	4.1B	76.0%
Scale ResNet-50 by depth (d=4)	16.2B	78.1%
Scale ResNet-50 by width (w=2)	14.7B	77.7%
Scale ResNet-50 by resolution (r=2)	16.4B	77.5%
ResNet-50 compound scale	**16.7B**	**78.8%**

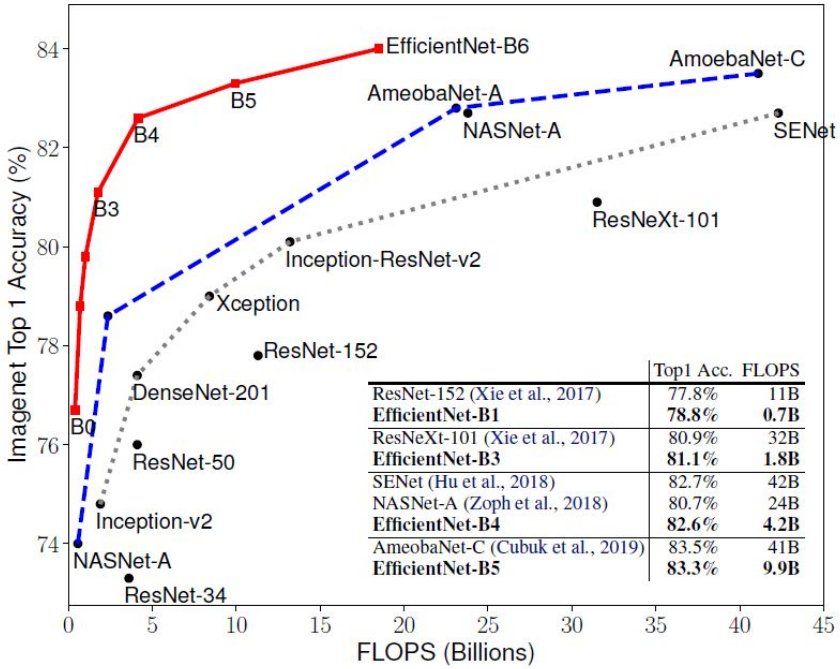


Figure 5. **FLOPS vs. ImageNet Accuracy.**

Journal of Imaging Science and Technology® 63(6): 060410-1–060410-12, 2019.

Deep Image Demosaicing for Submicron Image Sensors

Irina Kim, Seongwook Song, Soonkeun Chang, Sukhwan Lim, and Kai Guo
S.LSI, Device Solutions, Samsung Electronics, 1-1, Samsungjeonja-ro, Hwaseong-si, Gyeonggi-do, Korea
E-mail: irina.s.kim@gmail.com

1. INTRODUCTION

Most digital cameras capture color images using a single image sensor overlaid with a color filter array (CFA), acquiring only one color per pixel and therefore producing heavily subsampled raw image, which is further interpolated in the image signal processor (ISP) by the process called *demosaicing*. Despite of variety of available CFA patterns (RGBW, Fuji X-Trans, etc.), Bayer CFA remains "de facto" standard for most manufacturers for its simplicity and low cost [1]. Another underlying reason is that the majority of ISPs are manually designed and carefully tuned to process Bayer CFA, so significant effort is needed to redesign and retune those pipelines to support other CFA patterns. Nevertheless, *sub*μm image sensors adopted recently in many flagship smartphones use Quad Bayer CFA, where four pixels of one color are grouped in 2×2 cells (see Figure 1)

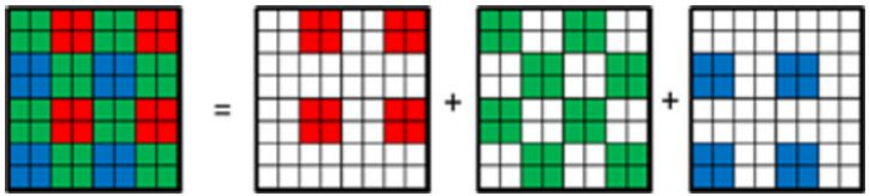


Figure 1. Tetracell or Quad Bayer color filter array decomposition into *R*, *G*, and *B* components.

These four pixels after averaging or so called *binning* allow increasing light sensitivity in low-light illumination conditions: *e.g.*, Samsung 0.8 μm sensors (ISOCELL Bright GD1, GM1) may achieve sensitivity equivalent to 1.6 μm. When four pixels are merged, Quad Bayer becomes regular Bayer CFA at quarter image resolution; therefore higher

Straightforward solution to improve quality is to increase network depth. Indeed increasing network depth level to level 30 yielded in better quality and helped to reduce artifacts. The problem with this approach was computational burden. For instance, to run inference for 48MP image we need more than 120 TOPs (Trillion OPerations), while best performing on-device Artifical Intelligence (AI) hardware can offer up to 7 TOPs. Recent approaches to network optimization can achieve up to 10 times speed-up. With reasonable budget x4 on network optimization, we need to design a network at least four times faster than EDSR.

Nevertheless, conventional methods cannot achieve good image quality and often suffer from the visual artifacts. Moreover, the majority of signal processing methods are designed assuming certain CFA: mostly standard Bayer, sometimes Fuji X-Trans; thus they cannot be easily generalized to support other CFA. Edge direction orientation detection followed by filtering, usually exploited in modern demosaicing algorithms, and needs to be redesigned. Machine learning approaches were also applied recently to demosaicing; see work based on energy minimization [25] or full ISP modeling proposed by Heide [26]. Despite of being very expensive computation-wise, they failed to produce competitive image quality.

Deep learning based methods surpassing human recognition ability for computer vision have started to gain attention for low-level vision tasks as well in the last five years. From the pioneering work of Dong for super-resolution [7], many solutions were proposed, e.g., refer to the best performing methods [27]. We review works related to image demosaicing as follows.

Several works were trying to exploit domain knowledge from conventional methods. For instance, multi-stage

Super-resolution networks were adopted by Syu for Bayer and X-Trans demosaicing in [6]: SRCNN [7] and VDSR. Syu showed that deep VDSR outperformed shallow SRCNN. Tan proposed to use residual networks for joint demosaicing and denoising or demosaicing [5]. He used plain stacked residual block architecture with injected noise channel inspired by Gharbi's approach [2]. He tried depth 20 residual network and got marginal improvement over Gharbi's work with similar complexity. Kokkinos in [30] proposed to use the iterative method using popular denoising network DnCNN [31] to ease training procedure using a small amount of training samples, but it provided only marginal improvement, with longer inference time due to 5–10 iterations (3 times versus Gharbi's).

Plain encoder–decoder architecture with symmetric skip connections (RED-Net) was proposed by Mao in [34]. RED-Net used skip connections to connect encoder and decoder parts, but their network is plain, not multi-resolution. They tried various network depths including deeper network with 30 layers (RED30). Performance of the deeper RED-Net was only marginally better than previous works.

A feature pyramid network (FPN) for object detection was proposed by Lin [35]. FPN constructs feature pyramid level by using feature maps from the network with lateral skip connections and 1×1 convolutions. The architecture of the FPN despite being somewhat similar to the DPN has different concept and purpose. It is not encoder–decoder architecture; on the contrary, constructed feature pyramid is used as input to other networks for region extraction.

Given Bayer CFA matrix H_{Bayer}:

$$H_{Bayer} = \begin{bmatrix} G & R \\ B & G \end{bmatrix}, \tag{1}$$

we can obtain the frequency structure matrix by applying symbolic DFT:

$$S_{Bayer} = DFT\left(H\|_{Bayer}\right) = \begin{bmatrix} FL & 2 \cdot FC_2 \\ -2 \cdot FC_2 & 2 \cdot FC_1 \end{bmatrix}, \tag{2}$$

Using Quad Bayer CFA matrix H_{Tetra}:

$$H_{Tetra} = \begin{bmatrix} G & G & R & R \\ G & G & R & R \\ B & B & G & G \\ B & B & G & G \end{bmatrix}, \tag{6}$$

we can obtain the frequency structure matrix for Tetracell CFA as follows:

$$S_{Tetra} = \begin{bmatrix} FL & FC_2 & 0 & FC_2 \\ -FC_2 & 0 & 0 & FC_1 \\ 0 & 0 & 0 & 0 \\ -FC_2 & 0 & 0 & 0 \end{bmatrix}. \tag{7}$$

4. PROPOSED METHOD

In our work, we use the following commonly used linear observation image model:

$$Y = M * X + \mu, \tag{8}$$

where $Y \in R^n$ is an observed raw image from the sensor, $X \in R^{3n}$ is a reconstructed RGB image, $M \in R^{n \times 3n}$ is a degradation matrix, $\mu \in R^n$ is a noise vector, and n is the number of measurements.

estimate model parameters Ω by minimizing Euclidean L_2 function.

$$L_2(\Omega) = \frac{1}{n} \sum_{i=1}^{n} \left\| F\left(X_i, \Omega\right) - Y_i \right\|_2^2. \tag{9}$$

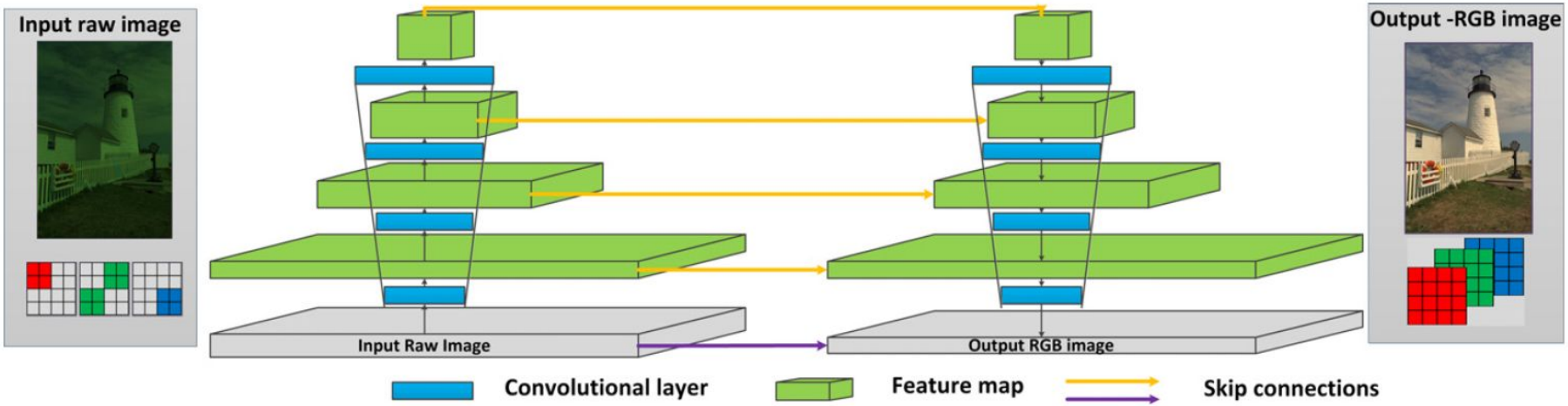


Figure 6. Duplex pyramid network: feature pyramids.

Table I. Duplex pyramid network configuration

Layer	FM1	FM2	FM3	FM4	FM5	FM6	FM7	FM8	FM9	FM10
#inputs	3	64	96	128	160	192	160	128	96	64
#outputs	64	96	128	160	192	160	128	96	64	64
Conv1	3 × 3	3 × 3	3 × 3	3 × 3	3 × 3	3 × 3	3 × 3	3 × 3	3 × 3	3 × 3
Conv2	3 × 3	3 × 3	3 × 3	3 × 3	3 × 3	3 × 3	3 × 3	3 × 3	3 × 3	3 × 3

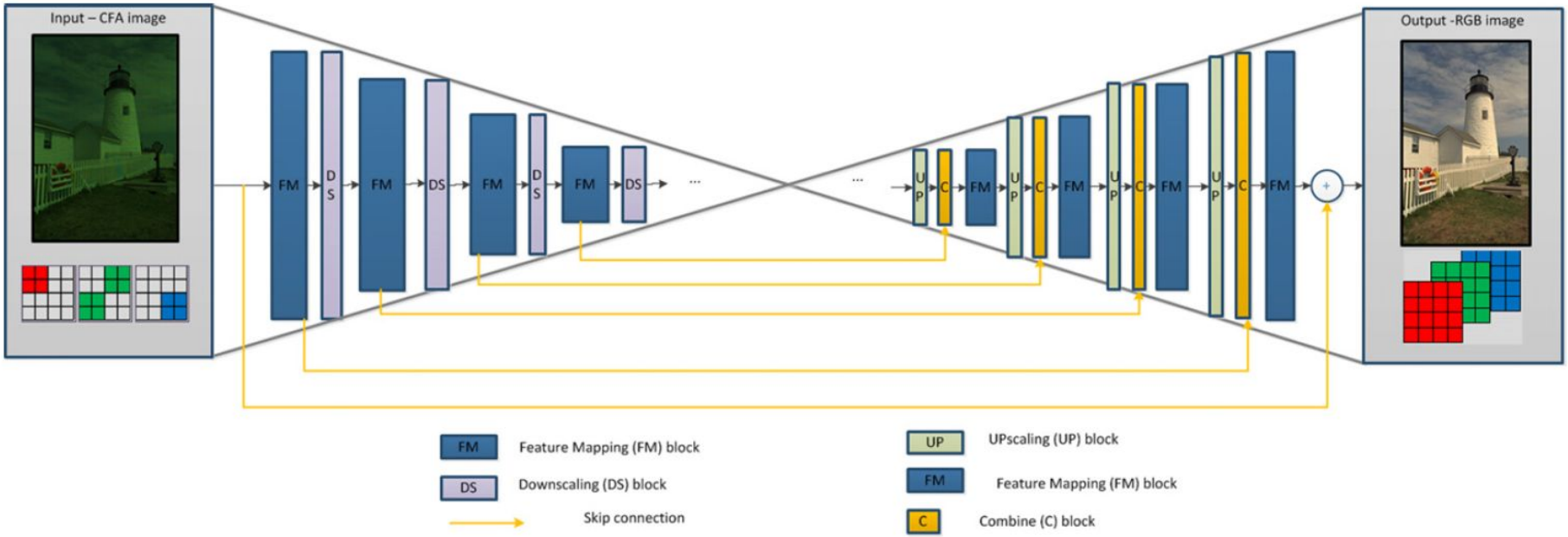


Figure 7. Block diagram of the proposed neural network.

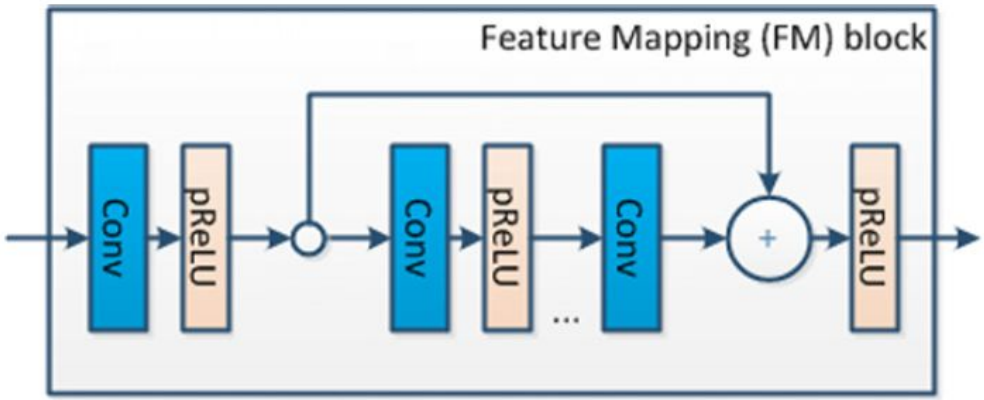


Figure 8. Block diagram of the feature mapping (FM) block.

In practice, exponential growth of feature map depth d leads to huge number of parameters when going deeper with resolution scale. To increase receptive field, we need to increase depth and resolution levels; however, for mobile applications with limited memory constraint, it becomes unfeasible: for instance, U-Net with five levels has 82 million parameters.

Indeed, for image processing applications, networks like VGG seem to be heavily over-parameterized, so some works succeed to largely reduce the number of parameters, up to 50 times [38]. If so, why do not we reduce the number of parameters in network, before doing optimization?

For low-level imaging problems like demosaicing, we propose to use a linear function:

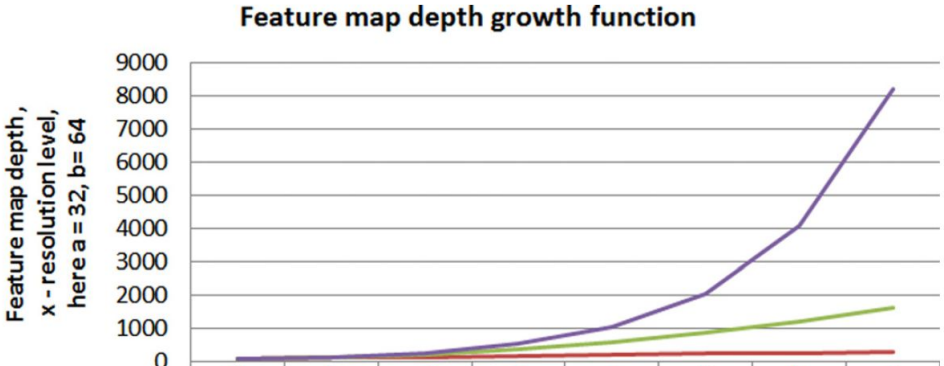
$$f(x) = ax + b, \tag{10}$$

where x is the resolution level, b is the initial feature map size, and a is the growth rate.

We suggest the practical rule of thumb for selecting lower bound of growth coefficient a:

$$a \geq b/2. \tag{11}$$

The proposed linear function with $b = 64$, $a = 32$ is shown in Figure 9. Experiments with this setting allow increasing resolution up to level 5 with affordable number of parameters and performance.



	0	1	2	3	4	5	6	7
f(x) = ax+b	64	96	128	160	192	224	256	288
f(x) = ax^2+b	64	96	192	352	576	864	1216	1632
f(x) = b*2^x	64	128	256	512	1024	2048	4096	8192

Figure 9. Feature map depth growth function: exponential versus linear.

5.1 *Reference Networks Details*

We provide details of several networks we trained to produce reference for Tetra CFA demosaicing. First we trained classical super-resolution networks SRCNN and VDSR. We adopted SRCNN and VDSR architectures as is. For residual network EDSR, we adopted single scale baseline EDSR. We removed upsampling block (pixel shuffler) from the original EDSR, because it produced visually disturbing artifacts, and processed image at full resolution.

In order to reduce memory and bandwidth we tried to use recursive architecture DRRN [13]. We trained network with one residual block, with 25 iterations. With only two convolutional layers it is very compact in terms of memory and shows good results in terms of artifacts reduction. However, it had lower CPSNR despite of using wider feature maps (128). Furthermore, DRRN can be difficult to use it due to performance issue (inference time is multiplied by the number of recursions).

For U-Net, we used original architecture except that each map was padded to keep original image resolution, we also removed last layer used for segmentation. We also pad image size after each convolution. In this article, we present results of U-Net with two levels, with 5 million parameters.

5.2 *Proposed Network Details*

Using concepts described in Section 3, we used following incarnation of the duplex pyramid network in this work. With total five resolution levels, we use same feature mapping block (FM) with two convolutions across all resolutions. The number of input/output channels changes with the proposed linear growth function across the resolution levels as shown in Table I. We use minimum kernel size 3 and stride 1 in all FM blocks across all resolution levels and parametric RELU (pRELU) as the activation function.

5.4 *Training Details*

We trained each network from scratch using sRGB image patches of size 128×128 as ground truth and mosaiced patches as input. We did not use any initial interpolation to mosaic patches, but used zero filling of each R/G/B plane. We augmented input data with random flip and rotation and random crop when possible.

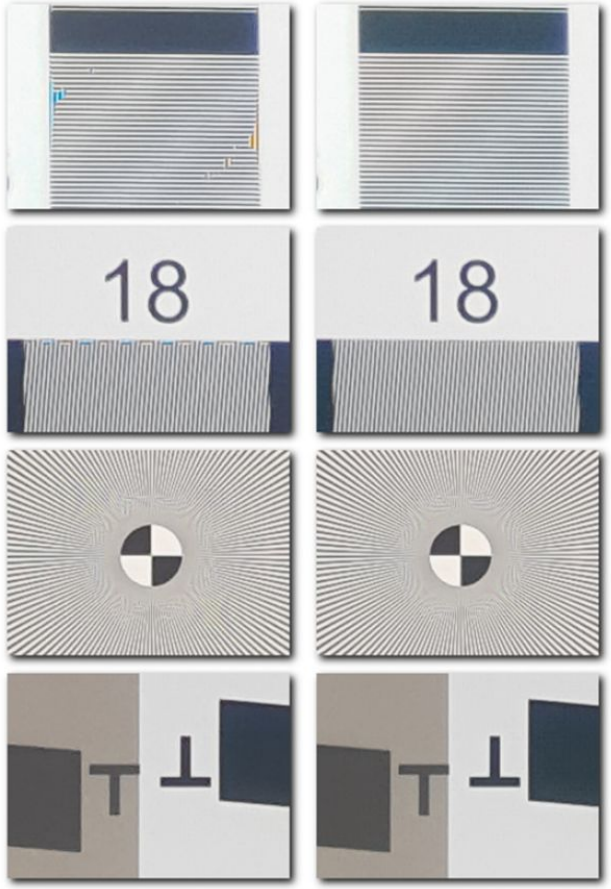


Figure 10. Artifacts mitigation on real Tetracell raw images: left—conventional, right—proposed.

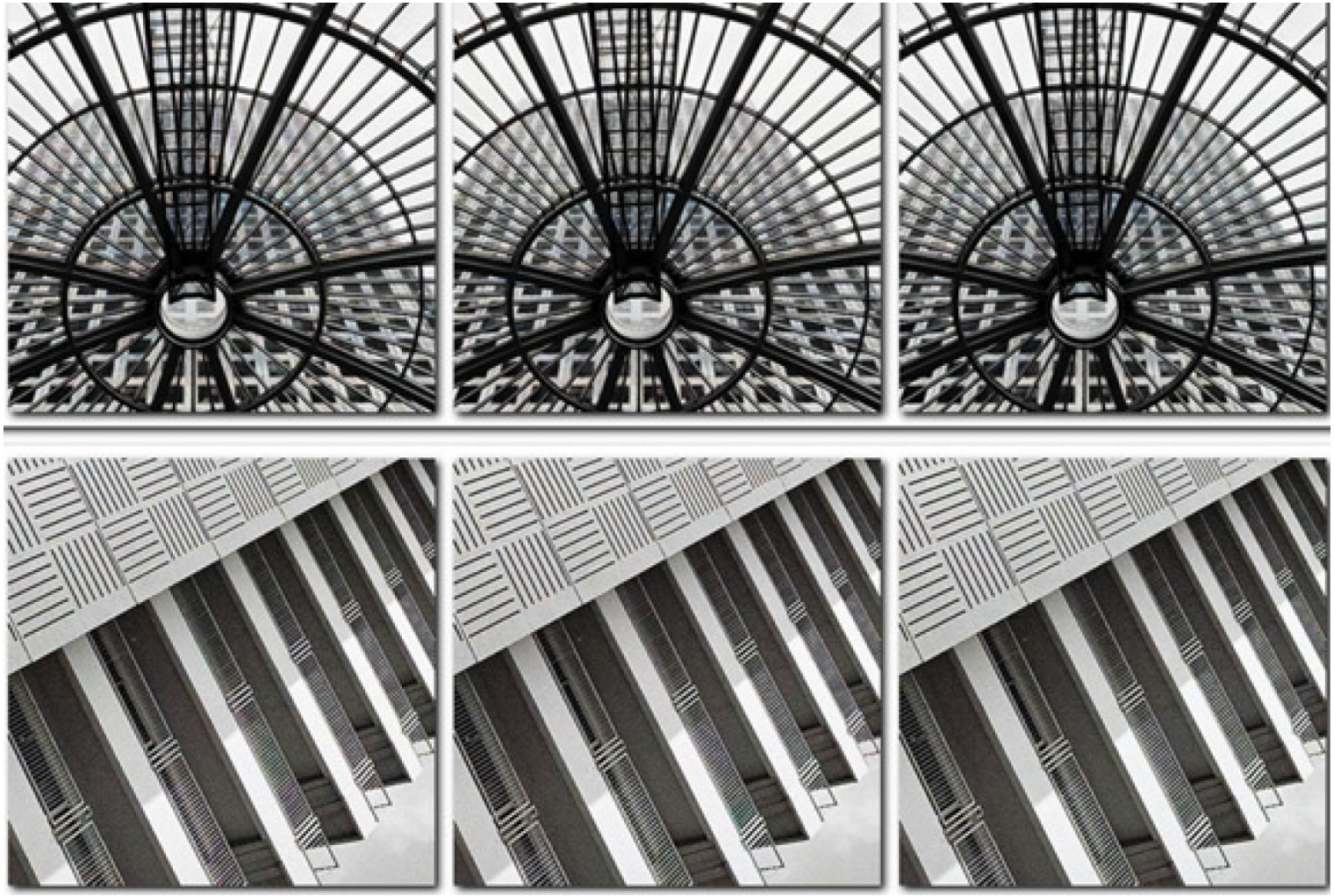


Table II. Objective quality evaluation results for Tetracell demosaicing [CPSNR/SSIM]. Best results are marked in bold.

Dataset	Metric	SRCNN	VDSR	Gharbi	EDSR	U-Net	Proposed
Kodak	CPSNR	34.6	38.0	38.4	39.6	39.8	**40.1**
	SSIM	0.9577	0.9760	0.9811	0.9836	0.9842	**0.9846**
McM	CPSNR	32.4	35.6	36.8	37.5	37.5	**37.6**
	SSIM	0.9156	0.9503	0.9589	0.9637	0.9634	**0.9842**
Moire	CPSNR	30.2	32.6	34.0	35.4	35.6	**35.9**
	SSIM	0.9021	0.9280	0.9451	0.9542	0.9562	**0.9572**
HDR-VDP	CPSNR	27.8	30.2	31.4	32.4	32.3	**32.6**
	SSIM	0.9105	0.9407	0.9567	0.9643	0.9644	**0.9655**
Urban100	CPSNR	31.1	34.7	36.4	37.3	37.4	**37.7**
	SSIM	0.9491	0.9685	0.9762	0.9794	0.9796	**0.9799**

Table III. Comparison with the state of the art for Bayer CFA.

Method	CPSNR[dB], McM dataset	CPSNR[dB], Kodak dataset
Bilinear	32.5	32.9
Hirakawa [19]	33.8	36.1
Buades [24]	35.5	37.3
Zhang [22]	36.3	37.9
Klatzer [25]	30.8	35.3
Heide [26]	38.6	40.0
Tan [5]	37.5	40.4
Gharbi [2]	39.5	41.2
DPN (ours)	**39.5**	**42.3**

Table IV. Performance estimation [TOPs]. Best number is highlighted in bold.

Metric	SRCNN	VDSR	Gharbi	EDSR	DRRN	U-Net	DPN	DPN (lite)
Number of parameters	20,099	740,736	596,099	1,259,075	301,824	5,128,064	4,984,320	4,416,128
Performance for 48MP [TOPs]	1.9	71.3	57	120.8	708	51.8	28	12.5
Performance for 36MP [TOPs]	1.4	53.2	42.8	90.6	530.6	38.8	21	9.4
Performance for 24MP [TOPs]	1.0	35.6	28.5	60.4	353.8	25.9	14	6.3
Performance for 20MP [TOPs]	0.6	22.3	17.8	37.7	221	16.2	8.8	3.9
CPSNR on Kodak dataset [dB]	34.6	38.0	38.4	39.6	39.2	39.8	**40.1**	39.6

Table V. Ablation study effect on image quality.

Factor	CPSNR [dB]	Difference [dB]
Exponential depth growth	40.1	–
Linear depth growth	40.1	0.0
No global residual learning	39.7	0.4
Using residual skip connections	39.6	0.5

We also performed ablation study to understand the effect of each improvement point. We removed global residual learning and kept other network structures same, and we also reverted residual skip connections from dense skip connections, i.e., have only summation instead of concatenations. Results are summarized in Table IV. We can conclude that combination of features contributed to the final network performance.

Questions?