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

Abstract

In this paper we describe a new mobile architecture, MobileNetV2, that improves the state of the art performance of mobile models on multiple tasks and benchmarks as well as across a spectrum of different model sizes. We also describe efficient ways of applying these mobile models to object detection in a novel framework we call SSDLite. Additionally, we demonstrate how to build mobile semantic segmentation models through a reduced form of DeepLabv3 which we call Mobile DeepLabv3.

is based on an inverted residual structure where the shortcut connections are between the thin bottleneck layers. The intermediate expansion layer uses lightweight depthwise convolutions to filter features as a source of non-linearity. Additionally, we find that it is important to remove non-linearities in the narrow layers in order to maintain representational power. We demonstrate that this improves performance and provide an intuition that led to this design.

Finally, our approach allows decoupling of the input/output domains from the expressiveness of the transformation, which provides a convenient framework for further analysis. We measure our performance on ImageNet [1] classification, COCO object detection [2], VOC image segmentation [3]. We evaluate the trade-offs between accuracy, and number of operations measured by multiply-adds (MAdd), as well as actual latency, and the number of parameters.

1. Introduction

Neural networks have revolutionized many areas of machine intelligence, enabling superhuman accuracy for challenging image recognition tasks. However, the drive to improve accuracy often comes at a cost: modern state of the art networks require high computational resources beyond the capabilities of many mobile and embedded applications.

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].

This module can be efficiently implemented using standard operations in any modern framework and allows our models to beat state of the art along multiple performance points using standard benchmarks. Furthermore, this convolutional module is particularly suitable for mobile designs, because it allows to significantly reduce the memory footprint needed during inference by never fully materializing large intermediate tensors. This reduces the need for main memory access in many embedded hardware designs, that provide small amounts of very fast software controlled cache memory.

2. Related Work

Tuning deep neural architectures to strike an optimal balance between accuracy and performance has been an area of active research for the last several years. Both manual architecture search and improvements in training algorithms, carried out by numerous teams has lead to dramatic improvements over early designs such as AlexNet [5], VGGNet [6], GoogLeNet [7]. , and ResNet [8]. Recently there has been lots of progress in algorithmic architecture exploration included hyperparameter optimization [9, 10, 11] as well as various

methods of network pruning [12, 13, 14, 15, 16, 17] and connectivity learning [18, 19]. A substantial amount of work has also been dedicated to changing the connectivity structure of the internal convolutional blocks such as in ShuffleNet [20] or introducing sparsity [21] and others [22].

Recently, [23, 24, 25, 26], opened up a new direction of bringing optimization methods including genetic algorithms and reinforcement learning to architectural search. However one drawback is that the resulting networks end up very complex. In this paper, we pursue the goal of developing better intuition about how neural networks operate and use that to guide the simplest possible network design. Our approach should be seen as complimentary to the one described in [23] and related work. In this vein our approach is similar to those taken by [20, 22] and allows to further improve the performance, while providing a glimpse on its internal operation. Our network design is based on MobileNetV1 [27]. It retains its simplicity and does not require any special operators while significantly improves its accuracy, achieving state of the art on multiple image classification and detection tasks for mobile applications.

3. Preliminaries, discussion and intuition

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.

Standard convolution takes an $h_i \times w_i \times d_i$ input tensor L_i, and applies convolutional kernel $K \in \mathcal{R}^{k \times k \times d_i \times d_j}$ to produce an $h_i \times w_i \times d_j$ output tensor L_j. Standard convolutional layers have the computational cost of $h_i \cdot w_i \cdot d_i \cdot d_j \cdot k \cdot k$.

Depthwise separable convolutions are a drop-in replacement for standard convolutional layers. Empirically they work almost as well as regular convolutions but only cost:

$$h_i \cdot w_i \cdot d_i(k^2 + d_j) \tag{1}$$

which is the sum of the depthwise and 1×1 pointwise convolutions. Effectively depthwise separable convolution reduces computation compared to traditional layers by almost a factor of k^2[1]. MobileNetV2 uses $k = 3$ (3×3 depthwise separable convolutions) so the computational cost is 8 to 9 times smaller than that of standard convolutions at only a small reduction in accuracy [27].

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.

It is easy to see that in general if a result of a layer transformation $\mathrm{ReLU}(Bx)$ has a non-zero volume S, the points mapped to $\mathrm{interior}\, S$ are obtained via a linear transformation B of the input, thus indicating that the part of the input space corresponding to the full dimensional output, is limited to a linear transformation. In other words, deep networks only have the power of a linear classifier on the non-zero volume part of the

[1] more precisely, by a factor $k^2 d_j/(k^2 + d_j)$

[2] Note that dimensionality of the manifold differs from the dimensionality of a subspace that could be embedded via a linear transformation.

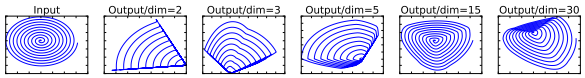


Figure 1: Examples of ReLU transformations of low-dimensional manifolds embedded in higher-dimensional spaces. In these examples the initial spiral is embedded into an n-dimensional space using random matrix T followed by ReLU, and then projected back to the 2D space using T^{-1}. In examples above $n = 2, 3$ 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.

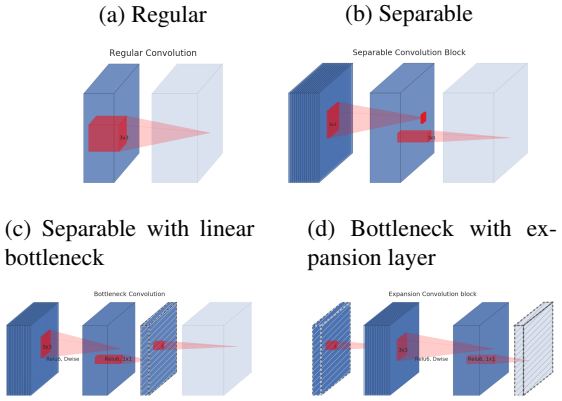


Figure 2: Evolution of separable convolution blocks. The diagonally hatched texture indicates layers that do not contain non-linearities. The last (lightly colored) layer indicates the beginning of the next block. Note: 2d and 2c are equivalent blocks when stacked. Best viewed in color.

output domain. We refer to supplemental material for a more formal statement.

On the other hand, when ReLU collapses the channel, it inevitably loses information in *that channel*. However if we have lots of channels, and there is a a structure in the activation manifold that information might still be preserved in the other channels. In supplemental materials, we show that if the input manifold can be embedded into a significantly lower-dimensional subspace of the activation space then the ReLU transformation preserves the information while introducing the needed complexity into the set of expressible functions.

To summarize, we have highlighted two properties that are indicative of the requirement that the manifold of interest should lie in a low-dimensional subspace of the higher-dimensional activation space:

1. If the manifold of interest remains non-zero volume after ReLU transformation, it corresponds to a linear transformation.

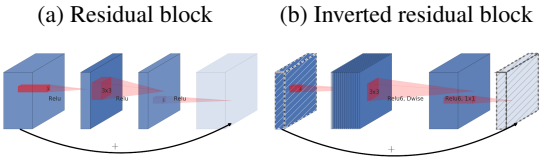


Figure 3: The difference between residual block [8, 30] and inverted residual. Diagonally hatched layers do not use non-linearities. We use thickness of each block to indicate its relative number of channels. Note how classical residuals connects the layers with high number of channels, whereas the inverted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

2. ReLU is capable of preserving complete information about the input manifold, but only if the input manifold lies in a low-dimensional subspace of the input space.

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 note that similar reports where non-linearity was helped were reported in [29] where non-linearity was removed from the input of the traditional residual block and that lead to improved performance on CIFAR dataset.

For the remainder of this paper we will be utilizing bottleneck convolutions. We will refer to the ratio between the size of the input bottleneck and the inner size as the *expansion ratio*.

3.3. Inverted residuals

The bottleneck blocks appear similar to residual block where each block contains an input followed by several bottlenecks then followed by expansion [8]. However, inspired by the intuition that the bottlenecks actually contain all the necessary information, while an expansion layer acts merely as an implementation detail that accompanies a non-linear transformation of the tensor, we use shortcuts directly between the bottlenecks.

[3]We note that in the presence of shortcuts the information loss is actually less strong.

Figure 3 provides a schematic visualization of the difference in the designs. The motivation for inserting shortcuts is similar to that of classical residual connections: we want to improve the ability of a gradient to propagate across multiplier layers. However, the inverted design is considerably more memory efficient (see Section 5 for details), as well as works slightly better in our experiments.

Running time and parameter count for bottleneck convolution The basic implementation structure is illustrated in Table 1. For a block of size $h \times w$, expansion factor t and kernel size k with d' input channels and d'' output channels, the total number of multiply add required is $h \cdot w \cdot d' \cdot t(d' + k^2 + d'')$. Compared with (1) this expression has an extra term, as indeed we have an extra 1×1 convolution, however the nature of our networks allows us to utilize much smaller input and output dimensions. In Table 3 we compare the needed sizes for each resolution between MobileNetV1, MobileNetV2 and ShuffleNet.

3.4. Information flow interpretation

One interesting property of our architecture is that it provides a natural separation between the input/output *domains* of the building blocks (bottleneck layers), and the *layer transformation* – that is a non-linear function that converts input to the output. The former can be seen as the *capacity* of the network at each layer, whereas the latter as the *expressiveness*. This is in contrast with traditional convolutional blocks, both regular and separable, where both expressiveness and capacity are tangled together and are functions of the output layer depth.

In particular, in our case, when inner layer depth is 0 the underlying convolution is the identity function thanks to the shortcut connection. When the expansion ratio is smaller than 1, this is a classical residual convolutional block [8, 30]. However, for our purposes we show that expansion ratio greater than 1 is the most useful.

This interpretation allows us to study the expressiveness of the network separately from its capacity and we believe that further exploration of this separation is warranted to provide a better understanding of the network properties.

4. Model Architecture

Now we describe our architecture in detail. As discussed in the previous section the basic building block is a bottleneck depth-separable convolution with residuals. The detailed structure of this block is shown in 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	Output
$h \times w \times k$	1x1 conv2d , ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

Table 1: *Bottleneck residual block* transforming from k to k' channels, with stride s, and expansion factor t.

With the exception of the first layer, we use constant expansion rate throughout the network. In our experiments we find that expansion rates between 5 and 10 result in nearly identical performance curves, with smaller networks being better off with slightly smaller expansion rates and larger networks having slightly better performance with larger expansion rates.

For all our main experiments we use expansion factor of 6 applied to the size of the input tensor. For example, for a bottleneck layer that takes 64-channel input tensor and produces a tensor with 128 channels, the intermediate expansion layer is then $64 \cdot 6 = 384$ channels.

Trade-off hyper parameters As in [27] we tailor our architecture to different performance points, by using the input image resolution and width multiplier as tunable hyper parameters, that can be adjusted depending on desired accuracy/performance trade-offs. Our primary network (width multiplier 1, 224×224), has a computational cost of 300 million multiply-adds and uses 3.4 million parameters. We explore the performance trade offs, for input resolutions from 96 to 224, and width multipliers of 0.35 to 1.4. The network computational cost ranges from 7 multiply adds to 585M MAdds, while the model size vary between 1.7M and 6.9M parameters.

One minor implementation difference, with [27] is that for multipliers less than one, we apply width multiplier to all layers except the very last convolutional layer. This improves performance for smaller models.

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	64/1600	16/400	32/800
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	1600K	**400K**	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 the non-bottleneck tensors.

5. Implementation Notes

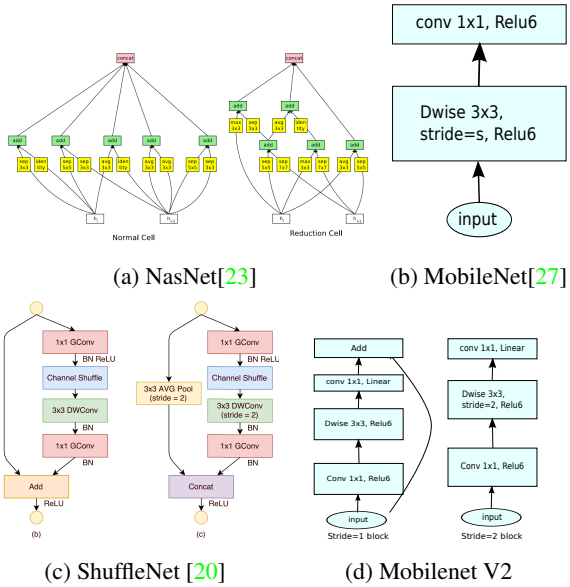
5.1. Memory efficient inference

The inverted residual bottleneck layers allow a particularly memory efficient implementation which is very important for mobile applications. A standard efficient implementation of inference that uses for instance TensorFlow[31] or Caffe [32], builds a directed acyclic compute hypergraph G, consisting of edges representing the operations and nodes representing tensors of intermediate computation. The computation is scheduled in order to minimize the total number of tensors that needs to be stored in memory. In the most general case, it searches over all plausible computation orders $\Sigma(G)$ and picks the one that minimizes

$$M(G) = \min_{\pi \in \Sigma(G)} \max_{i \in 1..n} \left[\sum_{A \in R(i,\pi,G)} |A| \right] + \text{size}(\pi_i).$$

where $R(i, \pi, G)$ is the list of intermediate tensors that are connected to any of $\pi_i \ldots \pi_n$ nodes, $|A|$ represents the size of the tensor A and $size(i)$ is the total amount of memory needed for internal storage during operation i.

For graphs that have only trivial parallel structure (such as residual connection), there is only one non-trivial feasible computation order, and thus the total amount and a bound on the memory needed for infer-



(a) NasNet[23] (b) MobileNet[27]

(c) ShuffleNet [20] (d) Mobilenet V2

Figure 4: Comparison of convolutional blocks for different architectures. ShuffleNet uses Group Convolutions [20] and shuffling, it also uses conventional residual approach where inner blocks are narrower than output. ShuffleNet and NasNet illustrations are from respective papers.