MobileNetV2: Inverted Residuals and Linear Bottlenecks

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.

在本文中，我们描述了一个新的移动架构--MobileNetV2，它提高了移动模型在多个任务和基准上的性能，并跨越了不同的模型规模的范围。我们还描述了在我们称之为SSDLite的新框架中应用这些移动模型进行物体检测的有效方法。此外，我们展示了如何通过DeepLabv3的缩小形式建立移动语义分割模型，我们称之为移动DeepLabv3是基于一个倒置的剩余结构，其中的快捷连接是在薄的瓶颈层之间。中间的扩展层使用轻量级的深度卷积来过滤特征，作为非线性的来源。此外，我们发现，为了保持表征能力，**去除窄层中的非线性非常重要**。我们证明这能提高性能，并提供了导致这种设计的直觉。

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.

最后，我们的方法允许将输入/输出**域**与转换的表达性解耦，这为进一步分析提供了一个方便的框架。我们在ImageNet[1]分类、COCO物体检测[2]、VOC图像分割[3]上衡量我们的性能。我们评估了准确度和用乘法加法（MAdd）衡量的操作数之间的权衡，以及实际延迟和参数数量。

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.

神经网络已经彻底改变了机器智能的许多领域，为具有挑战性的图像识别任务实现了超人的准确性。然而，提高准确性的动力往往是有代价的：现代最先进的网络需要高计算资源，超出了许多移动和嵌入式应用的能力。

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.

本文介绍了一种新的神经网络结构，它是专门为移动和资源受限的环境量身定做的。我们的网络推动了移动定制计算机视觉模型的技术水平，在保持相同精度的同时，大大减少了所需的操作和内存数量。

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].

我们的主要贡献是一个新的层模块：具有线性瓶颈的倒置残差。该模块将低维压缩表示作为输入，首先扩展到高维，并通过轻量级深度卷积进行过滤。随后用线性卷积将特征投射回低维表示。正式的实现是作为TensorFlow-Slim模型库的一部分，见[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.**

这个模块可以在任何现代框架中使用标准操作有效地实现，并使我们的模型在使用标准基准的多个性能点上击败最先进的水平。此外，这个卷积模块特别适用于移动设计，因为它可以通过不完全物化大型中间张量而大大减少推理过程中所需要的内存占用。这减少了许多嵌入式硬件设计中对主内存访问的需求，这些设计提供了少量非常快速的软件控制的高速缓存。

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].

在过去的几年里，**调整深度神经架构以实现准确性和性能之间的最佳平衡**一直是一个活跃的研究领域。许多团队进行的人工架构搜索和训练算法的改进都导致了对早期设计的巨大改进，如AlexNet[5]、VGGNet[6]、GoogLeNet[7]。和ResNet[8]。最近，在算法结构探索方面有很多进展，包括超参数优化[9, 10, 11]以及各种网络修剪方法[12, 13, 14, 15, 16, 17]和连接学习[18, 19]。大量的工作也**致力于**改变内部卷积块的连接结构，如ShuffleNet[20]或引入稀疏性[21]和其他[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.

最近，[23, 24, 25, 26]，**开辟**了一个新的方向，将包括遗传算法和强化学习在内的优化方法引入建筑搜索。然而，一个缺点是，所产生的网络最终会非常复杂。在本文中，我们追求的目标是发展关于神经网络如何运作的更好的直觉，并利用它来指导最简单的网络设计。我们的方法应该被看作是对
我们的方法应该被看作是与[23]和相关工作中描述的方法**相兼容的**。在这一点上，我们的方法与[20, 22]采取的方法类似，可以进一步提高性能，同时提供对其内部运作的一瞥。我们的网络设计是基于MobileNetV1[27]。它保留了它的简单性，不需要任何特殊的操作者，同时显著提高了它的准确性，在移动应用的多种图像分类和检测任务上达到了最先进的水平。

3. Preliminaries, discussion and intuition

序言、讨论和直觉

3.1. Depthwise Separable Convolutions

深度可分离的卷积

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.

深度可分离卷积是许多高效神经网络架构的一个关键构件[27, 28, 20]，我们在本工作中也使用了它们。其基本思想是用一个将卷积分成两个独立层的因子化版本来取代全卷积算子。第一层被称为深度卷积，它通过对每个输入通道应用单个卷积滤波器来进行轻量级过滤。第二层是一个1×1的卷积，称为点式卷积，负责通过计算输入通道的线性组合来建立新的特征。

Standard convolution takes an h i × w i × d i input tensor L i , and applies convolutional kernel K ∈R k×k×d i ×d j to produce an h i × w i × d j output tensor L j . Standard convolutional layers have the computational cost of h i · w i · d i · d j · k · k.

标准卷积需要一个h i × w i × d i的输入张量L i，并应用卷积核K∈R k×k×d i ×d j来产生一个h i × w i × d j的输出张量L j。标准卷积层的计算成本为h i - w i - d i - d j - k - 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:

深度可分离卷积是标准卷积层的直接替代品。从经验上看，它们的效果几乎与普通卷积层一样好，但只需要花费。

$$h_i \cdot w_i \cdot d_i(k^2 + d_j) \qquad (1)$$

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 21 . 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].

这是深度卷积和1×1点积卷积的总和。与传统层相比，深度可分离卷积有效地减少了计算量，几乎是k 21的倍数。MobileNetV2使用k=3（3×3深度可分离卷积），因此计算成本比标准卷积小8到9倍，而精度却只下降了一点[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 × w i × 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 × 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 .

考虑一个由n个层L i组成的深度神经网络，每个层都有一个尺寸为h i×w i×d i的激活张量。在本节中，我们将讨论这些激活张量的基本属性，我们将把它们视为具有d i维度的h i × w i "像素 "的容器。非正式地，对于一个真实图像的输入集，我们说层激活的集合（对于任何层L i）形成一个 "感兴趣的流形"。长期以来，人们一直认为神经网络中的兴趣流形可以嵌入到低维子空间中。换句话说，当我们看一个深度卷积层的所有单个d通道像素时，这些值所编码的信息实际上位于某个流形中，而这个流形又可以嵌入到一个低维子空间2中。

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 R n space, it generally results in a piece-wise linear curve with n-joints.

乍一看，这样的事实可以通过简单地减少一个层的维度来捕捉和利用，从而减少操作空间的维度。MobileNetV1[27]已经成功地利用了这一点，通过一个宽度乘数参数在计算和精度之间进行了有效的权衡，并且也被纳入其他网络的有效模型设计中[20]。根据这一直觉，宽度乘数方法允许人们减少激活空间的维度，直到感兴趣的流形横跨整个空间。然而，当我们回忆起深层卷积神经网络实际上有非线性的每个坐标变换，如ReLU，这种直觉就会被打破。例如，ReLU应用于一维空间中的一条线会产生一条 "射线"，而在R n空间中，它通常会产生一条有n个关节的片状线性曲线。

It is easy to see that in general if a result of a layer transformation ReLU(Bx) has a non-zero volume S, the points mapped to 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 theoutput domain. We refer to supplemental material for a more formal statement.

很容易看出，一般来说，如果一个层变换ReLU(Bx)的结果有一个非零体积S，那么映射到内部S的点是通过输入的线性变换B得到的，从而说明输入空间中对应于全维输出的部分，仅限于线性变换。换句话说，深度网络只在输出域的非零体积部分具有线性分类器的能力。关于更正式的陈述，我们参考补充材料。

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.

另一方面，当ReLU折叠通道时，它不可避免地失去了该通道的信息。然而，如果我们有很多通道，而且激活流形中存在一个结构，该信息可能仍然保留在其他通道中。在补充材料中，我们表明，如果输入流形可以被嵌入到激活空间的一个明显的低维子空间中，那么ReLU变换就可以保留信息，同时在可表达函数的集合中引入所需的复杂性。

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:

总而言之，我们强调了两个特性，它们表明了感兴趣的流形应该位于高维激活空间的低维子空间的要求。

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

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.

1.如果感兴趣的流形在ReLU变换后仍保持非零体积，则对应于一个线性变换。

2.ReLU能够保留关于输入流形的完整信息，但只有当输入流形位于输入空间的低维子空间时，才能够保留。输入空间的低维子空间。

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 nonlinearities 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.

这两个见解为我们提供了优化现有神经架构的经验提示：假设感兴趣的流形是低维的，我们可以通过在卷积块中插入线性瓶颈层来捕捉这一点。实验证据表明，使用线性层是至关重要的，因为它可以防止非线性破坏太多的信息。在第6节中，我们通过经验表明，在瓶颈层中使用非线性层确实会损害几个百分点的性能，进一步验证了我们的假设3。我们注意到，在[29]中也有类似的报告，其中非线性得到了帮助，从传统的残差块的输入中删除了非线性，这导致了CIFAR数据集的性能提高。在本文的其余部分，我们将利用瓶颈卷积。我们将把输入瓶颈的大小和内部大小之间的比率称为扩展率。

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. 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.

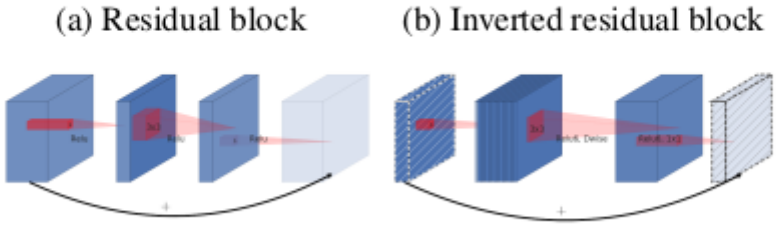


Figure 3: The difference between residual block [8, 30] and inverted residual. Diagonally hatched layers do not use non-linearities. We use thickness of each block to indicate its relative number of channels. Note how classical residuals connects the layers with high number of channels, whereas the inverted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

瓶颈区块看起来类似于残差区块，每个区块包含一个输入，然后是几个瓶颈，然后是扩展[8]。然而，受瓶颈实际上包含所有必要信息的直觉启发，而扩展层只是作为伴随着张量的非线性变换的一个实现细节，我们在瓶颈之间直接使用捷径。图3提供了一个设计差异的可视化示意图。插入捷径的动机与经典的剩余连接相似。
我们希望提高梯度在乘法器层中传播的能力。然而，倒置设计的内存效率要高得多（详见第5节），而且在我们的实验中效果也略好。

Running time and parameter count for bottleneck convolution

The basic implementation structure is illustrated in Table 1. For a block of size h × w, expansion factor t and kernel size k with d 0 input channels and d 00 output channels, the total number of multiply add required is h · w · d 0 · t(d 0 + k 2 + d 00). 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.

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.

基本的实现结构如表1所示。对于一个大小为h×w的块，扩展因子t和内核大小k，有d 0个输入通道和d 00个输出通道，所需的乘法加法总数为h - w - d 0 - t（d 0 + k 2 + d 00）。与（1）相比，这个表达式有一个额外的项，因为我们确实有一个额外的1×1卷积，然而我们网络的性质允许我们利用小得多的输入和输出尺寸。在表3中，我们比较了MobileNetV1、MobileNetV2和ShuffleNet之间每个分辨率所需的尺寸。

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.

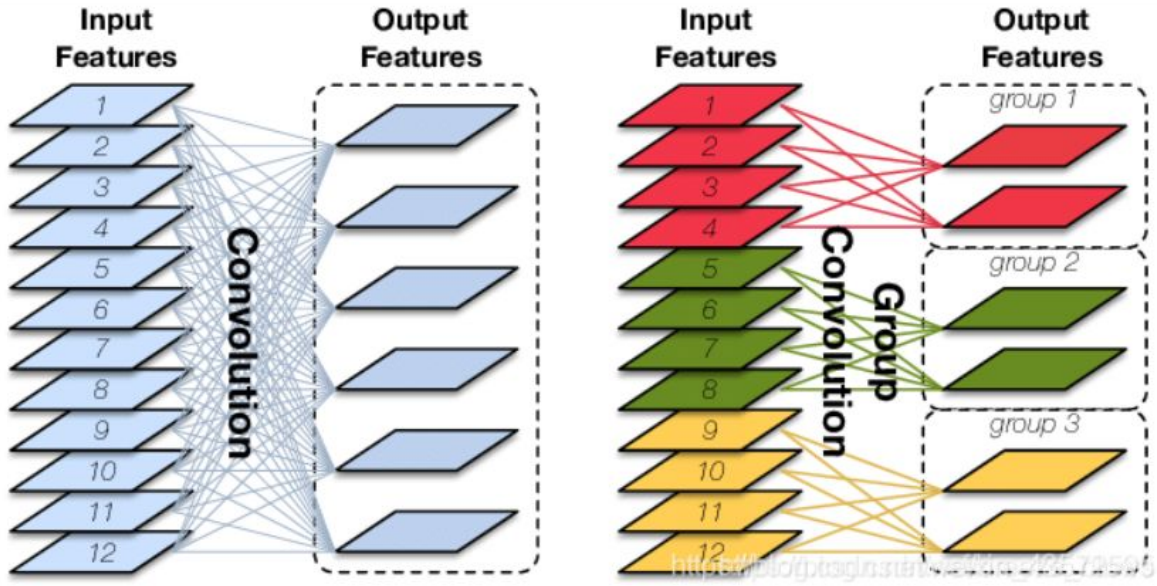
我们架构的一个有趣的特性是，它在构件（瓶颈层）的输入/输出域和层的转换之间提供了一个自然的分离--即把输入转换为输出的非线性函数。前者可以被看作是网络在每一层的能力，而后者则是表现力的体现。这与传统的卷积块形成了鲜明的对比，无论是常规的还是可分离的，表现力和容量都纠缠在一起，是输出层深度的函数。特别是，在我们的案例中，当内层深度为0时，由于捷径连接，底层卷积是身份函数。当扩展率小于1时，这是一个经典的残余卷积块[8, 30]。然而，对于我们的目的，我们表明，扩展比大于1是最有用的。这种解释使我们能够将网络的表现力与它的容量分开研究，我们认为有必要对这种分离进行进一步的探索，以便更好地理解网络的特性。

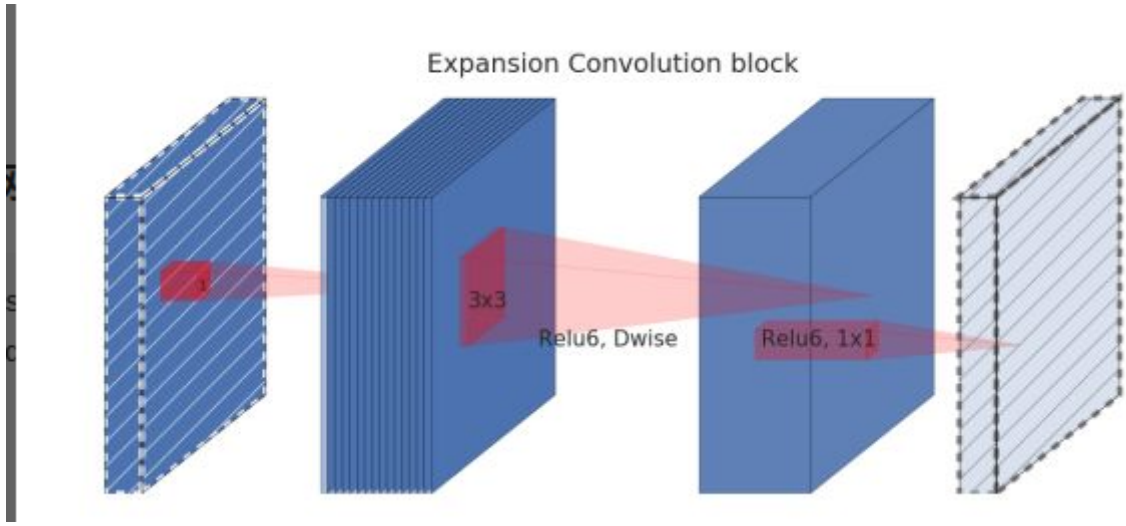
4. Model Architecture

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.

现在我们详细描述一下我们的架构。正如上一节所讨论的，基本构件是一个带残差的瓶颈深度分离卷积。这个模块的详细结构见 表1。MobileNetV2的结构包含有32个过滤器的初始完全卷积层，然后是表2中描述的19个残差瓶颈层。我们使用ReLU6作为非线性，因为它在用于低精度计算时具有鲁棒性[27]。我们总是使用内核大小为3×3的现代网络的标准，并在训练期间利用dropout和批量归一化。





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.

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.

input指该层的输入的shape，operator指该层的名字，t指逆向残差中第一个1x1普通卷积的升维倍数，c指输出的通道数，n指该层的重复次数，s指逆向残差中的depthwise conv的步距而且仅限多次重复中的第一个逆向残差。

```
from __future__ import absolute_import, division, print_function

import warnings

import torch.nn as nn

from ..module.activation import act_layers


class ConvBNReLU(nn.Sequential):
    def __init__(
        self,
        in_planes,
        out_planes,
        kernel_size=3,
        stride=1,
        groups=1,
        activation="ReLU",
    ):
        padding = (kernel_size - 1) // 2
        super(ConvBNReLU, self).__init__(
```

```
            nn.Conv2d(
                in_planes,
                out_planes,
                kernel_size,
                stride,
                padding,
                groups=groups,
                bias=False,
            ),
            nn.BatchNorm2d(out_planes),
            act_layers(activation),
        )


class InvertedResidual(nn.Module):
    def __init__(self, inp, oup, stride, expand_ratio, activation="ReLU"):
        super(InvertedResidual, self).__init__()
        self.stride = stride
        assert stride in [1, 2]

        hidden_dim = int(round(inp * expand_ratio))
        self.use_res_connect = self.stride == 1 and inp == oup

        layers = []
        if expand_ratio != 1:
            # pw
            layers.append(
                ConvBNReLU(inp, hidden_dim, kernel_size=1, activation=activation)
            )
        layers.extend(
            [
                # dw
                ConvBNReLU(
                    hidden_dim,
                    hidden_dim,
                    stride=stride,
                    groups=hidden_dim,
                    activation=activation,
                ),
                # pw-linear
                nn.Conv2d(hidden_dim, oup, 1, 1, 0, bias=False),
                nn.BatchNorm2d(oup),
            ]
        )
        self.conv = nn.Sequential(*layers)

    def forward(self, x):
        if self.use_res_connect:
            return x + self.conv(x)
        else:
            return self.conv(x)


class MobileNetV2(nn.Module):
    def __init__(
        self,
        width_mult=1.0,
        out_stages=(1, 2, 4, 6),
```

```
        last_channel=1280,
        activation="ReLU",
        act=None,
    ):
        super(MobileNetV2, self).__init__()
        # TODO: support load torchvison pretrained weight
        assert set(out_stages).issubset(i for i in range(7))
        self.width_mult = width_mult
        self.out_stages = out_stages
        input_channel = 32
        self.last_channel = last_channel
        self.activation = activation
        if act is not None:
            warnings.warn(
                "Warning! act argument has been deprecated, " "use activation
instead!"
            )
            self.activation = act
        self.interverted_residual_setting = [
            # t, c, n, s
            [1, 16, 1, 1],
            [6, 24, 2, 2],
            [6, 32, 3, 2],
            [6, 64, 4, 2],
            [6, 96, 3, 1],
            [6, 160, 3, 2],
            [6, 320, 1, 1],
        ]

        # building first layer
        self.input_channel = int(input_channel * width_mult)
        self.first_layer = ConvBNReLU(
            3, self.input_channel, stride=2, activation=self.activation
        )
        # building inverted residual blocks
        for i in range(7):
            name = "stage{}".format(i)
            setattr(self, name, self.build_mobilenet_stage(stage_num=i))

        self._initialize_weights()

    def build_mobilenet_stage(self, stage_num):
        stage = []
        t, c, n, s = self.interverted_residual_setting[stage_num]
        output_channel = int(c * self.width_mult)
        for i in range(n):
            if i == 0:
                stage.append(
                    InvertedResidual(
                        self.input_channel,
                        output_channel,
                        s,
                        expand_ratio=t,
                        activation=self.activation,
                    )
                )
            else:
                stage.append(
```

```
                    InvertedResidual(
                        self.input_channel,
                        output_channel,
                        1,
                        expand_ratio=t,
                        activation=self.activation,
                    )
                )
            self.input_channel = output_channel
        if stage_num == 6:
            last_layer = ConvBNReLU(
                self.input_channel,
                self.last_channel,
                kernel_size=1,
                activation=self.activation,
            )
            stage.append(last_layer)
        stage = nn.Sequential(*stage)
        return stage

    def forward(self, x):
        x = self.first_layer(x)
        output = []
        for i in range(0, 7):
            stage = getattr(self, "stage{}".format(i))
            x = stage(x)
            if i in self.out_stages:
                output.append(x)

        return tuple(output)

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.normal_(m.weight, std=0.001)
                if m.bias is not None:
                    m.bias.data.zero_()
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()
```

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.

除了第一层之外，我们在整个网络中使用恒定的扩展率。在我们的实验中，我们发现5到10之间的扩展率导致了几乎相同的性能曲线，较小的网络用稍小的扩展率会更好，较大的网络用较大的扩展率会有稍好的性能。在我们所有的主要实验中，我们使用适用于输入张量大小的扩展因子6。例如，对于一个接受64通道输入张量并产生128通道张量的瓶颈层，那么中间扩展层就是64 - 6 = 384通道。

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 585MMAdds, while the model size vary between 1.7M and 6.9M parameters.

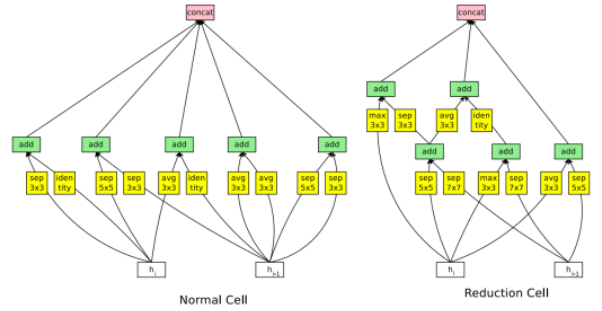
权衡超参数 在[27]中，我们通过使用输入图像的分辨率和宽度乘法器作为可调整的超参数，使我们的架构适应不同的性能点，这些参数可以根据所需的精度/性能权衡进行调整。我们的主要网络（宽度倍增器1，224×224）的计算成本为3亿次乘法，使用340万个参数。我们探索了输入分辨率从96到224，宽度乘数从0.35到1.4的性能权衡。网络计算成本从7个乘法运算到585MMAdds不等，而模型大小在1.7M和6.9M参数之间。

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

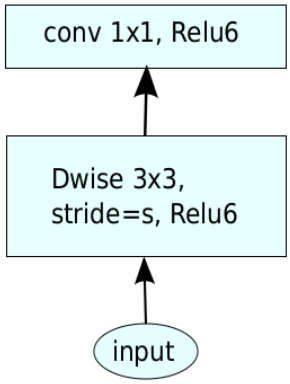
与[27]在实现上的一个小区别是，对于乘数小于1的情况，我们对所有的层都应用了宽度倍数，除了最后的卷积层。这提高了小模型的性能。

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

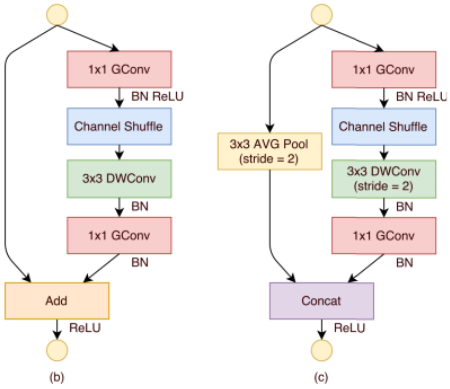
Table 3: The max number of channels/memory (in Kb) that needs to be materialized at each spatial resolution for different architectures. We assume 16-bit floats for activations. For ShuffleNet, we use $2x, g = 3$ that matches the performance of MobileNetV1 and MobileNetV2. For the first layer of MobileNetV2 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.



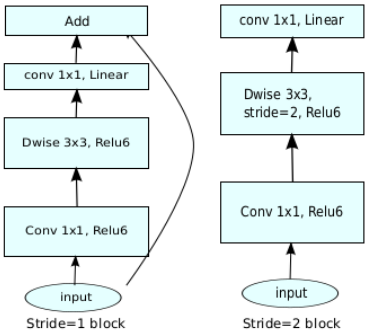
(a) NasNet[23]



(b) MobileNet[27]



(c) ShuffleNet [20]



(d) Mobilenet V2