ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices

Abstract

We introduce an extremely computation-efficient CNN architecture named ShuffleNet, which is designed specially for mobile devices with very limited computing power (e.g., 10-150 MFLOPs). The new architecture utilizes two new operations, pointwise group convolution and channel shuffle, to greatly reduce computation cost while maintaining accuracy. Experiments on ImageNet classification and MS COCO object detection demonstrate the superior perfor□mance of ShuffleNet over other structures, e.g. lower top-1 error (absolute 7.8%) than recent MobileNet [12] on Ima□geNet classification task, under the computation budget of 40 MFLOPs. On an ARM-based mobile device, ShuffleNet achieves ~13× actual speedup over AlexNet while maintaining comparable accuracy.

我们介绍了一种计算效率极高的CNN架构，该架构是专门为计算能力非常有限的移动设备（如：移动电话）设计的，名为ShuffleNet。
它是专门为计算能力非常有限的移动设备（例如。10-150 MFLOPs）。这个新的架构采用了两种新的操作，即pointwise 分组卷积和信道洗牌，以大大降低计算成本，同时保持准确性。在ImageNet分类和MSCOCO对象检测的实验证明了ShuffleNet比其他结构更优越的性能，例如，较低的top-1误差（绝对值为7.8%）比最近的MobileNet[12]在ImageNet分类任务上更低，在计算预算为40 MFLOPs。在一个基于ARM的移动设备上，ShuffleNet实现了比AlexNet高出13倍的实际速度，同时保持了相当的准确性。

1. Introduction

Building deeper and larger convolutional neural networks (CNNs) is a primary trend for solving major visual recognition tasks [21, 9, 33, 5, 28, 24]. The most accurate CNNs usually have hundreds of layers and thousands of channels [9, 34, 32, 40], thus requiring computation at billions of FLOPs. This report examines the opposite extreme: pursuing the best accuracy in very limited computational budgets at tens or hundreds of MFLOPs, focusing on common mobile platforms such as drones, robots, and smartphones. Note that many existing works [16, 22, 43, 42, 38, 27] focus on pruning, compressing, or low-bit representing a “basic” network architecture. Here we aim to explore a highly efficient basic architecture specially designed for our desired computing ranges.

建立更深更大的卷积神经网络（CNN）是解决主要视觉识别任务的主要趋势[21, 9, 33, 5, 28, 24]。最准确的CNN通常有数百个层和数千个通道[9, 34, 32, 40]，因此需要数十亿FLOPs的计算。 本报告研究了相反的极端：在非常有限的计算预算中，以几十或几百MFLOPs的速度追求最佳精度，重点是常见的移动平台，如无人机、机器人和智能手机。请注意，许多现有的工作[16, 22, 43, 42, 38, 27]专注于修剪、压缩或低位表示 "基本 "网络架构。在这里，我们的目标是探索一个专门为我们所期望的计算范围设计的高效的基本架构。

We notice that state-of-the-art basic architectures such as Xception [3] and ResNeXt [40] become less efficient in extremely small networks because of the costly dense 1 × 1 convolutions. We propose using pointwise group convolutions to reduce computation complexity of 1 × 1 convolutions. To overcome the side effects brought by group convolutions, we come up with a novel channel shuffle operation to help the information flowing across feature channels. Based

on the two techniques, we build a highly efficient architecture called ShuffleNet. Compared with popular structures like [30, 9, 40], for a given computation complexity budget, our ShuffleNet allows more feature map channels, which helps to encode more information and is especially critical to the performance of very small networks.

我们注意到，最先进的基本架构，如Xception[3]和ResNeXt[40]，在极小的网络中，由于昂贵的密集的1×1卷积，效率变得较低。我们建议使用点式分组卷积来降低1×1卷积的计算复杂性。为了克服分组卷积带来的副作用，我们提出了一个新的通道洗牌操作，以帮助信息在特征通道间流动。基于这两种技术，我们建立了一个高效的架构，称为ShuffleNet。与流行的结构如[30, 9, 40]相比，在给定的计算复杂度预算下，我们的ShuffleNet允许更多的特征图通道，这有助于编码更多的信息，对非常小的网络的性能尤为关键。

We evaluate our models on the challenging ImageNet classification [4, 29] and MS COCO object detection [23] tasks. A series of controlled experiments shows the effectiveness of our design principles and the better performance over other structures. Compared with the state-of-the-art architecture MobileNet [12], ShuffleNet achieves superior performance by a significant margin, e.g. absolute 7.8% lower ImageNet top-1 error at level of 40 MFLOPs.

我们在具有挑战性的ImageNet分类[4, 29]和MS COCO物体检测[23]任务上评估了我们的模型。一系列的控制性实验显示了我们设计原则的有效性和比其他结构更好的性能。与最先进的结构MobileNet[12]相比，ShuffleNet取得了显著的性能优势，例如，在40MFLOPs的水平上，ImageNet的top-1误差绝对低7.8%。

We also examine the speedup on real hardware, i.e. an off-the-shelf ARM-based computing core. The ShuffleNet model achieves 13× actual speedup (theoretical speedup is 18×) over AlexNet [21] while maintaining comparable accuracy.

我们还研究了在实际硬件上的速度提升，即一个现成的基于ARM的计算核心。ShuffleNet模型比AlexNet[21]实现了13倍的实际速度提升（理论速度提升为18倍），同时保持了相当的准确性。

2. Related Work

Efficient Model Designs

The last few years have seen the success of deep neural networks in computer vision tasks [21, 36, 28], in which model designs play an important role. The increasing needs of running high quality deep neural networks on embedded devices encourage the study on efficient model designs [8]. For example, GoogLeNet [33] increases the depth of networks with much lower complexity compared to simply stacking convolution layers. SqueezeNet [14] reduces parameters and computation significantly while maintaining accuracy. ResNet [9, 10] utilizes the efficient bottleneck structure to achieve impressive performance. SENet [13] introduces an architectural unit that boosts performance at slight computation cost. Concurrent with us, a very recent work [46] employs reinforcement learning and model search to explore efficient model designs. The proposed mobile NASNet model achieves comparable performance with our counterpart ShuffleNet model (26.0% @ 564 MFLOPs vs. 26.3% @ 524 MFLOPs for ImageNet classification error). But [46] do not report results on extremely tiny models (e.g. complexity less than 150 MFLOPs), nor evaluate the actual inference time on mobile devices.

在过去的几年中，深度神经网络在计算机视觉任务中取得了成功[21, 36, 28]，其中模型设计发挥了重要作用。在嵌入式设备上运行高质量的深度神经网络的需求越来越大，这鼓励了对高效模型设计的研究[8]。例如，GoogLeNet[33]增加了网络的深度，与简单堆叠卷积层相比，其复杂度要低得多。SqueezeNet[14]在保持精度的同时，大大减少了参数和计算量。ResNet[9, 10]利用高效的瓶颈结构实现了惊人的性能。SENet[13]引入了一个架构单元，以轻微的计算成本提升了性能。与我们同时，最近的一项工作[46]采用了强化学习和模型搜索来探索高效的模型设计。提出的移动NASNet模型取得了与我们的对应ShuffleNet模型相当的性能（26.0% @ 564 MFLOPs vs. 26.3% @ 524 MFLOPs的ImageNet分类错

误）。但是[46]并没有报告极小的模型的结果（例如复杂性低于150 MFLOPs），也没有评估移动设备上的实际推理时间。

Group Convolution

The concept of group convolution, which was first introduced in AlexNet [21] for distributing the model over two GPUs, has been well demonstrated its effectiveness in ResNeXt [40]. Depthwise separable convolution proposed in Xception [3] generalizes the ideas of separable convolutions in Inception series [34, 32]. Recently, MobileNet [12] utilizes the depthwise separable convolutions and gains state-of-the-art results among lightweight models. Our work generalizes group convolution and depthwise separable convolution in a novel form. Channel Shuffle Operation To the best of our knowledge, the idea of channel shuffle operation is rarely mentioned in previous work on efficient model design, although CNN library cuda-convnet [20] supports “random sparse convolution” layer, which is equivalent to random channel shuffle followed by a group convolutional layer. Such “random shuffle” operation has different purpose and been seldom exploited later. Very recently, another concurrent work [41] also adopt this idea for a two-stage convolution. However, [41] did not specially investigate the effectiveness of channel shuffle itself and its usage in tiny model design.

组卷积的概念是在AlexNet[21]中首次引入的，用于将模型分布在两个GPU上，在ResNeXt[40]中已经充分证明了其有效性。Xception[3]中提出的深度可分离卷积概括了Inception系列[34, 32]中可分离卷积的思想。最近，MobileNet[12]利用了深度可分离卷积，在轻量级模型中获得了最先进的结果。我们的工作以一种新的形式概括了分组卷积和纵深可分离卷积。通道洗牌操作 据我们所知，通道洗牌操作的想法在以前的高效模型设计工作中很少被提及，尽管CNN库cuda-convnet[20]支持 "随机稀疏卷积 "层，这相当于随机通道洗牌，然后是一个组卷积层。这种 "随机洗牌 "操作有不同的目的，后来很少被利用。最近，另一个同时进行的工作[41]也采用了这个想法来实现两级卷积。然而，[41]并没有专门研究通道洗牌本身的有效性以及它在微小模型设计中的使用。

Model Acceleration

This direction aims to accelerate inference while preserving accuracy of a pre-trained model. Pruning network connections [6, 7] or channels [38] reduces redundant connections in a pre-trained model while maintaining performance. Quantization [31, 27, 39, 45, 44] and factorization [22, 16, 18, 37] are proposed in literature to reduce redundancy in calculations to speed up inference. Without modifying the parameters, optimized convolution algorithms implemented by FFT [25, 35] and other methods [2] decrease time consumption in practice. Distilling [11] transfers knowledge from large models into small ones, which makes training small models easier.

这个方向旨在加速推理，同时保持预训练模型的准确性。修剪网络连接[6, 7]或通道[38]可以减少预训练模型中的冗余连接，同时保持性能。文献中提出量化[31, 27, 39, 45, 44]和因子化[22, 16, 18, 37]来减少计算中的冗余，以加快推理速度。在不修改参数的情况下，由FFT[25, 35]和其他方法[2]实现的优化卷积算法在实践中减少了时间消耗。Distilling[11]将大模型中的知识转移到小模型中，这使得小模型的训练更加容易。

3. Approach

3.1. Channel Shuffle for Group Convolutions

Modern convolutional neural networks [30, 33, 34, 32, 9, 10] usually consist of repeated building blocks with the same structure. Among them, state-of-the-art networks such as Xception [3] and ResNeXt [40] introduce efficient depthwise separable convolutions or group convolutions into the building blocks to strike an excellent trade-off between representation capability and

computational cost. However, we notice that both designs do not fully take the 1 × 1 convolutions (also called pointwise convolutions in [12]) into account, which require considerable complexity. For example, in ResNeXt [40] only 3 × 3 layers are equipped with group convolutions. As a result, for each residual unit in ResNeXt the pointwise convolutions occupy 93.4% multiplication-adds (cardinality = 32 as suggested in [40]). In tiny networks, expensive pointwise convolutions result in limited number of channels to meet the complexity constraint, which might significantly damage the accuracy.

现代卷积神经网络[30, 33, 34, 32, 9, 10]通常由具有相同结构的重复构建块组成。其中，最先进的网络如Xception[3]和ResNeXt[40]在构建模块中引入了高效的深度可分离卷积或分组卷积，在表示能力和计算成本之间取得了很好的平衡。然而，我们注意到，这两种设计都没有完全考虑到1×1的卷积（在[12]中也被称为点状卷积），这需要相当的复杂性。例如，在ResNeXt[40]中，只有3×3层配备了分组卷积。因此，对于ResNeXt中的每个残余单元，点式卷积占据了93.4%的乘法加成（如[40]中建议的cardinality = 32）。在微小的网络中，昂贵的点式卷积会导致有限的通道数量来满足复杂度的限制，这可能会大大损害准确性。

To address the issue, a straightforward solution is to apply channel sparse connections, for example group convolutions, also on 1 × 1 layers. By ensuring that each convolution operates only on the corresponding input channel group, group convolution significantly reduces computation cost. However, if multiple group convolutions stack together, there is one side effect: outputs from a certain channel are only derived from a small fraction of input channels. Fig 1 (a) illustrates a situation of two stacked group convolution layers. It is clear that outputs from a certain group only relate to the inputs within the group. This property blocks information flow between channel groups and weakens representation.

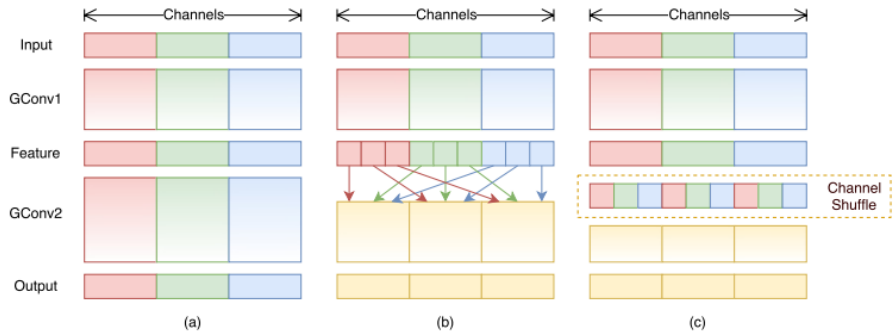

Figure 1. Channel shuffle with two stacked group convolutions. GConv stands for group convolution. a) two stacked convolution layers with the same number of groups. Each output channel only relates to the input channels within the group. No cross talk; b) input and output channels are fully related when GConv2 takes data from different groups after GConv1; c) an equivalent implementation to b) using channel shuffle.

为了解决这个问题，一个直接的解决方案是应用通道稀疏连接，例如组卷积，也是在1×1层。通过确保每个卷积只对相应的输入通道组进行操作，组卷积大大降低了计算成本。然而，如果多个组卷积在一起，有一个副作用：某个通道的输出只来自一小部分输入通道。图1（a）说明了两个堆叠的群卷积层的情况。很明显，某个组的输出只与该组内的输入有关。这一特性阻碍了通道组之间的信息流动，削弱了代表性。

If we allow group convolution to obtain input data from different groups (as shown in Fig 1 (b)), the input and output channels will be fully related. Specifically, for the feature map generated from the previous group layer, we can first divide the channels in each group into several subgroups, then feed each group in the next layer with different subgroups. This can be efficiently and elegantly implemented by a channel shuffle operation (Fig 1 (c)): suppose a convolutional layer with g groups whose output has g × n channels; we first reshape the output channel dimension into (g, n), transposing and then flattening it back as the input of next layer. Note that the operation still takes effect even if the two convolutions have different numbers of groups.

Moreover, channel shuffle is also differentiable, which means it can be embedded into network structures for end-to-end training.

如果我们允许分组卷积从不同的组获得输入数据（如图1（b）所示），输入和输出通道将完全相关。具体来说，对于前一个组层产生的特征图，我们可以先将每个组中的通道分成几个子组，然后在下一层中用不同的子组来输入每个组。这可以通过通道洗牌操作有效而优雅地实现（图1（c））：假设一个有g个组的卷积层，其输出有g×n个通道；我们首先将输出的通道维度重塑为（g，n），转置后再将其平移回作为下一层的输入。请注意，即使两个卷积的组数不同，该操作仍会生效。此外，通道洗牌也是可分的，这意味着它可以被嵌入到网络结构中进行端到端的训练。

Channel shuffle operation makes it possible to build more powerful structures with multiple group convolutional layers. In the next subsection we will introduce an efficient network unit with channel shuffle and group convolution.

通道洗牌操作使得建立具有多个组卷积层的更强大结构成为可能。在下一小节中，我们将介绍一个具有通道洗牌和群卷积的高效网络单元。

3.2. ShuffleNet Unit

Taking advantage of the channel shuffle operation, we propose a novel ShuffleNet unit specially designed for small networks. We start from the design principle of bottleneck unit [9] in Fig 2 (a). It is a residual block. In its residual branch, for the 3 × 3 layer, we apply a computational economical 3 × 3 depthwise convolution [3] on the bottleneck feature map. Then, we replace the first 1 × 1 layer with pointwise group convolution followed by a channel shuffle operation, to form a ShuffleNet unit, as shown in Fig 2 (b). The purpose of the second pointwise group convolution is to recover the channel dimension to match the shortcut path. For simplicity, we do not apply an extra channel shuffle operation after the second pointwise layer as it results in comparable scores. The usage of batch normalization (BN) [15] and nonlinearity is similar to [9, 40], except that we do not use ReLU after depthwise convolution as suggested by [3]. As for the case where ShuffleNet is applied with stride, we simply make two modifications (see Fig 2 (c)):

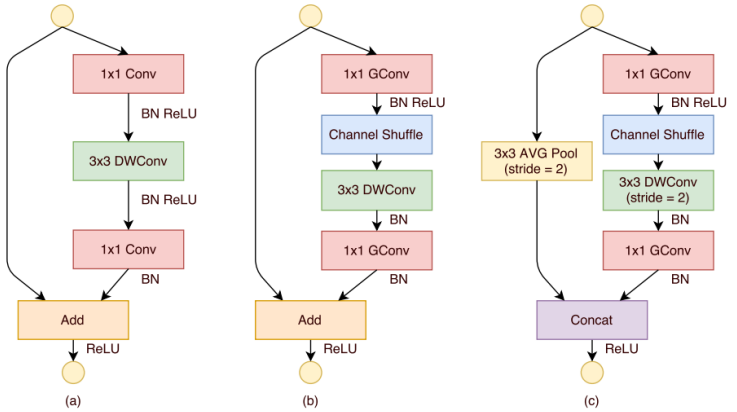

Figure 2. ShuffleNet Units. a) bottleneck unit [9] with depthwise convolution (DWConv) [3, 12]; b) ShuffleNet unit with pointwise group convolution (GConv) and channel shuffle; c) ShuffleNet unit with stride = 2.

利用信道洗牌操作的优势，我们提出了一种专门为小型网络设计的新型ShuffleNet单元。我们从图2（a）中的瓶颈单元[9]的设计原理出发。它是一个剩余块。在它的剩余分支中，对于3×3层，我们在瓶颈特征图上应用了一个计算经济的3×3深度卷积[3]。然后，我们将第一个1×1层替换为顺时针分组卷积，然后进行通道洗牌操作，形成一个ShuffleNet单元，如图2（b）所示。第二个点群卷积的目的是恢复信道维度以匹配捷径路径。为了简单起见，我们不在第二层顺时针分组后应用额外的信道洗牌操作，因为这样做的结果是分数相当的。批量归一化（BN）[15]和非线性的用法与[9, 40]类似，只是我们没有按照[3]的建议在深度卷积后使用ReLU。至于ShuffleNet的应用，我们只是做了两个修改（见图2（c））。

(i) add a 3 × 3 average pooling on the shortcut path;

(ii) replace the element-wise addition with channel concatenation, which makes it easy to enlarge channel dimension with little extra computation cost.

(i) 在捷径路径上增加一个3×3的平均池。

(ii)用通道连接法取代元素相加法，这样就可以很容易地扩大通道维度，而没有什么额外的计算成本。

Thanks to pointwise group convolution with channel shuffle, all components in ShuffleNet unit can be computed efficiently. Compared with ResNet [9] (bottleneck design) and ResNeXt [40], our structure has less complexity under the same settings. For example, given the input size c × h × w and the bottleneck channels m, ResNet unit requires hw(2cm + 9m2) FLOPs and ResNeXt has hw(2cm + 9m2/g) FLOPs, while our ShuffleNet unit requires only hw(2cm/g + 9m) FLOPs, where g means thnumber of groups for convolutions. In other words, given a computational budget, ShuffleNet can use wider feature maps. We find this is critical for small networks, as tiny networks usually have an insufficient number of channels to process the information.

由于采用了带有通道洗牌的点群卷积，ShuffleNet单元中的所有组件都可以被有效地计算出来。与ResNet[9]（瓶颈设计）和ResNeXt[40]相比，我们的结构在相同的设置下具有较低的复杂性。例如，给定输入尺寸c×h×w和瓶颈通道m，ResNet单元需要hw(2cm+9m2)FLOPs，ResNeXt有hw(2cm+9m2/g)FLOPs，而我们的ShuffleNet单元只需要hw(2cm/g+9m)FLOPs，其中g表示卷积的组数。换句话说，在计算预算的情况下，ShuffleNet可以使用更广泛的特征图。我们发现这对小型网络至关重要，因为小型网络通常没有足够的通道来处理信息。

In addition, in ShuffleNet depthwise convolution only performs on bottleneck feature maps. Even though depthwise convolution usually has very low theoretical complexity, we find it difficult to efficiently implement on lowpower mobile devices, which may result from a worse computation/memory access ratio compared with other dense operations. Such drawback is also referred in [3], which has a runtime library based on TensorFlow [1]. In ShuffleNet units, we intentionally use depthwise convolution only on bottleneck in order to prevent overhead as much as possible.

此外，在ShuffleNet中，纵深卷积只在瓶颈特征图上执行。尽管深度卷积通常具有很低的理论复杂度，但我们发现它很难在低功率的移动设备上有效实现，这可能是由于与其他密集操作相比，计算/内存访问率更差。这样的缺点在[3]中也有提及，它有一个基于TensorFlow[1]的运行时库。在ShuffleNet单元中，我们有意只在瓶颈处使用深度卷积，以尽可能地防止开销。

3.3. Network Architecture

Built on ShuffleNet units, we present the overall ShuffleNet architecture in Table 1. The proposed network is mainly composed of a stack of ShuffleNet units grouped into three stages. The first building block in each stage is applied with stride = 2. Other hyper-parameters within a stage stay the same, and for the next stage the output channels are doubled. Similar to [9], we set the number of bottleneck channels to 1/4 of the output channels for each ShuffleNet unit. Our intent is to provide a reference design as simple as possible, although we find that further hyper-parameter tunning might generate better results. In ShuffleNet units, group number g controls the connec□tion sparsity of pointwise convolutions. Table 1 explores different group numbers and we adapt the output channels to ensure overall computation cost roughly unchanged (140 MFLOPs). Obviously, larger group numbers result in more output channels (thus more convolutional filters) for a given complexity constraint, which helps to encode more information, though it might also lead to degradation for an individual convolutional filter due to limited corresponding input channels. In Sec 4.1.1 we will study the impact of this number subject to different computational constrains.

基于ShuffleNet单元，我们在表1中介绍了ShuffleNet的整体架构。拟议的网络主要由ShuffleNet单元的堆栈组成，分为三个阶段。每个阶段的第一个构件的跨度=2。一个阶段内的其他超参数保持不变，而对于下一个阶段，输出通道则增加一倍。与[9]类似，我们将每个ShuffleNet单元的瓶颈通道数量设置为输出通道的1/4。我们的意图是提供一个尽可能简单的参考设计，尽管我们发现进一步的超参数调整可能产生更好的结果。在ShuffleNet单元中，组数g控制着点状卷积的连接稀疏度。表1探讨了不同的组数，我们调整了输出通道，以确保整体计算成本大致不变（140 MFLOPs）。显然，在给定的复杂度约束下，较大的组数会导致更多的输出通道（因此有更多的卷积过滤器），这有助于编码更多的信息，不过由于相应的输入通道有限，也可能导致单个卷积过滤器的性能下降。在第4.1.1节中，我们将研究这个数字在不同计算约束下的影响。

To customize the network to a desired complexity, we can simply apply a scale factor s on the number of chan□nels. For example, we denote the networks in Table 1 as ”ShuffleNet 1×”, then ” ShuffleNet s×” means scaling the number of filters in ShuffleNet 1× by s times thus overall complexity will be roughly s 2 times of ShuffleNet 1×.

为了将网络定制为所需的复杂度，我们可以简单地在通道的数量上应用一个比例系数s。例如，我们将表1中的网络表示为 "ShuffleNet 1×"，那么 "ShuffleNet s×"意味着将ShuffleNet 1×中的过滤器数量放大s倍，因此整体复杂度大约是ShuffleNet 1×的s 2倍。

Layer	Output size	KSize	Stride	Repeat	Output channels (g groups)				
					$g=1$	$g=2$	$g=3$	$g=4$	$g=8$
Image	224×224				3	3	3	3	3
Conv1	112×112	3×3	2	1	24	24	24	24	24
MaxPool	56×56	3×3	2						
Stage2	28×28		2	1	144	200	240	272	384
	28×28		1	3	144	200	240	272	384
Stage3	14×14		2	1	288	400	480	544	768
	14×14		1	7	288	400	480	544	768
Stage4	7×7		2	1	576	800	960	1088	1536
	7×7		1	3	576	800	960	1088	1536
GlobalPool	1×1	7×7							
FC					1000	1000	1000	1000	1000
Complexity					143M	140M	137M	133M	137M

Table 1. ShuffleNet architecture. The complexity is evaluated with FLOPs, i.e. the number of floating-point multiplication-adds. Note that for Stage 2, we do not apply group convolution on the first pointwise layer because the number of input channels is relatively small.

Model	Complexity (MFLOPs)	Classification error (%)				
		$g=1$	$g=2$	$g=3$	$g=4$	$g=8$
ShuffleNet 1×	140	33.6	32.7	32.6	32.8	**32.4**
ShuffleNet 0.5×	38	45.1	44.4	43.2	**41.6**	42.3
ShuffleNet 0.25×	13	57.1	56.8	55.0	54.2	**52.7**

Table 2. Classification error vs. number of groups g (*smaller number represents better performance*)

- 每个阶段的第一个block的步长为2，下一阶段的通道翻倍
- 每个阶段内的除步长其他超参数保持不变
- 每个ShuffleNet unit的bottleneck通道数为输出的1/4(和ResNet设置一致)

```
def shuffle(x, groups):
    N, C, H, W = x.size()
    out = x.view(N, groups, C // groups, H, W).permute(0, 2, 1, 3,
4).contiguous().view(N, C, H, W)

    return out
class Bottleneck(nn.Module):
    def __init__(self, in_channels, out_channels, stride, groups):
        super().__init__()
        mid_channles = int(out_channels/4)
        if in_channels == 24:
            self.groups = 1
```

```
        else:
            self.groups = groups
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, mid_channles, 1, groups=self.groups,
bias=False),
            nn.BatchNorm2d(mid_channles),
            nn.ReLU(inplace=True)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(mid_channles, mid_channles, 3, stride=stride, padding=1,
groups=mid_channles, bias=False),
            nn.BatchNorm2d(mid_channles),
            nn.ReLU(inplace=True),
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(mid_channles, out_channels, 1, groups=groups, bias=False),
            nn.BatchNorm2d(out_channels)
        )
        self.shortcut = nn.Sequential(nn.AvgPool2d(3, stride=2, padding=1))
        self.stride = stride
    def forward(self, x):
        out = self.conv1(x)
        out = shuffle(out, self.groups)
        out = self.conv2(out)
        out = self.conv3(out)
        if self.stride == 2:
            res = self.shortcut(x)
            out = F.relu(torch.cat([out, res], 1))
        else:
            out = F.relu(out+x)
        return out

class ShuffleNet(nn.Module):
    def __init__(self, groups, channel_num, class_num=settings.CLASSES_NUM):
        super().__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 24, 3, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(24),
            nn.ReLU(inplace=True)
        )
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.stage2 = self.make_layers(24, channel_num[0], 4, 2, groups)
        self.stage3 = self.make_layers(channel_num[0], channel_num[1], 8, 2,
groups)
        self.stage4 = self.make_layers(channel_num[1], channel_num[2], 4, 2,
groups)
        self.avgpool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Linear(channel_num[2], class_num)
    def make_layers(self, input_channels, output_channels, layers_num, stride,
groups):
        layers = []
        layers.append(Bottleneck(input_channels, output_channels -
input_channels, stride, groups))
        input_channels = output_channels
        for i in range(layers_num - 1):
            Bottleneck(input_channels, output_channels, 1, groups)
        return nn.Sequential(*layers)
    def forward(self, x):
```

```
        x = self.conv1(x)
        x = self.maxpool(x)
        x = self.stage2(x)
        x = self.stage3(x)
        x = self.stage4(x)
        x = self.avgpool(x)
        x = x.flatten(1)
        x = self.fc(x)
        return x
```

```
def conv3x3(in_channels, out_channels, stride=1,
            padding=1, bias=True, groups=1):
    """3x3 convolution with padding
    """
    return nn.Conv2d(
        in_channels,
        out_channels,
        kernel_size=3,
        stride=stride,
        padding=padding,
        bias=bias,
        groups=groups)


def conv1x1(in_channels, out_channels, groups=1):
    """1x1 convolution with padding
    - Normal pointwise convolution When groups == 1
    - Grouped pointwise convolution when groups > 1
    """
    return nn.Conv2d(
        in_channels,
        out_channels,
        kernel_size=1,
        groups=groups,
        stride=1)


def channel_shuffle(x, groups):
    batchsize, num_channels, height, width = x.data.size()

    channels_per_group = num_channels // groups

    # reshape
    x = x.view(batchsize, groups,
        channels_per_group, height, width)

    # transpose
    # - contiguous() required if transpose() is used before view().
    #   See https://github.com/pytorch/pytorch/issues/764
    x = torch.transpose(x, 1, 2).contiguous()

    # flatten
    x = x.view(batchsize, -1, height, width)

    return x
```

```
class ShuffleUnit(nn.Module):
    def __init__(self, in_channels, out_channels, groups=3,
                 grouped_conv=True, combine='add'):

        super(ShuffleUnit, self).__init__()

        self.in_channels = in_channels
        self.out_channels = out_channels
        self.grouped_conv = grouped_conv
        self.combine = combine
        self.groups = groups
        self.bottleneck_channels = self.out_channels // 4

        # define the type of ShuffleUnit
        if self.combine == 'add':
            # ShuffleUnit Figure 2b
            self.depthwise_stride = 1
            self._combine_func = self._add
        elif self.combine == 'concat':
            # ShuffleUnit Figure 2c
            self.depthwise_stride = 2
            self._combine_func = self._concat

            # ensure output of concat has the same channels as
            # original output channels.
            self.out_channels -= self.in_channels
        else:
            raise ValueError("Cannot combine tensors with \"{}\"" \
                             "Only \"add\" and \"concat\" are" \
                             "supported".format(self.combine))

        # Use a 1x1 grouped or non-grouped convolution to reduce input channels
        # to bottleneck channels, as in a ResNet bottleneck module.
        # NOTE: Do not use group convolution for the first conv1x1 in Stage 2.
        self.first_1x1_groups = self.groups if grouped_conv else 1

        self.g_conv_1x1_compress = self._make_grouped_conv1x1(
            self.in_channels,
            self.bottleneck_channels,
            self.first_1x1_groups,
            batch_norm=True,
            relu=True
            )

        # 3x3 depthwise convolution followed by batch normalization
        self.depthwise_conv3x3 = conv3x3(
            self.bottleneck_channels, self.bottleneck_channels,
            stride=self.depthwise_stride, groups=self.bottleneck_channels)
        self.bn_after_depthwise = nn.BatchNorm2d(self.bottleneck_channels)

        # Use 1x1 grouped convolution to expand from
        # bottleneck_channels to out_channels
        self.g_conv_1x1_expand = self._make_grouped_conv1x1(
            self.bottleneck_channels,
            self.out_channels,
            self.groups,
```

```
            batch_norm=True,
            relu=False
            )


    @staticmethod
    def _add(x, out):
        # residual connection
        return x + out


    @staticmethod
    def _concat(x, out):
        # concatenate along channel axis
        return torch.cat((x, out), 1)


    def _make_grouped_conv1x1(self, in_channels, out_channels, groups,
        batch_norm=True, relu=False):

        modules = OrderedDict()

        conv = conv1x1(in_channels, out_channels, groups=groups)
        modules['conv1x1'] = conv

        if batch_norm:
            modules['batch_norm'] = nn.BatchNorm2d(out_channels)
        if relu:
            modules['relu'] = nn.ReLU()
        if len(modules) > 1:
            return nn.Sequential(modules)
        else:
            return conv


    def forward(self, x):
        # save for combining later with output
        residual = x

        if self.combine == 'concat':
            residual = F.avg_pool2d(residual, kernel_size=3,
                stride=2, padding=1)

        out = self.g_conv_1x1_compress(x)
        out = channel_shuffle(out, self.groups)
        out = self.depthwise_conv3x3(out)
        out = self.bn_after_depthwise(out)
        out = self.g_conv_1x1_expand(out)

        out = self._combine_func(residual, out)
        return F.relu(out)


class ShuffleNet(nn.Module):
    """ShuffleNet implementation.
    """

    def __init__(self, groups=3, in_channels=3, num_classes=1000):
```

```
        """ShuffleNet constructor.
        Arguments:
            groups (int, optional): number of groups to be used in grouped
                1x1 convolutions in each ShuffleUnit. Default is 3 for best
                performance according to original paper.
            in_channels (int, optional): number of channels in the input tensor.
                Default is 3 for RGB image inputs.
            num_classes (int, optional): number of classes to predict. Default
                is 1000 for ImageNet.
        """
        super(ShuffleNet, self).__init__()

        self.groups = groups
        self.stage_repeats = [3, 7, 3]
        self.in_channels =  in_channels
        self.num_classes = num_classes

        # index 0 is invalid and should never be called.
        # only used for indexing convenience.
        if groups == 1:
            self.stage_out_channels = [-1, 24, 144, 288, 567]
        elif groups == 2:
            self.stage_out_channels = [-1, 24, 200, 400, 800]
        elif groups == 3:
            self.stage_out_channels = [-1, 24, 240, 480, 960]
        elif groups == 4:
            self.stage_out_channels = [-1, 24, 272, 544, 1088]
        elif groups == 8:
            self.stage_out_channels = [-1, 24, 384, 768, 1536]
        else:
            raise ValueError(
                """{} groups is not supported for
                   1x1 Grouped Convolutions""".format(num_groups))

        # Stage 1 always has 24 output channels
        self.conv1 = conv3x3(self.in_channels,
                             self.stage_out_channels[1], # stage 1
                             stride=2)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        # Stage 2
        self.stage2 = self._make_stage(2)
        # Stage 3
        self.stage3 = self._make_stage(3)
        # Stage 4
        self.stage4 = self._make_stage(4)

        # Global pooling:
        # Undefined as PyTorch's functional API can be used for on-the-fly
        # shape inference if input size is not ImageNet's 224x224

        # Fully-connected classification layer
        num_inputs = self.stage_out_channels[-1]
        self.fc = nn.Linear(num_inputs, self.num_classes)
        self.init_params()


    def init_params(self):
```

```
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                init.kaiming_normal(m.weight, mode='fan_out')
                if m.bias is not None:
                    init.constant(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                init.constant(m.weight, 1)
                init.constant(m.bias, 0)
            elif isinstance(m, nn.Linear):
                init.normal(m.weight, std=0.001)
                if m.bias is not None:
                    init.constant(m.bias, 0)


    def _make_stage(self, stage):
        modules = OrderedDict()
        stage_name = "ShuffleUnit_Stage{}".format(stage)

        # First ShuffleUnit in the stage
        # 1. non-grouped 1x1 convolution (i.e. pointwise convolution)
        #   is used in Stage 2. Group convolutions used everywhere else.
        grouped_conv = stage > 2

        # 2. concatenation unit is always used.
        first_module = ShuffleUnit(
            self.stage_out_channels[stage-1],
            self.stage_out_channels[stage],
            groups=self.groups,
            grouped_conv=grouped_conv,
            combine='concat'
            )
        modules[stage_name+"_0"] = first_module

        # add more ShuffleUnits depending on pre-defined number of repeats
        for i in range(self.stage_repeats[stage-2]):
            name = stage_name + "_{}".format(i+1)
            module = ShuffleUnit(
                self.stage_out_channels[stage],
                self.stage_out_channels[stage],
                groups=self.groups,
                grouped_conv=True,
                combine='add'
                )
            modules[name] = module

        return nn.Sequential(modules)


    def forward(self, x):
        x = self.conv1(x)
        x = self.maxpool(x)

        x = self.stage2(x)
        x = self.stage3(x)
        x = self.stage4(x)

        # global average pooling layer
        x = F.avg_pool2d(x, x.data.size()[-2:])
```

```
        # flatten for input to fully-connected layer
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return F.log_softmax(x, dim=1)
```