Run, Don’t Walk: Chasing Higher FLOPS for Faster Neural Networks

Jierun Chen¹, Shiu-hong Kao¹, Hao He¹
Weipeng Zhuo¹, Song Wen², Chul-Ho Lee³, S.-H. Gary Chan¹
¹HKUST, ²Rutgers University, ³Texas State University

Abstract

To design fast neural networks, many works have been focusing on reducing the number of floating-point operations (FLOPs). We observe that such reduction in FLOPs, however, does not necessarily lead to a similar level of reduction in latency. This mainly stems from inefficiently low floating-point operations per second (FLOPs). To achieve faster networks, we revisit popular operators and demonstrate that such low FLOPS is mainly due to frequent memory access of the operators, especially the depthwise convolution. We hence propose a novel partial convolution (PConv) that extracts spatial features more efficiently, by cutting down redundant computation and memory access simultaneously. Building upon our PConv, we further propose FasterNet, a new family of neural networks, which attains substantially higher running speed than others on a wide range of devices, without compromising on accuracy for various vision tasks. For example, on ImageNet-1k, our tiny FasterNet-T0 is 2.8×, 3.3×, and 2.4× faster than MobileViT-XXS on GPU, CPU, and ARM processors, respectively, while being 2.9% more accurate. Our large FasterNet-L achieves impressive 83.5% top-1 accuracy, on par with the emerging Swin-B, while having 36% higher inference throughput on GPU, as well as saving 37% compute time on CPU. Code is available at https://github.com/JierunChen/FasterNet.

1. Introduction

Neural networks have undergone rapid development in various computer vision tasks such as image classification, detection and segmentation. While their impressive performance has powered many applications, a roaring trend is to pursue fast neural networks with low latency and high throughput for great user experiences, instant responses, safety reasons, etc.

How to be fast? Instead of asking for more costly computing devices, researchers and practitioners prefer to design cost-effective fast neural networks with reduced computational complexity, mainly measured in the number of floating-point operations (FLOPs)¹. MobileNets [20, 21, 47], ShuffleNets [40, 76] and GhostNet [13], among others, leverage the depthwise convolution (DWConv) [48] and/or group convolution (GConv) [27] to extract spatial features. However, in the effort to reduce FLOPs, the operators often suffer from the side effect of increased memory access. MicroNet [29] further decomposes and sparsifies the network to push its FLOPs to an extremely low level. Despite its improvement in FLOPs, this approach experiences inefficient fragmented computation. Besides, the above networks are often accompanied by additional data manipulations, such as concatenation, shuffling, and pooling, whose running time tends to be significant for tiny models.

Apart from the above pure convolutional neural networks (CNNs), there is an emerging interest in making vision transformers (ViTs) [11] and multilayer perceptrons (MLPs) architectures [57] smaller and faster. For example, MobileViTs [42, 43, 63] and MobileFormer [6] reduce the computational complexity by combining DWConv with a modified attention mechanism. However, they still suffer from the aforementioned issue with DWConv and also need dedicated hardware support for the modified attention mechanism. The use of advanced yet time-consuming nor-

¹We follow a widely adopted definition of FLOPs, as the number of multiply-adds [36, 76].

Figure 1. Our partial convolution (PConv) is fast and efficient by applying filters on only a few input channels while leaving the remaining ones untouched. PConv obtains lower FLOPs than the regular convolution and higher FLOPS than the depthwise/group convolution.
ormalization and activation layers may also limit their speed on devices. 

All these issues together lead to the following question: Are these “fast” neural networks really fast? To answer this, we examine the relationship between latency and FLOPs, which is captured by

$$\text{Latency} = \frac{\text{FLOPs}}{\text{FLOPs}}$$

where FLOPs is short for floating-point operations per second, as a measure of the effective computational speed. While there are many attempts to reduce FLOPs, they seldom consider optimizing FLOPs at the same time to achieve truly low latency. To better understand the situation, we compare the FLOPs of typical neural networks on an Intel CPU. The results in Fig. 2 show that many existing neural networks suffer from low FLOPs, and their FLOPs is generally lower than the popular ResNet50. With such low FLOPs, these “fast” neural networks are actually not fast enough. Their reduction in FLOPs cannot be translated into the exact amount of reduction in latency. In some cases, there is no improvement, and it even leads to worse latency. For example, CycleMLP-B1 [5] has half of FLOPs of ResNet50 [16] but runs more slowly (i.e., CycleMLP-B1 vs. ResNet50: 116.1 ms vs. 73.0 ms). Note that this discrepancy between FLOPs and latency has also been noticed in previous works [40, 42] but remains unresolved partially because they employ the DWConv/GConv and various data manipulations with low FLOPs. It is deemed there are no better alternatives available.

This paper aims to eliminate the discrepancy by developing a simple yet fast and effective operator that maintains high FLOPs with reduced FLOPs. Specifically, we reexamine existing operators, particularly DWConv, in terms of the computational speed – FLOPs. We uncover that the main reason causing the low FLOPs issue is frequent memory access. We then propose a novel partial convolution (PConv) as a competitive alternative that reduces the computational redundancy as well as the number of memory access. Fig. 1 illustrates the design of our PConv. It takes advantage of redundancy within the feature maps and systematically applies a regular convolution (Conv) on only a part of the input channels while leaving the remaining ones untouched. By nature, PConv has lower FLOPs than the regular Conv while having higher FLOPs than the DWConv/GConv. In other words, PConv better exploits the on-device computational capacity. PConv is also effective in extracting spatial features as empirically validated later in the paper.

We further introduce FasterNet, which is primarily built upon our PConv, as a new family of networks that run highly fast on various devices. In particular, our FasterNet achieves state-of-the-art performance for classification, detection, and segmentation tasks while having much lower latency and higher throughput. For example, our tiny FasterNet-T0 is 2.8×, 3.3×, and 2.4× faster than MobileViT-XXS [42] on GPU, CPU, and ARM processors, respectively, while being 2.9% more accurate on ImageNet-1k. Our large FasterNet-L achieves 83.5% top-1 accuracy, on par with the emerging Swin-B [35], while offering 36% higher throughput on GPU and saving 37% compute time on CPU. To summarize, our contributions are as follows:

- We point out the importance of achieving higher FLOPs beyond simply reducing FLOPs for faster neural networks.
- We introduce a simple yet fast and effective operator called PConv, which has a high potential to replace the existing go-to choice, DWConv.
- We introduce FasterNet which runs favorably and universally fast on a variety of devices such as GPU, CPU, and ARM processors.
- We conduct extensive experiments on various tasks and validate the high speed and effectiveness of our PConv and FasterNet.
2. Related Work

We briefly review prior works on fast and efficient neural networks and differentiate this work from them.

**CNN.** CNNs are the mainstream architecture in the computer vision field, especially when it comes to deployment in practice, where being fast is as important as being accurate. Though there have been numerous studies [7, 8, 17, 29, 48, 49, 75, 78] to achieve higher efficiency, the rationale behind them is more or less to perform a low-rank approximation. Specifically, the group convolution [27] and the depthwise separable convolution [48] (consisting of depthwise and pointwise convolutions) are probably the most popular ones. They have been widely adopted in mobile/edge-oriented networks, such as MobileNets [20, 21, 47], ShuffleNets [40, 76], GhostNet [13], EfficientNets [54, 55], TinyNet [14], Xception [8], CondenseNet [23, 70], TVConv [4], MnasNet [53], and FBNet [67]. While they exploit the redundancy in filters to reduce the number of parameters and FLOPs, they suffer from increased memory access when increasing the network width to compensate for the accuracy drop. By contrast, we consider the redundancy in feature maps and propose a partial convolution to reduce FLOPs and memory access simultaneously.

**ViT, MLP, and variants.** There is a growing interest in studying ViT ever since Dosovitskiy et al. [11] expanded the application scope of transformers [62] from machine translation [62] or forecasting [66] to the computer vision field. Many follow-up works have attempted to improve ViT in terms of training setting [51, 58, 59] and model design [12, 34, 35, 65, 77]. One notable trend is to pursue a better accuracy-latency trade-off by reducing the complexity of the attention operator [1, 25, 39, 56, 61], incorporating convolution into ViTs [6, 10, 50], or doing both [3, 30, 43, 48]. Besides, other studies [5, 31, 57] propose to replace the attention with simple MLP-based operators. However, they often evolve to be CNN-like [33]. In this paper, we focus on analyzing the convolution operations, particularly DWConv, due to the following reasons: First, the advantage of attention over convolution is unclear or debatable [36, 64]. Second, the attention-based mechanism generally runs slower than its convolutional counterparts and thus becomes less favorable for the current industry [22, 42]. Finally, DWConv is still a popular choice in many hybrid models, so it is worth a careful examination.

3. Design of PConv and FasterNet

In this section, we first revisit DWConv and analyze the issue with its frequent memory access. We then introduce PConv as a competitive alternative operator to resolve the issue. After that, we introduce FasterNet and explain its details, including design considerations.

### 3.1. Preliminary

**DWConv** is a popular variant of Conv and has been widely adopted as a key building block for many neural networks. For an input $I \in \mathbb{R}^{c \times h \times w}$, DWConv applies $c$ filters $W \in \mathbb{R}^{k \times k}$ to compute the output $O \in \mathbb{R}^{c \times h \times w}$. As shown in Fig. 1(b), each filter slides spatially on one input channel and contributes to one output channel. This depthwise computation makes DWConv have as low FLOPs as $h \times w \times k^2 \times c$ compared to a regular Conv with $h \times w \times c^2$. While effective in reducing FLOPs, a DWConv, which is typically followed by a pointwise convolution, or PWConv, cannot be simply used to replace a regular Conv as it would incur a severe accuracy drop. Thus, in practice the channel number $c$ (or the network width) of DWConv is increased to $c'$ ($c' > c$) to compensate the accuracy drop, e.g., the width is expanded by six times for the DWConv in the inverted residual blocks [47]. This, however, results in much higher memory access that can cause non-negligible delay and slow down the overall computation, especially for I/O-bound devices. In particular, the number of memory access now escalates to

$$h \times w \times 2c' + k^2 \times c' \approx h \times w \times 2c'$$

(2)

which is higher than that of a regular Conv, i.e.,

$$h \times w \times 2c + k^2 \times c^2 \approx h \times w \times 2c.$$  

(3)

Note that the $h \times w \times 2c'$ memory access is spent on the I/O operation, which is deemed to be already the minimum cost and hard to optimize further.

### 3.2. Partial convolution as a basic operator

We below demonstrate that the cost can be further optimized by leveraging the feature maps’ redundancy. As visualized in Fig. 3, the feature maps share high similarities among different channels. This redundancy has also

Figure 3. Visualization of feature maps in an intermediate layer of a pre-trained ResNet50, with the top-left image as the input. Qualitatively, we can see the high redundancies across different channels.
been covered in many other works [13, 74], but few of them make full use of it in a simple yet effective way.

Specifically, we propose a simple PConv to reduce computational redundancy and memory access simultaneously. The bottom-left corner in Fig. 4 illustrates how our PConv works. It simply applies a regular Conv on only a part of the input channels for spatial feature extraction and leaves the remaining channels untouched. For contiguous or regular memory access, we consider the first or last consecutive \( c_p \) channels as the representatives of the whole feature maps for computation. Without loss of generality, we consider the input and output feature maps to have the same number of channels. Therefore, the FLOPs of a PConv are only

\[
    h \times w \times k^2 \times c_p. \tag{4}
\]

With a typical partial ratio \( r = \frac{c_p}{c} = \frac{1}{4} \), the FLOPs of a PConv is only \( \frac{1}{16} \) of a regular Conv. Besides, PConv has a smaller amount of memory access, i.e.,

\[
    h \times w \times 2c_p + k^2 \times c_p^2 \approx h \times w \times 2c_p, \tag{5}
\]

which is only \( \frac{1}{2} \) of a regular Conv for \( r = \frac{1}{4} \).

Since there are only \( c_p \) channels utilized for spatial feature extraction, one may ask if we can simply remove the remaining \( (c - c_p) \) channels? If so, PConv would degrade to a regular Conv with fewer channels, which deviates from our objective to reduce redundancy. Note that we keep the remaining channels untouched instead of removing them from the feature maps. It is because they are useful for a subsequent PWConv layer, which allows the feature information to flow through all channels.

### 3.3. PConv followed by PWConv

To fully and efficiently leverage the information from all channels, we further append a pointwise convolution (PWConv) to our PConv. Their effective receptive field together on the input feature maps looks like a T-shaped Conv, which focuses more on the center position compared to a regular Conv uniformly processing a patch, as shown in Fig. 5. To justify this T-shaped receptive field, we first evaluate the importance of each position by calculating the position-wise Frobenius norm. We assume that a position tends to be more important if it has a larger Frobenius norm than other positions. For a regular Conv filter \( F \in \mathbb{R}^{k^2 \times c} \), the Frobenius norm at position \( i \) is calculated by \( \|F_i\|_F = \sqrt{\sum_{j=1}^c |F_{ij}|^2} \), for \( i = 1, 2, 3..., k^2 \). We consider a salient position to be the one with the maximum Frobenius norm. We then collectively examine each filter in a pre-trained ResNet18, find out their salient positions, and plot a histogram of the salient positions. Results in Fig. 6 show that the center position turns out to be the salient position most frequently among the filters. In other words, the center position weighs more than its surrounding neighbors. This is consistent with the T-shaped computation which concentrates on the center position.

While the T-shaped Conv can be directly used for efficient computation, we show that it is better to decompose the T-shaped Conv into a PConv and a PWConv because the decomposition exploits the inter-filter redundancy and further saves FLOPs. For the same input \( I \in \mathbb{R}^{c \times h \times w} \) and output \( O \in \mathbb{R}^{c \times h \times w} \), a T-shaped Conv’s FLOPs can be calculated as

\[
    h \times w \times (k^2 \times c_p \times c + c \times (c - c_p)), \tag{6}
\]

which is higher than the FLOPs of a PConv and a PWConv, i.e.,

\[
    h \times w \times (k^2 \times c_p^2 + c \times c_p), \tag{7}
\]

where \( c > c_p \) and \( c - c_p > c_p \) (e.g., when \( c_p = \frac{c}{4} \)). Besides, we can readily leverage the regular Conv for the two-step implementation.

### 3.4. FasterNet as a general backbone

Given our novel PConv and off-the-shelf PWConv as the primary building operators, we further propose FasterNet, a new family of neural networks that runs favorably fast and is highly effective for many vision tasks. We aim to keep the architecture as simple as possible, without bells and whistles, to make it hardware-friendly in general.

We present the overall architecture in Fig. 4. It has four hierarchical stages, each of which is preceded by an embedding layer (a regular Conv \( 4 \times 4 \) with stride 4) or a merging layer (a regular Conv \( 2 \times 2 \) with stride 4) for spatial downsampling and channel number expanding. Each stage has a stack of FasterNet blocks. We observe that the blocks in the last two stages consume less memory access and tend to have higher FLOPs, as empirically validated in Tab. 1. Thus, we put more FasterNet blocks and correspondingly assign more computations to the last two stages. Each FasterNet block has a PConv layer followed by two PWConv (or Conv \( 1 \times 1 \)) layers. Together, they appear as inverted residual blocks where the middle layer has an expanded number of channels, and a shortcut connection is placed to reuse the input features.

In addition to the above operators, the normalization and activation layers are also indispensable for high-performing neural networks. Many prior works [13, 16, 47], however, overuse such layers throughout the network, which may limit the feature diversity and thus hurt the performance. It can also slow down the overall computation. By contrast, we put them only after each middle PWConv to preserve the feature diversity and achieve lower latency. Besides, we use the batch normalization (BN) [26] instead of other alternative ones [2, 60, 68]. The benefit of BN is that it can be merged into its adjacent Conv layers for faster inference.
4. Experimental Results

We first examine the computational speed of our PConv and its effectiveness when combined with a PWConv. We then comprehensively evaluate the performance of our FasterNet for classification, detection, and segmentation tasks. Finally, we conduct a brief ablation study.

To benchmark the latency and throughput, we choose the following three typical processors, which cover a wide range of computational capacity: GPU (2080Ti), CPU (Intel i9-9900X, using a single thread), and ARM (Cortex-A72, using a single thread). We report their latency for inputs with a batch size of 1 and throughput for inputs with a batch size of 32. During inference, the BN layers are merged to their adjacent layers wherever applicable.

4.1. PConv is fast with high FLOPS

We below show that our PConv is fast and better exploits the on-device computational capacity. Specifically, we stack 10 layers of pure PConv and take feature maps of typical dimensions as inputs. We then measure FLOPs and latency/throughput on GPU, CPU, and ARM processors, which also allow us to further compute FLOPS. We repeat the same procedure for other convolutional variants and make comparisons.

Results in Tab. 1 show that PConv is overall an appealing choice for high FLOPS with reduced FLOPs. It has only 1/7 FLOPs of a regular Conv and achieves 10.5×, 6.2×, and 22.8× higher FLOPS than the DWConv on GPU, CPU, and ARM, respectively. We are unsurprised to see that the regular Conv has the highest FLOPS as it has been constantly optimized for years. However, its total FLOPs and...
We next show that a PConv followed by a PWConv is effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first build four datasets by effective in approximating a regular Conv to transform the feature maps. To this end, we first bu...
such promising results, we highlight that our FasterNet is
well as saving 37% and 15% compute time on CPU. Given
ing 36% and 28% higher inference throughput on GPU, as
the emerging Swin-B [35] and ConvNeXt-B [36] while hav-
FasterNet-L achieves 83.5% top-1 accuracy, comparable to
respectively, while being 2.9% more accurate. Our large
MobileViT-XXS [42] on GPU, CPU, and ARM processors,
in Tab. 3, FasterNet-T0 is
having similar top-1 accuracy. As quantitatively shown
ViT and MLP models on a wide range of devices, when
other perspective, FasterNet runs faster than various CNN,
cy/throughput among all the networks examined. From an-
the new state-of-the-art in balancing accuracy and laten-
time](M) is (G), FLOPs (G), Throughput (fps), Latency (ms)↑, Latency (ms)↓, Acc. (%) on CPU, on GPU, on ARM.

<table>
<thead>
<tr>
<th>Network</th>
<th>Params</th>
<th>FLOPs</th>
<th>Throughput (fps)</th>
<th>Latency (ms)↑</th>
<th>Latency (ms)↓</th>
<th>Acc. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShuffleNetV2 × 1.5 [40]</td>
<td>3.5</td>
<td>0.30</td>
<td>4878</td>
<td>12.1</td>
<td>266</td>
<td>72.6</td>
</tr>
<tr>
<td>MobileNetV2 [47]</td>
<td>3.5</td>
<td>0.31</td>
<td>4198</td>
<td>12.2</td>
<td>442</td>
<td>72.0</td>
</tr>
<tr>
<td>MobileViT-XXS [42]</td>
<td>1.3</td>
<td>0.42</td>
<td>2393</td>
<td>30.8</td>
<td>348</td>
<td>69.0</td>
</tr>
<tr>
<td>EdgeNeXt-XXS [41]</td>
<td>1.3</td>
<td>0.26</td>
<td>2765</td>
<td>15.7</td>
<td>239</td>
<td>71.2</td>
</tr>
<tr>
<td>FasterNet-T0</td>
<td>3.9</td>
<td>0.34</td>
<td>6807</td>
<td>9.2</td>
<td>143</td>
<td>71.9</td>
</tr>
<tr>
<td>GhostNet × 1.3 [13]</td>
<td>7.4</td>
<td>0.24</td>
<td>2988</td>
<td>17.9</td>
<td>481</td>
<td>71.7</td>
</tr>
<tr>
<td>ShuffleNetV2 × 2.40 [40]</td>
<td>7.4</td>
<td>0.59</td>
<td>3339</td>
<td>17.8</td>
<td>403</td>
<td>74.9</td>
</tr>
<tr>
<td>MobileNetV2 × 1.4 [47]</td>
<td>6.1</td>
<td>0.60</td>
<td>2711</td>
<td>22.6</td>
<td>650</td>
<td>74.7</td>
</tr>
<tr>
<td>MobileViT-XXS [42]</td>
<td>2.3</td>
<td>1.05</td>
<td>1392</td>
<td>40.8</td>
<td>648</td>
<td>74.8</td>
</tr>
<tr>
<td>EdgeNeXt-XXS [41]</td>
<td>2.3</td>
<td>0.54</td>
<td>1738</td>
<td>24.4</td>
<td>434</td>
<td>75.0</td>
</tr>
<tr>
<td>PVT-Tiny [65]</td>
<td>13.2</td>
<td>1.94</td>
<td>1266</td>
<td>55.6</td>
<td>708</td>
<td>75.1</td>
</tr>
<tr>
<td>FasterNet-T1</td>
<td>7.6</td>
<td>0.85</td>
<td>3782</td>
<td>17.7</td>
<td>285</td>
<td>76.2</td>
</tr>
<tr>
<td>CycleMLP-B1 [5]</td>
<td>15.2</td>
<td>2.10</td>
<td>865</td>
<td>116.1</td>
<td>892</td>
<td>79.1</td>
</tr>
<tr>
<td>PoolFormer-S12 [71]</td>
<td>11.9</td>
<td>1.82</td>
<td>1439</td>
<td>49.0</td>
<td>665</td>
<td>77.2</td>
</tr>
<tr>
<td>MobileViT-S [42]</td>
<td>5.6</td>
<td>2.03</td>
<td>1039</td>
<td>56.7</td>
<td>941</td>
<td>78.4</td>
</tr>
<tr>
<td>EdgeNeXt-S [41]</td>
<td>5.6</td>
<td>1.26</td>
<td>1128</td>
<td>39.2</td>
<td>743</td>
<td>79.4</td>
</tr>
<tr>
<td>ResNet50 [16, 36]</td>
<td>25.6</td>
<td>4.11</td>
<td>959</td>
<td>73.0</td>
<td>1131</td>
<td>78.8</td>
</tr>
<tr>
<td>FasterNet-T2</td>
<td>15.0</td>
<td>1.91</td>
<td>1991</td>
<td>33.5</td>
<td>497</td>
<td>78.9</td>
</tr>
<tr>
<td>CycleMLP-B2 [5]</td>
<td>26.8</td>
<td>3.90</td>
<td>528</td>
<td>186.3</td>
<td>1502</td>
<td>81.6</td>
</tr>
<tr>
<td>PoolFormer-S24 [71]</td>
<td>21.4</td>
<td>3.41</td>
<td>748</td>
<td>92.8</td>
<td>1261</td>
<td>80.3</td>
</tr>
<tr>
<td>PoolFormer-S36 [71]</td>
<td>30.9</td>
<td>5.00</td>
<td>507</td>
<td>138.0</td>
<td>1860</td>
<td>81.4</td>
</tr>
<tr>
<td>ConvNeXt-T [36]</td>
<td>28.6</td>
<td>4.47</td>
<td>657</td>
<td>86.3</td>
<td>1889</td>
<td>82.1</td>
</tr>
<tr>
<td>Swin-T [35]</td>
<td>28.3</td>
<td>4.51</td>
<td>609</td>
<td>122.2</td>
<td>1424</td>
<td>81.3</td>
</tr>
<tr>
<td>PVT-Small [65]</td>
<td>24.5</td>
<td>3.83</td>
<td>689</td>
<td>89.6</td>
<td>1345</td>
<td>79.8</td>
</tr>
<tr>
<td>PVT-Medium [65]</td>
<td>44.2</td>
<td>6.69</td>
<td>438</td>
<td>143.6</td>
<td>2142</td>
<td>81.2</td>
</tr>
<tr>
<td>FasterNet-S</td>
<td>31.1</td>
<td>4.56</td>
<td>1029</td>
<td>71.2</td>
<td>1103</td>
<td>81.3</td>
</tr>
<tr>
<td>PoolFormer-M36 [71]</td>
<td>56.2</td>
<td>8.80</td>
<td>320</td>
<td>215.0</td>
<td>2979</td>
<td>82.1</td>
</tr>
<tr>
<td>ConvNeXt-S [36]</td>
<td>50.2</td>
<td>8.71</td>
<td>377</td>
<td>153.2</td>
<td>3484</td>
<td>83.1</td>
</tr>
<tr>
<td>Swin-S [35]</td>
<td>49.6</td>
<td>8.77</td>
<td>348</td>
<td>224.2</td>
<td>2613</td>
<td>83.0</td>
</tr>
<tr>
<td>PVT-Large [65]</td>
<td>61.4</td>
<td>9.85</td>
<td>306</td>
<td>203.4</td>
<td>3101</td>
<td>81.7</td>
</tr>
<tr>
<td>FasterNet-M</td>
<td>53.5</td>
<td>8.74</td>
<td>500</td>
<td>129.5</td>
<td>2092</td>
<td>83.0</td>
</tr>
<tr>
<td>PoolFormer-M48 [71]</td>
<td>73.5</td>
<td>11.59</td>
<td>242</td>
<td>281.8</td>
<td>OOM</td>
<td>82.5</td>
</tr>
<tr>
<td>ConvNeXt-B [36]</td>
<td>88.6</td>
<td>15.38</td>
<td>253</td>
<td>257.1</td>
<td>OOM</td>
<td>83.8</td>
</tr>
<tr>
<td>Swin-B [35]</td>
<td>87.8</td>
<td>15.47</td>
<td>237</td>
<td>349.2</td>
<td>OOM</td>
<td>83.5</td>
</tr>
<tr>
<td>FasterNet-L</td>
<td>93.5</td>
<td>15.52</td>
<td>323</td>
<td>219.5</td>
<td>OOM</td>
<td>83.5</td>
</tr>
</tbody>
</table>

Table 3. Comparison on ImageNet-1k benchmark. Models with similar top-1 accuracy are grouped together. For each group, our FasterNet achieves the highest throughput on GPU and the lowest latency on CPU and ARM. All models are evaluated at 224 × 224 resolution except for the MobileViT and EdgeNeXt with 256 × 256. OOM is short for out of memory.

4.4. FasterNet on downstream tasks

To further evaluate the generalization ability of FasterNet, we conduct experiments on the challenging COCO dataset [32] for object detection and instance segmentation. As a common practice, we employ the ImageNet pre-trained FasterNet as a backbone and equip it with the popular Mask R-CNN detector [15]. To highlight the effectiveness of the backbone itself, we simply follow PoolFormer [71] and adopt an AdamW optimizer, a 1× training

much simpler than many other models in terms of architect-
tural design, which showcases the feasibility of designing
simple yet powerful neural networks.

Figure 7. FasterNet has the highest efficiency in balancing accuracy-throughput and accuracy-latency trade-offs for different devices. To save space and make the plots more proportionate, we showcase network variants within a certain range of latency. Full plots can be found in the appendix, which show consistent results.
schedule (12 epochs), a batch size of 16, and other training settings without further hyper-parameter tuning.

Tab. 4 shows the results for comparison between FasterNet and representative models. FasterNet consistently outperforms ResNet and ResNext by having higher average precision (AP) with similar latency. Specifically, FasterNet-S yields +1.9 higher box AP and +2.4 higher mask AP compared to the standard baseline ResNet50. FasterNet is also competitive against the ViT variants. Under similar FLOPs, FasterNet-L reduces PVT-Large’s latency by 38%, i.e., from 152.2 ms to 93.8 ms on GPU, and achieves +1.1 higher box AP and +0.4 higher mask AP.

### 4.5. Ablation study

We conduct a brief ablation study on the value of partial ratio \( r \) and the choices of activation and normalization layers. We compare different variants in terms of ImageNet top-1 accuracy and on-device latency/throughput. Results are summarized in Tab. 5. For the partial ratio \( r \), we set it to \( \frac{1}{2} \) for all FasterNet variants by default, which achieves higher accuracy, higher throughput, and lower latency at similar complexity. A too large partial ratio \( r \) would make PConv degrade to a regular Conv, while a too small value would render PConv less effective in capturing the spatial features. For the normalization layers, we choose BatchNorm over LayerNorm because BatchNorm can be merged into its adjacent convolutional layers for faster inference while it is as effective as LayerNorm in our experiment. For the activation function, interestingly, we empirically found that GELU fits FasterNet-T0/T1 models more efficiently than ReLU. It, however, becomes opposite for FasterNet-T2/S/M/L. Here we only show two examples in Tab. 5 due to space constraint. We conjecture that GELU strengthens FasterNet-T0/T1 by having higher non-linearity, while the benefit fades away for larger FasterNet variants.

### 5. Conclusion

In this paper, we have investigated the common and unresolved issue that many established neural networks suffer from low floating-point operations per second (FLOPS). We have revisited a bottleneck operator, DWConv, and analyzed its main cause for a slowdown – frequent memory access. To overcome the issue and achieve faster neural networks, we have proposed a simple yet fast and effective operator, PConv, that can be readily plugged into many existing networks. We have further introduced our general-purpose FasterNet, built upon our PConv, that achieves state-of-the-art speed and accuracy trade-off on various devices and vision tasks. We hope that our PConv and FasterNet would inspire more research on simple yet effective neural networks, going beyond academia to impact the industry and community directly.

**Acknowledgement** This work was supported, in part, by Hong Kong General Research Fund under grant number 16200120. The work of C.-H. Lee was supported, in part, by the NSF under Grant IIS-2209921.
References


conference on machine learning, pages 6105–6114. PMLR, 2019. 3
[72] Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In Proceedings of the IEEE/CVF international conference on computer vision, pages 6023–6032, 2019. 6