Convolutional Networks with Oriented 1D Kernels

Alexandre Kirchmeyer
Karnegie Mellon University
akirchme@cs.cmu.edu

Jia Deng
Princeton University
jiadeng@princeton.edu

Abstract

In computer vision, 2D convolution is arguably the most important operation performed by a ConvNet. Unsurprisingly, it has been the focus of intense software and hardware optimization and enjoys highly efficient implementations. In this work, we ask an intriguing question: can we make a ConvNet work without 2D convolutions? Surprisingly, we find that the answer is yes—we show that a ConvNet consisting entirely of 1D convolutions can do just as well as 2D on ImageNet classification. Specifically, we find that one key ingredient to a high-performing 1D ConvNet is oriented 1D kernels: 1D kernels that are oriented not just horizontally or vertically, but also at other angles. Our experiments show that oriented 1D convolutions can not only replace 2D convolutions but also augment existing architectures with large kernels, leading to improved accuracy with minimal FLOPs increase. A key contribution of this work is a highly-optimized custom CUDA implementation of oriented 1D kernels, specialized to the depthwise convolution setting. Our benchmarks demonstrate that our custom CUDA implementation almost perfectly realizes the theoretical advantage of 1D convolution: it is faster than a native horizontal convolution for any arbitrary angle. Code is available at https://github.com/princeton-vl/Oriented1D.

1. Introduction

Convolutional Networks (ConvNets) [27, 25] are widely used in computer vision. They have been successfully applied to a variety of tasks [15, 40, 16, 63] and domains [25, 34, 23, 39, 59], and many new ConvNet-based building blocks [47, 8, 57] and design practices [44, 50, 32] have emerged over the years.

In the computer vision context, a 2D convolution is arguably the most important operation performed by a ConvNet. In virtually all ConvNet architectures [25, 18, 32], 2D convolution is the default choice and accounts for the bulk of the computation. Unsurprisingly, 2D convolution has been the focus of intense software and hardware optimization and enjoys highly efficient implementations.

In this work, we ask an intriguing question: can we make a ConvNet work without 2D convolution? Surprisingly, we find that the answer is yes—we show that a ConvNet consisting entirely of 1D convolutions can do as well on ImageNet classification, a surprising result given that 2D convolution has been the go-to design choice.

Specifically, we find that one key ingredient to a high-performing 1D ConvNet is oriented 1D kernels: 1D kernels that are oriented not just horizontally or vertically, but also at other angles. This is a novel finding—although horizontal and vertical 1D convolutions have been frequently used in the past, 1D kernels oriented at arbitrary angles have not been well studied.

Oriented 1D kernels are motivated by the fact that 2D kernels can be approximated by 1D kernels, which are more efficient computationally. In particular, it is well known that convolution with a separable 2D kernel (i.e. rank 1 as a matrix) is equivalent to consecutive convolutions with a vertical 1D kernel and a horizontal 1D kernel, leading to significant efficiency gain. However, in practice, not all learned 2D kernels are rank 1; if we only use vertical and horizontal 1D kernels, 2D kernels with a full rank, such as diagonal matrices, are poorly approximated, which can lead to a loss in accuracy: for example, the network may be less able to detect a 45° edge. This is when oriented 1D kernels can be helpful. By orienting a 1D kernel at more angles, we expand the set of 2D kernels that can be well approximated by 1D kernels while retaining the efficiency of 1D kernels.

Oriented 1D kernels are also motivated by the increasing use of large 2D kernels in recent convolutional architectures. Large 2D kernels improve the modeling of long-range dependencies, which have been shown to result in better accuracy [32, 12, 30]. However, large 2D kernels are significantly more expensive because the cost scales quadratically. A 31 × 31 kernel is 19 times more costly in terms of multiply-add operations than the standard 7 × 7 kernels. This motivates oriented 1D kernels as an alternative for modeling long-range dependencies, because the cost of 1D kernels scale only linearly with the kernel size.

The concept of oriented 1D kernels is simple, but non-trivial to implement in a way that realizes its efficiency advantage over 2D kernels. This is because on a GPU, the pat-
tern of memory access matters as much as, if not more than, the number of floating point operations. Applying a 1D kernel oriented at an arbitrary angle requires accessing non-contiguous data; a naive implementation can easily negate the theoretical advantage of 1D kernels due to poor management of memory access. In addition, it is important for the implementation to not introduce significant memory overhead, which could be incurred by some naive implementations. Note that while horizontal/vertical 1D convolutions are well supported and optimized by existing libraries, 1D convolutions at an arbitrary angle is not.

A key contribution of this work is a highly-optimized custom CUDA implementation of oriented 1D kernels, specialized to the setting of depthwise convolution [5], where each kernel is applied to only one depth channel. We optimize for depthwise convolution, because it has become an essential building block in recent state-of-art architectures [32, 50], with superior accuracy-efficiency trade-offs. In addition, we find depthwise convolution to be the more useful setting in our experiments. Experiments show that our custom CUDA implementation almost perfectly realizes the theoretical advantage of 1D convolution: our 1D convolution at an arbitrary angle is faster than the native horizontal 1D convolution in PyTorch, which is highly optimized and achieves over 96% of the theoretical speedup over 2D convolution. Notably, our implementation incurs minimal memory overhead; it uses less than 5% more GPU memory than the native horizontal 1D convolution in PyTorch. Our implementation is open-sourced as a plug-and-play PyTorch module at https://github.com/princeton-vl/Oriented1D.

With our custom implementation, our experiments show that oriented 1D convolution can not only replace 2D convolution but also augment existing architectures with large kernels, leading to improved accuracy with minimal FLOPs increase. We expect our implementation to be a useful primitive for further innovation in neural architectures for computer vision.

Our main contributions are two-fold. First, we present the novel finding that state-of-the-art accuracy can be achieved with oriented 1D convolution alone, and that oriented 1D convolution can be a useful primitive to augment existing architectures. Second, we introduce an optimized CUDA implementation of depthwise oriented 1D convolution that nearly maxes out the theoretical efficiency of 1D convolution.

2. Related Work

Modern ConvNets. In recent years we have witnessed the emergence of novel training techniques [32, 50, 51] and block designs [5, 20, 44] inspired by transformers [32] and neural architecture search [50]. Depthwise convolutions [5] have become essential components of many ConvNets [44, 50, 32] for their superior accuracy/computation trade-off. In search of better accuracy/computation trade-offs, recent research has looked at better scaling [51, 57], fused operations [51, 12], neural architectural search [50] and sparsity [30] amongst others. Our work fits into this search for better efficiency by looking at 1D kernels that have the potential to scale better with kernel size.

1D convolutions. The use of 1D kernels has previously been explored in the context of image representation learning by [38, 46, 3]. These approaches rely primarily on decomposing a 2D convolution into a horizontal and vertical convolution [41], and are applied in ConvNets in a parallel [38, 16] or stacked design [46]. However the use of 1D kernels often results in a drop in accuracy [3]. Some approaches explore how to overcome this drawback, through SVD decomposition [10], or L2 reconstruction loss [24]. Our approach tries to address this problem differently: we generalize 1D kernels to allow for non-horizontal and non-vertical kernels, whilst preserving the linear cost and advantages of 1D kernels. A recent work [28] has explored a similar idea using diagonal kernels, but they only studied replacing horizontal/vertical kernels with diagonal/anti-diagonal kernels, and reported better accuracy than horizontal/vertical kernels but worse accuracy than the 2D baseline. In contrast, we study 1D kernels oriented at more varied angles including horizontal and vertical, while achieving no worse accuracy than the 2D baseline. Moreover, their work implements diagonal/anti-diagonal kernels by masking 2D kernels, thus gaining no efficiency advantage over 2D kernels, whereas we provide an optimized implementation that achieves near-perfect speedup.

Oriented kernels. The use of oriented kernels is not new and dates back at least to steerable filters [14]. A commonly used approach is to learn a decomposition of a kernel on a basis which supports rotation, like harmonic functions [54], wavelet transforms [36] or steerable filters [53]. The end goal is often to make the networks invariant or equivariant to rotation, and other transformations [6, 53]. Our approach is simpler than methods using external kernel functions. This allows us to design fast implementations that can be integrated easily into existing architectures. Contrary to group equivariant [6] methods, we do not attempt to design rotation-equivariant networks but only to expand the expressiveness of 1D kernels with arbitrary orientations.

Large kernel design. The emergence of ConvNets benefiting from large kernels [32] has defied the long-established and implied superiority of small kernels since first initiated by VGGNet [45]. This has led to a wave of research exploring new ways to improve performance with bigger kernel sizes. RepLKNet [12] achieves better performance with 31×31 kernels by integrating a small 3×3 or 5×5 kernel that compensates for the difficulty in learning large kernels. SlaK [30] pushes this principle further
with rectangular $5 \times 51$ and $51 \times 5$ kernels combined with dynamic sparsity. Other lines of research include dilated convolutions [58], deformable convolutions [8, 66] and continuous convolutions [43, 42]. Our approach differentiates itself from these past works by studying large oriented 1D kernels, a type of large kernels that offers unique efficiency advantages but have not been studied in the context of modern ConvNet architectures.

3. Oriented 1D kernels

In this section we introduce oriented 1D kernels for depthwise convolution, provide a computational analysis to justify their use and design models to test their effectiveness.

3.1. Definition

An oriented kernel is a kernel which is not always applied along the horizontal or vertical axis but along any axis as specified by an angle/direction $\theta$. We illustrate this concept in Figure 1. For more expressivity, we allow $\theta$ to change at every channel, and call this per-channel angle $\theta_c$. Definition 1 translates this intuition into a mathematical formulation. A justification can be found in the supplementary material.

**Definition 1 (Depthwise convolution of oriented 1D kernel)**

Let $x \in \mathbb{R}^{N \times H \times W \times C}$ be the input of a depthwise convolution by an oriented 1D kernel of weights $w \in \mathbb{R}^{K \times C}$ and per-channel angles $\theta \in \mathbb{R}^C$. We define the output $y \in \mathbb{R}^{N \times P \times Q \times C'}$ as:

$$y_{nppc} = \sum_{k=0}^{K-1} x_{nhwc} w_{kc}$$

(1)

where

$$h = str \cdot p + \lceil -(k - \text{pad}) \cdot \sin \theta_c \rceil$$

$$w = str \cdot q + \lceil (k - \text{pad}) \cdot \cos \theta_c \rceil$$

(2)

$N$ is the batch size, $C$ is the number of input channels, $K$ is the 1D kernel size, $H, W$ and $P, Q$ are input and output dimensions, $str$ is the stride and $pad$ is the padding.

The use of rounding in Equation (2) is necessary because of the discrete nature of input and filter coordinates. As an alternative to rounding down the filter offsets, bilinear interpolation could be considered. However in our experience, interpolation increases the computational cost substantially, making it impractical. See Supplementary material for more details.

In this paper we only consider fixed $\theta$, meaning that $\theta$ is not learnt. Instead, we pick a fixed number $D$ of angles $0, \frac{1}{D} \cdot 180^\circ, \frac{2}{D} 180^\circ, ..., \frac{D-1}{D} 180^\circ$ and partition the channels into $D$ equal groups such that angle $i$ is associated to group $i$. In other words, for every channel of group $i$, it is assigned to angle $\frac{i}{D} 180^\circ$. In practice, we consider $D \in \{2, 4, 8\}$ and the case $D = C$. For $D = 4$ and $C = 512$, this means that there are 4 groups of 128 channels, with respective angles $0^\circ, 45^\circ, 90^\circ, 135^\circ$. We will call $D$ the number of directions as it is more visual.

3.2. Computational cost analysis

Depthwise convolutions are nearly always used in conjunction with pointwise convolutions with normalizations [2, 22] and/or non-linearities [1, 19] added in between. The idea is that depthwise convolutions mix spatial information and pointwise convolutions mix channel information [17]. The combination of both is called a depthwise separable convolution (DSC) [3, 44] and allows depthwise convolutions to be used as 2D convolution approximators.

In this subsection, we show that switching to oriented 1D kernels leads to substantial speedups in computing both depthwise convolutions and DSCs.

To demonstrate this, notice that a DSC with $C'$ output channels is the mix of a depthwise convolution and of a pointwise convolution, with respective computational costs of $NCHWK^2$ and $NCHWC'$ Multiply-Adds (MADs). Normalizing by the input size $NCHW$, a DSC induces a cost of:

$$\frac{NCHWK^2 + NCHWC'}{NCHW} = K^2 + C'$$

By replacing the 2D kernel with a 1D kernel, the expression reduces to:

$$\frac{NCHWK + NCHWC'}{NCHW} = K + C'$$

According to our benchmarks Table 1 on an NVIDIA RTX 3090 with Pytorch 1.11, pointwise convolutions are at least 20× more efficient than depthwise convolutions. As such, depthwise convolutions account for more than 50% of the workload when $K^2 \geq C'/20$. For typical ConvNets, $C' \approx 10^3$, therefore the expression simplifies to $K \approx 7$, which is the kernel size used by ConvNeXt. This implies that we can expect a significant gain by replacing 2D kernels by 1D kernels on a network like ConvNeXt. This observation provides the basis for our study of oriented 1D kernels and motivates our search for fast implementations.

Table 1 provides a summary of the theoretical and practical speedups that we get from switching to oriented 1D kernels. First, we observe that replacing the 2D $7 \times 7$ kernel with the 1D $1 \times 31$ kernel reduces MADs by 58% even though we also switch to a larger kernel size. Second, we see that 1D kernels can be as efficient as 2D kernels: on PyTorch, switching to 1D leads to a 53% practical runtime
improvement which closely matches the theoretical 58% MAD improvement. Third, our implementation is more efficient than PyTorch: even though it supports oriented kernels, it is up to 51% faster for all angles. Finally, we demonstrate that oriented 1D kernels lead to significant overall speedups, as replacing the 2D 7×7 DSC with the oriented 1D K = 31 DSC reduces runtime as a whole by 35%.

4. Model Instantiation

To demonstrate the effectiveness of oriented 1D kernels, we gradually improve a ConvNeXt [32] architecture with 1D kernels. We propose 3 different models: ConvNeXt1D, ConvNeXt1D++ and ConvNeXt2D++ to show that oriented 1D ConvNets can be made as accurate as 2D ConvNets, and that we can use oriented 1D kernels to improve the accuracy of existing ConvNets. We structure our findings as follows: 1) Transition from 2D to 1D, 2) 1D-augmented design, 2) 1D/2D-mixed design.

4.1. ConvNeXt reference

Our models all use ConvNeXt [32] depicted in Figure 2 as base architecture, which can be decomposed into: 1. a stem layer, which downsamples an input image 4× before feeding it to the rest of the network. It constitutes what we call the stem design.

2. 4 stages each composed of a series of ConvNext blocks as defined by the block design. They are responsible for the bulk of the ConvNet and are typically very deep[18].

3. A downsampling layer separating each stage, inducing a hierarchical structure that defines ConvNets [45, 18].

We define the “2D Block design” and “2D Stem design” after ConvNeXt’s block and stem designs and present them in Figure 3a and Figure 3d. In short, the “2D Block design” used by ConvNeXt is an inverted bottleneck with depthwise convolutions [5], as proposed initially by MobileNetV2 [44]. It integrates GeLU [19] and LayerNorm [2], following similar adoption in transformers [11, 31]. ConvNeXt uses an aggressive downsampling strategy for its stem layer in the form of a 4×4 stride 4 convolution. ConvNeXt further innovates by using an even-sized 2×2 stride 2 convolution. Even-sized kernels tend to degrade performance because they introduce asymmetric padding [62, 55]. In our experience, modifying this layer leads to a drop in accuracy, which aligns with observations made in [32]. Consequently, we keep the original downsampling layer in our models.

In the following, we consider the ConvNeXt architecture and progressively improve the stem design and block design, using oriented 1D kernels. Table 2 presents a summary of our models and Figure 3 the designs that we use.

<table>
<thead>
<tr>
<th>Variant</th>
<th>MADs</th>
<th>Kernel Size K</th>
<th>Inference Runtime</th>
<th>Training Runtime</th>
<th>MADs Ratio</th>
<th>Runtime Ratio</th>
<th>Efficiency</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution (PyTorch)</td>
<td>C*K^2</td>
<td>7</td>
<td>34.0±0.2ms</td>
<td>78.4±0.3ms</td>
<td>0.096</td>
<td>2.3</td>
<td>24.3</td>
<td>9.7G</td>
</tr>
<tr>
<td>Depthwise 2D (PyTorch)</td>
<td>K^2</td>
<td>7</td>
<td>8.3±0.1ms</td>
<td>22.8±0.1ms</td>
<td>1.58</td>
<td>1.53</td>
<td>0.97</td>
<td>0.77G</td>
</tr>
<tr>
<td>Pointwise (PyTorch)</td>
<td>C'</td>
<td>1</td>
<td>4.2±0.1ms</td>
<td>9.8±0.1ms</td>
<td>1.0</td>
<td></td>
<td></td>
<td>9.7G</td>
</tr>
<tr>
<td>Depthwise 2D (PyTorch)</td>
<td>K</td>
<td>31</td>
<td>5.2±0.1ms</td>
<td>14.9±0.3ms</td>
<td>1.0</td>
<td>1.51</td>
<td>1.51</td>
<td>0.77G</td>
</tr>
<tr>
<td>Depthwise 1D (PyTorch)</td>
<td>K</td>
<td>31</td>
<td>4.2±0.1ms</td>
<td>14.9±0.3ms</td>
<td>1.0</td>
<td>1.51</td>
<td>1.51</td>
<td>0.77G</td>
</tr>
<tr>
<td>Oriented Depthwise 1D (Ours)</td>
<td>K</td>
<td>31</td>
<td>5.2±0.1ms</td>
<td>12.7±0.1ms</td>
<td>1.36</td>
<td>1.56</td>
<td>1.49</td>
<td>2.3G</td>
</tr>
<tr>
<td>2D Depthwise Separaied Conv. (DSC)</td>
<td>K^2+C'</td>
<td>7</td>
<td>12.7±0.1ms</td>
<td>37.4±0.2ms</td>
<td>1.03</td>
<td>1.56</td>
<td>1.49</td>
<td>2.3G</td>
</tr>
<tr>
<td>Oriented 1D DSC (Ours)</td>
<td>K+C'</td>
<td>31</td>
<td>8.4±0.1ms</td>
<td>23.8±0.2ms</td>
<td>1.03</td>
<td>1.56</td>
<td>1.49</td>
<td>2.3G</td>
</tr>
</tbody>
</table>

Table 2: Pairwise comparisons of the practical and theoretical performances of 2D and 1D convolutions. Oriented 1D kernels lead to significant speedups in theory and practice. We benchmark on a 56×56 input with FP32 on an NVIDIA RTX3090, against PyTorch[37] 1.11/CuDNN 8.2[4] with N = 64 and C = C' = 512. We compute the mean and standard deviation for 100 runs, preceded by 10 dry runs. MADs (Multiply-Adds) are divided by the input size NCHW for better readability. Inference Runtime measures only forward pass, Training Runtime includes backpropagation. Memory Usage measures peak memory usage difference before/after. Here, Efficiency means Runtime Ratio (practical gains) divided by MADs Ratio (theoretical gains), and measures how efficient an implementation is compared to theory (higher is better). For 1, we aggregate runs over all integer angles 0°, ..., 359° to demonstrate that our implementation is fast for all angles.

Figure 2: Illustration of ConvNeXt which acts as the base architecture for all of our models.

Table 2: Model Summary. We use ConvNeXt as baseline and propose ConvNeXt1D, a fully 1D ConvNet, ConvNeXt1D++, a 1D-augmented ConvNet, ConvNeXt2D++, a mixed 1D/2D ConvNet.
Oriented 1D kernels are very advantageous from a long-range modeling perspective. Because their cost is linear in \( K \), kernels of size \( 1 \times 31 \) are cheaper than 2D kernels of size \( 7 \times 7 \). The interest of having such large \( K \)'s is that the network is able to achieve global scale convolutions very early/deep in the network, in fact as soon as the 2nd stage. This stems from the observation that the input to stage 2 will be of size \( 28 \times 28 \) because of the downsampling operated at the stem layer and end of stage 1. As \( 31 > 28 \), this means that \( K \) is bigger than the input size, allowing it to encode all long-range dependencies at stage 2 and use the extra depth to model more complex spatial interactions. Note that we limit per-stage \( K \) to \([31, 31, 27, 15]\) or twice the stage input size, to avoid uninitialized weights.

Layer-wise rotation. We can improve the spatial mixing of oriented 1D kernels by adding an angular shift at every layer. We call this layer-wise rotation. By adding a layer-wise rotation of \( 90^\circ \) at every layer, we can reproduce a horizontal kernel in one layer and a vertical kernel in the next layer, effectively approximating a 2D kernel in a 2-layer setup. We integrate this idea in all of our models to improve performance, as validated by ablation Table 7.

Downsampling layer. We claim that we can make a 2D network fully 1D without actually changing \( 2 \times 2 \) downsampling layers. This is based on the observation that a \( 2 \times 2 \) kernel can be seen as the sum of a diagonal and anti-diagonal kernel, which are special cases of oriented 1D kernels. See more details in the supplemental material.

ConvNeXt1D. We can now combine all of these findings to define our fully oriented 1D network ConvNeXt1D. It is constructed on top of ConvNeXt with the Depthwise 1D Stem and 1D Block defined above, and uses a large kernel of size \( K = 31 \) with \( D = 8 \) directions, as suggested by our ablation study Table 8. We show in Section 5 that ConvNeXt1D is able to perform on par with ConvNeXt.

4.3. 1D-Augmented Block Design

With ConvNeXt1D, we have presented a 1D network that can compete with existing 2D networks. We now provide evidence that oriented 1D kernels can also improve the accuracy/computation trade-off of ConvNets by boosting performance for a negligible increase in FLOPS/parameters.

For that purpose, we propose the ConvNeXt1D+ model, constructed on top of ConvNeXt1D. ConvNeXt1D++ reuses the Depthwise 1D Stem that defines...
ConvNeXt1D and introduces the improved 1D++ Block design which builds upon 1D Block and is presented in Figure 3f. The idea is to reduce \( K = 31 \) to \( K = 15 \) and insert into the 1D Block a depthwise convolution of a large oriented \( 1 \times 31 \) kernel. This depthwise convolution is added residually to the network. We specifically target the inverted bottleneck layer as shown in Figure 3f, where we expect long-range dependencies to help the most. The goal is to integrate global scale interactions into local representations. With this design, we obtain better performance compared to ConvNeXt, even though the extra FLOPS/parameters represent less than 5\% of the total. See Section 5 for more results.

4.4. 1D/2D-Mixed Block Design

We now demonstrate the flexibility of our method by proposing the ConvNext2D++ model. This model is constructed by extending the 2D ConvNext architecture with a new block design, the 2D++ Block, as shown in Figure 3g. The 2D++ Block design is constructed on top of the ConvNeXt Block and adds to it the same 1D augmentations that were included in the 1D++ Block. Through this process, we show that oriented 1D kernels can be dropped in, as is, in existing architectures to improve long-range modeling and accuracy. Similarly as for ConvNeXt1D++, we get better performances than ConvNeXt, demonstrating that our approach is general and can be applied successfully to state-of-the-art models.

5. Experiments

In this section, we present experimental results conducted on ImageNet [9] image classification, and downstream tasks including ADE20K [65] semantic segmentation and COCO [29] object detection. We summarize our training setup below and present details in supplementary material.

**ImageNet Training.** We use the same training settings as ConvNeXt [32] to train our models. In summary, we train for 300 epochs with 20 epochs of linear warm-up, using the AdamW optimizer [33], data augmentations and regularizations like RandAugment [7], Mixup [61], Random Erasing [64], Cutmix [60], Stochastic Depth [21] and Label Smoothing [49]. The learning rate follows a cosine decay schedule and is set initially to 0.004. We use a batch size of 4096 and weight decay of 0.05.

5.1. Training Setup

**Model sizes.** We evaluate our models using the Tiny and Base complexities defined by ConvNeXt [32], which are parameterized by the channel sizes \( C = (C_1, C_2, C_3, C_4) \) and numbers of blocks \( B = (B_1, B_2, B_3, B_4) \) for all 4 stages. For our models using the Depthwise 1D Stem design, we also introduce the number of channels \( C_0 \) by which the stem layer initially expands the input, as described in Section 4.2. We name our model variants respectively ConvNeXt-T/B-1D, ConvNeXt-T/B-1D++ and ConvNeXt-T/B-2D++ and aggregate the configurations in Table 4.

**Table 4:** Model configurations, taken directly from ConvNeXt [32]

<table>
<thead>
<tr>
<th>Model</th>
<th>Image classification</th>
<th>Semantic segmentation</th>
<th>Object detection</th>
<th>Throughput (m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Params</td>
<td>FLOPs</td>
<td>Acc</td>
<td>EMA Acc</td>
</tr>
<tr>
<td>Tiny/T</td>
<td>C_0</td>
<td>C_1</td>
<td>C_2</td>
<td>C_3</td>
</tr>
<tr>
<td>Base/B</td>
<td>C_0</td>
<td>C_1</td>
<td>C_2</td>
<td>C_3</td>
</tr>
</tbody>
</table>

5.2. Experimental Results

**Image classification.** Table 3 compares our models against state-of-the-art ConvNets like ConvNeXt [32], Re-

<table>
<thead>
<tr>
<th>Model</th>
<th>Image classification</th>
<th>Semantic segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConvNeXt-T [32]</td>
<td>#Params</td>
<td>FLOPs</td>
</tr>
<tr>
<td>Slab-T [30]</td>
<td>28.6M</td>
<td>4.5G</td>
</tr>
<tr>
<td>ConvNeXt-T (reprod.)</td>
<td>28.6M</td>
<td>4.5G</td>
</tr>
<tr>
<td>ConvNeXt-1D++</td>
<td>28.5M</td>
<td>4.4G</td>
</tr>
<tr>
<td>ConvNeXt-T-2D++</td>
<td>29.2M</td>
<td>4.7G</td>
</tr>
<tr>
<td>ConvNeXt-B-1D</td>
<td>98M</td>
<td>15.4G</td>
</tr>
<tr>
<td>ConvNeXt-B-2D++</td>
<td>91M</td>
<td>15.9G</td>
</tr>
<tr>
<td>ReplKNet-31IB [12]</td>
<td>79M</td>
<td>15.3G</td>
</tr>
<tr>
<td>SlabK-B [30]</td>
<td>95M</td>
<td>17.1G</td>
</tr>
<tr>
<td>ConvNeXt-B (reprod.)</td>
<td>89M</td>
<td>15.4G</td>
</tr>
<tr>
<td>ConvNeXt-B-1D</td>
<td>88M</td>
<td>15.3G</td>
</tr>
<tr>
<td>ConvNeXt-B-1D++</td>
<td>90M</td>
<td>15.8G</td>
</tr>
<tr>
<td>ConvNeXt-B-2D++</td>
<td>91M</td>
<td>15.9G</td>
</tr>
</tbody>
</table>

Table 3: Experimental results on ImageNet image classification, ADE20K semantic segmentation using UperNet, and COCO object detection using Cascade Mask R-CNN. Overall, our models are competitive with ConvNeXt and state-of-the-art ConvNets on all tasks. Note that for downstream tasks, we use non-EMA backbones which means that performance improvements are not attributable to increased classification accuracy. Backbones are pre-trained on ImageNet-1K. We report image classification results with and without Exponential Moving Average (EMA). † SlabK only provides single-scale segmentation results. Throughput measures the inference speed of our proposed models on 1 NVIDIA RTX3090, PyTorch substitutes oriented kernels with competitive PyTorch horizontal kernels. FLOPs are computed on input sizes 224\textsuperscript{2}, 1280 \times 800 and 2048 \times 512 respectively.
We fine-tune UperNet [56] on the ADE20K [65] dataset. We follow the same setup as ConvNeXt, that is we preserve all parameters apart from the backbone. Similarly as ConvNeXt, we use model non-EMA weights and multi-scale training for 160k iterations with a 512 crop size. We present our semantic segmentation results in Table 3. Overall, ConvNeXt-1D Tiny lags behind ConvNeXt but Base catches up to it. The augmented networks achieve higher mIoU than ConvNeXt-T/B by at least +0.8 / +0.8 mIoU. Note that for Base models, non-EMA image classification accuracy does not change: because our networks do not use EMA backbones, this means that the IoU improvement is not attributable to better image classification, but to the choice of architecture, thereby validating the effectiveness of oriented kernels on Base models.

Object detection. We follow ConvNeXt and fine-tune a Cascade Mask R-CNN [26] on the COCO dataset using multi-scale training and a 3×3 schedule. As shown in Table 3, ConvNeXt-1D achieves similar or better performances compared to ConvNeXt. The results confirm the expressiveness of augmented networks: they have +0.5 AP$_{bbox}$ and +0.6 AP$_{mask}$ compared to both RepLKNet [12] and ConvNeXt.

Inference speed metrics. We complement our classification results with inference speeds on a 224 × 224 input, batch size 64, and 1 NVIDIA RTX 3090, as presented in Table 3. Even though 1D kernels are faster than 2D and the FLOPs increase is negligible, our ConvNeXt models are slower because of the modified Stem and Block designs. This suggests a model-level under-utilization due to memory related bottlenecks, which we will study as future work. To confirm our computational analysis, we replace oriented kernels with PyTorch 1D kernels. Note that we fuse certain 1D operations which improves 1D speed over 2D.

Additional architectures. To provide evidence that our 1D convolutions are generalizable, we compare the 2D ConvNets MobileNetV3 [13] and ConvMixer [52] against 1D equivalents, obtained by replacing every non-strided depthwise convolution with an oriented kernel. We train on ImageNet using the same setup as ConvNeXt. 1D models have 10%-25% higher inference throughput versus 2D models but lead to a drop of 0.8%-1% accuracy. Given that only substitution is done and no other tricks are used, this shows that 1D kernels show promise in working in other settings and architectures. Note that we target specifically an architecture which presents depthwise separable convolutions.

5.3. CUDA kernel speed comparison

We compare the speed of our oriented 1D kernel implementation versus the competitive PyTorch and RepLKNet CUTLASS [35] implementations and aggregate the results in Table 5. We conduct the experiments on a single NVIDIA RTX 3090, using single-precision FP32. In some instances, our custom kernel outperforms the CuDNN/CUTLASS implementation by a substantial margin. These algorithms rely on the assumption that computation is the main bottleneck, thus their design focuses on maximizing computational throughput in a cache-friendly manner, at the expense of reading more data. As a result, they are slower for smaller kernel sizes where data efficiency is critical. This also makes them hard to adapt to oriented kernels because they assume linear access patterns which results in bad performance, as we show in supplementary material.

<table>
<thead>
<tr>
<th>Model</th>
<th>Epochs</th>
<th>K</th>
<th>#Params</th>
<th>FLOPs</th>
<th>Throughput</th>
<th>Acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobileNetv3-small 2D</td>
<td>120</td>
<td>5.5M</td>
<td>0.06G</td>
<td>6800 img/s</td>
<td>60.9</td>
<td></td>
</tr>
<tr>
<td>MobileNetv3-small 1D</td>
<td>120</td>
<td>5.5M</td>
<td>0.06G</td>
<td>7200 img/s</td>
<td>60.1</td>
<td></td>
</tr>
<tr>
<td>ConvMixer-768/32 2D</td>
<td>100</td>
<td>5.5M</td>
<td>21.1M</td>
<td>30000 img/s</td>
<td>78.4</td>
<td></td>
</tr>
<tr>
<td>ConvMixer-768/32 1D</td>
<td>100</td>
<td>5.5M</td>
<td>20.3M</td>
<td>20000 img/s</td>
<td>77.3</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Runtime comparison of our oriented 1D implementation on 1 NVIDIA RTX 3090 for N = 64, C = 512, FP32. The mean is taken over 100 runs, preceded by 10 dry runs. We benchmark against the very competitive PyTorch/CuDNN and RepLKNet CUTLASS [35] implementations on horizontal convolutions. Our implementation outperforms PyTorch consistently regardless of angle θ thanks to intelligent data access patterns, but falls off when computation becomes a bottleneck.
5.4. Ablation study

In this subsection, we carry out ablations to determine the influence on accuracy of characteristic oriented 1D kernel parameters. Table 6 compares oriented 1D models against non-oriented 2D baselines, Table 7 studies the impact of layerwise rotation, Table 8 studies the best combination of kernel size and direction and Table 9 looks at the best $C_0$. We comment the results in the next subsection.

<table>
<thead>
<tr>
<th>Design</th>
<th>Block Design</th>
<th>K</th>
<th>D</th>
<th>#Params</th>
<th>FLOPs</th>
<th>Top-1 Acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D</td>
<td>2D</td>
<td>7^2</td>
<td>28.6M</td>
<td>4.5G</td>
<td>82.0</td>
<td></td>
</tr>
<tr>
<td>2D</td>
<td>2D</td>
<td>31^2</td>
<td>30.0M</td>
<td>5.8G</td>
<td>81.6</td>
<td></td>
</tr>
<tr>
<td>Oriented 1D</td>
<td>1D</td>
<td>7</td>
<td>8</td>
<td>28.3M</td>
<td>4.4G</td>
<td>81.5</td>
</tr>
<tr>
<td>Oriented 1D</td>
<td>1D</td>
<td>31</td>
<td>8</td>
<td>28.5M</td>
<td>4.4G</td>
<td>81.8</td>
</tr>
<tr>
<td>Depthwise 2D</td>
<td>2D</td>
<td>7^2</td>
<td>28.6M</td>
<td>3.9G</td>
<td>82.0</td>
<td></td>
</tr>
<tr>
<td>Depthwise 2D</td>
<td>1D</td>
<td>7</td>
<td>8</td>
<td>28.3M</td>
<td>4.4G</td>
<td>82.0</td>
</tr>
<tr>
<td>Depthwise 1D</td>
<td>1D</td>
<td>31</td>
<td>8</td>
<td>28.5M</td>
<td>4.4G</td>
<td>82.2</td>
</tr>
<tr>
<td>Depthwise 2D+2D</td>
<td>2D+2D</td>
<td>7^2+7^2</td>
<td>29.9M</td>
<td>4.3G</td>
<td>82.1</td>
<td></td>
</tr>
<tr>
<td>Depthwise 1D++</td>
<td>1D++</td>
<td>15+31</td>
<td>8</td>
<td>29.2M</td>
<td>4.7G</td>
<td>82.7</td>
</tr>
<tr>
<td>2D</td>
<td>2D++</td>
<td>7^2+31</td>
<td>8</td>
<td>29.4M</td>
<td>4.7G</td>
<td>82.5</td>
</tr>
</tbody>
</table>

Table 6: Comparison of oriented 1D models against 2D kernel baselines on ConvNeXt-T. Depthwise 2D is the 2D transposition of Depthwise 1D and 2D+2D is the 2D transposition of both 1D++ and 2D++ block designs. Overall, by switching to 1D and changing the stem, we are able to consistently match the performance of 2D baselines. We discuss these results in Section 5.5.

<table>
<thead>
<tr>
<th>K</th>
<th>D</th>
<th>Layer-wise rotation</th>
<th>Top-1 Acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>4</td>
<td>0°</td>
<td>82.01</td>
</tr>
<tr>
<td>31</td>
<td>4</td>
<td>alternating 90°</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: Layer-wise rotation on ConvNeXt-1D. We see that adding layer-wise rotation increases accuracy.

5.5. Discussion

No loss in expressiveness. 1D ConvNeXt networks are consistently able to perform on par or outperform their 2D equivalents according to Table 6. This shows that oriented 1D kernels can be made as expressive as 2D kernels provided they are integrated the right way.

Depthwise 1D stem. From Table 6, we see that our proposed Depthwise 1D Stem plays a key role in making 1D networks competitive with 2D networks, by reintroducing the stem spatial parameters that were lost switching to 1D.

Better long-range scaling with 1D. Table 6 shows that 2D models are unable to benefit from large kernel sizes $K$. This contrasts with the 1D kernel case and implies that 1D kernels exhibit better long-range scaling than 2D kernels.

6. Conclusion

Our study suggests that oriented 1D kernels are viable alternatives to 2D kernels as they can be just as expressive, and are empirically better suited for larger kernel sizes.

Acknowledgement This work was partially supported by the National Science Foundation under Award IIS-1942981.
References

national Conference on Learning Representations, 2015. 2, 4


[52] Asher Trockman and J. Zico Kolter. Patches are all you need?, 2022. 7


[60] Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. CutMix: Regularization Strategy to Train Strong Classifiers With Localizable Features. pages 6023–6032, 2019. 6


