Convolutional Networks with Oriented 1D Kernels: Supplementary Material

In this supplementary material, we present additional experiments including ERF analysis in Section 1 and sparsity in Section 2 and present additional training plots in Section 3.

We also provide a more theoretical overview of oriented 1D kernels, and describe how we come up with our formulation of oriented 1D kernels in Section 4, discuss design choices of oriented 1D kernels in Section 5, prove that a 2×2 downsampling layer can be seen as a sum of oriented kernels in Section 6, outline how to learn orientation in Section 7 and establish a connection with anisotropic gaussian kernels in Section 8.

Finally, we provide implementation notes in Section 9, describe training settings in Section 10 and outline limitations in Section 11.

1. Effective Receptive Field (ERF) Analysis

In this section we provide an analysis of the ERF of our models. According to [15], ERFs scale in $O(K\sqrt{L})$ which is linear in the kernel size K and only sub-linear in depth L. This demonstrates the advantage of kernel size over depth.

We follow the approaches provided by RepLKNet [8] and SLaK [13] to compute the ERF of our models. We sample 1000 images resized to 1024×1024 from the ImageNet validation set, and for each pixel of every image we compute its contribution to the central point of the feature map generated in the last layer. We then average the contribution across all input channels and images. Results are shown in Figure 1.

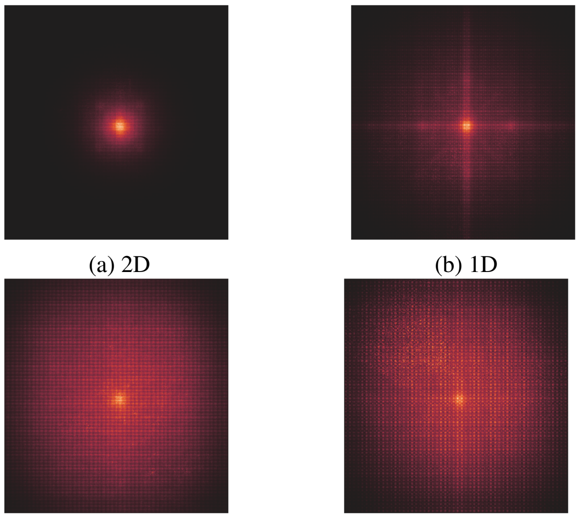

(a) 2D (b) 1D (c) 2D++ (d) 1D++

Figure 1: Effective Receptive Field (ERF) for *Base* models. A more widely distributed colored area indicates a larger ERF. We see that our networks increase the ERF significantly whilst preserving the overall shape.

From Figure 1, we see that ConvNeXt has a concentrated ERF around the input center. This is different from ConvNeXt-1D, which has a significantly wider ERF. It manages to preserve the concentration around the center whilst widening the receptive field, albeit at the cost of introducing a horizontal/vertical bias. This is fixed by the augmented networks, which are able to preserve the concentration around the center and introducee a wide circular receptive field. Our networks are therefore able to balance the focus between local details and the attention to long-range dependencies.

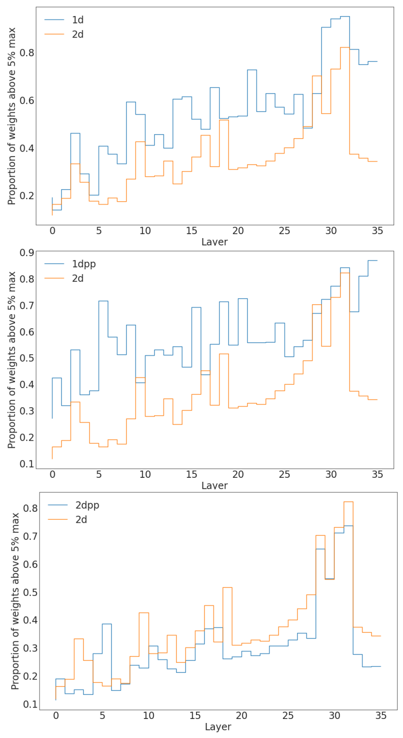
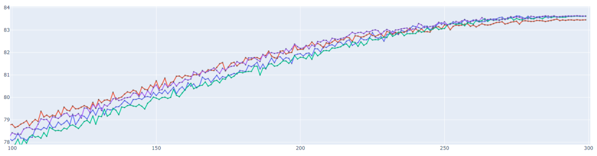
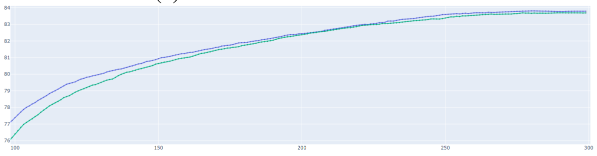


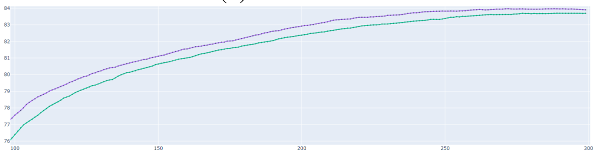
Figure 2: Sparsity analysis on *Base* models: Proportion of weights in absolute value above 5% of maximum. We consider only the 1st depthwise convolution in each layer of stages 1-4. We see that fully 1D networks (1D and 1D++) are up to +20% more dense than 2D networks (2D and 2D++).



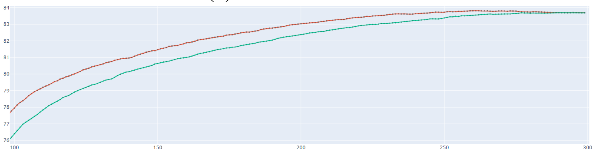
(a) no-EMA 1D/1D++/2D/2D++



(b) EMA 1D/2D



(c) EMA 2D++/2D



(d) EMA 1D++/2D

Figure 3: ImageNet EMA accuracy plots for *Base* models. 2D in green, 2D++ in purple, 1D in blue, 1D++ in red.

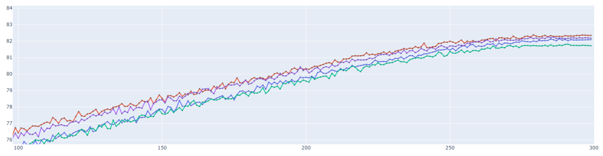
2. Sparsity analysis

In this section, we analyze the sparsity introduced by the use of oriented 1D kernels in our 1D, 1D++ and 2D++ models. We consider only the 1st depthwise convolution in each layer of stages 1-4. This depthwise convolution is present and common to all block designs. From Figure 2 we see that fully 1D networks (1D and 1D++) are up to +20% more dense than 2D networks (2D and 2D++). This is to be expected as we decrease the number of parameters from 7×7 to 1×31 by design. It also confirms the usefulness of oriented 1D kernels for introducing large kernels in ConvNets: not only do they widen the receptive field, but they also reduce the number of spatial parameters required to do so, thereby enabling more efficient learning.

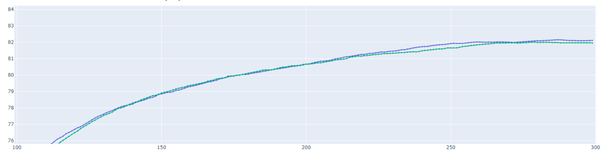
3. Training plots

In this section, we plot ImageNet validation set accuracies of *Base* models in Figure 3. We also aggregate statistics on these trajectories in Table 1. We see that 1D++/2D++ models exhibit +0.6 and +0.5 EMA accuracy improvements versus ConvNeXt when averaged over epochs 150-250. This suggests that augmented 1D networks perform better than ConvNeXt during training.

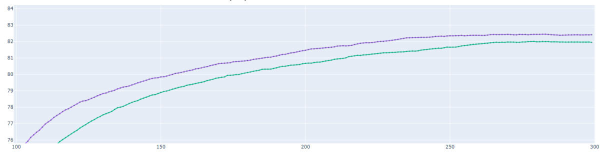
For the 1D++ *Base* model, this accuracy improvement during training is significantly larger compared to the +0.1 final accuracy difference versus ConvNeXt. This discrepancy suggests that our 1D++ model overfits and underperforms versus what it is expected to achieve. We have not fine-tuned any hyperparameter other than 1D parameters, to enable fair comparisons with ConvNeXt. We think that understanding the causes of this overfitting would enable 1D networks to perform better than they currently do on larger scales. We believe that they should be able to attain the +0.5 training average improvement presented in Table 1, as this bound is already achieved by *Tiny* models.



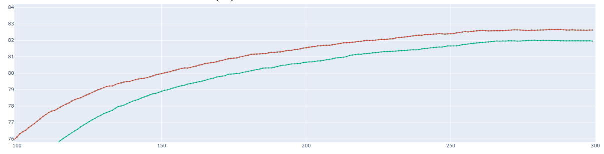
(a) no-EMA 1D/1D++/2D/2D++



(b) EMA 1D/2D



(c) EMA 2D++/2D



(d) EMA 1D++/2D

Figure 4: ImageNet EMA accuracy plots for *Tiny* models. 2D in green, 2D++ in purple, 1D in blue, 1D++ in red.

Model	Acc. Difference Epochs 150-250	Acc. Difference Final	Discrepancy
ConvNeXt-T	/	/	/
ConvNeXt-T-1D	+0.0	+0.2	+0.2
ConvNeXt-T-1D++	+0.8	+0.7	-0.1
ConvNeXt-T-2D++	+0.7	+0.5	-0.2
ConvNeXt-B	/	/	/
ConvNeXt-B-1D	+0.1	+0.1	0.0
ConvNeXt-B-1D++	+0.6	+0.1	-0.5
ConvNeXt-B-2D++	+0.5	+0.3	-0.2

Table 1: Accuracy differences w.r.t ConvNeXt. On the left, averaged over epochs 150-250, on the right, accuracy at epoch 300. *Discrepancy* measures the difference between these 2 quantities. We see that there is a strong discrepancy for ConvNeXt-B-1D++, which suggests that it underperforms.

4. Oriented 1D kernels: Mathematical formulation

In this section, we present the intuition behind the mathematical formulation of oriented 1D depthwise convolutions. We start by introducing oriented kernels in the more general 2D setting, discuss the intuition behind the formula and specialize it to the 1D case.

To simplify the problem, we will only consider a single angle θ but the formulation can be easily generalized to support a per-channel angle $\boldsymbol{\theta} \in \mathbb{R}^C$ as done in the paper.

4.1. 2D formulation

Let $\mathbf{x} \in \mathbb{R}^{N \times H \times W \times C}$ denote the input, $\mathbf{w} \in \mathbb{R}^{R \times S \times C}$ the filter and $\mathbf{y} \in \mathbb{R}^{N \times P \times Q \times C}$ the output of the depthwise convolution. N is the batch size, C the number of channels, H, W the input height and width, P, Q the output height and width and R, S the filter height and width. Let (pad_h, pad_w) be the padding, (str_h, str_w) the stride and θ the angle of the oriented kernel.

Definition 1 *We define a depthwise convolution of an oriented 2D kernel as:*

$$\forall n \in [\![0, N-1]\!], p \in [\![0, P-1]\!], q \in [\![0, Q-1]\!], c \in [\![0, C-1]\!],$$

$$y_{npqc} = \sum_{0 \le r < R, 0 \le s < S} x_{nhwc} w_{rsc} \tag{1}$$

$$\textit{where} \begin{pmatrix} h \\ w \end{pmatrix} = \mathbf{R}_\theta \begin{pmatrix} r - pad_h \\ s - pad_w \end{pmatrix} + \begin{pmatrix} p \cdot str_h \\ q \cdot str_w \end{pmatrix} \tag{2}$$

$$\textit{and } \mathbf{R}_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \textit{ is a rotation matrix of angle } \theta \tag{3}$$

We refer to Equation (2) as the *coordinate equation*. Because h, w, p, q, r, s are all integers we need to introduce a discretization scheme for this formula to make sense. We will ignore this issue for now and discuss it in Section 4.2.

Intuition. The goal of Equation (2) is to rotate the filter by an angle θ, which is accomplished by introducing the rotation matrix $\mathbf{R}_\theta$. Intuitively, every increment of r or s results in an increase by $(\cos\theta, \sin\theta)^T$ or $(-\sin\theta, \cos\theta)^T$ of the left-hand side of Equation (2) after rotation by $\mathbf{R}_\theta$. By doing so, we are effectively changing the direction of the increment of r and s from the vertical and horizontal axes to arbitrary oriented axes.

Convolution origin. The origin of the convolution is obtained by taking $r = s = 0$: we see that it is the same as the non-oriented case. This is guaranteed by our choice of Equation (2) and is not necessarily preserved by other formulations.

4.2. Discretization and Interpolation

Equation (2) makes the assumption that we can sample from a continuous input domain. However, due to the discrete nature of the input and filter, it becomes necessary to introduce a *discretization* scheme.

The simplest discretization scheme consists of rounding down the coordinates on the right-hand side of Equation (2): this approach is fast but results in coarse approximations of orientation.

An alternative approach would be to do bilinear interpolation which allows us to account for finer angles. However, bilinear interpolation increases the MADs by a factor of 4× and results in at least 2× the runtime on our internal benchmarks Table 2. This makes an oriented 1×31 kernel 2× as expensive as a 7×7 kernel.

In this paper, we choose to adopt the round-down discretization scheme, to showcase the usefulness and practicality of oriented 1D kernels. Under this scheme, Equation (2) becomes:

$$\begin{pmatrix} h \\ w \end{pmatrix} = \left\lfloor \mathbf{R}_\theta \begin{pmatrix} r - pad_h \\ s - pad_w \end{pmatrix} \right\rfloor + \begin{pmatrix} p \cdot str_h \\ q \cdot str_w \end{pmatrix} \tag{4}$$

As discussed in this section, there exists many valid discretization schemes, and we leave their exploration as future work.

Implementation	K	Inference Runtime	Total Runtime
Round-down 1D	31	4.2±0.1ms	9.9±0.1ms
Bilinear 1D	31	8.9± 0.1ms	22.3±0.1ms
2D	7	8.3±0.1ms	22.6±0.1ms

Table 2: Runtime comparison between bilinear interpolation and other approaches. Bilinear interpolation results in a 2× speed reduction. The tests are done on an NVIDIA RTX 3090 for $N = 64, C = 512, H = W = 56$, aggregated over 100 runs preceded by 10 dry runs. *2D* refers to a PyTorch/CuDNN 2D depthwise convolution. *Inference Time* measures only forward pass, *Training Time* includes backpropagation.

4.3. 1D formulation

Oriented depthwise 1D convolutions are specializations of Equation (2) for $r = pad_h = 0$, $pad_w = pad$ and $str_h = str_w = str$. We use round-down as our discretization, as shown in Equation (4), which results in the following equation:

$$\begin{pmatrix} h \\ w \end{pmatrix} = str \cdot \begin{pmatrix} p \\ q \end{pmatrix} + \left\lfloor (k - pad) \cdot \begin{pmatrix} -\sin\theta \\ \cos\theta \end{pmatrix} \right\rfloor \tag{5}$$

where K is the kernel size and k varies from 0 to $K - 1$. This is the formulation of oriented 1D kernels that we have considered in the paper.

From now on, we will restrict our study to Equation (5).

5. Oriented 1D kernels: Design choices

Let us now delve into the design choices regarding oriented 1D kernels.

5.1. Padding

The way we express padding in Equation (5) turns out to be essential in preserving accuracy. If we were to naively replicate the non-oriented case and write Equation (5) as:

$$\begin{pmatrix} h \\ w \end{pmatrix} = str \cdot \begin{pmatrix} p \\ q \end{pmatrix} + \left\lfloor k \cdot \begin{pmatrix} -\sin\theta \\ \cos\theta \end{pmatrix} \right\rfloor - pad \tag{6}$$

then the resulting formula would lead to different behaviors for vertical and horizontal convolutions and more generally result in uncentered convolutions. In turn, this greatly degrades accuracy. In contrast, Equation (5) adapts naturally to different angles and for $pad = \lfloor \frac{K}{2} \rfloor$ does not result in accuracy degradation. We consider only odd K in our experiments and fix $pad = \lfloor \frac{K}{2} \rfloor$ to obtain centered oriented convolutions.

5.2. Rotation vs Shearing

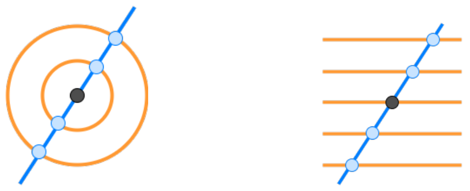
Instead of applying a rotation to the coordinates of the convolution we can consider more relevant transformations. Let's first rewrite Equation (5) as:

$$\begin{pmatrix} h \\ w \end{pmatrix} = str \cdot \begin{pmatrix} p \\ q \end{pmatrix} + \left\lfloor \begin{pmatrix} \delta h^k \\ \delta w^k \end{pmatrix} \right\rfloor \tag{7}$$

in terms of the *filter offsets*: $$\begin{pmatrix} \delta h^k \\ \delta w^k \end{pmatrix} = \mathbf{R}_\theta \begin{pmatrix} 0 \\ k - pad \end{pmatrix} \tag{8}$$

Intuitively, the *filter offset* $(\delta h^k, \delta w^k)^T$ is the offset in the input grid where we apply the filter weight w_k, relative to the convolution origin.

Instead of sampling *filter offsets* on concentric circles with increasing and regular k radii, we can choose to sample points lying on integer rows or columns. The first approach is equivalent to *rotating* the filter axis as shown in Figure 5a, whereas the second can be seen as *shearing* the filter axis parallel to the columns or rows as depicted in Figure 5b.



(a) Rotation (b) Shear

Figure 5: Different Parameterizations

The usefulness of formulation 5b becomes clear on the following example. For $\theta = -45°$, $pad = 0$ and k varying between 0 and $K-1$, formulation 5a outputs non-integer filter offsets $(\delta h^k, \delta w^k)^T = k(\sqrt{2}/2, \sqrt{2}/2)^T$, whereas formulation 5b outputs integer filter offsets $(\delta h^k, \delta w^k)^T = k(1,1)^T$. In this particular case, the latter approach removes the need for discretization.

More generally, formulation 5a results in filter offsets $(-k\sin\theta, k\cos\theta)^T$. With rounding down as discretization, this is sub-optimal as it leads to some level of redundancy in filter offsets.

Formulation 5b is more complicated to handle and gives rise to two separate cases. If we intersect the filter axis with integer columns we get filter offsets $(-k\tan\theta, k)^T$. If we instead intersect with integer rows, we get offsets $(k, -k\cot\theta)^T$. By forcing one coordinate to be an integer we remove the redundancy encountered in formulation 5a but we enlarge the kernel "length" and introduce higher complexity due to the separate handling of cases.

Mathematically we can introduce the *shear matrices* $\mathbf{S}^x_\theta = \begin{pmatrix} 1 & -\tan\theta \\ 0 & 1 \end{pmatrix}$ and $\mathbf{S}^y_\theta = \begin{pmatrix} 1 & 0 \\ \cot\theta & 1 \end{pmatrix}$ and rewrite filter offsets in terms of $\mathbf{S}^x_\theta$ and $\mathbf{S}^y_\theta$ instead of $\mathbf{R}_\theta$.

More thorough testing is necessary to compare both formulations, and is left for future work.

6. Proof that the downsampling layer is the sum of 2 oriented kernels

In this section, we prove the claim that we can express a downsampling layer as the sum of 2 oriented kernel convolutions by using an even-sized kernel *specialization* of Equation (5) and carefully choosing the padding.

Oriented convolution. First, we define a vanilla convolution with an oriented 1D kernel as:

$$y_{npqg} = \sum_{0 \le c < C, 0 \le r < R, 0 \le s < S} x_{nhwc} w_{grsc} \tag{9}$$

where g denotes the output channel and $(h,w)^T$ verifies the *coordinate* Equation (5). Note that we suppose $\boldsymbol{\theta} = \theta_{gc}$ so that the angle can vary both with respect to the output and input channels.

Even-sized kernel. Next, we generalize Equation (5) for even-sized kernels by introducing an exterior padding (pad_h, pad_w) which takes care of the padding asymmetry introduced by even-sized kernels [23].

$$\begin{pmatrix} h \\ w \end{pmatrix} = str \cdot \begin{pmatrix} p \\ q \end{pmatrix} + \left\lfloor (k - pad) \cdot \begin{pmatrix} -\sin\theta \\ \cos\theta \end{pmatrix} \right\rfloor - \begin{pmatrix} pad_h \\ pad_w \end{pmatrix} \tag{10}$$

Proof. Claim 1: We claim that a 2×2 downsampling layer can be decomposed as the sum of a diagonal and anti-diagonal convolution. This follows from the linearity of a convolution: if W_1 and W_2 are two 2D kernels and I is any input, then the convolution of I by $W = W_1 + W_2$ equals the sum of the convolutions of I by W_1 and convolution of I by W_2. By defining W_1 as the 2×2 diagonal kernel, and W_2 as the anti-diagonal kernel, this implies that the sum $W = W_1 + W_2$ can express any 2×2 kernel, which proves our claim.

Claim 2: We now claim that both diagonal and anti-diagonal kernels can be seen as even-sized oriented 1D kernels as defined in Equation (10). We do this by carefully choosing θ and paddings pad, pad_h and pad_w. In fact, a diagonal kernel can be expressed using $\theta = -\frac{\pi}{4}, pad = 1 - \sqrt{2}$ and $pad_h = pad_w = 0$. Similarly, an anti-diagonal kernel can be expressed using $\theta = \frac{\pi}{4}, pad = 1-\sqrt{2}, pad_h = 1$ and $pad_w = 0$. Contrary to the diagonal kernel, we see that the anti-diagonal kernel requires the introduction of a vertical padding $pad_h = 1$. As explained earlier, this is necessary because of the asymmetry of even-sized kernels.

Summary: We can now combine claims 1 and 2 to deduce that a downsampling layer can be seen as the sum of 2 oriented kernel convolutions.

In this section, we have introduced even-sized oriented kernels. We leave their exploration as future work.

7. Angle backpropagation

Instead of seeing θ as a fixed parameter, we can instead try to learn it. To accomplish this, it becomes necessary to introduce a formulation of Equation (5) which is differentiable w.r.t θ.

The first idea is to replace the sum over the kernel size k as a sum over (h, w, k) and introduce a soft distribution $\omega_{pqhwk}(\theta)$ to weight the sum, as such:

$$\mathbf{y}_{npqc} = \sum_{h,w,k} \omega_{pqhwk}(\theta)\mathbf{x}_{nhwc}\mathbf{w}_{ck} \tag{11}$$

Here h and w are bound variables. Contrast with the original formulation where h and w are functions of k as provided by Equation (5)

$$\mathbf{y}_{npqc} = \sum_{k} \mathbf{x}_{nhwc}\mathbf{w}_{ck} \tag{12}$$

The second idea is to relax the Equation (5) constraint to include non-zero contributions for (h, w, k) which do not verify Equation (5) but are close to it. A natural choice is to model $\omega_{pqhwk}(\theta)$ as a gaussian distribution with variance σ^2 over the squared Equation (5) error, as illustrated in Figure 6.

More formally, we introduce:

$$\omega_{pqhwk}(\theta) = \exp(-\frac{(h-\alpha_{pk}(\theta))^2 + (w-\beta_{qk}(\theta))^2}{2\sigma^2})$$
$$\text{where} \begin{cases} \alpha_{pk}(\theta) = p\cdot str_h - (k - pad)\cdot\sin\theta \\ \beta_{qk}(\theta) = q\cdot str_w + (k-pad)\cdot\cos\theta \end{cases} \tag{13}$$

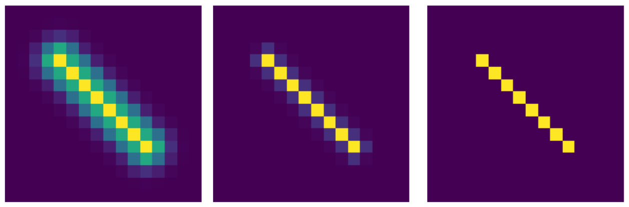

(a) $\sigma = 1.0$ (b) $\sigma = 0.5$ (c) $\sigma = 0.25$

Figure 6: Example of soft distribution for a diagonal kernel

Modulo discretization, this formulation generalizes Equation (12). Indeed, it can be seen as Equation (11) for the special case $\sigma = 0$ or in other words

$$\omega_{pqhwk}(\theta) = 1(f_{p,k}(\theta) = h)1(g_{q,k}(\theta) = w)$$

By introducing Equation (11) and our parameterization (7) of ω, we have described a differentiable formulation of Equation (12) which is differentiable w.r.t θ.

Intuitively, σ models the error in constraint Equation (5) and consequently how much we want neighboring pixels to influence the output of the convolution. By varying σ we control the width of this band as well as the pixels that have an influence over the optimization of θ.

However, by increasing σ, we sacrifice speed: the computational cost rises from $O(K)$ to $O(HWK)$. This can be offset to $O(r^2K)$ by leveraging the exponential decay of ω to cut off contributions below a threshold $\exp(-\frac{r^2}{2\sigma^2})$. r can be seen as the radius by which we expand the kernel.

Optimizing θ is beyond the scope of the paper and this section is provided only for reference. We leave exploration of this idea as future work.

8. Connection with 2D Gaussian anisotropic filters

In this section, we make a parallel between oriented 1D kernels and gaussian anisotropic filters. Before going further, let's first introduce 2D gaussian filters, which are commonly used in signal processing [2] as:

Definition 2 *A 2D gaussian filter $k(\mathbf{x}, \mathbf{y})$ with positive semi-definite covariance matrix $\mathbf{\Sigma} \in \mathbb{R}^{2\times2}$ is defined as:*

$$k(\mathbf{x},\mathbf{y}) = \frac{1}{\sqrt{2\pi}|\mathbf{\Sigma}|^{\frac{1}{2}}}\exp(-\frac{1}{2}(\mathbf{x}-\mathbf{y})^T\mathbf{\Sigma}^{-1}(\mathbf{x}-\mathbf{y}))$$

for given $\mathbf{x}, \mathbf{y} \in \mathbb{R}^2$. (14)

1. *In the case where $\mathbf{\Sigma} = \sigma\mathbf{I}_2$ for a given $\sigma \geq 0$, we say that the filter is* isotropic.
2. *In the case where $\mathbf{\Sigma} = Diag(\sigma_1, \sigma_2)$ we say that the filter is* orthogonal.
3. *Otherwise, the filter is said to be* anisotropic.

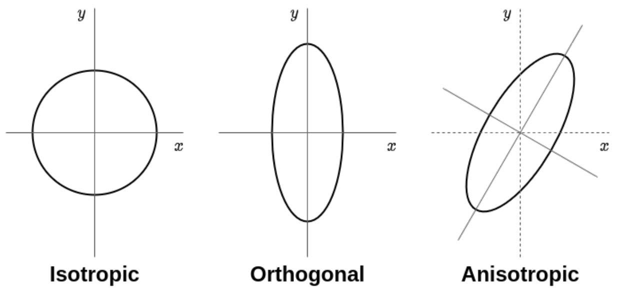


Figure 7: Examples of 2D gaussian filters.

8.1. Parallel with gaussian anisotropic filters

In gaussian filtering literature [9], it is commonly stated that 2D isotropic and orthogonal gaussian filters are separable and can be decomposed as the recursion of a horizontal 1D gaussian filter and of a vertical 1D gaussian filter. In practice, gaussian filters are used in the form of finite-sized convolutions. This means that 2D isotropic/orthogonal kernels can be decomposed as the recursion of a horizontal and vertical convolution.

According to [9, 12], this fact is not limited to "axis-aligned" gaussian filters: any 2D gaussian filter can be expressed as the recursion of two *oriented 1D* gaussian filters, as dictated by the eigenvectors of its covariance matrix. Consequently, the combination of two oriented 1D convolutions can represent *any* 2D gaussian filter including anisotropic ones, thereby suggesting that oriented 1D kernels are more expressive than non-oriented kernels. The situation is summarized in Table 3.

Convolution kernel	Isotropic	Orthogonal	Anisotropic
Non-oriented	**Yes**	**Yes**	*No*
Oriented	**Yes**	**Yes**	**Yes**

Table 3: Summary of the expressiveness of oriented and non-oriented convolution kernels

9. Implementation Notes

In this section, we look at the challenges involved in implementing fast depthwise convolutions for oriented 1D kernels. We present here two main approaches to implement oriented kernels: the *filter rotation* approach, which rotates the kernel, and the *input rotation* approach, which keeps the kernel fixed and instead rotates the input in the opposite direction. Our best algorithm runs up to 50% faster than PyTorch on 1 NVIDIA RTX 3090, as presented in the paper.

9.1. CUTLASS GEMM with Rotated Filter

Our goal is to design an implementation that works for large kernel sizes $K \geq 7$. To that effect, our first proposed implementation extends an open-source library that achieves state-of-the-art performance on 2D depthwise convolution, namely *MegEngine Cutlass*[8] [1]. It leverages specialized GEneral Matrix Multiplication (GEMM) primitives that are known to be very efficient [16]. The codebase was initially introduced in [8] and forked from *NVIDIA Cutlass*[17] [2].

The original non-oriented implementation computes the offsets (h, w) by linearly increasing k, thereby achieving contiguous memory reads. In this proposed implementation of oriented kernels, we instead compute (h, w) using Equation (5) which unfortunately introduces non-contiguous memory read issues.

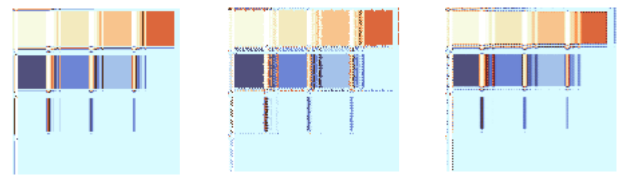
Using this proposed implementation results in uneven speeds with respect to a given angle θ: 45° angles are more than $2\times$ as slow compared to horizontal kernels (see Table 4).

9.2. CUTLASS GEMM with Rotated Input

As an alternative to rotating the filter, we can instead keep the filter fixed, and rotate the input in the opposite direction. The selling point of this approach is that we can use existing optimized depthwise convolution implementations without modification. The downside is that we introduce extra overhead by adding input and output image rotation steps. Furthermore, to preserve the information content, the rotations grow the image size HW up to $2\times$ its original value, which leads to slower convolutions. We consider additional optimization tricks for this approach such as image compression.

One of the drawbacks of the rotated input approach is that it introduces *aliasing* due to image rotation as shown in Figure 8. For small image sizes, this greatly perturbs the output and leads to significant drops in accuracy. We considered several interpolation schemes (bilinear, lanczos [22], ...) to account for aliasing but they do not help with accuracy.

[1] https://github.com/MegEngine/cutlass
[2] https://github.com/NVIDIA/cutlass



(a) Rotated filter convolution (b) Rotated input convolution: Nearest neighbor (c) Rotated input convolution: Bilinear interpolation

Figure 8: Qualitative analysis of aliasing in the rotated input approach. Input is a tiled square with tiles of varying color. Even with bilinear interpolation, border error is significant, which becomes a problem for real-world inputs.

9.3. CUDA from scratch

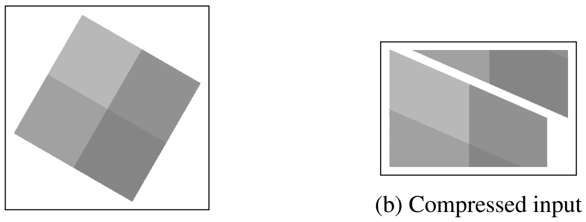
Our best performing algorithm is a custom CUDA kernel implemented from scratch. The idea of the algorithm stems from the observation that 1D depthwise convolutions incur a low computation-to-bandwidth ratio: in other words, optimizing data access pattern is critical in achieving good performances. Consequently, we design a CUDA kernel from scratch that optimizes these data patterns. In the end, we obtain an implementation that is even faster compared to non-oriented horizontal PyTorch[18]/CuDNN[4] convolutions.

We accomplish this by loading the whole input in shared GPU memory before starting our computations. This allows us to avoid the costly non-coalesced global memory accesses caused by the oriented nature of our computations. However, for larger image sizes, the input does not fit into shared GPU memory. As a result, we choose to cut the image into vertical bands and compute oriented 1D convolutions on each band. We also attempt to maximize data sharing by splitting the data loads between GPU threads so that every pixel of the input is read only once (if we ignore bordering bands). We further optimize data access by adopting vectorized loads, which increase data throughput significantly. Finally, we find the set of parameters that maximize performance on a given hardware, and achieve better performance compared to PyTorch, even though we support arbitrary oriented 1D kernels.

9.4. Rotated Input Compression

From a performance perspective, the *input rotation* approach suffers from two inefficiencies: 1) it introduces additional pre- and post-convolution steps, which can increase runtime by up to 40% compared to a non-oriented horizontal convolution (see Table 4), and 2) the image rotation can grow the intermediate image size to $2\times$ the original size, as shown in Figure 9a. This means that the convolution can be $2\times$ as costly.

To mitigate issue 2), we can compress the rotated image in a way that does not affect the convolution operation as shown in Figure 9b. Intuitively, we compress together the tips of the square in order to reduce H, and we align the sides of the square with the left border to reduce W. In theory, we can expect to reduce the image size to its original value, because area is preserved through rotation. We add a padding between the two blocks to preserve the result of the convolution. This increases the compressed size $H'W'$ by KW'.



(a) Rotated input (b) Compressed input

Figure 9: Visualization of a tiled square rotated by our rotated input kernel. After rotation, the input is fed to a standard non-oriented horizontal 1D kernel and the output is rotated back to obtain the final result. On the right, we show how compression intuitively splits the input before passing it to the horizontal kernel - the speedup comes from reducing the input size and avoiding unnecessary computations.

In practice, this compression improves overall speed. However, it requires careful implementation as it can slow down the rotation steps quite significantly.

9.5. Benchmarks

Table 4 compares training speeds for our 1D oriented kernel implementations. Table 5 presents a more compact summary, and shows training speed as well as inference speed. Our *CUDA from scratch* beats consistently all other implementations regardless of angle with intelligent data access patterns, but starts to fall off when computation becomes the bottleneck. Our CUTLASS based methods either rotate the filter, which leads to non-contiguous/slow memory accesses; or either rotate the input, which enable the use of the efficient horizontal CUTLASS kernel but introduce aliasing and grow the image size HW up to $2\times$.

We complement these kernel-level benchmarks with network-level benchmarks in Table 6. It compares training and inference throughputs measured in img/s of ConvNeXt-based models versus RepLKNet and SLaK, on an input of size 224^2.

In this paper, we focused on Depthwise Separable Convolutions (DSCs) as most modern ConvNets combine depthwise and pointwise together. Note that this setting is more challenging as improving the performance of depthwise convolutions does not necessarily lead to better DSC performance. If we remove this assumption, our oriented 1D depthwise convolutions are clearly faster than 2D depthwise convolutions, in theory and practice. We show this for large kernels in Table 1 of our paper, and add benchmarks in Table 7.

K	7								31							
Angle	0	22.5	45	67.5	90	112.5	135	157.5	0	22.5	45	67.5	90	112.5	135	157.5
$H = W = 14$																
PyTorch/CuDNN Horiz. only	0.4								0.4							
CUTLASS Horiz. only [8]	0.6								0.6							
CUTLASS Rotated Filter (Ours)	0.7	0.7	0.8	0.8	0.8	0.6	0.6	0.7	0.7	0.8	0.8	0.8	0.8	0.7	0.6	0.8
CUTLASS Rotated Input (Ours)	0.9	1.5	1.5	1.5	1.4	1.5	1.0	1.4	0.9	1.6	1.6	1.6	1.4	1.6	1.0	1.4
CUDA from scratch (Ours)	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.8	0.8	0.8	0.7	0.7	0.8	0.8	0.8
$H = W = 28$																
PyTorch/CuDNN Horiz. only	1.4								3.8							
CUTLASS Horiz. only [8]	2.3								2.5							
CUTLASS Rotated Filter (Ours)	3.1	5.3	6.2	6.0	6.2	5.1	2.5	5.1	3.1	6.8	7.8	7.6	7.8	6.6	2.7	6.4
CUTLASS Rotated Input (Ours)	3.3	5.7	5.7	5.9	5.6	5.7	4.0	5.2	3.5	6.1	6.0	6.2	5.9	6.0	4.2	5.5
CUDA from scratch (Ours)	0.9	0.9	0.9	0.9	1.0	0.9	0.9	0.9	2.6	2.6	2.6	2.6	2.5	2.6	2.6	2.6
$H = W = 56$																
PyTorch/CuDNN Horiz. only	5.5								15.0							
CUTLASS Horiz. only [8]	9.4								10.2							
CUTLASS Rotated Filter (Ours)	13.4	49.6	61.7	60.4	61.8	52.9	14.8	46.1	13.6	66.1	85.3	81.5	86.0	70.3	17.7	60.3
CUTLASS Rotated Input (Ours)	13.3	22.4	22.7	23.4	21.6	22.6	15.6	20.7	14.1	24.1	24.4	24.7	23.1	24.3	16.6	22.2
CUDA from scratch (Ours)	3.2	3.3	3.2	3.2	3.2	3.2	3.2	3.2	9.8	9.8	9.8	9.9	9.9	9.9	10.0	10.0

Table 4: Full runtime comparison of the different oriented 1D convolution implementations on an 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 CuDNN/PyTorch and CUTLASS implementations. Our *CUDA from scratch* outperforms consistently all other implementations regardless of angle with intelligent data access patterns, but starts to fall off when computation becomes the bottleneck. Our CUTLASS based methods either rotate the filter, which leads to non-contiguous/slow memory accesses; or either rotate the input, which enable the use of the efficient horizontal CUTLASS kernel but introduce aliasing and grow the image size HW up to $2\times$.

Implementation	K	Angle	Inference Time	Training Time
PyTorch Horiz.	31	$0°$	5.2±0.1ms	14.9±0.3ms
CUTLASS Horiz.	31	$0°$	3.4±0.1ms	10.1±0.1ms
CUDA From Scratch	31	$0°$	4.2±0.1ms	9.9±0.1ms
CUDA From Scratch	31	$45°$	4.2±0.1ms	9.9±0.1ms
Rotated Filter	31	$0°$	5.4±0.1ms	13.6±0.1ms
Rotated Filter	31	$45°$	30.7±0.2ms	85.3±0.3ms
Rotated Input	31	$0°$	5.3±0.1ms	14.1±0.1ms
Rotated Input	31	$45°$	9.4±0.1ms	24.6±0.1ms
Rotated Compressed	31	$0°$	5.3±0.1ms	13.9±0.1ms
Rotated Compressed	31	$45°$	4.8±0.1ms	13.0±0.1ms

Table 5: Comparison of our different implementations, for $N = 64, C = 512, H = W = 56$ on an NVIDIA RTX 3090, PyTorch 1.11, CUDA 11.3, CuDNN 8.2, FP32. We see that Rotated Compressed improves performance for non-zero angles. *Inference Time* measures only forward pass, *Training Time* includes backpropagation.

Kernel	K	Inference time	Train time
PyTorch 2D	31×31	110ms	410ms
PyTorch 1D	1×31	5.2ms	14.9ms
Ours 1D	1×31	4.2ms	14.7ms
PyTorch 2D	15×15	29ms	95ms
PyTorch 1D	1×15	3.4ms	8.7ms
Ours 1D	1×15	2.0ms	7.5ms
PyTorch 2D	7×7	8.2ms	22ms
PyTorch 1D	1×7	2.4ms	5.4ms
Ours 1D	1×7	1.0ms	4.2ms
PyTorch 2D	3×3	2.2ms	5.8ms
PyTorch 1D	1×3	2.0ms	4.0ms
Ours 1D	1×3	1.0ms	3.9ms

Table 7: Comparison between PyTorch 1D, 2D and our oriented kernels on an input of batch size 64, $C = 512$ and $H = W = 56$, FP32, measured on 1 NVIDIA RTX3090. We see that 1D kernels are clearly faster than 2D kernels of same kernel size and that our implementation is able to match PyTorch efficiency. *Inference* measures forward pass, *Training* includes backpropagation.

Model	Inference Throughput	Training Throughput
ConvNeXt-B	460	140
ConvNeXt-1D-B	450	140
RepLKNet-B	340	90
ConvNeXt-1D++-B	290	105
ConvNeXt-2D++-B	290	100
SLaK-B	210	60

Table 6: Benchmark of ConvNeXt, RepLKNet and SLaK on a 32×224^2 FP32 batch. *Inference Throughput* measures the number of images per second, forward pass only, *Training Throughput* includes backpropagation.

10. Training Settings

10.1. Pre-training

In this section, we provide the full training settings used in our experiments. We adapt them directly from ConvNeXt [14]. All settings apart from stochastic depth rate are the same for all model variants, as described in [14] and are listed in Table 8.

10.2. Downstream tasks

For ADE20K and COCO experiments, we follow the same settings as ConvNeXt, and use the same toolboxes MMDetection [3] and MMSegmentation [6]. We also use non-EMA weights. For COCO experiments, we train a Cascade Mask-RCNN [11] network for 36 epochs, with a $3\times$

Setting	ConvNeXt-1D/1D++/2D++ T/S/B
weight init	trunc. normal (0.2)
optimizer	AdamW
base learning rate	4e-3
weight decay	0.05
optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.999$
batch size	4096
training epochs	300
learning rate schedule	cosine decay
warmup epochs	20
warmup schedule	linear
layer-wise lr decay [1, 5]	None
randaugment [7]	(9, 0.5)
mixup [26]	0.8
cutmix [25]	1.0
random erasing [27]	0.25
label smoothing [20]	0.1
stochastic depth [10]	0.1/0.4/0.5
layer scale [21]	1e-6
head init scale [21]	None
gradient clip	None
exp. mov. avg. (EMA) [19]	0.9999

Table 8: **ImageNet training settings.** Input is of size 224^2. Settings taken from ConvNeXt [14]. Stochastic depth rates 0.1/0.4/0.5 are dependent on model size T/S/B.

schedule, a learning rate of 2e-4, a layer-wise l.r. decay of 0.7/0.8 and a stochastic depth rate of 0.4/0.7 for *Tiny*/*Base* respectively. For ADE20K, we train a Upernet [24] network, for 160k iterations, with a learning rate of 1e-4, layer-wise l.r. decay of 0.9, stochastic depth rate of 0.4 and report validation mIoU results using multi-scale testing.

11. Limitations and Future Work

As mentioned in previous sections, some aspects of oriented 1D kernels deserve more exploration. This includes looking at oriented non-depthwise convolutions, and doing angle backpropagation. We have mae a lot of design choices which constrain 1D kernels and their expressiveness. As future work, we would like to relax these hypotheses and come up with efficient implementations for more general cases. In this paper, we have demonstrated that we can expect measurable benefits by using oriented 1D kernels, which shows how promising such an approach can be if explored further.

References

[1] Hangbo Bao, Li Dong, Songhao Piao, and Furu Wei. BEit: BERT pre-training of image transformers. In *International Conference on Learning Representations*, 2022. 9

[2] John Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986. 5

[3] Kai Chen, Jiaqi Wang, Jiangmiao Pang, Yuhang Cao, Yu Xiong, Xiaoxiao Li, Shuyang Sun, Wansen Feng, Ziwei Liu, Jiarui Xu, Zheng Zhang, Dazhi Cheng, Chenchen Zhu, Tianheng Cheng, Qijie Zhao, Buyu Li, Xin Lu, Rui Zhu, Yue Wu, Jifeng Dai, Jingdong Wang, Jianping Shi, Wanli Ouyang, Chen Change Loy, and Dahua Lin. Mmdetection: Open mmlab detection toolbox and benchmark, 2019. 8

[4] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014. 7

[5] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. ELECTRA: Pre-training text encoders as discriminators rather than generators. In *ICLR*, 2020. 9

[6] MMSegmentation contributors. Mmsegmentation: Openmmlab semantic segmentation toolbox and benchmark. 8

[7] Ekin D. Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V. Le. Randaugment: Practical automated data augmentation with a reduced search space. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR Workshops 2020, Seattle, WA, USA, June 14-19, 2020*, pages 3008–3017. Computer Vision Foundation / IEEE, 2020. 9

[8] Xiaohan Ding, Xiangyu Zhang, Jungong Han, and Guiguang Ding. Scaling up your kernels to 31x31: Revisiting large kernel design in cnns. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11963–11975, June 2022. 1, 6, 8

[9] J.-M. Geusebroek, A.W.M. Smeulders, and J. van de Weijer. Fast anisotropic gauss filtering. *IEEE Transactions on Image Processing*, 12(8):938–943, 2003. 6

[10] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q. Weinberger. Deep Networks with Stochastic Depth. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, Lecture Notes in Computer Science, pages 646–661, Cham, 2016. Springer International Publishing. 9

[11] Durga Kumar and Xiaoling Zhang. Improving more instance segmentation and better object detection in remote sensing imagery based on cascade mask r-cnn. In *2021 IEEE International Geoscience and Remote Sensing Symposium IGARSS*, pages 4672–4675, 2021. 8

[12] C.H. Lampert and O. Wirjadi. An optimal nonorthogonal separation of the anisotropic gaussian convolution filter. *IEEE Transactions on Image Processing*, 15(11):3501–3513, 2006. 6

[13] Shiwei Liu, Tianlong Chen, Xiaohan Chen, Xuxi Chen, Qiao Xiao, Boqian Wu, Mykola Pechenizkiy, Decebal Mocanu, and Zhangyang Wang. More convnets in the 2020s: Scaling up kernels beyond 51x51 using sparsity. *arXiv preprint arXiv:2207.03620*, 2022. 1

[14] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11966–11976, 2022. 8, 9

[15] Wenjie Luo, Yujia Li, Raquel Urtasun, and Richard Zemel. Understanding the effective receptive field in deep convolutional neural networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. 1

[16] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA Tensor Core Programmability, Performance & Precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531, May 2018. arXiv:1803.04014 [cs]. 6

[17] NVIDIA. CUTLASS: CUDA templates for linear algebra, 2022. 6

[18] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019. 7

[19] B. T. Polyak and A. B. Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, 1992. 9

[20] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, June 2016. ISSN: 1063-6919. 9

[21] Hugo Touvron, Matthieu Cord, Alexandre Sablayrolles, Gabriel Synnaeve, and Hervé Jégou. Going deeper with image transformers. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 32–42, 2021. 9

[22] Ken Turkowski. Filters for common resampling tasks. In *Graphics gems*, pages 147–165. Academic Press Professional, Inc., USA, 1990. 6

[23] Shuang Wu, Guanrui Wang, Pei Tang, Feng Chen, and Luping Shi. Convolution with even-sized kernels and symmetric padding. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. 4

[24] Tete Xiao, Yingcheng Liu, Bolei Zhou, Yuning Jiang, and Jian Sun. Unified perceptual parsing for scene understanding. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision – ECCV 2018*, pages 432–448, Cham, 2018. Springer International Publishing. 9

[25] 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. 9

[26] Hongyi Zhang, Moustapha Cissé, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. 9

[27] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. Random erasing data augmentation. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 13001–13008. AAAI Press, 2020. 9